MOSAIC: Processing a Trillion-Edge Graph on a Single Machine

Steffen Maass Changwoo Min Sanidhya Kashyap Woonhak Kang Mohan Kumar Taesoo Kim

Georgia Institute of Technology

Abstract

Processing a one trillion-edge graph has recently been demonstrated by distributed graph engines running on clusters of tens to hundreds of nodes. In this paper, we employ a single heterogeneous machine with fast storage media (e.g., NVMe SSD) and massively parallel coprocessors (e.g., Xeon Phi) to reach similar dimensions. By fully exploiting the heterogeneous devices, we design a new graph processing engine, named MOSAIC, for a single machine. We propose a new locality-optimizing, space-efficient graph representation—*Hilbert-ordered tiles*, and a *hybrid execution model* that enables vertex-centric operations in fast host processors and edge-centric operations in massively parallel coprocessors.

Our evaluation shows that for smaller graphs, MOSAIC consistently outperforms other state-of-the-art out-of-core engines by 3.2–58.6× and shows comparable performance to distributed graph engines. Furthermore, MOSAIC can complete one iteration of the Pagerank algorithm on a trillion-edge graph in 21 minutes, outperforming a distributed disk-based engine by 9.2×.

1. Introduction

Graphs are the basic building blocks to solve various problems ranging from data mining, machine learning, scientific computing to social networks and the world wide web. However, with the advent of Big Data, the sheer increase in size of the datasets [11] poses fundamental challenges to existing graph processing engines. To tackle this issue, researchers are focusing on distributed graph processing engines like GraM [60], Chaos [53] and, Giraph [11] to process unprecedented, large graphs.

To achieve scalability with an increasing number of computing nodes, the distributed systems typically require very fast interconnects to cluster dozens or even hundreds of machines (e.g., 56 Gb Infiniband in GraM [60] and 40 GbE in Chaos [53]). This distributed approach, however, requires a costly investment of a large number of performant servers and their interconnects. More fundamentally though distributed engines have to overcome the straggler problem [53] and fault tolerance [2, 38] due to the imbalanced workloads on sluggish, faulty machines. These challenges often result in complex designs of graph processing engines that incur non-negligible performance overhead (e.g., two-phase protocol [53] or distributed locking [37]).

Another promising approach are single machine graph processing engines which are cost effective while lowering the entrance barrier for large-scale graphs processing. Similar to distributed engines, the design for a single machine either focuses on scaling out the capacity, via secondary storage–so-called *out-of-core graph analytics* [18, 33, 36, 52, 65, 66], or on scaling up the processing performance, by exploiting memory locality of high-end machines termed *in-memory graph analytics* [48, 55, 63]. Unfortunately, the design principles of out-of-core and in-memory engines for a single machine are hardly compatible in terms of performance and capacity, as both have contradicting optimization goals toward scalability and performance due to their use of differently constrained hardware resources (see Table 1).

To achieve the best of both worlds, in terms of capacity and performance, we divide the components of a graph processing engine explicitly for *scale-up* and *scale-out* goals. Specifically, we assign concentrated, memory-intensive operations (i.e., vertex-centric operations on a global graph) to fast host processors (scale-up) and offload the compute and I/O intensive components (i.e., edge-centric operations on local graphs) to coprocessors (scale-out).

Following this principle, we implemented MOSAIC, a graph processing engine that enables graph analytics on *one trillion edges* in a single machine. It shows superior performance compared to current single-machine, out-of-core processing engines on smaller graphs and shows even comparable performance on larger graphs, outperforming a distributed disk-based engine [53] by 9.2×, while only being 8.8× slower than a distributed in-memory one [60] on a trillion-edge dataset. MOSAIC exploits various recent technical developments, encompassing new devices on the PCIe bus. On one hand, MOSAIC relies on accelerators, such as Xeon Phis, to speed up the edge processing. On the other

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064191

	Single machine [36, 52, 55, 63]		GPGPU [31, 33, 64]		MOSAIC	Clusters [17, 43, 53, 60]	
Data storage	**In-memory**	**Out-of-core**	**In-memory**	**Out-of-core**	**Out-of-core**	**In-memory**	**Out-of-core**
Intended scale (#edges)	1–4 B	5–200 B	0.1–4 B	4–64 B	1 T	5–1000 B	> 1 T
Performance (#edges/s)	1–2 B	20–100 M	1–7 B	0.4 B	1–3 B	1–7 B	70 M
Performance bottleneck	CPU/Memory	CPU/Disk	PCIe	NVMe	NVMe	Network	Disk/Network
Scalability bottleneck	Memory size	Disk size	Memory size	Disk size	Disk size	Memory size	Disk size
Optimization goals	NUMA-aware memory access	I/O bandwidth	Massive parallelism	PCIe bandwidth	Locality across host and coprocessors	Load balancing & Network bandwidth	
Cost	Medium	Low	Low	Low	Low	High	Medium

Table 1: Landscape of current approaches in graph processing engines, using numbers from published papers for judging their intended scale and performance. Each approach has a unique set of goals (e.g., dataset scalability) and purposes (e.g., a single machine or clusters). In MOSAIC, we aim to achieve the cost effectiveness and ease-of-use provided by a single machine approach, while at the same time providing comparable performance and scalability to clusters by utilizing modern hardware developments, such as faster storage devices (e.g., NVMe) and massively parallel coprocessors (e.g., Xeon Phi).

hand, MOSAIC exploits a set of NVMe devices that allow terabytes of storage with up to 10× throughput than SSDs.

We would like to emphasize that existing out-of-core engines cannot directly improve their performance without a serious redesign. For example, GraphChi [36] improves the performance only by 2–3% when switched from SSDs to NVMe devices or even RAM disks. This is a similar observation made by other researchers [65, 66].

Furthermore, both single node and distributed engines have the inherent problem of handling *large datasets* (>8TB for 1 trillion edges), *low locality* and *load-balancing* issues due to skewed graph structures. To tackle these, we propose a new data structure, *Hilbert-ordered tiles*, an independent processing unit of batched edges (i.e., local graphs) that allows scalable, large-scale graph processing with high locality and good compression, yielding a simple load-balancing scheme. This data structure enables the following benefits: for coprocessors, it enables 1) better cache locality during edge-centric operations and 2) I/O concurrency through prefetching; for host processors, it allows for 1) sequential disk accesses that are small enough to circumvent load-balancing issues and 2) cache locality during vertex-centric operations.

In this paper, we make the following contributions:

- We present a trillion-scale graph engine on a single, heterogeneous machine, called MOSAIC, using a set of NVMes and Xeon Phis. For example, MOSAIC outperforms other state-of-the-art out-of-core engines by 3.2–58.6× for smaller datasets up to 4 billion edges. Furthermore, MOSAIC runs an iteration of the Pagerank algorithm with one trillion edges in 21 minutes (compared to 3.8 hours for Chaos [53]) using a single machine.
- We design a new data structure, Hilbert-ordered tiles, for locality, load balancing, and compression, that yields a compact representation of the graph, saving up to 68.8% on real-world datasets.
- We propose a hybrid computation and execution model that efficiently executes both vertex-centric operations (on host processors) and edge-centric operations (on coprocessors) in a scalable fashion. We implemented seven graph algorithms on this model, and evaluated them on two different single machine configurations.

We first present the challenges in processing current large-scale graphs in §2 and give a brief background of the technology trends underlying MOSAIC's design in §3. Then, we describe the design of MOSAIC in detail in §4, and its execution model in §5. §6 describes the implementation detail and §7 shows our implemented graph algorithms. §8 shows our evaluation results. §9 discusses, §10 compares MOSAIC with other approaches, and finally, §11 provides the conclusion.

2. Trillion Edge Challenges

With the inception of the internet, large-scale graphs comprising web graphs or social networks have become common. For example, Facebook recently reported their largest social graph comprises 1.4 billion vertices and 1 trillion edges. To process such graphs, they ran a distributed graph processing engine, Giraph [11], on 200 machines. But, with MOSAIC, we are able to process large graphs, even proportional to Facebook's graph, on a single machine. However, there is a set of nontrivial challenges that we have to overcome to enable this scale of efficient graph analytics on a single machine:

Locality. One fundamental challenge of graph processing is achieving locality as real world graphs are highly skewed, often following a powerlaw distribution [14]. Using a traditional, vertex-centric approach yields a natural API for graph algorithms. But, achieving locality can be difficult as, traditionally, the vertex-centric approach uses an index to locate outgoing edges. Though accesses to the index itself are sequential, the indirection through the index results in many random accesses to the vertices connected via the outgoing edges [20, 39, 50].

To mitigate random accesses to the edges, an edge-centric approach streams through the edges with perfect locality [52]. But, this representation still incurs low locality on vertex sets.

To overcome the issue of non-local vertex accesses, the edges can be traversed in an order that preserves vertex locality using, for example, the Hilbert order in COST [44] using delta encoding. However, the construction of the compressed Hilbert-ordered edges requires one global sorting step and a decompression phase during runtime.

MOSAIC takes input from all three strategies and mainly adopts the idea of using the Hilbert order to preserve locality between *batches of local graphs, the tiles* (see sections §4.1, §4.2).

Load balancing. The skewness of real-world graphs presents another challenge to load balancing. Optimal graph partitioning is an NP-complete problem [15], so in practice, a traditional hash-based partitioning has been used [40], but it still requires dynamic, proactive load-balancing schemes like work stealing [53]. MOSAIC is designed to balance workloads with a simple and independent scheme, balancing the workload between and within accelerators (see §5.5).

I/O bandwidth. The input data size for current large-scale graphs typically reaches multiple terabytes in a conventional format. For example, GraM [60] used 9 TB to store 1.2 T edges in the CSR format. Storing this amount of data on a single machine, considering the I/O bandwidth needed by graph processing engines, is made possible with large SSDs or NVMe devices.

Thanks to PCIe-attached NVMe devices, we can now drive nearly a million IOPS per device with high bandwidth (e.g., 2.5 GB/sec, Intel SSD 750 [22]). Now, the practical challenge is to exhaust this available bandwidth, a design goal of MOSAIC. MOSAIC uses a set of NVMe devices and accelerates I/O using coprocessors (see §4.3).

3. Background

We provide a short background on the hardware trends MOSAIC is designed to take into account.

