SYSTEMS ABSTRACTIONS FOR BIG DATA PROCESSING ON A SINGLE MACHINE

A Dissertation
Presented to
The Academic Faculty

By

Steffen Maass

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2019

SYSTEMS ABSTRACTIONS FOR BIG DATA PROCESSING ON A SINGLE MACHINE

Approved by:

Professor Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology

Professor Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Professor Tushar Krishna
School of Electrical Engineering
Georgia Institute of Technology

Professor Willy Zwaenepoel
Faculty of Engineering and Information Technologies
University of Sydney

Date Approved: April 3, 2019

To Anouksha, Lily,

and my family.

ACKNOWLEDGEMENTS

To begin with, I would like to express my gratitude and thanks to my advisor, Prof. Taesoo Kim, who provided me with the guidance and support along my path towards a dissertation. He provided me the opportunity to explore and build systems and helped shape my projects with many, invaluable comments. I would also like to thank Prof. Tushar Krishna and Prof. Abhishek Bhattacharjee for their support and guidance during the LATR project. Furthermore, I would like to thank the members of my committee, Prof. Ada Gavrilovska, Prof. Kishore Ramachandran, and Prof. Willy Zwaenepoel for their support and comments that helped shape this dissertation.

A special thanks to Mohan Kumar who I did not only collaborate with on many of my projects but who was also become a dear friend. Additionally, I would like to thank my colleagues at the SSLab and Insu, Ming-Wei, Sanidhya, and Meng as well as our former post-docs, Woonhak, and Changwoo for all the discussions and the time spent together. I would also like to thank our Ph.D. coordinator, Prof. Venkat for his help in navigating various PhD-related administrative issues. In addition, I am also grateful to our support staff, Elizabeth Ndongi and Trinh Doan for their help in all the administrative work.

I would like to especially thank Prof. Kishore Ramachandran for not only serving on my thesis committee but also giving me the chance to study at Georgia Tech and in his lab during my time as a masters student, in cooperation with Prof. Kurt Rothermel at the University of Stuttgart.

Finally, I would like to thank my family for their support and patience along this journey. I am indebted to Anouksha for her patience, love, and unwavering support throughout my years at Georgia Tech.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Large-scale internet services, such as Facebook or Google, are using clusters of many servers for problems such as search, machine learning, and social networks. However, while it may be possible to apply the tools used at this scale to smaller, more common problems as well, this dissertation presents approaches to large-scale data processing on only a single machine. This approach has obvious cost benefits and lowers the barrier of entrance to large-scale data processing. This dissertation approaches this problem by redesigning applications to enable trillion-scale graph processing on a single machine while also enabling the processing of evolving, billion-scale graphs.

First, this dissertation presents a new out-of-core graph processing engine, called MOSAIC, for executing graph algorithms on trillion-scale datasets on a single machine. MOSAIC makes use of many-core processors and fast I/O devices coupled with a novel graph encoding scheme to allow processing of graphs of up to one trillion edges on a single machine. MOSAIC also employs a locality-preserving, space-filling curve to allow for high compression and high locality when storing graphs and executing algorithms.

Second, while MOSAIC addresses the setting of processing static graph, this dissertation presents CYTOM, a new engine for processing billion-scale evolving graphs based on insights about achieving high compression and locality while improving load-balancing when processing a graph that changes rapidly. CYTOM introduces a novel programming model that takes advantage of its subgraph-centric approach coupled with the setting of evolving graphs. This is an important enabling step for emerging workloads when processing graphs that change over time. CYTOM's programming model allows algorithm developers to quickly react to graph updates, discarding uninteresting ones while focusing on updates that, in fact, change the algorithmic result.

CHAPTER 1
INTRODUCTION

Processing ever-larger datasets has become a common theme in recent years and is commonly deployed on large cluster across hundreds of machines. Frameworks such as Spark [1] and Hadoop [2] are widely used to process large, tera-scale datasets and solve non-trivial algorithmic problems. A subdomain of this data processing area is *graph processing* which has recently been demonstrated to scale to a scale of one trillion edges [3, 4, 5] by companies and academia alike. Graphs and their processing 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. In this dissertation, we explore approaches to allow scaling graph processing to the trillion-scale using only a single machine. This is an important stepping stone to scale up datasets and algorithms which use graph processing as an underlying layer. We also want to highlight the contributions made by this dissertation in an ever-more connected world: In a future with autonomous, connected vehicles, processing the evolving road graph and calculating shortest routes given changing congestion levels is a fitting scenario for the systems presented in this dissertation, given that this setting is inherently bound to the processing power of a single, on-board machine. Additionally, while we demonstrate the contributions for processing both static and evolving graphs using a single machine, we also discuss our contributions on applications for the distributed counterparts of graph processing systems as well.

However, to achieve trillion-scale processing of static graphs and billion-scale processing of evolving graphs, we argue that key system design challenges have to be addressed. These challenges include the size of the data set, load-balancing, fault tolerance, and support for graph algorithms. To address these challenges, this dissertation presents two approaches, MOSAIC and CYTOM, to allow scaling of big-data applications on a single

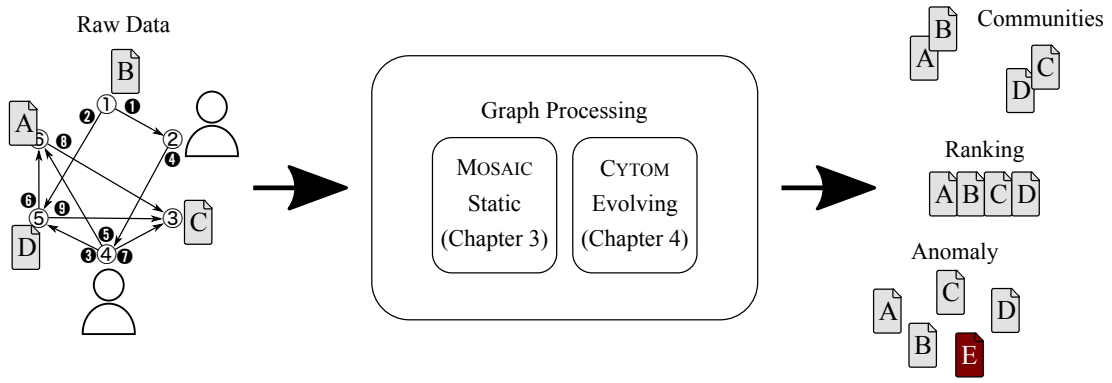


Figure 1.1: Overview of a data processing pipeline from the raw data to the algorithmic results. The dissertation covers two aspects of this pipeline, namely processing *static* and *evolving* graphs. In detail, ee scale the processing of static graph to the trillion-scale with MOSAIC while also showing a system design for processing billion-scale evolving graphs with CYTOM.

machine, Figure 1.1 gives an overview of the applicability of these techniques.

We first focus at a design-level optimization that improves processing of trillion-scale, static graphs using a novel subgraph-centric encoding of the graph which is built into a system called MOSAIC. This encoding lends itself well to compressing the size of the dataset as well as load-balancing the processing of these subgraphs. Both properties are essential when scaling to the trillion-scale on just a single machine, with constrained memory and a limited number of CPUs as compared to distributed systems. MOSAIC can also take advantage of a heterogeneous hardware setup with fast I/O devices and massively-parallel, many-core coprocessors.

On the other hand, we introduce CYTOM, a system to scale the processing of *evolving* or *streaming* graphs to the billion-scale, supporting more than hundred million updates per second on a single machine. CYTOM is built around a subgraph-centric encoding, similar to MOSAIC, however optimizes on MOSAIC's encoding by eliminating the need for meta data to be stored alongside tiles as well as allowing dynamic updates of its subgraphs compared to MOSAIC's immutable subgraph structure. CYTOM also proposes a set of APIs specifically tailored at the setting of evolving graphs to take advantage of the dynamic nature of updates, allowing the algorithm developer to cut down on unnecessary work when handling uninteresting updates that would not result in a change of the algorithmic result.

We will next discuss the specific problems addressed by both the design-level approaches as well as the OS-level approaches in detail.

1.1 Statement of Problem

- **Graph abstraction for compression *and* performance:** Graph datasets are continuously growing and have recently crossed the trillion edge threshold for a number of popular services (*e.g.*, Facebook). To allow storing larger graphs, compression of the graph data is an important feature but is usually associated with a runtime penalty when executing algorithms due to the computation needed to decompress data. To address this problem, MOSAIC [6] presents a simple scheme to allow for both compression as well as high computation performance by focusing on subgraphs. In addition, we demonstrate with CYTOM that a subgraph-centric approach can not only yield benefits in the setting of processing static graphs but is also applicable in the setting of processing evolving graphs that change over time. In fact, CYTOM proposes an encoding scheme that improves on the one presented by MOSAIC by removing the need for metadata to map subgraphs to the global pendants. This is of importance in the case of evolving graphs as the requirements call for lowered overhead of graph manipulations as well as efficiency for algorithmic processing.

- **Graph processing architecture to incorporate many-core processors:** Modern servers can be equipped with accelerators such as GPGPUs or Intel Xeon Phis to allow applications' to accelerate critical computation tasks. However, leveraging these devices for processing large, tera-scale graphs is not easily done as these devices commonly posses only a small amount of memory and are incapable of storing terabytes of data directly on the accelerator. MOSAIC approaches this problem using its abstraction of subgraphs, allowing the accelerator to focus on a small, independent subsection of the graph instead of processing the whole graph at once.

- **Scaling single-machine graph processing to a trillion edges:** Trillion-scale graph processing has been demonstrated using distributed systems of dozens to hundreds of servers. MOSAIC, on the other hand, presents an approach to allow processing of trillion-scale graphs on a *single machine* by employing its scheme of independent subgraphs, coupling it with many-core processors as well as fast I/O devices.

- **Allowing graphs to change while continuously running algorithms:** Real-world graphs change over time, but many current graph processing systems focus on optimizing the graph layout using expensive pre-processing steps which render the insertion of new data into the graph prohibitively expensive. To overcome these challenges, we propose CYTOM, a graph processing engine for evolving graphs which can ingest new graph data while keeping a graph computation up-to-date.

1.2 Thesis statement

Thesis statement: *Large-scale big data analytics is possible on a single machine using systems and design-level abstractions on commodity and heterogeneous single machines.*

Scaling computations on a single machine is important in an era of increasingly large datasets. To tackle this scalability challenge, this dissertation presents systems- and design-level abstractions for big data analytics and graph analytics in particular, to improve both their performance as well as enabling analytics on tera-scale datasets. Furthermore, this dissertation presents novel programming models for processing static and dynamically evolving graphs.

1.3 Contribution

First, this dissertation presents a new out-of-core graph processing engine, called MOSAIC, for executing graph algorithms on trillion-scale datasets on a single machine (§3). MOSAIC makes use of a heterogeneous machine setup coupled with a novel graph encoding scheme

to allow processing of graphs of up to one trillion edges on a single machine. MOSAIC also employs a locality-preserving space-filling curve to allow for high compression and high locality when storing graphs and executing algorithms. We demonstrate that MOSAIC can scale to a trillion edges on a single machine and can, in addition, outperform other, state-of-the-art engines due to its efficient encoding scheme and its hybrid execution model.

Second, this dissertation presents CYTOM (see §4), a new engine for processing evolving graphs, based on insights about achieving high compression and locality while also providing load-balancing. In addition, CYTOM incorporates a novel API designed for processing evolving graphs which allows to keep updates localized into a subgraph in many cases. Additionally, CYTOM proposes a set of APIs specifically tailored at the case of processing evolving graphs: These APIs allow the algorithm developer to specify important updates while skipping uninteresting updates which would not result in a change of the algorithmic result. This allows significant speedups for the algorithmic processing of evolving graphs. Furthermore, CYTOM is shown to be versatile enough to extend to other scenarios in the space of graph processing as well, namely approximate processing and asynchronous processing. Both modes are of importance in the space of evolving graphs due to the ability to react more quickly to graph changes while resulting in acceptable algorithmic results. We evaluate CYTOM and observe that it can optimize both throughput of graph updates as well as latency for obtaining new algorithmic results while outperforming other, state-of-the-art systems.

CHAPTER 2
RELATED WORK

In this chapter, we introduce the current state-of-the-art approaches for both the case of processing static graphs as well as approaches to process graphs evolving over time before providing a background on the kind of hardware architecture that both MOSAIC and CYTOM are built around.

2.1 Storing and Processing Static Graphs

In this section, we explore different storage mechanisms and their trade-offs as well as the algorithmic interface available for algorithm developers. The field of graph processing has seen efforts in a number of different directions, ranging from engines on single machines in-memory [7, 8, 9, 10, 11, 12, 13] to out-of-core engines [14, 15, 16, 17, 18, 19, 20] to clusters [21, 22, 23, 24, 25, 5, 4, 26, 27, 28, 29, 1, 30, 31, 32, 33], each addressing unique optimization goals and constraints.

2.1.1 Storing Static Graphs

Storing and processing static graphs has been extensively studied and optimized. As such, the storage mechanism of the *compressed sparse rows* (CSR) or *compressed sparse columns* (CSC) [34] are commonly used to reduce the overhead of storing the graph as either a raw edge list format (all edges are simply concatenated) or an adjacency list that requires extensive pointer-based traversals. Figure 2.1 gives an example of an edge list compared to a CSR encoding on a simple graph, demonstrating the benefits achievable (up to 44.5% storage reduction) using the CSR encoding. MOSAIC employs a CSR-style encoding at its core for its subgraphs while CYTOM employs a mix of edge lists and a CSR variant due to the differences in requirements for processing evolving graphs.

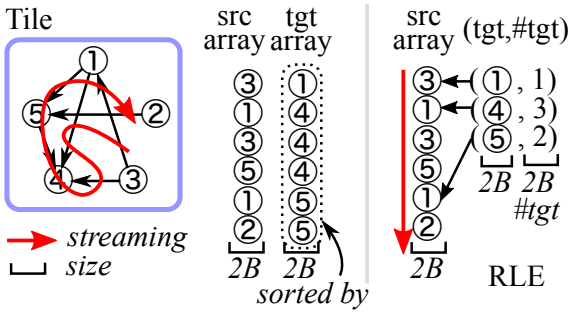


Figure 2.1: An illustration of a CSR encoding applied to a subgraph with sorted edges. All edges can be enumerated in sequence (red arrow) using less storage (saving us 44.5% I/O on average in real-world datasets) and with smaller cache footprint yet without runtime overhead.

2.1.2 Processing Static Graphs

Additionally, many of the recent engines for processing static graphs rely on a simple, yet effective processing model pioneered by Pregel [28] which is based on a *vertex-centric* abstraction. This abstraction allows algorithm developers to adopt their algorithm in a vertex-centric fashion with each vertex having a (limited) amount of local storage and the ability to communicate with its neighboring vertices via its in- and outgoing edges. PowerGraph [23] extends this model to the *gather-apply-scatter* (GAS) abstraction but retains the vertex-centric aspect while allowing for more efficient scheduling when processing large real-world, power-law skewed graphs. MOSAIC and CYTOM introduce an adoption of this familiar, yet powerful programming model to the settings of heterogeneous computing as well as an extension to processing evolving graphs.

We discuss, in more detail, how MOSAIC compares to other, state-of-the-art engines in a multitude of setups ranging from single machines to clusters to accelerators.

Single machine vs. clusters. Out-of-core graph processing on a single machine was presented by GraphChi [14] and X-Stream [15] in a scale of a few billion edges. Grid-Graph [18] and FlashGraph [19] 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.

Graph analytics at the trillion-edge scale has been demonstrated by Chaos [5] (out-of-core), GraM [4] (in-memory engine, RDMA), and Giraph [3] using over 200 servers. MOSAIC outperforms Chaos by a significant margin for trillion-edge processing, while also demonstrating the possibility of trillion-edge processing on a single machine.

In comparison to G-Store [20], 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 [35, 36, 37, 38, 39, 40, 41, 42, 43], for graph processing due to their massive parallelism. In practice, however, these engines have to deal with the problem of overheads for large data exchanges 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 [43], 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.

2.2 Reacting to Changes in Evolving Graphs

Processing graphs which change over time has been explored using, at a high-level, two different approaches: Systems that are optimized for processing static graphs but support large, batched updates, and systems that are optimized for handling evolving graphs and are able to handle small updates at the expense of less-optimized data structures and sub-optimal performance. We will examine the trade-offs involved in either of these options, especially in terms of data structures and optimizations which can be used.

2.2.1 Computations on Static Graphs

A common approach is to augment graph processing on static graphs with the ability to handle evolving graphs, e.g., GraphChi [14] or X-Stream [15, 44]. In these cases, graph updates are batched into large (e.g., millions of updates) batches to amortize overheads from optimizing the internal data structures (e.g., pre-processing through sorting edges, etc.). Commonly, an optimized graph representation called the *compressed-sparse rows* (CSR) representation [34] is used in these systems, which is a very suitable data structure optimized for high locality and minimized storage space. This data structure, however, can not be easily expanded when updates to the edges are necessary and require overflow areas to expand, as demonstrated in LLAMA [45]. While there are systems, such as X-Stream [15, 44], which do not rely on preprocessing the data set, crucially, all these systems lack the ability to continue processing from a previous snapshot of the algorithmic result and rather rely on re-execution of the algorithm, missing out on opportunities for speeding up the convergence of the algorithms.

2.2.2 Handling Evolving Graphs

Specialized systems, such as STINGER [46], GraphIn [47], or GRAPHONE [48], handling *evolving* or *streaming* graphs are fundamentally designed to be able to handle small batches of updates, up to even a single edge, by applying a different trade off in terms of data structures and algorithmic processing than their static counterparts. We first introduce the data structures possible for evolving graphs before discussing their use in other state-of-the-art systems and their benefits and drawbacks in the specific setting of processing evolving graphs (over the previously introduced use case of static graph processing).

Data structures. A simple, yet useful data structure is the *adjacency matrix*, as shown in Figure 2.2. While this is an appropriate data structure for dense graphs with a great compression ratio (1 bit for every possible edge), real-world graphs are sparse and skewed, rendering the adjacency matrix an unsuitable representation for billion-scale, real-world

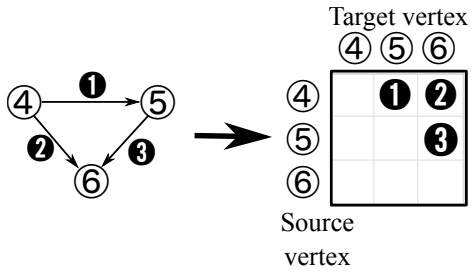


Figure 2.2: An illustration of the storage form of an adjacency matrix. Its benefit are a good compression ratio for dense graphs as well as simple update support. However, real-world graphs are skewed and sparse, rendering the adjacency matrix a non-suitable representation for billion-scale real-world graphs.

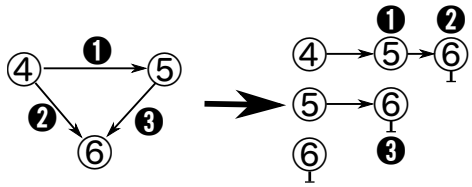


Figure 2.3: An illustration of the storage form of adjacency lists. Its benefits are support for updates and relatively low storage overhead while trading this off for poor locality and traversal overheads.

evolving graphs.

Another option is *adjacency lists*, as shown in Figure 2.3. Adjacency lists are well-suited for processing evolving graphs due to their ability of adding new graph data on the fly (potentially with the use of atomics in the presence of multiple threads inserting data at the same time). However, the storage of edges of the graph in separate memory locations, as is customary in linked lists, yields a large overhead for traversals and leads to low locality when traversing an adjacency list-based graph.

The storage form of *compressed sparse rows (CSR)*, as shown in Figure 2.4 (and previously introduced in the context of storing static graph snapshots) improves on the

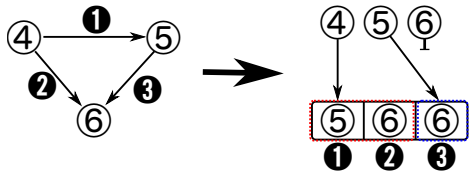


Figure 2.4: An illustration of the storage form of compressed sparse rows (CSR). Its benefits are good locality and low storage overhead, however, at the cost of inefficient update support with either complicated overflow areas [45] or expensive array shifts.

