FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay, ***Johns Hopkins University***

https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

Da Zheng, Disa Mhembere, Randal Burns
Department of Computer Science
Johns Hopkins University

Joshua Vogelstein
Institute for Computational Medicine
Department of Biomedical Engineering
Johns Hopkins University

Carey E. Priebe
Department of Applied Mathematics and Statistics
Johns Hopkins University

Alexander S. Szalay
Department of Physics and Astronomy
Johns Hopkins University

Abstract

Graph analysis performs many random reads and writes, thus, these workloads are typically performed in memory. Traditionally, analyzing large graphs requires a cluster of machines so the aggregate memory exceeds the graph size. We demonstrate that a multicore server can process graphs with billions of vertices and hundreds of billions of edges, utilizing commodity SSDs with minimal performance loss. We do so by implementing a graph-processing engine on top of a user-space SSD file system designed for high IOPS and extreme parallelism. Our semi-external memory graph engine called FlashGraph 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.

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 [10, 20], machine learning [18], and network analysis [22]. 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 [17] to localize access. While good partitions may be important for performance [8], leading systems partition natural graphs randomly [11].

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 memory [18, 11] or message passing [20, 10, 24] 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 [16, 23] 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 [7] 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 [2, 22] 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 [32] 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 [21], 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.

2 Related Work

MapReduce [9] is a general large-scale data processing framework. PEGASUS [13] 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 [7] and label propagation [33], but performs poorly for graph traversal algorithms.

Several other works [14, 19] 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 [20] 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 [10] 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 [18] and PowerGraph [11] 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 [24] optimizes message passing by restricting vertex communication to a vertex and its direct neighbors.

Ligra [25] 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 [21] 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 [16] and X-stream [23] 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 [12] 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. [2] introduced the semi-external memory algorithmic framework for graphs. Pearce et al. [22] implemented several semi-external memory graph traversal algorithms for SSDs. FlashGraph adopts and advances several concepts introduced by these works.

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) [32], 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.

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.

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.

3.1 SAFS

SAFS [32] 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 [31]. 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

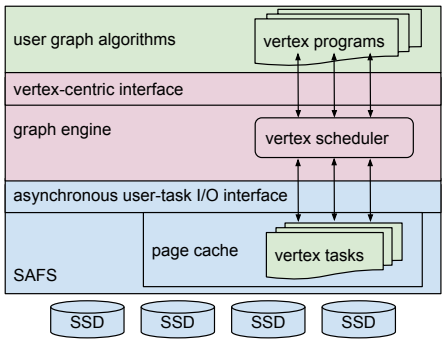


Figure 1: The architecture of FlashGraph.

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.

3.2 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 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. 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 Execution model

FlashGraph proceeds in iterations when executing graph algorithms, much like other engines. In each iteration, FlashGraph processes the vertices activated in the previous iteration. An algorithm ends when there are no active vertices in the next iteration.

As shown in Figure 2, 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.

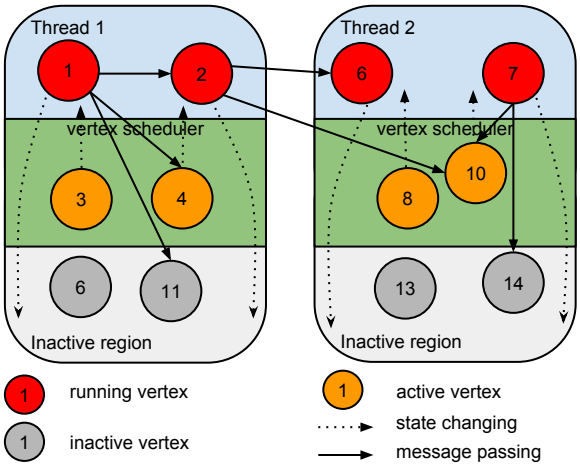


Figure 2: Execution model in FlashGraph.

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 it is scheduled. It remains in the running state until it finishes its task in the current iteration and becomes inactive. The running vertices interact with other vertices via message passing.

