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

Da Zheng, Disa Mhembere, Randal Burns
Department of Computer Science
Johns Hopkins University
Baltimore, Maryland 21218

Alexander S. Szalay
Department of Physics and Astronomy
Johns Hopkins University
Baltimore, Maryland 21218

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 size of the graph. We demonstrate that a multicore server can process graphs of billions of vertices and hundreds of billions of edges, utilizing commodity SSDs without much performance loss. We do so by implementing a graph-processing engine within a user-space SSD file system designed for high IOPS and extreme parallelism. This allows us to localize computation to cached data in a non-uniform memory architecture and hide latency by overlapping computation with I/O. Our semi-external memory graph engine, called FlashGraph, stores vertex state in memory and adjacency lists on SSDs. FlashGraph exposes a general and flexible programming interface that can express a variety of graph algorithms and their optimizations. FlashGraph in semi-external memory performs many algorithms up to 20 times faster than PowerGraph, a general-purpose, in-memory graph engine. Even breadth-first search, which generates many small random I/Os, runs significantly faster in FlashGraph.

I. 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 on the cloud with applications to web structure and social networking [12], [25], machine learning [22], and network analysis [27]. 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 real-world graphs exhibit a power-law distribution on the degree of vertices. As a result, graphs cannot be clustered or partitioned effectively [21] to localize access. While good partitions may be important for performance [9], leading systems partition natural-graphs randomly [13].

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 [22], [13], or message passing [25], [12], [29] 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 single nodes [20], [28] 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 [8] 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 a graph-processing engine, called FlashGraph, 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 [27] maintains vertices and algorithmic state in RAM and edges (adjacency lists) on storage. FlashGraph uses an array of flash-memory SSD (solid-state storage devices) to achieve high throughput and low latency to storage. Unlike magnetic disk-based engines, FlashGraph supports random and selective access to adjacency lists. Although this paper explores the scalability limits of graph processing on a shared-memory multiprocessor, we envision deploying this architecture as the building block for graph-processing clusters with the benefit that graph partitions that are orders of magnitude larger will better utilize hardware and result in less network traffic.

We build FlashGraph within an SSD file system. This design localizes computation to data in a non-uniform memory architecture (NUMA) and overlaps I/O with processing to hide latency. FlashGraph is deployed with the set-associative file system (SAFS) [35] that refactors I/O scheduling, data placement, and caching data structures for the extreme parallelism of modern NUMA multiprocessors. FlashGraph stores edges of vertices in the SAFS cache to reduce disk I/O. It partitions a graph and assigns vertices to CPU cores to minimize remote memory accesses. FlashGraph’s task scheduler is integrated with the asynchronous I/O interface of SAFS so that many parallel tasks can be scheduled and tasks execute when data are ready.

Although SSDs can deliver high IOPS—we demonstrated over 1M IOPS on this system last year [35]—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 with the application and file systems. Finally, high-speed I/O consumes a lot of processing power, which can interfere with processing for graph algorithms.

Our results show that FlashGraph in semi-external memory outperforms in-memory graph engines on a wide-variety of algorithms that generate diverse access patterns. We further demonstrate FlashGraph can process massive natural graphs in a single machine with relatively small memory footprint. e.g. we computed triangle counting for a graph of 3.4 billion vertices and 129 billion edges using only 94 GB of memory. This is even smaller than the memory used by PowerGraph to compute triangle counting on a graph that is nearly two orders of magnitude smaller. We conclude that FlashGraph offers new design possibilities for graph processing clusters, replacing memory with larger and cheaper SSDs and processing bigger problems on fewer nodes.

II. RELATED WORK

MapReduce \cite{11} is a general large-scale data processing framework. PEGASUS \cite{17} 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 \cite{6} and label propagation \cite{36}, but performs poorly for graph traversal algorithms.

Several other works \cite{18, 23} perform graph analysis linear algebraically, using 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.

Boost Graph Library \cite{5}, Parallel Boost Graph Library \cite{14} and iGraph \cite{10} provide a large collection of pre-written graph algorithms. Despite users benefiting from these libraries by invoking existing implementations and using the APIs, these libraries lack a runtime environment within which to express natively parallel algorithms that scale to match very large-scale graphs.

Pregel \cite{25} 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 in a cluster. Giraph \cite{12} is an open-source implementation of the Pregel programming model.

