# FlashX: Massive Data Analysis Using Fast I/O

by

Da Zheng

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

July, 2016

# Abstract

With the explosion of data and the increasing complexity of data analysis, large-scale data analysis imposes significant challenges in systems design. While current research focuses on scaling out to large clusters, these scale-out solutions introduce a significant amount of overhead. This thesis is motivated by the advance of new I/O technologies such as flash memory. Instead of scaling out, we explore efficient system designs in a single commodity machine with non-uniform memory architecture (NUMA) and scale to large datasets by utilizing commodity solid-state drives (SSDs). This thesis explores the impact of the new I/O technologies on large-scale data analysis. Instead of implementing individual data analysis algorithms for SSDs, we develop a data analysis ecosystem called FlashX to target a large range of data analysis tasks. FlashX includes three subsystems: SAFS, FlashGraph and FlashMatrix. SAFS is a user-space filesystem optimized for a large SSD array to deliver maximal I/O throughput from SSDs. FlashGraph is a general-purpose graph analysis framework that processes graphs in a semi-external memory fashion, i.e., keeping vertex state in memory and edges on SSDs, and scales to graphs with billions of

ABSTRACT

vertices by utilizing SSDs through SAFS. FlashMatrix is a matrix-oriented programming framework that supports both sparse matrices and dense matrices for general data analysis. Similar to FlashGraph, it scales matrix operations beyond memory capacity by utilizing SSDs. We demonstrate that with the current I/O technologies FlashGraph and FlashMatrix in the (semi-)external-memory meets or even exceeds state-of-the-art in-memory data analysis frameworks while scaling to massive datasets for a large variety of data analysis tasks.

Primary Reader: Randal Burns

Secondary Reader: Misha Kazhdan

Tertiary Reader: Joshua Vogelstein

# Acknowledgments

First, I would like to thank my advisor Dr. Randal Burns for his advice and support throughout my PhD research. The thesis would not be possible without him. He was really accommodating and understanding and provided me tremendous help for my work. I am also very grateful for the freedom, encouragement and research direction he gave me. I would also like to thank my other advisor Dr. Alex Szalay for his initial advice on my research in the first year and support and guidance throughout the rest of my PhD research.

I would like to thank Dr. Carey Priebe and Dr. Joshua Vogelstein for their advice and help during the last three years. Their support and guidance really helped me develop the data analysis frameworks in my thesis. I would also like to thank them for giving me so many personal lessons on statistics and machine learning.

I also want to thank my thesis committee members, Dr. Misha Kazhdan and Dr. Joshua Vogelstein, for their advice and comments on my work and thesis.

I also like to thank my other collaborators, Disa Mhembere, Dr. Heng Wang, Dr. Vince Lyzinski and Dr. Youngser Park, for providing great assistance in my research

and helping me achieve better goals.

I thank many of my other labmates, Alex Baden, Stephen Hamilton, Dr. Kalin Kanov, Kunal Lillaney, Dr. Leo Duan, Will Gray Roncal, Greg Kiar, Jesse Patsolic, Tyler Tomita, Cencheng Shen, Runze Tang and Shangsi Wang. Even though many of them are technically not my labmates, I feel lucky to have them working in the collaborative research group. Many of the discussions with them were extremely helpful and I learned many things from them.

This work has been supported by National Science Foundation under grants AST-0939767, OCI-1040114, CCF-0937810, ACI-1261715, by National Institutes of Health under grant NIBIB 1RO1EB016411-01, by DARPA GRAPHS N66001-14-1-4028, XDATA FA8750-12-2-0303 and SIMPLEX N66001-15-C-4041.

# Dedication

to future...

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In today's big data era, we face challenges in both the explosion of data and the increase in complexity of data analysis. Experiments, simulations and observations generate terabytes or even petabytes in many scientific and business areas. After collecting a massive amount of data, we often need to perform complex data analysis and machine learning techniques to gain insight from the data. These data analysis tasks exhibit various data access and computation patterns. Furthermore, the fields of data analysis and machine learning evolve rapidly. The community continuously develops many new algorithms to effectively extract value from massive datasets.

Graph analysis is a class of commonly used data analysis in both academia and industry. This models real-world problems in the form of graphs (vertices and edges) to study objects and the connection between the objects. Graph analysis becomes ubiquitous and has applications in many fields such as social networks, semantic

search, knowledge discovery and cybersecurity. A well-known example is that Google applies PageRank[3] on the Web page graph to determine the importance of Web pages so that Google can provide users more relevant search results.

When applying graph analysis to real-world problems, we often encounter massive and irregular graphs. For example, the Facebook social network graph has billions of vertices, hundreds of billions of edges; the neural network of a human brain has a fundamental size of $10^{11}$ vertices and $10^{15}$ edges; graphs that evolve over time can grow even larger. These graphs often have nearly random vertex connection and power law distribution, which cause random memory access and load imbalance. As such, leading systems process graph analysis in RAM and scale to large graphs in a cluster of machines. While good partitions may be important for performance,[4] these leading systems partition graphs randomly.[5] Network performance emerges as the bottleneck and large-scale graph analysis requires fast networks to realize good performance.

Besides graph analysis, linear algebra is another fundamental tool for many scientific and machine learning applications. In these applications, we store data in (sparse or dense) matrices, and express algorithms with matrix operations. Linear algebra is very expressive and covers many data analysis tasks including graph analysis. Matrix formulation is simple and intuitive for domain experts. This leads to the development of many popular matrix-oriented programming frameworks, such as MATLAB and R, to help scientists encode and deploy complex algorithms. Like graph analysis, we

encounter massive matrices. In addition, we need to support many different matrix operations to perform varieties of complex algorithms on massive datasets.

Complex data analysis at a large scale poses significant challenges for conventional tools. For instance, MATLAB and R are known to scale to large datasets poorly. Even though SQL database systems can process relatively large datasets, their programming interface is not designed for programming data analysi algorithms and their internal optimizations are not designed for this type of workloads either. These challenges lead to the redesign of data analysis tools. MapReduce[6] is one of the most well-known tools developed for data analysis at the petabyte scale. Since its appearance, researchers have applied MapReduce to many different data analysis tasks. Many other general-purpose frameworks, such as Dryad,[7] Spark[8] and Naiad,[9] were developed to process large datasets more efficiently. In addition to the general data analysis frameworks, more specialized frameworks have been developed to tackle subsets of data analysis tasks. For example, PowerGraph[5] and GraphX[10] are specialized frameworks for graph analysis.

The majority of current research focuses on scaling out to a large cluster, but many of the distributed frameworks achieve scalability at the cost of large overhead introduced to the system. This is a common problem in many distributed systems such as MapReduce. Even though MapReduce can process petabytes of data in a large cluster, the framework does not handle many data analysis tasks efficiently due to its limited primitive operations. As McSherry et al.[11] point out, many graph anal-

ysis frameworks such as PowerGraph[5] and GraphX[10] suffer from the same problem. As such, these systems are less economical in terms of dollar efficiency and energy efficiency, which are obstacles of moving to exascale computation.

This thesis explores a different direction of scaling data analysis. Instead of scaling out to a large cluster, we focus on large-scale data analysis using fast I/O technologies and exploring efficient solutions for these tasks on a large parallel machine. It is orthogonal to the distributed solutions because our work serve as a building block for the distributed solutions to process data at even a larger scale.

This work is motivated by the tremendous improvement in storage I/O technology in the recent years. For example, a single solid-state drive (SSD) today typically delivers 100K IOPS and half a gigabytes per second. As such, an array of such SSDs can reach over one million random IOPS and over ten gigabytes per second. This is only one order of magnitude slower than RAM. We believe the advance of I/O technologies opens a new direction for large-scale data analysis. The tremendous performance improvement in I/O potentially makes flash memory the true extension of RAM and enables an efficient and cost-effective solution for running complex data analysis on massive datasets.

The main question we address in this thesis is: to what extent can the performance of flash-based data analysis solutions approach that of RAM-based solutions? If data analysis using flash memory can achieve performance comparable to that in RAM, it will positively affect computer architecture and revolutionize large-scale data analysis.

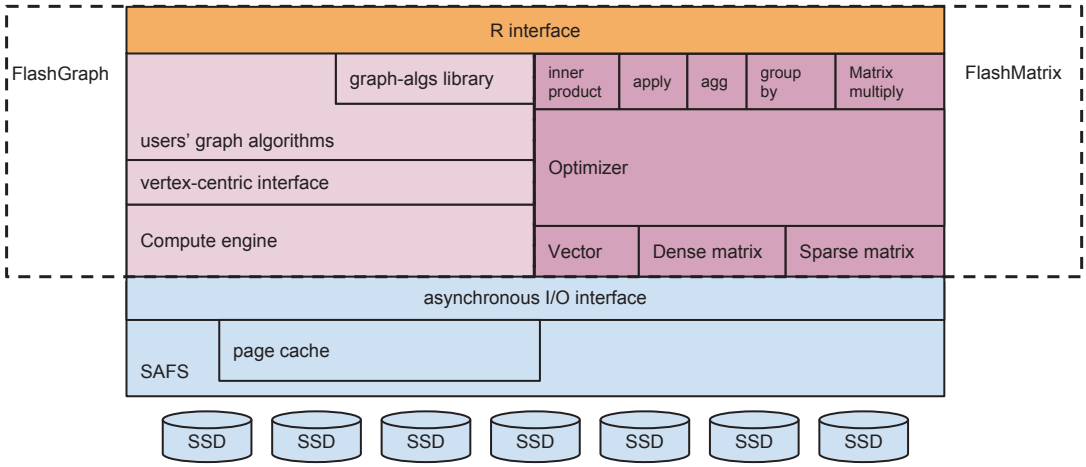



**Figure 1.1:** The architecture of FlashX.

This is the first work that explores this question with a large variety of data analysis tasks.

Hardware advances impose many new challenges in system design (both operating systems and data analysis frameworks). Operating systems were traditionally designed with an assumption of slow I/O. There exists significant lock contention in almost all layers of the block stack of the Linux kernal when it operates on fast storage. High-speed random I/O consumes significant CPU power, which requires us to minimize CPU overhead for I/O access in order to maximize the overall performance. Furthermore, although the latest I/O devices deliver unprecedented performance, they are still orders of magnitude slower than RAM, let alone CPU cache, in both throughput and latency. As such, it is crucial for data analysis frameworks to reduce I/O by reusing data in memory and constructing more compact data format to achieve performance better than what the raw hardware delivers.

We build an SSD-based data analysis ecosystem called FlashX to explore the ben-

efits that flash memory brings to large-scale data analysis. Instead of using fast but expensive non-volatile memory (NVM), we build a very fast I/O system with commodity SSDs. We build efficient programming frameworks, instead of implementing individual data analysis algorithms, to cover a variety of data analysis tasks including graph analysis and machine learning. FlashX has three main subsystems:

- SAFS[12] is a user-space filesystem optimized for large SSD arrays. It abstracts away the complexity of data access to an SSD array and delivers the maximal I/O throughput (millions of I/O per second for random I/O and tens of gigabytes per second for sequential I/O) in a large NUMA machine. In addition, SAFS also provides an efficient caching layer[13] to amplify user-perceived I/O performance.

- FlashGraph[14] is a semi-external memory graph analysis framework. For any graph algorithms, it keeps vertex state in memory and edges on SSDs. It takes advantage of SAFS and issues I/O requests carefully to bring data from SSDs to CPUs efficiently for graph analysis to achieve performance comparable to in-memory counterparts. FlashGraph is specifically optimized for graph applications that generates random I/O access to SSDs.

- FlashMatrix provides a matrix programming interface for general data analysis tasks using SSDs. It provides both basic and generalized matrix operations to express a large range of machine learning and data analysis applications. It sup-

ports both sparse matrices[15] and dense matrices.[16] The goal of FlashMatrix is to brings data from SSDs to CPUs efficiently. Unlike FlashGraph, FlashMatrix is optimized for applications with sequential I/O access to SSDs. To provide user-friendly programming interface, FlashMatrix reimplements many matrix operations in the R framework to execute R code in parallel and out of core automatically.

In this thesis, we explore graph analysis in two formulations. FlashGraph views a graph as a collection of vertices and edges, while FlashMatrix represents a graph as a sparse matrix and express graph analysis as matrix operations.[17] In the matrix formulation, a row or a column of a sparse matrix represents a vertex in a graph and a non-zero entry encodes the existence of an edge or the edge weight on a graph. Both formulations are commonly adopted and the leading graph processing frameworks use one of the formulations.[5,10,18–22] The first formulation is more flexible and can express very complex graph algorithms such as Louvain clustering[23] and METIS.[24] It is also more efficient for some commonly used graph algorithms such as breadth-first search and triangle counting. The matrix formulation, however, is more intuitive for domain experts because these users are familiar with using matrix operations to express algorithms. It also leads to more efficient implementations for some graph analysis applications such as PageRank[3] and spectral clustering.[25]

Throughout this thesis, we place data involved in computation in a semi-external memory fashion. We demonstrate that semi-external memory is an effective strat-

egy of achieving both scalability and computation efficiency for many data analysis tasks. For graph analysis, semi-external memory[26] maintains algorithmic vertex state in RAM and edge lists on storage. Only using memory for vertices increases the scalability of graph engines in proportion to the ratio of edges to vertices in a graph. In FlashMatrix, we introduce a similar construct for sparse matrix dense matrix multiplication in which one or more columns of a dense matrix are kept in memory and the sparse matrix is accessed from external memory. We can even extend this construct to some machine learning algorithms such as k-means,[27] where some of the intermediate computation state is kept in memory while input data matrices are stored on SSDs. For many data analysis tasks, semi-external memory effectively utilizes memory in a machine to reduce I/O access and achieve performance. In addition, this computation paradigm reduces the amount of data written to SSDs dramatically, which is essential to increase the lifetime of SSDs.

We also show that FlashX in semi-external memory realizes in-memory performance for many data analysis tasks while scaling to massive datasets in a single machine. These data analysis tasks exhibit varieties of I/O access patterns and different computation and I/O complexities. FlashGraph in semi-external memory achieves efficiency comparable to its in-memory version and Galois,[28] a high-performance, in-memory graph engine with a low-level API, on a variety of algorithms that generate diverse access patterns, while significantly outperforming PowerGraph, a popular distributed in-memory graph engine. We further demonstrate that FlashGraph can

process massive natural graphs in a single machine with relatively small memory footprint; e.g., we perform breadth-first search on a graph of 3.4 billion vertices and 129 billion edges using only 22 GB of memory. Similarly, the out-of-core execution of both sparse matrix and dense matrix operations in FlashMatrix achieves performance comparable to their in-memory execution; the implementations of machine learning algorithms in FlashMatrix significantly outperforms the same algorithms in Spark MLlib.[29] FlashMatrix effortlessly scales to datasets with billions of data points and its out-of-core execution uses a small fraction of resources required by in-memory implementations. Given the fast performance, we conclude that the new I/O technologies coupled with semi-external memory offer a very promising design choice for efficient and cost-effective data analysis and FlashX is a very good realization of such a design. We release the code of FlashX as open source at http://flashx.io.

## 1.1 Related work

MapReduce[6] led the trend of large-scale data analysis. It provides a very simple programming interface. In this framework, users only need to provde a *map* function and a *reduce* function. The system scales users' computation to very large datasets in a large cluster and handle machine failures automatically. The simplicity of its programming interface and the unprecedented scalability attracted tremendous academic and industrial attention, which leads to the development of Hadoop,[30] the open-source

clone of MapReduce. MapReduce works well for simple data processing tasks that can be partitioned easily and do not require much communication between partitions. Many data processing tasks at Google, such as count of URL access frequency and reverse Web links, are tasks that fit the MapReduce programming paradigm. However, if a task requires significant data exchange between partitions, MapReduce requires sorting to move data to the right location, which leads to significant overhead. As such, typical graph algorithms such as breadth-first search and triangle counting are slow in MapReduce.

Dryad[7] and Naiad[9] are distributed execution engines from Microsoft for processing large datasets. Unlike MapReduce, Dryad provides a much more flexible computation model and gives developers more fine-grain control over computation and data communication. Developers construct a directed acyclic graph to describe communication and define computation on the vertices in the graph to express algorithms. Naiad provides an even more flexible computation model, *timely dataflow*, that supports data flow graphs with loops. As such, it supports high throughput for batch processing, low latency for stream processing as well as iterative and incremental computations. Given the flexible computation model, both Dryad and Naiad implement varieties of data processing tasks more efficiently at the cost of higher programming complexity.

Spark[8] is specifically optimized for in-memory data processing in a distributed environment. Unlike MapReduce, which relies on writing data to the underlying distributed filesystem to pass data reliably between MapReduce jobs, Spark keeps

data passed between different computation stages in memory to avoid overhead of I/O access. The uniqueness of Spark is the data reliability model. Instead of replicating data to achieve reliability, Spark stores data transformation operations with RDDs (Resilient Distributed Datasets) and handles data loss due to node failure by reconstructing lost partitions on the fly.

Due to the programming complexity in the distributed execution engines, many programming frameworks have been developed on top of the distributed execution engines. Pig Latin[31] and FlumeJava[32] build on top of MapReduce to provide high-level SQL-like operations for general data analysis; DryadLINQ[33] builds on top of Dryad. The common strategy of optimizing these programming frameworks is to combine high-level operations to reduce the number of invocations of low-level primitives of the distributed execution engines. The achievable performance of these programming frameworks is bound by the underlying distributed execution engines.

Given the popularity of MapReduce, many systems have been developed to perform graph analysis and machine learning tasks on top of MapReduce. PEGASUS[21] is a popular graph processing engine built on MapReduce. PEGASUS respects the nature of the MapReduce programming paradigm and expresses graph algorithms as a generalized form of sparse matrix-vector multiplication. This form of computation works relatively well for graph algorithms such as PageRank[3] and label propagation,[34] but performs poorly for graph traversal algorithms. HEIGEN[22] implements an eigensolver on top of MapReduce for spectral analysis on billion-node graphs. De-

spite optimizations in the framework, the HEIGEN eigensolver is orders of magnitude slower than state-of-the-art distributed memory eigensolvers. Cheng-Tao Chu et. al[35] implements a set of machine learning algorithms using MapReduce to achieve speedup in a multicore machine. SystemML[36] builds a scalable declarative machine learning system on top of MapReduce, which exposes a declarative higher-level language for writing ML algorithms.

Due to the importance of graph analysis, both industry and academia build dedicated graph processing frameworks. Graph analysis results in random memory accesses, and thus state-of-the-art graph engines use distributed memory for large-scale graph analysis. Pregel[18] is a distributed graph-processing framework that allows users to express graph algorithms in vertex-centric programs using bulk-synchronous processing (BSP). It abstracts away the complexity of programming in a distributed-memory environment and runs users' code in parallel on a cluster. Giraph[37] is an open-source implementation of Pregel. GraphLab[38] and PowerGraph[5] prefer shared-memory to message passing and provide asynchronous execution. Trinity[39] optimizes message passing by restricting vertex communication to a vertex and its direct neighbors. GraphX[10] builds a graph processing framework on Spark using only a few basic dataflow operators such as join and groupby.

Despite random memory access required by graph analysis, significant efforts have been made to scale graph analysis using disks. GraphChi[19] and X-stream[20] are specifically designed for magnetic disks. They eliminate random data access from disks by

scanning the entire graph dataset in each iteration. Like graph processing frameworks built on top of MapReduce, they work relatively well for graph algorithms that require computation on all vertices, but share the same limitations, i.e., suboptimal performance in graph traversal algorithms. TurboGraph[40] is an external-memory graph engine optimized for SSDs and targets graph algorithms that can be expressed in sparse matrix vector multiplication.

Many graph algorithms can be formulated as sparse matrix multiplication or generalized sparse matrix multiplication.[17,41] In this abstraction, PageRank and label propagation are efficiently expressed as sparse-matrix, dense-vector multiplication, and breadth-first search as sparse-matrix, sparse-vector multiplication. KDT[42] is a system that realizes the abstraction and performs graph analysis using linear algebra with sparse adjacency matrices and vertex-state vectors as data representations. These frameworks target mathematicians and those with the ability to formulate and express their problems in the form of linear algebra.

Many machine learning frameworks have been developed to enable large-scale machine learning. H2O[43] and Spark MLlib[29] are machine learning libraries implemented on top of Spark. GraphLab[38] is a programming framework that provides primitives for graph-based machine learning algorithms in a distributed environment. OptiML[44] is a domain-specific language and supports vector, matrix and graph operations to implement machine learning algorithms. It is built on top of a parallel runtime system called Delite[45] to enable parallelism in a heterogeneous computation environment

in a single machine. Petuum[46] is a distributed programming framework for implementing machine learning algorithms. It leverages several fundamental properties of machine learning algorithms and aims at faster convergence of an algorithm instead of achieving higher throughput in a single iteration.

# Chapter 2

# Set-associative filesystem (SAFS)

This chapter describes the I/O layer of our data analysis ecosystem called SAFS (set-associative filesystem), a user-space filesystem that removes I/O bottlenecks in the operating system to achieve over one million IOPS from arrays of commodity SSDs. SAFS achieves the extreme I/O performance by refactoring I/O scheduling and placement for extreme parallelism and non-uniform memory and I/O. It also includes a set-associative, parallel page cache in the user space that eliminates CPU overhead and lock-contention in non-uniform memory architecture machines. We evaluate our design on a 32 core NUMA machine with four, eight-core processors. Experiments show that our design delivers 1.23 million 512-byte read IOPS. The page cache realizes the scalable IOPS of Linux asynchronous I/O (AIO) and increases user-perceived I/O performance linearly with cache hit rates. The parallel, set-associative cache matches the cache hit rates of the global Linux page cache under real workloads.

## 2.1 Introduction

Systems that perform fast, random I/O are revolutionizing commercial data services and scientific computing, creating the capability to quickly extract information from massive data sets.[47] For example, NoSQL systems underlying cloud stores generate small, incoherent I/Os that search key indexes and reference values, tables, documents, or graphs. The design of Amazon's DynamoDB testifies to this trend; it differentiates itself as a fast and scalable technology based on integrating SSDs into a key/value database.[48] In scientific computing, SSDs improve the throughput of graph and network analyses by an order of magnitude over magnetic disk.[49] Data sets that describe graphs are notoriously difficult to analyze on the steep memory hierarchies of conventional HPC hardware,[50] because they induce fine-grained, incoherent data accesses. The future of data-driven computing will rely on extending random access to large-scale storage, building on today's SSDs and other non-volatile memories as they emerge.

Specialized hardware for random access offers an effective solution, albeit costly. For example, Fusion-IO provides NAND-flash persistent memory that delivers over one million accesses per second. Fusion-IO represents a class of persistent memory devices that are used as application accelerators integrated as memory addressed directly from the processor. As another approach, the Cray XMT architecture implements a flat memory system so that all cores have fast access to all memory addresses. This approach is limited by memory size. All custom hardware approaches cost multiples

of commodity SSDs.

While recent advances in commodity SSDs have produced machines with hardware capable of over one million random IOPS, standard system configurations fail to realize the full potential of the hardware. Performance issues are ubiquitous in hardware and software, ranging from the assignment of interrupts, to non-uniform memory bandwidth, to lock contention in device drivers and the operating system. Problems arise because I/O systems were not designed for the extreme parallelism of multicore processors and SSDs. The design of file systems, page caches, device drivers and I/O schedulers does not reflect the parallelism (tens to hundreds of contexts) of the threads that initiate I/O or the multi-channel devices that service I/O requests.

None of the I/O access methods in Linux kernel perform well on a high-speed SSD array. I/O requests go through many layers in the kernel before reaching a device.[51] This produces significant CPU consumption under high IOPS. Each layer in the block subsystem uses locks to protect its data structures during concurrent updates. Furthermore, SSDs require many parallel I/Os to achieve optimal performance, while synchronous I/O, such as buffered I/O and direct I/O, issues one I/O request per thread at a time. The many threads needed to load the I/O system produce lock contention and high CPU consumption. Asynchronous I/O (AIO), which issues multiple requests in a single thread, provides a better option for accessing SSDs. However, AIO does not integrate with the operating system page cache so that SSD throughput limits user-perceived performance.

# CHAPTER 2. SAFS: TOWARD MILLIONS OF FILE SYSTEM IOPS ON LOW-COST, COMMODITY HARDWARE

The goal of our system design is twofold: (1) to eliminate bottlenecks in parallel I/O to realize the full potential of SSD arrays and (2) to integrate caching into SSD I/O to amplify the user-perceived performance to memory rates. Although the performance of SSDs has advanced in the past years, it does not approach memory both in random IOPS or latency (Table 2.1). Furthermore, RAM may be accessed at a finer granularity 64 versus 512 bytes, which can widen the performance gap by another factor of eight for workloads that perform small requests. We conclude that SSDs require a memory page cache interposed between an SSD file system and applications. This is in contrast to translating SSD storage into the memory address space using direct I/O. A major obstacle to overcome is that the page caches in operating systems do not scale to millions of IOPS. They were designed for magnetic disks that perform only about 100 IOPS per device. Performance suffers as access rates increase owing to lock contention and with increased mutli-core parallelism owing to processor overhead.

The first contribution of this paper is the design of a user-space file system that performs more than one million IOPS on commodity hardware. We implement a thin software layer that gives application programmers synchronous and asynchronous interfaces to file I/O. The system modifies I/O scheduling, interrupt handling, and data placement to reduce processor overhead, eliminate lock contention, and account for affinities between processors, memory, and storage devices.

We further present a scalable user-space cache for NUMA machines and arrays of

| | random IOPS | latency | granularity |
|---|---|---|---|
| ioDrive Octal[52] | 1,300,000 | 45$\mu$s | 512B |
| OCZ Vertex 4[53] | 120,000 | 20$\mu$s | 512B |
| DDR3-1333 | 7,300,000 | 15ns | 64B |

**Table 2.1:** The performance of specialized memory-addressable NAND flash (ioDrive Octal), a commodity SSD (OCZ Vertex 4), and memory (DDR3-1333). IOPS are measured with 512-byte random accesses.

SSDs that realizes I/O performance of Linux asynchronous I/O for cache misses and preserve the cache hit rates of the Linux page cache under real workloads. Our cache design is set-associative; it breaks the page buffer pool into a large number of small page sets and manages each set independently to reduce lock contention. The cache design extends to NUMA architectures by partitioning the cache by processors and using message passing for inter-processor communication.

The evaluation on a 32 core NUMA machine shows promising results. Our design delivers 1.23 million 512-byte read IOPS with our current hardware. The page cache realizes the scalable IOPS of Linux asynchronous I/O (AIO) and increases user-perceived I/O performance linearly with cache hit rates. We further demonstrate that our page cache achieves cache hit rates comparable to Linux page cache under real-world I/O workloads such as the ones from graph processing and cloud services. We conclude that SAFS maximizes the potential of fast SSDs in a large parallel machine and serves as an efficient I/O layer for SSD-based data analysis frameworks and cloud key-value stores.

## 2.2 Related work

This research falls into the broad area of the scalability operating systems with parallelism. Several research efforts[54,55] treat a multicore machine as a network of independent cores and implement OS functions as a distributed system of processes that communicate with message passing. We embrace this idea for processors and hybridize it with traditional SMP programming models for cores. Specifically, we use shared memory for communication inside a processor and message passing between processors.

As a counterpoint, a team from MIT[56] conducted a comprehensive survey on the kernel scalability and concluded that the traditional monolithic kernel can also have good parallel performance. We demonstrate that this is not the case for the page cache at millions of IOPS.

More specifically, our work relates to the scalable page caching. Yui et al.[57] designed a lock-free cache management for database based on Generalized CLOCK[58] and use a lock-free hashtable as index. They evaluated their design in a eight-core computer. We provide an alternative design of scalable cache and evaluate our solution at a larger scale.

The open-source community has improved the scalability of Linux page cache. Read-copy-update (RCU)[59] reduces contention through lock-free synchronization of parallel reads from the page cache (cache hits). However, the Linux kernel still relies on spin locks to protect page cache from concurrent updates (cache misses). In

contrast, our design focuses on random I/O, which implies a high churn rate of pages into and out of the cache.

Park et al.[60] evaluated the performance effects of SSDs on scientific I/O workloads and they used workloads with large I/O requests. They concluded that SSDs can only provide modest performance gains over mechanical hard drives. As the advance of SSD technology, the performance of SSDs have been improved significantly, we demonstrate that our SSD array can provide random and sequential I/O performance many times faster than mechanical hard drives to accelerate scientific applications.

The set-associative cache was originally inspired by theoretical results that shows that a cache with restricted associativity can approximate LRU.[61] We build on this result to create a set-associative cache that matches the hit rates of the Linux kernel in practice.

The high IOPS of SSDs have revealed many performance issues with traditional I/O scheduling, which has lead to the development of new fair queuing techniques that work well with SSDs.[62] We also have to modify I/O scheduling as one of many optimizations to storage performance.

## 2.3 A High IOPS File Abstraction

Although one can attach many SSDs to a machine, it is a non-trivial task to aggregate the performance of all SSDs. The default Linux configuration delivers

only a fraction of optimal performance owing to skewed interrupt distribution, device affinity in the NUMA architecture, poor I/O scheduling, and lock contention in Linux file systems and device drivers. The process of optimizing the storage system to realize the full hardware potential includes setting configuration parameters, the creation and placement of dedicated threads that perform I/O, and data placement across SSDs. Our experimental results demonstrate that our design improves system IOPS by a factor of 3.5.

### 2.3.1 Reducing Lock Contention

Parallel access to file systems exhibits high lock contention. Ext3/ext4 holds an exclusive lock on an inode, a data structure representing a file system object in the Linux kernel, for both reads and writes. For writes, XFS holds an exclusive lock on each inode that deschedules a thread if the lock is not immediately available. In both cases, high lock contention causes significant CPU overhead or, in the case of XFS, frequent context switch, and prevents the file systems from issuing sufficient parallel I/O. Lock contention is not limited to the file system, the kernel has shared and exclusive locks for each block device (SSD).

To eliminate lock contention, we create a dedicated thread for each SSD to serve I/O requests and use asynchronous I/O (AIO) to issue parallel requests to an SSD. Each file in our system consists of multiple individual files, one file per SSD, a design similar to PLFS.[63] By dedicating an I/O thread per SSD, the thread owns the file

and the per-device lock exclusively at all time. There is no lock contention in the file system and block devices. AIO allows the single thread to output multiple I/Os at the same time. The communication between application threads and I/O threads is similar to message passing. An application thread sends requests to an I/O thread by adding them to a rendezvous queue. The add operation may block the application thread if the queue is full. Thus, the I/O thread attempts to dispatch requests immediately upon arrival. Although there is locking in the rendezvous queue, the locking overhead is reduced by the two facts: each SSD maintains its own message queue, which reduces lock contention; the current implementation bundles multiple requests in a single message, which reduces the number of cache invalidations caused by locking.

### 2.3.2 Processor Affinity

Non-uniform performance to memory and the PCI bus throttles IOPS owing to the inefficiency of remote accesses. In recent multi-processor machines for both AMD and Intel architectures, each processor connects to its own memory and PCI bus. The memory and PCI bus of remote processors are directly addressable, but at increased latency and reduced throughput.

We avoid remote accesses by binding I/O threads to the processors connected to the SSDs that they access. This optimization leverages our design of using dedicated I/O threads, making it possible to localize all requests, regardless of how many threads

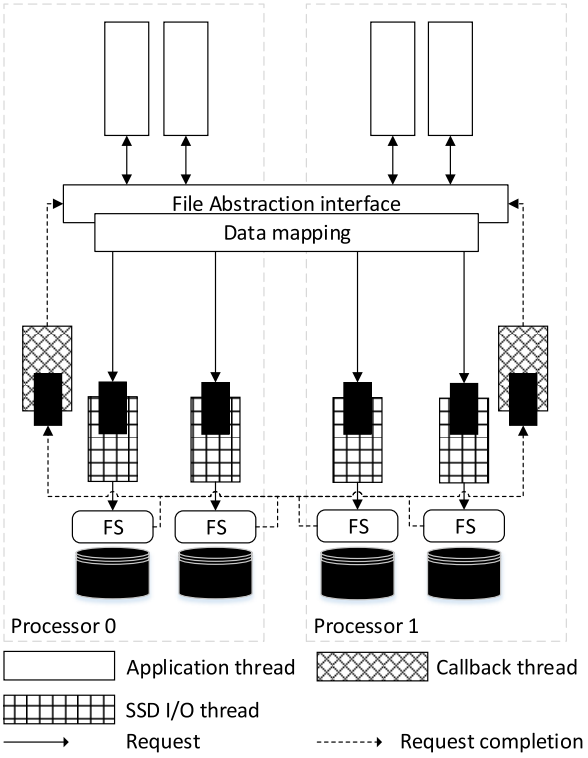



**Figure 2.1:** The architecture of SAFS.

perform I/O. By binding threads to processors, we ensure that all I/Os are sent to the local PCI bus.

### 2.3.3 Other Optimizations

**Distributing Interrupts:** With the default Linux setting, interrupts from SSDs are not evenly distributed among processor cores and we often witness that all interrupts are sent to a single core. Such large a number of interrupts saturates a CPU core which throttles system-wide IOPS.