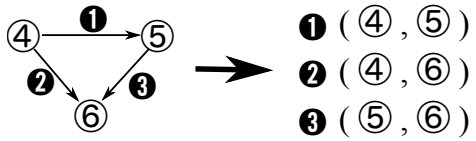


Figure 2.5: An illustration of the storage form of edge lists. Its benefits are good locality and efficient update support, however at the cost of storage overhead.

Table 2.1: Comparison of popular data structures and their applicability on real-world, evolving graphs using 64-bit vertex identifiers. We develop edge list$_{\text{CYTOM}}$ as the key data structure used in CYTOM's cells.

Representation	Sparse graphs	Updates	Traversal	Storage
Adjacency matrix	-	✓	✓	1 bit (all edges)
Adjacency lists	✓	✓	-	8 bytes
CSR	✓	-	✓	8 bytes
Edge list	✓	✓	✓	16 bytes
CYTOM	✓	✓	✓	4 bytes

adjacency lists by providing for locality and contiguity when storing the edges of the graph, however, at the cost of poor support for graph updates as either complicated overflow areas have to be used (e.g., in LLAMA [45]) or an expensive array shift operation has to be employed on every graph update.

Finally, *edge lists*, as shown in Figure 2.5, have good support for both traversals and updates, however, incur a large memory overhead due to the inability to compress the graph data.

Discussion. Table 2.1 summarizes the existing data structures for evolving graphs and introduces CYTOM's optimized edge list representation. Instead of relying on the optimized CSR representation, systems for processing evolving graphs, as introduced, commonly employ either simple adjacency lists or an edge list representation. As shown, either of these allow simple insertions (usually through the use of atomics for synchronization purposes) into their respective graph representation, allowing for a lowered latency when inserting new edges due to the missing preprocessing step. However, this practice also lowers the maximum performance achievable with these systems due to, e.g., traversal overheads (adjacency list) or the unsorted nature of the dataset (edge list).

Compared to these systems, CYTOM proposes a subgraph-centric representation of the graph, coupled with an API tailored to take advantage of this by cutting off computation on uninteresting updates at the subgraph level (using the `edgeChanged` API, §4.3.1). Additionally, CYTOM enables a lossless, zero-cost compression of the dataset using shorter, local identifiers within each subgraph, approximating the storage benefits of the CSR data structure.

In comparison to other systems targeting the case of evolving graphs [49, 50, 51, 52, 30, 53, 54], CYTOM uniquely incorporates all features outlined in Table 4.1 while outperforming comparative systems by at least 1.5. For example, CYTOM outperforms STINGER [46] by at least 60× and GRAPHONE by a factor of 1.5× due to its efficient design and specific extension tailored towards processing evolving graphs.

Additionally, a number of systems, including GraphIn [47], Kickstarter [55], and GraphBolt [56] explicitly optimize for the case of edge deletions. In comparison, CYTOM's edge deletion optimizations are simpler in nature but still effective while the optimizations of these systems could also be applied to CYTOM for an even larger gain in performance.

2.3 Hardware Trends

MOSAIC and CYTOM both run on machines that incorporate a number of hardware trends such as large amounts of memory, CPUs on multiple sockets, and novel storage and computing devices centered around the PCIe bus. We design both projects to take advantage of commodity machines (i.e., two-socket server machines with a few hundred GB of memory) while MOSAIC in particular extends this to a setting of high-performance computing while still being able to extract superior performance on a commodity machine.

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) [57]). 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 while CYTOM designs its fundamental mechanism to be easily extended to the setting of NUMA machines.

Non-volatile memory express (NVMe). NVMe is the interface specification [58] to allow high-bandwidth, low-latency access to SSD devices connected to the PCIe bus, 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 [59]). 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 [60]. 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 launch many subgraph-centric computations in parallel.

CHAPTER 3
SCALING GRAPH PROCESSING TO A TRILLION EDGES ON A SINGLE MACHINE WITH MOSAIC

We first present an approach to scale processing static graph of up to a trillion edges on a single machine by designing a new engine, MOSAIC, to take advantage of heterogeneous hardware setups while taking advantage of the characteristics of the underlying data.

In detail, MOSAIC presents the following approach: It presents an out-of-core graph processing engine, designed for many-core processors and fast I/O devices. In addition, MOSAIC takes advantage of a locality-preserving design and re-designs the execution pipeline to take advantage of a large number of cores offered by its many-core coprocessor. We will present the fundamental design decisions for MOSAIC and show that these decisions enable it to scale to a graph of one trillion edges on a single machine.

3.1 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 [3] poses fundamental challenges to existing graph processing engines. To tackle this issue, researchers are focusing on distributed graph processing engines like GraM [4], Chaos [5] and, Giraph [3] 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 [4] and 40 GbE in Chaos [5]). 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

Table 3.1: 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).

	Single machine [14, 15, 10, 11]		**GPGPU [38, 62, 43]**		**MOSAIC**	**Clusters [4, 5, 24, 63]**	
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

straggler problem [5] and fault tolerance [22, 61] 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 [5] or distributed locking [21]).

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* [14, 18, 15, 19, 17, 43], or on scaling up the processing performance, by exploiting memory locality of high-end machines termed *in-memory graph analytics* [11, 10, 9]. 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 3.1).

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 [5] by 9.2×, while only being 8.8× slower than a distributed in-memory one [4] 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 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 [14] 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 [19, 18].

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 chapter, 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 [5]) 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.

3.2 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 [3], 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 [64]. 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 [32, 40, 65].

To mitigate random accesses to the edges, an edge-centric approach streams through the edges with perfect locality [15]. 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 [66] 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 §3.3.1, §3.3.2).

Load balancing. The skewness of real-world graphs presents another challenge to load balancing. Optimal graph partitioning is an NP-complete problem [67], so in practice, a traditional hash-based partitioning has been used [28], but it still requires dynamic, proactive load-balancing schemes like work stealing [5]. MOSAIC is designed to balance workloads with a simple and independent scheme, balancing the workload between and within accelerators (see §3.4.5).

I/O bandwidth. The input data size for current large-scale graphs typically reaches multiple terabytes in a conventional format. For example, GraM [4] 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 [68]). 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 §3.3.3).

3.3 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 3.3 gives an overview of the interaction of the components of MOSAIC.

In this section, we first introduce the core data structure, called *tiles* (§3.3.1), and their ordering scheme for locality (§3.3.2), and explain each component of MOSAIC in detail (§3.3.3).

3.3.1 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 3.1 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

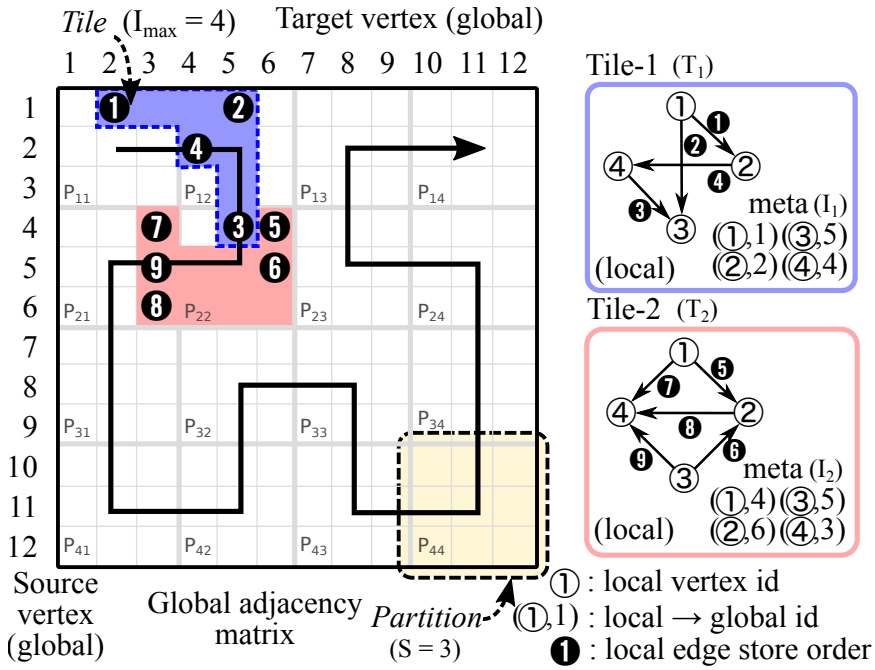


Figure 3.1: The Hilbert-ordered tiling and the data format of a tile in MOSAIC. There are two tiles (blue and red regions) illustrated by following the order of the Hilbert curve (arrow). Internally, each tile encodes a local graph using local, short vertex identifiers. The meta index I_j translates 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.

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 (§3.3.2).

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 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 3.9, $< 2.3\%$ of imbalance).

Data format. More concretely, the format of a tile is shown in Figure 3.2 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 vertex enables memory and cache locality due to repeated target vertices in a tile. For example, in Figure 3.1, 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 3.1) into partitions of size $S \times S$, where $S = 2^{16}$.

We consume partitions following the Hilbert order (§3.3.2) 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 3.1, using the parameters $I_{max} = 4$

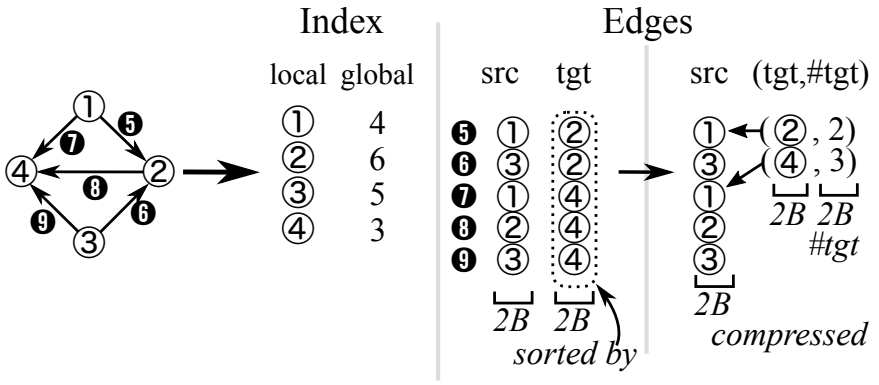


Figure 3.2: An overview of the on-disk data structure of MOSAIC, with both index and edge data structures, shown for tile T_2 (see Figure 3.1). The effectiveness of compressing the target-vertex array is shown, reducing the number of bytes by 20% in this simple example.

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 [66], the Hilbert-ordered tiles save a global sorting step, only arrange the tiles, not the edges, into the global Hilbert order.

3.3.2 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 [66, 69]. 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-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 3.1 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.

3.3.3 System Components

From the perspective of components, MOSAIC is subdivided according to its scale-up and scale-out characteristics, as introduced in Figure 3.3. 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

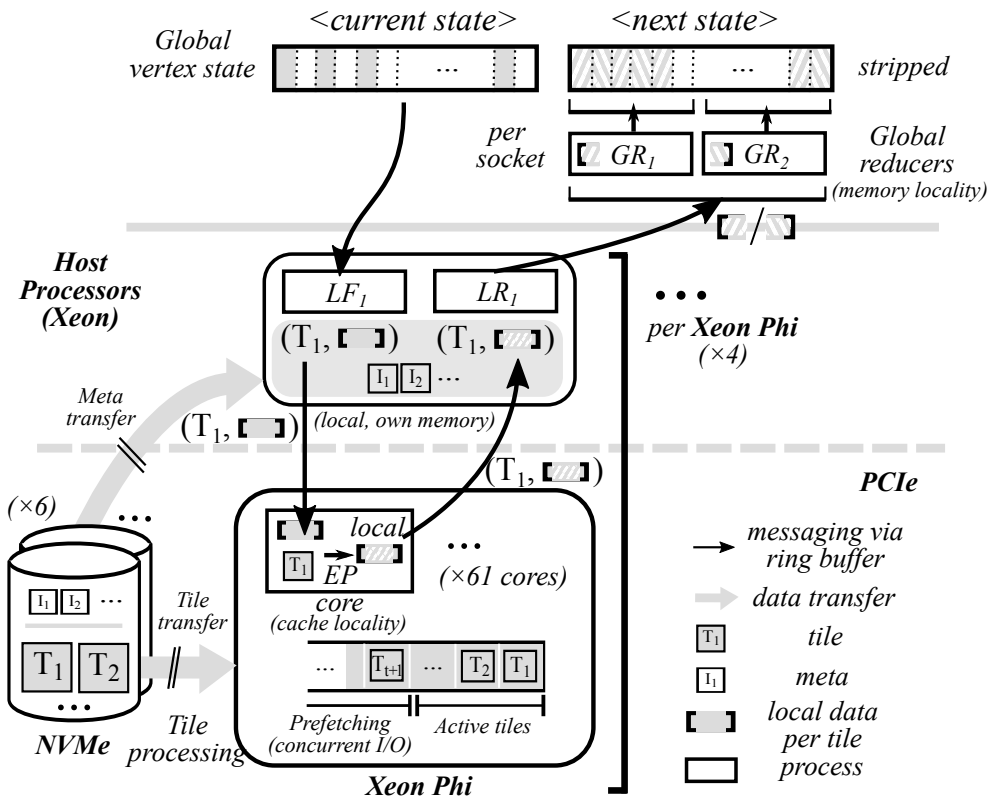


Figure 3.3: MOSAIC's components and the data flow between them. The components are split into *scale-out* for Xeon Phi (*local fetcher* (LF), *local reducer* (LR) and *edge processor* (EP)) and *scale-up* for host (*global reducer* (GR)). The *edge processor* runs on a Xeon Phi, operating on local graphs 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.

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 3.3). 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 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.

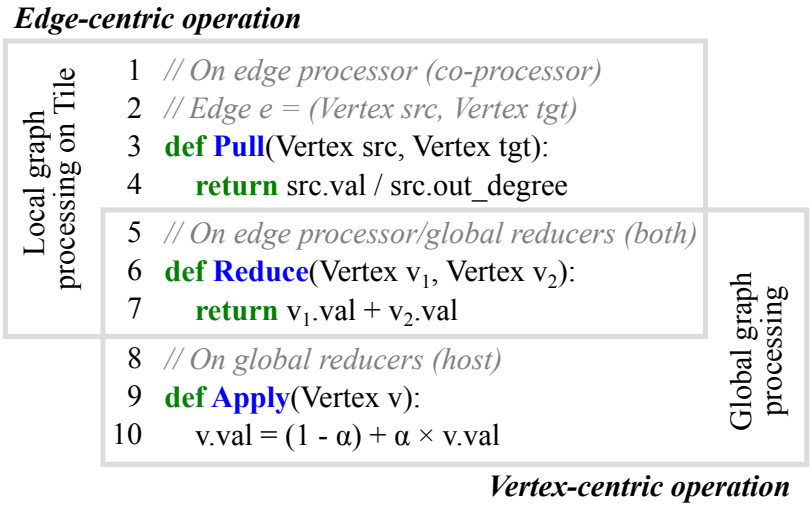


Figure 3.4: The Pagerank implementation on MOSAIC. *Pull()* operates on edges, returning the impact of the source vertex, *Reduce()* accumulates both local and global impacts, while *Apply()* applies the damping factor α to the vertices on each iteration.

3.4 The MOSAIC Execution Model

MOSAIC adopts the popular "think-like-a-vertex" programming model [28, 21], 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 [28, 24, 70]. 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 3.4), which ranks vertices according to their impact to the overall graph, as a running example.

3.4.1 Programming Abstraction

MOSAIC provides an API similar to the popular Gather-Apply-Scatter (GAS) model [23, 21, 28]. 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 3.5 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

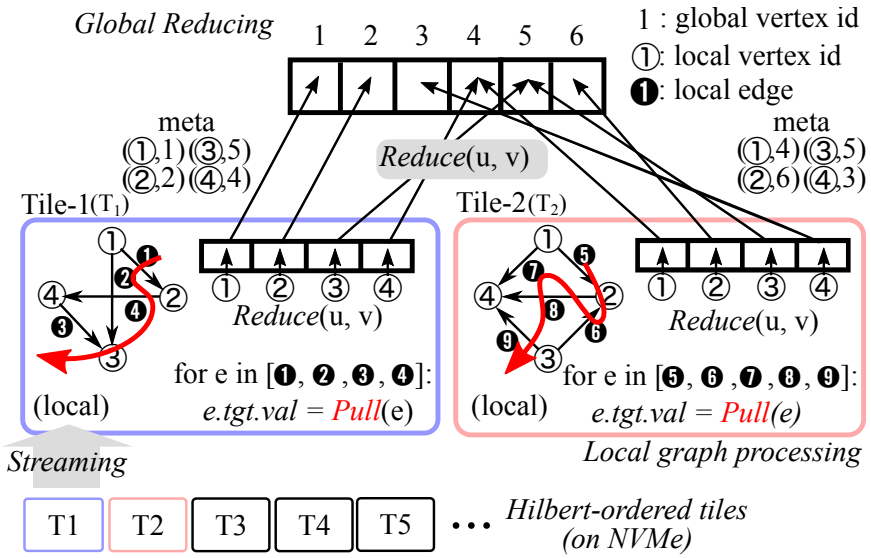


Figure 3.5: The execution model of MOSAIC: it runs *Pull()* on local graphs while *Reduce()* is being employed to merge vertex states, for both the local as well as the global graphs.

algorithms (Pagerank, Bayesian Belief Propagation, Approximate Triangle Counting, Sparse Matrix-Vector Multiplication).

3.4.2 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.

3.4.3 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.

3.4.4 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.

3.4.5 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 3.9).

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 3.9), 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.

3.4.6 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.

3.5 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 [71] 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, 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.

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

Table 3.2: 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.

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	$\times, +$	$O(2mE)$	$O(8mV)$

3.6 Graph Algorithms

We implement seven popular graph algorithms by using MOSAIC's programming model. Table 3.2 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 [11, 15].

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. [72].

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 [73].

Table 3.3: Two machine configurations represent both a consumer-grade gaming PC (`vortex`), and a workstation (`ramjet`, a main target for tera-scale graph processing).

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 $\times$ 1	30 MB $\times$ 2
PCIe	v3 (48L, 8 GT/s)	v3 (48L, 8 GT/s)
NVMe	1	6
Xeon Phi	1	4

Bayesian belief propagation (BP) operates on a weighted graph and propagates probabilities at each vertex to its neighbors along the weighted edges [74].

3.7 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? (§3.7.2, §3.7.3)
- **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? (§3.7.4)
- **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? (Figure 3.7.5)

3.7.1 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 3.3 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 [75], 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 [3]. We also use two types of real-world datasets, social networks (twitter [76]) and web graphs (uk2007-05 [77, 78], and hyperlink14 [79]). These datasets contain a range of 1.5–64.4 B edges and 41.6–1,724.6 M vertices (35–37$\times$ ratio), with raw data of 10.9–480.0 GB (see Table 3.4). 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 graph into 7.7 GB, while GridGraph's conversion is more than 4.3$\times$ 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 [14], X-Stream [15] and GridGraph [18] 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

[2] Default parameters are $a = 0.57, b = 0.19, c = 0.19$, `https://graph500.org/?page_id=12#sec-3_2`

Table 3.4: 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	#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

a diverset set of architectures for graph engines: For single machines, we show the results for in-memory processing, with Polymer [11] and Ligra [10], and GPGPU, with TOTEM [62] and GTS [43]. Furthermore, we show the results for distributed systems, using the results for Chaos [5], on a 32-node cluster, as an out-of-core system, as well as GraphX [24] as an in-memory system. Furthermore, we include a special, pagerank-only in-memory system by McSherry et al [63] 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 [63].

