Scaling Iterative Graph Computations with GraphMap

Kisung Lee ¶, Ling Liu †, Karsten Schwan †, Calton Pu †, Qi Zhang †, Yang Zhou †,
Emre Yigitoglu †, Pingpeng Yuan ‡
¶Louisiana State University
†Georgia Institute of Technology
‡Huazhong University of Science & Technology

ABSTRACT

In recent years, systems researchers have devoted considerable effort to the study of large-scale graph processing. Existing distributed graph processing systems such as Pregel, based solely on distributed memory for their computations, fail to provide seamless scalability when the graph data and their intermediate computational results no longer fit into the memory; and most distributed approaches for iterative graph computations do not consider utilizing secondary storage a viable solution. This paper presents GraphMap, a distributed iterative graph computation framework that maximizes access locality and speeds up distributed iterative graph computations by effectively utilizing secondary storage. GraphMap has three salient features: (1) It distinguishes data states that are mutable during iterative computations from those that are read-only in all iterations to maximize sequential access and minimize random access. (2) It entails a two-level graph partitioning algorithm that enables balanced workloads and locality-optimized data placement. (3) It contains a proposed suite of locality-based optimizations that improve computational efficiency. Extensive experiments on several real-world graphs show that GraphMap outperforms existing distributed memory-based systems for various iterative graph algorithms.

Keywords

graph data processing, distributed computing

1. INTRODUCTION

Graphs are pervasively used for modeling information networks of real-world entities with sophisticated relationships. Many applications from science and engineering to business domains use iterative graph computations to analyze large graphs and derive deep insight from a huge number of explicit and implicit relationships among entities. Considerable research effort on scaling large graph computations has been devoted to two different directions: One is to deploy a super powerful many-core computer with memory capacity of hundreds or thousands of gigabytes [19] and another is to explore the feasibility of using a cluster of distributed commodity servers.

Most of research effort on deploying a supercomputer for fast iterative graph computations assumes considerable computing resources and thus focuses primarily on parallel optimization techniques that can maximize parallelism among many cores and tasks performed on the cores. A big research challenge for efficient many-core computing is the tradeoff between the opportunities for massive parallel computing and the cost of massive synchronization for multiple iterations, which tends to make the overall performance of parallel processing significantly less optimal at the high ownership cost of supercomputers.

As commodity computers become pervasive for many scientists and small and medium-sized enterprise organizations, we witness a rapidly growing demand for distributed iterative graph computations on a cluster of commodity servers. Google's Pregel [17] and its open source implementations − Apache Giraph [1] and Hama [2] − have shown remarkable initial success. However, existing distributed graph processing systems, represented by Pregel, heavily rely on distributed memory-based computation model. Concretely, a large graph is first distributed using random or hash partitioning to achieve data-level load balance. Next, each compute node of the cluster needs to not only load the entire local graph in memory but also store both the intermediate results of iterative computations and all the communication messages it needs to send to and receive from every other node of the cluster. Therefore, existing approaches suffer from poor scalability when any compute node in the cluster fails to hold the local graph and all the intermediate (computation and communication) results in memory. The dilemma lies in the fact that simply increasing the size of the compute cluster often fails for iterative computations on large graphs. This is because, with a larger cluster, one can reduce the size of the graph that needs to be held in memory at the price of a significantly increased amount of distributed messages each node needs to send to and receive from a larger number of nodes in the cluster. For example, a recent study [23] shows that computing five iterations of PageRank on the twitter 2010 dataset with 42 millions of vertices and 1.5 billions of edges takes 487 seconds on a Spark [29] cluster with 50 nodes and 100 CPUs. Another study [24] shows that counting triangles on the twitter 2010 dataset takes 423 minutes on a large Hadoop cluster with 1636 nodes. Surprisingly, most of the existing approaches for iterative graph computations on a cluster of servers do

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807604

not explore the option of integrating secondary storage as a viable solution because of a potentially large number of random disk accesses.

