The Journal of Supercomputing
https://doi.org/10.1007/s11227-019-03097-w

GraphMap: scalable iterative graph processing using NoSQL

Sayan Goswami[1] · Ayam Pokhrel[2] · Kisung Lee[2] · Ling Liu[3] · Qi Zhang[4] · Yang Zhou[5]

© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
Despite having several distributed graph processing frameworks, scalable iterative processing of large graphs is a challenging problem since the graph and intermediate data need a global view of the graph topology in distributed memory. Although some systems support out-of-core iterative computations, they use a single machine and often require fast storage. In this paper, we present a new distributed iterative graph computation framework, called GraphMap, that utilizes a disk-based NoSQL database system for scalable graph processing while ensuring competitive performance. Extensive experiments on several real-world graphs show that GraphMap is more scalable and often faster than existing distributed memory-based systems for various graph processing workloads.

Keywords Graph processing · Distributed systems · NoSQL

1 Introduction

In this era of big data, various distributed big data systems, such as Apache Hadoop and Spark, are processing a massive amount of information generated from heterogeneous data sources, including online social networks, smartphones, and Internet of Things (IoT) devices such as smart light bulbs and thermostats, at an unprecedented rate. Among various kinds of data, graph data are getting a lot of attention because graphs are everywhere (e.g., online social networks, brain networks, transportation

✉ Sayan Goswami
sgoswami@lsus.edu

[1] Louisiana State University, Shreveport, USA

[2] Louisiana State University, Baton Rouge, USA

[3] Georgia Institute of Technology, Atlanta, USA

[4] IBM Thomas J. Watson Research Center, New York, USA

[5] Auburn University, Auburn, USA

Published online: 04 December 2019

networks) and, more importantly, people can get deeper insights into big data based on the explicit and implicit relationships among real-world entities. For example, in bioinformatics, scientists are building a De Bruijn graph or an overlap graph to construct a whole genome sequence based on short reads generated from a next-generation sequencing machine [9, 32].

Even though graph data are invaluable in most disciplines and applications, graph data processing has several technical challenges that need to be addressed for efficient large-scale graph-based analytics. First, the sizes of real-world graphs are already huge and, more importantly, increasing at a tremendous rate. For example, if we represent each social network user as a vertex, there are more than 2 billion vertices in the friendship graph of Facebook [6]. Therefore, large-scale graph processing requires massive computing and storage resources. To make matters worse, most graph-based applications generate a huge amount of intermediate data, and the size of the intermediate data is usually much larger (in some cases, several orders of magnitude bigger) than the original input graph size. Secondly, graph data have complicated relationships among data entities, and these relationships are essential for graph analytics to gain a deeper insight into big graph data. However, these complex relationships make it hard to partition the graph data for distributed graph data processing. Last but not the least, most real-world graph data have an extremely skewed distribution in terms of the number of connected edges. In other words, most real-world graphs have some vertices that have a huge number of connected edges. These high-degree vertices make it hard to ensure load balancing during large-scale graph data processing.

To address the challenges for efficient large-scale graph data processing, system researchers have devoted much effort to the study of big graph systems in recent years. Existing graph systems for iterative computations can be categorized into two types based on their system architecture: 1) centralized disk-based systems and 2) distributed memory-based systems. The disk-based systems on a single machine (e.g., GraphChi [15], X-Stream [25], PathGraph [31], TurboGraph [10], FlashGraph [34], GraphTwist [35], Mosaic [20]) focus on maximizing parallelism among computing cores and designing graph representations optimized for HDD or SSD accesses. Even though they demonstrate significant performance improvements for iterative graph computations on a single machine, they have limited scalability because they are incapable of processing a graph whose computing and storage requirements are bigger than the available resources on the single machine.

As for scalable solutions, several distributed memory-based systems on a cluster of commodity servers (e.g., Pregel [21], Giraph [1], Hama [2], GraphLab [19], PowerGraph [7], Giraph++ [27], GraphX [8], Pregelix [5]) have been developed. Even though they are designed to handle larger graphs by adding more compute nodes into the cluster, they heavily rely on distributed memory to store not only the entire input graph but also all intermediate data and communication messages. In graph-based applications, it is not uncommon that the size of the intermediate data is several orders of magnitude bigger than the original input graph size. Furthermore, since the input graph is partitioned and distributed among compute nodes, existing systems can fail when the least powerful compute node on the cluster cannot accommodate its graph partition, all intermediate results, and communication messages in

its main memory. Even though a few distributed systems, such as Giraph and Pregelix, support out-of-core computations to utilize external memory for processing large graphs, these systems typically focus only on decreasing the memory footprint, not on effectively utilizing the external memory for improving the performance of iterative computations, or require native storage modules lacking in fault tolerance.

In this paper, we claim that well-designed out-of-core graph systems for iterative computations can handle large-scale graphs while ensuring competitive performance by effectively partitioning and accessing graph data based on data locality. Such systems will enable us to run iterative graph computations on large-scale graphs using a small and affordable cluster (e.g., tens of nodes), instead of a huge and expensive cluster (e.g., hundreds or thousands of nodes) that is required by most existing graph systems to accommodate not only input graph data but also all intermediate results in its distributed memory. To validate this claim, we present a new distributed iterative graph computation framework, called GraphMap, that effectively utilizes a disk-based NoSQL database system for scalable graph processing while ensuring competitive performance.

GraphMap has four salient features for scalable and efficient iterative graph processing. First, it separates read-only graph data from modifiable data to maximize sequential accesses and minimize random disk accesses during iterative graph computations. By holding modifiable data in memory and immutable data in a disk-based NoSQL system, GraphMap can scale to large-scale graphs while demonstrating competitive performance through optimized disk I/O. Second, GraphMap is equipped with two-level graph data partitioning (inter-worker and intra-worker partitioning) for locality-optimized data placement and balanced workloads. In the inter-worker partitioning (level 1), vertices and their connected edges are partitioned and distributed among compute nodes for balanced graph processing. In the intra-worker partitioning (level 2), each level-1 partition is further split into smaller chunks based on ranges to efficiently support not only sequential accesses but also random accesses. Third, in the inter-worker partitioning, GraphMap supports various graph partitioning techniques including hash- and mincut-based partitioning so users can choose one based on their requirements and workloads. Lastly, GraphMap implements a collection of locality-aware optimization techniques to further improve the overall performance of iterative graph processing, including dynamic access patterns based on the number of active vertices, locality-based disk block accesses, partition-aware identifier assignments and message batching, and worker-partition colocation. Through the proposed techniques, GraphMap can utilize the secondary storage by reducing random disk I/O and demonstrate competitive performance for various iterative algorithms. We compare the experimental results of GraphMap generated using several real-world graphs for various iterative algorithms with those of state-of-the-art distributed graph frameworks. The evaluation results demonstrate not only the improved scalability of GraphMap but also competitive performance compared to the existing in-memory systems.

The rest of the paper is organized as follows. We first summarize the related works of this paper in Sect. 2. In Sect. 3, we provide a detailed overview of GraphMap's design and architecture. We describe the data placement scheme used in GraphMap in Sect. 4 and the locality-based dynamic optimization scheme in Sect. 5.

In Sect. 6, we present a strategy to move computation to data to reduce the network traffic. Lastly, we evaluate GraphMap in Sect. 7 and conclude the paper in Sect. 8.

2 Related works

Iterative graph algorithms have been studied extensively, and a number of graph processing frameworks have been developed specifically for them. Most of these frameworks can be broadly categorized into two groups. The first group is characterized by in-memory distributed programs built for commodity clusters. These frameworks [7, 8, 19, 21, 26] typically have to load entire graphs in memory so they require huge memory for large-scale graphs. Apache Hama and Giraph are two of the most popular examples built on the Pregel-like BSP paradigm with the "think like a vertex" programming model. On the contrary, systems like GraphX [8] and Pregelix [5] are implemented using a general-purpose distributed in-memory dataflow network where the graphs are stored as tables so the algorithms take advantage of database-style queries. Trinity [26] is another framework that utilizes a distributed in-memory key-value store for storing graphs and intermediate data. GraphLab [19] and PowerGraph [7] represent yet another type of distributed frameworks based on the asynchronous communications. PowerGraph can also be used synchronously. Although some frameworks such as Pregelix have out-of-core execution capabilities built in them, they are not optimized for slow storage media, and their execution times can be prohibitively high when using external storage while running large datasets.

The second group consists of disk-based standalone frameworks such as GraphChi [15], X-Stream [25], and a few others [10, 31, 34], which focus on optimizing performance of algorithms when the graphs are too large to fit in main memory. However, these frameworks are not designed to run on clusters. GraphChi, which is built on the "think like a vertex" model, divides the graph among several shards and accesses them in parallel using a sliding-window model. On the other hand, X-Stream uses an edge-centric model where the edges are partitioned and then streamed in memory. Contrary to the ones mentioned before, PathGraph [31] uses a path-centric approach that lets them utilize the access locality in both disk and memory. Yet another group of frameworks including FlashGraph [34] and TurboGraph [10] are designed to exploit the parallel I/O capabilities of SSDs. In addition, there are several graph frameworks optimized for GPUs such as graph analytics on multiple GPUs [23], Lux [12], and DiGraph [33].