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 3.6). We measure the

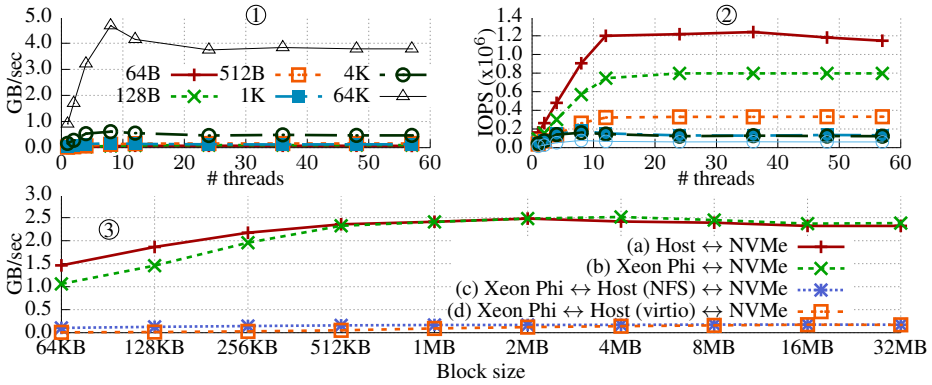


Figure 3.6: ① Throughput and ② IOPS of our ring buffer for various size messages between a Xeon Phi and its host, and ③ the throughput of random read operations on a file in an NVMe.

throughput of random read operations when reading from a single file on an NVMe in four configurations: (a) a host application directly accesses 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.

3.7.2 Overall Performance

We ran seven graph algorithms (Table 3.2) on six different datasets (Table 3.4) with two different classes of single machines (Table 3.3). Table 3.5 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 [11]) and distributed engines (e.g., 2,770–6,335 M edges/sec for McSherry et al.'s Pagerank-only in-memory cluster system [63]).

An example of the aggregated I/O throughput of MOSAIC with 6 NVMes is shown in Figure 3.7, 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

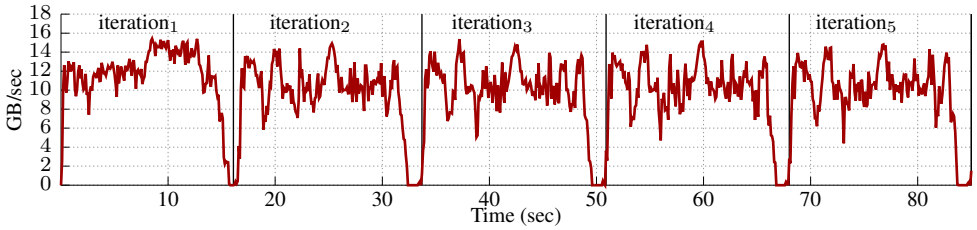


Figure 3.7: Aggregated I/O throughput for SpMV on the hyperlink14 graph for the first five iterations. The drop in throughput marks the end of an iterations while the maximum throughput of MOSAIC reaches up to 15 GB/sec.

Table 3.5: 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.

Graph	#edges (ratio)	PR‡	BFS†	WCC†	SpMV‡	TC‡	SSSP†	BP‡
⋆**rmat24**	0.3 B (1×)	0.31 s	3.52 s	3.92 s	0.30 s	1.46 s	11.71 s	0.47 s
twitter	1.5 B (5×)	1.87 s	11.20 s	18.58 s	1.66 s	9.42 s	54.06 s	5.34 s
⋆**rmat27**	2.1 B (8×)	3.06 s	17.02 s	22.18 s	2.74 s	16.52 s	101.95 s	8.42 s
uk2007-05	3.7 B (14×)	1.76 s	1.56 s	5.34 s	1.49 s	4.31 s	2.05 s	4.57 s
hyperlink14	64.4 B (240×)	21.62 s	6.55 s	708.12 s	19.28 s	68.03 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	-	-

a trillion-edge graph. We run all non-weighted algorithms (PR, BFS, WCC, SpMV, TC) on 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 [5] by a factor of 9.2×.

3.7.3 Comparison With Other Systems

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

- Single machine, out-of-core: These systems include GraphChi [14], X-Stream [15], and GridGraph [18] and are close in nature to MOSAIC.
- Single machine, in-memory: Operating on a high-end server with lots of memory, we include Polymer [11] and Ligra [10].
- Single machine, GPGPU: As an upper bound on the computing power on a single

Table 3.6: 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 [43]), 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. [63] 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.

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

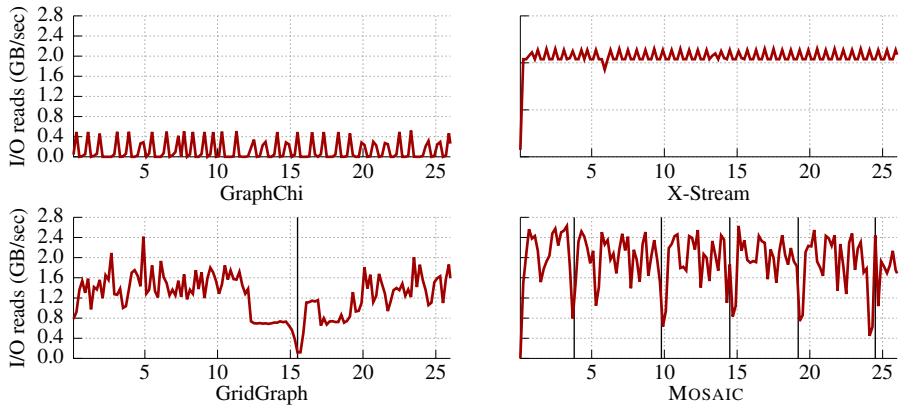


Figure 3.8: I/O throughput of out-of-core graph engines for 25 seconds of the Pagerank algorithm and the uk2007-05 dataset using a single NVMe on `ramjet`. The iteration boundaries are marked, neither GraphChi nor X-Stream finish the first iteration. GraphChi does not exploit the faster storage medium. 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.

machine, we include TOTEM [62] and GTS [43], 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 [5] 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 [24] running on 16 nodes as well as a specialized, Pagerank-only system running on 16 nodes by McSherry et al [63].

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 [43, 11, 10, 24, 5, 63], as we lack appropriate infrastructure for these systems.

The results of this comparison with Pagerank are shown in Table 3.6. 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 3.8 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 3.7.

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 3.8. 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, MOSAIC is able to

Table 3.7: 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.

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 3.8: 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×.

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×)

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 [63] 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.

3.7.4 Evaluating Design Decisions

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

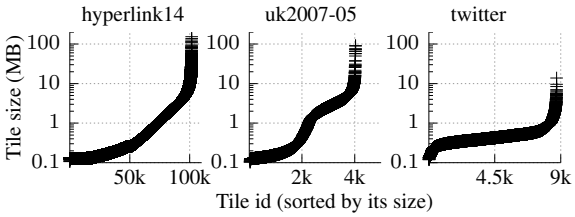
Hilbert-ordered Tiling

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 trillion-edge graph (rmat-trillion) (see Table 3.4), signifi-

Table 3.9: 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.

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



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

Figure 3.9: 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.

cantly 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 3.9. 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.

Table 3.10: 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 §3.4.5 and improves the total running time by up to 5.8× by preventing starvation.

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

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 3.9).

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 3.10 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).

Global Reducer

MOSAIC uses global reducers (see §3.3.3) 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

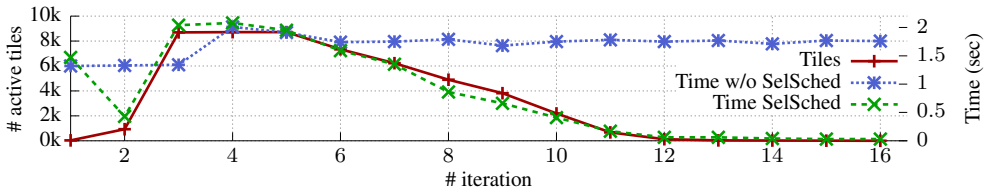


Figure 3.10: The number of active tiles per iteration and the execution time per iteration with and without the selective scheduling on twitter for BFS. It converges after 16 iterations and improves the performance by 2.2×.

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.

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 3.10 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).

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`.

3.7.5 Performance Breakdown

We evaluate which factors influence MOSAIC's overall performance and look at the storage mechanism (NVMe or SSD?), the number of storage devices used, and the impact of using the Xeon Phi compared to only using the CPUs of the host machine.

Storage medium used

For this evaluation, we run MOSAIC using a varying number of SSDs instead of the usual setup using a set of NVMes. Additionally, due to a limitation in the way our Xeon Phis access the disks (NVMes are accessed via the PCIe interface while SSDs are accessed via the legacy, serial-attached SCSI (SAS) interface), we run this experiment on the CPUs only. We show the results in Figure 3.11 and note that, while MOSAIC scales well for the first few SSDs added, up to three. However, after this the performance benefit stagnates due to saturation of the number of threads used by MOSAIC to read from these SSDs: When using four SSDs, we use two threads per SSD to read both the tile data itself as well as the associated meta data. Thus, only a limited number of threads is still available for data processing compared to simply reading from SSDs as MOSAIC does not include a scheduling component to re-assign idle threads. Additionally, MOSAIC's load balancing of tiles is not perfect, as seen in Figure 3.9. This causes an imbalance in the amount of time spent for each thread which is reading the graph from disk and degrades overall performance, especially as the data is now split among a larger number of devices (e.g., ten SSDs instead of four NVMes).

Scalability

We also investigate the scalability when using NVMes in place of SSDs, MOSAIC's typical configuration. 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 3.12). 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.

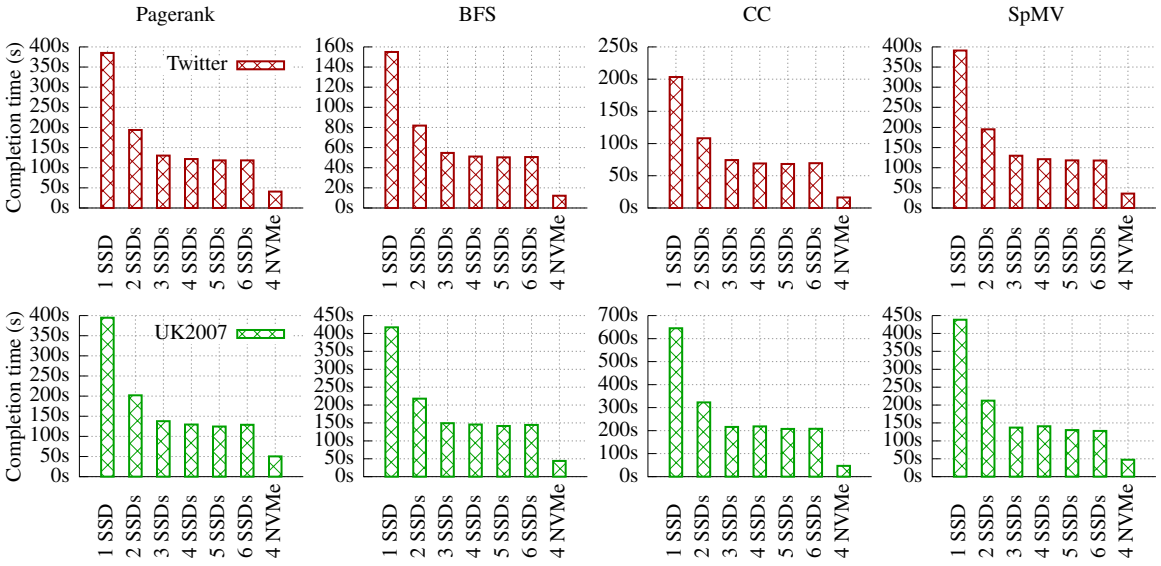


Figure 3.11: The performance of MOSAIC running on an increasing number of SSDs. MOSAIC scales well and saturates the available throughput until four SSDs when the overhead of polling threads does not allow for further scalability. Pagerank and SpMV are run for 20 iterations while BFS and CC are run to convergence.

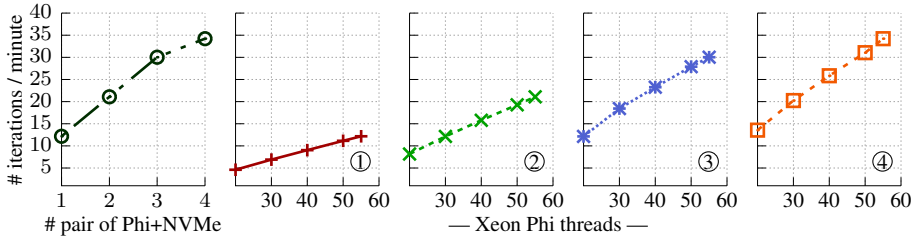


Figure 3.12: Time per iteration (a) with increasing pairs of a Xeon Phi and a NVMe (left one), and (b) with increasing core counts in multiple Xeon Phis from ① to ④. MOSAIC scales very well when increasing the number of threads and scales reasonably up to the fourth Xeon Phi, when NVMe I/O is starting to be saturated.

MOSAIC *using the CPU only*

Finally, 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 3.11 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

Table 3.11: 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 3.5 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.

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%)	-	-

memory while the Xeon Phi-enabled setup has to copy data over the slower PCIe interface.

Summary of performance breakdown

As seen, MOSAIC scales well when using NVMes and is effective for both the setting of using the Xeon Phi as well as the CPU-only setup. However, MOSAIC's scalability is limited when running on SSDs due to the much lower disk throughput offered by SSDs compared to NVMes on a per-thread basis. This shows MOSAIC's benefits when run in a modern hardware setup (e.g., SSDs) as compared to relying on SSDs or hard drives.

3.8 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 [80], and Intel Optane NVMe [81], is expected to resolve 1), 3) and 4) in the near future, but to resolve 2), devices such as a PCIe switch [82] 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 [23, 24, 5, 4] 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 [63],

40 GbE [5] or InfiniBand [4], 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 §3.7.1, the conversion time is comparable to other systems such as GridGraph [18].

3.9 Chapter summary

In this chapter, 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.

While MOSAIC scales to very large, tera-scale graphs, we note that real-world graphs often change over time and are dynamic in nature. We will now look and tackle the challenges in the space of evolving graphs while building on the design principles outlined with MOSAIC.

CHAPTER 4
PROCESSING BILLION-SCALE EVOLVING GRAPHS ON A SINGLE MACHINE WITH CYTOM

After introducing MOSAIC as a solution for processing tera-scale, static graphs we will now look at the case of processing graphs that *evolve* over time using our system called CYTOM. Compared to MOSAIC, CYTOM faces the more demanding challenge of having to both quickly answer to user-facing queries while allowing the underlying graph structure to change rapidly and efficiently.

4.1 Introduction

Large-scale graphs are a common building block in many real-world applications and scenarios like social networks, data science, the World Wide Web, or route finding. While there are many approaches to process static versions of these graphs in a diverse set of fields, processing graphs that can change over time, so-called *streaming* or *evolving* graphs, is a different, more latency-sensitive challenge. Thus, many real-world scenarios that could benefit from a real-time analysis, such as anomaly detection [83] or topic detection in streams of graph updates [84], are still handled in a batch-processing manner [3]. In addition, these graphs see a large number of concurrent changes to their graph structures, e.g., at Twitter [85] ($\approx 150{,}000$ tweets per second), Facebook [86] ($\approx 85{,}000$ updates per second), as well as millions of emails being sent per second [87]. This provides the motivation for the development of engines specifically targeted at processing evolving graphs while focusing on achieving high throughput when updating the graph and also providing low latency to obtain updated algorithmic results.

This problem of achieving both high throughput while providing low latency is prevalent in both single-machine-based systems as well as distributed systems. However, while

Table 4.1: Overview of CYTOM and other frameworks for processing evolving graphs. CYTOM will be open-sourced upon publication.

	Characteristics	Single Machine					Distributed System	
		LLAMA [45]	GraphIn [47]	GRAPHONE [48]	STINGER [46]	CYTOM	KickStarter [55]	Timely Dataflow [30]‡
Perf.	Scale (#edges)	1 B	100 M	3 B	100 M	3 B	2 B	3 B
	Ingestion rate (#edges/s)	10 M	10 M	40 M	1 M	140 M	-	30 M (per machine)
	Batch rate (#batches/s)	≈ 10	≈ 100	≈ 600	≈ 1,000	≈ 100,000	-	≈ 100,000
	Batch sizes (#edges)	> 1M	> 100K	> 65K	> 10K	> 1K	-	> 1K
Prog. model	Incremental proc.	-	✓	✓	✓	✓	✓	✓
	Synchronous proc.	✓	✓	✓	✓	✓	✓	✓
	Asynchronous proc.	-	-	-	-	✓	-	✓
	Approximation	-	-	-	-	✓†	-	✓
Features	Disk persistence	✓	-	✓	-	✓	-	-
	Deletions	✓	✓	✓	-	✓	✓	✓
	Snapshot view	✓	-	-	-	✓	-	✓
	Open source	✓	-	✓	✓	✓	-	✓

‡: Note that Timely Dataflow is not specialized for graph analytics.
†: Through CYTOM's `edgeChanged` API and varying δ for the pagerank algorithm.

distributed systems for this problem have been proposed [30, 55, 88, 89], they incur additional overhead for tasks such as persisting the state of the graph (needs a two-phase commit-like protocol to ensure strong consistency after graph updates) while presenting a cost-intensive solution to a problem that can be solved on a smaller set of machines or a single machine itself. On the other hand, approaches on a single machine face the problem of limited graph sizes due to memory and storage constraints, as well as low performance when trying to combine both high update throughput and low algorithmic latency.

To tackle these problems, we introduce CYTOM[1], a graph-processing engine for evolving graphs using a *cell*-based abstraction. At its core, CYTOM incorporates 2D cells of the graph's adjacency matrix that unlock a number of benefits for CYTOM, including a drastic reduction in required memory storage space, a lightweight load-balancing scheme, and a straightforward mechanism for parallel graph updates resulting in high throughput. In addition, CYTOM is able to perform updates on a *single edge* basis rather than relying on large update batches, if the application requires minimal latency for the updates to be reflected in the graph. CYTOM's design is also amenable to a simple, yet effective persistence scheme, allowing updates to be written to disk with modest overhead to provide for failure consistency and offline analysis.

Along with its cell abstraction for the data storage of the graph, CYTOM introduces an

[1] CYTOM is a shorthand for *cytometry*, the determination of the properties of cells.

API tailored specifically for the setting of evolving graphs. This API allows the algorithm to decide whether to heavily process new graph updates or to keep updates local inside a cell, allowing for a lowered latency for the completion of graph updates in many cases. Additionally, we demonstrate that CYTOM's cell-based execution model is general and flexible enough to support diverse operating modes like approximation, asynchronous graph processing, as well as a snapshot-based view. Table 4.1 incorporates these operating modes as well as key performance numbers for a representative set of graph-processing engines for evolving graphs, including both single-machine as well as distributed systems. This shows that CYTOM is able to uniquely combine multiple operating modes with high performance and small update batch sizes rivaling even the performance of distributed systems.

In this chapter, we present CYTOM and its contributions:

- We introduce the design of CYTOM's cell abstraction as well as the data structure trade-offs necessary to support dynamically evolving graphs, yielding a zero-cost compression scheme as well as a simple load-balancing scheme.
- We propose an API and execution model tailored for evolving graphs by incorporating a hybrid edge- and vertex-centric API coupled with the `edgeChanged` API to allow a timely reaction to graph changes.
- We demonstrate the benefits of CYTOM and its cell-based abstraction of the graph on a commodity server and show that CYTOM is able to outperform other state-of-the-art systems by 1.5×–200×.

4.2 CYTOM's Design

We introduce CYTOM's design, starting with CYTOM's components and its core idea of using *cells*. We then discuss the data structure at the core of CYTOM before introducing our approach for persistence. Algorithmic aspects of CYTOM are then discussed in §4.3.