We remove this bottleneck by distributing interrupts evenly among all physical

cores of a processor using the message signalled interrupts extension to PCI 3.0 (MSI-X).[64] MSI-X allows devices to select targets for up to 2048 interrupts. We distribute the interrupts of a storage controller host-bus adapter across multiple cores of its local processor.

**I/O scheduler:** Completely Fair Queuing (CFQ), the default I/O scheduler in the Linux kernel $>$ 2.6.18, maintains I/O requests in per-thread queues and allocates time slices for each process to access disks to achieve fairness. When many threads access many SSDs simultaneously, CFQ prevent threads from delivering sufficient parallel requests to keep SSDs busy. Performance issues with CFQ and SSDs have lead researchers to redesign I/O scheduling.[62] Future Linux releases plan to include new schedulers.

At present, there are two solutions. The most common is to use the noop I/O scheduler, which does not perform per-thread request management. This also reduces CPU overhead. Alternatively, accessing an SSD from a single thread allows CFQ to inject sufficient requests. Both solutions alleviate the bottleneck in our system.

**Data Layout:** To realize peak aggregate IOPS, we parallelize I/O among all SSDs by distributing data. We offer three data distribution functions implemented in the data mapping layer of Figure 2.1.

- Striping: Data are divided into fixed-size small blocks placed on successive disks in increasing order. This layout is most efficient for sequential I/O, but susceptible to hotspots.

- Rotated Striping: Data are divided into stripes but the start disk for each stripe is rotated, much like distributed parity in RAID5.[65] This pattern prevents strided access patterns from skewing the workload to a single SSD.
- Hash mapping: The placement of each block is randomized among all disks. This fully declusters hotspots, but requires each block to be translate by a hash function.

Workloads that do not perform sequential I/O benefit from randomization.

### 2.3.4 Implementation

We implement SAFS, a user-space filesystem that realizes our design. It exposes a simple file abstraction to user applications and supports basic operations such as file creation, deletion, open, close, read and write. It provides both synchronous and asynchronous read and write interface. Each SAFS file has metadata to keep track of the corresponding files on the underlying file system. Currently, it does not support directories.

The architecture of SAFS is shown in Figure 2.1. It builds on top of a Linux native file system on each SSD. Ext3/ext4 performs well in the system as does XFS, which we use in experiments. Each SSD has a dedicated I/O thread to process application requests. On completion of an I/O request, a notification is sent to a dedicated callback thread for processing the completed requests. The callback threads help to offload computation in the I/O threads and help applications to achieve processor

affinity. Each processor has a callback thread.

## 2.4 A Set-Associative Page Cache

The emergence of SSDs has introduced a new performance bottleneck into page caching: managing the high churn or page turnover associated with the large number of IOPS supported by these devices. Previous efforts to parallelize the Linux page cache focused on parallel read throughput from pages already in the cache. For example, read-copy-update (RCU)[59] provides low-overhead wait free reads from multiple threads. This supports high-throughput to in-memory pages, but does not help address high page turnover.

Cache management overheads associated with adding and evicting pages in the cache limit the number of IOPS that Linux can perform. The problem lies not just in lock contention, but delays from the L1-L3 cache misses during page translation and locking. We redesign the page cache to eliminate lock and memory contention among parallel threads by using set-associativity. The page cache consists of many small sets of pages (Figure 2.2). A hash function maps each logical page to a set in which it can occupy any physical page frame.

We manage each set of pages independently using a single lock and no lists. For each page set, we retain a small amount of metadata to describe the page locations. We also keep one byte of frequency information per page. We keep the metadata of a

page set in one or few cache lines to minimize CPU cache misses. If a set is not full, a new page is added to the first unoccupied position. Otherwise, a user-specified page eviction policy is invoked to evict a page. The current available eviction policies are LRU, LFU, Clock[66] and GClock.[58]

As shown in figure 2.2, each page contains a pointer to a linked list of I/O requests. When a request requires a page for which an I/O is already pending, the request will be added to the queue of the page. Once I/O on the page is complete, all requests in the queue will be served.

There are two levels of locking to protect the data structure of the cache:

- per-page lock: a spin lock to protect the state of a page.
- per-set lock: a spin lock to protect search, eviction, and replacement inside a page set.

A page also contains a reference count that prevents a page from being evicted while the page is being used by other threads.

### 2.4.1 Resizing

A page cache must support dynamic resizing to share physical memory with processes and swap. We implement dynamic resizing of the cache with linear hashing.[1] Linear hashing proceeds in rounds that double or halve the hashing address space. The actual memory usage can grow and shrink incrementally. We hold the total number of allocated pages through loading and eviction within the page sets. When

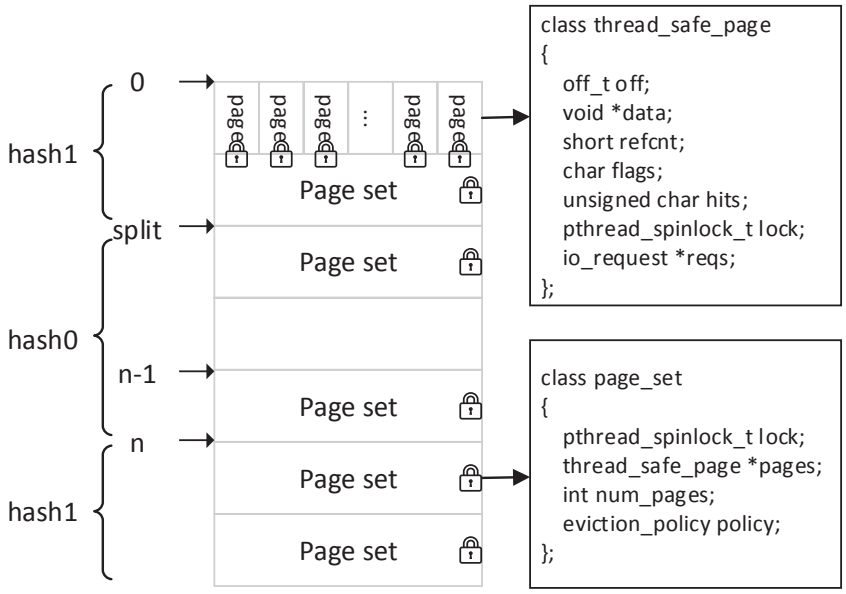



**Figure 2.2:** The organization of the set-associative cache showing the data structures and locks for pages and page sets. The hash0 and hash1 functions implement linear hashing[1] used to resize the cache. $n = init\_size \times 2^{level}$.

splitting a page set $i$, we rehash its pages to set $i$ and $init\_size \times 2^{level} + i$. The number of page sets is defined as $init\_size \times 2^{level} + split$. *level* indicates the number of times that pages have been split. *split* points to the page set to be split. The cache uses two hash functions within each level *hash0* and *hash1*:

- $hash0(v) = h(v, init\_size \times 2^{level})$
- $hash1(v) = h(v, init\_size \times 2^{level+1})$

If the result of *hash0* is smaller than *split*, *hash1* is used for the page lookup as shown in figure 2.2.

### 2.4.2 Read and write optimizations

Even though SSDs deliver high random IOPS, they still have higher throughput for larger I/O requests.[67] Furthermore, accessing a block of data on an SSD goes through a long code path in the kernel and consumes a significant number of CPU cycles.[51] By initiating larger requests, we can reduce CPU consumption and increase throughput.

Our page cache converts large read requests into a multi-buffer requests in which each buffer is single page in the page cache. Because we use the multi-buffer API of libaio, the pages need not be contiguous in memory. A large application request may be broken into multiple requests if some pages in the range read by the request are already in the cache or the request crosses a stripe boundary. The split requests are reassembled once all I/O completes and then delivered to the application as a single request.

The page cache has a dedicated thread to flush dirty pages. It selects dirty pages from the page sets where the number of dirty pages exceeds a threshold and write them with parallel asynchronous I/O to SSDs. Flushing dirty pages can reduce average write latency, which dramatically improves the performance of synchronous write issued by applications. However, the scheme may also increase the amount of data written to SSDs. To reduce the number of dirty pages to be flushed, the current policy within a page set is to select the dirty pages that are most likely to be evicted in a near future.

To reduce write I/O, we greedily flush all adjacent dirty pages using a single I/O, including pages that have not yet been scheduled for writeback. This optimization was originally proposed in disk file systems.[68] The hazard is that flushing pages early will generate more write I/O when pages are being actively written. To avoid generating more I/O, we tweak the page eviction policy, similar to CFLRU,[69] to keep dirty pages in the memory longer: when the cache evicts a page from a set, it tries to evict a clean page if possible.

### 2.4.3 NUMA design

Performance issues arise when operating a global, shared page cache on a non-uniform memory architecture. The problems stem from the increased latency of remote memory access, the reduced throughput of remote bulk memory copy.[70] A global, shared page cache treats all devices and memory uniformly. In doing so, it creates increasingly many remote operations as we scale the number of processors.

We extend the set-associative cache for the NUMA architectures (NUMA-SA) to optimize for workloads with relatively high cache hit rates and tackle hardware heterogeneity. The NUMA-SA cache design was inspired by multicore operating systems that treat each core a node in a message-passing distributed system.[55] However, we hybridize this concept with standard SMP programming models: we use message passing for inter-processor operations but use shared-memory among the cores within each processor. Figure 2.3 shows the design of NUMA-SA cache. Each processor at-

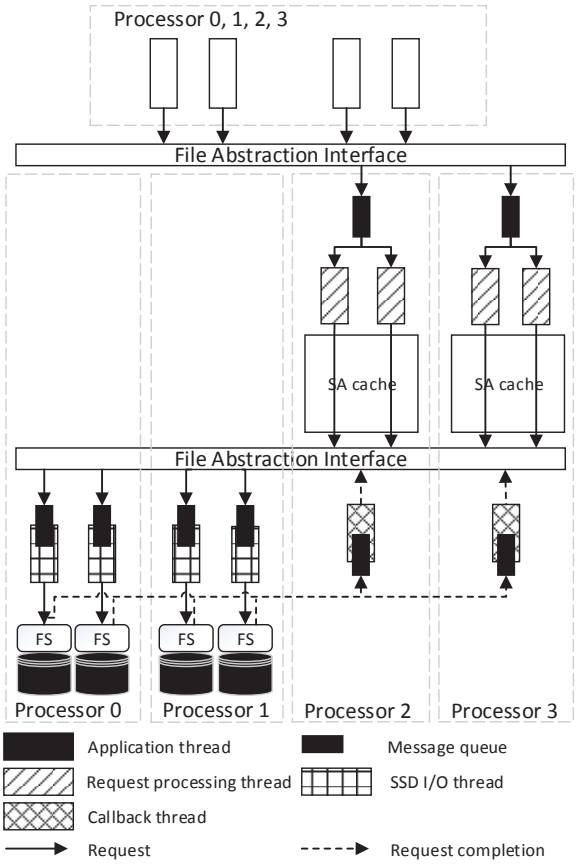



**Figure 2.3:** The architecture of the NUMA-SA cache on a four processor machine with two processors attached to SSDs. The page cache accesses SSDs with the same file abstraction interface as the one used by applications and expose the file abstraction interface to user applications.

tached to SSDs has threads dedicated to performing I/O for each SSD. The dedicated I/O thread removes contention for kernel and file locks. The processors without SSDs maintain page caches to serve applications I/O requests.

I/O requests from applications are routed to the caching nodes through message passing to reduce remote memory access. The caching nodes maintain message passing queues and a pool of threads for processing messages. On completion of an I/O request, the data is written back to the destination memory directly and then a reply

is sent to the issuing thread. This design opens opportunities to move application computation to the cache to reduce remote memory access.

We separate I/O nodes from caching nodes in order to balance computation. I/O operations require significant CPU and running a cache on an I/O node overloads the processor and reduces IOPS. This is a design decision, not a requirement, i.e. we can run a set-associative cache on the I/O nodes as well. In a NUMA machine, a large fraction of I/Os require remote memory transfers. This happens when application threads run on other nodes than I/O nodes. Separating the cache and I/O nodes does increase remote memory transfers. However, balanced CPU utilization makes up for this effect in performance. As systems scale to more processors, we expect that few processors will have PCI buses, which will increase the CPU load on these nodes, so that splitting these functions will continue to be advantageous.

Message passing creates many small requests and synchronizing these requests can become expensive. Message passing may block sending threads if their queue is full and receiving threads if their queue is empty. Synchronization of requests often involves cache line invalidation on shared data and thread rescheduling. Frequent thread rescheduling wastes CPU cycles, preventing application threads from getting enough CPU. We reduce synchronization overheads by amortizing them over larger messages.

## 2.5 Performance evaluation

We evaluate our implementation with various workloads. We start with the evaluation on SAFS without caching to demonstrate the raw performance of the hardware. We evaluate the page cache in SAFS with various workloads from commercial data services and scientific applications. We further integrate our system into the IOR benchmark[71] to measure its performance in a typical HPC environment.

We conduct experiments on a non-uniform memory architecture machine with four Intel Xeon E5-4620 processors, clocked at 2.2GHz, and 512GB memory of DDR3-1333. Each processor has eight cores with hyperthreading enabled, resulting in 16 logical cores. Only two processors in the machine have PCI buses connected to them. The machine has three LSI SAS 9217-8i host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 16 OCZ Vertex 4 SSDs are installed. In addition to the LSI HBAs, there is one RAID controller that connects to disks with root filesystem. The machine runs Ubuntu Linux 12.04 and Linux kernel v3.2.30.

To compare the best performance of our system design with that of the Linux, we measure the system in two configurations: an SMP architecture using a single processor and NUMA using all processors. On all I/O measures, Linux performs best from a single processor. Remote memory operations make using all four processors slower.

- SMP configuration: 16 SSDs connect to one processor through two LSI HBAs controlling eight SSDs each. All threads run on the same processor. Data are

| | |
|---|---|
| Linux I/O scheduler | noop |
| Page cache size | 512MB |
| Page eviction policy | GClock |
| Block size | 16 pages |
| Block mapping | striping (SMP)/hash (NUMA) |
| Page set size | 12 pages |
| AIO depth | 32 |
| Number of app threads | 16 |
| File system on SSDs | XFS |

**Table 2.2:** Default configuration of experiments.

striped across SSDs.

- NUMA configuration: 16 SSDs are connected to two processors. Processor 0 has five SSDs attached to an LSI HBA and one through the RAID controller. Processor 1 has two LSI HBAs with five SSDs each. Application threads are evenly distributed across all four processors. Data are distributed through a hash mapping that assigns 10% more I/Os to the LSI HBA attached SSDs. The RAID controller is slower.

Experiments use the configurations shown in Table 2.2 if not stated otherwise.

### 2.5.1 Optimizations in SAFS

This section enumerates the effectiveness of the hardware and software optimizations implemented in SAFS without caching, showing the contribution of each. The size of the smallest requests issued by the page cache is 4KB, so we focus on 4KB read and write performance. In each experiment, we read/write 40GB data randomly

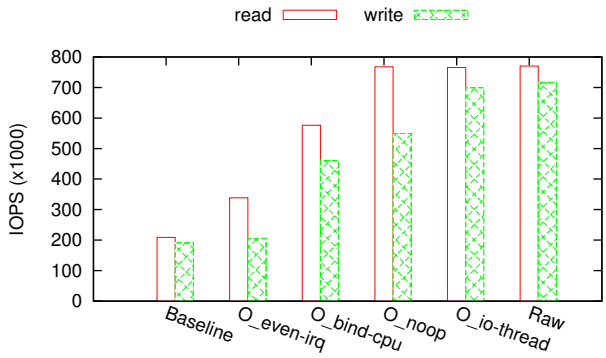



**Figure 2.4:** Optimizing page I/O in SAFS accessed from an 8 core processor (SMP). The bars show the aggregate IOPS when applying four optimizations successively in comparison with the hardware's capabilities (raw).

through SAFS in 16 threads.

We perform four optimizations on SAFS in succession to optimize performance.

- **O_even-irq:** distribute interrupts evenly among all CPU cores;
- **O_bind-cpu:** bind threads to the processor local to the SSD;
- **O_noop:** use the noop I/O scheduler;
- **O_io-thread:** create a dedicated I/O thread to access each SSD on behalf of the application threads.

Figure 2.4 shows I/O performance improvement in SAFS when applying these optimizations in succession. Performance reaches a peak 765,000 read IOPS and 699,000 write IOPS from a single processor up from 209,000 and 191,000 IOPS unoptimized. Distributing interrupts removes a CPU bottleneck for read. Binding threads to the local processor has a profound impact, doubling both read and write by eliminating remote operations. Dedicated I/O threads improves write throughput, which we

| | SAFS | ioDrive Octal 5TB |
|---|---|---|
| Read IOPS (512B) | 1,228,100 | 1,190,000 |
| Write IOPS (512B) | 386,976 | 1,180,000 |
| Read IOPS (4KB) | 946,700 | N/A |
| Write IOPS (4KB) | 766,082 | N/A |
| Read Bandwidth (64 kB) | 6.8GB/s | 6.0 GB/s |
| Write Bandwidth (64 kB) | 5.6GB/s | 4.4 GB/s |

**Table 2.3:** The performance of SAFS compared with FusionIO ioDrive Octal.

attribute to removing lock contention on the file system's inode.

When we apply all optimizations, the system realizes the performance of raw SSD hardware, as shown in Figure 2.4. It only loses less than 1% random read throughput and 2.4% random write throughput. The performance loss mainly comes from disparity among SSDs, because the system performs at the speed of the slowest SSD in the array. When writing data to SSDs, individual SSDs slow down due to garbage collection, which causes the entire SSD array to slow down. Therefore, write performance loss is higher than read performance loss. These performance losses compare well with the 10% performance loss measured by Caulfield.[72]

When we apply all optimizations in the NUMA configuration, we approach the full potential of the hardware, reaching 1.23 million read IOPS. We show performance alongside the the Fusion-IO ioDrive Octal[52] for a comparison with state of the art memory-integrated NAND flash products (Table 2.3). This reveals that our design realizes comparable read performance using commodity hardware. SSDs have a 4KB minimum block size so that 512 bytes write a partial block and, thus, slow. The 766K 4KB writes offer a better point of comparison.

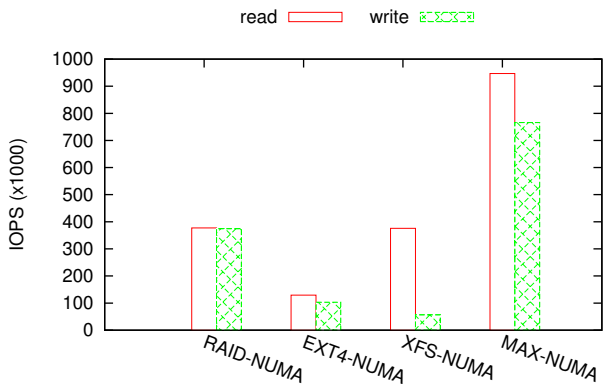



**Figure 2.5:** Performance of SAFS with Linux file systems and software RAID. All systems use optimizations O_even-irq and O_noop.

We further compare our system with Linux software options, including block interfaces (software RAID) and file systems (Figure 2.5). Although software RAID can provide comparable performance in SMP configurations, NUMA results in a performance collapse to less than half the IOPS. Locking structures in file systems prevent scalable performance on Linux software RAID. Ext4 holds a lock to protect its data structure for both reads and writes. Although XFS realizes good read performance, it performs poorly for writes due to the exclusive locks that deschedule a thread if they are not immediately available.

As an aside, we see a performance decrease in each SSD as more SSDs are accessed in a HBA, as shown in Figure 2.6. A single SSD can deliver 73,000 4KB-read IOPS and 61,000 4KB-write IOPS, while eight SSDs in a HBA deliver only 47,000 read IOPS and 44,000 write IOPS per SSD. Other work confirms this phenomena.[51] The aggregate IOPS of an SSD array increases as the number of SSDs increases. Multiple HBAs scale. Performance degradation may be caused by lock contention in the HBA

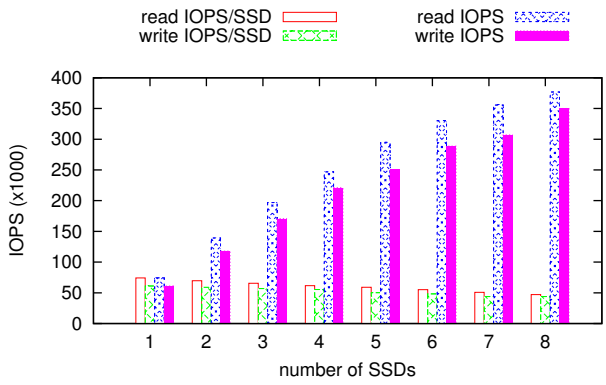



**Figure 2.6:** The 4KB read and write IOPS of individual SSDs and the aggregate IOPS of the SSD array with different numbers of SSDs in the array. All SSDs connect to a single HBA.

driver as well as by the interfere inside the hardware itself. As a design rule, we attach as few SSDs to a HBA as possible to increase the overall I/O throughput of the SSD array in the NUMA configuration.

### 2.5.2 Set-Associative Caching

We demonstrate the performance of set-associative and NUMA-SA caches under different workloads to illustrate their overhead and scalability and compare performance with the Linux page cache.

We choose workloads that exhibit high I/O rates and random access that are representatives of cloud computing and data-intensive science. We generated traces by running applications, capturing I/O system calls, and converting them into file accesses in the underlying data distribution. System call traces ensure that I/O are not filtered by a cache. Workloads include:

- Uniformly random: The workload samples 128 bytes from pages chosen randomly without replacement. The workload generates no cache hits, accessing 10,485,760 unique pages with 10,485,760 physical reads.

- Yahoo! Cloud Serving Benchmark (YCSB):[73] We derived a workload by inserting 30 million items into MemcacheDB and performing 30 million lookups according to YCSB's read-only Zipfian workload. The workload has 39,188,480 reads from 5,748,822 pages. The size of each request is 4096 bytes.

- Neo4j:[74] This workload injects a LiveJournal social network[75] in Neo4j and searches for the shortest path between two random nodes with Dijkstra algorithm. Neo4j sometimes scans multiple small objects on disks with separate reads, which biases the cache hit rate. We merge small sequential reads into a single read. With this change, the workload has 22,450,263 reads and 113 writes from 1,086,955 pages. The request size varies from 1 bytes to 1,001,616 bytes. Most requests are small. The mean request size is 57 bytes.

- Synapse labelling: This workload was traces at the Open Connectome Project `openconnecto.me` and describes the output of a parallel computer-vision pipeline run on a 4 Teravoxel image volume of mouse brain data. The pipeline detects 19 million synapses (neural connections) that it writes to spatial database. Write throughput limits performance. The workload labels 19,462,656 synapses in a 3-d array using 16 parallel threads. The workload has 19,462,656 unaligned writes of about 1000 bytes on average and updates 2,697,487 unique pages.

For experiments with multiple application threads, we dynamically dispatch small batches of I/O using a shared work queue so that all threads finish at nearly the same time, regardless of system and workload heterogeneity.

We measure the performance of Linux page cache with careful optimizations. We install Linux software RAID on the SSD array and install XFS on software RAID. We run 256 threads to issue requests in parallel to Linux page cache in order to provide sufficient I/O requests to the SSD array. We disable read ahead to avoid the kernel to read unnecessary data. Each thread opens the data file by itself because concurrent updates on a file handler in a NUMA machine leads to expensive inter-processor cache line invalidation. As shown in the previous section, XFS does not support parallel write, we only measure read performance.

**Random Workloads:**

The first experiment demonstrates that set-associative caching relieves the processor bottleneck on page replacement. We run the uniform random workload with no cache hits and measure IOPS and CPU utilization (Figure 2.7). CPU cycles bound the IOPS of the Linux cache when run from a single processor—its best configuration. Linux uses all cycles on all 8 CPU cores to achieves 641K IOPS. The set-associative cache on the same hardware runs at under 80% CPU utilization and increases IOPS by 20% to the maximal performance of the SSD hardware. Running the same workload across the entire machine increases IOPS by another 20% to almost 950K for NUMA-SA. The same hardware configuration for Linux results in an IOPS collapse.

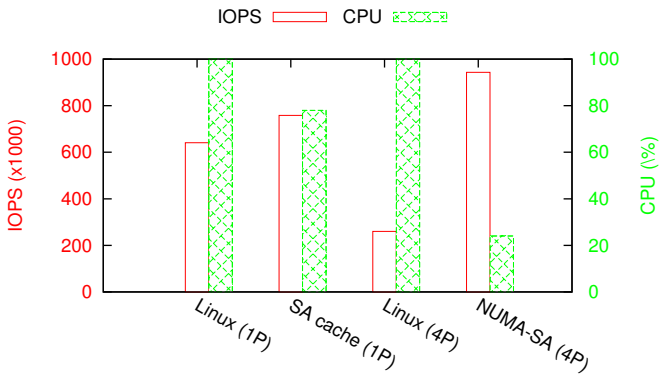



**Figure 2.7:** IOPS and CPU for random read (0% cache hit rate).

Besides the poor performance of software RAID, a NUMA machine also amplifies locking overhead on the Linux page cache. The severe lock contention in the NUMA machine is caused by higher parallelism and more expensive cache line invalidation.

A comparison of IOPS as a function of cache hit rate reveals that the set-associative caches outperform the Linux cache at high hit rates and that caching is necessary to realize application performance. We measure IOPS under the uniform random workload for the Linux cache, with set-associative caching, and without caching (SAFS). Overheads in the the Linux page cache make the set-associative cache realize roughly 30% more IOPS than Linux at all cache hit rates (Figure 2.8(a)). The overheads come from different sources at different hit rates. At 0% the main overhead comes from I/O and cache replacement. At 95% the main overhead comes from the Linux virtual file system[76] and page lookup on the cache index.

Non-uniform memory widens the performance gap (Figure 2.8). In this experiment application threads run on all processors. NUMA-SA effectively avoids lock

contention and reduces remote memory access, but Linux page cache has severe lock contention in the NUMA machine. This results in a factor of four improvement in user-perceived IOPS when compared with the Linux cache. Notably, the Linux cache does not match the performance of SAFS (with no cachcing) until a 75% cache hit rate, which reinforces the concept that lightweight I/O processing is equally important as caching to realize high IOPS.

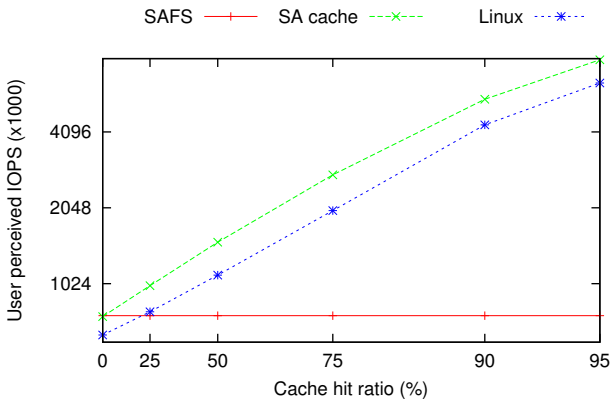
The user-perceived I/O performance increases linearly with cache hit rates. This is true for set-associative caching, NUMA-SA, and Linux. The amount of CPU and effectiveness of the CPU dictates relative performance. Linux is always CPU bound.

**The Impact of Page Set Size:**

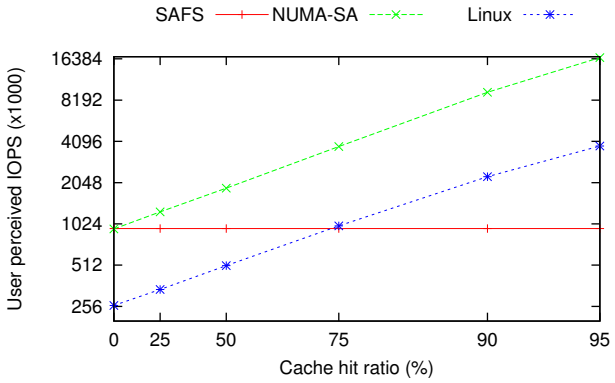
An important parameter in a set-associative cache is the size of a page set. The parameter defines a tradeoff between cache hit rate and CPU overhead within a page set. Smaller pages sets reduce cache hit rate and interference. Larger page sets better approximate global caches, but increase contention and the overhead of page lookup and eviction.

The cache hit rates provide a lower bound on the page set size. Figure 2.9 shows that the page set size has a limited impact on the cache hit rate. Although a larger page set size increases the hit rate in all workloads, it has more noticeable impact on the YCSB workload. Once the page set size increase beyond 12 pages per set, there are minimal benefits to cache hit rates.

We choose the smallest page set size that provides good cache hit rates across all




**(a)** SMP: one processor, eight applications cores.




**(b)** NUMA: four processors, 32 application cores.

**Figure 2.8:** User-perceived IOPS as a function of cache hit rate.

workloads. CPU overhead dictates small page sets. CPU increases with page set size by up to 4.3%. Cache hit rates result in better user-perceived performance by up to 3%. We choose 12 pages as the default configuration and use it for all subsequent experiments.

**Cache Hit Rates:**

We compare the cache hit rate of the set-associative cache with other page eviction policies in order to quantify how well a cache with restricted associativity emulates

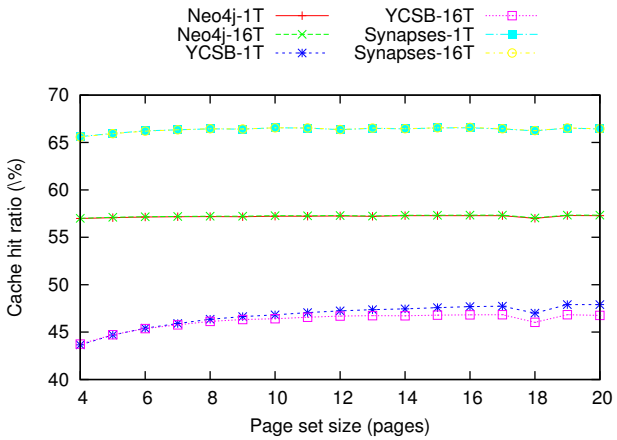



**Figure 2.9:** The impact of page set sizes on cache hit rate in the set-associative cache under real workloads both in one and 16 threads.

a global cache[61] on a variety of workloads. Figure 2.10 compares the Clock-Pro page eviction variant used by Linux.[77] We also include the cache hit rate of GClock[58] on a global page buffer. For the set-associative cache, we implement these replacement policies on each page set as well as least-frequently used (LFU). When evaluating the cache hit rate, we use the first half of a sequence of accesses to warm the cache and the second half to evaluate the hit rate.

The set-associative has a cache hit rate comparable to a global page buffer. It may lead to lower cache hit rate than a global page buffer for the same page eviction policy, as shown in the YCSB case. For workloads such as YCSB, which are dominated by frequency, LFU can generate more cache hits. It is difficult to implement LFU in a global page buffer, but it is simple in the set-associative cache due to the small size of a page set. We refer to[13] for more detailed description of LFU implementation in the set-associative cache.

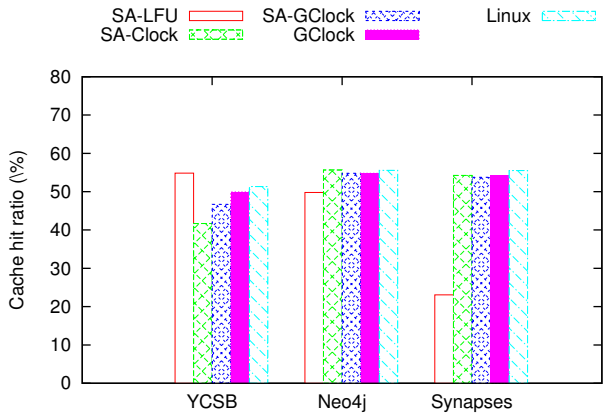



**Figure 2.10:** The cache hit rate of different cache designs under different workloads.

**Performance on Real Workloads:**

For user-perceived performance, the increased IOPS from hardware overwhelms any losses from decreased cache hit rates. Figure 2.11 shows the performance of set-associative and NUMA-SA caches in comparison to Linux's best performance under the Neo4j, YCSB, and Synapse workloads, Again, the Linux page cache performs best on a single processor.

The set-associative cache performs much better than Linux page cache under real workloads. The Linux page cache achieves around 50–60% of the maximal performance for read-only workloads (Neo4j and YCSB). Furthermore, it delivers only 8,000 IOPS for an unaligned-write workload (Synapses). The poor performance of Linux page cache results from the exclusive locking in XFS, which only allows one thread to access the page cache and issue one request at a time to the block devices.