In this paper we argue that secondary storage can play an important role in maximizing both in-memory and on-disk access locality for running iterative graph algorithms on large graphs. By combining it with efficient processing at each local node of a compute cluster, one can perform iterative graph computations more efficiently than the distributed memory-based model in many cases. The ability to intelligently integrate secondary storage into the cluster computing infrastructure for memory intensive iterative graph computations can be beneficial from multiple dimensions: (i) With efficient management of in-memory and on-disk data, one can not only reduce the size of the graph partitions to be held at each node of the cluster but also match the performance of the distributed memory-based system. (ii) One can carry out expensive iterative graph computations on large graphs using a much smaller and affordable cluster (tens of nodes), instead of relying on the availability of a large cluster with hundreds or thousands of compute nodes, which is still costly even with pay-as-you-go elastic cloud computing pricing model.

With these problems and design objectives in mind, we develop GraphMap, a distributed iterative graph computation framework, which can effectively utilize secondary storage for memory-intensive iterative graph computations by maximizing in-memory and on-disk access locality. GrapMap by design has three salient features. First, we distinguish those data states that are mutable during iterative computations from those that are read-only during iterative computations to maximize sequential accesses and minimize random accesses. We show that by keeping mutable data in memory and read-only data on secondary storage, we can significantly improve disk IO performance by minimizing random disk IOs. Second, we support three types of vertex blocks (VBs) for each vertex: in-VB for in-edges of a vertex, out-VB for out-edges of a vertex, and bi-VB for all edges of a vertex. We also devise a two-level graph partitioning algorithm that enables balanced workloads and locality-optimized data placement. Concretely, we use hash partitioning to distribute vertices and their vertex blocks to different compute nodes and then use range partitioning to group the vertices and their vertex blocks within each hash partition into storage chunks of fixed size. Last but not the least, we propose a suite of locality-based optimizations to improve computation efficiency, including progressive pruning of non-active vertices and edges to reduce the unnecessary memory and CPU consumption, partition-aware identifier assignment, partition-aware message batching and local merge of partial updates. These design features enable GraphMap to achieve two objectives at the same time: (1) to minimize non-sequential disk IOs and significantly improve the secondary storage performance, making the integration of external storage a viable solution for distributed processing of large graphs and (2) to minimize the communication cost among different graph partitions and improve the overall computation efficiency of iterative graph algorithms. We evaluate GraphMap on a number of real graph datasets using several graph algorithms by comparing with existing representative distributed memory-based graph processing systems, such as Apache Hama. Our experimental results show that GraphMap outperforms existing distributed graph systems for various iterative graph algorithms.

2. GRAPHMAP OVERVIEW

In this section, we first define some basic concepts used in GraphMap and then provide an overview of GraphMap including its partitioning technique, programming API, and system architecture.

2.1 Basic Concepts

GraphMap models all information networks as directed graphs. For an undirected graph, we convert each undirected edge into two directed edges.

DEFINITION 1. (Graph) A graph is denoted by $G = (V, E)$ where V is a set of vertices and E is a set of directed edges (i.e., $E \subseteq V \times V$). For an edge e such that $\{e = (u, v) \in E, u, v \in V\}$, we call u and v the *source vertex* and *destination vertex* of e respectively. e is an *in-edge* of vertex v and an *out-edge* of vertex u. $|V|$ and $|E|$ denote the number of vertices and edges respectively.

A unique vertex identifier is assigned to each vertex, and a vertex may be associated with a set of attributes describing the properties of the entity represented by the vertex. For presentation convenience, we interchangeably use the terms "vertex attribute" and "vertex state" throughout this paper. For a weighted graph where each edge has its modifiable, user-defined value, we model each edge weight as an attribute of its source vertex. This allows us to treat all vertices as mutable data and edges as immutable data during iterative graph computations. For instance, when a graph is loaded for PageRank computations and vertex u has its out-edge degree $d(u)$, the graph topology does not change and each of u's out-edges contributes the fixed portion (i.e., $1/d(u)$) to the next round of PageRank values during all the iterations. Thus, we can consider edges immutable for PageRank computations. Similarly for SSSP (Single-Source Shortest Path) computations, the edge weight usually denotes the distance of a road segment and thus is immutable during the computations. This separation between mutable and immutable data by design provides GraphMap an opportunity to employ compact and locality-aware graph storage structure for both in-memory and on-disk placement. Furthermore, since most of large graphs have at least tens or hundreds times more edges than vertices, we can significantly reduce the memory requirement for loading and processing large graphs. We will show in the subsequent sections that by utilizing such a clean separation between mutable and immutable data components in a graph, we can significantly reduce the amount of non-sequential accesses in each iteration for many iterative graph algorithms.