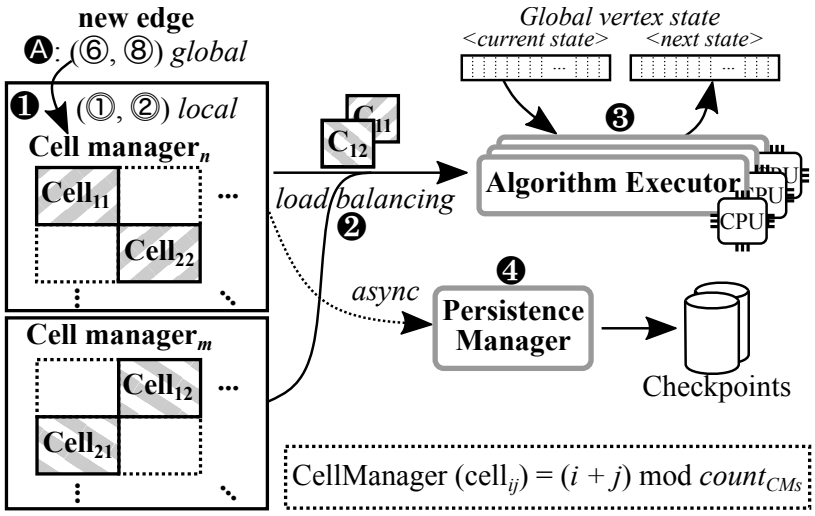


Figure 4.1: Overview of CYTOM's components along with an exemplary insertion of edge Ⓐ and the steps taken by CYTOM to generate an algorithmic output.

4.2.1 System Overview

CYTOM's components, tailored towards processing an evolving graph, are shown in Figure 4.1 with the addition of an edge (Ⓐ) and the steps taken by CYTOM to generate an algorithmic output. These components are designed to be NUMA-aware, balancing the number of cells between sockets.

Cell manager. (§4.2.2) Cell managers are responsible for storing the edges (❶ in Figure 4.1). Additionally, they invoke the `edgeChanged` API for every updated edge. Cell managers are not run as a thread and are rather a storage abstraction.

Load balancing. (§4.3.3) As part of the algorithm execution (step ❷ in Figure 4.1), the cell distributor achieves load balancing when executing an algorithm on CYTOM's cells.

Algorithm executor. (§4.3.1) Algorithm executors are run as threads and execute cells assigned to them by the cell distributor (see step ❸ in Figure 4.1). They use the abstraction of the cell manager to execute the given algorithm on every (active) cell, updating the values stored in the global vertex array.

Persistence writer. (§4.2.3) The persistence writer is notified of all graph updates (step ❹ in Figure 4.1) and asynchronously persists graph changes to disk.

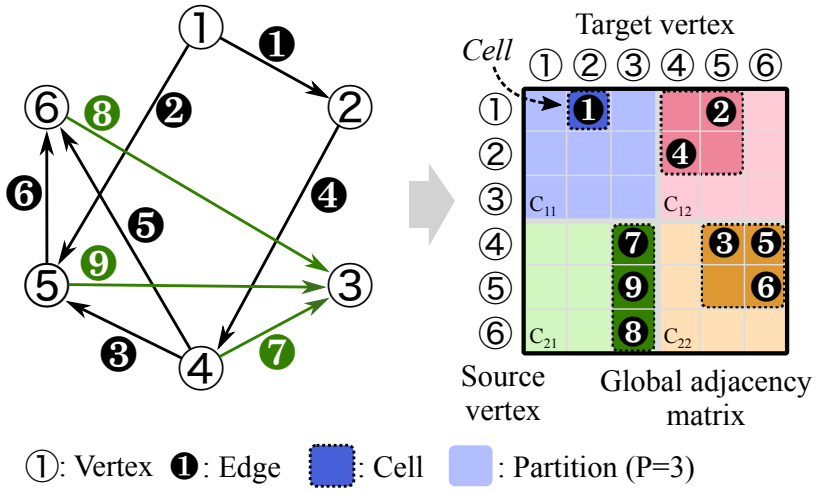


Figure 4.2: The construction of CYTOM's cell structure as submatrices of the global adjacency matrix with the submatrix size set to three.

4.2.2 Cell Model

The core idea of CYTOM is to form *cells* as distinct submatrices of size p of the adjacency matrix of the evolving graph, as shown in Figure 4.2. This technique enables a number of optimizations: local, short identifiers (e.g., 16 bits instead of 64 bits), implicit handling of high-degree vertices, and load balancing for insertions as well as the algorithm execution. Each cell represents a distinct subgraph of the global graph with no edges spanning two cells. More formally, we assign an edge (u, v) to the cell (i, j) with $i = \left\lfloor \frac{u}{p} \right\rfloor$ and $j = \left\lfloor \frac{v}{p} \right\rfloor$. This structure allows each cell to be treated as an independent unit of computation and alleviates load-balancing concerns. We show the individual data structure of a cell in Figure 4.3. CYTOM also supports weighted graphs by storing the edge weight as a 4-byte float.

While systems designed for static graphs have adopted a similar structure (e.g., *edge blocks* in GridGraph [18] or *tiles* in MOSAIC [6])[2], these structures are immutable in nature due to their tightly packed format to optimize I/O throughput. In comparison, CYTOM's cells are designed to dynamically expand or shrink as required. Furthermore, CYTOM's cells retain the benefits demonstrated by MOSAIC's tiles of achieving a lower memory footprint using short identifiers. In fact, with CYTOM's cells, we are able to even remove the metadata

[2]The tiles in MOSAIC would contain [❶, ❷, ❸, ❹] and [❺, ❻, ❼, ❽, ❾] as a second tile in Figure 4.2.

Table 4.2: Popular data structures and their applicability on real-world evolving graphs using 64-bit vertex identifiers. CYTOM meets all requirements while improving the storage overhead.

Representation	Sparse graphs	Updates	Traversal	Storage
Adjacency matrix	-	✓	✓	1 bit (all edges)
Adjacency lists	✓	✓	-	8 bytes
CSR	✓	-	✓	8 bytes
Edge list	✓	✓	✓	16 bytes
CYTOM	✓	✓	✓	4 bytes

overhead that MOSAIC suffers from, accounting for up to 30% of MOSAIC's overall graph storage cost.

Data structure. As shown, CYTOM stores new edges in an *edge list* format. Edge lists have all the desired features for storing evolving graphs: They support sparse graphs (a key feature of real-world graphs), updates are cheap, and there is no overhead for traversing the edges, as they are stored in a contiguous array. However, compared to other options, such as adjacency lists or the *compressed sparse rows* (CSR) format, edge lists have a higher storage cost, as shown in Table 4.2. However, using CYTOM's local, short identifiers, we are able to retain all the benefits of traditional edge lists while removing the storage overhead, yielding CYTOM's representation. The structure of an individual cell is also shown in Figure 4.3.

Local identifiers. A key advantage of CYTOM's cells is the opportunity to use shorter, cell-local identifiers (e.g., 16 bits instead of 64 bits). This is facilitated by enumerating a cell's vertices from zero to the maximum per the given bit size (e.g., with 16 bits, up to $2^{16} = 65536$ vertices per cell). This scheme yields a simple, arithmetic translation mechanism to convert identifiers, as shown in Figure 4.3. This scheme allows a reduction in memory cost of 50% (over 32-bit identifiers) or 75% (64-bit identifiers). CYTOM uses 16-bit identifiers as a balance between the number of cells and the potential overhead of dealing with a large number of cells (e.g., using 8-bit identifiers results in more than one billion cells for even the smallest real-world graph in our evaluation while 16-bit identifiers only result in 16K cells) and the benefits for memory footprint and cache hits associated with using 16-bit identifiers over 32-bit or 64-bit ones (see §4.4.3).

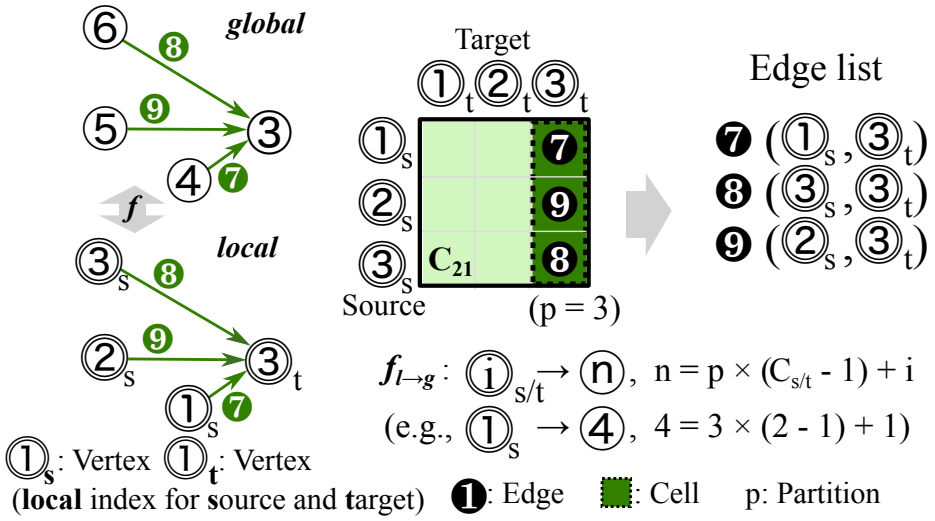


Figure 4.3: The construction of a single cell, C_{21}. Note that CYTOM is able to use shortened identifiers (double-circled) by translating the global identifiers into the cell-local ID space.

Handling of high-degree vertices. High-degree (or hub) vertices are common in real-world graphs that follow a power-law distribution [90] like Web graphs [91] or social networks [76]. As such, hub vertices may become a bottleneck for load-balancing and necessitate special treatment [23]. CYTOM handles these cases implicitly: CYTOM's cell structure balances the load for high-degree vertices, as incoming and outgoing edges will be split among a large number of cells, allowing for both simultaneous updates to the graph structure as well as parallelism when executing algorithms.

Load balancing. As another benefit of CYTOM's cell design, load balancing for skewed graphs is much simpler than with other schemes (e.g., splitting of high-degree vertices [26]). Load balancing with CYTOM is achieved by allowing multiple threads to operate on their assigned cells in parallel, while the *cell distributor* imposes load balancing decisions on the distribution of cells to the executors (see §4.3.3).

Growing and shrinking. As shown in Figure 4.4, CYTOM's cells can expand or shrink by adding or deleting edges. When appending new edges, CYTOM retains an amortized constant time per insertion by exponentially growing the underlying edge array. In addition, CYTOM stores a bit list, marking deleted edges that will not be used by the evolving algorithm. The potential overhead of already-deleted, but still-present edges can be recovered when loading

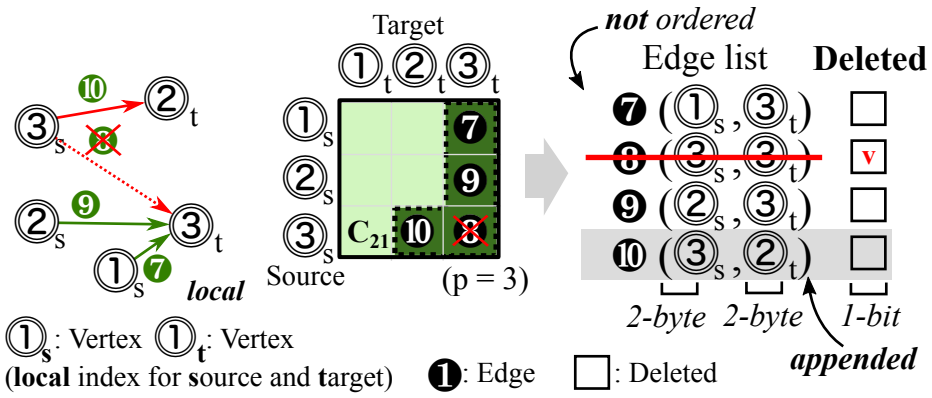


Figure 4.4: The dynamic behavior of a cell including deletion of edge ❽ and addition of edge ❿. CYTOM uses a deletion bit and compacts its format when recovering from a persisted snapshot.

a previously persisted snapshot from durable storage (see §4.2.3).

Cell management. Managing CYTOM's cells is challenging because of the large number of cells: A graph with 130 million vertices will result in a total number of cells of more than 4 million, i.e., many more cells than threads available. However, it is desirable to assign a cell exclusively to a thread to avoid synchronization overhead that would arise when multiple threads were to update edges within the same cell. To address this issue, we introduce the so-called *cell managers* (CMs): Each cell is exclusively assigned to a cell manager (i.e., a thread), and all cells are evenly divided among all cell managers. A simple partitioning, however, might be prone to hotspots. To alleviate this problem, we use a checkerboard pattern of the adjacency matrix to assign cells to the cell managers. This assignment is made by the simple formula of $\text{CM}_{\text{assigned}} = (i + j) \mod count_{\text{CMs}}$, with i being the row and j being the column of the cell. As a secondary benefit, this scheme is well-suited for being expandable when new cells are added by simply extending the checkerboard pattern.

Compaction. CYTOM supports the deletions of edges and provides abstractions to handle cases in which a deletion can result in incorrect algorithmic results (see §4.2.4). CYTOM uses a deletion flag to track edges that are no longer part of the graph. This can result in inefficiencies when a large number of edges is deleted; however, deletions are relatively rare (e.g., 1% of edges being deleted per batch) and CYTOM supports compacting its storage

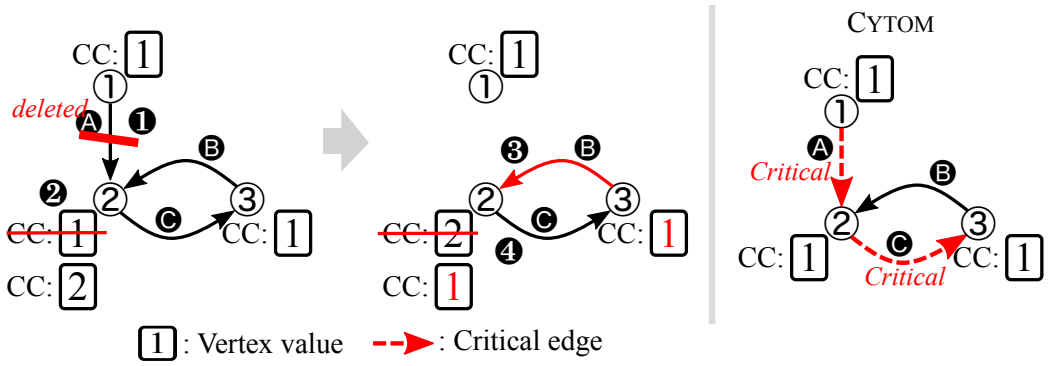


Figure 4.5: A deleted edge can lead to incorrect results, shown with the CC algorithm: Edge Ⓐ is deleted (❶), which in turn updates the result of vertex ② (❷). However, due to edge Ⓑ (❸), the result of vertex ② converges to an incorrect result (❹). CYTOM (on the right side) defines algorithm-specific *critical edges* to indicate when a deletion will result in incorrect algorithmic results.

when recovering from a persisted snapshot, permanently removing deleted edges.

4.2.3 Persistence

CYTOM enables on-disk persistence of the evolving graph. Persistence is an important part of every system processing evolving graphs; however, it is even more important on a single machine given the potential for catastrophic data loss on a crash. CYTOM writes edge insertions, changes (e.g., changes to edge weights), and deletions to a persistent, on-disk version. For persistence, CYTOM uses a per-cell-manager log file that is synchronously written to every batch. A synchronous write is important to allow for an immediate persistence of the on-disk log rather than relying on a faster, but eventually-consistent buffer cache write. CYTOM's persistence mechanism overlaps with the execution of the evolving algorithm, as the persistence operation will not change the graph structure. However, the persistence operation cannot overlap the graph updates to avoid inconsistent snapshots. CYTOM also supports restoring the persisted graph and allows a resumption of the graph updates from the last successfully persistent snapshot, e.g., in the case of a power loss.

4.2.4 Handling Deletions

Handling edge deletions is crucial in evolving graphs and can incur correctness issues [55, 47, 56], as shown in Figure 4.5, leading to extensive corrections or re-executions of the algorithm to ensure correctness. As such, CYTOM has full support for deleting edges of the evolving graph. CYTOM accomplishes this by carrying a vector of deleted edges inside every cell to identify edges that should be skipped, as demonstrated in Figure 4.4. Additionally, when reading back a graph from a previously persisted snapshot, deleted edges are permanently removed from the graph.

Example. Figure 4.5 shows a graph where the deletion of edge ❶ leads to an incorrect algorithmic result of the connected components algorithm. To handle this issue, CYTOM introduces a simple heuristic for path-based algorithms to control when a re-execution of the algorithm will be necessary: If the deleted edge carried the result (e.g., the connecting edge for a breadth first search), the edge is *critical* to the result of the algorithm and a re-execution of the algorithm is necessary to ensure correctness. If an edge did not carry the algorithmic result, deleting that edge does not change the result of the algorithm and can safely be done without re-executing the algorithm. This is an efficient strategy, as there is only a limited number of edges that can potentially carry the algorithmic result, i.e., one per vertex for path-based algorithms. Thus, the number of critical edges is only as large as the set of vertices V, which, in many real-world graphs, is much smaller than the overall number of edges E, reducing the chance of an algorithm re-execution. This strategy also only needs to store information of the size of $\mathcal{O}(V)$, a single identifier per vertex, to identify the critical connecting edge.

4.2.5 Batching of Graph Updates

Similar to other systems for processing evolving graphs, CYTOM allows the batching of graph updates to achieve higher throughput. This is relevant for many real-world applications in which eventual consistency coupled with higher performance is desirable (e.g., social

Table 4.3: The APIs and callbacks exposed by CYTOM to algorithm developers. Figure 4.6 shows an example usage of these APIs to implement a connected component analysis.

	Function	Description
API	`activate(v)`	Activates v for the next iteration
	`markCritical(src, tgt)`	Marks src as the critical edge of tgt
	`isCritical(src, tgt)`	Checks if src is the critical edge of tgt
	`reExecuteAlgorithm()`	Reset all vertex values
Callbacks	`pull/reduce/apply`	(Static) Graph computation
	`edgeChanged(edge, event)`	Handles update for given edge and event

networks [92]). CYTOM supports any batch size from a single edge up to millions of edges per batch; however, as our evaluation shows (see §4.4.2 and Figure 4.4.3), CYTOM achieves a high throughput with batch sizes of $1{,}024$ edges (for insertions) and $65{,}536$ edges (when also running an algorithm).

4.2.6 Implementation

We implement CYTOM in C++ in about $10{,}700$ lines of code of which about 750 lines are dedicated for implementing CYTOM's algorithms. CYTOM's components communicate via a custom ring buffer implementation that allows for high throughput and a decoupling of the individual components.

4.3 Execution Model

Using CYTOM's cell model, we show how algorithms execute on snapshots of the evolving graph using CYTOM's APIs specifically tailored for evolving graphs.

4.3.1 Algorithmic Interface

CYTOM proposes an API modeled after the gather-apply-scatter (GAS) model [28, 23] in which edge values are read from a *current* vertex array and updated in a *next* array, ensuring bulk-synchronous-parallel (BSP) semantics [93, 94]. CYTOM extends this model to the setting of evolving graphs with the `edgeChanged` interface and employs a hybrid edge-

centric (operations within a cell) and vertex-centric (global operations and non-associative steps) model. This allows for reacting to new edges, the deletion of edges, or a change in the edge metadata (e.g., the weight). Additionally, some algorithms might converge to an incorrect result in the presence of edge deletions [55, 56], necessitating a re-execution of the algorithm. The functions exposed by CYTOM are listed in Table 4.3, including the `edgeChanged` interface allowing the algorithm to quickly react to local, edge-centric *events*. Possible values for the event are: *insertion, update,* or *deletion* (updates are possible for weighted graphs). CYTOM incorporates an optimization for deleting edges that are *critical*, which can impact the algorithm's correctness while discarding most of the edges as non-critical cases for which a deletion does not necessitate a re-execution of the algorithm.