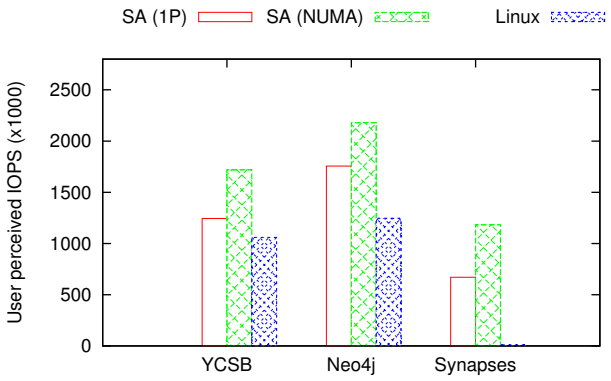



**Figure 2.11:** The performance of the set-associative cache on real workloads

## 2.6 Conclusion

We present a user-space filesystem called SAFS that achieves more than one million random read IOPS running on an array of commodity SSDs. SAFS builds on top of a local file system on each SSD in order to aggregates their IOPS. It also creates dedicated threads for I/O to each SSD. These threads access the SSD and file exclusively, which eliminates lock contention in the file and device interfaces. The design amplifies IOPS by 3.5 times and realizes nearly the full potential of the SSD hardware, less than 1% loss for reads and 2.4% for writes.

In SAFS, we deploy a set-associative parallel page cache designed for non-uniform memory architectures. The design divides the global page cache into many small, independent sets, which reduces lock contention. For NUMA architectures, the design minimizes the CPU overhead associated with remote memory copies through a hybrid SMP and message passing programming model. Each processor is treated as a node in a distributed system and inter-processor operations exchange messages through

rendezvous queues served by a dedicated thread pool. The multiple cores of each processor are programmed as an SMP. With page caching, user-perceived throughput grows linearly with the cache hit rate up to 16 million IOPS, more than four times that realized by Linux. Our optimizations in the parallel page cache achieve good performance for all request sizes and synchronous write performs nearly as well as asynchronous write.

As a whole, the design alleviates bottlenecks associated with lock contention, CPU overhead, and remote memory copies across many layers of hardware and software. The design captures parallelism and non-uniform performance of modern hardware to realize world-class performance for commodity SSDs.

# Chapter 3

# FlashGraph

This chapter describes FlashGraph, a general-purpose graph processing framework built on top of SAFS for massive graphs. By utilizing commodity SSDs through SAFS, FlashGraph enables a multicore server to process graphs with billions of vertices and hundreds of billions of edges, with minimal performance loss. FlashGraph processes graphs in a semi-external memory fashion, i.e., it stores vertex state in memory and edge lists on SSDs. It hides latency by overlapping computation with I/O. To save I/O bandwidth, FlashGraph only accesses edge lists requested by applications from SSDs; to increase I/O throughput and reduce CPU overhead for I/O, it conservatively merges I/O requests. These designs maximize performance for applications with different I/O characteristics. FlashGraph exposes a general and flexible vertex-centric programming interface that can express a wide variety of graph algorithms and their optimizations. We demonstrate that FlashGraph in semi-external memory

performs many algorithms with performance up to 80% of its in-memory implementation and significantly outperforms PowerGraph, a popular distributed in-memory graph engine.

## 3.1 Introduction

Large-scale graph analysis has emerged as a fundamental computing pattern in both academia and industry. This has resulted in specialized software ecosystems for scalable graph computing in the cloud with applications to web structure and social networking,[18,37] machine learning,[38] and network analysis.[49] The graphs are massive: Facebook's social graph has billions of vertices and today's web graphs are much larger.

The workloads from graph analysis present great challenges to system designers. Algorithms that perform edge traversals on graphs induce many small, random I/Os, because edges encode non-local structure among vertices and many real-world graphs exhibit a power-law distribution on the degree of vertices. As a result, graphs cannot be clustered or partitioned effectively[78] to localize access. While good partitions may be important for performance,[4] leading systems partition natural graphs randomly.[5]

Graph processing engines have converged on a design that *(i)* stores graph partitions in the aggregate memory of a cluster, *(ii)* encodes algorithms as parallel programs against the vertices of the graph, and *(iii)* uses either distributed shared mem-

ory[5,38] or message passing[18,37,39] to communicate between non-local vertices. Placing data in memory reduces access latency when compared to disk drives. Network performance, required for communication between graph partitions, emerges as the bottleneck and graph engines require fast networks to realize good performance.

Recent work has turned back to processing graphs from disk drives on a single machine[19,20] to achieve scalability without excessive hardware. These engines are optimized for the sequential performance of magnetic disk drives; they eliminate random I/O by scanning the entire graph dataset. This strategy can be wasteful for algorithms that access only small fractions of data during each iteration. For example, breadth-first search, a building block for many graph applications, only processes vertices in a frontier. PageRank[3] starts processing all vertices in a graph, but as the algorithm progresses, it narrows to a small subset of active vertices. There is a huge performance gap between these systems and in-memory processing.

We present FlashGraph, a semi-external memory graph-processing engine that meets or exceeds the performance of in-memory engines and allows graph problems to scale to the capacity of semi-external memory. Semi-external memory[26,49] maintains algorithmic vertex state in RAM and edge lists on storage. The semi-external memory model avoids writing data to SSDs. Only using memory for vertices increases the scalability of graph engines in proportion to the ratio of edges to vertices in a graph, more than 35 times for our largest graph of Web page crawls. FlashGraph uses an array of solid-state drives (SSDs) to achieve high throughput and low latency to

storage. Unlike magnetic disk-based engines, FlashGraph supports selective access to edge lists.

Although SSDs can deliver high IOPS, we overcome many technical challenges to construct a semi-external memory graph engine with performance comparable to an in-memory graph engine. The throughput of SSDs are an order of magnitude less than DRAM and the I/O latency is multiple orders of magnitude slower. Also, I/O performance is extremely non-uniform and needs to be localized. Finally, high-speed I/O consumes many CPU cycles, interfering with graph processing.

We build FlashGraph on top of a user-space SSD file system called SAFS[12] to overcome these technical challenges. The set-associative file system (SAFS) refactors I/O scheduling, data placement, and data caching for the extreme parallelism of modern NUMA multiprocessors. The lightweight SAFS cache enables FlashGraph to adapt to graph applications with different cache hit rates. We integrate FlashGraph with the asynchronous user-task I/O interface of SAFS to reduce the overhead of accessing data in the page cache and memory consumption, as well as overlapping computation with I/O.

FlashGraph issues I/O requests carefully to maximize the performance of graph algorithms with different I/O characteristics. It reduces I/O by only accessing edge lists requested by applications and using compact external-memory data structures. It reschedules I/O access on SSDs to increase the cache hits in the SAFS page cache. It conservatively merges I/O requests to increase I/O throughput and reduces CPU

overhead by I/O.

Our results show that FlashGraph in semi-external memory achieves performance comparable to its in-memory version and Galois,[28] a high-performance, in-memory graph engine with a low-level API, on a wide-variety of algorithms that generate diverse access patterns. FlashGraph in semi-external memory mode significantly outperforms PowerGraph, a popular distributed in-memory graph engine. We further demonstrate that FlashGraph can process massive natural graphs in a single machine with relatively small memory footprint; e.g., we perform breadth-first search on a graph of 3.4 billion vertices and 129 billion edges using only 22 GB of memory. Given the fast performance and small memory footprint, we conclude that FlashGraph offers unprecedented opportunities for users to perform massive graph analysis efficiently with commodity hardware.

## 3.2 Related Work

MapReduce[6] is a general large-scale data processing framework. PEGASUS[21] is a popular graph processing engine whose architecture is built on MapReduce. PEGASUS respects the nature of the MapReduce programming paradigm and expresses graph algorithms as a generalized form of sparse matrix-vector multiplication. This form of computation works relatively well for graph algorithms such as PageRank[3] and label propagation,[34] but performs poorly for graph traversal algorithms.

Several other works[41,42] perform graph analysis using linear algebra with sparse adjacency matrices and vertex-state vectors as data representations. In this abstraction, PageRank and label propagation are efficiently expressed as sparse-matrix, dense-vector multiplication, and breadth-first search as sparse-matrix, sparse-vector multiplication. These frameworks target mathematicians and those with the ability to formulate and express their problems in the form of linear algebra.

Pregel[18] is a distributed graph-processing framework that allows users to express graph algorithms in vertex-centric programs using bulk-synchronous processing (BSP). It abstracts away the complexity of programming in a distributed-memory environment and runs users' code in parallel on a cluster. Giraph[37] is an open-source implementation of Pregel.

Many distributed graph engines adopt the vertex-centric programming model and express different designs to improve performance. GraphLab[38] and PowerGraph[5] prefer shared-memory to message passing and provide asynchronous execution. FlashGraph supports both pulling data from SSDs and pushing data with message passing. FlashGraph does provide asynchronous execution of vertex programs to overlap computing with data access. Trinity[39] optimizes message passing by restricting vertex communication to a vertex and its direct neighbors.

Ligra[79] is a shared-memory graph processing framework and its programming interface is specifically optimized for graph traversal algorithms. It is not as general as other graph engines such as Pregel, GraphLab, PowerGraph, and FlashGraph.

Furthermore, Ligra's maximum supported graph size is limited by the memory size of a single machine.

Galois[28] is a graph programming framework with a low-level abstraction to implement graph engines. The core of the Galois framework is its novel task scheduler. The dynamic task scheduling in Galois is orthogonal to FlashGraph's I/O optimizations and could be adopted.

GraphChi[19] and X-stream[20] are specifically designed for magnetic disks. They eliminate random data access from disks by scanning the entire graph dataset in each iteration. Like graph processing frameworks built on top of MapReduce, they work relatively well for graph algorithms that require computation on all vertices, but share the same limitations, i.e., suboptimal graph traversal algorithm performance.

TurboGraph[40] is an external-memory graph engine optimized for SSDs. Like FlashGraph, it reads vertices selectively and fully overlaps I/O and computation. TurboGraph targets graph algorithms expressed in sparse matrix vector multiplication, so it is difficult to implement graph applications such as triangle counting. It uses much larger I/O requests than FlashGraph to read vertices selectively due to its external-memory data representation. Furthermore, it targets graph analysis on a single SSD or a small SSD array and does not aim at performance comparable to in-memory graph engines.

Abello et al.[26] introduced the semi-external memory algorithmic framework for graphs. Pearce et al.[49] implemented several semi-external memory graph traversal

algorithms for SSDs. FlashGraph adopts and advances several concepts introduced by these works.

## 3.3 Design

FlashGraph is a semi-external memory graph engine optimized for any fast I/O device such as Fusion I/O or arrays of solid-state drives (SSDs). It stores the edge lists of vertices on SSDs and maintains vertex state in memory. FlashGraph runs on top of the set-associative file system (SAFS),[12] a user-space filesystem designed to realize both high IOPS and lightweight caching for SSD arrays on non-uniform memory and I/O systems.

We design FlashGraph with two goals: to achieve performance comparable to in-memory graph engines while realizing the increased scalability of the semi-external memory execution model; to have a concise and flexible programming interface to express a wide variety of graph algorithms, as well as their optimizations.

To optimize performance, we design FlashGraph with the following principles:

**Reduce I/O**: Because SSDs are an order of magnitude slower than RAM, FlashGraph saturates the I/O channel in many graph applications. Reducing the amount of I/O for a given algorithm directly improves performance. FlashGraph *(i)* compacts data structures, *(ii)* maximizes cache hit rates and *(iii)* performs selective data access to edge lists.

**Perform sequential I/O when possible**: Even though SSDs provide high IOPS for random access, sequential I/O always outperforms random I/O and reduces the CPU overhead of I/O processing in the kernel.

**Maximize cache hit rates**: The high-speed I/O of SSDs is still an order of magnitude slower than RAM. I/O will be the bottleneck if SSDs serve all data for graph processing. Careful scheduling by the graph engine orders data accesses to increase data reuse in the page cache.

**Reduce random memory access:** Random access in RAM reduces the effectiveness of CPU caches and decreases memory bandwidth. It is as important to access vertices sequentially (from memory) as it is to access edges sequentially (from SSDs).

**Overlap I/O and Computation**: To fully utilize multicore processors and SSDs for data-intensive workloads, one must initiate many parallel I/Os and process data when it is ready.

**Avoid remote memory access:** Modern multi-proessor systems have non-uniform memory architectures (NUMA) in which regions of memory associate with processors. Accessing remote memory (of another processor) has higher latency, lower bandwidth, and causes overhead and contention on the remote processor.

**Minimize wearout**: SSDs wear out after many writes, especially for consumer SSDs. Therefore, it is important to minimize writes to SSDs. This includes avoiding writing data to SSDs during the application execution and reducing the necessity of loading

graph data to SSDs multiple times for the same graph.

In practice, selective data access and performing sequential I/O conflict. Selective data access prevents us from generating large sequential I/O, while using large sequential I/O may bring in unnecessary data from SSDs in many graph applications. For SSDs, FlashGraph places a higher priority in reducing the number of bytes read from SSDs than in performing sequential I/O because the random (4KB) I/O throughput of SSDs today is only two or three times less than their sequential I/O. In contrast, hard drives have random I/O throughput two orders of magnitude smaller than their sequential I/O. Therefore, other external-memory graph engines such as GraphChi and X-stream place a higher priority in performing large sequential I/O.

### 3.3.1 SAFS

SAFS[12] is a user-space filesystem for high-speed SSD arrays in a NUMA machine. It is implemented as a library and runs in the address space of its application. It is deployed on top of the Linux native filesystem.

SAFS reduces overhead in the Linux block subsystem, enabling maximal performance from an SSD array. It deploys dedicated per-SSD I/O threads to issue I/O requests with Linux AIO to reduce locking overhead in the Linux kernel; it refactors I/Os from applications and sends them to I/O threads with message passing. Furthermore, it has a scalable, lightweight page cache that organizes pages in a hashtable and places multiple pages in a hashtable slot.[13] This page cache reduces locking overhead

and incurs little overhead when the cache hit rate is low; it increases application-perceived performance linearly along with the cache hit rate.

To better support FlashGraph, we add an asynchronous user-task I/O interface to SAFS. This I/O interface supports general-purpose computation in the page cache, avoiding the pitfalls of Linux asynchronous I/O. To achieve maximal performance, SSDs require many parallel I/O requests. This could be achieved with user-initiated asynchronous I/O. However, this asynchronous I/O requires the allocation of user-space buffers in advance and the copying of data into these buffers. This creates processing overhead from copying and further pollutes memory with empty buffers waiting to be filled. When an application issues a large number of parallel I/O requests, the empty buffers account for substantial memory consumption. In the SAFS user-task programming interface, an application associates a user-defined task with each I/O request. Upon completion of a request, the associated user task executes inside the filesystem, accessing data in the page cache directly. Therefore, there is no memory allocation and copy for asynchronous I/O.

### 3.3.2 The architecture of FlashGraph

We build FlashGraph on top of SAFS to fully utilize the high I/O throughput provided by the SSD array (Figure 3.1). FlashGraph solely uses the asynchronous user-task I/O interface of SAFS to reduce the overhead of accessing data in the page cache, memory consumption, as well as overlapping computation with I/O.

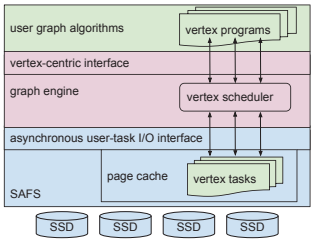



**Figure 3.1:** The architecture of FlashGraph.

FlashGraph uses the scalable, lightweight SAFS page cache to buffer the edge lists from SSDs so that FlashGraph can adapt to applications with different cache hit rates.

A graph algorithm in FlashGraph is composed of many vertex programs that run inside the graph engine. Each vertex program represents a vertex and has its own user-defined state and logic. The execution of vertex programs is subject to scheduling by FlashGraph. When vertex programs need to access data from SSDs, FlashGraph issues I/O requests to SAFS on behalf of the vertex programs and pushes part of their computation to SAFS.

### 3.3.3 Programming model

FlashGraph aims at providing a flexible programming interface to express a variety of graph algorithms and their optimizations. FlashGraph adopts the vertex-centric programming model commonly used by other graph engines such as Pregel[18] and PowerGraph.[5] In this programming model, each vertex maintains vertex state and performs user-defined tasks based on its own state. A vertex affects the state of others

by sending messages to them as well as activating them. We further allow a vertex to read the edge list of any vertex from SSDs.

The `run` method (Figure 3.2) is the entry point of a vertex program in an iteration. It is scheduled and executed exactly once on each active vertex. It is designed intentionally to have only access the vertex's own state in this method. A vertex must explicitly request its own edge list before accessing it because it is common that vertices are activated but do not perform any computation. Reading a vertex's edge list by default before executing its `run` method wastes I/O bandwidth.

The rest of FlashGraph's programming interface is event-driven to overlap computation and I/O, and receive notifications from the graph engine and other vertices. A vertex may receive three types of events:

- when it receives the edge list of a vertex, FlashGraph executes its `run_on_vertex` method.
- when it receives a message, FlashGraph executes its `run_on_message` method. This method is executed even if a vertex is inactive in the iteration.
- when the iteration comes to an end, FlashGraph executes its `run_on_iteration_end` method. A vertex needs to request this notification explicitly.

Given the programming interface, breadth-first search can be simply expressed as the code in Figure 3.3. If a vertex has not been visited, it issues a request to read its edge list in the `run` method and activates its neighbors in the `run_on_vertex` method. In this example, vertices do not receive other events.

```
class vertex {
  // entry point (runs in memory)
  void run(graph_engine &g);
  // per vertex computation (runs in the SAFS page cache)
  void run_on_vertex(graph_engine &g, page_vertex &v);
  // process a message (runs in memory)
  void run_on_message(graph_engine &g, vertex_message &msg);
  // run at the end of an iteration when all active vertices
  // in the iteration are processed.
  void run_on_iteration_end(graph_engine &g);
};
```

**Figure 3.2:** The programming interface of FlashGraph.

```
class bfs_vertex: public vertex {
  bool has_visited = false;

  void run(graph_engine &g) {
    if (!has_visited) {
      vertex_id_t id = g.get_vertex_id(*this);
      // Request the edge list of the vertex from SAFS
      request_vertices(&id, 1);
      has_visited = true;
    }
  }

  void run_on_vertex(graph_engine &g, page_vertex &v) {
    vertex_id_t dest_buf[];
    v.read_edges(dest_buf);
    g.activate_vertices(dest_buf, num_dests);
  }
};
```

**Figure 3.3:** Breadth-first search in FlashGraph.

This interface is designed for better flexibility and gives users fine-grained programmatic control. For example, a vertex has to explicitly request its own edge list so that a graph application can significantly reduce the amount of data brought to memory. Furthermore, the interface does not constrain the vertices that a vertex can communicate with or the edge lists that a vertex can request from SSDs. This flexibility allows FlashGraph to handle algorithms such as Louvain clustering,[23] in which changes to the topology of the graph occur during computation. It is difficult to express such algorithms with graph frameworks in which vertices can only interact

with direct neighbors.

### 3.3.4 Execution model

FlashGraph proceeds in iterations when executing graph algorithms written with the programming interface in Figure 3.2. In each iteration, FlashGraph processes the vertices activated in the previous iteration. An iteration ends when all active vertices complete computation; an algorithm ends when there are no active vertices in the next iteration.

As shown in Figure 3.4, FlashGraph splits a graph into multiple partitions and assigns a worker thread to each partition to process vertices. Each worker thread maintains a queue of active vertices within its own partition and executes user-defined vertex programs on them. FlashGraph's scheduler both manages the order of execution of active vertices and guarantees only a fixed number of running vertices in a thread.

There are three possible states for a vertex: *(i)* running, *(ii)* active, or *(iii)* inactive. A vertex can be activated either by other vertices or the graph engine itself. An active vertex enters the running state when the graph engine schedules it to run. During the execution, an active vertex can issue I/O requests to access the edge lists of itself and other vertices on SSDs. It remains in the running state until it finishes its task in the current iteration and becomes inactive. A running vertex interacts with other vertices via message passing and requests the notification of the end of an

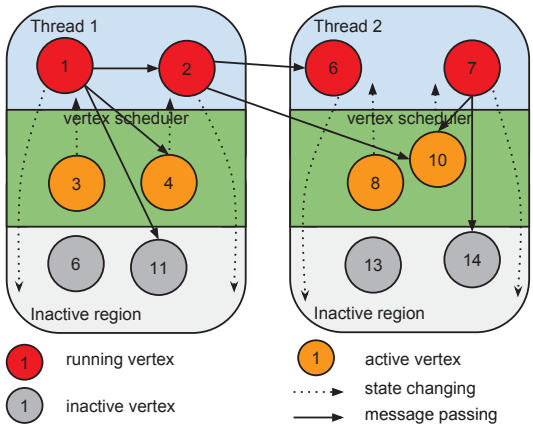



**Figure 3.4:** Execution model in FlashGraph.

iteration. When a vertex is inactive in an iteration, it still needs to respond to the messages from other vertices.

#### 3.3.4.1 Message passing

Message passing avoids concurrent data access to the state of other vertices. A semi-external memory graph engine cannot push data to other vertices by embedding data on edges like PowerGraph.[5] Writing data to other vertices directly can cause race conditions and requires atomic operations or locking for synchronization on vertex state. Message passing is a light-weight alternative for pushing data to other vertices. Although message passing requires synchronization to coordinate messages, it hides explicit synchronization from users and provides a more user-friendly programming interface. Furthermore, we can bundle multiple messages in a single packet to reduce synchronization overhead.

We implement a customized message passing scheme for vertex communication

in FlashGraph. The worker threads send and receive messages on behalf of vertices and buffer messages to improve performance. To reduce memory consumption, we process messages and pass them to vertices when the buffer accumulates a certain number of messages.

FlashGraph supports multicast to avoid unnecessary message duplication. It is common that a vertex needs to send the same message to many other vertices. In this case point-to-point communication causes unnecessary message duplication. With multicast, FlashGraph simply copies the same message once to each thread. We implement vertex activation with multicast since activation messages contain no data and are identical.

#### 3.3.4.2 Synchronous vs. asynchronous computation

A side effect of passing messages to vertices during an iteration is that FlashGraph supports asynchronous computation. When the state of a vertex is changed by a message, the new state can be immediately exposed to other vertices in the same iteration. It has been demonstrated that asynchronous computation has a faster convergence rate than synchronous computation for many graph algorithms.[38,80] However, asynchronous computation is non-deterministic and some graph algorithms do not converge when they run asynchronously.

FlashGraph provides an additional interface for programmers to enable synchronous computation. FlashGraph's approach is similar to Naiad.[9] It allows a vertex to re-

quest a notification at the end of an iteration independently. At the end of an iteration, FlashGraph invokes `run_on_iteration_end` on the vertices that requested notification during the iteration. To enable synchronous computation, each vertex maintains two copies of vertex state: *current state* and *future state*. When receiving a message, a vertex only updates the future state. At the end of an iteration, the vertex replaces its current state with its future state.

### 3.3.5 Data representation in FlashGraph

FlashGraph uses compact data representations both in memory and on SSDs. A smaller in-memory data representation allows us to process a larger graph and use a larger SAFS page cache to improve performance. A smaller data representation on SSDs allows us to pull more edge lists from SSDs in the same amount of time, resulting in better performance.

#### 3.3.5.1 In-memory data representation

FlashGraph maintains the following data structures in memory: *(i)* a graph index for accessing edge lists on SSDs; *(ii)* user-defined algorithmic vertex state of all vertices; *(iii)* vertex status used by FlashGraph; *(iv)* per-thread message queues. To save space, we choose to compute some vertex information at runtime, such as the location of an edge list on SSDs and vertex ID.

The graph index stores a small amount of information for each edge list and

compute their location and size at runtime (Figure 3.5). Storing both the location and size in memory would require a significant amount of memory: 12 bytes per vertex in an undirected graph and 24 bytes in a directed graph. Instead, for almost all vertices, we can use one byte to store the vertex degree for an undirected vertex and two bytes for a directed vertex. Knowing the vertex degree, we can compute the edge list size and further compute their locations, since edge lists on SSDs are sorted by vertex ID. To balance computation overhead and memory space, we store the locations of a small number of edge lists in memory. By default, we store one location for every 32 edge lists, which makes computation overhead almost unnoticeable while the amortized memory overhead is small. In addition, we store the degree of large vertices ($\geq 255$) in a hash table. Most real-world graphs follow the power-law distribution in vertex degree, so there are only a small number of vertices in the hash table. In our default configuration, each vertex in the index uses slightly more than 1.25 bytes in an undirected graph and slightly more than 2.5 bytes in a directed graph.

Users define algorithmic vertex state in vertex programs. The semi-external memory execution model requires the size of vertex state to be a small constant so FlashGraph can keep it in memory throughout execution. In our experience, the algorithmic vertex state is usually small. For example, breadth-first search only needs one byte for each vertex (Figure 3.3). Many graph algorithms we implement use no more than eight bytes for each vertex. Many graph algorithms need to access the vertex ID that vertex state belongs to in a vertex program. Instead of storing the vertex

ID with vertex state, we compute the vertex ID based on the address of the vertex state in memory. It is cheap to compute vertex ID most of the time. It becomes relatively more expensive to compute when FlashGraph starts to balance load because FlashGraph needs to search multiple partitions for the vertex state (Section 3.3.8.1).

#### 3.3.5.2 External-memory data representation

FlashGraph stores edges and edge attributes of vertices on SSDs. To amortize the overhead of constructing a graph for analysis in FlashGraph and reduce SSD wearout, we use a single external-memory data structure for all graph algorithms supported by FlashGraph. Since SSDs are still several times slower than RAM, the external-memory data representation in FlashGraph has to be compact to reduce the amount of data accessed from SSDs.

Figure 3.5 shows the data representation of a graph on SSDs. An edge list has a header, edges and edge attributes. Edge attributes are stored separately from edges so that graph applications avoid reading attributes when they are not required. This strategy is already successfully employed by many database systems.[81] All of the edge lists stored on SSDs are ordered by vertex ID, given by the input graph. Vertex numbering can greatly affect the performance. A good one increases data locality for adjacency list access on SSDs as well as message passing. In the future work, we will explore different vertex ordering schemes such as shingle ordering[82] and SlashBurn,[83] or even use graph clustering schemes such as spectral clustering[25] and

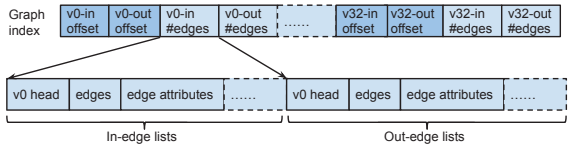



**Figure 3.5:** The data representation of a directed graph in FlashGraph. During computation, the graph index is maintained in memory and the in-edge and out-edge lists are accessed from SSDs.

Louvain clustering[23] to reorder vertices.

FlashGraph stores the in-edge and out-edge list of a vertex separately for a directed graph. Many graph applications require only one type of edge. As such, storing both in-edges and out-edges of a vertex together would cause FlashGraph to read more data from SSDs. If a graph algorithm does require both in-edges and out-edges of vertices, having separate in-edge and out-edge lists could potentially double the number of I/O requests. However, FlashGraph merges I/O requests (Section 3.3.6), which significantly alleviates this problem.

### 3.3.6 Edge list access on SSDs

Graph algorithms exhibit varying I/O access patterns in the semi-external memory computation model. The most prominent is that each vertex accesses only its own edge list. In this category, graph algorithms such as PageRank[3] access all edge lists of a graph in an iteration; graph traversal algorithms require access to many edge lists in some of their iterations on most real-world graphs. A less common category of graph algorithms, such as triangle counting, require a vertex to access the edge lists

of many other vertices as well. FlashGraph supports all of these access patterns and optimizes them differently.

Given the good random I/O performance of SSDs, FlashGraph selectively accesses the edge lists required by graph algorithms. Most graph algorithms only need to access a subset of edge lists within an iteration. External-memory graph engines such as GraphChi[19] and X-Stream[20] that sequentially access *all* edge lists in each iteration waste I/O bandwidth despite avoiding random I/O access. Selective access is superior to sequentially accessing the entire graph in each iteration and significantly reduces the amount of data read from SSDs.

FlashGraph merges I/O requests to maximize its performance. During an iteration of most algorithms, there are a large number of vertices that will likely request many edge lists from SSDs. Given this, it is likely that multiple edge lists required are stored nearby on SSDs, giving us the opportunity to merge I/O requests.

FlashGraph globally sorts and merges I/O requests issued by all *active state* vertices for applications where each vertex requests a single edge list within an iteration. FlashGraph relies on its vertex scheduler (Section 3.3.7) to order all I/O requests within the iteration. We only merge I/O requests when they access either the same page or adjacent pages on SSDs. To minimize the amount of data brought from SSDs, the minimum I/O block size issued by FlashGraph is one flash page (4KB). As a result, an I/O request issued by FlashGraph varies from as small as one page to as large as many megabytes to benefit graph algorithms with various I/O access

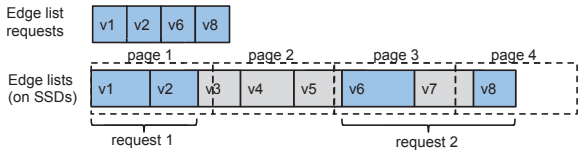



**Figure 3.6:** FlashGraph accesses edge lists and merges I/O requests.

patterns.

Figure 3.6 illustrates the process of selectively accessing edge lists on SSDs and merging I/O requests. In this example, the graph algorithm requests the in-edge lists of four vertices: `v1, v2, v6 and v8`. FlashGraph issues I/O requests to access these edge lists from SSDs. Due to our merging criteria, FlashGraph merges I/O requests for `v1` and `v2` into a single I/O request because they are on the same page, and merges `v6` and `v8` into a single request because they are on adjacent pages. As a result, FlashGraph only needs to issue two, as opposed to four, I/O requests to access four edge lists in this example.

In the less common case that a vertex requests edge lists of multiple vertices, FlashGraph must observe I/O requests issued by all *running state* vertices before sorting them. In this case, FlashGraph can no longer rely on its vertex scheduler to reorder I/O requests in an iteration. The more requests FlashGraph observes, the more likely it is to merge them and generate cache hits. FlashGraph is only able to observe a relatively small number of I/O requests, compared to the size of a graph, due to the memory constraint. It is in this less common case that FlashGraph relies on SAFS to merge I/O requests to reduce memory consumption. Finally, to further

increase I/O merging and cache hit rates, FlashGraph uses a flexible vertical graph partitioning scheme (Section 3.3.8).

### 3.3.7 Vertex scheduling

Vertex scheduling greatly affects the performance of graph algorithms. Intelligent scheduling accelerates the convergence rate and improves I/O performance. FlashGraph's default scheduler aims to decrease the number of I/O accesses and increase page cache hit rates. FlashGraph also allows users to customize the vertex scheduler to optimize for the I/O access pattern and accelerate the convergence of their algorithms. For example, scan statistics[84] in Section 3.4 requires large-degree vertices to be scheduled first to skip expensive computation on the majority of vertices.

FlashGraph deploys a per-thread vertex scheduler. Each thread schedules vertices in its own partition independently. This strategy simplifies implementation and results in framework scalability. The per-thread scheduler keeps multiple active vertices in the running state so that FlashGraph can observe then merge many I/O requests issued by vertex programs. In general, FlashGraph favors a large number of *running state* vertices because it allows FlashGraph to merge more I/O requests to improve performance. In practice, performance improvement is no longer noticeable past 4000 *running state* vertices per thread.

The default scheduler processes vertices ordered by vertex ID. This scheduling maximizes merging I/O requests for most graph algorithms because vertices request

their own edge lists in most graph algorithms and edge lists are ordered by vertex ID on SSDs. For algorithms in which vertex ordering does not affect the convergence rate, the default scheduler alternates the direction that it scans the queue of active vertices between iterations. This strategy results in pages accessed at the end of the previous iteration being accessed at the beginning of the current iteration, potentially increasing the cache hit rate.

### 3.3.8 Graph partitioning

FlashGraph partitions a graph in two dimensions at runtime (Figure 3.7), inspired by two-dimension matrix partitioning. It assigns each vertex to a partition for parallel processing, shown as *horizontal partitioning* in Figure 3.7. FlashGraph applies the horizontal partitioning in all graph applications. In addition, it provides a flexible runtime edge list partitioning scheme within a horizontal partition, shown as *vertical partitioning* in Figure 3.7. This scheme, when coupled with the vertex scheduling, can increase the page cache hit rate for applications that require a vertex to access the edge lists of many vertices because this increases the possibility that multiple threads share edge list data in the cache by accessing the same edge lists concurrently.