Many distributed graph engines adopt the vertex-centric programming model and express different designs to improve performance. GraphLab \cite{22} and PowerGraph \cite{13} 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 \cite{29} optimizes message passing by restricting vertex communication to a vertex and its direct neighbors.

Liga \cite{30} is a simple 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 and PowerGraph, and the graph size supported by the graph engine is limited by the memory size of a single machine.

Galos \cite{26} provides a low-level abstraction to implement graph engines. The core of the Galois framework is its novel task scheduler. These concepts are orthogonal to FlashGraph’s I/O optimizations and could be adopted.

GraphChi \cite{20} and X-stream \cite{28} are external-memory graph engines 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.

Abello et al. \cite{1} introduced the semi-external memory algorithmic framework for graphs. Pearce et al. \cite{27} implemented several semi-external memory graph traversal algorithms for SSDs and Fusion-I/O. FlashGraph partially adopts and advances several concepts introduced by these works.

III. 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 storage devices (SSDs). It stores the adjacency lists of vertices on SSDs and maintains vertex state in memory. FlashGraph is also designed to have a concise and flexible programming interface to express a wide variety of graph algorithms, as well as their optimizations. FlashGraph runs on top of the set-associative file system (SAFS) \cite{35}, a user-space filesystem designed to realize both high IOPS and caching for SSD arrays on non-uniform memory and I/O systems.

To optimize performance, the design of FlashGraph follows the following principles:

- **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 caused by interrupts and 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 workload, one must initiate many parallel I/Os and process data when it is ready.

- **Avoid remote memory access:** Modern multi-processor 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.

FlashGraph realizes these design principles through a close integration with the SAFS file system. It pushes data-oriented computing into the SAFS page cache, which allows it to use asynchronous I/O interfaces for parallel I/O and pushes processing to the data location, removing copies and data movement.

### A. SAFS

SAFS [35] is a user-space filesystem for high-speed SSD arrays in a NUMA machine. It eliminates overhead in the block subsystem of the Linux kernel to extract maximal performance from an SSD array. Furthermore, it has a scalable, lightweight page cache to amplify the user-perceived I/O performance.

SAFS is optimized for both small and large I/O requests. Natural graphs exhibit a power-law distribution on vertex degrees, so graph analysis requires the access of a few large vertices and many small vertices. Therefore, SAFS preserves large I/O requests from the upper layer and merges small requests into larger ones when they go through SAFS’s page cache. Large requests increase I/O throughput and reduce CPU overhead.

To better support FlashGraph, we add an asynchronous user-task I/O interface to SAFS that 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, with asynchronous I/O, user-space buffers must be allocated in advance and data must be copied into these buffers. This creates processing overhead for copying and pollutes memory with empty buffers waiting to be filled. In the 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 memory copy for asynchronous I/O.

The SAFS user-task interface provides a Java-style iterator that abstracts out the details of non-contiguous data access. Functions operate on multiple pages of data in the cache without copying them into a buffer. However, there is cost associated with accessing data in discontiguous pages. Accessing an element now requires extra CPU instructions to compute data locations or to check the end of pages. Even inexpensive instructions such as bit operations can cause considerable computational overhead. Our Java-style iterator allows functions to read the data across multiple, discontiguous pages as if they were sequential, while minimizing the number of CPU instructions used for data access.

### B. 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 and overlap computation with I/O. FlashGraph also takes advantage of the scalable, lightweight page cache of SAFS to buffer the edge lists from SSDs. The result is that FlashGraph does not need to maintain a cache itself. This eliminates the complexity of maintaining both a page cache and an object cache for vertices, and reduces data redundancy.

FlashGraph exposes a vertex-centric programming interface. As such, users express graph algorithms from the perspective of a vertex. Vertex programs, composed of user-defined vertex state and logic, encapsulate graph algorithms and run inside the graph engine. 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.

### C. Execution model

FlashGraph maintains algorithmic vertex state in memory and executes vertex programs on the vertex state. Figure 2 shows FlashGraph’s execution model.

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 it is scheduled to run. It remains in the running state until it finishes its task in the current iteration, and becomes inactive.

The running vertices can interact with other vertices via message passing. They can send messages to both active and
inactive vertices in any partition of a graph. All vertices, regardless of their states, process messages once they receive them.