Chaos [24] is another graph processing framework that uses secondary storage over a distributed cluster. However, unlike GraphMap that is designed for commodity clusters, Chaos does not handle fault tolerance and is not tailored to recover from storage failures. Another aspect in which Chaos differs from GraphMap is that it assumes the underlying network interconnects have a high bandwidth and does not depend on locality of access. This is in contrast to GraphMap because we try to extract as much locality as possible in order to reduce the dependency on the network. Specifically, Chaos reports their results on a 40 GigE network while we evaluate GraphMap on a 1 GigE network. In addition, for running Chaos on a distributed

system, the end user is required to split the input into roughly equal parts and store them on the different machines. GraphMap automatically handles this by effectively utilizing a distributed file system and a NoSQL database system. GraphD [30] is a newest out-of-core graph processing system. Unlike GraphMap, GraphD is not built atop general-purpose tools such as HDFS and HBase, and hence it lacks several important features for distributed computing such as fault tolerance.

A preliminary version of this paper appeared in [17]. In this extended version, we have new contributions as follows. First, we newly propose and develop a worker-partition colocation technique and experimentally demonstrate its benefits. Second, we extend GraphMap to support other types of graph partitioning techniques in addition to the hash-based partitioning, and so users can choose one based on the characteristics of graphs and workloads. We implement minimum cut-based partitioning in particular and experimentally compare its performance with that of hash-based partitioning. Moreover, we report our experimental results using a new type of compute nodes to show the effects of different machine specifications.

3 GraphMap overview

In this section, we first introduce the preliminaries of our design features and then present an overview of the proposed framework, including the partitioning techniques, programming model, and system architecture.

3.1 Preliminaries

In GraphMap, information networks are modeled as directed graphs, and undirected edges are converted into directed edges having opposite directions.

Definition 1 (*Graph*) A graph G consists of a set of vertices (V_G) and a set of directed edges (E_G) where $E_G \subseteq V_G \times V_G$. For an edge $e = (u, v) \in E_G$, u is called the *source vertex*, and v is called the *destination vertex*. u and v have e as an *out-edge* and *in-edge*, respectively. $|V_G|$ and $|E_G|$ mean the number of vertices and edges, respectively.

Each vertex has a unique vertex identifier and a set of attributes that characterize the properties of the vertex. In this paper, we interchangeably use the terms "attribute" and "state" of a vertex. If the edges have modifiable, user-defined values, we model them as attributes of source vertices. This permits us to treat all vertices as mutable data and edges as immutable data while processing the graphs.

The separation of mutable from immutable data lets GraphMap exploit a storage scheme where the mutable vertex data are compactly placed in memory and the immutable edge data are stored in a locality-aware fashion on disk. As most graphs generally have much more edges than vertices (e.g., up to 100 times more edges in datasets in Table 1), this scheme lets us significantly reduce the memory required to load and process large graphs. In subsequent sections, we show that by using this

clear separation between mutable and read-only components in a graph, we can substantially reduce random disk I/O for several iterative graph algorithms.

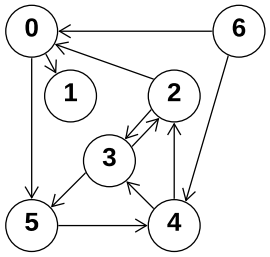
To optimize the access of edges in a graph, we categorize them into three classes based on their relative direction to a vertex as follows.

Definition 2 (*Out-edges, In-edges, and Bi-edges*) In graph G, the **out-edges** of vertex $v \in V_G$ are defined as $E_v^{out} = \{(v, v') | (v, v') \in E_G\}$. Conversely, the **in-edges** of v are defined as $E_v^{in} = \{(v', v) | (v', v) \in E_G\}$. The **bi-edges** of v are defined as $E_v^{bi} = E_v^{out} \cup E_v^{in}$ (i.e., the union of out-edges and in-edges).

For each vertex in the graph, we build a vertex block (VB) consisting of an *anchor* vertex and the edges directly connected to it with their properties. Since different iterative graph algorithms can have different computation characteristics, GraphMap supports three kinds of VBs based on the edge direction from the anchor vertex: out-edge vertex block (out-VB), in-edge vertex block (in-VB), and bi-edge vertex block (bi-VB). An out-VB comprises of a source vertex and the adjacency list of destination vertex IDs to which it has an out-edge. Similarly, an in-VB consists of a destination vertex and the adjacency list of source vertex IDs from which it has an in-edge. We formally define the concept of VBs as follows.

Definition 3 (*Vertex block*) In graph G, the **out-edge vertex block** of $v \in V_G$ is a 2-tuple that consists of v as its anchor vertex and the set of its out-edges, defined as $VB_v^{out} = (V_v^{out}, E_v^{out})$ such that $V_v^{out} = \{v\} \cup \{v^{out} | (v, v^{out}) \in E_v^{out}\}$. Similarly, the **in-edge vertex block** of v is denoted by $VB_v^{in} = (V_v^{in}, E_v^{in})$ such that $V_v^{in} = \{v\} \cup \{v^{in} | (v^{in}, v) \in E_v^{in}\}$. We define the **bi-edge vertex block** of v as $VB_v^{bi} = (V_v^{bi}, E_v^{bi})$ such that $V_v^{bi} = V_v^{in} \cup V_v^{out}$.

Figure 1 depicts the ideas described above using an unweighted directed graph in Fig. 1a. The numbers inside the vertices are the vertex identifiers (IDs). Figure 1b



(a) A sample graph

Vertex	State	In-edge VB	Out-edge VB
0	0	0, {2, 6}	0, {1, 5}
1	1	1, {0}	1, { }
2	3	2, {3, 4}	2, {0, 3}
3	3	3, {2, 4}	3, {2, 5}
4	2	4, {5, 6}	4, {2, 3}
5	1	5, {0, 3}	5, {4}
6	INF	6, { }	6, {0, 4}

(b) States and vertex blocks

Fig. 1 A sample graph, its set of vertex blocks and the states of the vertices at the convergence of the single-source shortest path algorithm from vertex ID 0

shows how the graph is conceptually represented in GraphMap. Each vertex has a state that may change after each iteration. In this case, the state represents the distance from the source vertex with ID 0 at the end of the last iteration. The corresponding in-edge VB and out-edge VB for each vertex are also shown in Fig. 1b. These vertex blocks store the topology information of the graph in an adjacency-list format. Note that if an algorithm does not use the incoming edge information for sending the updated state of vertices (as in the case of the single-source shortest path), it may be sufficient to store only the out-VB of a vertex.

3.2 Two-level graph partitioning

GraphMap uses two-level graph data partitioning (inter-worker and intra-worker partitioning) for locality-optimized data placement and balanced workloads. In the inter-worker partitioning (level 1), the graph is partitioned using edge-cuts, and the partitions are distributed among the compute nodes for balanced processing. A partition is composed of vertices and their corresponding vertex blocks. While vertices are stored in memory, their vertex blocks are stored in a distributed file system. In the intra-worker partitioning (level 2), each level-1 partition is further split into smaller chunks based on ranges to efficiently support not only sequential accesses but also random accesses on the level-1 partition. This is done by sorting the vertex blocks in each level-1 partition by their anchor vertex IDs and partitioning them into smaller chunks based on ranges so that each chunk is indexed by its smallest and largest vertex IDs. We perform the range-based intra-worker partitioning at all workers in parallel.

Figure 2 depicts the partitioning scheme employed by GraphMap running on a 2-node cluster when applied to the sample graph in Fig. 1a. In the level-1 partitioning where the vertices are distributed among the compute nodes, vertices with IDs 0, 2, 4, and 6 are assigned to node 0 and the rest are assigned to node 1. The vertices and their attributes are stored in memory as a 2-tuple. Next, in the level-2 partitioning where the VBs are sorted by their IDs and split into ranges, VBs corresponding to vertices 0 and 2 are assigned to level-2 partition 0.0, and those corresponding to vertices 4 and 6 are assigned to 0.1.

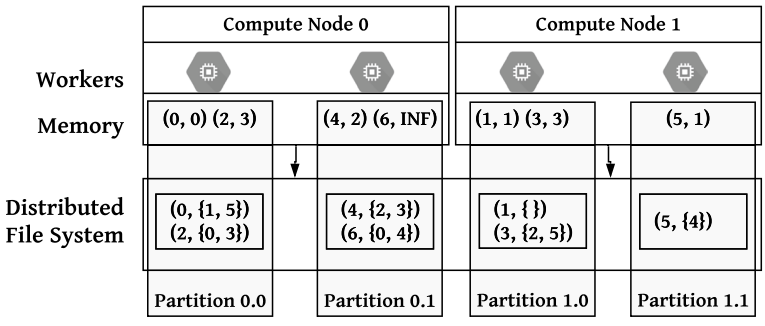


Fig. 2 GraphMap's 2-level partitioning scheme

In the inter-worker partitioning (level 1), GraphMap supports various graph partitioning techniques including hash- and mincut (minimum cut)-based partitioning so users can choose one based on their requirements and workloads. By default, GraphMap performs hash-based partitioning using the hash value of vertex IDs and assign their vertex block (VB) to one of the worker machines corresponding to the hash value. Hash-based partitioning is fast and lightweight because we do not need to maintain any additional data structure for storing the partition ID of each vertex. In addition to the hash-based partitioning, GraphMap also supports mincut-based partitioning to assign close vertices (and their vertex blocks) into the same partition. By using this locality-aware graph partitioning scheme, we can reduce the amount of inter-partition communication because it is likely that two connected vertices are located in the same worker machine. However, unlike the hash-based partitioning, this scheme requires a pre-processing step for minimum cut, and we also need to store the mapping information between vertices and partitions. GraphMap is designed to be equipped with other partitioning techniques such as SHAPE [16].