FlashGraph assigns a worker thread to each horizontal partition to process vertices in the partition independently. The worker threads are associated with specific hardware processors. When a thread processes vertices in its own partition, all memory accesses to the vertex state are localized to the processor. As such, our partitioning

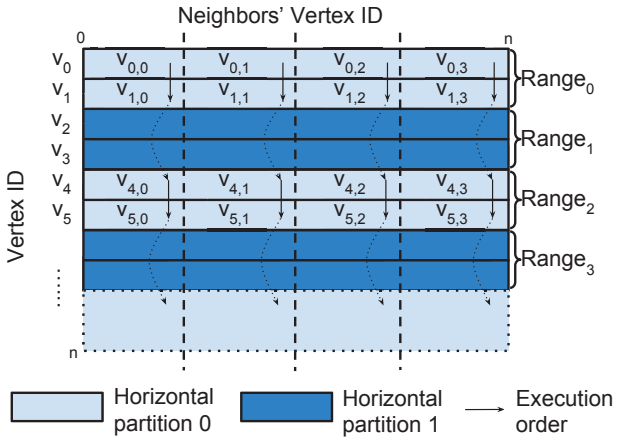



**Figure 3.7:** An example of 2D partitioning on a graph of $n$ vertices, visualized as an adjacency matrix. In this example, the graph is split into two horizontal partitions and four vertical partitions. The size of a range in a horizontal partition is two. $v_{i,j}$ represents vertical partition $j$ of vertex $i$. The arrows show the order in which the vertical partitions of vertices in horizontal partition 0 are executed in a worker thread.

scheme maximizes data locality in memory access within each processor.

FlashGraph applies a *range partitioning* function to horizontally partition a graph. The function performs a right bit shift on a vertex ID by a predefined number $r$ and takes the modulo of the shifted result:

```
range_id = vid >> r

partition_id = range_id % n
```

As such, a partition consists of multiple vertex ID ranges and the size of a range is determined by a tunable parameter $r$. $n$ denotes the number of partitions. All vertices in a partition are assigned to the same worker thread.

Range partitioning helps FlashGraph to improve spatial data locality for disk I/O in many graph applications. FlashGraph uses a per-thread vertex scheduler (Section

3.3.7) that optimizes I/O based on its local knowledge. With range partitioning, the edge lists of most vertices in the same partition are located adjacently on SSDs, which helps the per-thread vertex scheduler issue a single large I/O request to access many edge lists. The range size needs to be at least as large as the number of vertices being processed in parallel in a thread. However, a very large range may cause load imbalance because it is difficult to distribute a small number of ranges to worker threads evenly. We observe that FlashGraph works well for a graph with over 100 million vertices when $r$ is between 12 and 18.

The vertical partitioning in FlashGraph allows programmers to split large vertices into small parts at runtime. FlashGraph replicates vertex state of vertices that require vertical partitioning and each copy of the vertex state is referred to as a *vertex part*. A user has complete freedom to perform computation on and request edge lists for a *vertex part*. In an iteration, the default FlashGraph scheduler executes all active *vertex parts* in the first vertical partition and then proceeds to the second one and so on. To avoid concurrent data access to vertex state, a *vertex part* communicates with other vertices through message passing.

The vertical partitioning increases page cache hits for applications that require vertices to access the edge lists of their neighbors. In these applications, a user can partition the edge list of a large vertex and assign a *vertex part* with part of the edge list. For example, in Figure 3.7, vertex $v_0$ is split into four parts: $v_{0,0}$, $v_{0,1}$, $v_{0,2}$ and $v_{0,3}$. Each part $v_{0,j}$ is only responsible for accessing the edge lists of its neighbors with

vertex ID between $\frac{n}{4} \times j$ and $\frac{n}{4} \times (j+1)$. When the scheduler executes *vertex parts* in vertical partition $j$, only edge lists of vertices with vertex ID between $\frac{n}{4} \times j$ and $\frac{n}{4} \times (j+1)$ are accessed from SSDs, thus increasing the likelihood that an edge list being accessed is in the page cache.

#### 3.3.8.1 Load balancing

FlashGraph provides a dynamic load balancer to address the computational skew created by high degree vertices in scale-free graphs. In an iteration, each worker thread processes active vertices in its own partition. Once a thread finishes processing all active vertices in its own partition, it 'steals' active vertices from other threads and processes them. This process continues until no threads have active vertices left in the current iteration.

Vertical partitioning assists in load balancing. FlashGraph does not execute computation on a vertex simultaneously in multiple threads to avoid concurrent data access to the state of a vertex. In the applications where only a few large vertices dominate the computation of the applications, vertical partitioning breaks these large vertices into parts so that FlashGraph's load balancer can move computation of vertex parts to multiple threads, consequently leading to better load balancing.

## 3.4 Applications

We evaluate FlashGraph's performance and expressiveness with both basic and complex graph algorithms. These algorithms exhibit different I/O access patterns from the perspective of the framework, providing a comprehensive evaluation of FlashGraph.

**Breadth-first search (BFS)**: It starts with a single active vertex that activates its neighbors. In each subsequent iteration, the active and unvisited vertices activate their neighbors for the next iteration. The algorithm proceeds until there are no active vertices left. This requires only out-edge lists.

**Betweenness centrality (BC)**: We compute BC by performing BFS from a vertex, followed by a back propagation.[85] For performance evaluation, we perform this process from a single source vertex. This requires both in-edge and out-edge lists.

**PageRank (PR)**:[3] In our PR, a vertex sends the delta of its most recent PR update to its neighbors who then update thier own PR accordingly.[86] In PageRank, vertices converge at different rates. As the algorithm proceeds, fewer and fewer vertices are activated in an iteration. We set the maximal number of iterations to 30, matching the value used by Pregel.[18] This requires only out-edge lists.

**Weakly connected components (WCC)**: WCC in a directed graph is implemented with label propagation.[34] All vertices start in their own components, broadcast their component IDs to all neighbors, and adopt the smallest IDs they observe. A vertex that does not receive a smaller ID does nothing in the next iteration. This

requires both in-edge and out-edge lists.

**Triangle counting (TC)**:[87] A vertex computes the intersection of its own edge list and the edge list of each neighbor to look for triangles. We count triangles on only one vertex in a potential triangle and this vertex then notifies the other two vertices of the existence of the triangle via message passing. This requires both in-edge and out-edge lists.

**Scan statistics (SS)**:[84] The SS metric only requires finding the maximal locality statistic in the graph, which is the maximal number of edges in the neighborhood of a vertex. We use a custom FlashGraph user-defined vertex scheduler that begins computation on vertices with the largest degree first. With this scheduler, we avoid actual computation for many vertices resulting in a highly optimized implementation.[88] This requires both in-edge and out-edge lists.

These algorithms fall into three categories from the perspective of I/O access patterns. (1) BFS and betweenness centrality only perform computation on a subset of vertices in a graph within an iteration, thus they generate many random I/O accesses. (2) PageRank and (weakly) connected components need to process all vertices at the beginning, so their I/O access is generally more sequential. (3) Triangle counting and scan statistics require a vertex to read many edge lists. These two graph algorithms are more I/O intensive than the others and generate many random I/O accesses.

## 3.5 Experimental Evaluation

We evaluate FlashGraph's performance on the applications in section 3.4 on large real-world graphs. We compare the performance of FlashGraph with its in-memory implementation as well as other in-memory graph engines (PowerGraph[5] and Galois[28]). For in-memory FlashGraph, we replace SAFS with in-memory data structures for storing edge lists. We also compare semi-external memory FlashGraph with external-memory graph engines (GraphChi[19] and X-Stream[20]). We further demonstrate the scalability of FlashGraph on a web graph of 3.4 billion vertices and 129 billion edges. We also perform experiments to justify some of our design decisions that are critical to achieve performance. Throughout all experiments, we use 32 threads for all graph processing engines.

We conduct all experiments on a non-uniform memory architecture machine with four Intel Xeon E5-4620 processors, clocked at 2.2 GHz, and 512 GB memory of DDR3-1333. Each processor has eight cores. The machine has three LSI SAS 9207-8e host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 15 OCZ Vertex 4 SSDs are installed. The 15 SSDs together deliver approximately $900,000$ reads per second, or around $60,000$ reads per second per SSD. The machine runs Linux kernel v3.2.30.

We use the real-world graphs in Table 3.1 for evaluation. The largest graph is the page graph with 3.4 billion vertices and 129 billion edges. Even the smallest graph we use has 42 million vertices and 1.5 billion edges. The page graph is clustered by

| Graph datasets | # Vertices | # Edges | Size | Diameter |
|---|---|---|---|---|
| Twitter[89] | 42M | 1.5B | 13GB | 23 |
| Subdomain[90] | 89M | 2B | 18GB | 30 |
| Page[90] | 3.4B | 129B | 1.1TB | 650 |

**Table 3.1:** Graph data sets. These are directed graphs and the diameter estimation ignores the edge direction.

domain, generating good cache hit rates for some graph algorithms.

### 3.5.1 FlashGraph: in-memory vs. semi-external memory

We compare the performance of FlashGraph in semi-external memory with that of its in-memory implementation to measure the performance loss caused by accessing edge lists from SSDs.

FlashGraph scales by using semi-external memory on SSDs while preserving up to 80% performance of its in-memory implementation (Figure 3.8). In this experiment, FlashGraph uses a page cache of 1GB and has low cache hit rates in most applications. BC, WCC and PR perform the best and have only small performance degradation when running in external memory. Even in the worst cases, external-memory BFS and TC realize more than 40% performance of their in-memory counterparts on the subdomain Web graph.

Given around a million IOPS from the SSD array, we observe that most applications saturate CPU before saturating I/O. Figure 3.9 shows the CPU and I/O

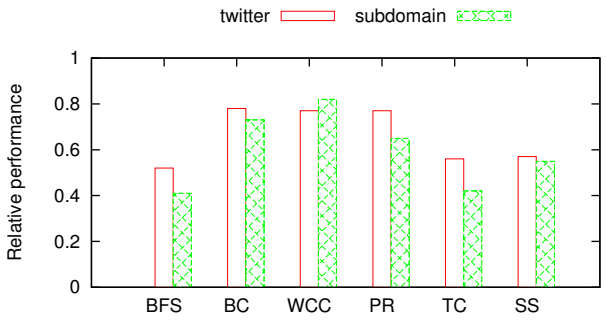
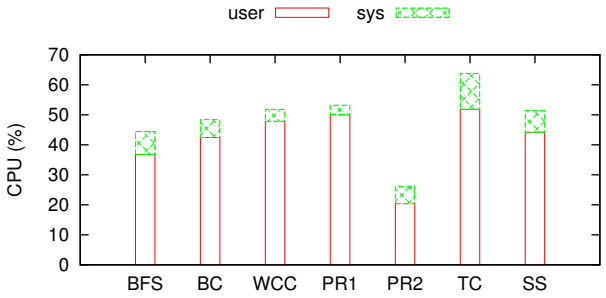
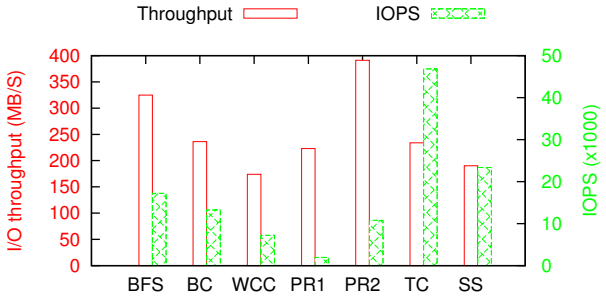



**Figure 3.8:** The performance of each application run on semi-external memory FlashGraph with 1GB cache relative to in-memory FlashGraph.

utilization of our applications in semi-external memory on the subdomain Web graph. Our machine has hyper-threading enabled, which results in 64 hardware threads in a 32-core machine, so 32 CPU cores are actually saturated when the CPU utilization gets to 50%. Both PageRank and WCC have very sequential I/O and are completely bottlenecked by the CPU at the beginning. Triangle counting saturates both CPU and I/O. It generates many small I/O requests and consumes considerable CPU time in the kernel space (almost 8 CPU cores). BFS generates very high I/O throughput in terms of bytes per second but has low CPU utilization, which suggests BFS is most likely bottlenecked by I/O. Although betweenness centrality has exactly the same I/O access pattern as BFS, it has lower I/O throughput and higher CPU utilization because it requires more computation than BFS. As a result, betweenness centrality is bottlenecked by CPU most of the time. The CPU-bound applications tend to have a small performance gap between in-memory and semi-external memory implementations.




**(a)** CPU utilization in the user and kernel space. Hyper-threading enables 64 hardware threads in a 32-core machine, so 50% CPU utilization means 32 CPU cores are saturated.




**(b)** I/O utilization.

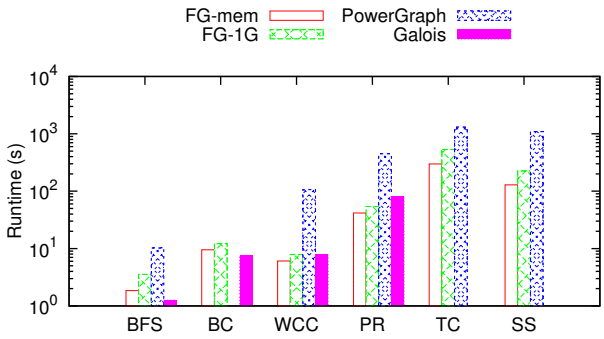
**Figure 3.9:** CPU and I/O utilization of FlashGraph on the subdomain Web graph. PR1 is the first 15 iterations of PageRank and PR2 is the last 15 iterations of PageRank.

### 3.5.2 FlashGraph vs. in-memory engines

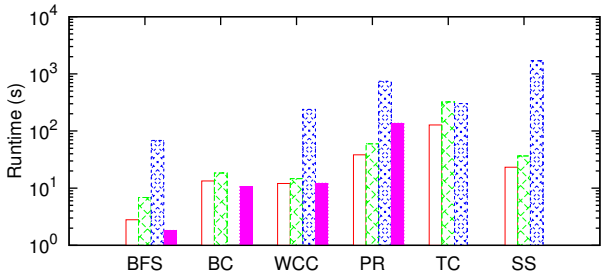
We compare the performance of FlashGraph to PowerGraph,[5] a popular distributed in-memory graph engine, and Galois,[28] a state-of-art in-memory graph engine. FlashGraph and Powergraph provide a general high-level vertex-centric programming interface, whereas Galois provides a low-level programming abstraction for building graph engines. We run these three graph engines on the Twitter and subdomain Web graphs. Unfortunately, the Web page graph is too large for in-memory graph engines. We run PowerGraph in multithread mode to achieve its best performance and use its synchronous execution engine because it performs better than the asynchronous one on both graphs.

FlashGraph has much smaller memory footprint than PowerGraph (Figure 3.11). FlashGraph only needs to maintain vertex state in memory and access data on SSDs via the page cache. Furthermore, FlashGraph's programming interface enables triangle counting and scan statistics to only need to maintain complex data structures when vertices are in the running state. In contrast, PowerGraph maintains all data in memory. It also requires much more memory to perform triangle counting and scan statistics because these two applications require every vertex to maintain much more complex data structures in PowerGraph.

Both in-memory and semi-external memory FlashGraph performs comparably to Galois, while significantly outperforming PowerGraph (Figure 3.10). In-memory FlashGraph outperforms Galois in WCC and PageRank. It performs worse than Ga-




**(a)** On the Twitter graph.




**(b)** On the subdomain Web graph.

**Figure 3.10:** The runtime of different graph engines. FG-mem is in-memory FlashGraph. FG-1G is semi-external memory FlashGraph with a page cache of 1 GB.

lois in graph traversal applications such as BFS and betweenness centrality, because Galois uses a different algorithm[91] for BFS. The algorithm reduces the number of edges traversed in both applications. The same algorithm could be implemented in FlashGraph but would not benefit semi-external memory FlashGraph because the algorithm requires access to both in-edge and out-edge lists, thus, significantly increasing the amount of data read from SSDs.

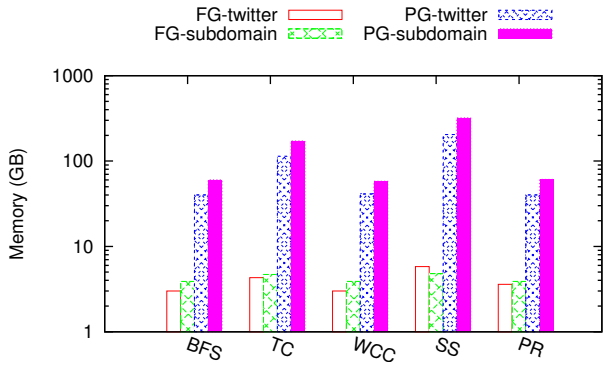


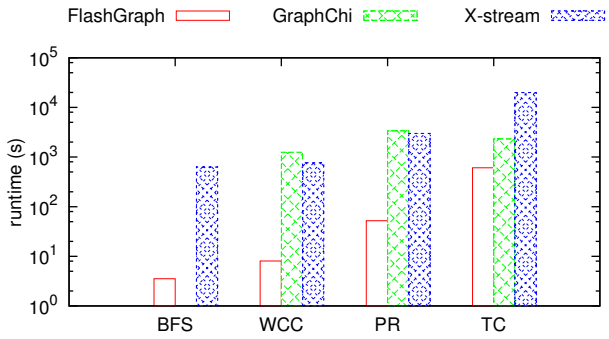

**Figure 3.11:** The memory consumption of the applications in FlashGraph (FG) with the page cache configuration of 1GB and PowerGraph (PG).

### 3.5.3 FlashGraph vs. external memory engines

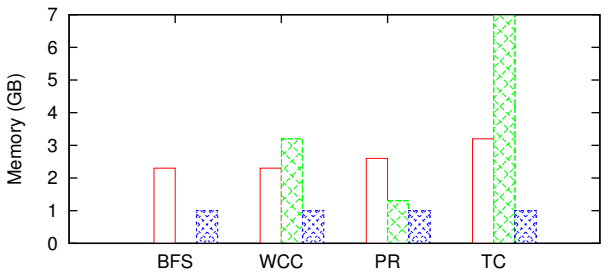
We compare the performance of FlashGraph to that of two external-memory graph engines, X-Stream[20] and GraphChi.[19] We run FlashGraph in semi-external memory and use a 1 GB page cache. We construct a software RAID on the same SSD array to run X-Stream and GraphChi. Note that GraphChi does not provide a BFS implementation, and X-Stream implements triangle counting via a semi-streaming algorithm.[92]

FlashGraph outperforms GraphChi and X-Stream by one or two orders of magnitude (Figure 3.12a). FlashGraph only needs to access the edge lists and performs computation on only the vertices required by the graph application. Even though FlashGraph generates random I/O accesses, it saves both CPU and I/O by avoiding unnecessary computation and data access. In contrast, GraphChi and X-Stream sequentially read the entire graph dataset multiple times.

Although FlashGraph uses its semi-external memory mode, it consumes a rea-




**(a)** Runtime




**(b)** Memory consumption

**Figure 3.12:** The runtime and memory consumption of semi-external memory FlashGraph and external memory graph engines on the Twitter graph.

sonable amount of memory when compared with GraphChi and X-Stream (Figure 3.12b). In some applications, FlashGraph even has smaller memory footprint than GraphChi. FlashGraph's small memory footprint allows it to run on regular desktop computers, comfortably processing billion-edge graphs.

### 3.5.4 Scaling to billion-node graphs

We further evaluate the performance of FlashGraph on the billion-scale page graph in Table 3.1. FlashGraph uses a page cache of 4GB for all applications. To the best of our knowledge, the page graph is the largest graph used for evaluating a graph

processing engine to date. The closest one is the random graph used by Pregel,[18] which has a billion vertices and 127 billion edges. Pregel processed it on 300 multicore machines. In contrast, we process the page graph on a single multicore machine.

FlashGraph can perform all of our applications within a reasonable amount of time and with relatively small memory footprint (Table 3.2). For example, FlashGraph achieves good performance in BFS on this billion-node graph. It takes less than five minutes with a cache size of 4GB; i.e., FlashGraph traverses nearly seven million vertices per second on the page graph, which is much higher than the maximal random I/O performance (900,000 IOPS) provided by the SSD array. In contrast, Pregel[18] used 300 multicore machines to run the shortest path algorithm on their largest random graph and took a little over ten minutes. More recently, Trinity[39] took over ten minutes to perform BFS on a graph of one billion vertices and 13 billion edges on 14 12-core machines.

Our solution allows us to process a graph one order of magnitude larger than the page graph on a single commodity machine with half a terabyte of RAM. The maximal graph size that can be processed by FlashGraph is limited by the capacity of RAM and SSDs. Our current hardware configuration allows us to attach 24 1TB SSDs to a machine, which can store a graph with over one trillion edges. Furthermore, the small memory footprint suggests that FlashGraph is able to process a graph with tens of billions of vertices.

FlashGraph results in a more economical solution to process a massive graph. In

| Algorithm | Runtime (sec) | Init time (sec) | Memory (GB) |
|---|---|---|---|
| BFS | 298 | 30 | 22 |
| BC | 595 | 33 | 81 |
| TC | 7818 | 31 | 55 |
| WCC | 461 | 32 | 47 |
| PR | 2041 | 33 | 46 |
| SS | 375 | 58 | 83 |

**Table 3.2:** The runtime and memory consumption of FlashGraph on the page graph using a 4GB cache size.

contrast, it is much more expensive to build a cluster or a supercomputer to process a graph of the same scale. For example, it requires 48 machines with 512GB RAM each to achieve 24TB aggregate RAM capacity, so the cost of building such a cluster is at least $24-48$ times higher than our solution. In addition, FlashGraph minimizes SSD wearout and the only write required by FlashGraph is to load a new graph to SSDs for processing. Therefore, we can further reduce the hardware cost, by using consumer SSDs instead of enterprise SSDs to store graphs, as well as reducing the maintenance cost.

### 3.5.5 The impact of optimizations

In this section, we perform experiments to justify some of our design decisions that are critical to achieve performance for FlashGraph in semi-external memory.

#### 3.5.5.1 Preservation of sequential I/O

We demonstrate the importance of taking advantage of sequential I/O access in graph applications, using BFS and weakly connected components. We start with vertex execution performed in random order, and then sequentially order vertex execution by vertex ID. Finally, we show the performance difference between merging I/O requests in SAFS vs. FlashGraph. All experiments are run on the subdomain web graph.

The huge gap (Figure 3.13) between random execution and sequential execution suggests that there exists a degree of sequential I/O in both applications, as described in Section 3.3.6. If FlashGraph did not take advantage of these sequential I/O accesses, it would suffer substantial performance degradation. Therefore, the first priority of the vertex scheduler in FlashGraph is to schedule vertex execution to generate sequential I/O. Consequently, FlashGraph's vertex scheduler is highly constrained by I/O ordering requirements and is not able to schedule vertex execution freely like Galois.[28]

Figure 3.13 also shows that I/O accesses generated by a graph algorithm are well merged in FlashGraph as opposed to the filesystem level or the block subsystem level. Although SAFS, the Linux filesystem and the Linux block subsystem are capable of merging I/O requests, they require more CPU computation to merge I/O requests and do not have a global view for merging I/O requests. Consequently, it is much more light-weight and effective to merge I/O requests in FlashGraph. By doing so,

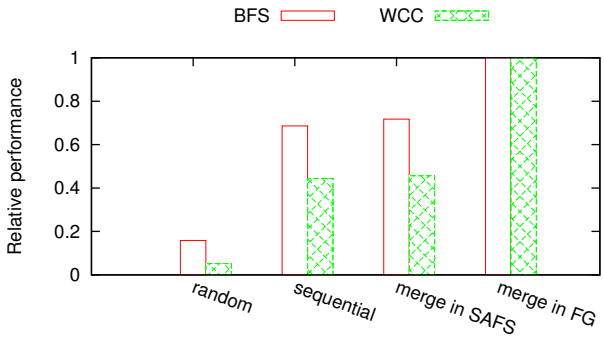



**Figure 3.13:** The impact of preserving sequential I/O access in graph applications. All performance is relative to that of the implementation of merging I/O requests in FlashGraph.

we achieve 40% speedup for BFS and more than 100% speedup for WCC.

### 3.5.5.2 The impact of the page size

In this section, we investigate the impact of the page size in SAFS. A page is the smallest I/O block that FlashGraph can access from SSDs. The experiments are run on the subdomain web graph.

Figure 3.14 shows that FlashGraph should use 4KB as the SAFS page size. SSDs store and access data at the granularity of 4KB flash pages, so using an SAFS page smaller than 4KB does not increase the I/O rate of SSDs much. A larger SAFS page size brings in more unnecessary data and wastes I/O bandwidth, which leads to performance degradation. When we increase the SAFS page size from 4KB to 1MB, the performance of BFS and triangle counting (TC) decreases to a small fraction of their maximal performance. Even WCC, whose I/O access is more sequential, performs better with 4KB pages because WCC also needs to selectively access edge lists in all

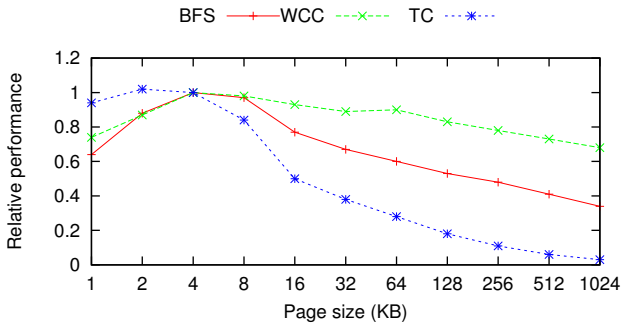



**Figure 3.14:** The impact of the page size in FlashGraph. All performance is relative to that of the implementation with 4KB page size.

iterations but the first. This result suggests that TurboGraph,[40] which uses a block size of multiple megabytes, may perform general graph analysis suboptimally. It also suggests that when using 4KB pages, selectively accessing edge lists and merging I/O enables FlashGraph to adapt to different I/O access patterns.

### 3.5.6 The impact of page cache size

We investigate the effect of the SAFS page cache size on the performance of FlashGraph. We vary the cache size from 1 GB to 32GB, which is sufficiently large to accommodate the twitter graph and the subdomain web graph. We omit Twitter graph results as they mirror subdomain graph results.

FlashGraph performs well even with a small page cache (Figure 3.15). With a 1GB page cache, all applications realize at least 65% of their performance with a 32GB page cache, and WCC and betweenness centrality even achieve around 90% of the performance with a 32GB page cache. Although PageRank has a similar I/O

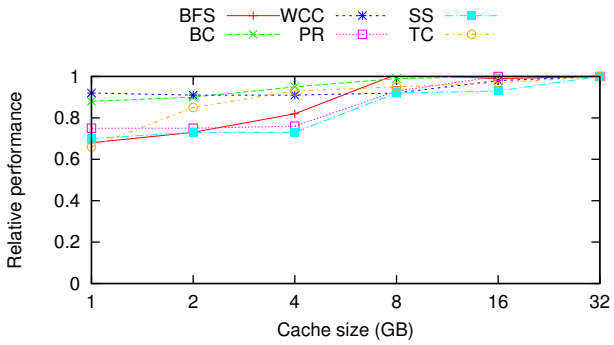



**Figure 3.15:** The impact of cache size in FlashGraph.

access pattern to WCC, it converges more slowly than WCC, so a large cache has more impact on PageRank. By varying the page cache size, we show FlashGraph can smoothly transition from a semi-external memory graph engine to an in-memory graph engine.

## 3.6 Conclusions

We present the semi-external memory graph engine called FlashGraph that closely integrates with an SSD filesystem to achieve maximal performance. It uses an asynchronous user-task I/O interface to reduce overhead associated with accessing data in the filesystem and overlap computation with I/O. FlashGraph selectively accesses edge lists required by a graph algorithm from SSDs to reduce data access; it conservatively merges I/O requests to increase I/O throughput and reduce CPU consumption; it further schedules the order of processing vertices to help merge I/O requests and maximize the page cache hit rate. All of these designs maximize performance for

applications with different I/O access patterns. We demonstrate that a semi-external memory graph engine can achieve performance comparable to in-memory graph engines.

We observe that in many graph applications a large SSD array is capable of delivering enough I/Os to saturate the CPU. This suggests the importance of optimizing for CPU and RAM in such an I/O system. It also suggests that SSDs have been sufficiently fast to be an important extension for RAM when we build a machine for large-scale graph analysis applications.

FlashGraph provides a concise and flexible programming interface to express a wide variety of graph algorithms and their optimizations. Users express graph algorithms in FlashGraph from the perspective of vertices. Vertices can interact with any other vertices in the graph by sending messages, which localizes user computation to the local memory and avoids concurrent data access to algorithmic vertex state.

Unlike other external-memory graph engines such as GraphChi and X-stream, FlashGraph supports selective access to edge lists. We demonstrate that streaming the entire graph to reduce random I/O leads to a suboptimal solution for high-speed SSDs. Reading and computing on data only required by graph applications saves computation and increases the I/O access rate to the SSDs.

We further demonstrate that FlashGraph is able to process graphs with billions of vertices and hundreds of billions of edges on a single commodity machine. FlashGraph, on a single machine, meets and surpasses the performance of distributed

graph processing engines that run on large clusters. Furthermore, the small memory footprint of FlashGraph suggests that it can handle a much larger graph in a single commodity machine. Therefore, FlashGraph results in a much more economical solution for processing massive graphs, which makes massive graph analysis more accessible to users and provides a practical alternative to large clusters for such graph analysis.

# Chapter 4

# Sparse matrix multiplication

This chapter describes another view of graph analysis. Instead of viewing a graph as a collection of vertices and edges, we encode a graph as a sparse matrix and express graph analysis as matrix operations.[17] In this formulation, a row or a column of a sparse matrix represents a vertex in a graph and a non-zero entry encodes the existence of an edge or the edge weight on a graph. As such, many graph analysis algorithms such as PageRank and spectral clustering are expressed as sparse matrix multiplication. In this chapter, we describe an efficient implementation of sparse matrix dense matrix multiplication (SpMM), a key operation for graph analysis. We scale this matrix operation by utilizing commodity SSDs and implement it in a semi-external memory (SEM) fashion, i.e., we keep the sparse matrix on SSDs and dense matrices in memory. Our SEM SpMM incorporates many in-memory optimizations that require a small memory footprint. Coupled with many I/O optimizations, our

SEM SpMM achieves performance comparable to our in-memory implementation on a large parallel machine and outperforms the implementations in Trilinos and Intel MKL. Our experiments show that the SEM SpMM achieves almost 100% performance of the in-memory implementation on graphs when the dense matrix has more than four columns; it achieves at least 65% performance of the in-memory implementation for all of our graphs when the dense matrix has only one column. We apply our SpMM to three important graph analysis applications and show that our SSD-based implementations can significantly outperform state of the art of these applications and scale to billion-node graphs.

# 4.1 Introduction

Sparse matrix multiplication is a very important computation with a wide variety of applications in scientific computing, machine learning and data mining. For example, matrix factorization algorithms on a sparse matrix such as singular value decomposition (SVD)[93] and non-negative matrix factorization (NMF)[94] requires sparse matrix multiplication. Graph analysis algorithms such as PageRank[3] can be formulated as sparse matrix multiplication or generalized sparse matrix multiplication.[17] Some of the algorithms, such as PageRank and SVD, require sparse matrix vector multiplication. Others, such as NMF, require sparse matrix dense matrix multiplication.

# CHAPTER 4. SEMI-EXTERNAL MEMORY SPARSE MATRIX MULTIPLICATION ON BILLION-NODE GRAPHS IN A MULTICORE ARCHITECTURE

The largest sparse matrices arise from graph datasets such as social networks and Web graphs, in which one performs sparse matrix multiplication for graph analysis such as community detection with NMF and spectral analysis with SVD. These matrices inherit structure from natural graphs. Specifically, these matrices are typically very sparse and have near-random distribution for non-zero entries. They also have a power law distribution that governs the number of non-zero entries per row and column.

It is challenging to have an efficient implementation of sparse matrix multiplication, especially for sparse matrices that encode real-world graphs. Sparse matrix multiplication has very low computation density and its performance is limited by memory access. As such, this operation usually achieves only a small fraction of the peak performance of a modern processor.[95] It becomes even more challenging to perform this operation on graphs due to random memory access caused by near-random vertex connection and load imbalancing caused by the power-law distribution in vertex degree. Furthermore, many real-world graphs are enormous. For example, Facebook's social network has billions of vertices and today's Web graphs are even much larger. Therefore, sparse matrix multiplication on graphs is frequently the bottleneck in an application.

Current research focuses on sparse matrix vector multiplication (SpMV) in memory for small matrices and scaling to a large sparse matrix in a large cluster, where the aggregate memory is sufficient to store the sparse matrix.[95–97] The distributed

solution for sparse matrix multiplication leads to significant network communication and network bandwidth is usually the bottleneck. As such, this operation requires a fast network to achieve performance. A supercomputer or a large cluster with a fast network is inaccessible or too expensive for many users.

On the other hand, a current trend for hardware design is to scale up a single machine for high performance computing. These machines typically have multiple processors with many CPU cores and a large amount of memory. They are also equipped with fast flash memory such as solid-state drives (SSDs) to further extend memory capacity. This conforms to the node design for supercomputers.[98]

We explore a solution that scales sparse matrix dense matrix multiplication (SpMM) on a multi-core machine with commodity SSDs and perform SpMM in semi-external memory (SEM). The concept of semi-external memory arose as a functional computing approach for graphs[26] in which the vertex state of a graph is stored in memory and the edges accessed from external memory. We introduce a similar construct for SpMM in which one or more columns of a dense matrix are kept in memory and the sparse matrix is accessed from external memory. In semi-external memory, we assume that the memory of a machine is sufficient to keep at least one column of the input dense matrix but is insufficient to hold the sparse matrix and the dense matrices. Even though SpMM could be implemented with SpMV, such an implementation would fail to explore data locality in SpMM and result in higher I/O access in semi-external memory. We optimize SpMM directly to overcome these problems. Given fast SSDs,

we demonstrate that the SEM solution uses the resources of a multi-core machine well and achieves performance comparable to state-of-the-art in-memory implementations while increasing the scalability in proportion to the ratio of non-zero entries to rows or columns in a sparse matrix.