In order to provide access locality-optimized grouping of edges in GraphMap, we categorize all edges connected to a vertex into three groups based on their direction: out-edges, in-edges, and bi-edges.

DEFINITION 2. (Out-edges, In-edges, and Bi-edges) Given a graph $G = (V, E)$, the set of **out-edges** of vertex $v \in V$ is denoted by $E_v^{out} = \{(v, v') | (v, v') \in E\}$. Conversely, the set of **in-edges** of v is denoted by $E_v^{in} = \{(v', v) | (v', v) \in E\}$. We also define **bi-edges** of v as the union of its out-edges and in-edges, denoted by $E_v^{bi} = E_v^{out} \cup E_v^{in}$.

For each graph to be processed by GraphMap, we build a vertex block (VB) for each vertex. A vertex block consists of an *anchor* vertex and its directly connected edges and vertices. Since different graph algorithms may have different computation characteristics, in GraphMap, we support three different types of vertex blocks based on the edge direction from the anchor vertex: (1) out-edge vertex block (out-VB), (2) in-edge vertex block (in-VB), and (3) bi-edge vertex block (bi-VB). One may view an out-edge vertex block as a source vertex and its adjacency list via out-edges (i.e., the list of destination vertex IDs connected to the same source vertex via its out-edges). Similarly, an in-edge vertex block can be viewed as a destination vertex and its adjacency list via its in-edges (i.e., the list of source vertex IDs connected to the same destination vertex via its in-edges). We formally define the vertex block as follows.

DEFINITION 3. (Vertex block) Given a graph $G = (V, E)$ and vertex $v \in V$, the **out-edge vertex block** of vertex v is a subgraph of G, which consists of v as its anchor vertex and all of its out-edges, denoted by $VB_v^{out} = (V_v^{out}, E_v^{out})$ such that $V_v^{out} = \{v\} \cup \{v^{out} | v^{out} \in V, (v, v^{out}) \in E_v^{out}\}$. Similarly, the **in-edge vertex block** of v is defined as $VB_v^{in} = (V_v^{in}, E_v^{in})$ such that $V_v^{in} = \{v\} \cup \{v^{in} | v^{in} \in V, (v^{in}, v) \in E_v^{in}\}$. We define the **bi-edge vertex block** of v as $VB_v^{bi} = (V_v^{bi}, E_v^{bi})$ such that $V_v^{bi} = V_v^{in} \cup V_v^{out}$.

In the subsequent sections we will describe several highlights of the GraphMap design.

2.2 Two-Phase Graph Partitioning

We design a two-phase graph partitioning algorithm, which performs global hash partitioning followed by local range partitioning at each of the n worker machines in a compute cluster. Hash partitioning on vertex IDs first divides a large graph into a set of vertex blocks (VBs) and then assigns each VB to one worker machine. By using the lightweight global hash partitioning, a large graph can be rapidly distributed across the cluster of n worker machines while ensuring data-level load balance. In order to reduce non-sequential disk accesses at each worker machine, we sort all VBs assigned to each worker machine in the lexical order of their anchor vertex IDs and further partition the set of VBs at each worker machine into r chunks such that VBs are clustered physically by their chunk ID. The parameter r is chosen such that each range partition (chunk) can fit into the working memory available at the worker machine. The range partitioning is performed in parallel at all the worker machines.