3.3 Supporting vertex-centric API

A clear majority of the iterative graph processing frameworks adopt a vertex-centric ("think like a vertex") programming model [7, 19, 21]. The implementation of an iterative graph algorithm (e.g., PageRank, single-source shortest path computations, triangle counting) in the vertex-centric model requires the users to write a function that defines what each vertex performs for *each* iteration of the algorithm. At every iteration, all vertices of the graph run the same function in parallel. Each vertex typically performs three steps during an iteration. (1) It gathers the states of its neighboring vertices, typically along its in-edges. (2) Depending on some user-defined logic, it updates its value based on its current state and that of its neighboring vertices. (3) If its status value is modified, it propagates the updated status value to its neighboring vertices, typically along its out-edges.

Every vertex is in one of two states during the lifetime of the program—*active* or *inactive*. During an iteration, only those vertices that are in an active state can execute the vertex program. The number of active vertices varies between the different classes of algorithms as well as from iteration to iteration within the same algorithm. For instance, all vertices are active in PageRank during all the iterations whereas, in case of connected components (CC), the percentage of active vertices starts from 100% and tends toward 0% as the program advances. In case of single-source shortest path (SSSP), the number of active vertices at a particular iteration may vary even within the same graph depending on the choice of the source vertex. A vertex can deactivate itself, typically at the end of an iteration, but can be reactivated through messages from other vertices. The program terminates when either all vertices become inactive or it satisfies a predefined condition for convergence (e.g., the number of iterations).

Most of the existing distributed vertex-centric graph processing frameworks are based on the Bulk Synchronous Parallel (BSP) [28] model of computation designed for shared-nothing architectures. Applications based on the BSP model typically

start with an initialization step in which the input graph is read and scattered across the cluster nodes. In each subsequent iteration, worker processes execute the user program in parallel and independent of each other. At the end of each iteration, the workers perform a global barrier synchronization during which they communicate with each other to merge their results with those of their peers. Since vertices communicating with each other may reside on different machines, most distributed graph processing frameworks provide some mechanism of interaction between vertices, usually along their edges. For instance, Pregel [21] operates in a pure message passing model in which vertices send messages along their outgoing edges at the end of an iteration. During iteration i, each vertex processes all incoming messages received during iteration $i-1$. On the other hand, in GraphLab/PowerGraph [7, 19], vertices can directly access the data within their neighboring vertices through a shared state.

ALGORITHM 1: SSSP Program in the Vertex-Centric model

```
1  Function Compute(messages):
2      if getSuperstepCount() == 0 then setValue(INFINITY)
3      int minDistance = isStartVertex() ? 0 : INFINITY
4      foreach message ∈ messages do minDistance = min(minDistance, message)
5      if minDistance < getValue() then
6          setValue(minDistance)
7          foreach edge ∈ getEdges() do sendMessage(edge,
             minDistance+edge.getValue())
8      end
9      voteToHalt()
10
11 Function Combine(messages):
12     return min(messages)
```

Algorithm 1 shows pseudo-code for a SSSP program based on the vertex-centric and BSP model. At the very first iteration (also known as *superstep*), each vertex initializes its value with *infinity* (line 2). In the following iterations, each vertex updates its value using the smallest value of all the incoming messages (line 4) and its own previous value (lines 5, 6) and broadcasts the updated value to all its neighboring vertices along its out-edges (line 7). At the end of each iteration, the vertex changes its status to *inactive* (line 9) and will be reactivated again in subsequent iterations if it gets incoming messages from other vertices. To minimize the volume of messages transferred over the network, we often create a combiner that merges the messages bound for a particular destination (lines 11–12).

3.4 GraphMap system architecture

Figure 3 shows the architectural overview of GraphMap. It is built on the BSP computation paradigm using the message passing model.

The system consists of a master node and a set of worker nodes. The master node is responsible for accepting user requests and coordinating with the worker

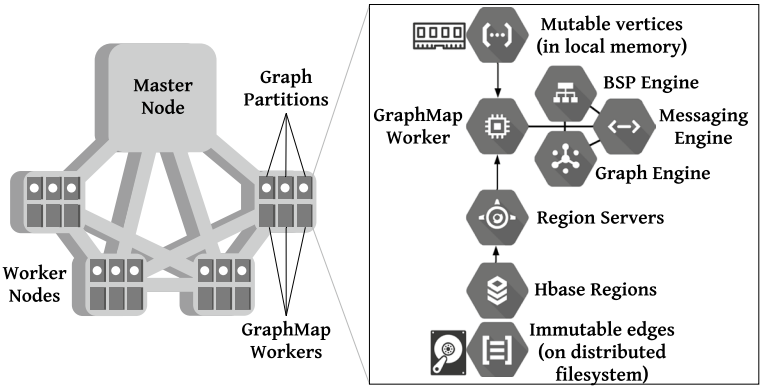


Fig. 3 GraphMap system architecture

machines. The worker nodes synchronize and communicate with each other through messages.

For a task-level parallelism, every worker node has multiple slots for running worker tasks, and each of which is assigned a single partition. The worker tasks keep the mutable vertex data in memory and update them at every iteration by reading the immutable VBs from disk. To store VBs on disk, GraphMap utilizes HBase, a disk-based NoSQL database system (further explained in Sect. 4). Additionally, worker tasks communicate between themselves with the help of a messaging engine that is responsible for coalescing messages sent to the same worker. A global barrier synchronization is performed by the workers at the end of every iteration with the help of a BSP engine. Moreover, GraphMap is equipped with an optimization scheme that dynamically switches between sequential and random accesses at every iteration depending on the computation patterns at each GraphMap worker (further explained in Sect. 5). GraphMap also provides a worker-partition colocation technique that allows workers to process partitions that reside in the same machine for reducing the amount of data transferred through the network (further explained in Sect. 6).

4 Locality-aware data storage

This section introduces the storage scheme that GraphMap uses to exploit the locality in graph datasets. As mentioned before, most of the iterative graph algorithms only modify the vertex state whereas the edges remain unchanged throughout the entire computation. Thus, through a clean separation between the *mutable* and *immutable* parts of the graph, we can keep most or all of the mutable data in memory and access the immutable data from disk thereby minimizing non-sequential I/O. Contrary to the existing distributed BSP-based frameworks where workers store the entire graph as well as the intermediate data in memory, GraphMap judiciously integrates secondary storage in memory-intensive graph algorithms.

Figure 4 depicts GraphMap's storage scheme. The anchor vertices and their data are stored in a vertex data map in memory. The disk contains the corresponding vertex blocks automatically sorted by HBase and split in multiple ranges (two ranges in Fig. 4), which are then indexed in a region-index block. Lastly, incoming and outgoing messages are buffered in in-memory queues so that they can be delivered to their targets at the end of each iteration.

More specifically, GraphMap stores all anchor vertices along with their states in memory whereas their vertex blocks consisting of the edges along with their properties (such as edge weights) are maintained on disk. The advantages of this storage scheme are twofold: (1) By storing only the mutable data in memory and consequently reducing the memory requirement, we can process larger graphs using fewer nodes, and; (2) By storing the vertex blocks belonging to the same partition in contiguous locations on disk, we can access the immutable data using sequential accesses thereby improving I/O performance. Since most graphs have a much higher number of edges than that of vertices, and their degree distribution is often very skewed, representing the edges as immutable data and storing them on disk reduces the memory required to load the graph.

The mutable vertex data are stored in a mapping table in memory and is used to update the values of anchor vertices. For instance, in case of SSSP, the mapping table stores the current minimum distance of each vertex from the source. Likewise, in case of PageRank, it stores the current rank of each vertex. For the immutable edge data that are stored in the vertex blocks of anchor vertices, there are two types of access characteristics that we exploit to reduce the cost of reading them from disk in every iteration:

1. *Edge access locality*—edges of a vertex are accessed together to modify its state.

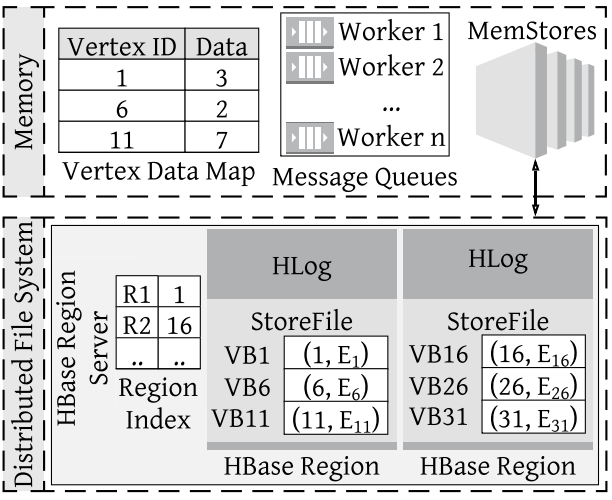


Fig. 4 Storage scheme in GraphMap (single worker)

2. *Vertex access locality*—the same anchor vertices are accessed by a worker in every iteration.

The edge access locality is utilized by packing all edges of a vertex into a single vertex block. On the other hand, the vertex access locality is used by storing the vertex blocks of all anchor vertices belonging to a partition in contiguous locations on disk, allowing the worker to read them sequentially at each iteration. Moreover, to improve the performance of random accesses, the vertex blocks are sorted by their anchor vertex identifiers and indexed by storing them in regions and retaining the addresses of the first vertex block of each region.