We overcome many technical challenges to construct a sparse matrix multiplication implementation on SSDs to achieve performance. Specifically, SSDs are an order of magnitude slower in throughput and multiple orders of magnitude slower in latency than DRAM. Furthermore, sequential I/O of SSDs is still much faster than random I/O[12] and reads are faster than writes. In addition, SSDs wear out when we write data to them and random writes further shorten the lives of SSDs.[99] As such, our solution needs to sequentialize I/O access and reduce I/O, especially writes.

Semi-external memory provides a scalable and efficient SpMM solution that meets the I/O challenges and incorporates substantial computation optimizations. During the computation, each thread streams its own partitions of the sparse matrix from SSDs, maximizing I/O throughput and avoiding thread synchronization. We buffer all intermediate computation results in memory and stream the output matrix to SSDs at most once, minimizing writes to SSDs. We design a very compact sparse matrix format to accelerate retrieving a sparse matrix from SSDs. Semi-external memory has memory constraints. As such, we deploy only computation optimizations that require a small memory footprint, such as dynamic task scheduling and cache blocking.

Our semi-external memory solution adapts to machines with different memory

capacities. When the dense matrix is larger than memory, we split it vertically into multiple partitions so that each partition can fit in memory. As such, the minimum memory requirement of our solution is $O(n)$, where $n$ is the number of rows in the input dense matrix. By keeping more columns in the dense matrix in memory, we reduce I/O from SSDs in SpMM. When the number of columns in a dense matrix increases, SEM SpMM becomes CPU bound, instead of I/O bound on fast SSDs.

We develop three important applications in scientific computing and data mining with our SEM SpMM: PageRank,[3] eigensolver[100] and non-negative matrix factorization.[94] Each of them requires SpMM with different numbers of columns in dense matrices, resulting in different strategies of placing data in memory. With the three applications, we demonstrate data placement choices for different memory capacities in a machine and the impact of the memory size on the performance of the applications.

Our result shows that for real-world sparse graphs, our SEM SpMM achieves almost 100% performance of our in-memory implementation on a large parallel machine with 48 CPU cores when the dense matrix has more than four columns. Even for SpMV, our SEM implementation achieves at least 65% performance of our in-memory implementation and outperforms Trilinos[101] and MKL[102] by a factor of $2 - 9$. The applications implemented with our SpMM significantly outperform the state-of-the-art implementations of these applications. As such, we conclude that semi-external memory coupled with SSDs delivers an efficient solution for large-scale sparse ma-

trix multiplication. It serves as a building block and offers new design possibilities for large-scale data analysis, replacing memory with larger, cheaper, more energy-efficient SSDs and processing bigger problems on fewer machines. The code of this work is released as open source at http://flashx.io.

## 4.2 Related Work

Recent sparse matrix multiplication studies focus on in-memory optimizations for sparse matrix vector multiplication (SpMV). Williams et al.[95] describe optimizations for SpMV in multicore architectures. Yoo et al.[96] and Boman et al.[97] optimize distributed SpMV for large scale-free graphs with 2D partitioning to reduce communication between machines. In contrast, Sparse matrix dense matrix multiplication (SpMM) receives less attention from the high-performance computing community. Even though SpMM can be implemented with SpMV, SpMV fails to explore data locality in SpMM. Aktulga et al.[103] optimizes SpMM with cache blocking. Koanantakool et al.[?] experiments different parallel algorithms for sparse matrix dense matrix multiplication and analyzes their communication cost in distributed memory. We advance SpMM with a focus on optimizations for semi-external memory.

Compressed row storage (CSR) and compressed column storage (CSC) formats are commonly used sparse matrix formats in many numeric libraries such as Intel MKL[102] and Trilinos.[101] However, these formats are not designed for graphs. Sparse matrix

multiplication with these formats on graphs incurs many random memory accesses. More modern sparse matrix formats have been designed. Sparsity[104] proposes both register blocking and cache blocking to increase data reuse in the CPU cache for sparse matrix multiplication. Register blocking requires explicit storage of zero values in register blocks. This strategy potentially wastes space and computation for graphs because graphs are very sparse and have nearly random vertex connection. Buluc et al.[2] further advances sparse matrix format by doubly compressed sparse column (DCSC) for hypersparse submatrices after 2D partitioning on a sparse matrix. This format significantly reduces the storage size of a 2D-partitioned sparse matrix. We adopt some of the optimizations and further advance the sparse matrix format with a focus on reducing the storage size of a sparse matrix.

Abello et al.[26] introduced the semi-external memory algorithmic framework for graphs. Pearce et al.[49] implement several semi-external memory graph traversal algorithms for SSDs. FlashGraph[14] adopted the concept and performs graph algorithms with vertex state in memory and edge lists on SSDs. This work extends the semi-external memory concept to matrix operations.

Zhou et al.[105] implemented an LOBPCG[106] eigensolver in an SSD cluster. Their implementation targets nuclear many-body Hamiltonian matrices, which are much denser and have smaller dimensions than many sparse graphs. Therefore, their solution stores the sparse matrix on SSDs and keep the entire vector subspace in RAM. They focus on optimizations in the distributed environment. In contrast, our eigen-

solver based on our SEM SpMM stores both the sparse matrix and the vector subspace on SSDs due to the large number of vertices in our target graphs. We focus on external-memory optimizations in a single machine.

Anasazi[100] is an eigensolver framework in the Trilinos project.[101] This framework implements block extension of multiple eigensolver algorithms such as Block Krylov-Schur,[107] Block Davidson[106] and LOBPCG.[106] This is a very flexible framework that allows users to redefine sparse matrix dense matrix multiplication and dense matrix operations. By default, Anasazi uses the matrix implementations in Trilinos that runs in distributed memory.

Intel Math Kernel Library[102] is an efficient and parallel linear algebra library with matrix operations specifically optimized for Intel platforms. It provides an efficient sparse matrix multiplication optimized for regular sparse matrices. In contrast, our sparse matrix multiplication optimizes for power-law graphs with near-random vertex connection.

## 4.3 Sparse matrix multiplication

Sparse matrix multiplication leads to many random memory accesses and its performance is usually limited by random memory throughput of DRAM. We perform sparse matrix multiplication in semi-external memory (SEM) to scale to a sparse matrix with billions of rows and columns. This strategy enables nearly in-memory

performance while achieving scalability in proportion to the ratio of non-zero entries to rows or columns in a sparse matrix.

### 4.3.1 Semi-external memory

Our definition of semi-external memory for sparse matrix multiplication keeps the sparse matrix on SSDs and the input dense matrix or some columns of the input dense matrix in memory. During the computation, we stream data in the sparse matrix from SSDs to maximize I/O throughput.

There are two options for keeping the output dense matrix. In applications such as PageRank and many other graph algorithms, dense matrices have only a few columns, so we can keep the output dense matrix in memory. If a machine has insufficient memory to keep the output matrix, we stream the output matrix to SSDs or to the subsequent computation to reduce memory consumption and potentially I/O as well.

In some applications such as non-negative matrix factorization (Section 4.4), even the input dense matrix cannot fit in memory. In this case, we partition the input dense matrix vertically so that each partition has complete columns of the original input dense matrix and can fit in memory. Each vertical partition stores elements in the row-major order to increase data locality. For each partition, we perform sparse matrix multiplication in semi-external memory as before and stream the output matrix to SSDs.

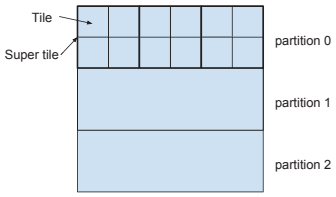



**Figure 4.1:** The format of a sparse matrix.

### 4.3.2 Sparse matrix format

To support efficient sparse matrix multiplication on graphs in semi-external memory, we use an alternative format for sparse matrices to increase CPU cache hits and reduce I/O from SSDs. Compressed row storage (CSR) or compressed column storage (CSC) are not designed for graphs and incur many CPU cache misses. They also require a relatively large storage size. For a sparse matrix with billions of non-zero entries, we have to use eight bytes to store row and column indices.

To increase CPU cache hits, we deploy cache blocking[104] and store non-zero entries of a sparse matrix in tiles (Figure 4.1). When a tile is small, the rows from the input and output dense matrices involved in multiplication with the tile are always kept in the CPU cache during the computation. The optimal tile size should fill the CPU cache with the rows from the dense matrices and is affected by the number of columns of the dense matrices. To handle dense matrices with different numbers of columns, we deploy both static cache blocking and dynamic cache blocking. We generate sparse matrices with a relatively small tile size and rely on the runtime system to optimize for different numbers of columns (Section 4.3.4). However, a small tile size potentially increases the storage size of a sparse matrix. In semi-external memory,

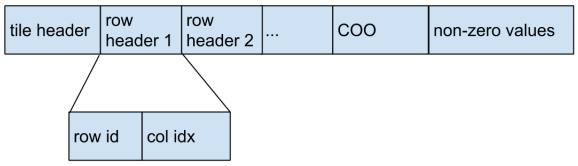



**Figure 4.2:** The storage format (SCSR + COO) of a tile in a sparse matrix.

the dense matrices usually have a very small number of columns in sparse matrix multiplication. Therefore, we use the tile size of $16K \times 16K$ by default to balance the matrix storage size and the adaptibility to different numbers of columns.

We need a very compact format to store non-zero entries in a tile to reduce the number of bytes in a sparse matrix read from SSDs. A compact format is performance critical because in semi-external memory SpMM SSDs may be the bottleneck. In very sparse matrices many rows in a tile do not have any non-zero entries. The CSR or CSC formats waste space because they require an entry for each row/column in the row/column index. Doubly compressed sparse column (DCSC)[2] is proposed to avoid this problem and is compact for hypersparse matrices such as submatrices of a sparse matrix. However, even DCSC wastes space because it requires the storage of pointers to columns and fails to reference to non-zero values with small integers (e.g., two-byte integers).

We use a very compact format to store non-zero entries and refer to this format as SCSR (Super Compressed Row Storage) (Figure 4.2). Like DCSC, this format keeps data only for rows with non-zero entries in a tile. Each non-empty row has a row header that only contains an identifier to indicate the row number, followed

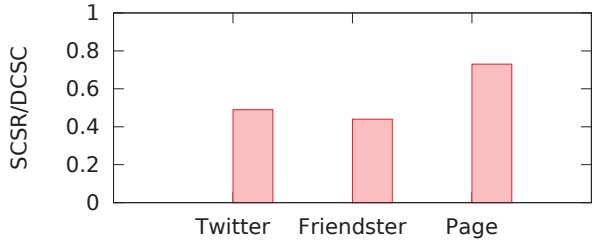



**Figure 4.3:** The ratio of the storage size required by SCSR and DCSC[2] format for real-world graphs. SCSR is much more compact than DCSC for graphs.

by column indices. To determine the end of a row, the most significant bit of the identifier is always set to 1, while the most significant bit of a column index entry is always set to 0. Owing to the small size of a tile, we use two bytes to store a row number and a column index entry, which further reduces the storage size. As such, each non-zero entry requires at most four bytes to indicate its location in a matrix. Because the most significant bit is used to indicate the beginning of a row, this format allows a maximum tile size of $32K \times 32K$.

SCSR is much more compact than DCSC.[2] In the best case (each non-zero entry is stored in a separate row/column), SCSR requires only 40% of the space than DCSC for binary sparse matrices. In the worst case (all non-zero entries are stored in a single row/column), SCSR uses the same space as DCSC. Figure 4.3 shows that SCSR uses 45%-70% of the storage size used by DCSC for large real-world graphs (Table 4.1). In practice, SCSR saves more space in a sparse matrix where non-zero entries are more randomly distributed.

Inside each cache tile of the SCSR, we use the coordinate format (COO) for the

rows that have only a single non-zero entry. For the adjacency matrices of real-world graphs, many rows in a cache tile have only one non-zero entry, owing to the sparsity of the graphs and nearly random vertex connection. Iterating over single-entry rows in the SCSR format requires to test the end of a row for every non-zero entry, which leads to many conditional jumps. In contrast, COO is more suitable for storing these single-entry rows. It does not increase the storage size but significantly reduces the number of conditional jump instructions. As a result, we combine SCSR with COO and store non-zero entries in the COO format behind the row headers of SCSR (Figure 4.2).

### 4.3.3 Dense matrices

In many applications, the dense matrices in SpMM are tall and skinny with millions or even billions of rows but only a small number of columns. The number of columns is determined by applications. In semi-external memory, we keep the input dense matrix in memory, so its size governs memory consumption of sparse matrix multiplication. To increase data locality in SpMM, the elements in the dense matrices are stored in row-major order.

For a non-uniform memory architecture (NUMA), we partition the input dense matrix horizontally and store partitions evenly across NUMA nodes. The NUMA architecture is prevalent in today's multi-processor servers, where each processor connects to its own memory banks. Therefore, keeping partitions evenly across all NUMA

nodes helps to fully utilize the bandwidth of memory and inter-processor links. For horizontal partitioning, we assign multiple contiguous rows in a row interval to a partition, which is assigned to a NUMA node. A row interval in a partition always has $2^i$ rows for efficiently locating a row with bit operations. The row interval size is multiple of the tile size of a sparse matrix so that multiplication on a tile only needs to access rows from a single row interval.

### 4.3.4 Parallel Execution

This section describes parallel execution of sparse matrix multiplication in semi-external memory. Memory is precious resource in this computation model because memory should be used to keep more columns in the input dense matrix to reduce I/O from SSDs (as discussed in Section 4.3.6). As such, we deploy only computation optimizations with a small memory footprint.

Semi-external memory favors horizontal partitioning on a sparse matrix for parallelization because this partitioning scheme minimizes writes to SSDs and remote memory with small memory consumption. Horizontal partitioning requires only one thread to allocate local memory buffers for computation on a tile row. All intermediate computation results on tiles are merged into the local memory buffers. As such, we write the output matrix at most once to SSDs and there are no remote memory writes. In contrast, both vertical partitioning and 2D partitioning require each thread to maintain a local memory buffer for the same tile rows in order to reduce writes to

SSDs and remote memory.

For parallel computation, we construct a global task queue *trQ* and each thread runs *ProcessTileRows* (in Algorithm 1) that gets computation tasks (tile rows) to achieve load balancing and large I/O writes to SSDs. At the beginning, a thread gets larger computation tasks (one super tile row at a time) with *get_super_tile_row*; as the computation approaches completion, a thread gets smaller tasks (one tile row at a time) with *get_tile_row*. This design reduces concurrent access to the global data structure while realizing good load balancing. When a thread gets a computation task, it reads the corresponding tile rows asynchronously, invokes the callback function *ProcessSTRow* to process the tile rows once I/O is complete, and write computation results back to SSDs asynchronously. To ensure sustainable write throughput to SSDs,[99] we need large writes. *write_rows_async*() postpones a write, merges it with the ones from other threads and writes them to SSDs with a single I/O. To assist in I/O merging, we use *get_tile_row()* and *get_super_tile_row()* to control a global execution order that ensures that all threads are processing contiguous tile rows and the results from the tile rows are located closely on SSDs.

When processing tile rows, we organize tiles into super tiles with *get_super_tiles* to better utilize the CPU cache. The tile size of a sparse matrix is specified when the sparse matrix image is created and is relatively small to handle different numbers of columns in the dense matrices. A super tile is composed of tiles from multiple contiguous tile rows (Figure 4.1) and its size is determined at runtime by three factors:

**Algorithm 1** Parallel execution of sparse matrix dense matrix multiplication.

1: **procedure** SPARSEMATRIXMULTIPLY($spm$, $inM$)
   **Input**: $spm$, a $n \times n$ sparse matrix on SSDs
   **Input**: $inM$, a $n \times p$ dense matrix in memory
   **Output**: $outM$, a $n \times p$ dense matrix on SSDs
2: $\vec{trQ} \leftarrow get\_tile\_row\_ids(spm)$
3: $outM \leftarrow zeros\_SSD(n, p)$
4: **parfor** $thread \in \vec{threads}$ **do**
5: $ProcessTileRows(spm, \vec{trQ}, inM, outM)$
6: **end parfor**
7:
8: **procedure** PROCESSTILEROWS($spm$, $\vec{trQ}$, $inM$, $outM$)
   **Input**: $spm$, a $n \times n$ sparse matrix on SSDs
   **Input**: $\vec{trQ}$, a queue that contains all tile row ids
   **Input**: $inM$, a $n \times p$ input dense matrix in memory
   **In-Out**: $outM$, a $n \times p$ output dense matrix on SSDs
9: **while** $|\vec{trQ}| > 0$ **do**
10: **if** $|\vec{trQ}| > \#threads$ **then**
11: $\vec{ids} \leftarrow get\_super\_tile\_row(\vec{trQ})$
12: **else**
13: $\vec{ids} \leftarrow get\_tile\_row(\vec{trQ})$
14: $t \leftarrow read\_tilerow\_async(spm, \vec{ids})$
15: $res \leftarrow ProcessSTRow(t, inM)$ when $t$ is ready
16: $write\_rows\_async(outM, res)$
17:
18: **procedure** PROCESSSTROW($trs$, $inM$)
   **Input**: $trs$, a $s \times n$ submatrix in the sparse matrix
   **Input**: $inM$, $n \times p$ dense matrix
   **Output**: $outBuf$, a $s \times p$ dense matrix
19: $outBuf \leftarrow zeros(s, p)$
20: $\vec{sts} \leftarrow get\_super\_tiles(trs)$
21: **for** $st \in \vec{sts}$ **do**
22: $\vec{tiles} \leftarrow get\_tiles(st)$
23: **for** $tile \in \vec{tiles}$ **do**
24: $lInMat \leftarrow$ rows from $inM$ for $tile$
25: $lOutMat \leftarrow$ rows from $outBuf$ for $tile$
26: $lOutMat += tile * lInMat$

the number of columns in the dense matrices, the CPU cache size and the number of threads that share the CPU cache. An optimal size for a super tile fills the CPU cache with the rows from the dense matrices involved in the computation with the super tile.

*ProcessSTRow* in Algorithm 1 processes tile rows. It splits the input tile rows into super tiles with *get_super_tiles* and then further into tiles with *get_tiles*. It processes all tiles in a super tile before moving to the next one. As such, the computation on a tile reuses data in the CPU cache from the computation on the previous tile. We maintain a local memory buffer $outBuf$ to store the computation results, which minimizes remote memory access. Once the computation in *ProcessSTRow* is complete, $outBuf$ contains complete results.

In spite of nearly random edge connection in a real-world graph, we explore regularity in multiplying a tile with a partition of a row-major dense matrix. In this case we multiply a non-zero entry from a tile with all elements in a row of the input dense matrix and add the results to the corresponding row of the output dense matrix. We perform these operations with vector CPU instructions, such as AVX[108] to enable more efficient memory access and computation. The current implementation relies on GCC's auto-vectorization to translate the C code to vector CPU instructions by predefining the matrix width in the code.

### 4.3.5 I/O optimizations

Semi-external memory sparse matrix multiplication streams a sparse matrix from SSDs, which results in sequential I/O. This I/O access pattern does not generate any cache hits in the Linux page cache when the sparse matrix size is larger than main memory. As such, we access a sparse matrix on SSDs with direct I/O. For accessing data in fast SSDs sequentially, the overhead of operating systems such as thread context switch and memory allocation becomes noticeable. We tackle these obstacles to maximize I/O throughput.

We issue asynchronous I/O and poll for I/O to avoid thread context switches because the latency of a context switch can undermine the sequential I/O throughput of a high-speed SSD array. When a thread issues an I/O request and waits for I/O completion, the operating system switches the thread out; the operating system reschedules the thread for execution once I/O is complete. However, there is latency for thread rescheduling and the latency from frequent rescheduling can cause noticeable performance degradation on a high-speed SSD array. As such, we use I/O polling to avoid a thread from being switched out after the thread completes all computation available to it.

When accessing a sparse matrix or a dense matrix from SSDs, we maintain a set of memory buffers for I/O access to reduce the overhead of memory allocation. We use large I/O to access matrices on SSDs to increase I/O throughput. Large memory allocation is expensive because the operating system usually allocates a large memory

buffer with *mmap()* and populates the buffer with physical pages when it is used. Therefore, we keep a set of memory buffers allocated previously and reuse them for new I/O requests. For accessing a sparse matrix, tile rows usually have different sizes, so we resize a previously allocated memory buffer if it is too small for a new I/O request.

### 4.3.6 The impact of the memory size on I/O

More memory reduces I/O in semi-external memory. The minimum memory requirement for semi-external memory sparse matrix multiplication is $nc + t\epsilon$, where $n$ is the number of rows of the input dense matrix, $c$ is the element size in bytes, $t$ is the number of threads processing the sparse matrix and $\epsilon$ is the buffer size for the sparse matrix and the output dense matrix. When a machine does not have sufficient memory to keep the entire input dense matrix in memory, we need multiple passes on the sparse matrix to complete the computation. Reducing memory consumption is essential to achieve performance in semi-external memory. By keeping more columns of the input dense matrix in memory, we reduce the number of I/O passes.

When a machine does not have sufficient memory to keep the entire input dense matrix, we use the existing memory to keep as many columns in the input dense matrix in memory as possible. Although we can use some memory to cache part of the sparse matrix, keeping more columns of the input dense matrix in memory saves more I/O than using the same amount of memory to cache the sparse matrix. Assume

the input dense matrix has $n$ rows and $k$ columns. Again, $c$ is the element size in bytes. The storage size of the sparse matrix is $E$ bytes and the memory size is $M$ bytes. We further assume we use $M'$ bytes to keep some columns of the dense matrices in memory ($M' < M$, $nck \mod M' \equiv 0$) and the remaining memory ($M - M'$) to cache the sparse matrix. The amount of data in the sparse matrix read from SSDs is

$$IO_{in} = \frac{nck}{M'}[E - (M - M')]$$

Because $E > M$ in semi-external memory, we minimize $IO_{in}$ by maximizing $M'$. Therefore, using memory for the input dense matrix always results in a smaller amount of I/O than using memory for caching the sparse matrix.

As the number of columns in memory from the input dense matrix increases, the bottleneck of the system may switch. When we keep only one column of the input dense matrix in memory, the system is usually I/O bound; when we keep more columns of the dense matrix in memory, the system will become CPU bound and the I/O complexity does not affect its performance.

### 4.3.7 I/O complexity

The semi-external memory (SEM) solution for sparse matrix multiplication leads to no more I/O than the external-memory (EM) solution for many real-world graphs.

When a machine has sufficient memory to keep the entire input dense matrix in

memory, the SEM solution only needs to read the sparse matrix and the input dense matrix once and write the output dense matrix once. This is the minimum amount of I/O.

When a machine has insufficient memory to keep the input dense matrix, the SEM solution still leads to less I/O than the EM solution when $E < nckt$ if we minimize writes to SSDs. In this analysis, we assume a square sparse matrix. The same analysis applies to a rectangular sparse matrix as well. In this case, the SEM solution scans the sparse matrix multiple times.

$$read_{SEM} = \frac{nck}{M}E + nck$$

To minimize writes, the EM solution scans the sparse matrix once but reads the input dense matrix multiple times. Due to near random vertex connection in real-world graphs, the EM solution needs to read the entire input dense matrix each time. In the parallel setting, the EM solution requires each thread to keep local memory buffers for portions of the input and output dense matrices. Assume the EM solution keeps $j$ rows from the input dense matrix and $i$ rows from the output dense matrix in memory in each thread.

$$(ick + jck)t = M \implies i < \frac{M}{ckt}$$

$$read_{EM} = \frac{n}{i}nck + E \implies read_{EM} > \frac{n^2c^2k^2t}{M} + E$$

When $nck < E < nckt$, $read_{EM} > read_{SEM}$. When $E \leq nck$, $read_{EM} > read_{SEM}$ for any $t \geq 2$.

As such, the SEM solution in practice causes less I/O in many natural graphs. For the natural graphs that we have seen, such as Twitter,[89] the Page graph[90] and Friendster,[109] the number of edges is of $10-100\times$ the number of vertices. Essentially, natural graphs have sparse edge matrices. We target multi-core machines with 10s to 100s of threads. For most of our applications, $k$ is of size 1-30. For very small $k$, the SEM solution can keep the entire input dense matrix in memory and leads to the minimum I/O. For a relatively larger $k$, $E < nckt$ holds for most of natural graphs when the graphs are processed in a large parallel machine. Therefore, the SEM solution usually performs less I/O than EM.

## 4.4 Applications

We apply sparse matrix multiplication to three important applications widely used in data mining and machine learning: PageRank,[3] eigensolver[100] and non-negative matrix factorization.[94] Each application demonstrates a different strategy of using memory for sparse matrix multiplication.

### 4.4.1 PageRank

PageRank is an algorithm to rank the Web pages by using hyperlinks between Web pages. It was first used by Google and is identified as one of the top 10 data mining algorithms.[110] PageRank is a representative of a set of graph algorithms that can be expressed with sparse matrix multiplication or generalized sparse matrix multiplication. Other important examples are label propagation[34] and belief propagation.[111] The algorithm runs iteratively and its update rule for each Web pages in an iteration is

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B(u)} \frac{PR(v)}{L(v)}$$

where $B(u)$ denotes the neighbor list of vertex $u$ and $L(v)$ denotes the out-degree of vertex $v$.

### 4.4.2 Eigensolver

An eigensolver is another commonly used application that requires sparse matrix multiplication. Many algorithms[107,112,113] and frameworks[100,114,115] have been developed to solve a large eigenvalue problem.

We take advantage of the Anasazi eigensolver framework[100] and replace its original matrix operations with our SEM sparse matrix multiplication and external-memory dense matrix operations. To compute eigenvalues of a $n \times n$ matrix, many eigenvalue algorithms for a large sparse matrix require to construct a vector subspace with

a sequence of sparse matrix multiplications and each vector in the subspace has the length of $n$. Due to the sparsity of real-world graphs, the vector subspace is large and we keep vectors in the subspace on SSDs. In addition to sparse matrix multiplication, eigensolvers perform some dense matrix operations on the subspace. For example, eigensolvers need to orthogonalize the vectors in the subspace with dense matrix multiplication. The Anasazi eigensolvers have block extension to update multiple vectors in the subspace simultaneously, which results in sparse matrix dense matrix multiplication. The most efficient Anasazi eigensolver on sparse graphs is the KrylovSchur eigensolver,[107] which updates a small number of vectors (1-4) in the subspace simultaneously. Zheng et al.[116] provides the details of extending the Anasazi eigensolver with external-memory matrix operations.

### 4.4.3 Non-negative matrix factorization

Non-negative matrix factorization (NMF)[94] finds two non-negative low-rank matrices $W$ and $H$ to approximate a matrix $A \approx WH$. NMF has many applications in machine learning and data mining. A well-known example is collaborative filtering[117] in recommender systems. NMF can also be applied to graphs to find communities.[118,119]

Many algorithms are designed to solve NMF and here we describe an algorithm[94] that requires a sequence of sparse matrix multiplications. The algorithm uses multiplicative update rules and updates matrices $W$ and $H$ alternately. In each iteration,

the algorithm first fixes $W$ to update $H$ and then fixes $H$ to update $W$.

$$H_{a\mu} \leftarrow H_{a\mu}\frac{(W^TA)_{a\mu}}{(W^TWH)_{a\mu}}, W_{ia} \leftarrow W_{ia}\frac{(AH^T)_{ia}}{(WHH^T)_{ia}}$$

We apply SEM sparse matrix multiplication to NMF differently based on the memory size and the number of columns in $W$ and $H$. Due to the sparsity of a graph, $W$ and $H$ may require storage as large as the sparse matrix and can no longer fit in memory. Therefore, we partition $W$ and $H$ vertically and run multiple sparse matrix multiplications to compute $W^TA$ and $AH^T$, if the memory is not large enough.

## 4.5 Experimental Evaluation

We evaluate the performance of semi-external memory sparse matrix multiplication on multiple real-world billion-scale graphs including a web-page graph with 3.4 billion vertices. We first measure the performance of our semi-external memory implementation and compare it with multiple in-memory implementations: *(i)* our in-memory implementation, *(ii)* MKL (*mkl_dcsrmm*) and *(iii)* Trilinos Tpetra. We also demonstrate the effectiveness of CPU and I/O optimizations on sparse matrix multiplication. We then evaluate the overall performance of the applications in Section 4.4 and demonstrate the impact of the memory size on the applications.

We conduct experiments on a non-uniform memory architecture machine with four Intel Xeon E7-4860 processors, clocked at 2.6 GHz, and 1TB memory of DDR3-

| Graph datasets | # Vertices | # Edges | Directed |
|---|---|---|---|
| Twitter[89] | 42M | 1.5B | Yes |
| Friendster[109] | 65M | 1.7B | No |
| Page graph[90] | 3.4B | 129B | Yes |
| RMAT-40[120] | 100M | 3.7B | Yes & No |
| RMAT-160[120] | 100M | 14B | Yes & No |

**Table 4.1:** Graph data sets. We construct a directed and undirected version for both RMAT-40 and RMAT-160.

1600. Each processor has 12 cores. The machine has three LSI SAS 9300-8e host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 24 OCZ Intrepid 3000 SSDs are installed. The 24 SSDs together are capable of delivering 12 GB/s for read and 10 GB/s for write at maximum. The machine runs Linux kernel v3.13.0. We use 48 threads for our in-memory and semi-external implementation.

We use the adjacency matrices of the graphs in Table 4.1 for performance evaluation. The smallest graph we use has 42 million vertices and 1.5 billion edges. The largest graph is the Page graph with 3.4 billion vertices and 129 billion edges, which is two orders of magnitude larger than the smallest graphs. We generate two synthetic graphs with R-Mat[120] to fill the size gap between the smallest and largest graph. We construct a directed and undirected version for each of the synthetic graphs because some applications in Section 4.4 run on directed graphs and others run on undirected graphs. The real-world datasets are publically available and the synthetic datasets are generated with the RMAT implementation in the *boost* library[1]. We always use the undirected version of the synthetic graphs for the performance evaluation of sparse

[1]We use the parameters of $a = 0.57$, $b = 0.19$, $c = 0.19$, $d = 0.05$.

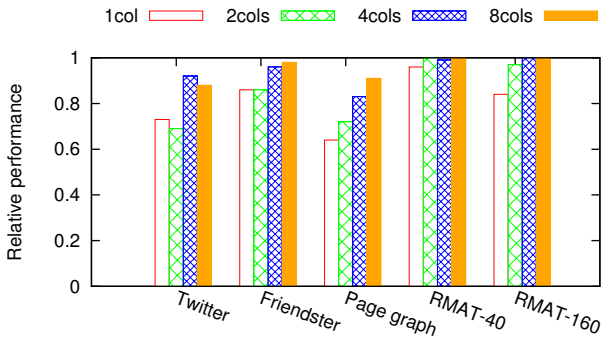



**Figure 4.4:** The performance of SEM-SpMM with dense matrices of different numbers of columns, normalized to IM-SpMM for the dense matrix with the same number of columns.

matrix multiplication. The Page graph is clustered by domain.

### 4.5.1 The performance of sparse matrix multiplication

We evaluate the performance of our semi-external memory implementation (SEM-SpMM) and compare its performance with our in-memory implementation (IM-SpMM) and other state-of-the-art in-memory implementations, including the ones in Intel MKL and Trilinos Tpetra, on the graphs in Table 4.1. The MKL and Tpetra implementations cannot run on the Page graph because its size exceeds the memory capacity of our NUMA machine. We use Intel MKL 2015 and Trilinos v12.0.1 for the experiments.

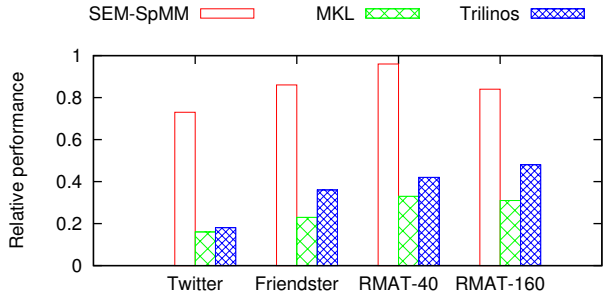
#### 4.5.1.1 SEM-SpMM vs. IM-SpMM

We first compare the performance of SEM-SpMM against IM-SpMM on all graphs with the input and output dense matrices stored in memory. In this case, the dense matrices involved in SpMM have a small number of columns.