3.4 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 [20] and PowerGraph [11]. 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) 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.

```
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: 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 4: Breadth-first search in FlashGraph.

- 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 4. 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.

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 [5], 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.

3.4.1 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 [11]. 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.

3.5 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.

3.5.1 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 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 (≥ 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 4). 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 3.8.1).

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 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 [1]. All of the edge lists stored on SSDs are ordered by vertex ID, given by the input graph.

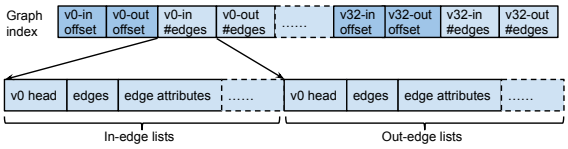


Figure 5: The data representation of a directed graph in FlashGraph. During computation, the graph index is maintained in memory and the in-edge and out-edge lists are accessed from SSDs.

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 3.6), which significantly alleviates this problem.

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 [7] 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 [16] and X-Stream [23] 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

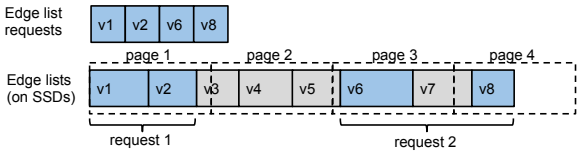


Figure 6: FlashGraph accesses edge lists and merges I/O requests.

each vertex requests a single edge list within an iteration. FlashGraph relies on its vertex scheduler (Section 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 patterns.

Figure 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 3.8).

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 [26] in Section 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.

3.8 Graph partitioning

FlashGraph partitions a graph in two dimensions at runtime (Figure 7), inspired by two-dimension matrix partitioning. It assigns each vertex to a partition for parallel processing, shown as *horizontal partitioning* in Figure 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 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 scheme maximizes data locality in memory access within each processor.

FlashGraph applies a *range partitioning* function to

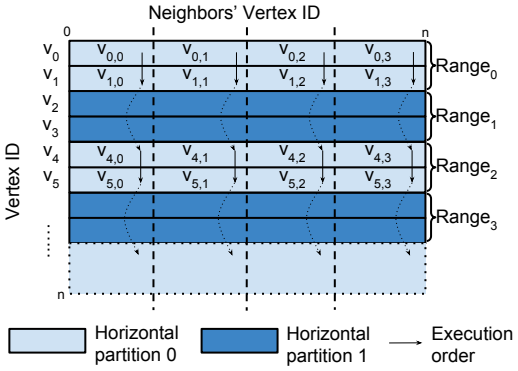


Figure 7: An example of 2D partitioning on a graph of n vertices, visualized as an adjacency matrix. In this example, the graph is split into two horizontal partitions and four vertical partitions. The size of a range in a horizontal partition is two. $v_{i,j}$ represents vertical partition j of vertex i. The arrows show the order in which the vertical partitions of vertices in horizontal partition 0 are executed in a worker thread.

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 3.7) 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 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.

3.8.1 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.

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 [6]. For performance evaluation, we perform this