Non-uniform memory access (NUMA). Modern architectures for high-performance servers commonly include multiple processors as well as memory on separate sockets, connected by a high-bandwidth on-chip interconnect (e.g. Intel QuickPath Interconnect (QPI) [25]). In such an architecture, the cost of accessing memory on a remote socket is potentially much higher than the local memory, thus coining the term non-uniform memory access (NUMA) while a single socket is sometimes referred to as a *domain*. MOSAIC optimizes for such a NUMA architecture with its striped partitioning to enable balanced accesses to multiple NUMA domains.

Non-volatile memory express (NVMe). NVMe is the interface specification [1] to allow high-bandwidth, low-latency access to SSD devices connected to the PCIe bus. In this paper, we refer to these SSD devices as *NVMe*s. These devices allow much improved throughput and higher IOPS than conventional SSDs on the SATA interface and currently reach up to 5 GB/s and 850K IOPS (Intel DC P3608 [23]). MOSAIC makes extensive use of these devices by exploiting their ability to directly copy data from the NVMe to e.g. a coprocessor, as well as serving hundreds of requests at the same time.

Intel Xeon Phi. The Xeon Phi is a massively parallel coprocessor by Intel [21]. This coprocessor has (in the first generation, *Knights Corner*) up to 61 cores with 4 hardware threads each and a 512-bit single instruction, multiple data (SIMD) unit per core. Each core runs at around 1 GHz (1.24 GHz for the Xeon Phi 7120A, used in MOSAIC). Each core has access to a shared L2 cache of 512 KB per core (e.g. 30.5 MB for 61 cores). MOSAIC uses the Xeon Phi for massively parallel operations and optimizes for the small amount of L2 cache by keeping the vertex state per subgraph bounded to this small amount of cache (512 KB per core). Furthermore, MOSAIC uses the many-core aspect to lauch many subgraph-centric computations in parallel.

4. The MOSAIC Engine

Overview. We adopt the "think-like-a-vertex" abstraction that allows for the implementation of most popular algorithms for graph processing. It uses the edges of a graph to transport intermediate updates from a source vertex to a target vertex during the execution of graph algorithms. Each vertex is identified by a 32-bit or 64-bit integer based on the scale of the graph and has associated meta information (e.g., in and out degree) as well as algorithm-dependent states (e.g., a *current* and a *next* value per vertex in the Pagerank algorithm).

MOSAIC extends this abstraction to a heterogeneous architecture, enabling tera-scale graph analytics on a single machine. In particular, it uses multiple coprocessors (i.e., Xeon Phis) to perform computation-heavy edge processing as well as I/O operations from NVMe devices by exploiting the large number of cores provided by each coprocessor. At the same time, MOSAIC dedicates host processors with faster single-core performance to synchronous tasks: vertex-centric operations (e.g., reduce) and orchestration of all components on each iteration of the graph algorithms; Figure 3 gives an overview of the interaction of the components of MOSAIC.

In this section, we first introduce the core data structure, called *tiles* (§4.1), and their ordering scheme for locality (§4.2), and explain each component of MOSAIC in detail (§4.3).

4.1 Tile: Local Graph Processing Unit

In MOSAIC, a graph is broken down into disjoint sets of edges, called *tiles*, each of which represents a subgraph of the graph. Figure 1 gives an example of the construction of the tiles and their corresponding meta structures. The advantage of the tile abstraction is two-fold: 1) each tile is an independent unit of edge processing—thus the name *local graph*—which does not require a global, shared state during execution, and 2) tiles can be structured to have an inherent locality for memory writes by sorting local edges according to their target vertex. Tiles can be evenly distributed to each coprocessor through simple round-robin scheduling, enabling a simple first-level load-balancing scheme. Furthermore, all tiles can be enumerated by following the Hilbert order for better locality (§4.2).

Inside a tile, the *number of unique vertices is bounded* by I_{max}, which allows the usage of local identifiers ranging from 0 to I_{max} (i.e., mapping from 4-8 bytes to 2 bytes with $I_{max} = 2^{16}$). MOSAIC maintains per-tile meta index

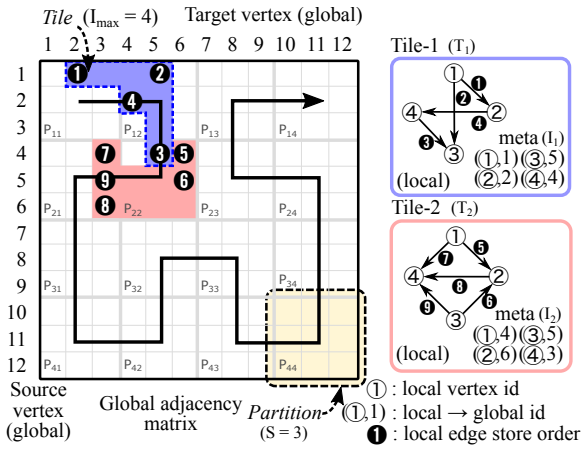


Figure 1: The Hilbert-ordered tiling and the data format of a tile in MOSAIC. There are two tiles (blue and red regions) illustrated by following the order of the Hilbert curve (arrow). Internally, each tile encodes a local graph using local, short vertex identifiers. The meta index I_j translates between the local and global vertex identifiers. For example, edge ❷, linking vertex 1 to 5, is sorted into tile T_1, locally mapping the vertices 1 to ① and 5 to ③.

structures to map a local vertex identifier of the subgraph (2 bytes) to its global identifier in the original graph (4-8 bytes). This yields a compact representation and is favorable to fit into the last level cache (LLC), for example with floats the vertex states per tile amount to $2^{16} * 4$ bytes $= 256$ KB. This easily fits into the LLC of the Xeon Phi (512 KB per core). Note that, even with the number of unique vertices in a tile being fixed, the number of edges per tile *varies*, resulting in varying tile sizes. The static load balancing scheme is able to achieve a mostly balanced workload among multiple coprocessors (see Figure 9, $< 2.3\%$ of imbalance).

Data format. More concretely, the format of a tile is shown in Figure 2 and comprises the following two elements:

- The index of each tile, stored as meta data. The index is an array that maps a local vertex identifier (the index of the array) to the global vertex identifier, which translates to 4 or 8 bytes, depending on the size of a graph.
- The set of edges, sorted by target vertices (tagged with weights for weighted graphs). The edges are stored either as an *edge list* or in a *compressed sparse rows (CSR)* representation, using 2 bytes per vertex identifier. MOSAIC switches between either representation based on which results in a smaller overall tile size. The CSR representation is chosen if the number of target vertices is larger than twice the number of edges. This amortizes storing the offset in the CSR representation.

Locality. Two dimensions of locality are maintained inside a tile; 1) sequential accesses to the edges in a local graph and 2) write locality enabled by storing edges in sorted order according to their target vertices. In other words, linearly enumerating the edges in a tile is favorable to prefetching memory while updating the vertex state array (i.e., an intermediate computation result) in the order of the target

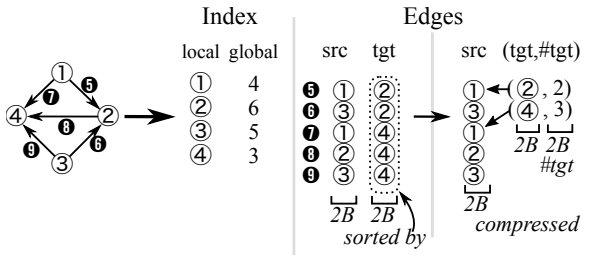


Figure 2: An overview of the on-disk data structure of MOSAIC, with both index and edge data structures, shown for tile T_2 (see Figure 1). The effectiveness of compressing the target-vertex array is shown, reducing the number of bytes by 20% in this simple example.

vertex enables memory and cache locality due to repeated target vertices in a tile. For example, in Figure 1, the local vertex ④ in tile T_2 is the target of all three consecutive edges ❼, ❽, and ❾.

Conversion. To populate the tile structure, we take a stream of partitions as an input, and statically divide the adjacency matrix representation of the global graph (Figure 1) into partitions of size $S \times S$, where $S = 2^{16}$.

We consume partitions following the Hilbert order (§4.2) and add as many edges as possible into a tile until its index structure reaches the maximum capacity (I_{max}) to fully utilize the vertex identifier (i.e., 2 bytes) in a local graph.

An example of this conversion is given in Figure 1, using the parameters $I_{max} = 4$ and $S = 3$: After adding edges ❶ through ❹ (following the Hilbert order of partitions: $P_{11}, P_{12}, P_{22}, \ldots$), there are no other edges which could be added to the existing four edges without overflowing the local vertex identifiers. Thus, tile T_1 is completed and tile T_2 gets constructed with the edges ❺ through ❾, continuing to follow the Hilbert order of partitions. This conversion scheme is an embarrassingly parallel task, implementable by a simple sharding of the edge set for a parallel conversion. It uses one streaming step over all partitions in the Hilbert order and constructs the localized CSR representations.

Compared to other, popular representations, the overhead is low. Like the CSR representation, the Hilbert-ordered tiles also require only one streaming step of the edge set. In comparison to Hilbert-ordered edges [44], the Hilbert-ordered tiles save a global sorting step, only arrange the tiles, not the edges, into the global Hilbert order.

4.2 Hilbert-ordered Tiling

Although a tile has inherent locality in processing edges on coprocessors, another dimension of locality can also be achieved by traversing them in a certain order, known as the Hilbert order [19, 44]. In particular, this can be achieved by traversing the partitions ($P_{i,j}$) in the order defined by the Hilbert curve during the conversion. The host processors can preserve the locality of the global vertex array across sequences of tiles.

Hilbert curve. The aforementioned locality is a well-known property of the Hilbert curve, a continuous fractal space-

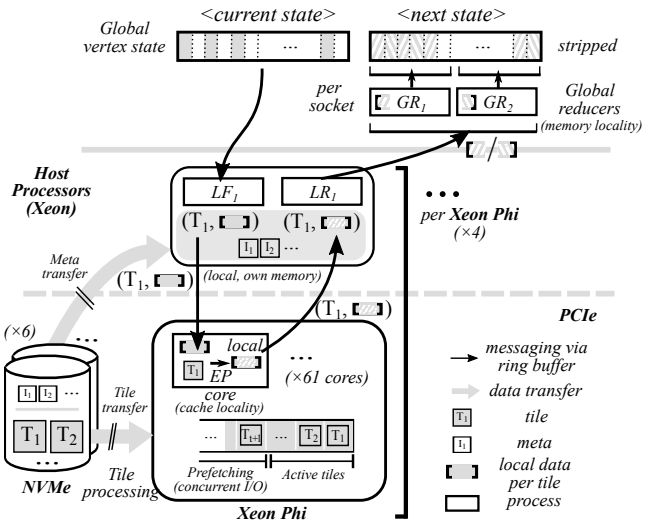


Figure 3: MOSAIC's components and the data flow between them. The components are split into *scale-out* for Xeon Phi (*local fetcher* (LF), *local reducer* (LR) and *edge processor* (EP)) and *scale-up* for host (*global reducer* (GR)). The *edge processor* runs on a Xeon Phi, operating on local graphs while host components operate on the global graph. The vertex state is available as both a read-only *current* state as well as a write-only *next* state.

filling curve that translates a pair of coordinates (i, j) (a partition id) to a scalar value d (the partition order), and vice versa. This operation preserves partial locality between neighboring scalar values (i.e., d), as they share parts of the respective coordinate sets. In MOSAIC, this allows close tiles in the Hilbert order (not limited to immediate neighbors) to share parts of their vertex sets.