5 Locality-based optimizations

Although the storage scheme used in GraphMap is designed to minimize non-sequential I/O on slower storage media commonly found in commodity clusters, we keep in mind that there exists a class of graph algorithms where the data access patterns do not comply with only sequential reads/writes. A consequence of this can be observed in Fig. 5, which shows the number of active vertices in a partition of the Orkut graph [22] at every iteration during the execution of three algorithms—SSSP, CC, and PageRank. In case of algorithms like PageRank where all the vertices are active during every iteration, the sequential access pattern used to read the vertex blocks from disk would be the most efficient option. On the other hand, in algorithms similar to CC and SSSP where the number of active vertices may exhibit large variations between iterations, sequential access at every iteration may not be ideal especially during the ones in which only a few vertices are active and yet all the vertex blocks must be read from disk.

Based on this observation, we propose an optimization scheme that dynamically switches between sequential and random accesses at every iteration depending on the computation patterns at each GraphMap worker. This adaptation not only lets us gradually filter out the non-active vertices at every iteration but also avoids unnecessary disk accesses. Recall that on disk, the vertex blocks are sorted by their anchor vertex identifiers and are indexed into regions for efficient random accesses. During an iteration, if the number of active vertices is less than a system-defined (and

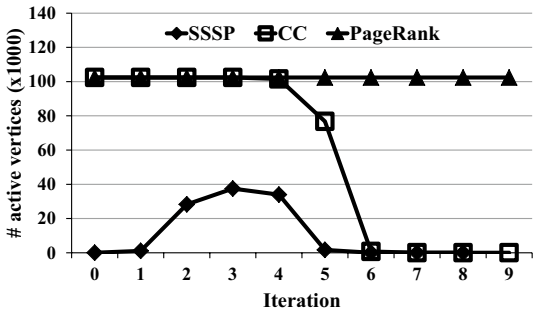


Fig. 5 The number of active vertices at each iteration

user-modifiable) threshold θ, the vertex blocks are read with random accesses using the index block, which is cached in main memory. Since the index block contains the first vertex ID and the address of each region, when querying for a vertex v belonging to some partition p, we consult the index block of p and obtain the region b_i such that the vertex ID of v is greater or equal to that of the first vertex at b_i but less than the first vertex ID at b_{i+1}. Thus, we can perform a scan on the region b_i to find the vertex block of v. When the number of active vertices is greater or equal to θ, we read all vertex blocks sequentially irrespective of whether the corresponding vertex is active or not.

Since disk access latency can vary widely across clusters and even within the same cluster, the threshold θ dynamically adapts itself on each worker node. At every iteration, we compare the latency of sequential and random disk accesses and calculate the number of vertices (i.e., θ) such that the total time required to randomly access θ vertex blocks is equal to the time required to sequentially access all vertex blocks in a partition. Specifically, we determine the value of θ as follows. Let θ_{iw}, s_{iw}, r_{iw}, and a_{iw} denote the threshold, the total time required to sequentially read all VBs, the total time required to randomly access all active VBs, and the total number of vertices that are active on worker w during iteration i, respectively. The threshold before the start of the first iteration θ_{0w} is obtained empirically as described before. We define m and n ($m, n \in \mathbb{Z}_{\geq 0}$) to store the IDs of latest iteration where vertex blocks were read using a sequential access (range scan) and random accesses, respectively. Before the start of each iteration i ($i \in \mathbb{Z}_{>0}$), we update the threshold using the following rule.

$$\theta_{iw} = \begin{cases} \theta_{(i-1)w}, & \text{if } m = 0 \text{ or } n = 0 \\ s_{mw} \frac{a_{nw}}{r_{nw}}, & \text{otherwise.} \end{cases}$$

6 Moving computation to data

In GraphMap, each partition of the graph is assigned to a worker, and the number of workers launched by the master equals the number of partitions. Each partition is composed of anchor vertices, their states, and their vertex blocks. All vertices (and their states) belonging to a partition reside in the memory of the same compute node as the worker assigned to the partition. However, since the corresponding vertex blocks are stored in a distributed file system, the workers are oblivious of the physical locations of the vertex blocks. This is useful when the cluster contains dedicated storage servers underneath the distributed file system, which are different from the compute nodes. However, in situations where the local disks in compute nodes are used to make up the distributed file system (as is commonly done in HDFS), the storage transparency has the unintended consequence that the anchor vertices and their VBs may physically reside on different machines. In such cases, the workers have to fetch the vertex blocks corresponding to the anchor vertices in the partition assigned to it from a remote node at every iteration before processing can commence. On commodity clusters where the nodes may not be connected by

high-speed networks, these data migrations can have a significant impact on the total execution times.

In order to avoid this network overhead in GraphMap, we move the computation toward the data instead of the other way around. Ideally, this would imply that each worker will process a partition that resides on the same machine. However, the difficulty in this approach is presented by the location transparency of the underlying distributed storage system, which abstracts the low-level details from the workers. To get around this impediment, we partition the graph and store the partitions in a distributed NoSQL database in a way that each partition (both vertices and vertex blocks) is completely contained in a single node. We also maintain a globally accessible data structure (distributed in-memory key-value store), which stores the locations of the individual partitions. The NoSQL database is split in such a way that the number of regions equals the number of graph partitions (and the number of GraphMap workers), and the regions are distributed among the nodes in the cluster. When the partitions are loaded into the database, each of them is uniquely mapped to a single region in the database, and the mapping information is stored in the global key-value store.

During the iterative graph processing stage, when the GraphMap master launches its workers, each worker consults the key-value store to find out the list of partitions residing it its node and selects one that has not yet been selected by its peers running on the same node. Next, workers update the store with 2-tuples consisting of their worker IDs and their assigned partition IDs, and this information is used in later stages to facilitate inter-worker message passing. Throughout the entire lifetime of the graph algorithm, a worker continues to work on the same local partition, hence the step of selecting a partition and transmitting the mapping information to the peers is a one-time process. Moreover, this scheme limits the network traffic to relatively smaller mutable vertex data that are maintained in memory as opposed to the much larger immutable edge data that have to be accessed from disk.

It is worth noting that, to ensure fault tolerance and provide a higher throughput, it is likely that the distributed file system will replicate the partitions into multiple nodes. Besides, workers can only interact with the database using its APIs and have no control on the physical location from where a partition is fetched. Still, it is fair to assume that, if a partition is present in the same node as the worker requesting it, the database will try to use the local copy before it decides to fetch a remote one unless working with the remote copy is faster due to issues such as disk contention at the local node.

7 Experimental evaluation

This section presents an experimental analysis of GraphMap using various iterative algorithms on real graph datasets of different sizes. We begin by explaining the characteristics of the graphs used for evaluating GraphMap. Next, we perform a set of experiments that can be classified into six categories: (1) We show the execution times of GraphMap for several iterative graph algorithms and compare them with those of a Pregel-like system; (2) We show the performance improvement incurred

by moving the computation closer to data; (3) We present the consequences of the dynamic access scheme when applied to the different datasets; (4) We demonstrate GraphMap's scalability on different cluster configurations; (5) We show the efficiency of the hash-based global partitioning scheme by comparing execution times of several algorithms after partitioning the graph using hash-based and minimum cut-based scheme, and; (6) We compare GraphMap against other state-of-the-art graph processing frameworks.

7.1 Datasets and iterative graph algorithms

To evaluate GraphMap, we use several real-world graph datasets of different sizes, as summarized in Table 1. The experiments are performed using three classes of iterative graph algorithms to adequately examine the various computation and communication characteristics shown in Fig. 5. The first category of algorithms is illustrated using PageRank where all vertices are active throughout all the iterations in the algorithm. The second category is illustrated using Connected Components (CC) where the ratio of active vertices to the total number of vertices is close to 1 during the first few iterations but quickly becomes close to 0 as the algorithm approaches convergence. The final type is illustrated using Single-Source Shortest Path (SSSP) where most vertices are inactive during the first and last few iterations and about half of them are active during the intermediate ones.

7.2 Setup and implementation

The testbed we use to evaluate GraphMap consists of a cluster of 21 nodes (1 master and 20 workers) on Emulab [29] in which we consider two types of nodes. The first type (d710) is equipped with 12 GB RAM, one quad-core Intel Xeon E5530 processor, and two 7200 rpm SATA disks (500 GB and 250 GB). They run CentOS 5.5 and are connected to each other with a 1 Gbps Ethernet network. The second type (d430) has 64 GB RAM, two 8-core Intel Xeon E5-2630 processors with two threads per core, and a 162 GB local hard drive. It has access to network-mounted storage but was not used in our experiments. Each worker machine runs three JVM processes, each with a maximum heap size of 3 GB and 16.5 GB in the d710 and d430 nodes, respectively, unless stated otherwise.

Table 1 Graph datasets for evaluation

Graph	# Vertices	# Edges
hollywood-2011 [4]	2.2 M	229 M
orkut [22]	3.1 M	224 M
cit-Patents [18]	3.8 M	16.5 M
soc-LiveJournal1 [3]	4.8 M	69 M
uk-2005 [4]	39 M	936 M
twitter [14]	42 M	1.5 B

GraphMap utilizes the BSP and messaging modules of Apache Hama (version 0.6.3), an open source implementation of Pregel, so in view of fairness, we directly compare the performance of GraphMap with Hama. The vertex blocks are represented as key-value pairs and stored on disk using Apache HBase (Version 0.96), which is an open source wide-column key-value store running on Apache Hadoop's (Version 1.0.4) Distributed File System (HDFS). There are multiple reasons for choosing HBase as the underlying NoSQL database. Firstly, HBase on top of HDFS provides replication and fault tolerance, which aligns with GraphMap's philosophy of running on cheaper commodity clusters. Secondly, HBase is well suited for both non-sequential and sequential accesses of vertex blocks since it indexes the keys on disk by sorting them using a log-structured merge (LSM) tree. This implies that a chunk of adjacent keys is stored contiguously in the same block in HDFS, which can be read sequentially using a *range-scan* operation. Lastly, HBase tables can be split into *regions* where the key-value pairs in the same region will be stored in the same machine (or in a single region server in HBase terms). Therefore, if each partition in the graph can be mapped to an HBase region, the VBs in it will be stored in a single node. This makes it convenient to partition the graph globally since we can rename the vertex identifiers in a way that they can be mapped to one of the regions. Specifically, when we perform the inter-worker partitioning (level 1), we create one HBase table that is pre-split into regions, one for each level-1 partition. For the intra-worker partitioning (level 2), we combine the level-1 partition ID with the vertex ID as an HBase key to store all vertex blocks belong to the same level-1 partition together, sorted by their vertex IDs. Thus, both the layers of our two-level partitioning scheme are on top of HBase—first by distributing vertices to different regions according to their assigned partition ID (inter-worker partitioning) and then sorting and indexing the keys on each region (intra-worker partitioning).