DEFINITION 4. (Hash partitioning) Let $G = (V, E)$ denote an input graph. Let $hash(v)$ be a hash function for partitioning and $VB(v)$ denote the vertex block anchored at vertex v. The hash partitioning P of G is represented by a set of n partitions, denoted by $\{P_1, P_2, \ldots, P_n\}$ where each partition P_i $(1 \leq i \leq n)$ consists of a set of vertices V_i and a set of VBs B_i such that $V_i = \{v | hash(v) = i, v \in V\}$, $B_i = \{VB(v) | v \in V_i\}$ and $\bigcup_i V_i = V$, $V_i \bigcap V_j = \emptyset$ for $1 \leq i, j \leq n$, $i \neq j$.

In the first prototype implementation of GraphMap, given a cluster of n worker machines, we physically store a graph at n worker machines by hash partitioning and then divide the set of VBs assigned to each of the n worker machines into r range partitions. GraphMap uses the hash partitioning by default for global graph partitioning across the cluster of n worker machines because it is super fast and we do not need to keep any additional data structure to record the partition assignment for each vertex. However, GraphMap can be easily extended to support any other partitioning techniques because its in-memory and on-disk representation is designed to store partition assignments generated by any partitioning techniques, such as Metis [10], ParMetis [6], and SHAPE [14].

2.3 Supporting Vertex-Centric API

Most iterative graph processing systems adopt the "think like a vertex" vertex-centric programming model [17, 7, 16]. To implement an iterative graph algorithm based on the vertex-centric model, users write a vertex-centric program, which defines what each vertex does for *each* iteration of the user-defined iterative graph algorithm, such as PageRank, SSSP, and Triangle Counting. In each iteration, vertices of the input graph execute the same vertex program in parallel. A typical vertex program consists of three steps in each iteration: (1) A vertex reads its current value and gathers its neighboring vertices' values, usually along its in-edges. (2) The vertex may update its value based on its current value and gathered values. (3) If updated, the vertex propagates its updated value to its neighboring vertices, usually along its out-edges.

Each vertex has its transition state flag with either *active* or *inactive*. In each iteration, only active vertices run the vertex program. For some algorithms such as PageRank and Connected Component (CC), every vertex is active in the first iteration and thus all vertices participate in the computation. On the other hand, for some algorithms such as SSSP, only one vertex is active in the first iteration and some vertices may be inactive during all the iterations. A vertex can deactivate itself, usually at the end of an iteration, and can also be reactivated by other vertices. The iterative graph algorithm terminates if all vertices are inactive or a user-defined convergence condition, such as the number of iterations, is satisfied.

Existing distributed iterative graph processing systems provide a mechanism for interaction among vertices, mostly along edges. Pregel [17] employs a pure message passing model in which vertices interact by sending messages along their outgoing edges and, in the current iteration, each vertex receives messages sent by other vertices in the previous iteration. In GraphLab/PowerGraph [7, 16], vertices directly read their neighboring vertices' data through shared state.

One representative category of existing distributed graph processing systems is based on the Bulk Synchronous Parallel (BSP) [26] computation model and the shared-nothing architecture. A typical graph application based on the BSP model starts with an initialization step in which the input graph is read and partitioned/distributed across the worker machines in the cloud. In subsequent iterations, the worker machines compute independently in parallel in each iteration and the iterations are separated by global synchronization barriers in which the worker machines communicate each other to integrate the results from distributed computations performed at different workers. Finally, the graph application finishes by writing down its results.

Algorithm 1 shows an example of the Single-Source

Algorithm 1 SSSP in Apache Hama

```
    compute(messages)
 1: if getSuperstepCount() == 0 then
 2:     setValue(INFINITY);
 3: end if
 4: int minDist = isStartVertex() ? 0 : INFINITY;
 5: for int msg : messages do
 6:     minDist = min(minDist, msg);
 7: end for
 8: if minDist < getValue() then
 9:     setValue(minDist);
10:     for Edge e : getEdges() do
11:         sendMessage(e, minDist+e.getValue())
12:     end for
13: end if
14: voteToHalt();
    combine(messages)
15: return min(messages)
```

Shortest Path (SSSP) algorithm, based on the vertex-centric model and the BSP model, implemented in Apache Hama's graph package, an open-source implementation of Pregel. In iteration (or *superstep*) 0, each vertex sets its vertex value as *infinity* (line 2). In subsequent iterations, each vertex picks the smallest distance among the received messages (line 5-7) and, if the distance is smaller than its current vertex value, the vertex updates its vertex value using the smallest distance (line 9) and propagates the updated distance to all its neighboring vertices along out-edges (line 10-12). At the end of each iteration, it changes its status to *inactive* (line 14). If the vertex receives any message, it will be reactivated in the next iteration and then run the vertex program again. To reduce the number of messages over the network, users can define a combiner, which finds the minimum value of messages for each destination vertex (line 15).