Locality. MOSAIC processes multiple tiles in parallel on coprocessors: MOSAIC runs the edge processing on four Xeon Phis, each of which has 61 cores; thus 244 processing instances are running in parallel, interleaving neighboring tiles among each other following the Hilbert order of tiles. Due to this scale of concurrent accesses to tiles, the host processors are able to exploit the locality of the shared vertex states associated with the tiles currently being processed, keeping large parts of these states in the cache. For example, vertex 4 in Figure 1 is the common source vertex of three edges (i.e., ❸ , ❺ , and ❼) in T_1 and T_2, allowing locality between the subsequent accesses.

I/O prefetching. Traversing tiles in the Hilbert order is not only beneficial to the locality on the host, but also effective for prefetching tiles on coprocessors. While processing a tile, we can prefetch neighboring tiles from NVMe devices to memory by following the Hilbert-order in the background, which allows coprocessors to immediately start the tile processing as soon as the next vertex state array arrives.

4.3 System Components

From the perspective of components, MOSAIC is subdivided according to its scale-up and scale-out characteristics, as introduced in Figure 3. Components designed for scaling out are instantiated per Xeon Phi, allowing linear scaling when adding more pairs of Xeon Phis and NVMes. These components include the *local fetcher* (LF, fetches vertex information from the global array as input for the graph algorithm), the *edge processor* (EP, applies an algorithm-specific function per edge), and *local reducer* (LR, receives the vertex output from the *edge processor* to accumulate onto the global state). A global component, the *global reducer* (GR), is designed to take input from all *local reducers* to orchestrate the accumulation of vertex values in a lock-free, NUMA-aware manner.

Local fetcher (LF). Orchestrates the data flows of graph processing; given a tile, it uses the prefetched meta data (i.e., index) to retrieve the current vertex states from the vertex array on the host processor, and then feeds them to the *edge processor* on the coprocessor.

Edge processor (EP). The *edge processor* executes a function on each edge, specific to the graph algorithm being executed. Each *edge processor* runs on a core on a coprocessor and independently processes batches of tiles (streaming). Specifically, it prefetches the tiles directly from NVMe without any global coordination by following the Hilbert-order. It receives its input from the *local fetcher* and uses the vertex states along with the edges stored in the tiles to execute the graph algorithm on each edge in the tile. It sends an array of updated target vertex states back to the *local reducer* running on the host processor after processing the edges in a tile.

Local reducer (LR). Once the *edge processor* completes the local processing, the *local reducer* receives the computed responses from the coprocessors, aggregates them for batch processing, and then sends them back to *global reducers* for updating the vertex states for the next iteration. This design allows for large NUMA transfers to the *global reducers* running on each NUMA socket and avoids locks for accessing the global vertex state.

Global reducer (GR). Each *global reducer* is assigned a partition of the global vertex state and receives its input from the *local reducer* to update the global vertex state with the intermediate data generated by the graph algorithm on a local graph (i.e., tiles). As modern systems have multiple NUMA domains, MOSAIC assigns disjoint regions of the global vertex state array to dedicated cores running on each NUMA socket, allowing for large, concurrent NUMA transfers in accessing the global memory.

Striped partitioning. Unlike typical partitioning techniques, which assign a contiguous array of state data to a single NUMA domain, MOSAIC conducts *striped partitioning*, assigning "stripes" of vertices, interleaving the NUMA domains (as seen in Figure 3). This scheme exploits an inherent parallelism available in modern architectures (i.e., multiple sockets). Without the striped partitioning, a single core has to handle a burst of requests induced by the Hilbert-ordered

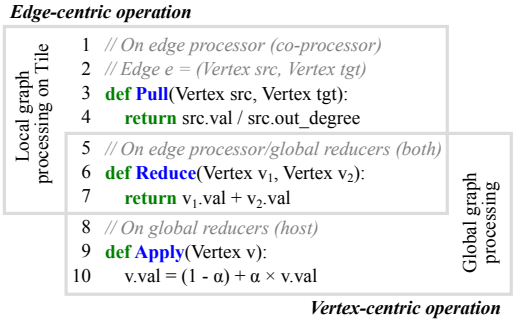


Figure 4: The Pagerank implementation on MOSAIC. *Pull()* operates on edges, returning the impact of the source vertex, *Reduce()* accumulates both local and global impacts, while *Apply()* applies the damping factor α to the vertices on each iteration.

tiles in a short execution window due to the inherent locality in the vertex states accessed.

In addition, the dedicated *global reducers*, which combine the local results with the global vertex array, can avoid global locking or atomic operations during the reduce operations, as each core has exclusive access to its set of vertex states.

At the end of a superstep, after processing all edges, MOSAIC swaps the *current* and *next* arrays.

5. The MOSAIC Execution Model

MOSAIC adopts the popular "think-like-a-vertex" programming model [37, 40], but slightly modifies it to fully exploit the massive parallelism provided by modern heterogeneous hardware. In the big picture, coprocessors perform edge processing on *local graphs* by using numerous, yet slower cores, while host processors reduce the computation result to their *global vertex states* by using few, yet faster cores. To exploit such parallelism, two key properties are required in MOSAIC's programming abstraction, namely commutativity and associativity [12, 17, 40]. This allows MOSAIC to schedule computation and reduce operations in any order.

Running example. In this section, we explain our approach by using the Pagerank algorithm (see Figure 4), which ranks vertices according to their impact to the overall graph, as a running example.

5.1 Programming Abstraction

MOSAIC provides an API similar to the popular Gather-Apply-Scatter (GAS) model [16, 37, 40]. The GAS model is extended as the Pull-Reduce-Apply (PRA) model, introducing a *reduce* operation to accommodate the heterogeneous architecture MOSAIC is running on. The fundamental APIs of the PRA model in MOSAIC for writing graph algorithms are as follows:

- **Pull(e)**: For every edge (u, v) (along with a weight in case of a weighted graph), *Pull(e)* computes the result of the edge e by applying an algorithm-specific function on the value of the source vertex u and the related data such as in- or out-degrees. For Pagerank, we first pull the impact of a source vertex (the state value divided by its out-degree) and then gather this result for the state of the target vertex by adding to the previous state.
- **Reduce(v1, v2)**: Given two values for the same vertex, *Reduce()* combines both results into a single output. This function is invoked by *edge processors* on coprocessors as well as *global reducers* on the host. It operates on the new value for the next iteration rather than the current value being used as an input to *Pull(e)*. For Pagerank, the reduce function simply adds both values, aggregating the impact on both vertices.
- **Apply(v)**: After reducing all local updates to the global array, *Apply()* runs on each vertex state in the array, essentially allowing the graph algorithm to perform *non-associative* operations. The *global reducers* on the host run this function at the end of each iteration. For Pagerank, this step normalizes (a factor α) the vertex state (the sum of all impacts on incoming vertices).

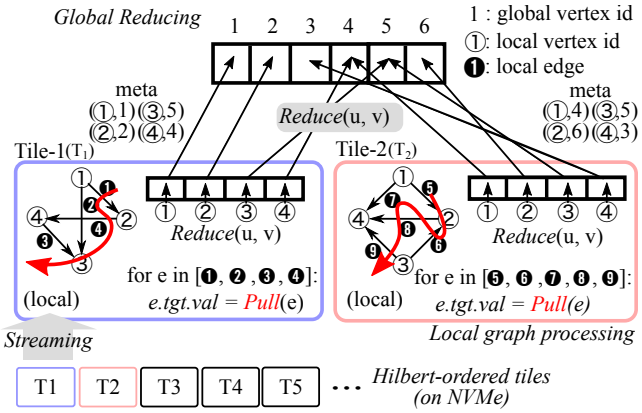


Figure 5: The execution model of MOSAIC: it runs *Pull()* on local graphs while *Reduce()* is being employed to merge vertex states, for both the local as well as the global graphs.

Figure 5 illustrates an execution using these APIs on two tiles and a global vertex array. When processing tiles in parallel, the result might overlap, but the tiles themselves can be processed independently of each other.

Generality. Our programming abstraction is general enough to express the most common graph algorithms, equivalent to the popular GAS abstraction. We implemented seven different algorithms as an example, ranging from graph traversal algorithms (Bread-First Search, Weakly Connected Components, Single-Source Shortest Path) to graph analytic algorithms (Pagerank, Bayesian Belief Propagation, Approximate Triangle Counting, Sparse Matrix-Vector Multiplication).

5.2 Hybrid Computation Model

The PRA model of MOSAIC is geared towards enabling a hybrid computation model: edge-centric operations (i.e., *Pull()*) are performed on coprocessors and vertex-centric operations (i.e., *Apply()*) on host processors. Aggregating intermediate results is done on both entities (i.e., *Reduce()*). This separation caters to the strengths of the specific entities: While host processors have faster single-core performance with larger caches, the number of cores is small. Thus, it

is suitable for the operations with considerably more cycles for execution (i.e., synchronous operations such as *Apply()*). On the contrary, the coprocessor has a larger number of cores, albeit with smaller caches and a lower clock speed, rendering it appropriate for massively parallel computation (i.e., processing edges). Our current implementation follows a synchronous update of the vertex states, but it would be straightforward to adopt an asynchronous update model with no changes in the current programming abstraction.

Edge-centric operations. In this hybrid computation model, coprocessors carry out the edge-centric operations; each core processes one tile at a time by executing *Pull()* on each edge, locally accumulating the results by using *Reduce()* to reduce the amount of data to be sent over PCIe, and sending the result back to *global reducer* on the host processor.