D. Programming model

One of FlashGraph’s goals is to provide a flexible programming interface to express a variety of graph algorithms and their optimizations. FlashGraph adopts the actor programming model [2], a concurrent programming model used by well-known languages such as Erlang and Scala. This model treats actors (vertices in our case) as the focal point of computation and control, and constructs a graph algorithm from the perspective of a vertex. In this programming model, each vertex performs user-defined tasks independently and interacts with other vertices as defined by program logic. A vertex affects the state of others by sending messages to them as well as activating them. Notably, FlashGraph allows a vertex to send messages to any vertex in the graph. We extend this model by allowing a vertex to read the vertex information of any vertex from SSDs as well as the state of any vertex in memory.

Like other graph engines, graph algorithms in FlashGraph proceed in iterations. 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.

The FlashGraph API provides three run methods for users to express graph algorithms (Figure 3). FlashGraph executes the run method exactly once for each active vertex in an iteration; the order of execution of this method on vertices is subject to scheduling by FlashGraph. The execution of the run_on_vertex and run_on_message methods is event-driven. FlashGraph pushes the execution of run_on_vertex to SAFS, which executes the method on a vertex if the vertex information requested by the vertex from SSDs is ready in the page cache. FlashGraph executes run_on_message on a vertex if the vertex receives messages from other vertices. The run_on_message method may be executed even if a vertex is inactive in an iteration.

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 neighbor list in run and activates its neighbors in run_on_vertex. In this example, vertices do not need to send messages to one another so we do not need to implement run_on_message.

This interface is designed for better flexibility and gives users fine-grained programmatic control. For example, a vertex has to explicitly request its own neighbor list when it is needed, as shown in Figure 4. This flexibility can significantly reduce the amount of data brought to memory for many graph algorithms. Furthermore, the interface does not constrain the vertices that a vertex can communicate with or vertex information that a vertex can request from SSDs. This flexibility can potentially allow FlashGraph to handle algorithms such as Louvain clustering [6], in which changes to the topology of the graph occur during computation. It is difficult to express such algorithms with graph frameworks where vertices can only interact with direct neighbors.

E. Message passing

Despite all vertices being on a single machine, we support message passing for vertices to push data to other vertices. Message passing helps ensure data race freedom on vertex state (section III-F). In a semi-external memory graph engine, vertices are no longer able to push data by embedding data on edges like PowerGraph [13]. Writing data to other vertices directly causes race condition on vertex state and requires atomic operations or locking for synchronization. Message passing is necessary for algorithms such as Louvain clustering [6] and several others, including graph algorithms. To support message passing, FlashGraph instrumented BFS’s run and run_on_message methods to execute the run_on_message method on a vertex if the vertex receives a message from another vertex.
passing is a light-weight alternative for pushing data to other vertices.

The worker threads in FlashGraph send and receive messages on behalf of vertices. They maintain buffers for both sending and receiving messages. When a vertex sends a message, the sending thread buffers the message. A worker thread maintains a sending buffer for every other thread and it chooses a buffer for a message based on a graph partition function (section III-F). A worker thread flushes messages when the buffer gets full. Each worker thread maintains a single receiving buffer. To reduce memory consumption, it processes messages when the buffer accumulates a certain number of messages. Therefore, FlashGraph naturally supports asynchronous computation, demonstrated to have better performance than synchronous computation [4], [22].

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 which significantly reduces memory copy. However, there is some overhead associated with multicast. When a vertex multicasts a message to a small number of recipient vertices, FlashGraph uses point-to-point message passing. The current implementation only switches to multicast when a vertex sends the same message to vertices more than the number of worker threads.

We implement vertex activation with message passing, more specifically, with multicast, since activation messages contain no data and are identical. Each thread maintains a bitmap to keep track of vertices activated for the next iteration in its own partition. This design minimizes the overhead for vertex activation.

F. Graph partitioning

FlashGraph splits a graph into multiple partitions and creates a thread to process each partition independently. Each thread is associated with a specific processor and maintains the state of vertices in its own partition. When a thread processes a vertex in its own partition, all memory access to the vertex state is localized to the processor. As such, our partitioning scheme maximizes data locality within each processor.

FlashGraph applies a range partitioning function to graphs at runtime. The function performs right bit shift operations on a vertex ID by a predefined number of bits and takes the modulo of the shifted result.

\[ \text{partition}_\text{id} = (\text{vid} >> \text{bits}) \% \# \text{partitions} \]

The number of shifted bits determines the size of the range within the vertex ID space in a partition and is tunable by individual programmers. A larger range generally improves performance, but a very large range may cause load imbalance. We empirically determined 10 bits to work well for most graphs.