process from a single source vertex. This requires both in-edge and out-edge lists.

PageRank (PR) [7]: In our PR, a vertex sends the delta of its most recent PR update to its neighbors who then update thier own PR accordingly [30]. 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 [20]. This requires only out-edge lists.

Weakly connected components (WCC): WCC in a directed graph is implemented with label propagation [33]. 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) [28]: 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) [26]: 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 [27]. 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.

5 Experimental Evaluation

We evaluate FlashGraph's performance on the applications in section 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 [11] and Galois [21]). 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 [16] and X-Stream [23]). 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.

Graph datasets	# Vertices	# Edges	Size	Diameter
Twitter [15]	42M	1.5B	13GB	23
Subdomain [29]	89M	2B	18GB	30
Page [29]	3.4B	129B	1.1TB	650

Table 1: Graph data sets. These are directed graphs and the diameter estimation ignores the edge direction.

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

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 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 9 shows the CPU and I/O utilization of

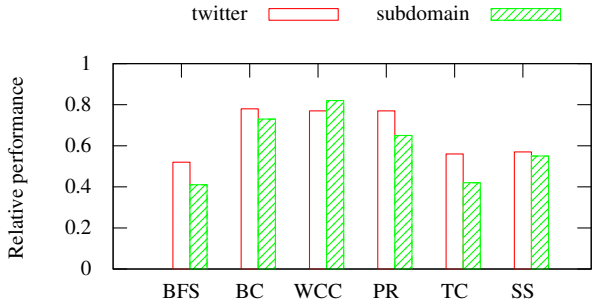


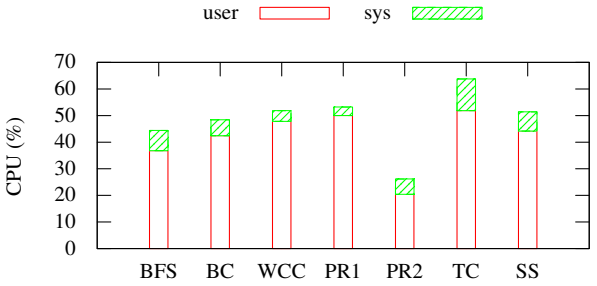
Figure 8: The performance of each application run on semi-external memory FlashGraph with 1GB cache relative to in-memory FlashGraph.

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.

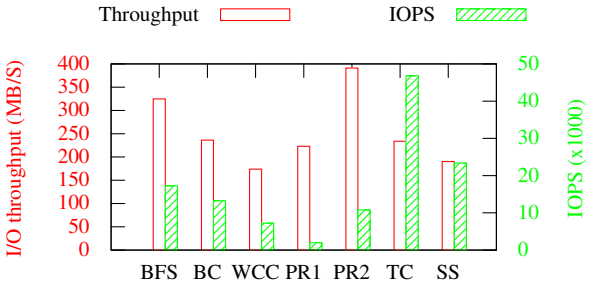
5.2 FlashGraph vs. in-memory engines

We compare the performance of FlashGraph to PowerGraph [11], a popular distributed in-memory graph engine, and Galois [21], 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 multi-thread mode to achieve its best performance and use its synchronous execution engine because it performs better than the asynchronous one on both graphs.

Both in-memory and semi-external memory FlashGraph performs comparably to Galois, while signif-

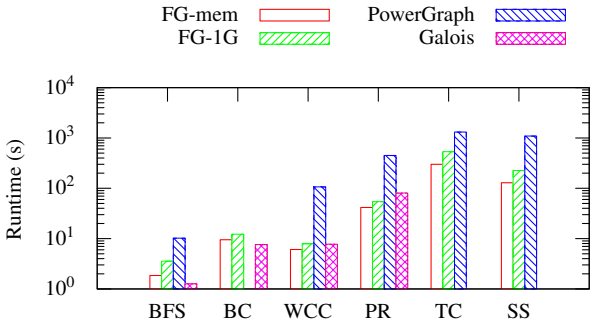


(a) CPU utilization in the user and kernel space. Hyper-threading enables 64 hardware threads in a 32-core machine, so 50% CPU utilization means 32 CPU cores are saturated.

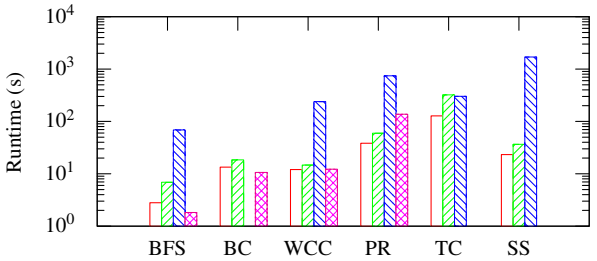


(b) I/O utilization.

Figure 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.



(a) On the Twitter graph.



(b) On the subdomain Web graph.

Figure 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.

icantly outperforming PowerGraph (Figure 10). In-memory FlashGraph outperforms Galois in WCC and PageRank. It performs worse than Galois in graph traversal applications such as BFS and betweenness centrality, because Galois uses a different algorithm [3] 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.

5.3 FlashGraph vs. external memory engines

We compare the performance of FlashGraph to that of two external-memory graph engines, X-Stream [23] and GraphChi [16]. 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 [4].

FlashGraph outperforms GraphChi and X-Stream by one or two orders of magnitude (Figure 11a). 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 reasonable amount of memory when compared with GraphChi and X-Stream (Figure 11b). 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.

5.4 Scale to billion-node graphs