We exemplify the API used by CYTOM in Figure 4.6 along with the `edgeChanged` interface with a simple algorithm to compute the connected components (CC) of a graph. As a general flow, CYTOM operates on cells as the smallest scheduling unit. Inside a cell, the algorithm executor will execute the *pull* function on every (active) edge to compute a result in a temporary result buffer. After all edges of a cell have been processed, this result buffer will be atomically *reduced* onto the global vertex array. The use of an atomic operation is necessary, as all algorithm executors operate in parallel. On conflict, the reduce operation has to be repeated to ensure a correct result. Finally, the *apply* function is used to activate non-converged vertices in a single, *non-associative* step. The algorithm converges once all vertices are inactive.

Algorithms. We implement five algorithms: *breadth first search* (BFS), *single-source shortest paths* (SSSP), *pagerank calculation* (PR), finding *connected components* (CC), and *single-source widest path* (SSWP), with SSSP and SSWP operating on a weighted graph. Table 4.4 gives an overview of the implemented algorithms. All algorithms use the `edgeChanged` API to reduce computation whenever possible when the algorithmic result does not change significantly on an edge update. We use a variant of pagerank outlined in PowerGraph [23], which only propagates changes in the value of a vertex, skipping the

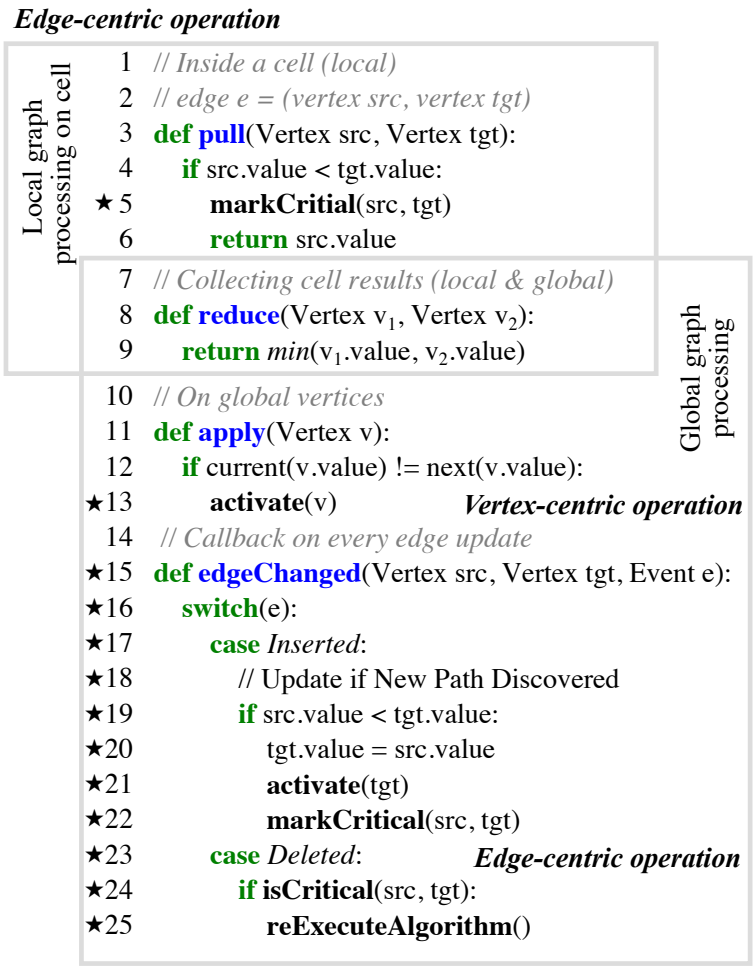


Figure 4.6: The connected components algorithm using CYTOM's API. CYTOM includes cell-local APIs as well as global, vertex-level APIs general enough to implement common graph algorithms. Additionally, the `edgeChanged` API can significantly reduce the amount of work done after an edge is changed. The base algorithm can be implemented similarly compared to computing on static graphs and CYTOM adds the lines marked with ★.

Table 4.4: The algorithms implemented in CYTOM with their associative and commutative *reduce* operators and the lines of code, especially for the `edgeChanged` interface added by CYTOM.

Type	Algorithm	*Reduce* operator	Lines of code	
			Total	*EdgeChanged*
unweighted	PR	+	176	31
	BFS	`min`	138	25
	CC	`min`	148	25
weighted	SSSP	`min`	140	26
	SSWP	`max`	140	27

computation on inactive edges.

4.3.2 Selective Scheduling

Especially for evolving graphs, it is common for large parts of the graph to already have converged during the execution of an algorithm with only a portion of the vertices remaining active. State-of-the-art frameworks for processing static graphs (e.g., GridGraph [18] and Mosaic [6]) incorporate an optimization, called *selective scheduling*, which schedules only the *active* parts of the graph. A key enabler for this technique is the use of a submatrix structure of the adjacency matrix; consequently, CYTOM is able to incorporate this optimization. Furthermore, CYTOM also skips the computation on inactive edges (as defined by the algorithm) to further reduce the computational work. CYTOM is the first system processing evolving graphs to take advantage of this optimization as a result of its cell-oriented design.

4.3.3 Load Balancing and Cell Distribution

Load balancing is a common concern when processing real-world, skewed graphs. While other systems approach this problem with vertex-based load-balancing schemes [47, 46], CYTOM is uniquely positioned to take advantage of the two-dimensional partitioning of its cells to achieve load balancing. However, the two-dimensional nature of CYTOM's cells also poses a problem in terms of scalability, as the number of cells grows quadratically with the number of vertices. We present CYTOM's approach for load balancing before exploring CYTOM's approach to dealing with a large number of cells.

Load balancing. A classic option for load balancing is *static partitioning* of the cells. It does not rely on any synchronization, but, suffers from stragglers [95] due to the dynamic, evolving graph and algorithm-dependent behavior. For static graphs, work stealing can be a solution [5, 96]; however, CYTOM chooses a different approach because of the synchronization overhead of work stealing, which can be effective on static graphs, whereas evolving graphs often include smaller sets of active vertices, necessitating a quicker intervention.

We prototype a *work-queue*-based approach and atomically assign cells to executors. This approach mitigates most stragglers; however, it is not free of them: Real-world graphs exhibit a skewed distribution in cell sizes (shown in Figure 4.7), inducing stragglers. Furthermore, atomically incrementing a shared variable can become a bottleneck [97].

Finally, we avoid synchronization issues while mitigating load imbalance with the *cell distributor*. It assigns batches of cells in a single-threaded manner once a threshold of cells has been reached, similar to the work-queue approach. However, the cell distributor also tracks the number of edges in each batch of cells and assigns a batch with less cells than the threshold if a significant number of edges are present in a single batch (set to $\frac{1}{128}$ of all edges in the graph). This results in a load-balanced system, avoiding stragglers.

Mitigate large number of cells. We explore an optimization of the cell distributor mitigating potential scalability issues, using a *hierarchical cell-distribution* scheme. The number of cells grows quadratically with the number of vertices (divided by the size of each cell, e.g., 2^{16}), but, many cells will not contain any edges because of the highly skewed nature of real-world graphs. We thus propose a *two-level* cell distribution scheme by introducing a hierarchical scheduling scheme. For this, the space of cells is subdivided into a small number of *abstract cells* (e.g., $1{,}024$). The hierarchical cell distributor first enumerates all abstract cells and checks if they can be skipped, saving up to 85% of cell explorations.

4.3.4 Optimizations Enabled by the Cell Model

We explore two optimizations enabled by CYTOM's cells.

Cell batching. As introduced in §4.3.1, results from all edges in a cell are first collected in a temporary buffer before (atomically) reducing them to the global vertex array. However, doing this after every cell misses optimization opportunities, as multiple cells might write to the same temporary output buffer (i.e., their target vertex sets are identical). Thus, CYTOM batches cells and only reduces the temporary results onto the global array after a batch of cells has been processed, allowing for potentially reusing intermediate results.

Table 4.5: The support of different operating modes for the algorithms implemented for CYTOM.

Algorithm	BSP	Approximation	Async. Proc.	Timelapse
Pagerank	✓	✓	-	✓
BFS	✓	-	✓	✓
CC	✓	-	✓	✓
SSSP	✓	-	✓	✓
SSWP	✓	-	✓	✓

Cell traversals. Different traversals of CYTOM's cells allow further optimizations, as the edges' processing order is not fixed and does not influence the algorithms' correctness. CYTOM implements three traversal directions: *column-first* (read-optimized), *row-first* (write-optimized), and the *hilbert* order [69] (both). However, while the hilbert order was shown to be an effective mechanism to improve locality and overall execution speed for static graphs [66, 6], our evaluation shows the row-first traversal to be the most effective (see Figure 4.4.3). This is attributed to fundamental differences in the execution characteristics: While static graphs have a large amount of work in every iteration, processing evolving graphs results in less work every time the algorithm is invoked, as only the recently updated edges have to be processed. In fact, with a million edge updates and the associated vertex data, only 12 MB of the LLC have to be used due to CYTOM's efficient encoding scheme. This allows the row-first traversal to achieve better store cache locality and results in better overall performance (see Figure 4.4.3).

4.3.5 Specialized Operating Modes

CYTOM supports specialized operating modes. These include approximate analysis as well as asynchronous processing, two modes that are of special importance in the setting of evolving graphs, as they allow significant reductions in computational work and can be used to cut down on uninteresting updates at the time of an edge update. We demonstrate the applicability of these techniques for evolving graphs and show the algorithms supporting each mode in Table 4.5.

Approximate graph analytics. First, we prototype an approach to approximate graph

processing with CYTOM's `edgeChanged` API and the pagerank algorithm to allow for a trade-off between execution speed and the quality of the algorithmic result. This is an emerging area of interest for static graph processing [98, 99] and, as discussed, carries even larger implications for evolving graphs, potentially reducing work on edge updates. In CYTOM, we demonstrate this using the pagerank algorithm. Whenever an edge is updated, CYTOM only propagates the result if it changes the pagerank value of a vertex by at least a factor of δ (default: 0.01). We explore the effect of varying δ in §4.4.5 and show that a speedup of more than $60\times$ is possible with a reasonable approximation compared to the non-approximated case.

Asynchronous processing. CYTOM furthermore supports asynchronous processing of a subset of algorithms, i.e., BFS, SSSP, and CC. For this, the barrier at the end of an iteration is removed and a periodic convergence checker runs to terminate the execution. Additionally, both the input and output value of an edge are read and updated in the *next* array, allowing a quicker dissemination of results than with synchronous, BSP-style processing.

Supporting timelapse-based analytics. Finally, we introduce the ability to take snapshots of the evolving graph as well as the associated algorithmic result, persisting them on disk for further analysis. This mode allows popular analysis tasks such as "how has the pagerank of a specific webpage changed over time?" similar to systems like Kineograph [50] and Chronos [49]. In addition, CYTOM can be extended with per-edge timestamps to produce snapshots of the graph and allow for ad-hoc analysis scenarios [100, 54]. These timestamps can be stored in CYTOM's cells and are persisted to disk as well, increasing CYTOM's memory footprint by 50% (16-bit timestamps) to 100% (32-bit timestamps). However, CYTOM still retains its favorable memory footprint over the traditional, edge list-based approach.

4.4 Evaluation

We evaluate CYTOM by answering the following questions:

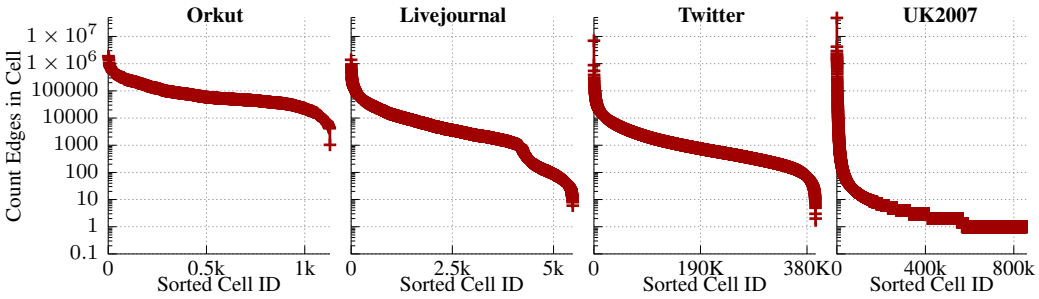


Figure 4.7: The distribution of cell sizes for the real-world graphs used in CYTOM. As CYTOM's cells can vary widely in size due to the nature of real-world, skewed graphs, load-balancing is an important consideration for CYTOM's cell distributor.

Table 4.6: The graphs used in the evaluation of CYTOM and their characteristics (⋆marks synthetic data sets). CYTOM reduces the data size by 48% due to its short, 16-bit identifiers.

Graph	#vert.	#edges	Data size		#Cells	#edges in cell	
			Raw	CYTOM		avg	stddev
⋆**rmat-22**	4.2 M	67.1 M	512.0 MB	264.0 MB	4.1 K	16.4 K	(64.9 K)
livejournal	4.8 M	69.0 M	526.4 MB	271.4 MB	5.5 K	12.6 K	(42.7 K)
orkut	3.1 M	117.2 M	894.1 MB	461.0 MB	1.1 K	103.9 K	(151.8 K)
twitter	41.7 M	1.5 B	10.9 GB	5.6 GB	395.4 K	3.7 K	(34.2 K)
uk2007-05	105.9 M	3.7 B	27.9 GB	14.4 GB	859.1 K	4.4 K	(158.9 K)

- How does CYTOM compare to other systems processing evolving graphs in terms of throughput and latency?
- How do each design decisions of CYTOM impact its performance, including CYTOM's cells, its persistent mode, and the `edgeChanged` interface?
- How versatile is CYTOM with other processing modes like approximate or asynchronous processing?

4.4.1 Evaluation Setup

Machine. We run CYTOM on a two-socket, 24-core machine (E5-2670 v3 2.3 GHz) with 256 GB of RAM. We run all experiments with an SSD (maximum transfer rate of 470 MB/s) and empty the buffer cache before every experiment.

Datasets. We list the graphs used in the evaluation with their characteristics in Table 4.6.

The synthetic graphs are generated using the R-MAT generator [75] using the configuration of the graph500 benchmark.[3] We use social network-based graphs (livejournal [101], orkut [102], and twitter [76]) as well as a Web graph (uk2007-05 [77, 78]). These graphs range from a few million vertices and edges up to a hundred million vertices and more than 3 billion edges with a raw data amount of 0.5 GB up to 30 GB. We run all algorithms on all real-world graphs where possible but restrict the evaluation to the two smallest, real-world graphs in cases where the runtime for the billion-scale graphs would be excessive (e.g., very small insertion batches in Figure 4.4.3).

Methodology. We run all experiments five times and report the average. The predominant metric is the throughput of graph updates, either with or without running an algorithm (as specified). Where possible, we run other systems ourselves (e.g., STINGER [46] and GRAPHONE [48]) but take numbers from respective papers if the system is not available (e.g., GraphIn [47]). If not specified, we use a default batch size of 1 million updates and run all algorithms to convergence or for 100 iterations, whichever occurs earlier.

4.4.2 Overall Performance

We first investigate CYTOM's graph storage and its overall performance, the algorithmic performance, and how CYTOM compares to other state-of-the-art systems for processing evolving graphs (i.e., GRAPHONE, STINGER, and GraphIn).

Performance of Storing the Evolving Graph. We first evaluate the performance of inserting and updating edges in the graph store itself. We vary the size of the batch of updates from one edge to $2^{24} \approx 16M$ edges and set the deletion ratio to 1%. Note that the maximum achievable performance is about 58 million edges per second, as the maximum achievable throughput of the underlying SSD is 470 MB/s. The results in Figure 4.8 show that CYTOM with an update size of one edge achieves about 1.5 million edges per second, while increasing the batch size to 128–512 allows the full usage of the offered SSD

[3] The default parameters are $a = 0.57, b = 0.19, c = 0.19, d = 0.05$, `https://graph500.org/?page_id=12#sec-3_2`

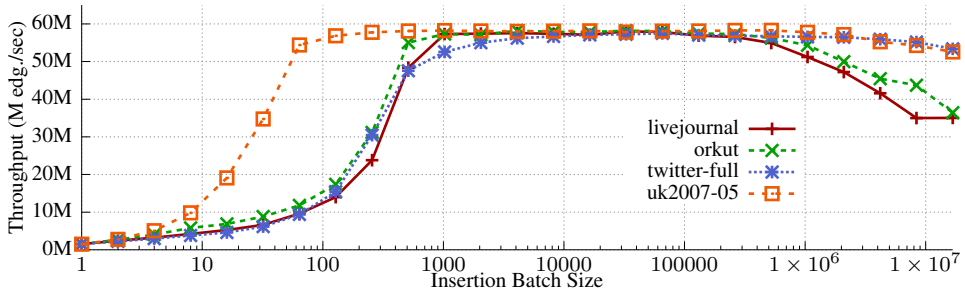


Figure 4.8: Performance when inserting edges into CYTOM without running any algorithms, varying the batch sizes from one to 2^{24} edges. CYTOM achieves up to 57 million edges per second due to exhausting the I/O performance of the underlying SSD and drops in performance for very large batch sizes due to limited parallelism.

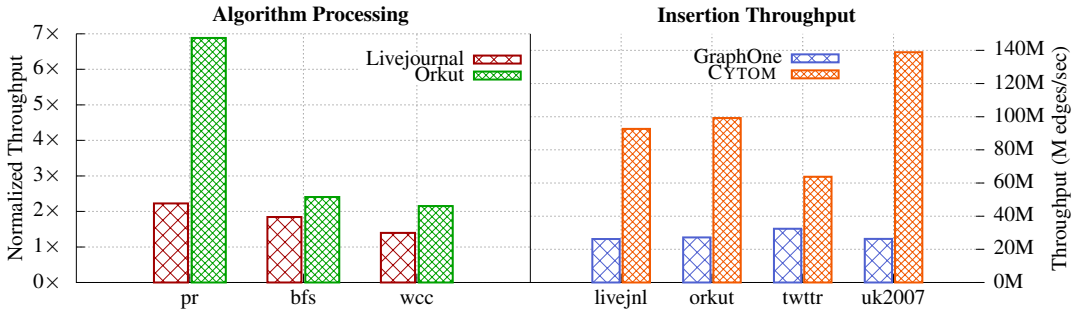


Figure 4.9: Comparison to GRAPHONE for algorithm processing and insertion throughput (without an algorithm). CYTOM outperforms GRAPHONE by up to 6.9× with PR and 5.3× during insertion as a result of CYTOM's cell-based graph representation.

bandwidth. For the smaller graphs (orkut and livejournal), the throughput drops for batch sizes larger than 1M updates, as the parallelism achievable inside the graph is limited due to the very few batches necessary to insert the entire graph (i.e., five batches for the livejournal graph at a batch size of 16 million).

Comparison to GRAPHONE. We compare CYTOM to GRAPHONE [48], a recent system optimizing the throughput of graph updates. We run CYTOM in an in-memory mode for this experiment, similar to the setup GRAPHONE is designed for. As shown in Figure 4.9, CYTOM outperforms GRAPHONE by up to 6.9× when running PR with smaller improvements for less complicated algorithms. Additionally, CYTOM is able to support up to 140 million graph updates per second (without running any algorithm), outperforming GRAPHONE by up to 5.3× for large graphs.

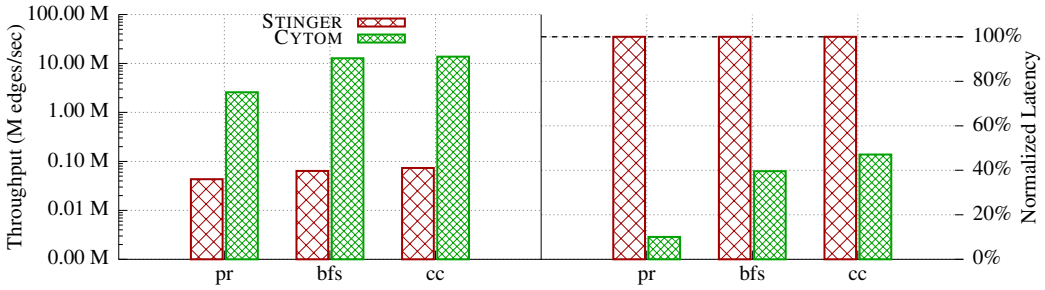


Figure 4.10: Comparison to STINGER on an rmat-22 graph with 4M vertices and 67M edges. CYTOM achieves a maximum speedup of 192× in terms of edge update rates and at least 60× for the algorithmic performance. CYTOM furthermore reduces the algorithm's latency by up to 89.9%.