FlashGraph benefits greatly from range partitioning because it helps to improve spatial data locality for disk I/O in many graph algorithms. FlashGraph uses a per-thread I/O scheduler (Section III-I) that optimizes I/O based on its local knowledge. With range partitioning, most vertices in the same partition are stored adjacently on SSDs, which helps the per-thread I/O scheduler issue larger I/O requests.

Range partitioning also reduces random access to the message buffers for message passing. Range partitioning assigns messages sent to a certain vertex ID range to the same message buffer. When FlashGraph sends a list of messages, sorted by their destinations, it is likely multiple messages that are adjacent in the list will eventually reside in the same message buffer. Since the edge lists of vertices on SSDs are sorted, it is common that messages sent by a vertex are sorted by their destinations.

G. Load balancing

FlashGraph provides a dynamic load balancer to address the computational skew that high degree vertices in scale-free graphs may create. Since we partition vertices statically, the load balancer assists in ensuring we avoid computational imbalance at runtime. In each 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. When it finishes processing a vertex, it returns the vertex to its owner thread. This process continues until no threads have active vertices left in the current iteration.

Load balancing only moves computation in the run and run_on_vertex methods to other threads, while message processing always remains in the thread where the recipient vertex belongs. We make this decision for simpler implementation as well as smaller memory consumption. For a large graph with very skewed workload, it is likely that a thread may need to process many more vertices than other threads. Moving message processing to other threads would require keeping track of vertex movement, which may result in substantial memory consumption and computation overhead.

H. Race free guarantee on the state of a vertex

FlashGraph provides a strong guarantee that the computation on vertex state is always race-condition free, i.e., the three run methods described in Section III-D are never executed simultaneously on the same vertex. FlashGraph splits the computation on a vertex into three parts and re-enters the computation multiple times, which could potentially lead to concurrent computation on a vertex without this guarantee. This guarantee simplifies programming and removes the necessity of using locking to protect vertex state. The significance is the improvement in performance and the reduction in memory consumption for applications that otherwise would require locking.

When a worker thread processes vertices in its own partition, SAFS supports the race free guarantee. SAFS always executes the user task associated with an I/O request in the thread that issues the I/O request. That is, when a worker thread executes the run method to request vertex information from SSDs, the run_on_vertex method will be invoked in the same worker thread. FlashGraph always processes messages in the threads that own the destination vertices. Therefore, all computation of a vertex runs in the same thread.
FlashGraph continues to support the race free guarantee even when load balancing is triggered. During load balancing, FlashGraph may execute the run and run_on_vertex methods of a vertex in a different thread from where its messages are processed. To ensure race freedom on a vertex, a worker thread maintains a bitmap to keep track of the stolen vertices in its own partition. Before processing messages, a thread checks the bitmap and buffers the messages if the recipient’s vertices are stolen. It resumes processing the messages when the stolen vertices are returned. This lock-free scheme is lightweight, but still causes noticeable overhead. To further reduce the cost of providing the race freedom guarantee during load balancing, FlashGraph splits an iteration into two phases for processing messages: the normal phase and the load-balancing phase. A thread processes messages normally in the normal phase. Once a thread detects a stolen vertex, it switches to the load-balancing phase. As a result, the cost of enforcing the race freedom on a vertex is negligible in most cases.

I. Vertex and I/O scheduling

Vertex scheduling can greatly affect the performance of graph algorithms. Intelligent scheduling accelerates the rate of convergence of graph algorithms and improves I/O performance. FlashGraph’s default scheduler focuses on increasing page cache hit rates and I/O request sizes. FlashGraph also allows users to customize the vertex scheduler to optimize their specific algorithms.

FlashGraph deploys a per-thread vertex scheduler. Each thread schedules vertices in its own partition independently. Such a greedy approach may occasionally result in a globally suboptimal scheduling decision, but this strategy simplifies implementation and results in a very high degree of framework scalability. The per-thread scheduler processes multiple vertices simultaneously within an iteration so that many I/O requests can be issued to the underlying filesystem to improve I/O performance. The number of vertices being processed in each thread is user-configurable. It is a tradeoff between the overall throughput of the graph engine and the latency of completing processing an individual vertex. The higher the number is, the higher throughput the graph engine can provide, but the longer latency may occur for a vertex.