Vertex-centric operations. In MOSAIC, operations for vertices are executed on the host processor. MOSAIC updates the global vertex array via *Reduce()*, merging local and global vertex states. At the end of each iteration, *Apply()* allows the execution of non-associative operations to the global vertex array.

5.3 Streaming Model

In MOSAIC, by following the predetermined Hilbert-order in accessing graph data (i.e., tile as a unit), each component can achieve both sequential I/O by streaming tiles and meta data to proper coprocessors and host processors, as well as concurrent I/O by prefetching the neighboring tiles on the Hilbert curve. This streaming process is implemented using a message-passing abstraction for all communications between components without any explicit global coordination in MOSAIC.

5.4 Selective Scheduling

When graph algorithms require only part of the vertex set in each iteration (e.g., BFS), one effective optimization is to avoid the computation for vertices that are not activated. For MOSAIC, we avoid prefetching and processing of the tiles without active source vertices, reducing the total I/O amount. This leads to faster computation in general while still maintaining the opportunity for prefetching tiles.

5.5 Load Balancing

MOSAIC employs load balancing on two levels: 1) between Xeon Phis, 2) between threads on the same Xeon Phi. The first level is a static scheme balancing the number of tiles between all Xeon Phis. This macro-level scheme results in mostly equal partitions even though individual tiles may not all be equally sized. For example, even in our largest real-world dataset (hyperlink14), tiles are distributed to four Xeon Phis in balance (30.7–31.43 GB, see Figure 9).

On a second level, MOSAIC employs a dynamic load-balancing scheme between all *edge processors* on a Xeon Phi. Unfortunately, tiles are not equally sized (see Figure 9), resulting in varying processing times per tile. As MOSAIC prefetches and receives input from the host on the Xeon Phi into a circular buffer, one straggler in processing a tile might block all other *edge processors*. To avoid this, multiple cores are assigned to tiles with many edges. Each core is assigned a disjoint set of edges in a tile, to enable parallel processing without interactions between cores. Each core creates a separate output for the host to reduce onto the global array.

To decide the number of *edge processors* per tile, MOSAIC computes the number of partitions, $optPartitions$, per tile such that blocking does not occur. Intuitively, the processing time of any individual tile always has to be smaller than the processing time of all other tiles in the buffer to avoid head-of-line blocking. To determine the optimal number of partitions, MOSAIC uses a worst-case assumption that all other tiles in the buffer are minimally sized (i.e. contain 2^{16} edges). The resulting formula then simply depends on the processing rate ($\frac{edges}{second}$).

Specifically, MOSAIC uses the following calculation to determine $optPartitions$: Using the number of buffers $count_{buffers}$, the number of workers $count_{workers}$ and the rate at which edges can be processed both for small tiles $rate_{min}$ as well as for large tiles $rate_{max}$, the following formula determines $optPartitions$:

$$bestSplit = \frac{\frac{minEdges}{rate_{min}} * count_{buffers}}{count_{workers}} * rate_{max}$$

$$optPartitions = \left\lceil \frac{countEdges}{bestSplit} \right\rceil$$

The rate of processing is the only variable to be sampled at runtime, all other variables are constants.

5.6 Fault Tolerance

To handle fault tolerance, distributed systems typically use a synchronization protocol (e.g., two-phase commits) for consistent checkpointing of global states among multiple compute nodes. In MOSAIC, due to its single-machine design, handling fault tolerance is as simple as checkpointing the intermediate state data (i.e., vertex array). Further, the read-only vertex array for the current iteration can be written to disk parallel to the graph processing; it only requires a barrier on each superstep. Recovery is also trivial; processing can resume with the last checkpoint of the vertex array.

6. Implementation

We implemented MOSAIC in C++ in 16,855 lines of code. To efficiently fetch graph data on NVMe from Xeon Phi, we extended the 9p file system [4] and the NVMe device driver for direct data transfer between the NVMe and the Xeon Phi without host intervention. Once DMA channels are set up between NVMe and Xeon Phi's physical memory through the extended 9p commands, actual tile data transfer is performed by the DMA engines of the NVMe. To achieve higher throughput, MOSAIC batches the tile reading process and aligns the starting block address to 128KB in order to fully exploit NVMe's internal parallelism. In addition, for efficient messaging between the host and the Xeon Phi,

Algorithm	Lines of code	Reduce-operator	Complexity	
			Runtime I/O	Memory
PR	86	+	$O(E)$	$O(2V)$
BFS	102	min	$O(E^\star)$	$O(2V)$
WCC	88	min	$O(E^\star)$	$O(2V)$
SpMV	95	+	$O(E)$	$O(2V)$
TC	194	min, +	$O(2E)$	$O(8V)$
SSSP	91	min	$O(E^\star)$	$O(2V)$
BP	193	×, +	$O(2mE)$	$O(8mV)$

Table 2: Graph algorithms implemented on MOSAIC: associative and commutative operations used for the reducing phase, and their runtime complexity per iteration. E is the set of edges, V the set of vertices while $E^\star$ denotes the active edges that MOSAIC saves with selective scheduling. In BP, m denotes the number of possible states in a node.

MOSAIC switches between PIO and DMA modes depending on the message size. The messaging mechanism is exposed as a ring buffer for variable sized elements. Due to the longer setup time of the DMA mechanism, MOSAIC only uses DMA operations for requests larger than 32 KB. Also, to use the faster DMA engine of the host[1] and avoid costly remote memory accesses from the Xeon Phi, MOSAIC allocates memory for communication on the Xeon Phi side.

7. Graph Algorithms

We implement seven popular graph algorithms by using MOSAIC's programming model. Table 2 summarizes the algorithmic complexity (e.g., runtime I/O and memory overheads) of each algorithm.

Pagerank (PR) approximates the impact of a single vertex on the graph. MOSAIC uses a synchronous, push-based approach [52, 63].

Breadth-first search (BFS) calculates the minimal edge-hop distance from a source to all other vertices in the graph.

Weakly connected components (WCC) finds subsets of vertices connected by a directed path by iteratively propagating the connected components to its neighbors.

Sparse matrix-vector multiplication (SpMV) calculates the product of the sparse edge matrix with a dense vector of values per vertex.

Approximate triangle counting (TC) approximates the number of triangles in an unweighted graph. We extend the semi-streaming algorithm proposed by Becchetti et al. [3].

Single source shortest path (SSSP) operates on weighted graphs to find the shortest path between a single source and all other vertices. It uses a parallel version of the Bellman-Ford algorithm [5].

Bayesian belief propagation (BP) operates on a weighted graph and propagates probabilities at each vertex to its neighbors along the weighted edges [28].

8. Evaluation

We evaluate MOSAIC by addressing the following questions:

- **Performance**: How does MOSAIC perform with real-world and synthetic datasets, including a trillion-edge graph, compared to other graph processing engines? (§8.2, §8.3)
- **Design decisions**: What is the performance impact of each design decision made by MOSAIC, including the Hilbert-ordered tiles, selective-scheduling, fault-tolerance and the two-level load balancing scheme? (§8.4)
- **Scalability**: Does MOSAIC scale linearly with the increasing number of Xeon Phis and NVMes, and does each algorithm fully exploit the parallelism provided by Xeon Phis and the fast I/O provided by NVMes? (§8.5)

Type Nickname	Game PC (vortex)	Workstation (ramjet)
Model	E5-2699 v3	E5-2670 v3
CPU	2.30 GHz	2.30 GHz
# Core	18 × 1	12 × 2
RAM	64 GB	768 GB
LLC	45 MB × 1	30 MB × 2
PCIe	v3 (48L, 8 GT/s)	v3 (48L, 8 GT/s)
NVMe	1	6
Xeon Phi	1	4

Table 3: Two machine configurations represent both a consumer-grade gaming PC (vortex), and a workstation (ramjet, a main target for tera-scale graph processing).

8.1 Experiment Setup

To avoid optimizing MOSAIC for specific machine configurations or datasets, we validated it on two different classes of machines—a gaming PC and a workstation, using six different real-world and synthetic datasets, including a synthetic trillion-edge graph following the distribution of Facebook's social graph.

Machines. Table 3 shows the specifications of the two different machines, namely vortex (a gaming PC) and ramjet (a workstation). To show the cost benefits of our approach, we demonstrate graph analytics on 64 B edges (hyperlink14) on a gaming PC. To process one trillion edges, we use ramjet with four Xeon Phis and six NVMes.

Datasets. We synthesized graphs using the R-MAT generator [7], following the same configuration used by the graph500 benchmark.[2] The trillion-edge graph is synthesized using 2^{32} vertices and 1 T edges, following Facebook's reported graph distribution [11]. We also use two types of real-world datasets, social networks (twitter [35]) and web graphs (uk2007-05 [6, 49], and hyperlink14 [45]). These datasets contain a range of 1.5–64.4 B edges and 41.6–1,724.6 M vertices (35–37× ratio), with raw data of 10.9–480.0 GB (see Table 4). We convert each dataset to generate the tile structure, resulting in conversion times of 2 to 4 minutes for small datasets (up to 30 GB). Larger datasets finish in about 51 minutes (hyperlink14, 480 GB, 21 M edges/s) or about 30 hours for the trillion-edge graph (8,000 GB). In comparison, GridGraph takes 1 to 2 minutes to convert the smaller datasets into its internal representation. Furthermore, MOSAIC yields a more compact representation, e.g., MOSAIC fits the twitter

[1] Based on our measurements, a host-initiated DMA operation is nearly 2 times faster than a Xeon Phi-initiated one.

[2] Default parameters are $a = 0.57, b = 0.19, c = 0.19$, http://www.graph500.org/specifications

Graph	#vertices	#edges	Raw data	MOSAIC	
				Data size (reduction, bytes/edge)	Prep. time
⋆rmat24	16.8 M	0.3 B	2.0 GB	1.1 GB (-45.0%, 4.4)	2 m 10 s
twitter	41.6 M	1.5 B	10.9 GB	7.7 GB (-29.4%, 5.6)	2 m 24 s
⋆rmat27	134.2 M	2.1 B	16.0 GB	11.1 GB (-30.6%, 5.5)	3 m 31 s
uk2007-05	105.8 M	3.7 B	27.9 GB	8.7 GB (-68.8%, 2.5)	4 m 12 s
hyperlink14	1,724.6 M	64.4 B	480.0 GB	152.4 GB (-68.3%, 2.5)	50 m 55 s
⋆rmat-trillion	4,294.9 M	1,000.0 B	8,000.0 GB	4,816.7 GB (-39.8%, 5.2)	30 h 32 m

Table 4: The graph datasets used for MOSAIC's evaluation. The data size of MOSAIC represents the size of complete, self-contained information of each graph dataset, including tiles and meta-data generated in the conversion step. The ⋆ mark indicates synthetic datasets. Each dataset can be efficiently encoded with 29.4–68.8 % of its original size due to MOSAIC tile structure.

graph into 7.7 GB, while GridGraph's conversion is more than 4.3× larger at 33.6 GB.