We further evaluate the performance of FlashGraph on the billion-scale page graph in Table 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 [20], 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 2). For example, FlashGraph

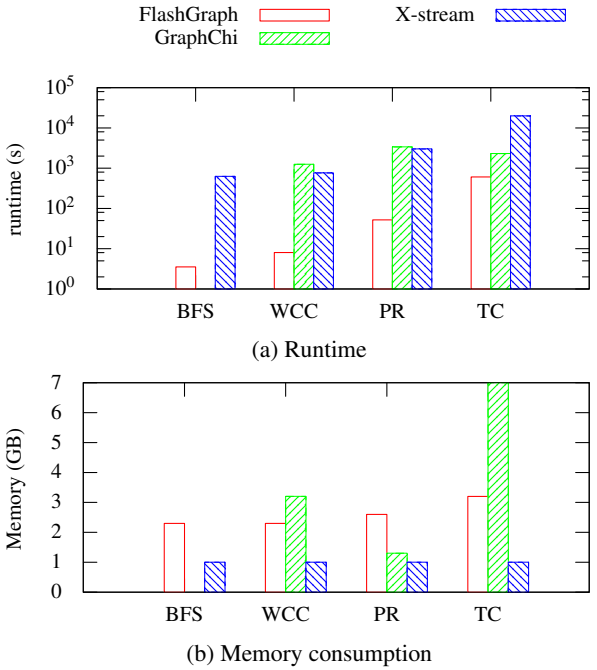


Figure 11: The runtime and memory consumption of semi-external memory FlashGraph and external memory graph engines on the Twitter graph.

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 [20] 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 [24] 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 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.

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

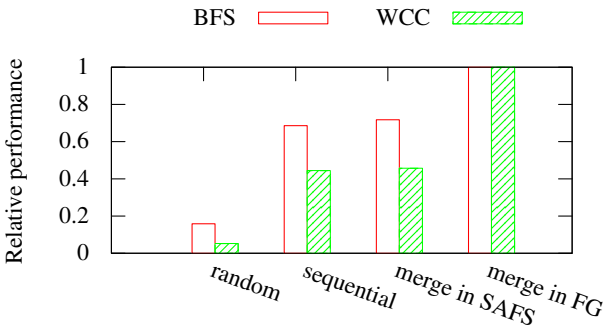


Figure 12: The impact of preserving sequential I/O access in graph applications. All performance is relative to that of the implementation of merging I/O requests in FlashGraph.

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.

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.

5.5.1 Preserve 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 12) between random execution and sequential execution suggests that there exists a degree of sequential I/O in both applications, as described in Section 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 [21].

Figure 12 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, we achieve 40% speedup for BFS and more than 100% speedup for WCC.

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 13 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 iterations but the first. This result suggests that TurboGraph [12], 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.

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.

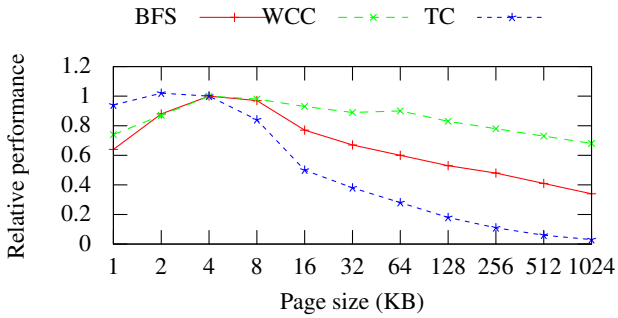


Figure 13: The impact of the page size in FlashGraph. All performance is relative to that of the implementation with 4KB page size.

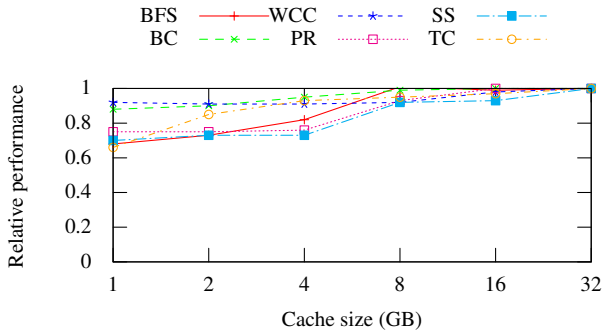


Figure 14: The impact of cache size in FlashGraph.

FlashGraph performs well even with a small page cache (Figure 14). 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 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.

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.

7 Acknowledgments

We would like to thank the FAST reviewers, especially our shepherd John Ousterhout, for their insightful comments and guidance for revising the paper. We also thank Heng Wang and Vince Lyzinski in Department of Applied Mathematics & Statistics of the Johns Hopkins University for discussing the applicatins of FlashGraph. This work is supported by DARPA N66001-14-1-4028, NSF ACI-1261715, NIH NIBIB 1RO1EB016411-01, and DARPA XDATA FA8750-12-2-0303.

References

[1] ABADI, D., BONCZ, P., HARIZOPOULOS, S., IDREOS, S., AND MADDEN, S. The design and implementation of modern column-oriented database systems. Foundations and Trends in Databases 5 (2013), 197–280.

[2] ABELLO, J., BUCHSBAUM, A. L., AND WESTBROOK, J. R. A functional approach to external graph algorithms. In Algorithmica (1998), Springer-Verlag, pp. 332–343.

[3] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (2012), SC '12.

[4] BECCHETTI, L., BOLDI, P., CASTILLO, C., AND GIONIS, A. 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).

[5] BLONDEL, V. D., LOUP GUILLAUME, J., LAMBIOTTE, R., AND LEFEBVRE, E. Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment (2008).

[6] BRANDES, U. A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25 (2001), 163–177.