The default scheduler processes vertices ordered by vertex ID. This scheduler works well for many graph algorithms. Since vertices on the SSDs are also sorted by vertex IDs, this scheduling increases data locality and also gives the underlying filesystem the opportunity to merge adjacent I/O requests. A similar scheme is employed by Pearce et al. [27].

The default vertex scheduler, like the disk elevator algorithm [16], alternates the direction in which it scans the queue of active vertices between iterations so as to increase the cache hit rate. If the graph engine processes active vertices in the same order in each iteration, it is likely to cause cache thrashing, especially for graph traversal algorithms. Pages accessed at the end of the previous iteration are still in the page cache when FlashGraph enters the next iteration. These pages are likely to be accessed again if we alternate the direction in which we scan the active vertex queue in the next iteration.

FlashGraph also reschedules I/O requests generated by user-defined vertex tasks to increase the cache hit rate. FlashGraph also supports accessing part of a vertex’s data from SSDs to reduce the amount of data brought to memory.
and reduce page cache pollution. With this optimization, a graph application can read the in-edge list and the out-edge list of a vertex separately in a directed graph. We refer to a request for part of a vertex on SSDs as a “partial vertex request” and a request for a complete vertex as a “full vertex request”.

This optimization only targets high degree vertices because there is overhead associated with invoking it. When reading part of a vertex, FlashGraph needs to maintain an additional data structure in memory to recognize data brought from SSDs by a partial vertex request. A partial vertex request for a small vertex that resides in a single page may actually reduce performance by introducing extra overhead. Therefore, FlashGraph converts partial vertex requests into full vertex requests for small vertices.

IV. 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 and thus provide a comprehensive evaluation of FlashGraph.

Breadth-first search (BFS): This algorithm is a building block for many graph applications such as computing betweenness centrality and diameter estimation. The algorithm proceeds in iterations. It starts with a single active vertex; this vertex activates its neighbors. In each subsequent iteration, the vertices that are active but have not been visited activate their neighbors. The algorithm proceeds until there are no active vertices left. Our implementation only uses out-edges to traverse a graph.

Diameter estimation (DE): Computing the diameter of a graph requires one to perform BFS from all vertices in a graph. This is generally unrealistic for most large graphs. To compute an estimate, we instead perform $K \in \mathbb{N}^+$ BFS procedures in parallel from $K$ randomly chosen vertices and use the longest distance as the diameter of the graph. The current implementation performs $K = 5$ parallel BFS procedures to trade off between performance and memory consumption. We use this implementation to estimate the diameter of the real-world graphs in Table I.

PageRank (PR): PageRank is used to rank the relevance of a web page given a query. In our implementation, a vertex sends the delta of its most recent PageRank update to its neighbors if the delta is larger than a threshold. Vertices receive the delta from their neighbors and update their own PageRank. In PageRank, vertices converge at different rates. As the algorithm proceeds, fewer and fewer vertices are activated in an iteration. The algorithm ends when all vertices converge or it reaches the maximal number of iterations. We set the maximal number of iterations to 30, to match the value used by Pregel.

(Weakly) connected components (WCC): Both finding connected components in an undirected graph and weakly connected components in a directed graph can be implemented with label propagation. All vertices start in their own components, broadcast their component IDs to all neighbors, and adopt the smallest IDs they observe. The algorithm ends when no vertices change their component IDs.

Triangle counting (TC): Triangle counting is used in the analysis of graph connectivity. A vertex computes the intersection of its own edge list and the edge list of each neighbor. This computation is fairly expensive. Therefore, 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.

Scan statistics (SS): Scan statistics are commonly used in signal processing to detect a local signal. Large values of the scan statistic suggest existence of nonhomogeneity, for example, a local region in the graph with significantly excessive communications. Computing scan statistics on a graph requires the computation of the intersection of a vertex’s edge list and its neighbors’ edge lists much like triangle counting. In contrast, scan statistics only require finding the maximum in a graph. We use the assumption that vertex degrees in real-world graphs often approximately follow a power-law distribution, to help us avoid actual computation for many vertices. To compute scan statistics efficiently, we instantiate a user-defined vertex scheduler that begins computing local statistics on vertices with the largest degree first. We then update a global variable with the largest local statistic that has been observed. Before computing the local statistic on a vertex, we estimate its upper bound with a cost-effective calculation and only perform actual computation if the upper bound is larger than the current largest local statistic.