7.3 Iterative graph computations

Since GraphMap uses the Hama's BSP engine, we compare the total execution times of both frameworks in Table 2 using the algorithms and datasets mentioned earlier.

Table 2 Total execution time of GraphMap compared to that of Apache Hama on d430 nodes

Datasets	Total execution time (s)					
	SSSP		CC		PageRank	
	Hama	GraphMap	Hama	GraphMap	Hama	GraphMap
hollywood-2011	75.690	11.732 (6.5 ×)	90.698	23.709 (3.8 ×)	135.771	35.711 (3.8 ×)
orkut	36.664	11.737 (3.1 ×)	60.653	23.744 (2.6 ×)	81.963	35.759 (2.3 ×)
cit-Patents	15.622	8.747 (1.8 ×)	15.638	8.706 (1.8 ×)	24.769	14.747 (1.7 ×)
soc-LiveJournal1	30.688	11.755 (2.6 ×)	42.636	20.745 (2.1 ×)	54.690	26.769 (2.0 ×)
uk-2005	*	59.743	*	302.914	*	215.899
twitter	*	74.838	*	167.938	*	420.051

*Failed because of out of memory

In case of SSSP, for each of the datasets except uk-2005, the vertex with the largest number of outgoing edges is chosen as the source vertex. For uk-2005, we choose the one with the third highest out-degree since only about 0.01% are reachable from the first two. In case of PageRank, we set the termination condition to ten iterations. The execution times reported in Table 2 are obtained from the *fastest* of five runs, clearing the cache before each run, and we also report their variation in Table 3.

It is evident from the results that not only does GraphMap consistently outperform Hama in all the algorithms on all the datasets, but is also more memory efficient than Hama. Even for the smaller datasets such as cit-patents and soc-LiveJournal1 having 16.5 and 69 million edges, respectively, GraphMap is about twice as fast as Hama. The performance gap widens with an increase in the number of edges as seen in the cases of orkut and hollywood-2011 datasets with 224 and 229 million edges, respectively. This can be observed especially in case of SSSP where GraphMap is about 6 times faster than Hama. The most noteworthy results are in the cases of the uk-2005 (936 million edges) and twitter (1.5 billion edges) datasets where Hama fails to execute altogether. This demonstrates the impact of the memory and I/O efficient elements used in designing GraphMap.

For a more fine-grained examination of the difference in execution times between Hama and GraphMap, we have broken down and analyzed each individual iteration in both frameworks while running PageRank on the Orkut dataset. The time spent by a worker during an iteration was split into two parts—time taken for processing and for synchronization. The processing time is the total time taken by vertices to process messages received in the previous iteration, update their state by running the user program, and queue outgoing messages. In case of GraphMap, the processing time also includes the HBase access time. On the other hand, the synchronization time comprises of the time spent waiting for other nodes to synchronize as well as the time taken for transferring messages to peers. We observed that the average processing time of all workers in case of Hama was consistently about twice as long as that of GraphMap in every iteration of PageRank (the closest one being 1.7 ×). On the other hand, the average synchronization time of workers in Hama was 2.5 × to 3.7 × longer than that of GraphMap. On the whole, each iteration of Hama was about 2.2 to 2.5 times slower than that of GraphMap. Note that, even though Hama stores all its edge data in memory, its vertex

Table 3 Maximum, mean, and standard deviation of five runtimes of GraphMap for various algorithms and datasets on d430 nodes

Datasets	SSSP			CC			PageRank		
	Max	Avg	StDev	Max	Avg	StDev	Max	Avg	StDev
hollywood-2011	11.81	11.77	0.03	23.80	23.77	0.04	35.82	35.78	0.04
orkut	11.83	11.80	0.04	26.74	24.37	1.32	35.88	35.81	0.05
cit-Patents	8.83	8.79	0.04	8.83	8.78	0.05	14.78	14.76	0.01
soc-LiveJournal1	11.84	11.80	0.03	20.86	20.79	0.05	26.82	26.79	0.02
uk-2005	62.86	61.62	1.68	312.04	308.40	3.95	227.92	220.71	6.57
twitter	77.85	76.08	1.62	188.83	175.71	8.61	447.04	430.86	11.35

processing time was longer than that of GraphMap, which has to access the disk at each iteration. Even though this might seem counter-intuitive at first, this is expected since HBase utilizes a *block cache*, which keeps a data-block resident in memory even after its read. This is done so that adjacent records that reside in the same block can be read without multiple disk accesses, thereby improving sequential access performance. This reaffirms the validity of GraphMap's data layout and the choice of HBase for implementing it.

Figure 6 offers a closer look at the iterations in GraphMap to show how long the different components take in the different classes of algorithms on the uk-2005 dataset. Note that the time taken to update the vertex is relatively small compared to the other components. This is because the execution time is dominated by disk accesses. In case of PageRank, since all vertices are active, each worker executes the vertex program the same number of times and processes the same number of messages at every iteration (except the first and the last one). Moreover, since messages are passed along every edge in the graph, there are a lot of inter-worker messages, which increases the synchronization time. In case of SSSP and CC, the number of active vertices differs between iterations and so does the HBase access time. However, depending on the number of active vertices, GraphMap decides to perform sequential reads (iterating over range-scans), which is what happens from iterations 5 through 15 in SSSP. This is evident from the fact that the total disk access times in those iterations are similar even though the vertex update times vary.

7.4 Effects of worker-partition colocation

Table 4 demonstrates the performance improvements that result from the colocation of data and workers in GraphMap. On small datasets such as cit-Patents and orkut, we do not observe much improvement in performance with colocation, and for some algorithms such as SSSP and CC, we even notice a deterioration in the execution times. This is because, when the partitions are small, the overhead of fetching the locations of partitions for colocating workers is non-negligible compared to the time taken to fetch and process the partitions from remote peers.

However, as the size of the data (and also the partitions) increases, the access locality starts to yield noticeable performance improvements. For instance, in case of twitter and uk-2005 graphs, we can observe that for algorithms such as SSSP and CC which require a large number of iterations, the execution times with worker-data colocation are about 1.5 to 2 times faster than those without it.

The efficiency of worker-partition colocation is reinforced by the difference in network traffic per node per iteration in the two scenarios as shown in Table 5. Depending on the graph algorithm and the dataset, this scheme reduces the network traffic by up to an order of magnitude, with the highest performance gain in case of graphs containing a large number of edges.

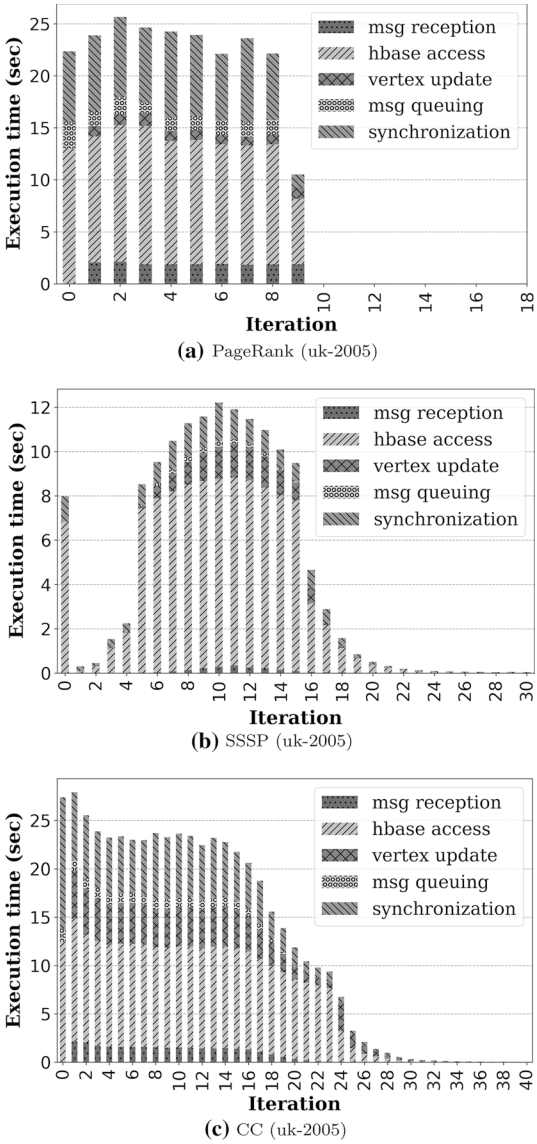


Fig. 6 Breakdown of execution time per iteration (average per worker)