Methodology. We compare MOSAIC to a number of different systems, running on a diverse set of architectures. Primarily, we compare MOSAIC with GraphChi [36], X-Stream [52] and GridGraph [66] as these systems focus on a single-machine environment using secondary storage. We run these systems on `ramjet` using all 6 NVMe in a RAID-0 setup, enabling these systems to take advantage of the faster storage hardware. Additionally, we use the published results of other graph engines, allowing a high-level comparison to MOSAIC, from a diverset set of architectures for graph engines: For single machines, we show the results for in-memory processing, with Polymer [63] and Ligra [55], and GPGPU, with TOTEM [64] and GTS [33]. Furthermore, we show the results for distributed systems, using the results for Chaos [53], on a 32-node cluster, as an out-of-core system, as well as GraphX [17] as an in-memory system. Furthermore, we include a special, pagerank-only in-memory system by McSherry et al [43] to serve as an estimation of a lower bound on processing time. Both GraphX and McSherry's system were run on the same 16-node cluster with a 10G network link [43].

With respect to datasets, we run other out-of-core engines for a single machine on the four smaller datasets (rmat24, twitter, rmat27, uk2007-05). We run both the Pagerank (PR) as well as the Weakly Connected Components (WCC) algorithm on the out-of core engines. These algorithms represent two different classes of graph algorithms: PR is an *iterative* algorithm, where in the initial dozens of iterations almost all vertices are active while WCC is a *traversal* algorithm with many vertices becoming inactive after the first few iterations. This allows a comparison on how well the processing system can handle both a balanced as well as an imbalanced number of active vertices.

I/O performance. Our ring buffer serves as a primitive transport interface for all communications. It can achieve 1.2 millions IOPS with 64 byte messages and 4 GB/sec throughput with larger messages between a Xeon Phi and a host (see Figure 6). We measure the throughput of random read operations when reading from a single file on an NVMe in four configurations: (a) a host application directly accesses

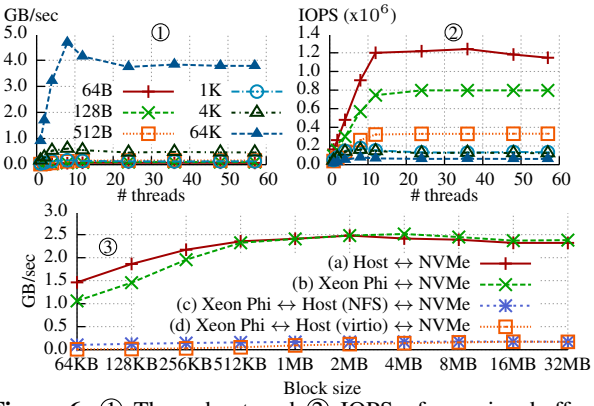


Figure 6: ① Throughput and ② IOPS of our ring buffer for various size messages between a Xeon Phi and its host, and ③ the throughput of random read operations on a file in an NVMe.

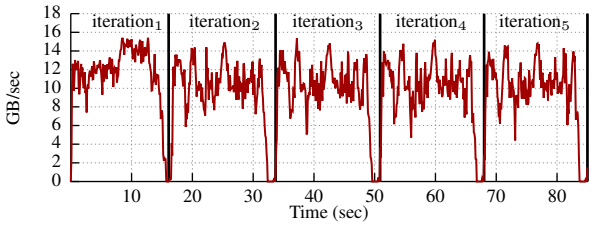


Figure 7: Aggregated I/O throughput for SpMV on the hyperlink14 graph for the first five iterations. The drop in throughput marks the end of an iterations while the maximum throughput of MOSAIC reaches up to 15 GB/sec.

the NVMe, showing the best performance, 2.5 GB/sec; (b) the Xeon Phi initiates the file operations and the data from the NVMe is directly delivered to the Xeon Phi using P2P DMA, 2.5 GB/sec. For comparison, we also evaluated two more scenarios: (c) the Xeon Phi initiates file operations, while the host delivers the data to the Xeon Phi through NFS; and (d) using `virtio` over PCIe, resulting in 10× slower performance.

8.2 Overall Performance

We ran seven graph algorithms (Table 2) on six different datasets (Table 4) with two different classes of single machines (Table 3). Table 5 shows our experimental results. In summary, MOSAIC shows 686–2,978 M edges/sec processing capability depending on datasets, which is even comparable to other *in-memory* engines (e.g., 695–1,390 M edges/sec in Polymer [63]) and distributed engines (e.g., 2,770–6,335 M edges/sec for McSherry et al.'s Pagerank-only in-memory cluster system [43]).

An example of the aggregated I/O throughput of MOSAIC with 6 NVMes is shown in Figure 7, with the hyperlink14 graph and the SpMV algorithm. The maximum throughput reaches up to 15 GB/sec, close to the maximum possible throughput per NVMe, highlighting MOSAIC's ability to saturate the available NVMe throughput.

Trillion-edge processing. MOSAIC on `ramjet` can perform out-of-core processing over a trillion-edge graph. We run all non-weighted algorithms (PR, BFS, WCC, SpMV, TC) on

Graph	#edges (ratio)	PR‡		BFS†		WCC†		SpMV‡		TC‡		SSSP†		BP‡	
		vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet	vortex	ramjet
⋆**rmat24**	0.3 B (1×)	0.72 s	0.31 s	18.04 s	3.52 s	18.51 s	3.92 s	0.58 s	0.30 s	2.06 s	1.46 s	45.32 s	11.71 s	1.09 s	0.47 s
twitter	1.5 B (5×)	4.99 s	1.87 s	51.65 s	11.20 s	59.46 s	18.58 s	4.59 s	1.66 s	13.90 s	9.42 s	269.99 s	54.06 s	7.78 s	5.34 s
⋆**rmat27**	2.1 B (8×)	6.28 s	3.06 s	71.79 s	17.02 s	78.07 s	22.18 s	6.02 s	2.74 s	22.10 s	16.52 s	353.75 s	101.95 s	10.64 s	8.42 s
uk2007-05	3.7 B (14×)	5.75 s	1.76 s	11.35 s	1.56 s	18.85 s	5.34 s	5.68 s	1.49 s	11.32 s	4.31 s	11.41 s	2.05 s	15.02 s	4.57 s
hyperlink14	64.4 B (240×)	100.85 s	21.62 s	15.61 s	6.55 s	2302.39 s	708.12 s	85.45 s	19.28 s	-	68.03 s	17.32 s	8.68 s	-	70.67 s
⋆**rmat-trillion**	1,000.0 B (3,726×)	-	1246.59 s	-	3941.50 s	-	7369.39 s	-	1210.67 s	-	5660.35 s	-	-	-	-

Table 5: The execution time for running graph algorithms on MOSAIC with real and synthetic (marked ⋆) datasets. We report the seconds per iteration for iterative algorithms (‡: PR, SpMV, TC, BP), while reporting the total time taken for traversal algorithms (†: BFS, WCC, SSSP). TC and BP need more than 64 GB of RAM for the hyperlink14 graph and thus do not run on `vortex`. We omit results for rmat-trillion on SSSP and BP as a weighted dataset of rmat-trillion would exceed 8 TB which currently cannot be stored in our hardware setup.

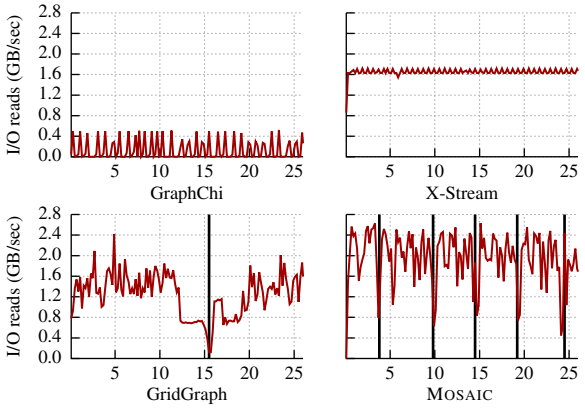


Figure 8: I/O throughput of out-of-core graph engines for 25 seconds of the Pagerank algorithm and the uk2007-05 dataset using a single NVMe on `ramjet`. The iteration boundaries are marked, neither GraphChi nor X-Stream finish the first iteration. GraphChi does not exploit the faster storage medium. X-Stream shows high throughput but reads 3.8× more per iteration than MOSAIC. GridGraph suffers from the total amount of data transfer and does not fully exploit the medium. MOSAIC with a single Xeon Phi and NVMe can drive 2.5 GB/s throughput.

this graph but exclude weighted algorithms due to storage space constraints of our experimental setup for the weighted dataset (>8 TB). The results show that MOSAIC can perform one iteration of Pagerank in 1,246 seconds (20.8 minutes), outperforming Chaos [53] by a factor of 9.2×.

8.3 Comparison With Other Systems

We compare MOSAIC with five different types of graph processing engines:

- Single machine, out-of-core: These systems include GraphChi [36], X-Stream [52], and GridGraph [66] and are close in nature to MOSAIC.
- Single machine, in-memory: Operating on a high-end server with lots of memory, we include Polymer [63] and Ligra [55].
- Single machine, GPGPU: As an upper bound on the computing power on a single machine, we include TOTEM [64] and GTS [33], running in the in-memory mode on two GPGPUs.
- Distributed, out-of-core: To scale to very large graphs, up to a trillion edges, Chaos [53] proposes a distributed, out-of-core disk-based system using 32 nodes.
- Distributed, in-memory: As an upper bound on the distributed computing power, we include GraphX [17] running on 16 nodes as well as a specialized, Pagerank-only system running on 16 nodes by McSherry et al [43].

We replicate the experiments for single machine, out-of-core engines by ourselves on `ramjet` with 6 NVMes in a RAID 0, but take experimental results for all other types of graph engines directly from the authors [17, 33, 43, 53, 55, 63], as we lack appropriate infrastructure for these systems.