These graph algorithms fall into three categories from the perspective of I/O access patterns. BFS and diameter estimation only perform computation on a subset of vertices in a graph within an iteration, thus they generate many random I/O accesses. PageRank and (weakly) connected components need to process all vertices at the beginning, so they generally do not generate many random I/O accesses. Triangle counting and scan statistics require a vertex to read its own edge list as well as its neighbors’ edge lists. These two graph algorithms are more I/O intensive than the others and generate many random I/O accesses.

V. EXPERIMENTAL EVALUATION

We evaluate the performance of FlashGraph with the applications in section IV on large real-world graphs. We first perform a microbenchmark on an SSD array that we use to evaluate the performance of FlashGraph. We compare the performance of FlashGraph with external-memory graph engines (GraphChi and X-Stream) and an in-memory graph engine (PowerGraph).

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 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 12 OCZ Vertex 4 SSDs are installed. In addition to the LSI HBAs, there is one RAID controller that connects to the disks that contain the root filesystem. The machine runs Ubuntu Linux 12.04 and Linux kernel v3.2.30.
TABLE I. GRAPH DATA SETS.

| Graph datasets | # Vertices | # Edges | Size (GB) | Diameter |
|----------------|------------|---------|-----------|----------|
| Twitter [19]   | 42M        | 1.5B    | 12        | 15       |
| Subdomain [34] | 89M        | 2B      | 17        | 140      |
| Page [34]      | 3.4B       | 129B    | 1013      | 3274     |

A. Graph datasets

We use the real-world graphs in Table I for evaluation. Among them, the largest graph is the web 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 web page graph is clustered by domain; this generates good cache hit rates for some graph algorithms.

We estimate the lower bound of the diameter of these directed graphs with the implementation we describe in Section IV. The Twitter graph has a relatively small diameter, but the subdomain graph and the page graph have very large diameters.

B. SSD I/O benchmark

The performance of FlashGraph highly depends on the performance of the SSD array where FlashGraph stores edge lists of vertices. We benchmark the SSD array with various request sizes to set the expected baseline performance of FlashGraph.

As shown in Figure 6, smaller I/O requests achieve higher IOPS (I/O per second) but lower throughput (bytes per second), at the cost of higher CPU consumption. With I/O requests of 4 KB, the SSD array can perform 742, 442 IOPS or 2900 MB/s, and consumes 13% CPU. In contrast, the SSD array delivers 19, 1624 IOPS or 4800 MB/s with I/O requests of 256 KB, and CPU utilization drops to 1.4%. As suggested by the highest IOPS of the SSD array, FlashGraph is able to visit at least 742, 442 vertices per second in graph traversal algorithms. Furthermore, FlashGraph should always favor larger I/O requests to increase I/O throughput and reduce CPU consumption.

C. The impact of page cache size

We investigate the effect of the SAFS page cache size on the performance of FlashGraph. We vary the size of the page cache from 1 GB to the size of the graph datasets for the Twitter graph and subdomain web graphs. We show the results on the subdomain graph in Figure 7. We omit Twitter graph results as they mirror subdomain graph results and thus are redundant. We normalize the runtime of each application to the best performance of that application to show the relative performance impact of the page cache size.

Figure 7 shows that the degree to which a graph algorithm benefits from a larger page cache varies. Algorithm specific I/O access patterns and graph topology can limit the effect of a larger cache. For example at one end of the spectrum, weakly connected components requires a sequential read of the entire graph at its onset, but thereafter rapidly converges. As such, it generates large I/O requests to SSDs and is predominantly bottlenecked by the CPU, limiting cache size influence. At the other end, consider the I/O intensive triangle counting, characterized by heavy random read requests to SSDs. Accordingly, it stands to reason that cache size has its greatest impact on triangle counting. Lastly, consider the middle ground, PageRank, which also performs a sequential graph read at its onset, but converges at a significantly slower rate than weakly connected components. As the algorithm progresses, more vertices converge and the algorithm begins to read vertices more selectively. Eventually, the algorithm is bottlenecked by I/O, so a larger page cache still improves I/O performance. By varying the page cache size, we show how FlashGraph can smoothly transition from a semi-external memory graph engine to an in-memory graph engine.