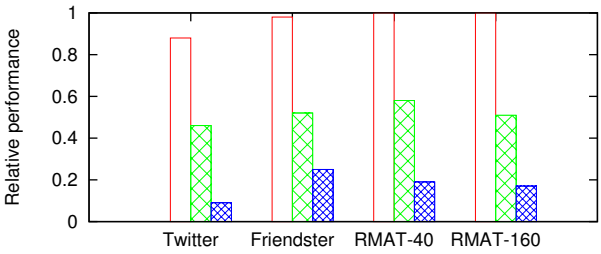
There is only a small performance penalty for semi-external memory (Figure 4.4). The performance gap between IM-SpMM and SEM-SpMM is affected by randomness of vertex connection. The gap is smaller if vertex connection in a graph is more random. The Page graph is relatively well clustered, so SpMM on this graph is less CPU-bound than others. Even for the Page graph, SEM-SpMM gets 65% performance of IM-SpMM. The other factor of affecting the performance gap is the number of columns in the dense matrices. The gap gets smaller as the number of columns in the dense matrices increases. For all graphs, SEM-SpMM requires a very small number of columns to become CPU-bound and achieve 100% performance of IM-SpMM.

#### 4.5.1.2 SEM-SpMM vs. other in-memory SpMM

In this section, we compare SEM-SpMM with the Intel MKL and Trilinos Tpetra implementations. Intel MKL runs on shared-memory machines. Trilinos Tpetra can run in both shared memory and distributed memory, so we measure its performance in our 48-core NUMA machine as well as an EC2 cluster. We run Tpetra in the largest EC2 instances r3.8xlarge, where each has 16 physical CPU cores and 244GB of RAM and is optimized for memory-intensive applications. The EC2 instances are




**(a)** SpMV




**(b)** SpMM with a dense matrix of 8 columns.

**Figure 4.5:** The performance of different sparse matrix multiplication implementations on the 48-core machine normalized to IM-SpMM for the same graphs.

connected with 10Gbps network in the same placement group.

Our SEM-SpMM significantly outperforms Intel MKL and Trilinos Tpetra on the natural graphs on our NUMA machine (Figure 4.5). In this case, we compare performance of our SEM-SpMM with Intel MKL and Trilinos Tpetra for both sparse matrix vector multiplication (SpMV) and sparse matrix dense matrix multiplication (SpMM). The Tpetra implementation is optimized for SpMV. Our SEM-SpMM still constantly outperforms Tpetra by a factor of $2-3$ even for SpMV. The MKL implementation has better optimizations for SpMM than Trilinos Tpetra. Our SEM-SpMM is still almost twice as fast as MKL in SpMM with a dense matrix of eight columns.

SEM-SpMM only consumes a small fraction of memory compared with IM-SpMM

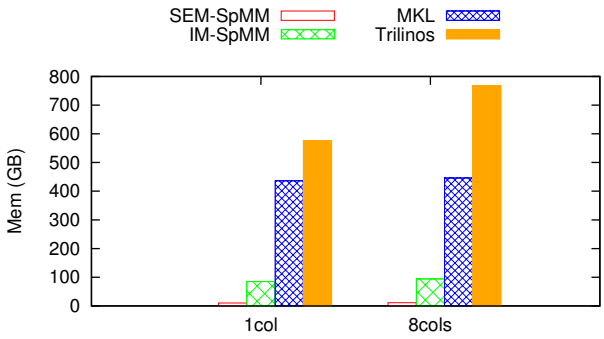
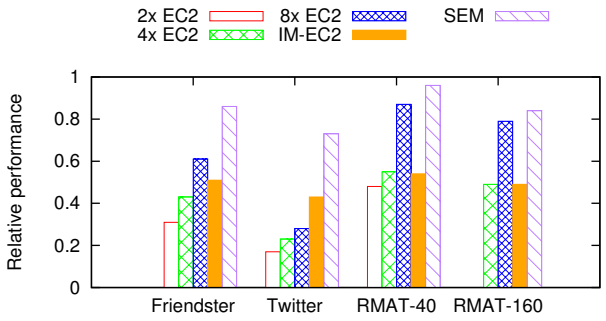


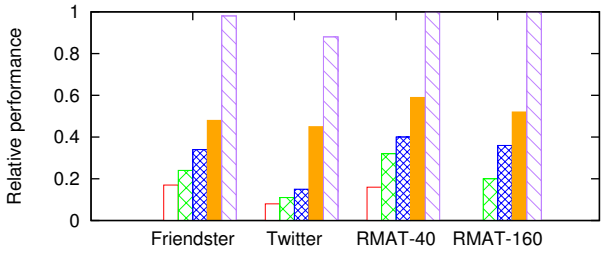

**Figure 4.6:** Memory consumption of different SpMM implementations on RMAT-160.

and other SpMM implementations (Figure 4.6). SEM-SpMM consumes memory for the input dense matrix as well as per-thread local memory buffers for the sparse matrix and the output dense matrix. When we use 48 threads for SpMM, the memory used by local memory buffers in each thread is significant but is relatively constant for different graph sizes. We only show the memory consumption on the largest graph RMAT-160 in Figure 4.5. Despite considerable memory consumed by local memory buffers for SEM-SpMM, SEM-SpMM uses about one tenth of the memory used by IM-SpMM. We also observe that IM-SpMM consumes much less memory than MKL and Tpetra owing to its compact format for sparse matrices.

Our SpMM implementation uses much less computation resources to achieve comparable performance and, in many cases, outperform Trilinos Tpetra that runs in the Amazon cloud, especially on real-world graphs (Figure 4.7). In this experiment, we run our SpMM implementation on both our NUMA machine with 48 CPU cores and one of the EC2 machines with 16 CPU cores. Owing to the compact format for a




**(a)** SpMV




**(b)** SpMM with a dense matrix of 8 columns.

**Figure 4.7:** The performance of SEM-SpMM on our 48-core machine (SEM) and Trilinos Tpetra on EC2 clusters (2xEC2, 4xEC2 and 8xEC2), normalized to IM-SpMM on our 48-core machine for the same graphs. We also show the performance of IM-SpMM on one of the EC2 instance (IM-EC2) where Trilinos Tpetra runs.

sparse matrix, our SpMM implementation can run on all of the graphs in memory on an EC2 instance. When Tpetra runs on 8 EC2 instances, it has 2.5 times as many CPU cores as our NUMA machine. Tpetra is not able to run SpMV on RMAT-160 on two EC2 nodes. Even though an EC2 instance has only 16 physical CPU cores, our IM-SpMM on an EC2 instance achieves around half of the performance of our IM-SpMM on our NUMA machine. In contrast, Trilinos Tpetra uses many more computation resources and still barely reaches the same performance as our IM-SpMM and SEM-SpMM on our NUMA machine. One of the main reasons that our SpMM implementation performs much better on real-world graphs is that these graphs are more likely to cause load imbalance. Our SpMM implementation balances load much better than distributed implementations that partition data.

#### 4.5.1.3 SEM-SpMM with a large input dense matrix

We further measure the performance of SEM-SpMM with a large input dense matrix, in which neither the sparse matrix nor the dense matrices can fit in memory. In this experiment, we measure the performance of multiplying a sparse matrix with a dense matrix of 32 columns and the input dense matrix is stored on SSDs initially. We study the impact of memory size on the performance of SEM-SpMM by artificially varying the number of columns that can fit in memory. SEM-SpMM accesses the sparse matrix with direct I/O and, thus, varying the number of columns in the dense matrix that fit in memory does not affect data access to the sparse matrix. In each

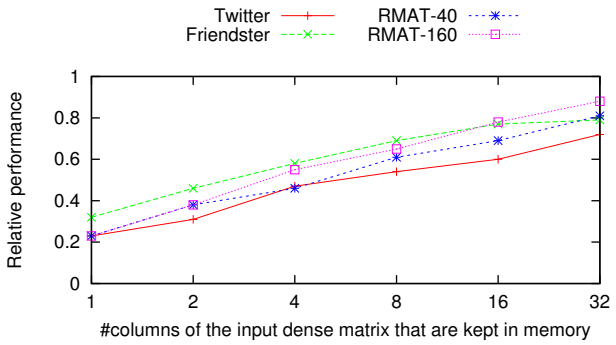



**Figure 4.8:** The performance of SEM-SpMM with a dense matrix of 32 columns relative to IM-SpMM, when the number of columns of the input dense matrix kept in memory varies.

run, we need to load the input dense matrix from SSDs and stream the output dense matrix to SSDs. We do not show the result on the Page graph because the dense matrix with 32 columns for the Page graph cannot fit in memory.

As more columns in the input dense matrix can fit in memory, the performance of SEM-SpMM constantly increases (Figure 4.8). When the memory can fit over four columns of the input dense matrix, SEM-SpMM gets over 50% of the performance of IM-SpMM. Even when only one column of the input dense matrix can fit in memory, SEM-SpMM still gets 25% of the in-memory performance. When the entire input dense matrix can fit in memory, we get about 80% of the in-memory performance.

Two main factors lead to performance loss in SEM-SpMM when the input dense matrix cannot fit in memory. We illustrate the contribution of four potential overheads in SEM-SpMM on the Friendster graph (Figure 4.9). The main performance loss comes from the loss of data locality in SpMM caused by vertical partitioning of the input dense matrix (Vert-part). Partitioning the dense matrix into one-column

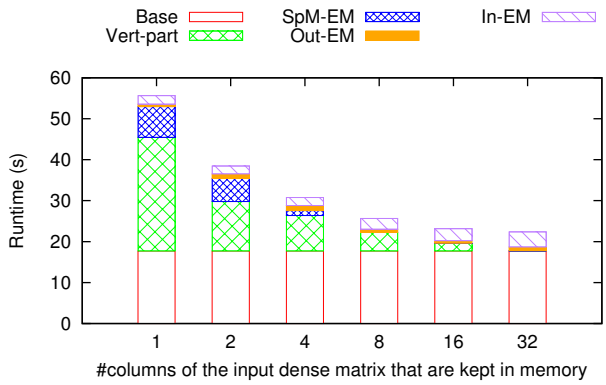



**Figure 4.9:** The overhead breakdown of SEM-SpMM on the Friendster graph with a dense matrix of 32 columns when the number of columns in the input dense matrix kept in memory varies.

matrices contributes 60% of performance loss. It drops quickly when the vertical partition size increases. Keeping the sparse matrix on SSDs (SpM-EM) also contributes some performance loss when the dense matrix is partitioned into small matrices. The overhead almost goes away when more than four columns of the dense matrix can fit in memory. The overhead of streaming the output dense matrix to SSDs (Out-EM) and reading the input dense matrix to memory (In-EM) is less significant and remains the same for different memory sizes.

### 4.5.2 Optimizations on sparse matrix multiplication

Accelerating SEM-SpMM requires both computation and I/O optimizations. We first evaluate the effectiveness of computation optimizations by deploying them on IM-

SpMM. We further show the effectiveness of I/O optimizations by deploying them on SEM-SpMM with all computation optimizations.

Here we illustrate the most significant computation optimizations from Section 4.3. We start with an in-memory implementation that performs sparse matrix multiplication on a sparse matrix in the CSR format and apply the optimizations incrementally in the following order:

- dispatch partitions of a sparse matrix to threads dynamically to balance load (*Load balance*),
- partition dense matrices for NUMA (*NUMA*),
- organize the non-zero entries in a sparse matrix into tiles to increase CPU cache hits (*Cache blocking*),
- use CPU vectorization instructions to accelerate arithmetic computation (*Vec*),

All of these optimizations have positive effects on sparse matrix multiplication and all optimizations together speed up SpMM by $3-5$ times (Figure 4.10). The degree of effectiveness varies between different graphs and different numbers of columns in the dense matrices. The largest performance boost is from cache blocking, especially for SpMV. This is expected because the main overhead of SpMV comes from random memory access and cache blocking significantly increases CPU cache hits to reduce random memory access. CPU vectorization is only effective on SpMM because it optimizes computation on a row of the dense matrix. With all optimizations, we have a fast in-memory implementation for both sparse matrix vector multiplication and

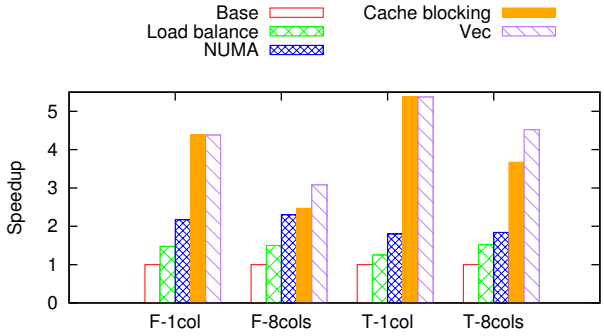



**Figure 4.10:** The speedup of computation optimizations for SpMM on the Friendster graph (F) and the Twitter graph (T) for different numbers of columns in the dense matrices.

sparse matrix dense matrix multiplication.

We evaluate I/O optimizations on SEM-SpMV against a base implementation that has all of the computation optimizations and use doubly compressed sparse row format (DCSR) to store tiles of a sparse matrix. We illustrate their effectiveness on the Friendster graph and the Page graph. The first one represents a graph that is not well clustered; the other one is clustered with domain names. We apply the I/O optimizations in the following order:

- use SCSR to reduce the number of bits read from SSDs (SCSR),
- reduce memory allocation overhead for I/O with per-thread buffer pools (*buf-pool*),
- reduce the number of thread context switches for I/O accesses with I/O polling (*IO-poll*),

The I/O optimizations lead to substantial speedup over the base implementation,

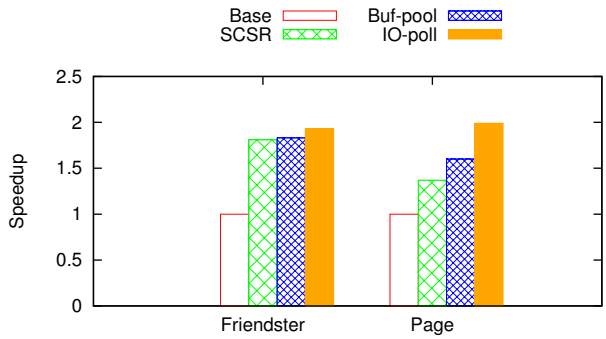



**Figure 4.11:** The speedup of I/O optimizations for SpMV on the Friendster graph and the Page graph.

but behave very differently on these two graphs (Figure 4.11). On the unclustered graph (Friendster), SCSR requires a much smaller storage size than DCSR (Figure 4.3) and thus achieves significant speedup. The Page graph, on the other hand, is well clustered and DCSR already achieves a small storage size. SCSR further reduces the storage size, but is less significant. SpMV on the Page graph has less random memory access and is I/O-bound even on a large SSD array. *Buf-pool* and *IO-poll* increases I/O throughput and, thus, improves performance. In contrast, SEM-SpMV with SCSR on the Friendster graph already achieves almost 80% of IM-SpMV and, thus, further I/O optimizations have less noticeable speedup.

### 4.5.3 Performance of the applications

We evaluate the performance of our implementations of the applications in Section 4.4. We show the effectiveness of additional memory for these applications and compare their performance with state-of-the-art implementations on smaller graphs.

#### 4.5.3.1 PageRank

We evaluate the performance of our SpMM-based PageRank implementation (SpMM-PageRank). This implementation requires the input vector to be in memory, but it is optional to keep the output vector and the degree vector in memory. PageRank is a benchmarking graph algorithm implemented by many graph processing frameworks. We compare the performance of SpMM-PageRank with state-of-the-art implementations in FlashGraph,[14] a semi-external memory graph engine, and GraphLab Create, the next generation of PowerGraph.[5] The PageRank implementation in FlashGraph computes approximate PageRank values while SpMM-PageRank and GraphLab Create compute exact PageRank values. We run GraphLab Create completely in memory and FlashGraph in semi-external memory. GraphLab Create is not able to compute PageRank on the Page graph. We use FlashGraph v0.3 and a trial version of GraphLab Create v1.9.

SpMM-PageRank in memory and in semi-external memory both significantly outperform the implementations in FlashGraph and GraphLab Create (Figure 4.12) even though FlashGraph computes approximate PageRank and GraphLab Create runs completely in memory. The main computation of PageRank is to access PageRank values from neighbor vertices, which is essentially the same computation in sparse matrix vector multiplication. Our SpMM is highly optimized for both CPU and I/O. Even though SpMM-PageRank performs more computation than FlashGraph, it performs the computation much more efficiently and reads less data from SSDs than

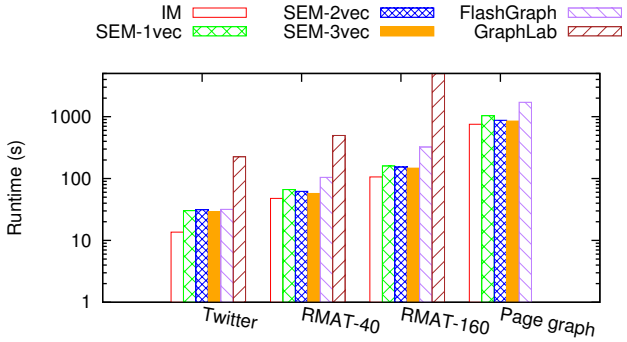



**Figure 4.12:** The runtime of SpMM-PageRank in 30 iterations. The SEM implementation keeps different numbers of vectors in memory (SEM-1vec, SEC-2vec, SEM-3vec). We compare them with the implementations in FlashGraph and GraphLab Create.

FlashGraph. SpMM-PageRank and the implementation in GraphLab create performs the same computation, but SpMM-PageRank performs the computation much more efficiently.

The experiment results also show that keeping more vectors in memory has modest performance improvement for SpMM-PageRank. As such, SpMM-PageRank only needs to keep one vector in memory, which results in very small memory consumption.

#### 4.5.3.2 Eigensolver

We evaluate the performance of our SEM KrylovSchur eigensolver and compare its performance with our in-memory eigensolver and the Trilinos KrylovSchur eigensolver. Usually, spectral analysis only requires a very small number of eigenvalues, so we compute eight eigenvalues in this experiment. We run the eigensolvers on the smaller undirected graphs in Table 4.1. To evaluate the scalability of the SEM eigen-

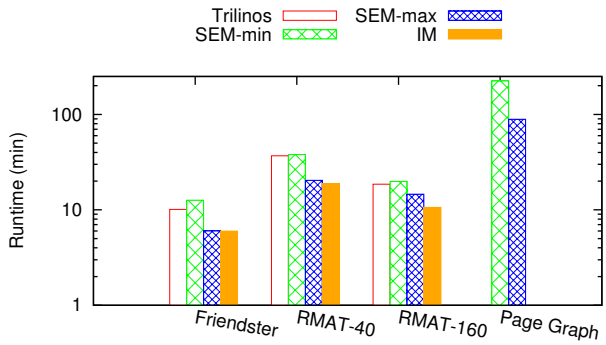



**Figure 4.13:** The runtime of our SEM KrylovSchur, our in-memory eigensolver and the Trilinos eigensolvers when computing eight eigenvalues. SEM-min keeps the entire vector subspace on SSDs and SEM-max keeps the entire vector subspace in memory.

solver, we compute singular value decomposition (SVD) on the Page graph. Among all of the eigensolvers, only our SEM eigensolver is able to compute eigenvalues on the Page graph.

For computing 8 eigenvalues, our SEM eigensolver achieves performance comparable to our in-memory eigensolver and the Trilinos eigensolver and can scale to very large graphs (Figure 4.13). Unlike PageRank, an eigensolver has many more vector or dense matrix operations. As such, the memory size has noticeable impact on performance. For the setting with the minimum memory consumption, it has at least 45% performance of our in-memory eigensolver; when keeping the entire subspace in memory, it has almost the same performance as our in-memory eigensolver.

#### 4.5.3.3 NMF

We evaluate the performance of our NMF implementation (SEM-NMF) on the directed graphs in Table 4.1. The dense matrices for NMF can be as large as the

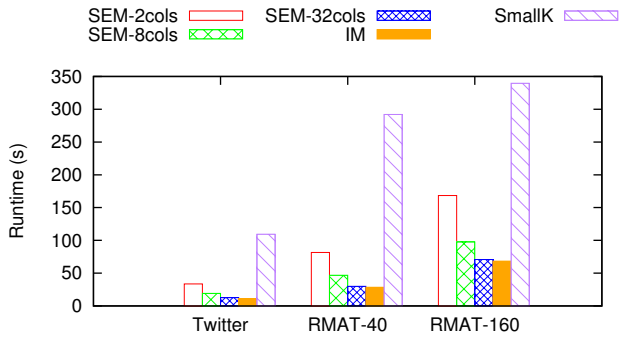



**Figure 4.14:** The runtime per iteration of SEM-NMF on directed graphs. We vary the number of columns in the dense matrices that are kept in memory to evaluate effect of the memory size on the performance of SEM-NMF.

sparse matrix. As such, we experiment with the effect of the memory size on the performance of SEM-NMF by varying the number of columns in memory from the dense matrices. We also compare the performance of SEM-NMF with a high-performance NMF implementation SmallK,[121] built on top of the numeric library Elemental.[122] We factorize each of the graphs into two $n \times k$ non-negative dense matrices and we use $k = 16$ because 16 is the largest $k$ that SmallK supports for the graphs in Table 4.1. We use SmallK v1.6 and Elemental v0.85.

We significantly improve the performance of SEM-NMF by keeping more columns of the input dense matrix in memory (Figure 4.14). The performance improvement is more significant when the number of columns that fit in memory is small. When we keep eight columns of the input dense matrix in memory, SEM-NMF achieves over 60% of the performance of the in-memory implementation.

SEM-NMF significantly outperforms other NMF implementations in the literature. SmallK is the closest competitor. We run the same NMF algorithm in SmallK.

As shown in Figure 4.14, SEM-NMF outperforms SmallK by a large factor on all graphs. There are many MapReduce implementations in the literature.[123–125] They run on sparse matrices with tens of millions of non-zero entries but generally take one or two orders of magnitude more time than our SEM-NMF on the sparse matrices with billions or even tens of billions of non-zero entries.

## 4.6 Conclusions

We present an alternative solution for scaling sparse matrix dense matrix multiplication (SpMM) to large sparse matrices by utilizing commodity SSDs in a large parallel machine. We perform this operation in semi-external memory (SEM), in which we keep the sparse matrix on SSDs and the dense matrices in memory. Semi-external memory increases scalability in proportion to the ratio of non-zero entries to rows or columns in a sparse matrix. SEM SpMM requires both memory optimizations, such as cache blocking and NUMA organization, and I/O optimizations, such as I/O polling and memory buffer pools, to realize performance.

Our SEM SpMM achieves performance comparable to our highly optimized in-memory implementation while significantly outperforming the Intel MKL and Trilinos implementations. Our SEM implementation achieves almost 100% performance of the in-memory implementation on some graphs when the dense matrices can fit in memory and have more than four columns. Even when the dense matrix has only

one column, it achieves at least 65% of the performance of its in-memory counterpart on different graphs. Our SEM sparse matrix multiplication also scales to very large graphs with billions of vertices and hundreds of billions of edges.

For a machine with insufficient memory to keep the entire input dense matrix in memory, we partition the dense matrix vertically and run SEM SpMM multiple times. In this case, the main overhead of SEM SpMM comes from the loss of data locality caused by vertical partitioning on the dense matrix. However, given sufficient memory to keep a small number of columns of the input dense matrix, we achieve performance comparable to the in-memory counterpart.

We apply our sparse matrix multiplication to three important applications: PageRank, eigendecomposition and non-negative matrix factorization. We demonstrate how additional memory should be used in semi-external memory in each application. We further demonstrate that each of our implementations significantly outperform state of the art and scale to very large graphs.

Through thorough evaluation, we demonstrate that semi-external memory coupled with fast SSDs achieves performance very close to highly optimized in-memory implementations and scales to massive datasets. As such, our approach provides a very promising alternative to distributed computation for large-scale data analysis.

Our SSD-based solution also achieves very high energy efficiency even though we have not measured energy consumption explicitly. SSDs are energy-efficient storage media[126] compared with RAM and hard drives. When processing large datasets,

our solution only uses a single machine and requires a relatively small amount of memory. In contrast, a distributed solution requires many more machines and much more aggregate memory in order to process datasets of the same size. As such, our solution introduces an energy-efficient architecture for large-scale data analysis tasks.

# Chapter 5

# FlashMatrix

This chapter describes FlashMatrix, a matrix-oriented programming framework for general data analysis with seamless integration with the R framework. FlashMatrix incorporates the efficient sparse matrix multiplication in Chapter 4. In this chapter, we focus on dense matrix operations in FlashMatrix. It provides a small number of generalized matrix operations (GenOps) to achieve generality, and reimplements a large number of matrix operations in the R framework with GenOps to executes R code in parallel and out of core automatically. FlashMatrix uses vectorized user-defined functions (VUDF) to reduce the overhead of function calls and fuses matrix operations to reduce data movement between CPU and SSDs. We implement multiple machine learning algorithms in R to benchmark the performance of FlashMatrix. The execution of the R implementations in FlashMatrix has performance comparable to optimized C implementations. When scaling beyond memory

capacity on a large parallel machine, the out-of-core execution of these R implementations in FlashMatrix has performance comparable to their in-memory execution and significantly outperforms the in-memory execution of Spark MLlib.

## 5.1 Introduction

The National Strategic Computing Initiative (NSCI[127]) puts forth a critical problem as we move to exascale: *"Increasing coherence between the technology base used for modeling and simulation and that used for data analytic computing."* A key challenge lies in providing statistical analysis and machine learning tools that are simple and efficient. Simple tools need to be programmable, interactive, and extensible, allowing scientists to encode and deploy complex algorithms. Successful examples include R, SciPy, and Matlab. Efficiency dictates that tools should leverage modern HPC architectures, including scalable parallelism, high-speed networking, and fast I/O from memory and solid-state storage.

Large-scale data analysis requires a large parallel machine or a cluster to gain computation power and memory capacity. Currently, there are two approaches of implementing parallel algorithms to process large datasets. One can write an efficient implementation with low-level parallel primitives such as the ones provided by MPI[128] or OpenMP.[129] This approach requires expertise in parallel programming and significant effort from programmers. The other approach is to use high-level programming

frameworks that provide high-level operations to reduce the burden of programmers. In general, the second approach is less computationally efficient but can significantly increase productivity and lower the barrier to writing parallel implementations. The second approach is preferred in the rapidly evolving fields of machine learning and data mining.

It is challenging to provide a programming framework that has a high-level programming interface and achieves both generality and efficiency. Some highly-optimized linear algebra libraries[101,102,122,130,131] provides a matrix programming interface that has a limited set of matrix operations with efficient implementations, e.g. BLAS provides only matrix multiplication and not integer operations or row/column operations. Users have to parallelize the remaining matrix operations themselves that are not supported by the libraries. High-level programming frameworks, such as R and Matlab, provide a general programming interface for users to express varieties of algorithms, but do not produce efficient parallel code.

We present FlashMatrix, a programming framework that provides a high-level matrix-oriented functional programming interface and supports automatic parallelization and out-of-core execution for large-scale data analysis; *users write R programs and FlashMatrix executes them efficiently.* Unlike most of the linear algebra libraries, FlashMatrix provides a small set of highly-optimized generalized matrix operations (GenOps) to achieve generality. GenOps represent common data access patterns. By accepting different functions that define operations on individual elements in the

input matrices, each GenOp covers a very large number of matrix operations. FlashMatrix reimplements many matrix operations from the R *base* package with GenOps to execute R code in parallel and out of core automatically. FlashMatrix focuses on optimizations in a single machine and scales matrix operations beyond memory capacity by utilizing solid-state drives (SSDs). This design choice conforms with a current trend of hardware design that scales up a single machine for high performance computing,[98] including analysis of data stored on SSDs of I/O burst buffers.[132]

We overcome many technical challenges to move data from SSDs to CPU efficiently, overcoming the large speed disparity between CPU and memory, as well as between memory and SSDs. The speed disparity of CPU and DRAM has increased exponentially over the past decades.[133] Even though the I/O performance of SSDs has advanced to outperform hard drives by a large factor, they remain an order of magnitude slower than RAM. On the other hand, many analysis tasks are data-intensive. Matrix formulation further increases data movement between CPU and SSDs because a matrix computation framework typically performs an operation on the entire input matrices before moving to the next operation.

Another challenge in FlashMatrix is to reduce computation overhead. A GenOp in FlashMatrix takes some functions (EleFuns) as additional arguments that define operations on individual elements in the input matrices. To support the matrix operations in the R *base* package, a GenOp accepts functions at run time. As such, each operation on an element potentially results in a function call, incurring computational

overhead.

To move data efficiently, FlashMatrix evaluates expressions lazily and fuses operations aggressively in a single parallel execution job. FlashMatrix builds a directed acyclic graph (DAG) to represent all operations in a single execution. When evaluating the computation in a DAG, FlashMatrix performs two levels of matrix partitioning to improve data utilization in memory hierarchy and reduce data movement between memory and SSDs as well as between CPU and memory. To access data stored on SSDs, FlashMatrix streams data from SSDs to maximize I/O throughput.

To reduce computation overhead, we deploy vectorized EleFuns (VEleFuns) that operates on a vector of elements, instead of an individual element. We define multiple forms for each VEleFun and automatically select the right form for each GenOp to amortize the function call overhead. When invoking VEleFuns, GenOps choose the right vector length to balance the amortization of the function call overhead and CPU cache misses. Inside VEleFuns, we use vector CPU instructions, such as AVX,[108] to further improve performance.

We implement multiple machine learning algorithms, including k-means[27] and Gaussian Mixture Models[134] in FlashMatrix with its R programming interface to benchmark its performance. On a large parallel machine with 48 CPU cores and fast SSDs, the out-of-core execution of these R implementations in FlashMatrix achieves performance comparable to the in-memory execution, while significantly outperforming the same algorithms in Spark MLlib.[8] FlashMatrix effortlessly scales to datasets

with billions of data points and its out-of-core execution uses a small fraction of resources required by in-memory implementations. When running in a single thread, the FlashMatrix implementations outperform the C and FORTRAN optimized implementations of the R framework. In addition, FlashMatrix achieves almost linear speedup in a multicore NUMA machine for all algorithms. We believe FlashMatrix significantly lowers the requirements for writing parallel and scalable implementations of data analysis algorithms; it also offers new design possibilities for data analysis clusters, replacing memory with larger and cheaper SSDs and processing bigger problems on fewer nodes.

## 5.2 Related Work

Basic Linear Algebra Subprograms (BLAS) defines a small set of vector and matrix operations commonly used in scientific computing. There exist a few highly-optimized BLAS implementations such as MKL,[102] OpenBLAS,[130] GotoBLAS[135] and ATLAS.[136] However, these libraries optimize the vector and matrix operations in shared memory. BLAS provides only a small number of vector and matrix operations.

Distributed memory matrix computation libraries[101, 122, 131] speed up computation and scale to larger vectors and matrices. These libraries in general build on top of BLAS and distribute computation with MPI. They provide a limited set of predefined matrix operations and require users to manually parallelize the remaining matrix op-

erations. Instead of providing predefined matrix operations, the core of FlashMatrix provides a few GenOps that represent some common data access patterns. Thus, each GenOp covers a very large number of matrix operations.

There are many distributed data processing frameworks. MapReduce[6] is a general large-scale data processing framework. It provides a single primitive that takes two user-defined functions. Due to the lack of efficient primitives for varieties of data access patterns, algorithms implemented in MapReduce are inefficient. Dryad[7] and Naiad[9] provide more primitives than MapReduce to support various data access patterns more efficiently.

Due to complexity of programming in the distributed execution engines, many programming frameworks have been developed on top of the distributed execution engines. Pig Latin[31] and FlumeJava[32] build on top of MapReduce to provide high-level operations for general data analysis. SystemML[36] builds on top of MapReduce with a focus on machine learning. DryadLINQ[33] builds on top of Dryad and exposes a high-level language to express data analysis tasks. The performance of these programming frameworks is bound by the underlying distributed execution engines.

Spark[8] is a distributed in-memory data processing framework. It provides a highly-optimized machine learning library called MLlib.[29] Spark also provides an R programming interface called SparkR, which focuses on computation on data frames, a table-like data structure in R.

Both academia and industry are making significant effort to bring parallelization

to array programming languages and scale them to large datasets. Revolution R[137] and parallel computing toolbox in MatLab[138] provide parallel linear algebra and data analysis routines as well as explicit parallel programming interface such as MPI and MapReduce. Other works bring implicit parallelization to programming frameworks. Presto[139] extends R to support sparse matrix operations in distributed memory for graph analysis. Ching et. al[140] parallelizes APL code by compiling it to paralllelized C code. Accelerator[141] compiles data-parallel operations on the fly to execute programs in GPU.

## 5.3 Design

FlashMatrix is a matrix-oriented programming framework for general data analysis. It supports both sparse matrix operations and dense matrix operations. This work mainly focuses on dense matrices and scales dense matrix operations beyond memory capacity by utilizing fast I/O devices, such as solid-state drives (SSDs), in a non-uniform memory architecture (NUMA). The implementation and optimization of sparse matrix multiplication is described in.[15] FlashMatrix uses R as its main programming interface and executes R code automatically in parallel and out of core.