The results of this comparison with Pagerank are shown in Table 6. We split our discussion of the comparison into the different types of graph engines mentioned:

Single machine, out-of-core. MOSAIC outperforms other out-of-core systems by far, up to 58.6× for GraphChi, 29.8× for X-Stream, and 8.4× for GridGraph, although they are all running on `ramjet` with six NVMes in a RAID 0. Figure 8 shows the I/O behavior of the other engines compared to MOSAIC over 25 seconds of Pagerank. MOSAIC finishes 5 iterations while both X-Stream and GraphChi do not finish any iteration in the same timeframe. GridGraph finishes one iteration and shows less throughput than MOSAIC even though its input data is 4.3× larger. Furthermore, compared to the other engines, MOSAIC's tiling structure not only reduces the total amount of I/O required for computation, but is also favorable to hardware caches; it results in a 33.30% cache miss rate on the host, 2.34% on the Xeon Phi, which is distinctively smaller than that of the other engines (41.92–80.25%), as shown in Table 7.

Compared with other out-of-core graph engines running the WCC algorithm to completion, MOSAIC shows up to 801× speedup while maintaining a minimum speedup of 1.4× on small graphs, as shown in Table 8. MOSAIC achieves this speedup due to its efficient selective scheduling scheme, saving I/O as well as computation, while the edge-centric model in X-Stream is unable to take advantage of the low number of active edges. GraphChi shows poor I/O throughput and is unable to skip I/O of inactive edges in a fine-grained manner. MOSAIC outperforms GridGraph due to reduced I/O and better cache locality.

Single machine, in-memory. MOSAIC shows performance close to other in-memory systems, only being slower by at most 1.8× than the fastest in-memory system, Polymer, while outperforming Ligra by up to 2×. Compared to these systems,

Dataset	Single machine								Distributed systems		
	Out-of-core				In-memory		GPGPU		Out-of-core	In-memory	
	MOSAIC	GraphChi †	X-Stream †	GridGraph †	Polymer	Ligra	TOTEM	GTS	$Chaos_{32}$	$GraphX_{16}$	$McSherry_{16}$
rmat24	0.31 s	14.86 s (47.9×)	4.36 s (14.1×)	1.12 s (3.6×)	0.37 s	0.25 s	-	-	-	-	-
twitter	1.87 s	65.81 s (35.2×)	19.57 s (10.5×)	5.99 s (3.2×)	1.06 s	2.91 s	0.56 s	0.72 s	-	12.2 s	0.53 s
rmat27	3.06 s	100.02 s (32.7×)	27.57 s (9.0×)	8.38 s (2.7×)	1.93 s	6.13 s	1.09 s	1.42 s	28.44 s	-	
uk2007-05	1.76 s	103.18 s (58.6×)	52.39 s (29.8×)	14.84 s (8.4×)	-	-	0.85 s	1.24 s	-	8.30 s	0.59 s

Table 6: The execution time for one iteration of Pagerank on out-of-core, in-memory engines and GPGPU systems running either on a single machine or on distributed systems (subscript indicates number of nodes). Note the results for other out-of-core engines (indicated by †) are conducted using six NVMes in a RAID 0 on `ramjet`. We take the numbers for the GPGPU (from [33]), in-memory systems and the distributed systems from the respective publications as an overview of different architectural choices. We include a specialized in-memory, cluster Pagerank system developed by McSherry et al. [43] as an upper bound comparison for in-memory, distributed processing and show the GraphX numbers on the same system for comparison. MOSAIC runs on `ramjet` with Xeon Phis and NVMes. MOSAIC outperforms the state-of-the-art out-of-core engines by 3.2–58.6× while showing comparable performance to GPGPU, in-memory and out-of-core distributed systems.

Graph Engine	LLC miss	IPC	CPU usage	I/O bandwidth	I/O amount
GraphChi	80.25%	0.28	2.10%	114.0 MB/s	16.03 GB
X-Stream	55.93%	0.91	40.6%	1,657.9 MB/s	46.98 GB
GridGraph	41.92%	1.16	45.08%	1,354.8 MB/s	55.32 GB
MOSAIC Host	33.30%	1.21	21.28%	2,027.3 MB/s	1.92 GB
MOSAIC Phi	2.34%	0.29	44.94%		10.17 GB

Table 7: Cache misses, IPC and I/O usages for out-of-core engines and MOSAIC for Pagerank on uk2007-05, running in `ramjet` with one NVMe. The Hilbert-ordered tiling in MOSAIC results in better cache footprint and small I/O amount.

Dataset	Single machine			
	MOSAIC	GraphChi	X-Stream	GridGraph
rmat24	3.92 s	172.079 s (43.9×)	26.91 s (6.9×)	5.65 s (1.4×)
twitter	18.58 s	1,134.12 s (61.0×)	436.34 s (23.5×)	46.19 s (2.5×)
rmat27	22.18 s	1,396.6 s (63.0×)	274.33 s (12.4×)	62.97 s (2.8×)
uk2007-05	5.34 s	4,012.48 s (751.4×)	4,277.60 s (801.0×)	71.13 s (13.3×)

Table 8: The execution time for WCC until completion on single machine out-of-core engines. GraphChi, X-Stream and GridGraph use six NVMes in a RAID 0 on `ramjet`. MOSAIC outperforms the state-of-the-art out-of-core engines by 1.4×–801×.

MOSAIC is able to scale to much bigger graph sizes due to its design as an out-of-core engine, but it can still show comparable performance.

Single machine, GPGPU. Compared to GPGPU systems, MOSAIC is slower by a factor of up to 3.3× compared against TOTEM, an in-memory system. In comparison, MOSAIC is able to scale to much larger graphs due to its out-of-core design. Compared to GTS in the in-memory mode, MOSAIC is 2.6×–1.4× slower. However, when running in an out-of-core mode, MOSAIC can achieve up to 2.9 B edges per second (hyperlink14), while GTS achieves less than 0.4 B edges per second as an artifact of its strategy for scalability being bound to the performance of a single GPGPU.

Distributed system, out-of-core. Compared with Chaos, an out-of-core distributed engine, MOSAIC shows a 9.3× speedup on the rmat27 dataset. Furthermore, MOSAIC outperforms Chaos by a factor of 9.2× on a trillion-edge graph. Chaos is bottlenecked by network bandwidth to the disks, while MOSAIC 1) reduces the necessary bandwidth due to its compact tile structure and 2) uses a faster interconnect, the internal PCIe bus.

Distributed system, in-memory. MOSAIC shows competitive performance to in-memory distributed systems, only being outperformed by up to 3.5× by a specialized cluster implementation of Pagerank [43] while outperforming GraphX by 4.7×–6.5×. Even though these systems might show better performance than MOSAIC, their distributed design is very costly compared to the single machine approach of MOSAIC. Furthermore, although these systems can theoretically support larger graphs – beyond a trillion edges – we believe there is an apparent challenge in overcoming the network bottleneck in bandwidth and speed to demonstrate this scale of graph processing in practice.

Impact of Preprocessing. We compare the impact of MOSAIC's preprocessing against other out-of-core single machine systems. GridGraph preprocesses the twitter graph in 48.6 s and the uk2007-05 graph in 2 m 14.7 s, thus MOSAIC outperforms GridGraph after 20 iterations of Pagerank for twitter and 8 iterations for the uk2007-05 graph. X-Stream does not have an explicit preprocessing phase, but is much slower per iteration than MOSAIC, thus MOSAIC is able to outperform it after 8 (twitter) and 5 (uk2007-05) iterations. This demonstrates that MOSAIC's optimizations are effective and show a quick return on investment, even though at the cost of a more sophisticated preprocessing step.

8.4 Evaluating Design Decisions

We perform experiments to show how our design decisions impact MOSAIC's performance.

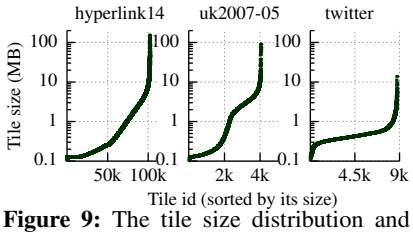
8.4.1 Hilbert-ordered Tiling

Tile encoding. MOSAIC's compact data format is key to reducing I/O during graph processing. The use of short (2 bytes), tile-local identifiers and the CSR representation of the local graph in a tile saves disk storage by 30.6–45.0% in synthetic datasets (4.4–5.5 byte/edge) and 29.4–68.8% in real-world datasets (2.5–5.6 byte/edge).

In quantity, it saves us 327 GB (-68.3%) in our largest real-world dataset (hyperlink14) and over 3,184 GB (-39.8%) in a

Traversal	#tiles	Pagerank		BFS		WCC	
		miss	time	miss	time	miss	time
Hilbert	8,720	33.30%	1.87 s	25.99%	11.20 s	26.12%	18.58 s
Row First	7,924	58.87%	1.97 s	46.33%	13.60 s	47.49%	19.03 s
Column First	10,006	34.38%	2.92 s	45.53%	18.78 s	45.83%	32.45 s

Table 9: Performance of different traversal strategies on Mosaic using the twitter graph. MOSAIC achieves a similar locality than the column first strategy which is focused on perfect writeback locality while providing better performance due to less tiles and better compression. MOSAIC shows up to 81.8% better cache locality on the host than either of the traversal strategies.



#Phi	Tiles data
1	30.98 GB
2	31.43 GB
3	30.85 GB
4	30.73 GB
Meta	28.81 GB

Figure 9: The tile size distribution and the total amount tiles allocated for each Xeon Phis for real-world datasets. The average tile size is 1.2 MB for hyperlink14, 1.9 MB for uk2007-05 and 1.1 MB for the twitter dataset. The tiles are evenly distributed among Xeon Phis.

partitions	PR	BFS	WCC	SpMV	TC
1	4.41 s	1.54 s	31.05 s	2.65 s	8.01 s
$optPartitions$	1.76 s	1.56 s	5.34 s	1.49 s	4.31 s
$optPartitions * 10$	2.74 s	1.58 s	5.35 s	2.19 s	15.65 s

Table 10: The effects of choosing different split points for tiles on the uk2007-05 graph. The optimal number of partitions, $optPartitions$, is calculated dynamically as described in §5.5 and improves the total running time by up to 5.8× by preventing starvation.

trillion-edge graph (rmat-trillion) (see Table 4), significantly reducing the I/O required at runtime.