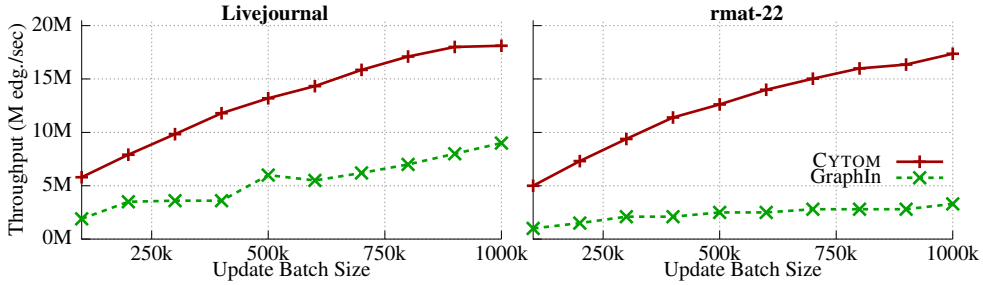


Figure 4.11: Comparison of the size of the insertion batch used in CYTOM and its effect on the overall throughput compared to GraphIn. CYTOM improves on GraphIn's performance by up to 5.5× and consistently outperforms GraphIn with both small and large batch sizes.

Comparison to STINGER.

We compare CYTOM to STINGER, a popular open-source system for processing evolving graphs. STINGER processes evolving graphs with atomic updates of its adjacency list. We compare STINGER's performance and algorithmic latency in Figure 4.10 with a small-scale rmat-22 graph due to STINGER's limited scalability. In comparison to CYTOM, STINGER does not include APIs to accelerate and skip uninteresting updates and suffers from a poor update rate due to the use of many atomic operations as well as long traversals to insert new edges at the end of its adjacency lists. As a result, CYTOM is able to outperform STINGER by a factor of up to 192× when purely updating edges and 60× to 200× when also running an algorithm while updating the graph as well as reducing the latency for algorithmic updates by up to 89.9%.

Comparison to GraphIn. We also compare CYTOM to GraphIn [47], a system optimizing

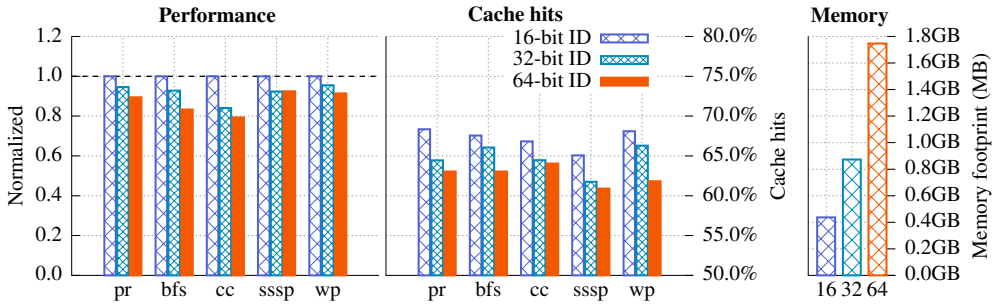


Figure 4.12: Performance and locality of CYTOM compared to a version of CYTOM using longer identifiers (32 or 64 bits) on the orkut graph. CYTOM performs up to 24% better using 16-bit identifiers due to better cache hits as well as lowered memory footprint.

edge updates with an incremental, GAS-like programming model. Unfortunately, GraphIn is not open-sourced; however, we obtain GraphIn's numbers from its paper given that both systems run on a relatively comparable system configuration. We show the comparison in Figure 4.11, which demonstrates that CYTOM can outperform GraphIn by at least 2.1× and up to 5.5× due to CYTOM's efficient graph representation and better load balancing. Additionally, GraphIn only scales to graphs with up to 70 million edges due to its expensive programming model. CYTOM on the other hand is designed to scale to billions of edges.

4.4.3 Evaluating Taken Design Decisions

We now evaluate specific design decisions taken in CYTOM to see their impact on the overall performance.

Compression and locality. We first evaluate CYTOM's decision for short 16-bit identifiers: The results in Figure 4.12 show that CYTOM with 16-bit identifiers achieves up to 25% better performance, up to 5 percentage points better cache locality, while saving up to 75% of memory when compared to a version with 32-bit or 64-bit identifiers. We do not include 8-bit identifiers as they result in more than a billion cells (compared to 16K cells with 16-bit identifiers) and are not runnable for even the smallest real-world graph evaluated.

Traversals. As introduced in §4.3.4, CYTOM's cells enable the use of different traversal strategies, three of which are evaluated here: row-first, column-first, and the hilbert order

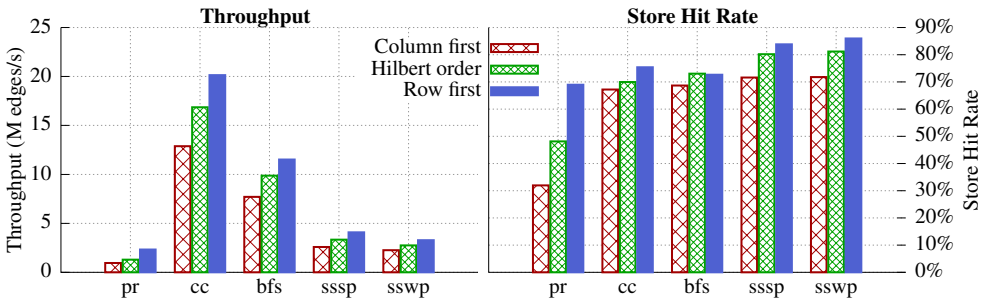


Figure 4.13: Different traversal strategies and their throughput, using the livejournal graph. The row-first, write-optimized strategy shows the highest throughput, as it induces significantly better cache hit rates for store instructions compared to other traversals.

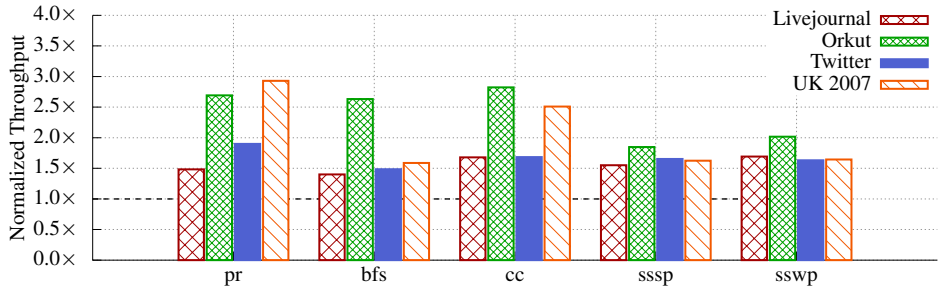


Figure 4.14: Impact of CYTOM's `edgeChanged` callback on the overall performance. Using CYTOM's APIs, the overall throughput improves by up to 2.9× due to the reduction in overall work and the potential for skipping uninteresting updates.

traversal. The results in Figure 4.13 reveal that CYTOM performs best using the row-first, write-optimized traversal. This surprising result is rooted in the nature of evolving graphs. Only very little data needs to be processed in every iteration of the evolving algorithm (e.g., even 1 million updates result in less than 12 MB of edge and vertex data), allowing it to fit into the cache. This in turn allows a write-optimized approach to work well, as it achieves significantly better cache locality for stores, as seen in Figure 4.13, while the overall cache locality remains similar between all traversals. This allows an improvement in throughput by up to 2.4× (PR) when using the row-first traversal.

Impact of edge APIs. CYTOM uses the `edgeChanged` callback to allow on-the-fly updates to the algorithmic result, while filtering out uninteresting updates. We evaluate the impact of this callback by comparing CYTOM to a version of CYTOM that simply activates any edge that is updated. The results in Figure 4.14 demonstrate that CYTOM achieves a speedup

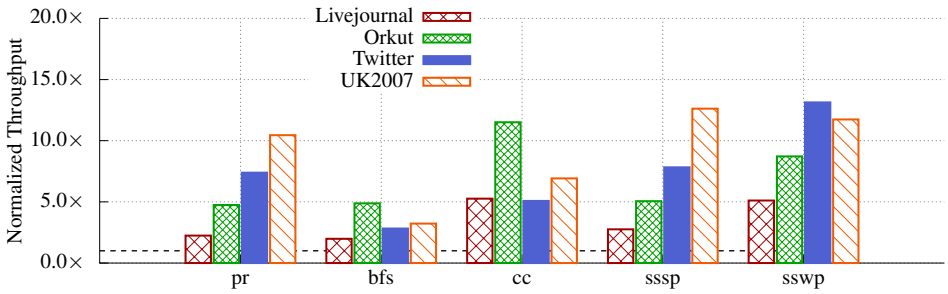


Figure 4.15: Comparison between incrementally executing the algorithm and re-executing the algorithm after every insertion. The incremental execution is up to 13.1× faster than re-executing, with larger benefits for more complicated algorithms and larger graphs.

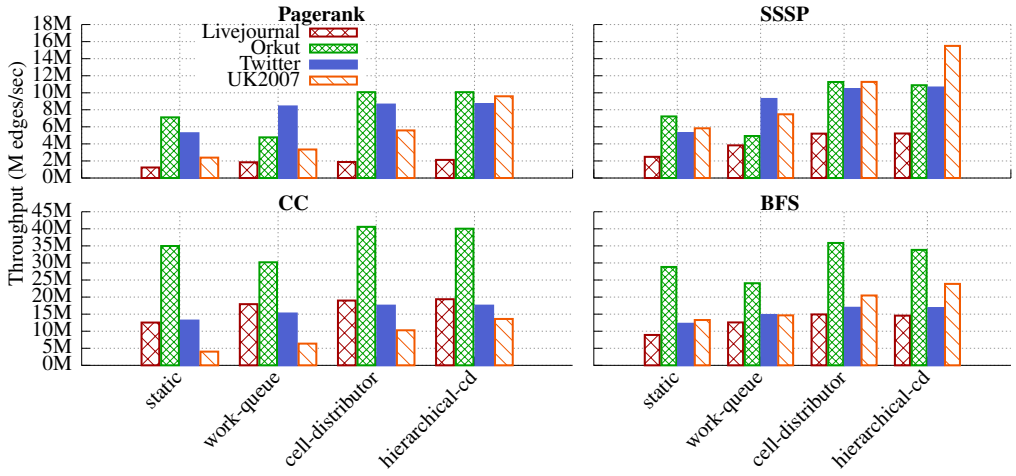


Figure 4.16: Comparison of the cell distribution mechanisms used in CYTOM. The hierarchical cell distributor improves the overall throughput by up to 4.0× over a static partitioning due to the improved load balancing and mitigated straggler problem.

of up to 2.9× due to the reduced amount of work to be done while still ensuring a correct algorithmic result.

Incremental execution. We compare CYTOM to a version of CYTOM that simply re-executes the algorithm from scratch after every batch of edge updates has been processed. The results in Figure 4.15 show the importance of the incremental execution scheme, as the overall throughput increases by 2.1×-13.1× when switching to CYTOM's incremental execution model, including its `edgeChanged` callback.

Cell distribution scheme & load balancing. We evaluate the importance of a dedicated cell-distribution scheme in place of a simple static assignments of cells to threads. For this,

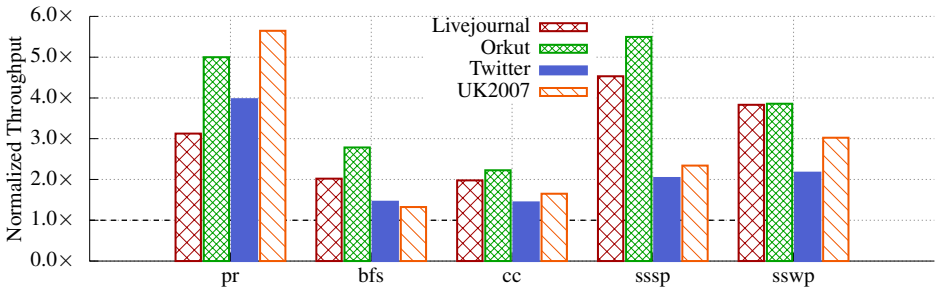


Figure 4.17: Impact of CYTOM's selective scheduling, which uses the active state of each edge to determine which cells to process. Selective scheduling improves the performance by up to 5.6×.

we implement four different cell-distribution mechanisms: A *static* partitioning approach, a load-balancing, *work-queue*-based approach, a *cell distributor*-based approach using a dedicated thread to distribute cells to all executors while capping the amount of work per executor, and a *hierarchical* cell distributor that uses two levels of partitioning to focus on active subsets of the graph. All four options are shown in Figure 4.16 and demonstrate the importance of the dedicated cell distributor, as it improves performance by up to 4.0× compared to the static partitioning strategy. Additionally, the hierarchical cell distributor is shown to perform better with larger graphs, with more opportunities for skipping uninteresting parts of the graph.

Selective Scheduling. To further skip parts of the graph without activity, CYTOM provides *selective scheduling* of its cells, skipping cells without any active edges, as defined by each algorithm. Selective scheduling plays a crucial role in processing evolving graphs due to the small number of *active* vertices after each batch of graph updates. As shown in Figure 4.17, this allows speedups of up to 5.6×, as the amount of work per iteration is reduced once large parts of the graph have become inactive, skipping up to 95% of cells due to inactivity (PR). Selective scheduling is more important in algorithms that are slower to converge (e.g., PR) and weighted algorithms (e.g., SSSP, SSWP), as these cases can benefit significantly from a reduced volume of data to be processed.

Batching. CYTOM utilizes batching at two levels (see §4.2.5 and §4.3.4): when updating edges and when executing the algorithm on the cells. On graph updates, every batch of

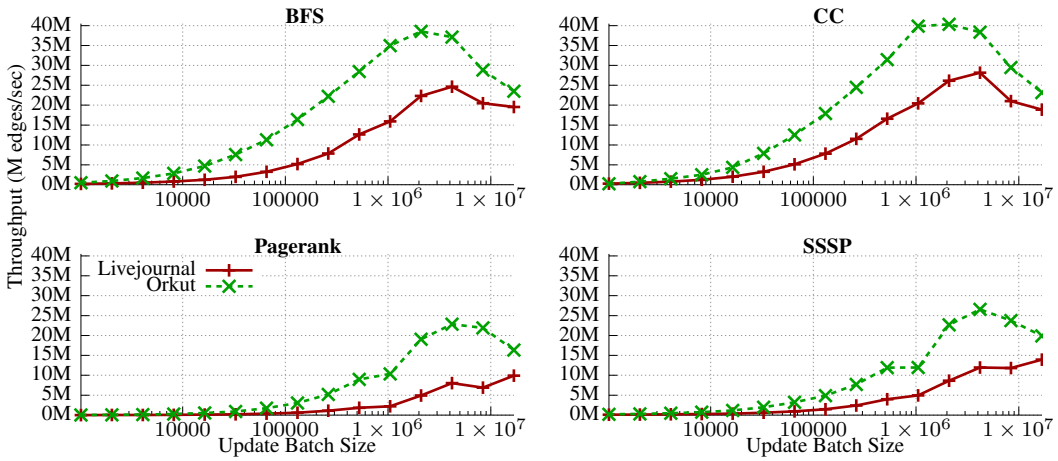


Figure 4.18: Impact of the chosen size of insertion batches on the overall throughput. With batch sizes larger than 1M, CYTOM reaches a throughput of up to 40 million updates per second. As the parallelism is limited with very large batch sizes, the throughput drops.

edges triggers an execution of the evolving algorithm with small batch sizes, inducing a large overhead on the throughput of graph updates. Compared to batch-oriented systems (e.g., GraphChi or X-Stream), CYTOM is able to support fine-grained insertion of edges. We vary the batch size from 1024 to 16 million in Figure 4.18, demonstrating CYTOM's ability to support both fine-grained as well as large-volume updates. CYTOM is able to reach up to 40 million updates per second with large batches suffering from limited parallelism.

CYTOM also uses cell batching when running the algorithm. This reduces the synchronization of results to the global vertex array done after completing the execution on a batch of cells. We show the effect of varying the size of the cell batch from one to 1024 cells using the livejournal graph in Figure 4.19. This shows that cell batching improves the throughput by up to 63% with a batching size larger than eight cells and we select 128 cells as the default batch size.

Impact of deletions. We analyze the impact that varying the percentages of deletions has on CYTOM's algorithmic performance with CYTOM's optimization of *critical* edges (see §4.2.4). The results, shown in Figure 4.20, show that with an edge deletion ratio of 1%, CYTOM is able to reduce up to 84% of re-executions and improves the algorithm throughput by up to 4.0× compared to the baseline of re-executing to ensure correctness. CYTOM shows

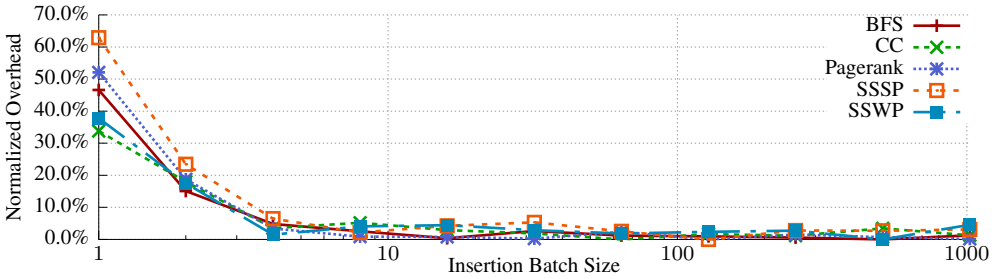


Figure 4.19: Comparison of the size of the cell batching used in CYTOM and its induced overhead compared to the maximum, normalized performance on the livejournal graph. Cell batching improves the algorithm performance by up to 63%. Using this analysis, we choose a default value of 128 for cell batching.

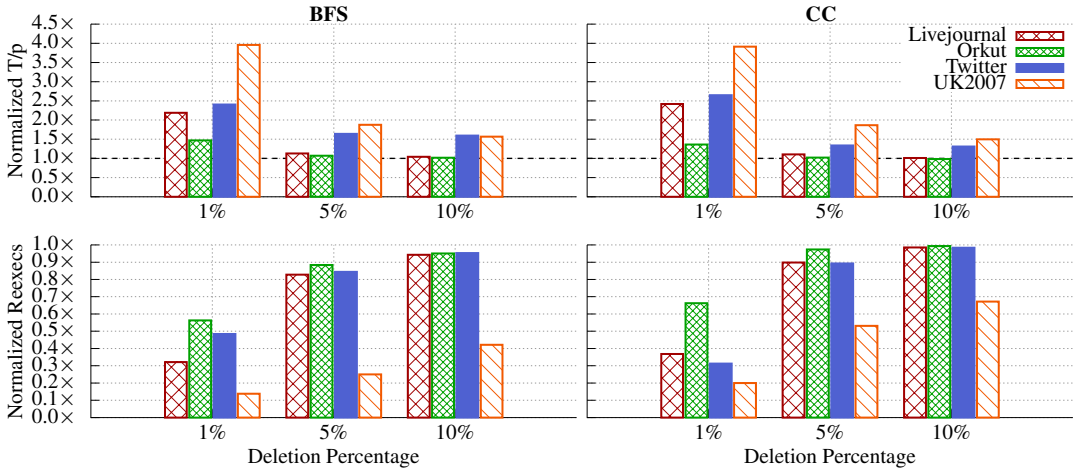


Figure 4.20: Impact of edge deletions on the throughput compared to re-executing the algorithm to ensure correctness. CYTOM's approach of re-executing only when changing a *critical* edge skips up to 84% of re-execution and achieves up to 4.0× the throughput compared to the baseline of re-executing. Larger percentages of deletions have a higher probability of hitting a critical edge, resulting in lower performance.