Figure 5.1 shows the architecture of FlashMatrix. The core of FlashMatrix provides a small number of generalized matrix operators (GenOps) to simplify the implementation and improve expressiveness of the framework. The optimizer in Flash-

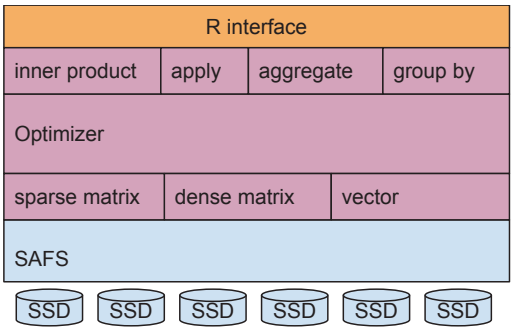



**Figure 5.1:** The architecture of FlashMatrix.

Matrix aggressively merges operations to reduce CPU cache misses and I/O accesses and achieve better parallelization. FlashMatrix stores large matrices on SSDs through SAFS,[12] a user-space filesystem for a large SSD array, to fully utilize high I/O throughput of SSDs and deploys a set of I/O optimizations to improve its sequential I/O throughput.[15]

### 5.3.1 Dense matrices

Dense matrices are the main data types in FlashMatrix. A vector is stored as a one-column dense matrix. In FlashMatrix, a dense matrix can be stored physically in memory or on SSDs or represented virtually by a sequence of computation. FlashMatrix specifically optimizes for tall-and-skinny matrices and short-and-wide matrices, and views tall matrices and wide matrices as groups of tall-and-skinny matrices and short-and-wide matrices, respectively.

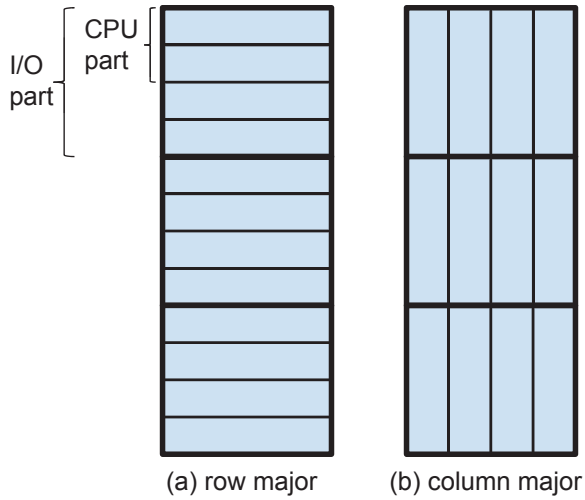



**Figure 5.2:** The format of a tall-and-skinny dense matrix.

#### 5.3.1.1 Tall-and-skinny matrices

FlashMatrix optimizes for tall-and-skinny (TAS) dense matrices due to their frequent occurrence in data analysis. In this field, many data matrices contain a large number of samples with a relatively few features, so data matrices are usually tall and skinny. If a data matrix has many features, the first step is often dimension reduction,[142] which results in a TAS matrix. FlashMatrix specifically optimizes for TAS dense matrices with tens of columns or fewer.

FlashMatrix supports row-major and column-major matrix layout (Figure 5.2). As such, we avoid data copy for common matrix operations such as matrix transpose. FlashMatrix optimizes matrix operations for both data layouts. Each GenOp has its own preferred matrix layout and determines the layout of an output matrix based on the input matrices.

FlashMatrix uses two-level horizontal partitioning on TAS matrices for efficient

data access to SSDs (Figure 5.2). FlashMatrix partitions TAS matrices horizontally into I/O-level partitions. All elements in an I/O-level partition are stored contiguously regardless of the data layout in the matrix. The I/O-level partition size determines an I/O size, usually on the order of megabytes, because each I/O access reads an entire I/O-level partition. The number of rows in an I/O-level partition is always $2^i$. This produces column-major TAS matrices whose data are well aligned in memory to help CPU vectorization. We further split an I/O-level partition horizontally into CPU-level partitions during computation. We use a small CPU-level partition (on the order of kilobytes) so that it fits in CPU L1/L2 cache to reduce CPU cache misses when evaluating a sequence of matrix operations (Section 5.3.4.1). FlashMatrix determines the number of rows in a CPU-level partition based on the number of columns in a matrix.

#### 5.3.1.2 Virtual matrices

In many cases, we do not need to store the data of a matrix physically. Instead, we compute and generate its data on the fly. *Virtual matrices* store computation and potentially the reference to some other matrices required by computation. A simple example is a matrix with all elements having the same value. For such a matrix, we only need to store a single value and construct its matrix partitions during computation.

*Virtual matrices* are essential for lazy evaluation (Section 5.3.4.1). All GenOps

may output *virtual matrices* that represent computation results by storing only the computation of GenOps and the references to input matrices. This strategy is essential for both in-memory and external-memory optimizations to improve performance. It significantly reduces data access to memory and SSDs as well as memory allocation overhead for creating new matrices.

#### 5.3.1.3 A group of dense matrices

FlashMatrix represents a tall matrix with a group of tall-and-skinny matrices and a wide matrix with a group of short-and-wide matrices. We construct a special *virtual matrix* to represent a group of dense matrices. To take advantage of the optimizations on matrix operations on TAS matrices, we decompose a matrix operation on a group of matrices into operations on individual matrices in the group (Section 5.3.3.4). Coupled with the two-level partitioning on TAS matrices, this strategy enables 2D-partitioning on a dense matrix and each partition fits in main memory or CPU cache.

### 5.3.2 Programming interface

FlashMatrix provides a matrix-oriented functional programming interface. The main interface is a small set of GenOps that take matrices and some element functions (EleFuns) as input and output new matrices that store computation results. EleFuns define computation on individual elements in input matrices. We implement both GenOps and EleFuns with C++ and provides a large set of built-in EleFuns.

```
# X is the data matrix
# C is the cluster centers from the previous iteration.
kmeans.iter <- function(X, C)
{
        # Compute the pair-wise distance between a data
        # point and a center.
        D <- fm.inner.prod(X, t(C), "euclidean", "+")
        # Find the closest center to a data point.
        I <- fm.agg.row(D, "which.min")
        # Count the number of data points in each cluster.
        one <- fm.rep.int(1, nrow(I))
        CNT <- fm.groupby.row(one, I, "+")
        # Compute the new centers.
        C <- fm.groupby.row(X, I, "+")
        C <- fm.mapply.row(C, CNT, "/")
        list(C=C, I=I)
}
```

**Figure 5.3:** The R code of computing an iteration of k-means using GenOps.

FlashMatrix exposes GenOps and the built-in EleFuns in its R interface and reimplements many functions from the R *base* package with GenOps.

The R interface provides many functions to support varieties of data analysis algorithms. We categorize the functions into three classes.

- The generalized matrix operators (GenOps), listed in Table 5.1.
- Utility functions include functions that construct FlashMatrix vectors and matrices; functions that convert between FlashMatrix objects and R objects; functions that transform the shape of a matrix; functions that provide additional control on computation and data storage in FlashMatrix. Examples are shown in Table 5.2.
- R matrix computation functions implemented with GenOps. Examples are shown in Table 5.3.

| GenOp | Description |
|---|---|
| $C = fm.inner.prod(A, B, f1, f2)$ | $t = f1(A_{i,k}, B_{k,j})$, $C_{i,j} = f2(t, C_{i,j})$, over all $k$ |
| $C = fm.sapply(A, f)$ | $C_{i,j} = f(A_{i,j})$ |
| $C = fm.mapply(A, B, f)$ | $C_{i,j} = f(A_{i,j}, B_{i,j})$ |
| $C = fm.mapply.row(A, B, f)$ | $C_{i,j} = f(A_{i,j}, B_j)$ |
| $C = fm.mapply.col(A, B, f)$ | $C_{i,j} = f(A_{i,j}, B_i)$ |
| $c = fm.agg(A, f)$ | $c = f(A_{i,j}, c)$, over all $i$, $j$ |
| $C = fm.agg.row(A, f)$ | $C_i = f(A_{i,j}, C_i)$, over all $j$ |
| $C = fm.agg.col(A, f)$ | $C_j = f(A_{i,j}, C_j)$, over all $i$ |
| $C = fm.groupby.row(A, B, f)$ | $C_{k,j} = f(A_{i,j}, C_{k,j})$, where $B_i = k$, over all $i$ |
| $C = fm.groupby.col(A, B, f)$ | $C_{i,k} = f(A_{i,j}, C_{i,k})$, where $B_j = k$, over all $j$ |

**Table 5.1:** The list of generalized matrix operators (GenOps) in FlashMatrix. $A$, $B$ and $C$ are matrices, and $c$ is a scalar.

| Class | Function | Description |
|---|---|---|
| Create | $fm.rep.int$ | Create a vector of a repeated value |
| | $fm.seq.int$ | Create a vector of sequence numbers |
| | $fm.runif.matrix$ | Create a uniformly random matrix |
| | $fm.rnorm.matrix$ | Create a random matrix under a normal distribution |
| Convert | $fm.conv.FM2R$ | Convert a FM matrix to an R matrix |
| | $fm.conv.R2FM$ | Convert an R matrix to a FM matrix |
| Reshape | $t$ | Matrix transpose |
| | $fm.rbind$ | Bind matrices by rows |
| | $fm.cbind$ | Bind matrices by columns |
| Control | $fm.conv.layout$ | Convert the data layout of a matrix |
| | $fm.set.mate.level$ | Set the materialization level of a *virtual matrix* |
| | $fm.materialize$ | Materialize a *virtual matrix* |
| | $fm.conv.store$ | Move a matrix to a specified storage |

**Table 5.2:** Some of the utility functions in FlashMatrix.

| Class | Function | Description |
|---|---|---|
| Element-wise | $C = A + B$ | $C_{i,j} = A_{i,j} + B_{i,j}$ |
| | $C = A - B$ | $C_{i,j} = A_{i,j} - B_{i,j}$ |
| | $C = A * B$ | $C_{i,j} = A_{i,j} * B_{i,j}$ |
| | $C = A/B$ | $C_{i,j} = A_{i,j}/B_{i,j}$ |
| | $C = pmin(A, B)$ | $C_{i,j} = pmin(A_{i,j}, B_{i,j})$ |
| | $C = pmax(A, B)$ | $C_{i,j} = pmax(A_{i,j}, B_{i,j})$ |
| | $C = sqrt(A)$ | $C_{i,j} = sqrt(A_{i,j})$ |
| | $C = abs(A)$ | $C_{i,j} = abs(A_{i,j})$ |
| | $C = exp(A)$ | $C_{i,j} = exp(A_{i,j})$ |
| Aggregate | $c = sum(A)$ | $c = \sum_{i=1}^{n}\sum_{j=1}^{p} A_{i,j}$ |
| | $C = rowSums(A)$ | $C_i = \sum_{j=1}^{p} A_{i,j}$ |
| | $C = colSums(A)$ | $C_j = \sum_{i=1}^{n} A_{i,j}$ |
| | $c = any(A)$ | true if any element is true |
| | $c = all(A)$ | true if all elements are true |
| multiply | $\% * \%$ | matrix multiplication |

**Table 5.3:** Some of the R functions implemented with GenOps.

Figure 5.3 shows an example of computing an iteration of k-means[27] using GenOps. It first uses *fm.inner.prod* to compute the Euclidean distance between every data point and every cluster center, and outputs a matrix with each row representing the distances from a data point to every cluster center. It uses *fm.agg.row* to find the closest cluster for each data point and the output matrix represents how data points are assigned to clusters. It then uses *fm.groupby.row* to count the number of data points in each cluster and compute the mean of each cluster.

### 5.3.3 Efficient generalized operations

FlashMatrix provides efficient implementations of a small number of generialized matrix operations (GenOps) to achieve both generality and efficiency. For generality,

FlashMatrix allows users to pass functions to GenOps to define actual matrix computation. For efficiency, FlashMatrix requires the functions passed to GenOps to be vectorized.

#### 5.3.3.1 Generalized matrix operations

FlashMatrix provides only four GenOps on matrices: *inner product*, *apply*, *aggregation* and *groupby* (Table 5.1). Each operator represents a data access pattern and accepts some element functions as additional arguments that define computation on individual elements in matrices (Section 5.3.3.2).

*Inner product* is a generalized matrix multiplication (*fm.inner.prod*). It replaces multiplication and addition in matrix multiplication with two functions. We define many operations with *inner product*. For example, we use it to compute various pair-wise distances, such as Euclidean distance and Hamming distance, between data points. For dense matrices, we mainly focus on optimizing two cases: *inner product* of a wide matrix and a tall matrix and *inner product* of a tall matrix and a small matrix. It is impractical to materialize *inner product* of a large tall matrix and a large wide matrix owing to space complexity. This holds for all matrix algebra frameworks. When evaluating an *inner product* expression, FlashMatrix uses the BLAS implementation of matrix multiplication for floating-point matrices. This achieves the speed and precision required by numeric libraries, such as eigensolvers.[100,116]

*Apply* is a generalized form of element-wise operations and has multiple variants.

The simplest variant (*fm.sapply*) is an element-wise unary operation. We use it to implement many unary operations such as negation, square root or element type casting on a matrix. The second variant (*fm.mapply*) is an element-wise binary operation. We use it to implement many binary matrix operations such as matrix addition and subtraction. The third (*fm.mapply.row*) and the fourth variants (*fm.mapply.col*) perform element-wise binary operations on the input vector with every row or column of the input matrix and output a matrix of the same shape as the input matrix.

*Aggregation* takes multiple elements and outputs a single element. It has three variants on a matrix. The first variant (*fm.agg*) aggregates over all elements on a matrix, e.g., matrix summation. The second (*fm.agg.row*) and the third variants (*fm.agg.col*) compute aggregation over each individual row or column. *rowSums* and *colSums* in R are examples.

Similarly, *Groupby* on a matrix has two variants. The first variant (*fm.groupby.row*) groups rows and the second variant (*fm.groupby.col*) groups columns of a matrix based on a vector of categorical values and performs aggregation on the rows or columns associated with the same categorical value. Matrix *groupby* is used in classification and clustering algorithms that compute aggregation on the data points in a class or in a cluster.

#### 5.3.3.2 Vectorized element functions

Invoking a function on each element individually would result in significant function call overheads. Instead, all of the GenOps take vectorized element functions (VEleFuns) that operate on a vector of elements instead of an individual element. By transforming the operations on individual elements to the ones on a vector, we amortize the overhead of function calls significantly.

We balance the amortization of function call overhead and CPU cache misses. To reduce latency of accessing data in VEleFuns, the input data has to be small enough to fit in the CPU L1 cache. On the other hand, passing a longer vector to a VEleFun amortizes the overhead of function calls more aggressively. We use 128 as the maximum length of the input vector of a VEleFun.

We have three types of VEleFuns to support the GenOps in FlashMatrix. Each VEleFun type may have multiple forms to allow GenOps to transform operations to increase the length of an input vector to a VEleFun and reduce function call overhead.

- A *unary* VEleFun (*uVEleFun*) takes a vector as input and outputs a vector of the same length.

- A *binary* VEleFun has three forms: the first form (*bVEleFun1*) takes two vectors of the same length and outputs a vector of the same length as the input vectors; the second form (*bVEleFun2*) takes a vector as the left argument and a scalar as the right argument and outputs a vector with the same length as the input

vector; the third form (*bVEleFun3*) takes a scalar as the left argument and a vector as the right argument and outputs a vector. The second and third forms support non-commutative binary operations such as division and subtraction.

- An *aggregation* VEleFun consists of two functions: *aggregate* and *combine*. Both functions may have two forms: the first one (*aVEleFun1*) takes a vector and outputs a scalar; the second one (*aVEleFun2*) takes two vectors of the same length and outputs a vector. For many aggregation VEleFuns such as summation, *aggregate* and *combine* are the same and have both *aVEleFun1* and *aVEleFun2* forms. For some aggregation such as *count*, *aggregate* and *combine* are different.

FlashMatrix provides many commonly used VEleFuns that wrap basic operations built in many programming languages and libraries. For example, FlashMatrix provides arithmetic operations (*addition* and *subtraction*), relational operations (*equal to* and *less than*), logical operations (*logical AND* and *logical OR*), as well as commonly used math functions (*absolute value* and *square root*). FlashMatrix also provides a set of VEleFuns to cast primitive element types.

For each basic operation, FlashMatrix provides multiple VEleFun implementations to support different element types. To reduce the number of binary VEleFun implementations, FlashMatrix only provides the ones that take two input arguments of the same type. If a GenOp gets two matrices with different element types, it first casts the element type of one matrix to match the other. Type casting operations are

implemented with *fm.sapply* and are performed lazily.

FlashMatrix allows programmers to extend the framework by registering new VEleFuns. Like built-in VEleFuns, a new VEleFun needs to provide multiple implementations to support different element types; based on the type of the VEleFun, it may need to provide different forms as described above. FlashMatrix currently requires a VEleFun to be implemented with C/C++.

We use CPU vector instructions such as AVX[108] to accelerate the computation in a VEleFun. The current implementation of FlashMatrix heavily relies on auto-vectorization of a compiler, such as GCC, to vectorize computaion. FlashMatrix provides hints and transforms code to help auto-vectorization. For example, a VEleFun in FlashMatrix frequently operates on vectors with data aligned in memory and of the length defined at compile time, so we inform the compiler of the data alignment and the vector length. Some compilers do not automatically vectorize aggregation operations well. In this case, we manually create a small vector of reduction variables, flatten the loop and transform the original aggregation operation into aggregation onto the vector of reduction variables to help auto-vectorization.

#### 5.3.3.3 Implementation of GenOps with VEleFun

GenOps invoke VEleFuns on the elements of CPU-level partitions intelligently to increase the length of vectors passed and to reduce the overhead of function calls. Different GenOps choose different forms of VEleFuns based on the data layout and

the shape of the input matrices.

Some GenOps invoke VEleFuns on the elements of matrices efficiently regardless of data layout and matrix shape. For example, *fm.sapply* and *fm.mapply* only require the input matrices and the output matrix to have the same data layout. For tall column-major matrices and wide row-major matrices, each CPU-level partition has long columns and long rows, respectively. These GenOps invoke a VEleFun on the long columns and rows. For tall row-major matrices and wide column-major matrices, all rows and columns in a CPU-level partition are stored in a single piece of memory. These GenOps invoke a VEleFun only once on all elements in a partition.

Most of the GenOps require a matrix with a specific data layout to reduce function call overhead. Many of the GenOps favor the column-major order for a tall-and-skinny matrix and the row-major order for a short-and-wide matrix. These data layouts increase the length of a vector passed to a VEleFun and align data in memory. For example, the column-major order ensures that each column in a partition of a tall matrix is aligned in memory, regardless of the number of columns in the matrix. A GenOp, such as inner product, converts the data layout of a CPU-level partition to the preferred layout if an input matrix does not have the preferred layout.

Given a matrix with the preferred data layout, a GenOp selects different forms of a VEleFun automatically based on the shape of the input matrix. For example, for a tall column-major matrix, *fm.mapply.col* invokes the *bVEleFun1* form of the binary VEleFun on a column from the input matrix and the input vector; for a wide

row-major matrix, *fm.mapply.col* invokes the *bVEleFun2* form on a row from the input matrix and an element from the vector. We apply the similar strategy to other GenOps. When applying *inner product* on a tall column-major matrix, FlashMatrix uses the *bVEleFun2* form of the first VEleFun to computes the outer product of a column from the left matrix and a row from the right matrix, and uses the *aVEleFun2* of the second VEleFun to compute the final result. Because *inner product* operates on a CPU-level partition, all intermediate results in the computation reside in CPU cache. Inner product on a wide matrix and a tall matrix invokes the *bVEleFun1* form of the first VEleFun on a row from the left matrix and a column from the right matrix, and invokes the *aVEleFun1* form of the second VEleFun on the output from the first VEleFun to compute an element in the output matrix for the input partitions.

#### 5.3.3.4 Implementation of GenOps on a group of matrices

When applying a GenOp on a group of matrices (Section 5.3.1.3), we decompose the computation into multiple GenOps and apply them to individual matrices in the group if the GenOp supports decomposition. Decomposing computation to individual matrices reduces memory copies and increases CPU cache hits. For the GenOps that cannot be decomposed, we combine the individual matrices on the fly and apply the GenOps on the combined matrix directly.

We apply some of the GenOps to individual matrices directly without transformation. For example, *fm.sapply* and *fm.agg* run on individual matrices directly regard-

less of the shape and data layout of the matrices. Other GenOps may be applied to individual matrices directly if the input matrices have certain shape. For example, we apply *fm.mapply.col* and *fm.agg.col* to individual matrices in a group of tall matrices directly. Similarly, we apply *fm.mapply.row* and *fm.agg.row* to individual matrices in a group of wide matrices directly.

Applying other GenOps to a group of matrices requires transformation. If an aggregation VEleFun provides a *combine* function, applying *fm.agg.row* to a group of tall matrices is transformed into two steps: apply the *aggregate* function on each row of individual matrices and apply the *combine* function on the partial aggregation results. When applying *fm.mapply.row* to a group of tall matrices, we break the input vector into parts to match the number of columns in the individual matrices in the group and apply *fm.mapply.row* to individual matrices separately. We apply the same strategies to *fm.agg.col* and *fm.mapply.col* on a group of wide matrices.

### 5.3.4 Reduce data movement in memory hierarchy

Although GenOps coupled with VEleFuns achieves efficiency in a single matrix operation, they alone cannot achieve the overall performance of users' code, especially when matrices are stored on SSDs. Evaluation of individual matrix operations results in significant data movement between CPU and SSDs. As such, we deploy lazy evaluation to construct a directed acyclic graph (DAG); when evaluating computation in a DAG, we take advantage of two-level partitioning on matrices to reuse data in

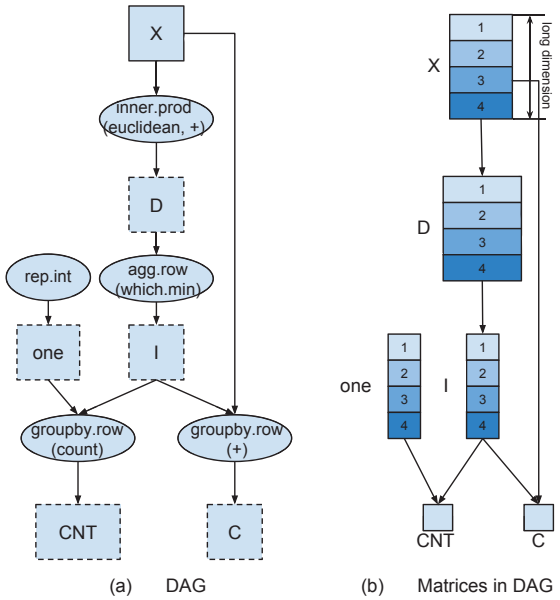



**Figure 5.4:** A directed acyclic graph of computing an iteration of k-means show in Figure 5.3.

memory and CPU cache.

#### 5.3.4.1 Lazy evaluation

FlashMatrix allows to evaluate many matrix operations lazily, which includes all GenOps in Table 5.1 and some utility functions in Table 5.2. The lazily evaluated matrices are put together to construct a DAG to represent the computation. To reduce data movement, the goal is to grow a DAG as large as possible to increase the ratio of computation and I/O in a DAG.

Figure 5.4 (a) shows a DAG for the R code of k-means in Figure 5.3. A DAG comprises a set of matrix nodes (shown as rectangles) and computation nodes (shown as ellipses). Majority of matrix nodes represent *virtual matrices*, shown as dashed

line rectangles, which only contains the corresponding matrix operations and input matrices. In the case of k-means, only the input matrix $X$ contains materialized data. A computation node references to a matrix operation and input matrices and may contain some immutable computation state, such as scalar variables and small matrices involved in the matrix computation.

*Virtual matrices* in a DAG do not need to have the same shape (Figure 5.4 (b)). All *virtual matrices* in the internal matrix nodes need to have the same *long dimension* to simplify evaluation and data flow in matrix materialization. The other dimension of the internal matrices can vary. The GenOps that generate matrices with the same *long dimension* as the input matrices include *fm.sapply* and *fm.mapply*. In addition, FlashMatrix allows *virtual matrices* of different sizes in both dimensions, such as matrices $CNT$ and $C$ (Figure 5.4 (b)), in a DAG. These *virtual matrices* usually form the edge node of a DAG because any computation that uses these *virtual matrices* cannot be connected to the same DAG. We refer to them as *sink matrices*. The GenOps that generate *sink matrices* include *fm.agg* and the variants of *groupby*.

To enable lazy evaluation, all matrices in FlashMatrix are immutable and every matrix operation generates a new matrix. As such, materialization of *virtual matrices* always generates the same result. FlashMatrix garbage collects a matrix when there are no references to it.

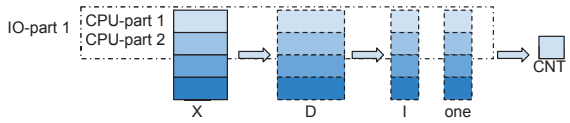



**Figure 5.5:** Materialization of partitions of matrices in a DAG.

#### 5.3.4.2 Matrix materialization in the memory hierarchy

Lazy evaluation postpones computation in matrix operations, but we eventually have to materialize some *virtual matrices* to perform actual computation. When we perform computation in a DAG, FlashMatrix only reads the input matrices of the DAG from SSDs and materializes other matrices on the fly. FlashMatrix utilizes the two-level partitioning in dense matrices to bring data from SSDs to CPU efficiently.

FlashMatrix allows users to materialize any *virtual matrix* in a DAG. By default, FlashMatrix materializes only *sink matrices* in a DAG to minimize data written to SSDs because *sink matrices* are small and are always kept in memory. In the example of k-means (Figure 5.4), we materialize the two *sink matrices* together. In some cases, especially in iterative algorithms, we need to materialize some non-*sink matrices* in a DAG to avoid redundant computation and I/O across iterations. In the current implementation, FlashMatrix allows users to set a flag on the non-*sink matrices* to inform FlashMatrix to save materialized data of these matrices to memory or SSDs during computation.

FlashMatrix partitions matrices in a DAG in the *long dimension* and materializes partitions separately (Figure 5.5). This is realized because all *virtual matrices* except *sink matrices* in a DAG share the same *long dimension* size and partition size. As

such, a partition $i$ of a *virtual matrix* only requires data from partitions $i$ of the parent matrices. When materializing a *sink matrix*, each thread first computes partial aggregation results independently on the partitions of the parent matrix assigned to the thread. In the end, FlashMatrix merges per-thread partial aggregation results to construct the *sink matrix*.

FlashMatrix takes advantage of the two-level partitioning on dense matrices to reduce data movement between SSDs and CPU. It assigns I/O-level partitions to a thread as computation tasks for parallelization. We choose a relatively small partition size to balance the overhead of accessing a partition, computation skew and memory consumption. A thread further splits an I/O-level partition into CPU-level partitions at run time and materializes one CPU-level partition at a time. Materialization of a CPU-level partition is triggered recursively. As shown in Figure 5.5, materializing matrix $CNT$ triggers materialization of CPU-level partitions of matrices $I$ and *one*, which in turn triggers materialization of partitions of matrix $D$, and so on. Eventually, it triggers data access to an I/O-level partition of input matrix $X$ from SSDs. After materializing a CPU-level partition, the thread passes it to the subsequent operation, instead of materializing the next CPU-level partition in the same matrix. A CPU-level partition is sufficiently small to fit in the CPU cache so that the partition still resides in the CPU cache when the subsequent operation consumes it. This significantly reduces data movement between CPU and memory. In each thread, all intermediate matrices have only one CPU-level partitions materialized at any time to reduce CPU

cache pollution and thus increase CPU cache hits.

## 5.4 Experimental evaluation

We evaluate the performance of FlashMatrix with statistics and machine learning algorithms both in memory and on SSDs. We compare their performance with the implementations in Spark MLlib,[29] a highly-optimized parallel machine learning library, and the C and FORTRAN implementations in the R framework. We further illustrate the effectiveness of the optimizations deployed in FlashMatrix when running both in memory and on SSDs.

We conduct experiments on a non-uniform memory architecture machine with four Intel Xeon E7-4860 processors, clocked at 2.6 GHz, and 1TB memory of DDR3-1600. Each processor has 12 cores. The machine has three LSI SAS 9300-8e host bus adapters (HBA) connected to a SuperMicro storage chassis, in which 24 OCZ Intrepid 3000 SSDs are installed. The 24 SSDs together are capable of delivering 12 GB/s for read and 10 GB/s for write at maximum. The machine runs Linux kernel v3.13.0. By default, we use 48 threads for both in-memory and out-of-core execution of FlashMatrix. We use Spark v1.5.0 and R v3.2.4.

### 5.4.1 Statistics and Machine learning algorithms

We implement multiple important algorithms in the field of statistics and machine learning. We implement these algorithms completely with the R interface of FlashMatrix and rely on FlashMatrix to perform computation in parallel and out of core.

- Multivariate statistical summary: this computes column-wise minimum, maximum, mean, L1 norm, L2 norm, the number of non-zero values and variance on a data matrix.

- Correlation: this computes pair-wise Pearson's correlation[143] among multiple series of data and is commonly used in statistics.

- Singular value decomposition (SVD) factorizes a matrix into three matrices: $U$, $\Sigma$ and $V$ such that $A = U\Sigma V^T$, where $U$ and $V$ are orthonormal matrices and $\Sigma$ is a diagonal matrix with non-negative diagonals in descending order. To compute SVD on a $n \times p$ matrix $A$ ($n \gg p$), we first compute Gramian matrix $A^T A$ and compute eigenvalues and eigenvectors to derive singular values and singular vectors of the matrix $A$. SVD is commonly used for dimension reduction. In the experiments, we compute 10 singular values.

- K-means[27] is an iterative algorithm of partitioning a set of data points into $k$ clusters so that each cluster has minimal mean of distances between the data points in the cluster and the cluster center. K-means is one of the most

| Algorithm | Computation | I/O |
|---|---|---|
| Summary | $O(n \times p)$ | $O(n \times p)$ |
| Correlation | $O(n \times p^2)$ | $O(n \times p)$ |
| SVD | $O(n \times p^2)$ | $O(n \times p)$ |
| K-means (1 iteration) | $O(n \times p \times k)$ | $O(n \times p)$ |
| GMM (1 iteration) | $O(n \times p^2 \times k + p^3 \times k)$ | $O(n \times p + n \times k)$ |

**Table 5.4:** The computation and I/O complexity of the algorithms for the five algorithms. $n$ is the number of data points in the dataset, $p$ is the number of the features in a data point and $k$ is the number of clusters k-means and GMM partition the dataset. We assume $n \gg p$.

popular clustering algorithms and is identified as one of the top 10 data mining algorithms.[110] In the experiments, we run k-means to split a dataset into 10 clusters by default.

- Gaussian Mixture Model (GMM)[134] is another iterative clustering algorithm that assumes data points are sampled from a mixture of Gaussian distributions and use expectation maximization (EM)[134] algorithm to fit the model. This algorithm is also identified as one of the top 10 data mining algorithms.[110] In the experiments, we run GMM to split a dataset into 10 clusters by default.

These algorithms have various ratios of computation complexity and I/O complexity (Table 5.4), which helps to evaluate the perforamnce of FlashMatrix on SSDs thoroughly. The first three algorithms only require a constant number of passes over the input matrix. K-means and GMM run iteratively and we show their computation and I/O complexity in a single iteration. GMM typically run on a dataset with a small number of features. Therefore, the first term of its computation complexity dominates the computation. Although correlation and SVD have lower asymptotic

| Data Matrix | n | p | size |
|---|---|---|---|
| Friendster-32[109] | 65M | 32 | 16GB |
| MixGaussian-1B | 1B | 32 | 251GB |
| Random-65M | 65M | 8-512 | 4-248GB |

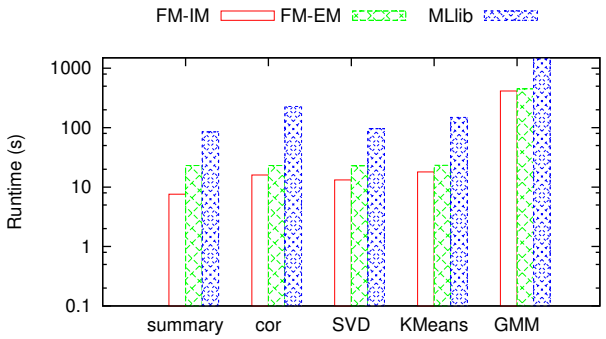
**Table 5.5:** Datasets ($n \times p$ matrices) for performance evaluation.

complexity than GMM, they may run on datasets with many features and, thus, may have very high computation overhead.