2.4 GraphMap Programming API

GraphMap supports two basic programing abstractions at API level: the vertex-centric model, similar to Pregel-like systems, and the VB partition-centric model. Given a vertex-centric program, such as SSSP in Algorithm 1, GraphMap converts it into a GraphMap program, which utilizes the in-memory and on-disk representation of GraphMap and the performance optimizations enabled by GraphMap in terms of access locality and efficient memory consumption (See the next sections for detail).

The VB partition-centric API is provided by GraphMap for advanced users who are familiar with GraphMap's advanced features and the BSP model. Note that, unlike the vertex-centric model, a VB partition-centric program defines what each *VB partition* (a set of VBs) does for *each* iteration. Table 1 shows some core methods provided by GraphMap. Algorithm 2 demonstrates how SSSP is implemented using the VB partition-centric API. We emphasize that we are not claiming that the VB partition-centric API is more concise than the vertex-centric API. Our main goal of the VB partition-centric API is to expose partition-level methods and thus provide more optimization opportunities for advanced users. Recall that users can run their vertex-centric programs in GraphMap as they are without the need to learn the VB partition-centric API. Because of the space constraint, we here omit the further detail on this advanced API design.

2.5 System Architecture

Fig. 1 shows the system architecture of GraphMap. Similar to Pregel-like systems, GraphMap supports the BSP model and the message passing model for iterative graph computations. GraphMap consists of a master machine and a set of worker machines. The master accepts graph analysis requests from users and coordinates the worker machines to run the graph algorithms on the input graph datasets. For large graphs, the two-phase graph partitioning task is also distributed by the master to its worker machines. The worker machines execute the graph programs by interacting with each other through messages.

Each worker machine can define a set of worker slots for task-level parallelism and each worker is in charge of a single partition. Each worker task keeps the mutable data of its assigned partition in memory (the set of vertices) and in each iteration, it reads the invariant data of the partition from disk (VB blocks) for graph computations and updating the mutable data. In addition, each worker task receives messages from and sends messages to other workers using the messaging engine and enters the global synchronization barrier of the BSP model at the end of each iteration using the BSP engine. We categorize the messages into two types based on the use of the network: *intra-machine* messages between two workers in the same worker machine and *inter-machine* messages between two workers in the different worker machines. In GraphMap, we bundle a set of messages to be transferred to the same worker at the end of

method	description
setValue(vertex, value)	update the value of the vertex
getValue(vertex)	return the value of the vertex
readAllVertices()	return an iterator for all vertices of this partition
readActiveVertices()	return an iterator for all active vertices of this partition
setActive(vertex)	set the vertex as active
createMessage(vertex, value)	create a message including the destination vertex and value
sendMessage(worker, msg)	send the message to the worker
getWorker(vertex)	return the worker, which is in charge of the vertex
deactivateAll()	deactivate all vertices of this partition

Table 1: GraphMap Core Methods

Algorithm 2 SSSP in GraphMap

```
    compute(messages)
 1: if getSuperstepCount() == 0 then
 2:     for Vertex v : readAllVertices() do
 3:         if v.isStartVertex() then
 4:             setValue(v, 0);
 5:             setActive(v);
 6:         else
 7:             setValue(v, INFINITY);
 8:         end if
 9:     end for
10: end if
11: for Message msg : messages do
12:     if msg.value < getValue(msg.target) then
13:         setValue(msg.target, msg.value);
14:         setActive(msg.target);
15:     end if
16: end for
17: for Vertex v : readActiveVertices() do
18:     for Edge e : v.getEdges() do
19:         msg = createMessage(e.destination,
                    getValue(v)+e.getValue());
20:         sendMessage(getWorker(e.destination), msg);
21:     end for
22: end for
23: deactivateAll();
```