7.5 Effects of dynamic access methods

To evaluate the impact incurred by dynamically switching between sequential and random disk accesses, we compare the execution times of GraphMap for the CC and SSSP algorithms with two baseline results using only sequential and only random accesses as shown in Fig. 7. We do not include PageRank in the study since all vertices remain active during every iteration of PageRank, and therefore sequential access always performs the best. We can see that using the dynamic

Table 4 Execution time (in seconds) of algorithms on different datasets with and without worker-data colocation

Datasets	SSSP		CC		PageRank	
	Colocated	Dispersed	Colocated	Dispersed	Colocated	Dispersed
cit-Patents	8.747	6.407	8.706	6.405	14.747	15.404
soc-LiveJournal1	11.755	15.347	20.745	21.406	26.769	30.377
orkut	11.737	15.411	23.744	24.466	35.759	42.421
hollywood-2011	11.732	12.409	23.709	27.418	35.711	69.428
uk-2005	59.743	123.547	302.914	433.063	215.899	336.648
twitter	74.838	114.5	167.938	225.587	420.051	705.801

Table 5 Average data transferred (in MB) *per node per iteration* and the number of iterations (**Iter**) in various algorithms on different datasets with and without worker-data colocation (Col and Disp, respectively)

Datasets	SSSP			CC			PageRank		
	Iter	Col	Disp	Iter	Col	Disp	Iter	Col	Disp
cit-Patents	15	0.32	0.42	13	1.25	2.22	10	6.41	19.21
soc-LiveJournal1	16	1.80	12.68	17	6.06	22.26	10	16.92	51.25
orkut	17	2.19	17.06	10	15.43	45.32	10	24.77	75.44
hollywood-2011	11	3.37	26.98	14	10.05	51.78	10	22.45	105.84
uk-2005	198	1.16	26.70	203	10.13	63.65	10	128.52	577.82
twitter	15	17.82	217.10	48	13.16	90.96	10	170.73	915.93

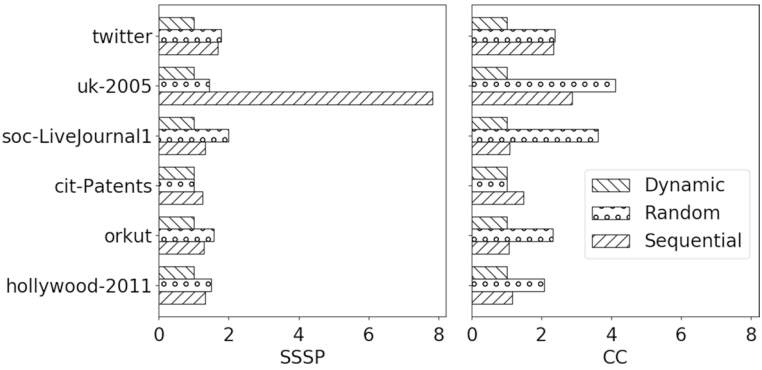


Fig. 7 Normalized execution times using different access schemes

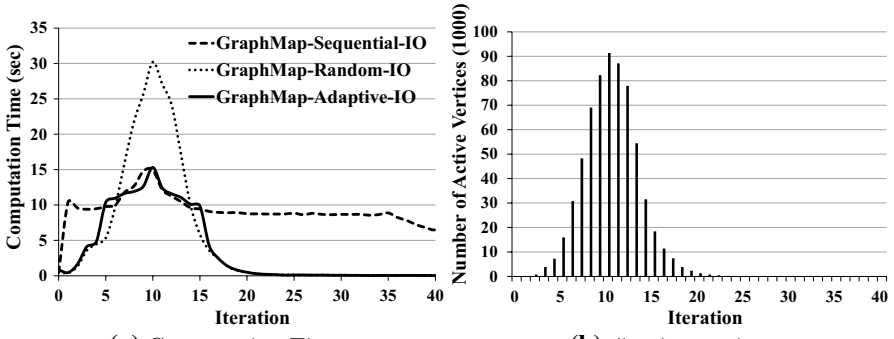
access type yields the best performance compared to the baselines since it makes a more informed decision on how to read data from the disk based on the hardware performance as well as algorithmic characteristics.

We can also observe from Fig. 7 that, in case of cit-Patents, the vertex blocks are accessed in a random pattern in all the iterations as opposed to a full sequential scan because only a tiny fraction of the vertices is active throughout the computation. We can also witness a huge performance gain (about 8 ×) for SSSP on uk-2005 compared to the baseline using only sequential accesses because it has a large number of iterations (198 iterations) until it converges, and most iterations have only a few active vertices.

Figure 8 presents a more fine-grained analysis of the dynamic access on a worker with the uk-2005 dataset (iterations 0–40). We can see that the choice of access type corresponds to the number of active vertices for most iterations. An interesting observation here is that in iterations 5 and 15, even though random accesses are faster, GraphMap selects the full sequential scan, which signifies that there is still room for improvement. One way this can be done is by fine-tuning the value of θ, which was set to 2% for all experiments performed.

7.6 Scalability

In this part of the evaluation, we begin by demonstrating GraphMap's scalability in Table 6 by running SSSP on all the datasets using different worker configurations. We perform three sets of experiments, each time changing the number of workers (60, 120 and 180) but keeping the heap size the same (1 GB). As expected, GraphMap requires less memory than Hama and can process the input graph using fewer workers. Increasing the number of workers decreases the sizes of the partitions and consequently the number of active vertices handled by each worker. This is shown in Fig. 9a, b. However, an increase in the number of workers will incur a higher inter-worker communication cost especially on commodity clusters with slower network interconnects, resulting in diminished performance improvements. As shown in Fig. 9c, d, if we raise the number of workers, the vertex update time decreases but at the cost of longer synchronization time for coordinating more workers.



(a) Computation Time **(b)** #active vertices

Fig. 8 Effects of dynamic access methods

Table 6 Scalability of Hama versus GraphMap with SSSP on d430 machines

Total execution time (sec)				
Dataset	Framework	#Workers		
		60	120	180
hollywood-2011	Hama	75.690	48.721	54.729
	GraphMap	12.383	12.399	12.393
orkut	Hama	36.664	33.688	45.705
	GraphMap	15.395	12.474	15.375
cit-Patents	Hama	15.622	18.683	24.724
	GraphMap	9.388	9.429	12.407
soc-LiveJournal1	Hama	30.688	33.714	42.753
	GraphMap	15.371	12.415	12.427
uk-2005	Hama	*	*	201.955
	GraphMap	78.426	45.464	42.463
twitter	Hama	*	*	*
	GraphMap	81.438	57.447	48.476

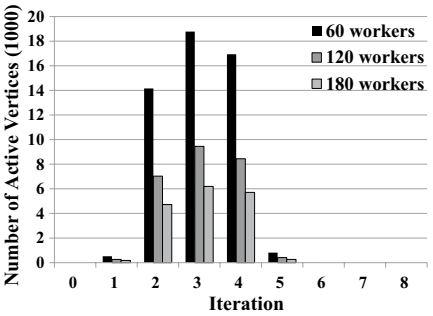
*Failed because of out of memory

7.7 Effects of partitioning scheme on performance

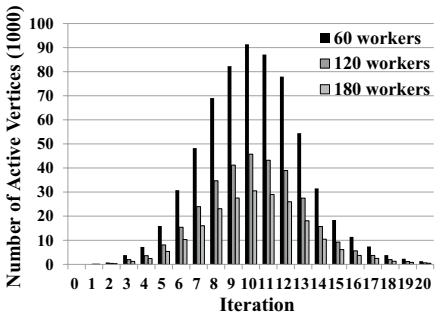
In the following set of experiments, we demonstrate the efficiency of hash-based partitioning scheme used in GraphMap by comparing it against the performance of minimum cut-based partitioning. For the minimum cut-based partitioning, we use Metis [13] to find the minimum edge-cuts in our graphs and then distribute the resulting partitions among GraphMap workers. For the hash-based partitioning, we assign vertices to workers based on the result of hashing their vertex identifiers. Next, we run the various graph algorithms on both partition assignments and gather the execution times and the total number of messages sent by all workers to their peers (both local and remote) as shown in Tables 7 and 8 respectively. Note that we do not use the worker-partition colocation during these experiments to isolate the effect of inter-node messages on the total execution times.

As expected, the number of messages in the graph partitioned using minimum cuts is much smaller (up to 8 times) than that using hash-based partitioning because connected vertices are typically assigned to the same partition. However, we observe that, for most of the datasets, the minimum cut-based partitioning scheme provides no improvement in execution times. For some algorithms (e.g., PageRank on twitter), the minimum cut-based partitioning scheme is even slower (by about four times) than the hash-based one. Note that the execution time does not include the time taken to calculate the minimum cuts using Metis.