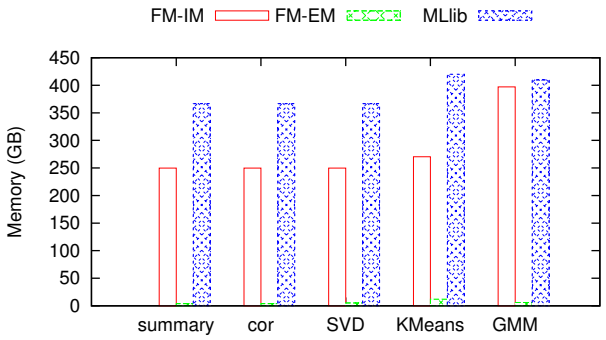
K-means and GMM typically run on a dataset with a small number of features in each data point due to curse of dimensionality[142] while the other algorithms may be applied to datasets with various numbers of features. We use the datasets in Table 5.5 for performance evaluation. In all datasets, the number of data points is far more than the number of features. We run k-means and GMM on the Friendster-32 matrix, constructed from 32 eigenvectors of the Friendster graph,[109] as well as the MixGaussian-1B matrix with one billion data points and 32 features in each data point, sampled from 10 mixtures of multivariate Gaussian distributions with the identity covariance matrix and different means. We measure the performance of the other three algorithms on all of the matrices in Table 5.5, including the random matrices with 65 million rows and the number of columns varying from 8 to 512.

### 5.4.2 Comparative performance of FlashMatrix

We compare the performance of the FlashMatrix implementations with the ones in Spark MLlib[29] and the R framework. We run the MLlib implementations with




**(a)** Runtime




**(b)** Memory consumption

**Figure 5.6:** The performance and memory consumption of FlashMatrix both in memory (FM-IM) and on SSDs (FM-EM) compared with Spark MLlib on the MixGaussian-1B matrix.

their native Scala interface and use a very large heap size to ensure that all input data is cached in memory. We use 48 threads for both FlashMatrix and MLlib to run on the MixGaussian-1B matrix. The R framework provides C implementations for correlation, SVD and k-means. The R package mclust[144] provides a FORTRAN implementation of GMM. These implementations run in a single thread. We run the FlashMatrix implementations in a single thread and compare their performance with the C and FORTRAN implementations on the Friendster-32 matrix.

FlashMatrix both in memory and on SSDs outperforms Spark MLlib significantly in all algorithms (Figure 5.6 (a)). For some algorithms such as correlation, SVD and GMM, even though both FlashMatrix and MLlib implementations heavily rely on BLAS for matrix multiplication, FlashMatrix outperforms MLlib significantly owing to our heavy optimizations on GenOps such as aggressive matrix operation fusion and VEleFuns. In contrast, MLlib materializes operations such as aggregation separately and implements non-BLAS operations with Scala.

Even though FlashMatrix provides a matrix-oriented functional programming interface, it easily scales to datasets with billions of data points and its scalability is bound by the capacity of SSDs (Figure 5.6 (b)). For out-of-core execution, FlashMatrix keeps large matrices on SSDs and has a very small memory footprint. The functional programming interface generates a new matrix in each matrix operation, which potentially leads to high memory consumption. Owing to lazy evaluation, FlashMatrix does not store majority of matrices in the computation physically. As such, its in-memory execution barely increases memory consumption from the minimum memory requirement of the algorithms. This indicates that the out-of-core execution consumes small space on SSDs, which leads to very high scalability.

FlashMatrix running both in memory and on SSDs significantly outperforms R even with a single thread in all of these algorithms (Figure 5.7). We exclude statistic summary in the experiment because R does not provide a C or FORTRAN implementation of computing the same statistics. The performance results indicate that

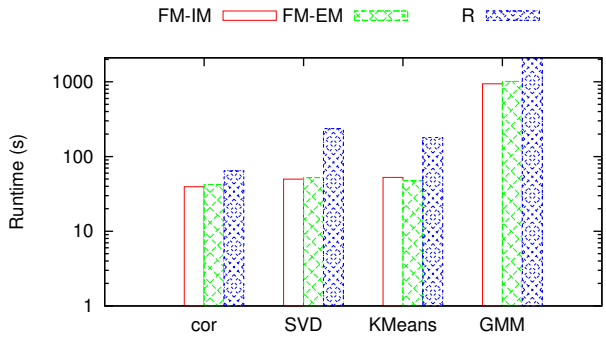



**Figure 5.7:** The performance of FlashMatrix in a single thread both in memory (FM-IM) and on SSDs (FM-EM) compared with the C and FORTRAN implementations in the R framework on the Friendster-32 matrix.

FlashMatrix executes R code efficiently to even outperform some optimized C and FORTRAN implementations when processing large datasets.

The in-memory execution of FlashMatrix achieves almost linear speedup in all algorithms while the out-of-core execution only starts to flatten out after 32 threads (Figure 5.8). Owing to operation fusion in CPU cache, FlashMatrix significantly reduces data movement between CPU and main memory. As such, memory bandwidth is no longer the bottleneck for in-memory execution and the algorithms speed up linearly with more CPU cores. For out-of-core execution, I/O is the bottleneck for the algorithms with lower computation complexity when they run with 48 threads. However, GMM still speeds up almost linearly even when running on SSDs, due to its high computation complexity. The performance results in Figure 5.7 and Figure 5.8 indicate that FlashMatrix can potentially execute R code with performance comparable to parallel C or FORTRAN implementations.

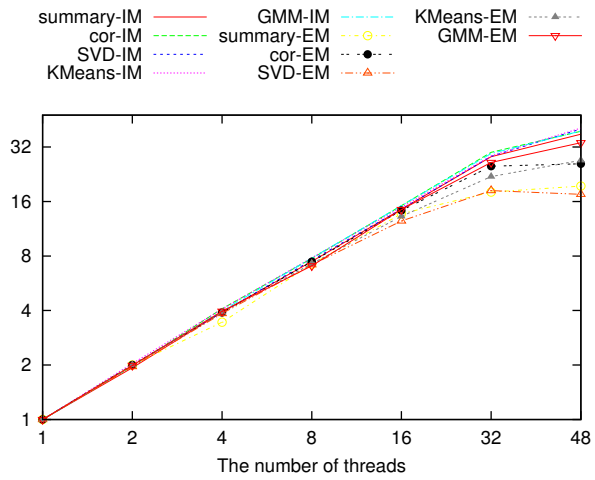



**Figure 5.8:** The speedup of FlashMatrix with multithreading both in memory (IM) and on SSDs (EM).

### 5.4.3 Performance of FlashMatrix in memory and on SSDs

We further measure the in-memory and external-memory performance of FlashMatrix thoroughly with different datasets and different parameters. We run the first three algorithms on random-65M matrices with the number of columns varying from 8 to 512. We run k-means and GMM on the Friendster-32 matrix and vary the number of clusters from 2 to 64.

As the number of features or the number of clusters increases, the performance gap between in-memory and external-memory execution narrows and eventually the external-memory performance gets almost 100% of in-memory performance for some algorithms (Figure 5.9 and 5.10). This observation conforms with the computation and I/O complexity of the algorithms in Table 5.4. When the number of features gets

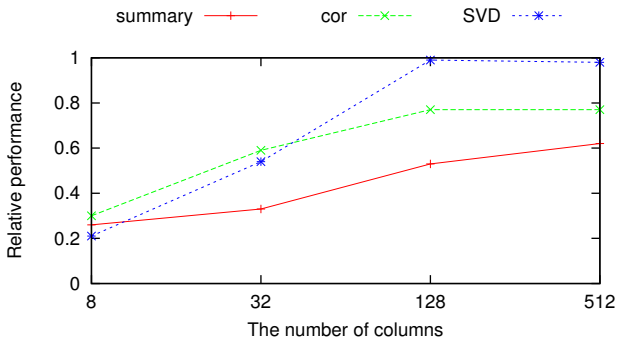



**Figure 5.9:** The relative performance of FlashMatrix on SSDs for statistics computation on random-65M matrices with the number of columns varying from 8 to 512, normalized by its performance in memory. As the number of columns increases, the external-memory performance of these implementations approach to its in-memory performance.

larger, the computation of matrix multiplication in correlation and SVD grows more rapidly than I/O and eventually CPU becomes the bottleneck. The current implementation of correlation requires an additional pass on the input matrix to compute column-wise mean values, which results in lower external-memory performance. Similarly, as the number of clusters increases, the computation of k-means and GMM increases rapidly and these algorithms are dominated by their CPU computation as the number of clusters gets larger. Given an I/O throughput of 10 GB/s, the algorithms do not require many features or clusters to have their external-memory performance close to their in-memory performance.

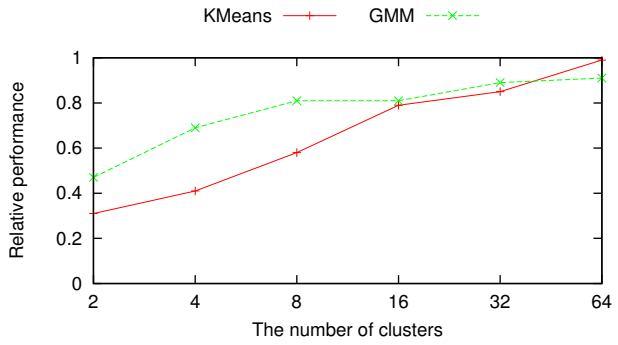



**Figure 5.10:** The relative performance of FlashMatrix on SSDs for clustering algorithms with different numbers of clusters, normalized by its performance in memory. As the number of clusters increases, the external-memory performance of these implementations approach to their in-memory performance.

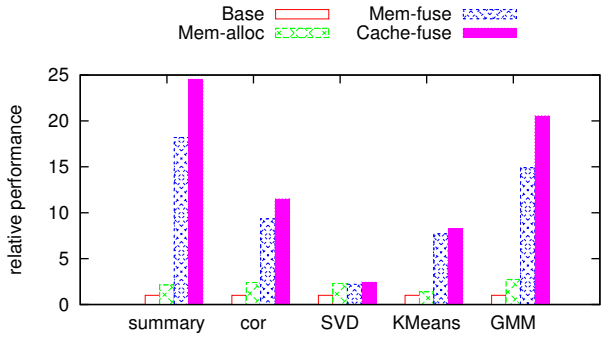
### 5.4.4 Effectiveness of optimizations

In this section, we illustrate the effectiveness of our memory and CPU optimizations in FlashMatrix. To reduce memory overhead, we focus on three main optimizations: *(i)* reusing memory chunks for new in-memory matrices and I/O access to reduce large memory allocation (mem-alloc), *(ii)* matrix operation fusion in main memory to reduce data movement between SSDs and main memory (mem-fuse), *(iii)* matrix operation fusion in CPU cache to reduce data movement between main memory and CPU cache (cache-fuse). To reduce computation overhead, we illustrate the effectiveness of using VEleFuns.

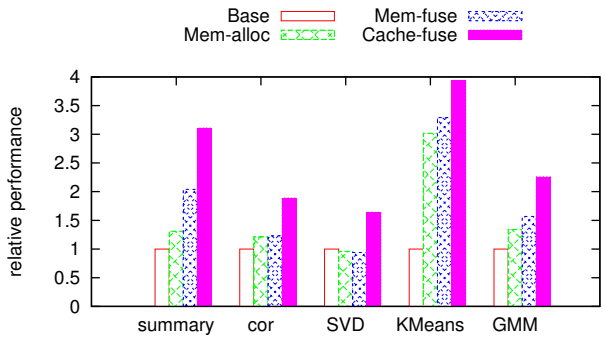
Each memory optimization has significant performance improvement on most of the algorithms when FlashMatrix runs on SSDs (Figure 5.11 (a)). Operation fusion in main memory achieves the highest performance improvement in almost all algorithms, even in GMM, which has the highest asymptotic computation complexity. Even

though the SSDs deliver an I/O throughput of 10GB/s, materializing every matrix operation separately causes SSDs to be the main bottleneck in the system. Fusing matrix operations in memory significantly reduces I/O access to SSDs and improves performance by a large factor. Operation fusion in the CPU cache also has very positive performance impact on some algorithms even when the algorithms run on SSDs. This suggests that with sufficient I/O optimizations, many machine learning algorithms that run on fast SSDs can be bottlenecked by the bandwidth of main memory, instead of I/O. Even though it is less noticeable, reducing large memory allocation improves I/O performance and almost doubles the overall performance of all algorithms.

The same memory optimizations also have very positive impact on the performance of most of the algorithms when FlashMatrix runs in memory (Figure 5.11 (b)). Operation fusion in CPU cache has performance improvement on all algorithms because reducing data movement between main memory and CPU cache in a sequence of matrix operations always improves performance. Operation fusion in main memory further reduces memory allocation overhead when FlashMatrix runs in memory. Thus, like the optimization of reducing large memory allocation, its effectiveness heavily depend on the computation patterns in the algorithms. Both of the optimizations are more effective for the algorithms such as statistical summary and k-means, which require to generate new large matrices but do not have very heavy computation.




**(a)** on SSDs




**(b)** in memory

**Figure 5.11:** The effectiveness of memory optimizations on different algorithms running on SSDs. The three memory optimizations are applied to FlashMatrix incrementally.

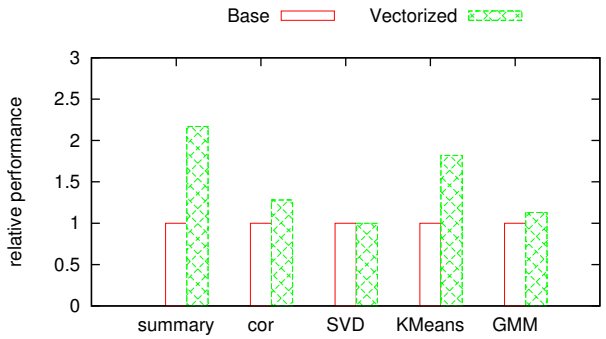



**Figure 5.12:** The effectiveness of using VEleFuns on different algorithms running in memory.

VEleFuns improve the performance of the algorithms that rely on GenOps for computation (Figure 5.12). In this experiment, the base implementations deploy all memory optimizations to avoid memory from being the bottleneck of the system, and invoke functions on individual elements. The FlashMatrix implementations of statistical summary and k-means solely rely on GenOps. Therefore, their performance is almost doubled when we use VEleFuns. The main computation in correlation and GMM is matrix multiplication, but they still rely on GenOps for the remaining computation. As such, the use of VEleFuns helps their performance. SVD solely uses matrix multiplication, so switching to VEleFuns has no performance impact.

## 5.5 Conclusions

We present FlashMatrix, a matrix-oriented programming framework for general data analysis. FlashMatrix scales to large datasets by utilizing commodity SSDs. It

provides a high-level functional programming interface for users to write data analysis algorithms in R and executes the R implementations in parallel and out of core automatically. For simplicity and generality, the core of FlashMatrix only implements a small number of generalized matrix operators (GenOps). It reimplements many matrix operations in R *base* package with GenOps to provide a familiar programming environment to users. To improve performance, FlashMatrix uses vectorized element functions (VEleFuns) to reduce the overhead of function calls and fuses matrix operations to reduce data movement between CPU and SSDs.

We demonstrate that the matrix-oriented functional programming interface in FlashMatrix can achieve high performance and scalability for many data analysis algorithms. We implement multiple statistics and machine learning algorithms in R and compare their performance with Spark MLlib, a highly-optimized parallel machine learning library, on large datasets. The R implementations executed in FlashMatrix significantly outperforms the implementations in Spark MLlib. We further demonstrate that the R implementations running FlashMatrix with a single thread can outperform the C and FORTRAN implementations in the R framework. In addition, FlashMatrix also achieves linear speedup with multithreading in all of these algorithms.

Even though SSDs are still an order of magnitude slower than DRAM, the external-memory execution of many data analysis algorithms in FlashMatrix can achieve performance comparable to their in-memory execution. We demonstrate that an I/O

throughput of 10 GB/s is able to saturate CPU for many algorithms, even in a large parallel NUMA machine. As such, the external-memory execution also benefits from many in-memory optimizations.

FlashMatrix simplifies significantly the programming effort of writing parallel and out-of-core implementations for large-scale data analysis. It provides domain experts a familiar programming environment for implementing their algorithms designed to process large datasets. It also significantly increases productivity of writing an efficient implementation with performance comparable to low-level programming languages. We believe FlashMatrix opens a new opportunity for large-scale data analysis.

# Chapter 6

# Conclusion

This thesis addresses the challenges in large-scale data analysis using commodity SSDs. Instead of developing data analysis algorithms on SSDs directly, we develop programming frameworks for users to implement complex data analysis algorithms and hides the complexity of external-memory data analysis and parallel computation.

One of the main contributions of this thesis is developing a comprehensive data analysis ecosystem called FlashX, which covers a large range of data analysis tasks. FlashX contains three major subsystems: SAFS, FlashGraph and FlashMatrix. By seaminglessly integrating these subsystems together, FlashX achieves efficiency, scalability and generality.

Over the course of studying data analysis tasks on SSDs, we also conclude that the semi-external memory strategy is a simple but effective way of achieving both performance and scalability. We adopt semi-external memory in FlashGraph for

graph analysis and in FlashMatrix for sparse matrix multiplication. In FlashGraph, we keep vertex state in memory and edge lists of graphs on SSDs. This partitioning enables in-memory vertex communication that generates majority of small random memory accesses in graph analysis. Thus, we achieve in-memory performance while scaling beyond memory capacity. We further extend the concept of semi-external memory to sparse matrix dense matrix multiplication, where we keep the sparse matrix on SSDs and the dense matrix or some columns of the dense matrix in memory. This strategy enables us to achieve in-memory performance while performing sparse matrix multiplication on a massive sparse matrix.

By implementing the efficient data analysis ecosystem, we are able to thoroughly study the role of flash memory in varieties of data analysis tasks at a large scale. We demonstrate that many graph analysis and machine learning tasks benefit from flash memory. With careful design and engineering in the data analysis framework and programming efforts from users, external-memory implementations of these data analysis tasks achieve performance comparable to that of state-of-the-art in-memory implementations, while scaling to very large datasets. This indicates that fast SSDs can replace RAM in data analysis, which potentially shapes the design of future machines for large-scale data analysis.

# Bibliography

[1] W. Litwin, "Linear hashing: A new tool for file and table addressing," in *Sixth International Conference on Very Large Data Bases*, 1980.

[2] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–11.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *Proceedings of the Seventh International Conference on World Wide Web 7*, 1998.

[4] R. Chen, X. Weng, B. He, M. Yang, B. Choi, and X. Li, "On the efficiency and programmability of large graph processing in the cloud," Microsoft Research, Tech. Rep., 2010.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of*

*the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004.

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012.

[9] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013.

[10] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework," in *11th*

*USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[11] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?" in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[12] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system IOPS on low-cost, commodity hardware," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[13] ——, "A parallel page cache: IOPS and caching for multicore systems," in *USENIX conference on Hot Topics in Storage and File Systems*, 2012.

[14] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "Flashgraph: Processing billion-node graphs on an array of commodity ssds," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[15] D. Zheng, D. Mhembere, V. Lyzinski, J. Vogelstein, C. E. Priebe, and R. Burns, "Semi-external memory sparse matrix multiplication on billion-node graphs in a multicore architecture," *CoRR*, vol. abs/1602.02864, 2016.

[16] D. Zheng, D. Mhembere, J. T. Vogelstein, C. E. Priebe, and R. Burns, "Flashmatrix: Parallel, scalable data analysis with generalized matrix operations using commodity ssds," *arXiv preprint arXiv:1604.06414*, 2016.

[17] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo,

J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo, "Standards for graph algorithm primitives," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, Sept 2013.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, 2010.

[19] A. Kyrola, G. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[20] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A peta-scale graph mining system implementation and observations," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, 2009.

[22] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: Discoveries and implementation," in *Proceedings of the 15th Pacific-*

*Asia Conference on Advances in Knowledge Discovery and Data Mining - Volume Part II*, 2011, pp. 13–25.

[23] V. D. Blondel, J. loup Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[24] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," University of Minnesota, Department of Computer Science, Tech. Rep., 1995.

[25] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, 2001.

[26] J. Abello, A. L. Buchsbaum, and J. R. Westbrook, "A functional approach to external graph algorithms," in *Algorithmica*. Springer-Verlag, 1998, pp. 332–343.

[27] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theor.*, vol. 28, no. 2, Sep. 2006.

[28] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[29] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in Apache Spark," *CoRR*, vol. abs/1505.06807, 2015.

[30] "Apache hadoop," http://hadoop.apache.org/, Accessed 6/10/2016.

[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2008.

[32] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "FlumeJava: Easy, efficient data-parallel pipelines," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2010.

[33] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008.

[34] X. Zhu and Z. Ghahramani, "Learning from labeled and unlabeled data with label propagation," Carnegie Mellon University, Tech. Rep., 2002.

[35] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng, "Map-Reduce for machine learning on multicore," in *Advances in Neural Information Processing Systems 19*, 2007.

[36] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan, "SystemML: Declarative machine learning on mapreduce," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2011.

[37] "Apache giraph," https://giraph.apache.org/, Accessed 4/9/2014.

[38] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, 2012.

[39] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013.

[40] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc," in *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2013, pp. 77–85.

[41] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*. Society for Industrial & Applied Mathematics, 2011.

[42] A. Lugowski, D. Alber, A. Bulu, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis, "A flexible open-source toolbox for scalable complex graph analysis," in *Proceedings of the 2012 SIAM International Conference on Data Mining*, 2012.

[43] "H2O," http://www.h2o.ai/, Accessed 5/19/2016.

[44] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, "Optiml: an implicitly parallel domainspecific language for machine learning," in *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*, 2011.

[45] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming*, 2011.

[46] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015.

BIBLIOGRAPHY

[47] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[48] AmazonWebServices, "Amazon dynamodb overview, a fully managed nosql database service," Available at http://www.youtube.com/watch?v=oz-7wJJ9HZ0, 2011.

[49] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[50] B. Hendrickson, "Data analytics and high performance computing: When worlds collide," in *Los Alamos Computer Science Symposium*, 2009.

[51] A. Foong, B. Veal, and F. Hady, "Towards SSD-ready enterprise platforms," in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2010.

[52] "Fusion-IO ioDrive Octal," http://www.fusionio.com/platforms/iodrive-octal/, Accessed 3/11/2013.

[53] "OCZ VERTEX 4," http://www.ocztechnology.com/ocz-vertex-4-sata-iii-2-5-ssd.html, Accessed 3/11/2013.

[54] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a

scalable operating system for multicores," in *ACM SIGOPS Operating System Review (OSR)*, 2009.

[55] A. Baumann, P. Barham, P. E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *Symposium on Operating Systems Principles*, 2009.

[56] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *Conference on Operating systems design and implementation*, 2010.

[57] M. Yui, J. Miyazaki, S. Uemura, and H. Yamana, "Nb-GCLOCK: A non-blocking buffer management based on the generalized CLOCK," in *International Conference on Data Engineering (ICDE)*, 2010.

[58] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Transactions on Database Systems*, vol. 3, no. 3, 1978.

[59] P. McKenney, D. Sarma, A. Arcangeli, A. Kleen, and O. Krieger, "Read-copy update," in *Linux Symposium*, 2002.

[60] S. Park and K. Shen, "A performance evaluation of scientific I/O workloads on flash-based SSDs," in *IEEE International Conference on Cluster Computing and Workshops.*, 2009.

[61] S. Sen, S. Chatterjee, and N. Dumir, "Towards a theory of cache-efficient algorithms," *Journal of the ACM*, vol. 49, no. 6, 2002.

[62] S. Park and K. Shen, "FIOS: A fair and efficient I/O scheduler," in *Conference on File and Storage Technology*, 2012.

[63] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "PLFS: A checkpoint filesystem for parallel applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[64] "MSI-HOWTO," http://lwn.net/Articles/44139/, Accessed 3/6/2012.

[65] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," in *ACM SIGMOD international conference on Management of data*, 1988.

[66] F. J. Corbato, "A paging experiment with the multics system," 1969.

[67] L. Bouganim, B. Jónssonn, and P. Bonnet, "uFLIP: Understanding flash IO patterns," in *Fourth Biennial Conference on Innovative Data Systems Research*, 2009.

[68] A. Batsakis, R. Burns, A. Kanevsky, J. Lentini, and T. Talpey, "AWOL: An adaptive write optimizations layer," in *Conference on File and Storage Technologies*, 2008.

[69] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, New York, NY, USA, 2006.

[70] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman, "NUMA-aware algorithms: the case of data shuffling," in *The biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[71] H. Shan, K. Antypas, and J. Shalf, "Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, Austin, TX, 2008.

[72] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[73] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Symposium on Cloud computing*, 2010.

[74] "Neo4j," www.neo4j.org, Accessed 3/11/2013.

[75] "Livejournal social network," http://snap.stanford.edu/data/soc-LiveJournal1.html, Accessed 3/11/2013.

[76] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, 2005.

[77] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *USENIX Annual Technical Conference*, 2005.

[78] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[79] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2013.

[80] J. N. Bertsekas, Dimitri P.; Tsitsiklis, in *Parallel and Distributed Computation:Numerical Methods*. PrenticeHall, Inc., 1989.

[81] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, "The design and implementation of modern column-oriented database systems," *Foundations and Trends in Databases*, vol. 5, pp. 197–280, 2013.

[82] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proceedings of the 15th*

*ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[83] U. Kang and C. Faloutsos, "Beyond 'caveman communities': Hubs and spokes for graph compression and mining," in *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, 2011.

[84] H. Wang, M. Tang, Y. Park, and C. E. Priebe, "Locality statistics for anomaly detection in time series of graphs," *IEEE Transactions on Signal Processing*, vol. 62, no. 3, 2014.

[85] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, pp. 163–177, 2001.

[86] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation," *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[87] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 440-442, 1998.

[88] H. Wang, D. Zheng, R. Burns, and C. Priebe, "Active community detection in massive graphs," *CoRR*, vol. abs/1412.8576, 2015.

[89] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or

a news media?" in *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[90] "Web graph," http://webdatacommons.org/hyperlinkgraph/, Accessed 4/18/2014.

[91] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012.

[92] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis, "Efficient semi-streaming algorithms for local triangle counting in massive graphs," in *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2008.

[93] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *Journal of the Society for Industrial and Applied Mathematics Series B Numerical Analysis*, vol. 2, no. 2, pp. 205–224, 1965.

[94] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in Neural Information Processing Systems 13*, 2001.

[95] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, 2007.

[96] A. Yoo, A. H. Baker, R. Pearce, and V. E. Henson, "A scalable eigensolver for large scale-free graphs using 2D graph partitioning," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[97] E. G. Boman, K. D. Devine, and S. Rajamanickam, "Scalable matrix computations on large scale-free graphs using 2D graph partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[98] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright, "Abstract machine models and proxy architectures for exascale computing," in *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, ser. Co-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014.

[99] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[100] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro, "A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned

iterative methods," *International Journal for Numerical Methods in Engineering*, 2005.

[101] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, 2005.

[102] "Intel math kernel library," https://software.intel.com/en-us/intel-mkl, Accessed 1/24/2016.

[103] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014.

[104] E.-J. Im, K. Yelick, and R. Vuduc, "Sparsity: Optimization framework for sparse matrix kernels," *Int. J. High Perform. Comput. Appl.*, 2004.

[105] Z. Zhou, E. Saule, H. Aktulga, C. Yang, E. Ng, P. Maris, J. Vary, and U. Catalyurek, "An out-of-core eigensolver on SSD-equipped clusters," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012.

[106] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro, "A compari-

son of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods," *International Journal for Numerical Methods in Engineering*, 2005.

[107] G. W. Stewart, "A KrylovSchur algorithm for large eigenproblems," *SIAM Journal on Matrix Analysis and Applications*, 2002.

[108] C. Lomont, "Introduction to intel advanced vector extensions," https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions, 2011.

[109] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," in *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, 2012.

[110] X. Wu, V. Kumar, J. Ross Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. McLachlan, A. Ng, B. Liu, P. Yu, Z.-H. Zhou, M. Steinbach, D. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, 2008.

[111] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Exploring artificial intelligence in the new millennium." San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, ch. Understanding Belief Propagation and Its Generalizations.

[112] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of

linear differential and integral operators," *Journal of Research of the National Bureau of Standards*, 1950.

[113] D. Calvetti, L. Reichel, and D. C. Sorensen, "An implicitly restarted lanczos method for large symmetric eigenvalue problems," *ETNA*, vol. 2, pp. 1–21, 1994.

[114] R. Lehoucq, D. Sorensen, and C. Yang, *ARPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1998.

[115] V. Hernandez, J. E. Roman, and V. Vidal, "SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems," *ACM Trans. Math. Softw.*, 2005.

[116] D. Zheng, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "An SSD-based eigensolver for spectral analysis on billion-node graphs," *CoRR*, vol. abs/1602.01421, 2016.

[117] X. Su and T. M. Khoshgoftaar, "A survey of collaborative filtering techniques," *Adv. in Artif. Intell.*, vol. 2009, pp. 4:2–4:2, Jan. 2009.

[118] J. Yang and J. Leskovec, "Overlapping community detection at scale: A non-negative matrix factorization approach," in *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, ser. WSDM '13. New York, NY, USA: ACM, 2013, pp. 587–596.

[119] F. Wang, T. Li, X. Wang, S. Zhu, and C. Ding, "Community discovery using nonnegative matrix factorization," *Data Min. Knowl. Discov.*, vol. 22, no. 3, pp. 493–521, May 2011.

[120] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004.

[121] R. Boyd, B. Drake, D. Kuang, and H. Park, "Smallk is a C++/Python high-performance software library for nonnegative matrix factorization (nmf) and hierarchical and flat clustering using the nmf; current version 1.2.0," http://smallk.github.io/, 2014.

[122] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw.*, vol. 39, no. 2, pp. 13:1–13:24, Feb. 2013.

[123] R. Liao, Y. Zhang, J. Guan, and S. Zhou, "Cloudnmf: A mapreduce implementation of nonnegative matrix factorization for large-scale biological datasets," *Genomics, Proteomics & Bioinformatics*, vol. 12, no. 1, pp. 48 – 51, 2014.

[124] J. Yin, L. Gao, and Z. Zhang, "Scalable nonnegative matrix factorization with block-wise updates," in *ECML PKDD*, 2014.

[125] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang, "Distributed nonneg-

ative matrix factorization for web-scale dyadic data analysis on mapreduce," in *Proceedings of the 19th International Conference on World Wide Web*, New York, NY, USA, 2010.

[126] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2010, pp. 231–242.

[127] B. OBAMA, "Executive order – creating a national strategic computing initiative," *The White House Office of the Press Secretary*, 2015.

[128] B. Barney, "Message passing interface (MPI)," https://computing.llnl.gov/tutorials/mpi/, 2015.

[129] "OpenMP," http://openmp.org/wp/, 2016.

[130] Q. Wang, X. Zhang, Y. Zhang, and Q. Yi, "AUGEM: Automatically generate high performance dense linear algebra kernels on x86 cpus," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, Nov 2013, pp. 1–12.

[131] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang,

"PETSc Web page," http://www.mcs.anl.gov/petsc, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[132] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *MSST/SNAPI*, 2012.

[133] M. V. Wilkes, "The memory gap and the future of high performance memories," *SIGARCH Comput. Archit. News*, vol. 29, no. 1, pp. 2–7, Mar. 2001.

[134] T. L. BAILEY and C. ELKAN, "Fitting a mixture model by expectation maximization to discover motifs in biopolymers," in *Proceedings of International Conference on Intelligent Systems for Molecular Biology*, 1994.

[135] K. Goto and R. A. v. d. Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 12:1–12:25, May 2008.

[136] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimization of software and the atlas project," *PARALLEL COMPUTING*, vol. 27, p. 2001, 2000.

[137] "Revolution R Enterprise," http://www.revolutionanalytics.com/revolution-r-enterprise, Accessed 4/9/2016.

[138] "Parallel computing toolbox in MATLAB," http://www.mathworks.com/products/parallel-computing/, Accessed 4/9/2016.

[139] S. Venkataraman, E. Bodzsar, I. Roy, A. AuYoung, and R. S. Schreiber, "Presto: Distributed machine learning and graph processing with sparse matrices," in *Proceedings of the 8th ACM European Conference on Computer Systems*. New York, NY, USA: ACM, 2013.

[140] W.-M. Ching and D. Zheng, "Automatic parallelization of array-oriented programs for a multi-core machine," *International Journal of Parallel Programming*, vol. 40, no. 5, pp. 514–531, 2012.

[141] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data parallelism to program GPUs for general-purpose uses," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2006.

[142] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: a review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, pp. 4–37, Jan 2000.

[143] K. Pearson, "Notes on regression and inheritance in the case of two parents," in *Proceedings of the Royal Society of London*, 1895, pp. 240–242.

[144] C. Fraley, A. E. Raftery, T. B. Murphy, and L. Scrucca, "mclust version 4 for R: Normal mixture modeling for model-based clustering, classification, and density estimation," *Technical Report No. 597 , Department of Statistics, University of Washington*, 2012.

# Vita

Da Zheng received the BS degree in computer science from Zhejiang University, China, in 2006, the MS degree in computer science from École polytechnique fédérale de Lausanne, in 2009. Since 2010, he is a PhD student of computer science at Johns Hopkins University. His PhD research focuses on building frameworks to scale data analysis for massive datasets using commodity SSDs.