The cache size has a much larger impact on a single BFS than on the multiple parallel BFSs employed in diameter estimation. BFS requires selective reading of vertices from SSDs and generates many small random reads. Although all vertices are only visited once, a page may be read multiple times because it contains multiple small vertices. A larger page cache captures more of such cache hits. When we perform multiple BFSs in parallel, data brought to the page cache by one BFS can be reused by another BFS, so diameter estimation potentially generates higher cache hits and the application becomes more CPU-intensive. Therefore, it is advisable to perform parallel BFSs in FlashGraph if possible. Many other graph algorithms such as betweenness centrality [7] and strongly connected components [15] call for this.

D. FlashGraph vs. external memory graph engines

We compare the performance of FlashGraph to that of two external-memory graph engines, X-Stream [28] and GraphChi [20]. In this experiment, we set FlashGraph’s page cache size to 1 GB and run it from SSDs. To insure optimal performance
from X-Stream and GraphChi, we store their graph data files, and run them directly from RAM disk. The RAM disk’s read/write performance is an order of magnitude faster than the SSDs on which FlashGraph runs. Existing implementations of PageRank within distributions of both X-Stream and GraphChi cannot converge by themselves, hence we set a hard limit of 30 iterations for all frameworks. Furthermore, note that GraphChi does not provide a BFS implementation, and X-Stream implements triangle counting via a semi-streaming approximation algorithm [33].

Despite the considerable hardware advantages given to X-Stream and GraphChi, FlashGraph still outperforms them by one or two orders of magnitude. FlashGraph’s main advantage is that it only needs to perform computation on the vertices required by the graph algorithms. Even though it generates random I/O accesses, avoiding unnecessary computation saves both CPU and I/O. In contrast, GraphChi and X-Stream scan the entire graph dataset multiple times, wasting CPU and I/O bandwidth. As the authors of X-Stream point out themselves, the problem gets worse on a graph with a larger diameter. As we show in Table II the Twitter graph has a relatively small diameter, so we anticipate both X-Stream and GraphChi’s performance will degrade with the other graphs.

FlashGraph consumes a reasonable amount of memory compared to GraphChi and X-Stream, as we show in Table III. X-Stream has the most stable memory consumption, but cannot perform exact triangle counting. FlashGraph has comparable memory consumption to GraphChi. The performance of triangle counting in GraphChi highly depends on the memory size available for computation. It requires a considerable amount of memory to achieve reasonable performance.

### E. FlashGraph vs. PowerGraph

We compare the performance of FlashGraph against PowerGraph [13], a general-purpose in-memory graph engine. We use the Twitter and subdomain web graphs to process in a single machine. We run FlashGraph in two configurations: (i) Page cache of 20 GB with the entire graph preloaded and (ii) Page cache of 1 GB without preloading. The first configuration turns FlashGraph into an in-memory graph engine. Note that although we preload the entire graph to the page cache, there is small overhead associated with accessing data in the page cache. We use PowerGraph’s synchronous execution engine because it performs better than the asynchronous one on both graphs.

For a complete comparison, we implement the same algorithm for computing scan statistics in PowerGraph as the one in FlashGraph. The implementation requires the use of a PowerGraph asynchronous engine to perform any pruning. However, PowerGraph does not allow users to customize scheduling, so the pruning optimization is less effective than in FlashGraph.

Table IV and Figure 8 show that FlashGraph outperforms PowerGraph on both graphs in almost all applications. When FlashGraph preloads the entire graph, it outperforms PowerGraph in all applications and can run up to an order of magnitude faster than PowerGraph for breadth-first search, weakly connected components and PageRank. FlashGraph with a cache size of 1 GB has the same performance advantage over PowerGraph, running up to 20 times faster than it for certain applications. FlashGraph has much smaller overhead for transition between iterations, which is more noticeable in a large-diameter graph. Therefore, FlashGraph with a cache size of 1 GB can outperform PowerGraph in breadth-first search by a factor of five on the subdomain web graph. Since PowerGraph uses a synchronous execution engine, its implementations for finding weakly connected components and computing PageRank converge at a slower rate. Furthermore, PowerGraph cannot perform effective pruning when computing scan statistics so it needs to perform actual computation on many more vertices than necessary FlashGraph.

FlashGraph has much smaller memory footprint than PowerGraph (Figure 9). 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.

---

1We are unable to reproduce the BFS result shown in the original paper.
We further evaluate the performance of FlashGraph on the billion-scale page graph in Table II. 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 [25], which has a billion vertices and 127 billion edges, and Pregel processed it on 300 multicore machines. In contrast, we process the page graph on a single multicore machine.