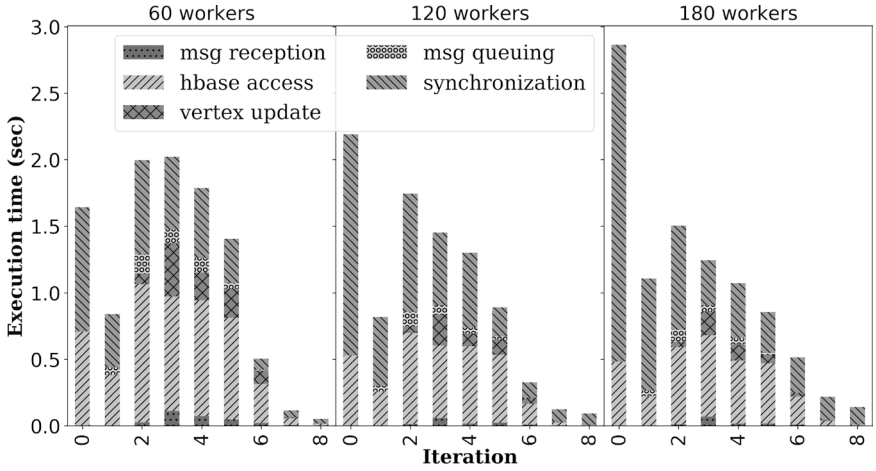
The reason behind the diminished performance when using mincut-based partitioning lies in the degree distribution and small-world property of real-world graphs combined with the bulk synchronous nature of GraphMap. More specifically, while trying to reduce the number of edges across partitions, the mincut-based partitioning may inadvertently assign two connected hubs (i.e., vertices with a large number



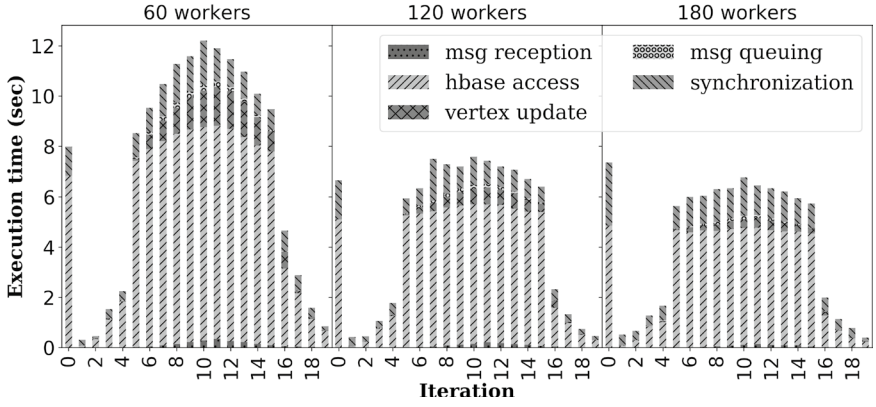
(a) #active vertices (orkut)



(b) #active vertices (uk-2005)



(c) computation time (orkut)



(d) computation time (uk-2005)

Fig. 9 Scalability (varying #workers)

Table 7 Effects of various partitioning schemes on total execution time (d430 nodes without the worker-partition colocation)

Datasets	SSSP		CC		PageRank	
	Mincut	Hash	Mincut	Hash	Mincut	Hash
hollywood-2011	24.419	12.409	63.485	27.418	111.51	42.811
orkut	21.418	15.411	39.427	24.466	69.453	39.818
cit-Patents	9.447	6.407	7.231	6.405	18.378	18.632
soc-LiveJournal1	15.415	15.347	24.391	21.406	39.427	33.408
uk-2005	204.561	123.547	517.255	433.063	280.266	229.351
twitter	354.681	114.5	694.493	225.587	2060.011	531.716

Table 8 Average number of messages transferred *between all workers* (running on same or different compute nodes) *per iteration* in various algorithms on different datasets with mincut and hash-based partitioning

Datasets	SSSP		CC		PageRank	
	Mincut	Hash	Mincut	Hash	Mincut	Hash
hollywood-2011	1.40M	7.30M	4.35M	24.87M	10.23M	56.05M
orkut	1.65M	4.58M	11.49M	38.56M	18.42M	62.11M
cit-Patents	11.87K	14.07K	0.39M	1.16M	4.527M	13.487M
soc-LiveJournal1	2.90M	3.20M	12.68M	14.28M	36.08M	40.36M
uk-2005	0.37M	2.25M	3.26M	25.41M	37.68M	308.3M
twitter	18.90M	39.01M	13.57M	31.17M	174.1M	422.0M

of edges) to the same partition. This means that a few partitions can have a much higher number of edges than the rest and may need to process a much larger number of messages per iteration. This can be observed in Fig. 10, which depicts the skewness in the edge distribution for the three largest datasets across a fixed number of partitions (60) for both partitioning schemes. In a bulk synchronous processing model where all workers must synchronize before every iteration, the execution time for each iteration is determined by the workers taking the longest processing time. Therefore, when the number of edges is skewed, the execution time of an iteration is dominated by the worker processing the partition containing the largest number of edges.

7.8 Comparison with state-of-the-art systems

In this part, we evaluate GraphMap's performance against the popular distributed graph processing frameworks such as Hama, GraphX, PowerGraph (GraphLab 2.2), Giraph, and Giraph++, as shown in Table 9. For the comparisons, we show the results of running CC and PageRank on two of the largest datasets, twitter and uk

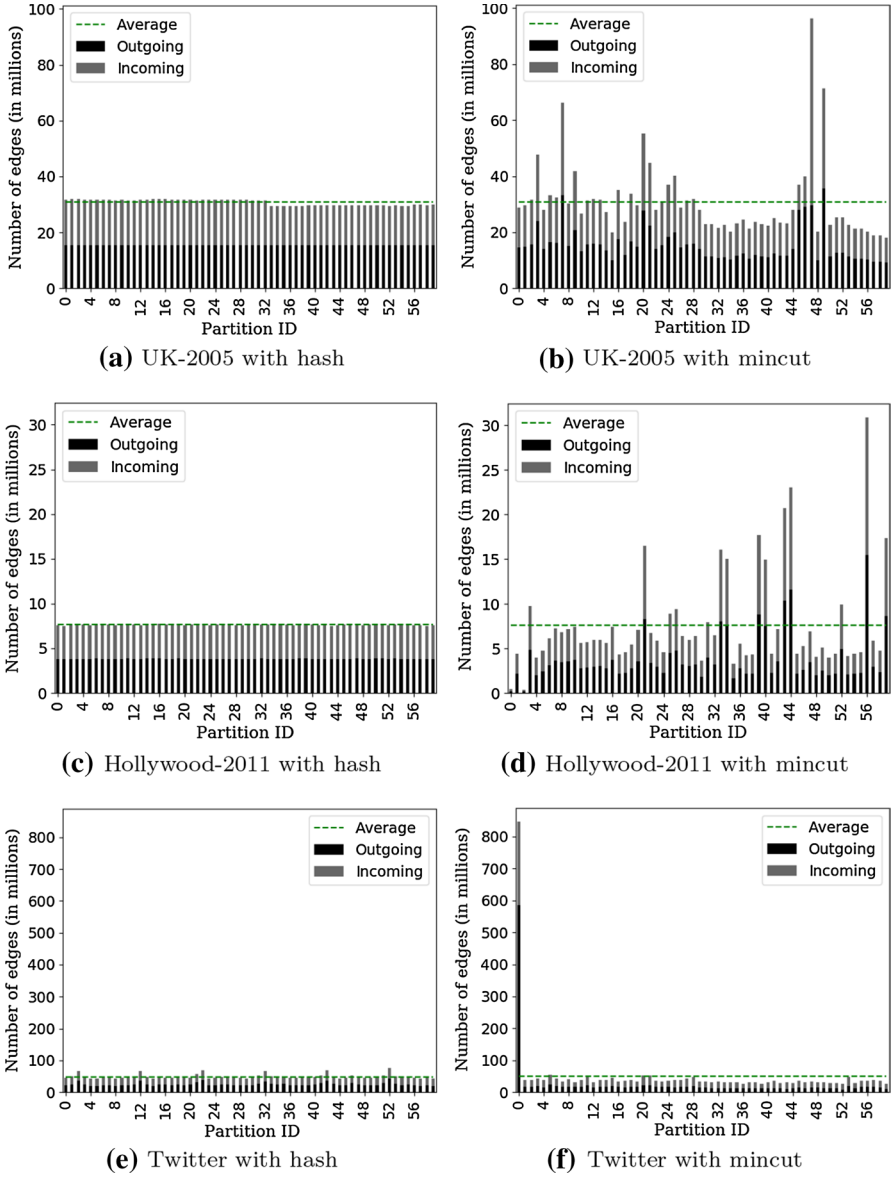


Fig. 10 Distribution of total number of incoming and outgoing edges per partition

datasets. To prevent the effects of our sub-optimal system configurations, we adopt the results reported in literature [8, 31]. Moreover, since Chaos [24] reports normalized runtimes, we use the ones reported by GraphD [30]. The results are annotated with the hardware configurations used to generate them.

The results provide some interesting observations. Firstly, the other frameworks were evaluated on clusters with larger aggregate main memory and processing