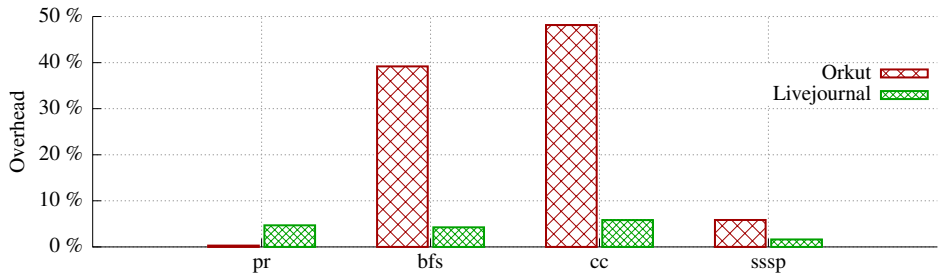


Figure 4.21: Impact of enabling the persistent mode of CYTOM. The persistence mode incurs a significant reduction in throughput due to exhausting the available disk bandwidth for both reading the graph from disk as well as writing the processed edges back to the same medium.

smaller improvements with larger percentages of deletions as the probability of deleting critical edges rises; however, it still retains advantages even for these cases. CYTOM's critical edge optimization is more effective on larger graphs due to the reduced chance of hitting a critical edge on deletion as well as the increased cost of re-executions for larger graphs (as seen in Figure 4.15).

4.4.4 Disk Persistence

CYTOM supports persisting as well as restoring the state of the evolving graph to and from disk, providing fault tolerance (see §4.2.3). The results show that the overhead induced by this mechanism ranges from close to 0% (for PR) to 48% (for CC), as CC usually converges in a small number of iterations. This does not allow for enough time for the synchronous writing of updates to disk to complete, thus stalling the updates to the graph, resulting in higher overhead. Nonetheless, persistent CYTOM still retains a competitive performance compared to other systems that run *without* persistence.

4.4.5 Evaluating Alternative Operating Modes

Approximate analytics. To show the effects of approximation on the quality of the algorithmic result, we run the following experiment: We vary the δ value of PR, controlling at what tolerance the algorithm terminates. In evolving graphs, δ determines whether or not an edge is "important" when inserting it by testing if this additional edge changes the

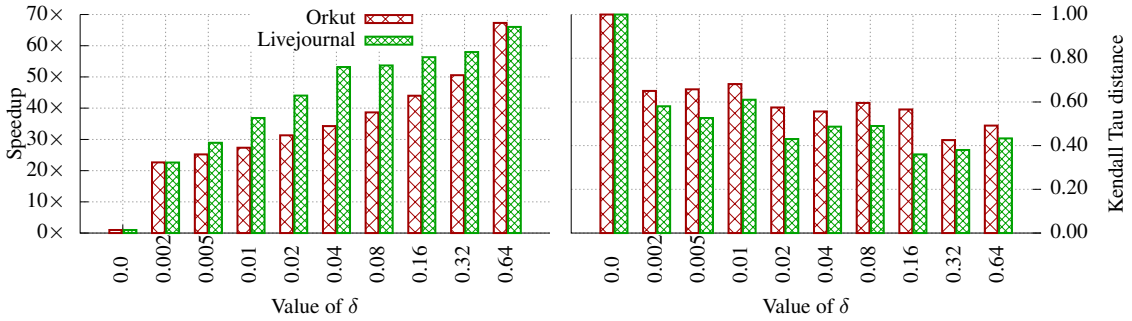


Figure 4.22: Impact of changing δ for PR on the throughput and the associated *Kendall tau distance* in the associated vertex ranking with an increasingly larger approximation. A larger δ value allows speedups of more than 60× while we choose a value of $\delta = 0.01$ as the default, balancing speedup and approximation.

Table 4.7: CYTOM's throughput in an asynchronous mode compared to BSP-style processing when processing the livejournal graph. We also show the reduction in the number of iterations (synchronous mode) or convergence checks (asynchronous mode). The asynchronous mode is up to 3.2× faster while saving up to 78.9% of iterations.

Algo.	Tput (M edges/s)			#Iteration		
	Sync.	Async.	(±%)	Sync.	Async.	(±%)
BFS	17.69	24.55	(+38.8%)	145	64	(-55.9%)
CC	22.29	28.60	(+28.3%)	125	35	(-72.0%)
SSSP	6.01	19.78	(+229.3%)	323	68	(-78.9%)

algorithmic result by more than δ. We vary δ from 0 (no approximation) to 0.64 and show the effect on both the algorithm speedup (run to convergence) as well as the algorithmic result in terms of the *Kendall tau distance* [103] in Figure 4.22. The Kendall tau distance is a metric to compare two ranking lists and is 1 when both lists are equal and 0 if they are inverted. A speedup of more than 60× can be obtained by using a larger δ; however at the expense of the quality of the algorithmic result, although we note that the top-ranked vertices are closely matched regardless of the approximation strategy. Using this analysis, we decide for all other experiments to use a δ value of 0.01, as it balances the overall speedup with the quality of the algorithmic result.

Asynchronous processing. CYTOM supports asynchronous processing for a subset of algorithms (see Table 4.5), including BFS, CC, and SSSP, demonstrating CYTOM's versatility. We show the results of running CYTOM in its asynchronous mode in Table 4.7,

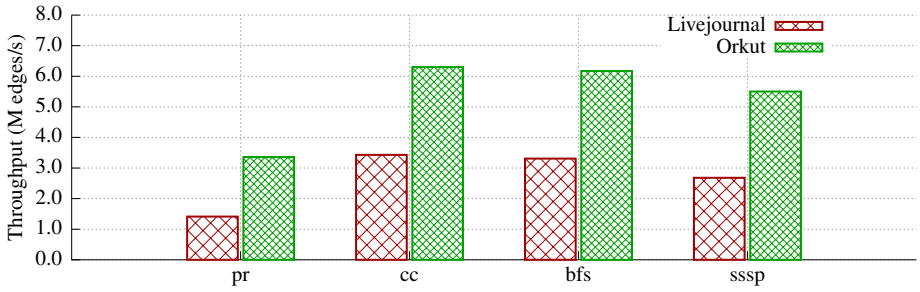


Figure 4.23: CYTOM's throughput when the snap-shotting mode is enabled. The snap-shotting mode runs faster on the Orkut graph as it has only about half the number of vertices compared to livejournal, resulting in lower overhead when writing to disk.

demonstrating that it improves performance by up to 3.2× as a result of a reduced number of iterations to convergence. SSSP benefits the most due to its larger data size, with the added weight per edge. Additionally, more iterations are saved for SSSP compared to both BFS and CC due to the slower convergence of SSSP.

Timelapse-based analytics. In addition, CYTOM allows the analysis of intermediate results of the evolving graph. This allows new analysis scenarios such as observing the evolution of the pagerank of a specific vertex over time, similar to the API provided in other systems such as Kineograph [50] or Chronos [49]. CYTOM maintains a reasonable performance with up to 6.2 million updates per second for CC. The performance of snapshotting with CYTOM is depicted in Figure 4.23, demonstrating the versatility of CYTOM for this kind of analysis.

4.5 Chapter summary

We present CYTOM, a graph-processing engine for evolving graphs based on the abstraction of cells, allowing a significant reduction in overall storage space as well as enabling a simple, yet effective load-balancing strategy. Additionally, CYTOM's APIs, including the `edgeChanged` interface, are tailored toward processing evolving graphs and its incremental processing mode provides significant benefits, outperforming other state-of-the-art engines for evolving graphs by 1.5×–200×. Finally, as a result of its design decisions, CYTOM

allows processing billion-scale real-world evolving graphs.

CHAPTER 5
DISCUSSION

In this chapter, we discuss the applicability of the techniques developed in MOSAIC and CYTOM to other systems as well other kind of system architectures (e.g., distributed systems). Additionally, we discuss a few other, related topics, in particular the amount of overhead added by MOSAIC and CYTOM compared to a single-threaded implementation of a graph algorithm before concluding with the key lessons learned by designing both MOSAIC and CYTOM.

5.1 Applicability of Techniques

MOSAIC as well as CYTOM propose a new programming interface for processing static, respective evolving graphs in addition to the data structures proposed to store and process both a static as well as an evolving graph.

5.1.1 Data Structures

Both MOSAIC and CYTOM base its data structure on a subgraph-centric approach. However, CYTOM's data structure design is targeted at removing the overheads associated with keeping metadata to map a local subgraph to its global counterpart. As such, CYTOM's data structure design can form the basis of other systems processing evolving graph, especially with a different programming model. As discussed in §4.4.3, CYTOM not only shows great performance benefits when run with its native, 16-bit based identifiers, but is also applicable when using longer, 32-bit or 64-bit identifiers. This in turn allows CYTOM's data structure design of using a cell-based approach to be applicable to other systems with an edge list-based design (e.g., GraphOne [48] or Graphin [47]).

5.1.2 Interface for Processing Static Graphs

For processing static graphs, MOSAIC's interface is well-suited at giving the algorithm developer enough flexibility to implement a wide range of common graph algorithms as it is based on the popular and well-explored GAS (Gather, Scatter, Apply) model pioneered by PowerGraph [23].

5.1.3 Interface for Processing Evolving Graphs

For processing evolving graph efficiently, CYTOM introduces a new API, the `edgeChanged` interface. This interface allows the algorithm developer to quickly cut down on the set of updated edges to remove uninteresting updates which would not result in a change in the algorithmic result (e.g., an inserted edge connecting two vertices which are already in the same connected component). The `edgeChanged` interface is thus not only applicable to CYTOM but can be ported to other systems processing evolving graphs as well to cut down on the amount of work to be done after every graph update. In particular, this API is not restricted to the use case of processing evolving graphs on a single machine only but can also be applicable to distributed systems for evolving graphs (e.g., GraphTau [54], Kickstarter [55], or Aspire [88]). The `edgeChanged` interface is of special interest as it does not rely on CYTOM's conventional, graph processing API and as such is already decoupled and applicable to other systems processing evolving graphs.

5.2 Overhead over Single-Threaded Implementation

Given their focus on optimizing the case of massively parallel execution, both for MOSAIC as well as CYTOM the question for how much overhead is added by the infrastructure supporting the multi-threaded computation over an optimized, single-threaded implementation is of interest. This overhead is known as the ***c**onfiguration that **o**utperforms a **s**ingle **t**hread* (COST) where a low COST (e.g., 1) indicates very low overhead for supporting the parallel

execution while a high or infinite COST indicates that no matter the number of cores, a single thread will always be faster than even a highly parallel execution [66].

5.2.1 COST for MOSAIC

For MOSAIC, we apply the COST metric to its heterogeneous execution and report the number of *Xeon Phi* cores needed to outperform a single threaded implementation. The COST [66] can be applied to MOSAIC to evaluate its inherent overhead for scalability: for the uk2007-05 graph, a single-threaded, host-only implementation (in-memory) [66] 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×. Thus, we conclude that MOSAIC has a COST of up to 31 Xeon Phi cores. Please note that while this might look like a relatively large COST, Xeon Phi cores are much less powerful than a conventional, full-featured Xeon core. In addition, MOSAIC runs in a disk-based mode while the single-threaded implementation operates in an in-memory setup.

5.2.2 COST for CYTOM

While we evaluate the COST value for MOSAIC, we also investigate the scalability behavior and the overhead added by both the parallel runtime of CYTOM as well as the overhead added by the support for evolving graphs using data structures that are be less optimized than those for a graph processing engine processing static graphs (e.g., the CSR representation). We again evaluate the *configuration that outperforms a single thread* (COST) [66] for CYTOM to determine whether the support for evolving graphs added results in a significant overhead for running (static) graph algorithms. We run both the optimized, single-threaded

implementation of pagerank and CYTOM on the machine presented in §4.4.1. In this setup, we note that CYTOM yields a COST of 3 threads for the livejournal graph while achieving a maximum speedup of 5.6×. In addition, on the orkut graph, CYTOM has a COST of 3 threads as well while achieving a maximum speedup of 6.7×. This low COST value demonstrates that CYTOM's data structures are a good fit even for static graph computations and emphasize CYTOM's lightweight support for evolving graphs, especially as CYTOM does not include any pre-processing phase compared to the COST implementation. We specifically do not include this preprocessing cost of the COST implementation in the final performance results, highlighting the performance benefits of CYTOM. Finally, it demonstrates that CYTOM can scale well when adding more threads.

5.3 Generality

We demonstrate the generality of both systems developed in this dissertation, MOSAIC and CYTOM, by investigating their applicability to a wider range of algorithms. We show that this generality is an outcome of both MOSAIC and CYTOM being designed around a familiar set of APIs, allowing the implementation of many common graph algorithms.

5.3.1 Extensibility of MOSAIC

We argue that the underlying design of MOSAIC can easily be extended to non-graph algorithms as well to support more algorithms in a heterogeneous environment. We demonstrated this by implementing k-means clustering in 122 lines of code as shown in §3.6 (the k-means algorithm is also used at Facebook [3] with the extension of using the edge cut as a distance metric). This algorithm runs in 3.89 seconds per iteration on the uk2007-05 dataset on our heterogeneous machine as introduced in §3.7.1 demonstrating MOSAIC's ability to support other, more general algorithms in an efficient manner.

5.3.2 Algorithm support for CYTOM

We demonstrate CYTOM's API-level generality by implementing five, common graph algorithms. However, CYTOM is not restricted to only these algorithms but can in fact be extended to more, and especially, more complicated algorithms as well due to its use of a variant of the popular and well-understood *Gather, Apply, Scatter* (GAS) model introduced by PowerGraph [23]. In fact, many algorithms that can form the base for machine learning applications like triangle counting [72] or alternating least squares [104] can be implemented using the GAS model and are thus applicable to CYTOM as well.

5.4 Lessons learned

Before we conclude, we will address the lessons learned by designing both MOSAIC and CYTOM to scale graph processing on a single machine to ever-larger data sets. In particular, these lessons are essential to show the statement we set out to prove, that *large-scale analytics is possible on a single machine*.

5.4.1 Compression

An important and essential lesson is that compression of the input data set is a key to being able to store and process tera-scale static datasets as well as billion-scale evolving data sets. However, while other systems (e.g., Ligra+ [105]) are using heavy-weight compression algorithms to achieve this goal, both MOSAIC and CYTOM employ a light-weight, near-zero-cost scheme to achieve a similar compression ratio.

5.4.2 Load Balancing

Load balancing is another important factor for any system processing skewed data sets, and is of special importance for graph processing [23, 26] due to its tendency of heavily skewed real-world data sets. Both MOSAIC and CYTOM address this challenge, however,

for different contexts and execution environments. While MOSAIC mainly has to cater to the static load imbalance and the need to balance load between a large number of relatively slow cores, CYTOM has to cater to a much more dynamic environment, with the load changing whenever the graph is updated.

As such, both systems incorporate components for active load balancing, e.g., MOSAIC to *split* tiles between the slow coprocessor cores, while CYTOM incorporates a novel cell distributor which allows load balancing of incoming updates to the waiting algorithm execution cores. Addressing this challenge allows both MOSAIC and CYTOM to also scale well when adding more cores, as demonstrated with the COST analysis.

5.4.3 Algorithmic Interface

Finally, both MOSAIC and CYTOM incorporate an algorithmic interface that shields the algorithm developer from all the intricate details of both systems. In fact, for MOSAIC, the algorithmic interface even hides the fact that part of the algorithm is executed on a many-core coprocessor from the algorithm developer to enable a seamless transition between the fast host processors and the slower, but plentiful cores of the many-core coprocessor. Additionally, the algorithm developer does not need to worry about the data transfer to and from the coprocessor or the low-level details of MOSAIC's tile scheme as both of these details are hidden behind MOSAIC's algorithmic interface.

Similarly, CYTOM enables a diverse set of algorithms by hiding the details of its cell scheme from the algorithm developer, focusing instead on the graph-centric aspects of the algorithm. In addition, CYTOM offers the algorithm developer the option to quickly react to graph updates using the `edgeChanged` interface. This interface allows CYTOM to realize significant speedups by excluding uninteresting graph updates which would not result in changes to the algorithmic result.

5.4.4 Putting it all together

In tandem, these techniques, coupled with additional, domain-specific optimizations (e.g., the efficient usage of fast I/O devices for MOSAIC or the incremental execution model of CYTOM) work together to allow both MOSAIC and CYTOM to scale graph data processing to the trillion-scale for static graphs and the billion-scale for evolving graphs, thereby validating the thesis statement we formulate in §1.2.

CHAPTER 6
CONCLUSION AND FUTURE WORK

This dissertation covers two aspects of scaling big data processing on a single machine for both processing tera-scale static and evolving graphs. We first summarize key contributions laid out in this dissertation before discussing avenues for future work.

6.1 Conclusion

In response to the ever-increasing size of data sets coupled with the desire to be able to efficiently process these data sets in an inexpensive setting, we propose two approaches at scaling big data processing on a single machine for both the case of static and evolving graphs.

We first introduce MOSAIC as a system to take advantage of a commodity, heterogeneous machine to process trillion-scale graphs on a single machine. MOSAIC introduces 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. Furthermore, MOSAIC is shown to be effective in processing very large graphs even using the host CPUs only and scales well when adding disk capacity and compute power.

Our evaluation shows that for smaller graphs, MOSAIC consistently outperforms other state-of-the-art out-of-core engines by 3.2–58.6$\times$ 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$\times$.

However, while static graphs can be processed efficiently, many real-world graphs are dynamic in nature. To address the problems in *evolving* graphs, namely low throughput of

graph updates coupled with a high latency to retrieve updated results, we propose CYTOM based on a subgraph-centric graph representation. This approach is effective at reducing both the storage overhead of state-of-the-art systems, as well as allowing for a highly parallel process when updating the graph structure. Additionally, CYTOM couples this subgraph-centric, *cell*-based graph representation with a novel execution model which allows to process many graph updates locally within a subgraph instead of starting a global computation step.

We show that CYTOM is effective in scaling to billion-edge graphs as well as providing higher throughput when updating the graph structure (2.0×–192×) and higher throughput (1.5×–200×) when additionally processing an algorithm.

Finally, we open source MOSAIC (`https://github.com/sslab-gatech/mosaic/`) to allow further development of large-scale graph processing on a single machine and will open source CYTOM in the future as well.

As a final note, both MOSAIC and CYTOM provide evidence to validate the thesis statement this dissertation is built around (see §1.2) which states that *large-scale big-data analytics is possible on a single machine* by demonstrating the scalability of graph processing at the trillion-scale on a single machine.

6.2 Future Work

This dissertation presents techniques for scaling computations on a single, commodity machine. However, there are a number of pointers towards future research in this space that are worth exploring. As previously discussed in §5, both MOSAIC's and CYTOM's algorithmic interface are already applicable to a wider range of systems and algorithms. As such, employing the ideas implemented into MOSAIC and CYTOM can be applicable to distributed graph processing engines as well. However, there are three additional aspects of graph processing (both static and evolving graphs) that are worthy of exploration.

6.2.1 Hardware-related optimizations

First, we take a look at optimizations that are using hardware primitives to improve the performance of graph processing. Optimizing graph processing for a setting of specific accelerators has recently become a topic of interest[106, 107, 108, 109, 110, 111] due to the diminishing returns in terms of performance offered by new general-purpose architectures. However, many of these projects focus on the usage of FPGAs or custom ASIC solutions to improve the performance of graph analytics on *static* graphs in particular. Additionally, the challenges (as hinted at with the Emu Chick prototype [106]) for processing evolving graphs in hardware are even more significant due to the limited amount of memory available on these accelerators.