Table V shows that FlashGraph achieves good performance in breadth-first search on this billion-node graph. It takes only eight minutes with a cache size of 4GB and less than six minutes with a cache size of 256 GB. This indicates that FlashGraph traverses over five million vertices per second on the page graph, which is much higher than the maximal random I/O performance (740,000 IOPS) provided by the SSD array. In contrast, Pregel [25] 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 [29] took over ten minutes to perform breadth-first search on a graph of one billion vertices and 13 billion edges on 14 12-core machines.

As shown in Table V, FlashGraph can perform all of our applications within a reasonable amount of time and with relatively small memory footprint. The small memory footprint suggests that FlashGraph is able to process much larger graphs on a single machine with half a terabyte of RAM.

We further demonstrate that FlashGraph is able to process graphs with billions of vertices and hundreds of billions of edges on a single machine. All of our graph applications use a reasonable amount of memory and likewise finish within a reasonable amount of time.

**VI. Conclusions**

We present a semi-external memory graph engine called FlashGraph that closely integrates with an SSD filesystem to achieve maximal performance. It uses the asynchronous user-task I/O interface to reduce overhead of accessing data in the filesystem and overlap computation with I/O. The graph engine partitions a graph and assigns a thread to each partition to process vertices independently, thus localizing computation to the local memory in a non-uniform memory architecture. The graph engine schedules the order of processing vertices to maximize the page cache hit rate and generate larger I/O requests. With careful design and implementation, we demonstrate that a semi-external memory graph engine can achieve performance comparable to in-memory graph engines.

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 vertex in the graph by sending messages as well as reading data directly. When using message passing, FlashGraph localizes user computation to the local memory and provides a race free guarantee on the user computation of a single vertex.

Unlike other external-memory graph engines such as GraphChi and X-stream, FlashGraph supports random and selective access to vertices. 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 CPU 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 machine. All of our graph applications use a reasonable amount of memory and likewise finish within a reasonable amount of time.

**Acknowledgments**

We would like to thank Carey E. Priebe, Joshua T. Vogelstein and Heng Wang for their many insightful discussions regarding the applications.

**References**

[1] J. Abello, A. L. Buchbaum, and J. R. Westbrook. A functional approach to external graph algorithms. In Algorithmica, pages 332–343, Springer-Verlag, 1998.

[2] G. A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. PhD thesis, Massachusetts Institute of Technology, 1985.
[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skoumbakis. Weaving relations for cache performance. In Proceedings of the 27th International Conference on Very Large Data Bases, 2001.

[4] J. N. Bertsekas, Dimitri P.; Tsitsiklis. In Parallel and Distributed Computation: Numerical Methods. Prentice Hall, Inc., 1989.

[5] Boost graph library. http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html Accessed 4/9/2014.

[6] V. D. Blondel, J. Loup Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks, 2008.

[7] U. Brandes. A faster algorithm for betweenness centrality. Journal of Mathematical Sociology, 25:163–177, 2001.

[8] 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.

[9] 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. Technical report, Microsoft Research, 2010.

[10] G. Csardi and T. Nepusz. The igraph software package for complex network research. InterJournal, Complex Systems:1695, 2006.

[11] 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, 2004.

[12] Apache giraph. https://giraph.apache.org/ Accessed 4/9/2014.

[13] 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.

[14] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In In Parallel Object-Oriented Scientific Computing (POOSEC), 2005.

[15] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2013.

[16] D. M. Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Citeseer, 1991.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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, 6(1):29–123, 2009.

[22] 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.

[23] A. Lugowskii, D. Alber, A. Bulu, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis, chapter 79, pages 930–941.

[24] C. Magnien, M. Latapy, and M. Habib. Fast computation of empirically tight bounds for the diameter of massive graphs. J. Exp. Algorithmics, 13:10.1.10–10.1.9, Feb. 2009.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O’Neil, P. O’Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A column-oriented DBMS. In Proceedings of the 31st International Conference on Very Large Data Bases, 2005.

[32] 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, 62(3), 2014.

[33] D. J. Watts and S. H. Strogatz. Collective dynamics of ‘small-world’ networks. Nature, 393(440–442), 1998.

[34] Web graph. http://webdatacommons.org/hyperlinkgraph/ Accessed 4/18/2014.

[35] D. Zheng, R. Burns, and A. S. Szalay. Toward millions of file system IOPS on low-cost, commodity hardware. In Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, 2013.

[36] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Carnegie Mellon University, 2002.