Table 9 Comparison of GraphMap on thin (d710) nodes with other systems

System	Settings	CC (s)		PageRank (sec./iteration)		Type
		twitter	uk-2005 (*uk-2007)	twitter	uk-2005 (*uk-2007)	
GraphMap on Hadoop	21 nodes ($21 \times 4 = 84$ cores, $21 \times 12 = 252$ GB RAM)	319	695	83	46	Out-of-core
Hama on Hadoop	21 nodes ($21 \times 4 = 84$ cores, $21 \times 12 = 252$ GB RAM)	Fail	Fail	Fail	Fail	In-memory
GraphX on Spark	16 nodes ($16 \times 8 = 128$ cores, $16 \times 68 = 1088$ GB RAM)	251	800*	21	23*	In-memory
GraphLab 2.2 (PowerGraph)	16 nodes ($16 \times 8 = 128$ cores, $16 \times 68 = 1088$GB RAM)	244	714*	12	42*	In-memory
Giraph 1.1 on Hadoop	16 nodes ($16 \times 8 = 128$ cores, $16 \times 68 = 1088$ GB RAM)	200	Fail*	30	62*	In-memory
Giraph++ on Hadoop	10 nodes ($10 \times 8 = 80$ cores, $10 \times 32 = 320$ GB RAM)	No result reported	723	No result reported	89	In-memory
Chaos	15 nodes ($15 \times 12 = 180$ cores, $15 \times 48 = 720$ GB RAM)	No result reported	No result reported	470	No result reported	Out-of-core
GraphD	16 nodes ($16 \times 4 = 64$ cores, $16 \times 8 = 128$ GB RAM)	No result reported	No result reported	46	No result reported	Out-of-core

power. For instance, Giraph, GraphLab, and GraphX were evaluated on an aggregate memory of 1 TB and 128 cores whereas GraphMap was tested using 84 cores on 256 GB RAM. In case of CC, GraphMap's performance is comparable to that of the other frameworks even while using fewer resources than the rest. In case of the uk dataset, GraphMap is even faster than some of the others. GraphMap demonstrates competitive performance by effectively accessing disk through a set of optimization techniques such as the dynamic disk access scheme and worker-partition colocation. In case of PageRank, the difference in execution times is more noticeable since GraphMap has to read all vertex blocks from disk at every iteration while using fewer cores than the rest. Note that Chaos takes significantly longer than GraphMap since it admittedly performs well only on high-speed networks, whereas all experiments reported on Table 9 were performed on 1 Gbps links. GraphD is a newest out-of-core graph processing system. Unlike GraphMap, GraphD is not built atop general-purpose tools such as HDFS and HBase and hence it does not have to incur the performance costs that come with them due to several important features such as fault tolerance. Moreover, GraphD is programmed in C++, which automatically puts GraphMap at a disadvantage since it is programmed in Java that can be 2–3 times slower than C++ [11]. These results and observations exhibit the efficacy of GraphMap in iterative processing of large datasets on constrained environments.

8 Conclusion

In this work, we present GraphMap, a distributed iterative framework capable of processing large graphs on a small cluster by effectively utilizing secondary storage through access locality-optimized techniques. This paper makes the following contributions. Firstly, we propose a clean separation of storage between mutable and immutable graph data during the lifetime of the computation. With this approach, we can optimize the storage scheme to exploit the access locality in graphs thereby increasing sequential rather than random disk I/O. Secondly, we present a two-level graph data partitioning scheme (inter-worker and intra-worker partitioning) for locality-optimized data placement and balanced workloads. Moreover, we introduce a collection of optimization techniques based on the access locality to improve I/O performance and execution time. Lastly, we demonstrate GraphMap's performance through a comprehensive set of experiments and establish that it even outperforms distributed in-memory graph processing frameworks for several classes of graph algorithms.

Acknowledgements Funding was provided by Louisiana Board of Regents (Grant No. LEQSF(2016-19)-RD-A-08) and National Science Foundation (Grant No. IBSS-L-1620451 and RAPID-1762600).

References

1. Apache Giraph. http://giraph.apache.org/. Accessed 12 Mar 2019
2. Apache Hama. https://hama.apache.org/. Accessed 12 Mar 2019

3. Backstrom L, Huttenlocher D, Kleinberg J, Lan X (2006) 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, KDD'06. ACM, New York, NY, USA, pp 44–54. https://doi.org/10.1145/1150402.1150412
4. Boldi P, Vigna S (2004) The WebGraph framework I: compression techniques. In: Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004). ACM Press, Manhattan, USA, pp 595–601
5. Bu Y, Borkar V, Jia J, Carey MJ, Condie T (2014) Pregelix: Big(Ger) graph analytics on a dataflow engine. Proc VLDB Endow 8(2):161–172
6. Facebook Reports Third Quarter 2019 Results. https://investor.fb.com/investor-news/press-release-details/2019/Facebook-Reports-Third-Quarter-2019-Results/default.aspx. Accessed 12 Mar 2019
7. Gonzalez JE, Low Y, Gu H, Bickson D, Guestrin C (2012) PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12). USENIX Association, Berkeley, CA, USA, pp 17–30. http://dl.acm.org/citation.cfm?id=2387880.2387883. Accessed 12 Mar 2019
8. Gonzalez JE, Xin RS, Dave A, Crankshaw D, Franklin MJ, Stoica I (2014) GraphX: graph processing in a distributed dataflow framework. In: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14). USENIX Association, Berkeley, CA, USA, pp 599–613. http://dl.acm.org/citation.cfm?id=2685048.2685096. Accessed 12 Mar 2019
9. Goswami S, Das AK, Płatania R, Lee K, Park SJ (2016) Lazer: Distributed memory-efficient assembly of large-scale genomes. In: 2016 IEEE International Conference on Big Data (Big Data), pp 1171–1181. https://doi.org/10.1109/BigData.2016.7840721
10. Han WS, Lee S, Park K, Lee JH, Kim MS, Kim J, Yu H (2013) 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). ACM, New York, NY, USA, pp 77–85. https://doi.org/10.1145/2487575.2487581
11. Hundt R (2011) Loop recognition in C++/Java/go/scala. Proc Scala Days 2011:38
12. Jia Z, Kwon Y, Shipman G, McCormick P, Erez M, Aiken A (2017) A distributed multi-GPU system for fast graph processing. Proc VLDB Endow 11(3):297–310
13. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392. https://doi.org/10.1137/S1064827595287997
14. Kwak H, Lee C, Park H, Moon S (2010) What is Twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web, WWW'10. ACM, New York, NY, USA, pp 591–600. https://doi.org/10.1145/1772690.1772751
15. Kyrola A, Blelloch G, Guestrin C (2012) GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12. USENIX Association, Berkeley, CA, USA, pp 31–46. http://dl.acm.org/citation.cfm?id=2387880.2387884
16. Lee K, Liu L (2013) Scaling queries over Big RDF graphs with semantic hash partitioning. Proc VLDB Endow 6(14):1894–1905
17. Lee K, Liu L, Schwan K, Pu C, Zhang Q, Zhou Y, Yigitoglu E, Yuan P (2015) Scaling iterative graph computations with GraphMap. In: SC15: International Conference for High Performance Computing, Networking, Storage and Analysis, pp 1–12. https://doi.org/10.1145/2807591.2807604
18. Leskovec J, Kleinberg J, Faloutsos C (2005) Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining (KDD'05). ACM, New York, NY, USA, pp 177–187. https://doi.org/10.1145/1081870.1081893
19. Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein JM (2012) Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc VLDB Endow 5(8):716–727
20. Maass S, Min C, Kashyap S, Kang W, Kumar M, Kim T (2017) Mosaic: processing a trillion-edge graph on a single machine. In: Proceedings of the Twelfth European Conference on Computer Systems, EuroSys'17. ACM, New York, NY, USA, pp 527–543. https://doi.org/10.1145/3064176.3064191
21. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD'10). ACM, New York, NY, USA, pp 135–146. https://doi.org/10.1145/1807167.1807184

22. Mislove A, Marcon M, Gummadi KP, Druschel P, Bhattacharjee B (2007) Measurement and analysis of online social networks. In: Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07), San Diego, CA
23. Pan Y, Wang Y, Wu Y, Yang C, Owens JD (2017) Multi-GPU graph analytics. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp 479–490
24. Roy A, Bindschaedler L, Malicevic J, Zwaenepoel W (2015) Chaos: scale-out graph processing from secondary storage. In: Proceedings of the 25th Symposium on Operating Systems Principles. ACM, pp 410–424
25. Roy A, Mihailovic I, Zwaenepoel W (2013) X-Stream: edge-centric graph processing using streaming partitions. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13). ACM, New York, NY, USA, pp 472–488. https://doi.org/10.1145/2517349.2522740
26. Shao B, Wang H, Li Y (2013) Trinity: a distributed graph engine on a memory cloud. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13). ACM, New York, NY, USA, pp 505–516. https://doi.org/10.1145/2463676.2467799
27. Tian Y, Balmin A, Corsten SA, Tatikonda S, McPherson J (2013) From "think like a vertex" to "think like a graph. Proc VLDB Endow 7(3):193–204
28. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111. https://doi.org/10.1145/79173.79181
29. White B, Lepreau J, Stoller L, Ricci R, Guruprasad S, Newbold M, Hibler M, Barb C, Joglekar A (2002) An integrated experimental environment for distributed systems and networks. In: Proceedings of the 5th Symposium on Operating Systems Design and implementation (OSDI'02). ACM, New York, NY, USA, pp 255–270. https://doi.org/10.1145/1060289.1060313
30. Yan D, Huang Y, Liu M, Chen H, Cheng J, Wu H, Zhang C (2018) Graphd: distributed vertex-centric graph processing beyond the memory limit. IEEE Trans Parallel Distrib Syst 29(1):99–114
31. Yuan P, Zhang W, Xie C, Jin H, Liu L, Lee K (2014) Fast iterative graph computation: a path centric approach. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14). IEEE Press, Piscataway, NJ, USA, pp 401–412. https://doi.org/10.1109/SC.2014.38
32. Zerbino DR, Birney E (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Res 18(5):821–829
33. Zhang Y, Liao X, Jin H, He B, Liu H, Gu L (2019) Digraph: an efficient path-based iterative directed graph processing system on multiple GPUs. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, pp 601–614
34. Zheng D, Mhembere D, Burns R, Vogelstein J, Priebe CE, Szalay AS (2015) FlashGraph: processing billion-node graphs on an array of commodity SSDs. In: 13th USENIX Conference on File and Storage Technologies (FAST 15). USENIX Association, Santa Clara, CA, pp 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng. Accessed 12 Mar 2019
35. Zhou Y, Liu L, Lee K, Zhang Q (2015) Graphtwist: fast iterative graph computation with two-tier optimizations. Proc VLDB Endow 8(11):1262–1273. https://doi.org/10.14778/2809974.2809987

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.