[7] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. In Proceedings of the Seventh International Conference on World Wide Web 7 (1998).

[8] CHEN, R., WENG, X., HE, B., YANG, M., CHOI, B., AND LI, X. On the efficiency and programmability of large graph processing in the cloud. Tech. rep., Microsoft Research, 2010.

[9] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (2004).

[10] Apache giraph. `https://giraph.apache.org/`, Accessed 4/9/2014.

[11] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. PowerGraph: Distributed graph-parallel computation on natural graphs. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (2012).

[12] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. 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 (2013), ACM, pp. 77–85.

[13] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. PEGASUS: A peta-scale graph mining system implementation and observations. In Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (2009).

[14] KEPNER, J., AND GILBERT, J. Graph Algorithms in the Language of Linear Algebra. Society for Industrial & Applied Mathematics, 2011.

[15] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is twitter, a social network or a news media? In Proceedings of the 19th International Conference on World Wide Web (2010).

[16] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a PC. In Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (2012).

[17] LESKOVEC, J., LANG, K. J., DASGUPTA, A., AND MAHONEY, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Mathematics 6, 1 (2009), 29–123.

[18] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow. (2012).

[19] LUGOWSKI, A., ALBER, D., BULU, A., GILBERT, J., REINHARDT, S., TENG, Y., AND WARANIS, A. A flexible open-source toolbox for scalable complex graph analysis. In Proceedings of the 2012 SIAM International Conference on Data Mining (2012).

[20] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (2010).

[21] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (2013).

[22] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multi-threaded 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).

[23] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (2013).

[24] SHAO, B., WANG, H., AND LI, Y. Trinity: A distributed graph engine on a memory cloud. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (2013).

[25] SHUN, J., AND BLELLOCH, G. E. 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).

[26] WANG, H., TANG, M., PARK, Y., AND PRIEBE, C. E. Locality statistics for anomaly detection in time series of graphs. IEEE Transactions on Signal Processing 62, 3 (2014).

[27] WANG, H., ZHENG, D., BURNS, R., AND PRIEBE, C. Active community detection in massive graphs. CoRR abs/1412.8576 (2015).

[28] WATTS, D. J., AND STROGATZ, S. H. Collective dynamics of 'small-world' networks. Nature 393, 440-442 (1998).

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

[30] ZHANG, Y., GAO, Q., GAO, L., AND WANG, C. Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation. IEEE Transactions on Parallel and Distributed Systems (2014).

[31] ZHENG, D., BURNS, R., AND SZALAY, A. S. A parallel page cache: Iops and caching for multicore systems. In Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems (2012).

[32] ZHENG, D., BURNS, R., AND SZALAY, A. S. 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).

[33] ZHU, X., AND GHAHRAMANI, Z. Learning from labeled and unlabeled data with label propagation. Tech. rep., Carnegie Mellon University, 2002.