Hilbert-ordering. MOSAIC achieves cache locality by following the Hilbert-order to traverse tiles. The impact of this order is being evaluated using different strategies to construct the input for MOSAIC. In particular, two more strategies are being evaluated: *Row First* and *Column First*. These strategies are named after the mechanism in which the adjacency matrix is being traversed, with the rows being the source and the columns being the target vertex sets. The *Row First* strategy obtains perfect locality in the source vertex set while not catering to the locality in the target set. The *Column First* strategy, similar to the strategy used in GridGraph, obtains perfect locality in the target vertex set while not catering to locality in the source set. The results with respect to locality and execution times are shown in Table 9. These results show that the Hilbert-ordered tiles obtain the best locality of all three strategies, with up to 81.8% better cache locality on the host, reducing the execution time by 2.4%-74.7%.

Static load balancing. The size of tiles varies although the number of unique vertices is fixed. The static load balancing is able to keep the data sizes between all Xeon Phis similar. In hyperlink14, 65% of the tiles are sized between 120 KB and 1 MB, with the largest tile being about 150 MB. However, this inequality of tile sizes does not result in issues with any imbalanced workloads between Xeon Phis. In MOSAIC, the tiles get distributed in a round-robin fashion to the multiple Xeon Phis, resulting in even distribution (e.g., less than 3% in hyperlink14) among Xeon Phis (see Figure 9).

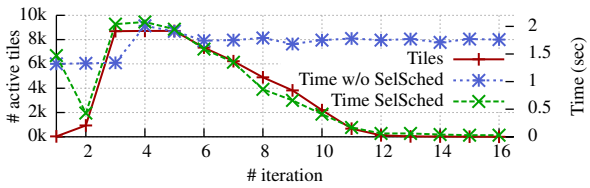


Figure 10: The number of active tiles per iteration and the execution time per iteration with and without the selective scheduling on twitter for BFS. It converges after 16 iterations and improves the performance by 2.2×.

Dynamic load balancing. The dynamic load-balancing mechanism for MOSAIC allows tiles to be split to avoid a head-of-line blocking situation to occur. The calculated number of tile partitions is $optPartitions$. Table 10 details the impact of an improperly chosen tile split point using the uk2007-05 graph. Disabling the dynamic load balancing increases the running time by up to 2.5× (Pagerank) and 5.8× (WCC). Similarly, dynamic load balancing with too many partitions ($optPartitions * 10$) results in degraded performance as well due to increased overhead from processing more partial results. Too many partitions result in overheads of 57% (Pagerank) and up to 3.6× (TC).

8.4.2 Global Reducer

MOSAIC uses global reducers (see §4.3) to 1) enable NUMA-aware memory accesses on host processors, yet fully exploit parallelism, and 2) avoid global synchronization in updating the global vertex array for reduce operations.

Striped partitioning. The impact of striped partitioning is significant: with two *global reducers* on `ramjet` (one per NUMA domain) with twitter, it increases the end-to-end performance by 32.6% compared to a usual partitioning.

Avoiding locks. The dedicated *global reducers* update the vertex values in a lock-free manner. This allows 1.9× end-to-end improvement over an atomic-based approach and a 12.2× improvement over a locking-based approach, with 223-way-hashed locks to avoid a global point of contention on the twitter graph.

8.4.3 Execution Strategies

Selective scheduling. MOSAIC keeps track of the set of active tiles (i.e., tiles with at least one active source vertex), and fetches only the respective, active tiles from disk. Figure 10 shows the number of active tiles in each iteration when running the BFS algorithm with twitter until convergence on `vortex`. It improves the overall performance by 2.2× (from 25.5 seconds to 11.2 seconds) and saves 59.1% of total I/O (from 141.6 GB to 57.9 GB).

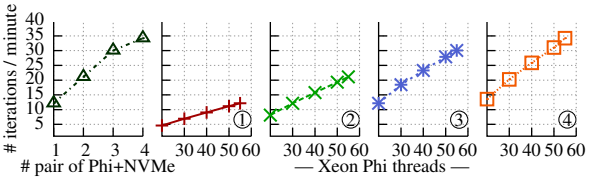


Figure 11: Time per iteration (a) with increasing pairs of a Xeon Phi and a NVMe (left one), and (b) with increasing core counts in multiple Xeon Phis from ① to ④. MOSAIC scales very well when increasing the number of threads and scales reasonably up to the fourth Xeon Phi, when NVMe I/O is starting to be saturated.

Fault tolerance. We implement fault tolerance by flushing the state of the vertices to disk in every superstep. As this operation can be overlapped with reading the states of the vertices for the next iteration, it imposes negligible performance overhead. For example, MOSAIC incurs less than 0.2% overhead in case of the Pagerank algorithm (i.e., flushing 402 MB in every iteration) with uk2007-05 and twitter on `ramjet`.

8.5 Scalability

When increasing the number of Xeon Phi and NVMe pairs from one to four, it scales the overall performance of Pagerank (uk2007-05) by 2.8× (see Figure 11). MOSAIC scales well up to the fourth Xeon Phi, when NVMe I/O starts to become saturated. The graphs on the right side (①–④) show in detail how an increasing number of cores of each Xeon Phi affects MOSAIC's performance, highlighting MOSAIC's scalability when adding more cores.

8.6 MOSAIC using the CPU only

To show the effectiveness of the techniques developed for MOSAIC, we compare the execution of MOSAIC on `ramjet` using the previous setup of four Xeon Phis and six NVMes with a CPU-only setup using only the six NVMes. The results of this setup are shown in Table 11 and show that the CPU-only setup is mostly competitive with the Xeon Phi-enabled setup. The CPU-only setup is at most 2.1× slower than the Xeon Phi-enabled setup while outperforming it by up to 5.5× for algorithms with very small data movements per iteration (such as SSSP and BFS), due to the CPU-only setup being able to move data in memory while the Xeon Phi-enabled setup has to copy data over the slower PCIe interface.

9. Discussion

Limitations. To scale MOSAIC beyond a trillion edges, a few practical challenges need to be addressed: 1) the throttled PCIe interconnect, 2) the number of attachable PCIe devices, 3) slow memory accesses in a Xeon Phi, and 4) the NVMe throughput. Next-generation hardware, such as PCIe-v4, Xeon Phi KNL [26], and Intel Optane NVMe [24], is expected to resolve 1), 3) and 4) in the near future, but to resolve 2), devices such as a PCIe switch [29] need to be further explored.

Cost effectiveness. In terms of absolute performance, we have shown that out-of-core processing of MOSAIC can be comparable to distributed engines [16, 17, 53, 60] in handling tera-scale graphs. Moreover, MOSAIC is an attractive solution in terms of cost effectiveness, contrary to the distributed systems requiring expensive interconnects like 10 GbE [43], 40 GbE [53] or InfiniBand [60], and huge amount of RAM. The costly components of MOSAIC are coprocessors and NVMes, not the interconnect among these. Their regular prices are around $750 (e.g., 1.2 TB Intel SSD 750) and around $549 (Xeon Phi 7120A, used in MOSAIC), respectively.

Conversion. In MOSAIC, we opt for an active conversion step, offline and *once per dataset*, to populate the tile abstraction from the raw graph data. In a real-world environment, these conversion steps can easily be integrated into an initial data collection step, amortizing the cost of creating partitions while only revealing the costs for populating tiles. As shown in §8.1, the conversion time is comparable to other systems such as GridGraph [66].

COST. Many large-scale graph systems fundamentally suffer from achieving their scalability at the expense of absolute performance. The COST [44] can be applied to MOSAIC to evaluate its inherent overhead for scalability: for uk2007-05, a single-threaded, host-only implementation (in-memory) [44] on `ramjet` took 8.06 seconds per iteration, while the *out-of-core* computation of MOSAIC on `ramjet` with one Xeon Phi/NVMe using 31 cores on the Xeon Phi matches this performance. At its maximum, MOSAIC is 4.6× faster than the single-threaded, in-memory implementation. Similarly, for the twitter graph, a single-threaded, host-only implementation spends 7.23 s per iteration, while MOSAIC matches this performance using one Xeon Phi with 18 cores with a maximum speedup of 3.86×.

10. Related work

The field of graph processing has seen efforts in a number of different directions, ranging from engines on single machines in-memory [30, 42, 48, 51, 55, 58, 63] to out-of-core engines [18, 34, 36, 52, 59, 65, 66] to clusters [10, 13, 16, 17, 27, 37–41, 46, 53, 60–62, 67], each addressing unique optimization goals and constraints (see Table 1).

Single machine vs. clusters. Out-of-core graph processing on a single machine was presented by GraphChi [36] and X-Stream [52] in a scale of a few billion edges. GridGraph [66] and FlashGraph [65] are both single-machine engines that are most similar in spirit to MOSAIC in terms of internal data structure and faster storage, but MOSAIC is by far faster (i.e., better locality and smaller amount of disk I/O) and more scalable (i.e., beyond a tera-scale graph) due to its novel internal data structure and the use of coprocessors.

Trillion-edge graph analytics has been demonstrated by Chaos [53] (out-of-core), GraM [60] (in-memory engine, RDMA), and Giraph [11] using over 200 servers. MOSAIC outperforms Chaos by a significant margin for trillion-edge

Graph	PR‡	BFS†	WCC†	SpMV‡	TC‡	SSSP†	BP‡
⋆**rmat24**	0.35 s (+13%)	2.33 s (-51%)	2.74 s (-43%)	0.24 s (-25%)	1.55 s (+6%)	9.7 s (-21%)	1.01 s (+115%)
twitter	2.04 s (+9%)	12.13 s (+8%)	16.26 s (-14%)	1.79 s (+8%)	10.75 s (+14%)	63.96 s (+18%)	7.94 s (+49%)
⋆**rmat27**	3.20 s (+5%)	23.68 s (+39%)	28.77 s (+30%)	3.16 s (+15%)	17.03 s (+3%)	117.20 s (+15%)	11.82 s (+40%)
uk2007-05	2.51 s (+43%)	0.44 s (-255%)	4.71 s (-13%)	2.36 s (+58%)	6.64 s (+54%)	0.37 s (-454%)	3.66 s (-25%)
hyperlink14	38.53 s (+78%)	5.03 s (-30%)	1007.56 s (+42%)	32.82 s (+70%)	110.85 s (+63%)	7.15 s (-21%)	65.10 s (-9%)
⋆**rmat-trillion**	1358.67 s (+9%)	6984.46 s (+77%)	8650.66 s (+17%)	1257.50 s (+4%)	6128.57 s (+8%)	-	-

Table 11: The execution times for running graph algorithms on MOSAIC with real and synthetic (marked ⋆) datasets, comparing the execution times on `ramjet` using the CPU only with the times report in Table 5 on `ramjet` with four Xeon Phis. We report the seconds per iteration for iterative algorithms (‡: PR, SpMV, TC, BP), while reporting the total time taken for traversal algorithms (†: BFS, WCC, SSSP). A red percentage indicates cases where the CPU-only execution ran slower compared to the Xeon Phi-enabled one while a green percentage indicates faster execution times.

processing, while having the added benefits of being a single machine engine, with a lower entrance barrier, easier maintenance (e.g., fault tolerance), and extensibility (see §9).