As such, one potential accelerator for the setting of both static as well as evolving graphs is a simple optimization of the cache: When processing graphs, the algorithm data (and thus the output) is stored at a vertex-level granularity while the edges simply define the topology of the graph. Currently, there is no distinction between either of these data structures and both are handled equally when it comes to which data should be stored in the CPU's caches. However, the observation is that most of the topology data (i.e., the edges) is accessed in a streaming, read-once fashion without making actual use of the cache. The vertex data, however, is accessed much more frequently and thus benefits from a cache. From an accelerator point-of-view, allowing a selective caching of only the frequently accessed vertex values in, e.g., a scratchpad memory [112], while not caching the streaming-oriented set of edges can unlock significant improvements for both the case of static as well as evolving graphs.

6.2.2 Complex algorithms

While we explore a significant subset of graph algorithms with both MOSAIC and CYTOM, there exists a large body of work of more complex and computationally involved algorithms, for example in the domain of machine learning for both static [113] as well as evolving

graphs [114]. While not currently supported in either MOSAIC or CYTOM, scaling these kind of complex algorithms to a scale demonstrated in this dissertation uncovers a number of design and systems challenges which need to be addressed. In particular, these algorithms usually carry a significant amount of data per edge (e.g., a few hundred bytes per edge), compared to a single integer (BFS, CC, ...) or float (PR, SSSP, ...) value per edge as shown with MOSAIC and CYTOM.

This poses an obvious design challenge of storing these much larger graphs efficiently and potentially opens the opportunity of using heavy-weight compression effectively to reduce the data set size and allow it to be handled in an efficient manner. In addition, these complex algorithms usually require significantly more complicated processing compared to the relatively simple processing done for the algorithms presented for both MOSAIC and CYTOM and due to this usually rely on accelerators (e.g., GPGPUs) to allow for reasonable run times.

This opens an opportunity for new design- and systems-level abstractions to allow expressing these kind of complex and heavy-weight algorithms in a unified framework while shielding the algorithm developer from the current low-level, accelerator-specific implementation of the algorithm and its associated data set. We explore this kind of direction, albeit for less complicated algorithms, with MOSAIC and envision a system encompassing all mentioned algorithmic and data set-specific features into a unified system and algorithmic abstraction.

6.2.3 Approximation of graph algorithms

Finally, we propose exploring a topic that was briefly introduced for CYTOM already (however, with a very simple algorithm and strategy)—approximation of algorithmic results in the setting of evolving graphs. Approximating algorithmic results has the obvious benefit of allowing for better scalability at the expense of slightly lowered accuracy for the algorithmic result. This trade off is even more important in the setting of evolving graphs,

where timeliness of the algorithmic results is more important than a perfectly accurate algorithmic result, if an approximated version can give certain error bound guarantees. In reality, this trade off might be worth exploring as the performance differences can be drastic, For example, when mining a graph for motifs, the state-of-the-art approach for accurately finding all patterns, Arabesque [115] can take hours to find all 3-motifs even on a "small" graph such as the twitter graph [76] with 1.5 billion edges. In contrast, an approximated version of the same graph and pattern can be found in a couple of minutes within a 5% error bound [98, 116, 99]. However, all these systems are built with static graphs in mind and are not directly applicable to the case of evolving graphs.

Using these results as motivation, a future avenue for processing evolving graphs can evolve around the question how to efficiently approximate complicated algorithms given the unique challenges of frequent updates and the requirement for low-latency algorithmic updates. We present a first step in this direction with CYTOM and the approximation of the pagerank algorithm which unlocks large algorithm speedups at modest costs to the quality of the algorithmic result. This motivates further research into the question how to approximate more complicated algorithms while guaranteeing error bounds for the resulting approximation.

REFERENCES

[1] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion 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)*. San Jose, CA, Apr. 2012, pp. 15–28.

[2] Hadoop. *Apache Hadoop*. `http://hadoop.apache.org/`. 2018.

[3] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. "One Trillion Edges: Graph Processing at Facebook-scale". In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1804–1815.

[4] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. "GraM: Scaling Graph Computation to the Trillions". In: *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*. Kohala Coast, Hawaii, Aug. 2015.

[5] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy 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, pp. 410–424.

[6] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. "Mosaic: Processing a Trillion-Edge Graph on a Single Machine". In: *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*. Belgrade, SR, Apr. 2017, pp. 527–543.

[7] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. "Managing Large Graphs on Multi-cores with Graph Awareness". In: *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA, June 2012, pp. 41–52.

[8] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. "Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing". In: *Proceedings of the 8th European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 169–182.

[9] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. "A Lightweight Infrastructure for Graph Analytics". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Farmington, PA, Nov. 2013, pp. 456–471.

[10] Julian Shun and Guy 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)*. Shenzhen, China, Feb. 2013, pp. 135–146.

[11] Kaiyuan Zhang, Rong Chen, and Haibo Chen. “NUMA-aware Graph-structured Analytics”. In: *Proceedings of the 20th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. San Francisco, CA, Feb. 2015, pp. 183–193.

[12] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R. Dulloor, Michael J. Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. “GraphMat: High Performance Graph Analytics Made Productive”. In: *Proceedings of the VLDB Endowment* 8.11 (July 2015), pp. 1214–1225.

[13] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. “Exploiting NVM in Large-scale Graph Analytics”. In: *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*. New York, NY, USA: ACM, 2015, 2:1–2:9.

[14] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. “GraphChi: Large-scale Graph Computation on Just a PC”. In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, Oct. 2012, pp. 31–46.

[15] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. “X-Stream: Edge-centric Graph Processing Using Streaming Partitions”. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, Nov. 2013, pp. 472–488.

[16] Kai Wang, Guoqing Xu, Zhendong Su, and Yu David 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)*. Santa Clara, CA, July 2015, pp. 387–401.

[17] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo 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. Chicago, Illinois, USA, 2013, pp. 77–85.

[18] Xiaowei Zhu, Wentao Han, and Wenguang 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)*. Santa Clara, CA, July 2015, pp. 375–386.

[19] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. “FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs”. In: *13th USENIX Conference on File and Storage Technologies (FAST) (FAST 15)*. Santa Clara, CA, Feb. 2015, pp. 45–58.

[20] Pradeep Kumar and H. Howie 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)*. Salt Lake City, UT, Nov. 2016, 71:1–71:12.

[21] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. “GraphLab: A New Framework For Parallel Machine Learning”. In: *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI 2010)*. Catalina Island, CA, July 2010, pp. 340–349.

[22] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. In: *Proceedings of the VLDB Endowment* 5.8 (Aug. 2012), pp. 716–727.

[23] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. “PowerGraph: Distributed Graph-parallel Computation on Natural Graphs”. In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, CA, Oct. 2012, pp. 17–30.

[24] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. “GraphX: Graph Processing in a Distributed Dataflow Framework”. In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, Oct. 2014, pp. 599–613.

[25] Jasmina Malicevic, Amitabha Roy, and Willy 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. New York, NY, USA: ACM, 2014, 5:1–5:6.

[26] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. “PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs”. In: *Proceedings of the 10th European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015, 1:1–1:15.

[27] Chenning Xie, Rong Chen, Haibing Guan, Binyu Zang, and Haibo 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)*. San Francisco, CA, Feb. 2015, pp. 194–204.

[28] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD/PODS Conference*. Indianapolis, IN, June 2010, pp. 135–146.

[29] Jeffrey Dean and Sanjay 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, pp. 137–150.

[30] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. “Naiad: A Timely Dataflow System”. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, Nov. 2013, pp. 439–455.

[31] U. Kang, Charalampos E. Tsourakakis, and Christos 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. IEEE Computer Society, 2009, pp. 229–238.

[32] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. “Challenges in Parallel Graph Processing”. In: *Parallel Processing Letters* 17.01 (2007), pp. 5–20.

[33] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. “Gemini: A Computation-Centric Distributed Graph Processing System”. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, Nov. 2016, pp. 301–316.

[34] J. Dongarra, P. Koev, X. Li, J. Demmel, and H. van der Vorst. “10. Common Issues”. In: *Templates for the Solution of Algebraic Eigenvalue Problems*. SIAM, 2000, pp. 315–336.

[35] 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)*. May 2015, pp. 17–27.

[36] Dipanjan Sengupta, Shuaiwen Leon Song, Kapil Agarwal, and Karsten 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)*. Austin, TX, Nov. 2015, 28:1–28:12.

[37] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan. “Scalable SIMD-Efficient Graph Processing on GPUs”. In: *Proceedings of 2015 International Conference on Parallel Architecture and Compilation (PACT)*. Oct. 2015, pp. 39–50.

[38] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi 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. New York, NY, USA: ACM, 2014, pp. 239–252.

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

[40] Sungpack Hong, Tayo Oguntebi, and Kunle 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)*. 2011, pp. 78–88.

[41] 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. May 2015, pp. 819–828.

[42] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung 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)*. Austin, TX, Nov. 2015, 69:1–69:12.

[43] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook 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, June 2016.

[44] Jasmina Malicevic, Baptiste Lepers, and Willy Zwaenepoel. “Everything you always wanted to know about multicore graph processing but were afraid to ask”. In: *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. Santa Clara, CA, July 2017, pp. 631–643.

[45] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. “LLAMA: Efficient graph analytics using Large Multiversioned Arrays”. In: *2015 IEEE 31st International Conference on Data Engineering*. Apr. 2015, pp. 363–374.

[46] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. “STINGER: High Performance Data Structure for Streaming Graphs”. In: *2012 IEEE Conference on High Performance Extreme Computing*. Sept. 2012, pp. 1–5.

[47] Dipanjan Sengupta, Narayanan Sundaram, Xia Zhu, Theodore L. Willke, Jeffrey Young, Matthew Wolf, and Karsten Schwan. “GraphIn: An Online High Performance

Incremental Graph Processing Framework". In: *Proceedings of the 22nd International European Conference on Parallel and Distributed Computing (EuroPar)*. Grenoble, France, Aug. 2016, pp. 319–333.

[48] Pradeep Kumar and H. Howie Huang. "GraphOne: A Data Store for Real-time Analytics on Evolving Graphs". In: *17th USENIX Conference on File and Storage Technologies (FAST)*. Boston, MA, Feb. 2019, pp. 249–263.

[49] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. "Chronos: A Graph Engine for Temporal Graph Analysis". In: *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. Amsterdam, The Netherlands, Apr. 2014, 1:1–1:14.

[50] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. "Kineograph: Taking the Pulse of a Fast-changing and Connected World". In: *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*. Bern, Switzerland, Apr. 2012, pp. 85–98.

[51] Dipanjan Sengupta and Shuaiwen Leon Song. "EvoGraph: On-the-Fly Efficient Mining of Evolving Graphs on GPU". In: *High Performance Computing*. Ed. by Julian M. Kunkel, Rio Yokota, Pavan Balaji, and David Keyes. Springer International Publishing, 2017, pp. 97–119.

[52] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *Proceedings of the 2013 ACM SIGMOD/PODS Conference*. New York, NY, June 2013, pp. 505–516.

[53] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. "Tornado: A System For Real-Time Iterative Analysis Over Evolving Data". In: *Proceedings of the 2016 ACM SIGMOD/PODS Conference*. San Francisco, CA, June 2016, pp. 417–430.

[54] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. "Time-evolving Graph Processing at Scale". In: *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*. GRADES '16. Redwood Shores, California, 2016, 5:1–5:6.

[55] Keval Vora, Rajiv Gupta, and Guoqing Xu. "KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations". In: *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China, Apr. 2017, pp. 237–251.

[56] Mugilan Mariappan and Keval Vora. “GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs”. In: *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*. Dresden, DE, Mar. 2019.

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

[58] *NVM Express*. http://www.nvmexpress.org/. 2017.

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

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

[61] Apache. *Apache Giraph*. http://giraph.apache.org/. 2011.

[62] Tao Zhang, Jingjie Zhang, Wei Shu, Min-You Wu, and Xiaoyao Liang. “Efficient graph computation on hybrid CPU and GPU systems”. In: *The Journal of Supercomputing* 71.4 (2015), pp. 1563–1586.

[63] Frank McSherry and Malte Schwarzkopf. *The impact of fast networks on graph analytics*. https://github.com/frankmcsherry/blog/blob/master/posts/2015-07-31.md. July 2015.

[64] Michalis Faloutsos, Petros Faloutsos, and Christos 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. Cambridge, Massachusetts, USA: ACM, 1999, pp. 251–262.

[65] Roger Pearce, Maya Gokhale, and Nancy 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) (SC’10)*. New Orleans, LA, Nov. 2010, pp. 1–11.

[66] Frank McSherry, Michael Isard, and Derek G. Murray. “Scalability! But at what COST?” In: *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland, May 2015.

[67] 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. Seattle, Washington, USA: ACM, 1974, pp. 47–63.

[68] *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.

[69] David Hilbert. "Über die stetige Abbildung einer Linie auf ein Flächenstück". In: *Mathematische Annalen* 38.3 (1891), pp. 459 –460.

[70] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA, Nov. 2013, pp. 1–17.

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

[72] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides 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. Las Vegas, Nevada, USA, 2008, pp. 16–24.

[73] Richard Bellman. *On a Routing Problem*. Tech. rep. DTIC Document, 1956.

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

[75] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. "R-MAT: A Recursive Model for Graph Mining". In: *Proceedings of the 2004 SIAM International Conference on Data Mining*. Lake Buena Vista, FL, Apr. 2004, pp. 442–446.

[76] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a Social Network or a News Media?" In: *Proceedings of the 19th International World Wide Web Conference (WWW)*. Raleigh, NC, Apr. 2010, pp. 591–600.

[77] Paolo Boldi and Sebastiano Vigna. "The WebGraph Framework I: Compression Techniques". In: *Proceedings of the 13th International World Wide Web Conference (WWW)*. New York, NY, Apr. 2004, pp. 595–601.

[78] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. "Layered Label Propagation: A MultiResolution Coordinate-Free Ordering for Compressing Social Networks". In: *Proceedings of the 20th International World Wide Web Conference (WWW)*. Hyderabad, India, Apr. 2011, pp. 587–596.

[79] Robert Meusel, Oliver Lehmberg, Christian Bizer, and Sebastiano Vigna. *Web Data Commons Hyperlink Graphs*. http://webdatacommons.org/hyperlinkgraph. 2014.

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

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

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

[83] Leman Akoglu, Hanghang Tong, and Danai Koutra. "Graph Based Anomaly Detection and Description: A Survey". In: *Data Mining and Knowledge Discovery* 29.3 (May 2015), pp. 626–688.

[84] Daniel M. Romero, Brendan Meeder, and Jon Kleinberg. "Differences in the Mechanics of Information Diffusion Across Topics: Idioms, Political Hashtags, and Complex Contagion on Twitter". In: *Proceedings of the 20th International World Wide Web Conference (WWW)*. Hyderabad, India, Apr. 2011, pp. 695–704.

[85] *New Tweets per second record, and how!* https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html. 2013.

[86] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. "LinkBench: A Database Benchmark Based on the Facebook Social Graph". In: *Proceedings of the 2013 ACM SIGMOD/PODS Conference*. New York, NY, June 2013, pp. 1185–1196.

[87] *Internet live stats: Emails sent per second*. http://internetlivestats.com/one-second/#email-band. 2019.

[88] Keval Vora, Sai Charan Koduru, and Rajiv Gupta. "ASPIRE: Exploiting Asynchronous Parallelism in Iterative Algorithms Using a Relaxed Consistency Based DSM". In: *Proceedings of the 25th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Portland, OR, Oct. 2014, pp. 861–878.

[89] Keval Vora, Rajiv Gupta, and Guoqing Xu. "Synergistic Analysis of Evolving Graphs". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.4 (Oct. 2016), 32:1–32:27.

[90] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. "Statistical Properties of Community Structure in Large Social and Information Networks". In: *Proceedings of the 17th International World Wide Web Conference (WWW)*. Beijing, China, Apr. 2008, pp. 695–704.

[91] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. "Graph Structure in the Web". In: *Proceedings of the 9th International World Wide Web Conference (WWW)*. Amsterdam, Netherlands, May 2000, pp. 309–320.

[92] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. "Existential Consistency: Measuring and Understanding Consistency at Facebook". In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 295–310.

[93] Leslie G. Valiant. "A Bridging Model for Parallel Computation". In: *Commun. ACM* 33.8 (Aug. 1990), pp. 103–111.

[94] Leslie G. Valiant. "A Bridging Model for Multi-core Computing". In: *J. Comput. Syst. Sci.* 77.1 (Jan. 2011), pp. 154–166.

[95] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Effective Straggler Mitigation: Attack of the Clones". In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, IL, Apr. 2013, pp. 185–198.

[96] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. "Rock You Like a Hurricane: Taming Skew in Large Scale Analytics". In: *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*. Porto, PT, Apr. 2018, 20:1–20:15.

[97] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. "Understanding Manycore Scalability of File Systems". In: *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*. Denver, CO, June 2016, pp. 71–85.

[98] Anand Padmanabha Iyer, Aurojit Panda, Shivaram Venkataraman, Mosharaf Chowdhury, Aditya Akella, Scott Shenker, and Ion Stoica. "Bridging the GAP: Towards Approximate Graph Analytics". In: *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA '18. Houston, TX, 2018, 10:1–10:5.

[99] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. "ASAP: Fast, Approximate Graph Pattern Mining at Scale". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, Oct. 2018, pp. 745–761.

[100] Anand Padmanabha Iyer, Qifan Pu, Kishan Patel, Joseph E. Gonzalez, and Ion Stoica. *Tegra: Efficient Ad-Hoc Analytics on Time-Evolving Graphs*. 2019.

[101] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. “Group Formation in Large Social Networks: Membership, Growth, and Evolution”. In: *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Philadelphia, PA, Aug. 2006, pp. 44–54.

[102] Jaweon Yang and Jure Leskovec. “Defining and Evaluating Network Communities Based on Ground-Truth”. In: *2012 IEEE 12th International Conference on Data Mining*. ISDM’12. Dec. 2012, pp. 745–754.

[103] M. G. Kendall. “A New Measure of Rank Correlation”. In: *Biometrika* 30.1/2 (1938), pp. 81–93.

[104] Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. “Large-Scale Parallel Collaborative Filtering for the Netflix Prize”. In: *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management*. AAIM ’08. Shanghai, China, 2008, pp. 337–348.

[105] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. “Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+”. In: *Proceedings of the 2015 Data Compression Conference*. DCC ’15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 403–412.

[106] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy. “An Initial Characterization of the Emu Chick”. In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops*. IPDPSW’18. Vancouver, BC, May 2018, pp. 579–588.

[107] Gushu Li, Guohao Dai, Shuangchen Li, Yu Wang, and Yuan Xie. “GraphIA: An In-situ Accelerator for Large-scale Graph Processing”. In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS ’18. Alexandria, Virginia, 2018, pp. 79–84.

[108] C. Xu, C. Wang, L. Gong, L. Jin, X. Li, and X. Zhou. “Domino: Graph Processing Services on Energy-Efficient Hardware Accelerator”. In: *2018 IEEE International Conference on Web Services*. ICWS’18. San Francisco, CA, July 2018, pp. 274–281.

[109] S. Jun, A. Wright, S. Zhang, S. Xu, and Arvind. “GraFBoost: Using Accelerated Flash Storage for External Graph Analytics”. In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*. ISCA’18. Los Angeles, CA, June 2018, pp. 411–424.

[110] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. “Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO’18. Fukuoka, Japan, Oct. 2018, pp. 1–14.

[111] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO’16. Taipei, Taiwan, Oct. 2016, 56:1–56:13.

[112] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. “Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems”. In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*. CODES ’02. Estes Park, Colorado, 2002, pp. 73–78.

[113] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. “Learning Combinatorial Optimization Algorithms over Graphs”. In: *Advances in Neural Information Processing Systems 30 (NIPS)*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 6348–6358.

[114] Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. “Know-Evolve: Deep Temporal Reasoning for Dynamic Knowledge Graphs”. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, Aug. 2017, pp. 3462–3471.

[115] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. “Arabesque: A System for Distributed Graph Mining”. In: *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, Oct. 2015, pp. 425–440.

[116] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. “Towards Fast and Scalable Graph Pattern Mining”. In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA, 2018.

All URLs were last accessed on May 1, 2019.