In comparison to G-Store [34], which aims at compressing *undirected* graphs with up to a trillion edges, MOSAIC is able to significantly reduce the size and increase the cache locality of *directed* graphs.

Using accelerators. A handful of researchers have tried using accelerators, especially GPGPUs [8, 9, 20, 31–33, 47, 54, 57], for graph processing due to their massive parallelism. In practice, however, these engines have to deal with the problem of large data exchange between the host and one or a group of GPGPUs to achieve better absolute performance. MOSAIC solves this problem by tunneling P2P DMA between Xeon Phi and NVMes, which is the dominant data exchange path in out-of-core graph analytics. In comparison to GTS [33], an out-of-core GPGPU graph engine, MOSAIC achieves better scalability in larger datasets due to its strategy of using local graphs, while the strategy for scalability (Strategy-S) in GTS is bound by the performance of a single GPGPU.

Graph representation. Real-world graph data tends to be skewed and highly compressible [49], so a few graph engines, such as LigraPlus [56], attempt to exploit this property. MOSAIC's tile structure provides a similar compression ratio while at the same time omitting the overhead from decoding the edge set at runtime.

11. Conclusion

We present MOSAIC, an out-of-core graph processing engine that scales up to one trillion edges by using a single, heterogeneous machine. MOSAIC opens a new milestone in processing a trillion-edge graph: 21 minutes for an iteration of Pagerank on a single machine. We propose two key ideas, namely Hilbert-ordered tiling and a hybrid execution model, to fully exploit the heterogeneity of modern hardware, such as NVMe devices and Xeon Phis. MOSAIC shows a 3.2–58.6× performance improvement over other state-of-the-art out-of-core engines, comparable to the performance of distributed graph engines, yet with significant cost benefits.

12. Acknowledgment

We thank the anonymous reviewers, and our shepherd, Marco Canini, for their helpful feedback. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851, ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

References

[1] NVM Express. http://www.nvmexpress.org/, 2017.

[2] Apache. Apache Giraph. http://giraph.apache.org/, 2011.

[3] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient Semi-streaming Algorithms for Local Triangle Counting in Massive Graphs. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '08, pages 16–24, 2008.

[4] Bell Labs. intro - introduction to the Plan 9 File Protocol, 9P. http://man.cat-v.org/plan_9/5/intro, 2016.

[5] R. Bellman. On a Routing Problem. Technical report, DTIC Document, 1956.

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In *Proceedings of the 20th International World Wide Web Conference (WWW)*, pages 587–596, Hyderabad, India, Apr. 2011.

[7] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446, Lake Buena Vista, FL, Apr 2004.

[8] S. Che. GasCL: A vertex-centric graph model for GPUs. In *Proceedings of High Performance Extreme Computing Conference (HPEC), 2014 IEEE*, pages 1–6, Sept 2014.

[9] L. Chen, X. Huo, B. Ren, S. Jain, and G. Agrawal. Efficient and Simplified Parallel Graph Processing over CPU and MIC. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 819–828, May 2015.

[10] R. Chen, J. Shi, Y. Chen, and H. Chen. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*, pages 1:1–1:15, Bordeaux, France, Apr. 2015.

[11] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One Trillion Edges: Graph Processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8

(12):1804–1815, Aug 2015.

[12] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, Nov. 2013.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[14] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 251–262, New York, NY, USA, 1999. ACM.

[15] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-complete Problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. ACM.

[16] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, Hollywood, CA, Oct. 2012.

[17] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 599–613, Broomfield, Colorado, Oct. 2014.

[18] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. TurboGraph: A Fast Parallel Graph Engine Handling Billion-scale Graphs in a Single PC. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, 2013.

[19] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38(3):459 – 460, 1891.

[20] S. Hong, T. Oguntebi, and K. Olukotun. Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88, 2011.

[21] Intel Corporation. Intel Xeon Phi Coprocessor 7120A. http://ark.intel.com/products/80555/Intel-Xeon-Phi-Coprocessor-7120A-16GB-1238-GHz-61-core, 2012.

[22] Intel Corporation. Intel SSD 750 Series, 1.2 TB. http://ark.intel.com/products/86741/Intel-SSD-750-Series-1_2TB-2_5in-PCIe-3_0-20nm-MLC, 2015.

[23] Intel Corporation. Intel SSD DC P3608 Series. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-p3608-series.html, 2015.

[24] Intel Corporation. Intel Optane NVMe. http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html, 2017.

[25] Intel Corporation. Intel QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, 2017.

[26] Intel Corporation. Intel Xeon Phi Knights Landing (KNL). http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-processor-product-brief.html, 2017.

[27] 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*, ICDM '09, pages 229–238. IEEE Computer Society, 2009.

[28] U. Kang, D. Chau, and C. Faloutsos. Inference of Beliefs on Billion-Scale Graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[29] P. Kennedy. Avago and PLX - Future of PCIe? http://www.servethehome.com/avago-plx-future-pcie, Aug 2015.

[30] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*, pages 169–182, Prague, Czech Republic, Apr. 2013.

[31] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM.

[32] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable SIMD-Efficient Graph Processing on GPUs. In *Proceedings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 39–50, Oct 2015.

[33] M.-S. Kim, K. An, H. Park, H. Seo, and J. Kim. GTS: A Fast and Scalable Graph Processing Method Based on Streaming Topology to GPUs. In *Proceedings of the 2016 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

[34] P. Kumar and H. H. Huang. G-store: High-performance Graph Store for Trillion-edge Processing. In *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*, pages 71:1–71:12, Salt Lake City, UT, Nov. 2016.

[35] 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 World Wide Web Conference (WWW)*, pages 591–600, Raleigh, NC, Apr. 2010.

[36] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 31–46, Hollywood, CA, Oct. 2012.

[37] Y. Low, J. E. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. GraphLab: A New Framework For Parallel Machine Learning. In *Proceedings of the Twenty-*

Sixth Conference on Uncertainty in Artificial Intelligence (UAI 2010), pages 340–349, Catalina Island, CA, July 2010.

[38] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, Aug. 2012.

[39] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[40] 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/PODS Conference*, pages 135–146, Indianapolis, IN, June 2010.

[41] J. Malicevic, A. Roy, and W. Zwaenepoel. Scale-up Graph Processing in the Cloud: Challenges and Solutions. In *Proceedings of the Fourth International Workshop on Cloud Data and Platforms*, CloudDP '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[42] J. Malicevic, S. Dulloor, N. Sundaram, N. Satish, J. Jackson, and W. Zwaenepoel. Exploiting NVM in Large-scale Graph Analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, pages 2:1–2:9, New York, NY, USA, 2015. ACM.

[43] F. McSherry and M. Schwarzkopf. The impact of fast networks on graph analytics. https://github.com/frankmcsherry/blog/blob/master/posts/2015-07-31.md, Jul 2015.

[44] F. McSherry, M. Isard, and D. G. Murray. Scalability! But at what COST? In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015.

[45] R. Meusel, O. Lehmberg, C. Bizer, and S. Vigna. Web Data Commons - Hyperlink Graphs. http://webdatacommons.org/hyperlinkgraph/, 2014.

[46] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 439–455, Farmington, PA, Nov. 2013.

[47] L. Nai, Y. Xia, I. G. Tanase, H. Kim, and C.-Y. Lin. GraphBIG: Understanding Graph Computing in the Context of Industrial Solutions. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, pages 69:1–69:12, Austin, TX, Nov. 2015.

[48] 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*, pages 456–471, Farmington, PA, Nov. 2013.

[49] Paolo Boldi and Sebastiano Vigna. The WebGraph Framework I: Compression Techniques. In *Proceedings of the 19th International World Wide Web Conference (WWW)*, pages 595–601, New York, NY, Apr. 2004.

[50] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11, New Orleans, LA, Nov. 2010.

[51] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing Large Graphs on Multi-cores with Graph Awareness. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 41–52, Boston, MA, June 2012.

[52] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, Farmington, PA, Nov. 2013.

[53] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out Graph Processing from Secondary Storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[54] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. GraphReduce: Processing Large-scale Graphs on Accelerator-based Systems. In *Proceedings of the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, pages 28:1–28:12, Austin, TX, Nov. 2015.

[55] J. Shun and G. E. Blelloch. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 135–146, Shenzhen, China, Feb. 2013.

[56] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *Proceedings of the 2015 Data Compression Conference*, DCC '15, pages 403–412, Washington, DC, USA, 2015. IEEE Computer Society.

[57] G. M. Slota, S. Rajamanickam, and K. Madduri. High-Performance Graph Analytics on Manycore Processors. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 17–27, May 2015.

[58] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey. GraphMat: High Performance Graph Analytics Made Productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, Jul 2015.

[59] K. Wang, G. Xu, Z. Su, and Y. D. Liu. GraphQ: Graph Query Processing with Abstraction Refinement: Scalable and Programmable Analytics over Very Large Graphs on a Single PC. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 387–401, Santa Clara, CA, July 2015.

[60] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. GraM: Scaling Graph Computation to the Trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*, Kohala Coast, Hawaii, Aug. 2015.

[61] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 194–204, San Francisco, CA, Feb. 2015.

[62] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, San Jose, CA, Apr. 2012.

[63] K. Zhang, R. Chen, and H. Chen. NUMA-aware Graph-structured Analytics. In *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 183–193, San Francisco, CA, Feb. 2015.

[64] T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, and X. Liang. Efficient graph computation on hybrid CPU and GPU systems. *The Journal of Supercomputing*, 71(4):1563–1586, 2015.

[65] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, Feb. 2015.

[66] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale Graph Processing on a Single Machine Using 2-level Hierarchical Partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, pages 375–386, Santa Clara, CA, July 2015.

[67] X. Zhu, W. Chen, W. Zheng, and X. Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 301–316, Savannah, GA, Nov. 2016.