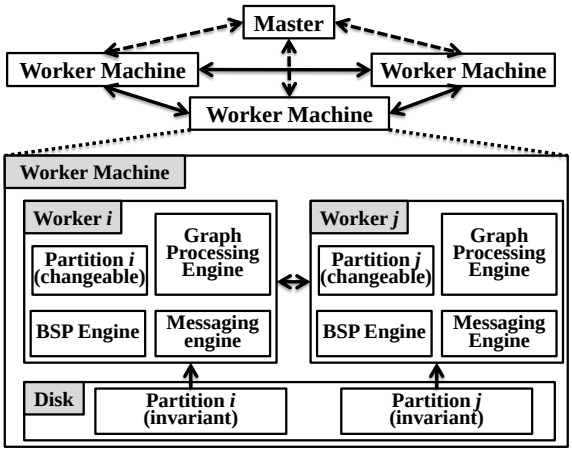


Figure 1: GraphMap System Architecture

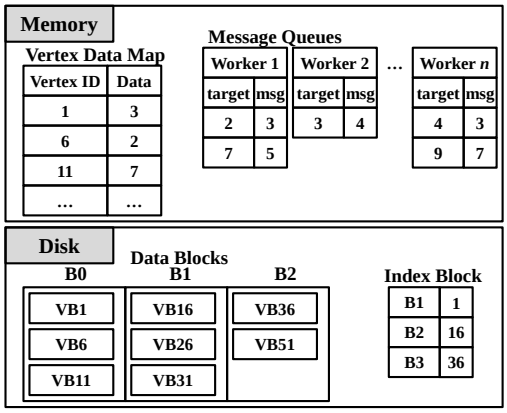


Figure 2: Graph Representation in GraphMap (single worker)

each iteration for batch transmission across workers.

3. LOCALITY-BASED DATA PLACEMENT

In this section, we introduce our locality-based storage structure for GraphMap. We provide an example to illustrate our in-memory and on-disk representation of graph partitions. In the next section we describe how GraphMap can benefit from the locality-optimized data partitions and data placements to speed up iterative graph computations.

We have mentioned earlier that in most iterative graph algorithms, only vertex data are mutable while edge data are invariant during the entire iterative computations. By cleanly separating graph data into *mutable* and *invariant* (or read-only) data, we can store most or all of the mutable data in memory and access invariant data from disk by minimizing non-sequential IOs. In contrast to existing Pregel-like distributed graph processing systems where each worker machine needs to hold not only the graph data but also its intermediate results and messages in memory, the GraphMap approach promotes the locality-aware integration of external storage with memory-intensive graph computations. The GraphMap design offers two significant advantages: (1) We can considerably reduce the memory requirement for running iterative graph applications by keeping only mutable data in memory and thus enable many more graphs and algorithms to run on GraphMap with respectable performance. (2) By designing a locality-aware data placement strategy such that vertex blocks belonging to the same partition will be stored contiguously on disk, we can speed up the access to the graph data stored on disk through sequential disk IOs for each iteration of the graph computations. For example, we keep all vertices and their values belonging to one partition in memory while keeping all the edges associated with these vertices and the edge property values, if any, on disk. In other words, in the context of vertex blocks in a partition, we maintain only anchor vertices and their values in memory and store all the corresponding vertex blocks contiguously on disk. In each iteration, for each *active* anchor vertex, we read its vertex block from disk and execute the graph computation in three steps as outlined in Section 2.3. For those graphs in which the number of edges is much larger than the number of vertices (e.g., more than two orders of magnitude larger in some real-world graphs), this design can considerably reduce the memory requirement for iterative graph computations even in the presence of long radius and skewed vertex degree distribution, because we do not require keeping edges in memory.

For anchor vertex values, which may be read and updated over the course of the iterative computations, such as the current shortest distance in SSSP and the current PageRank value, we maintain a mapping table that stores the vertex value for each anchor vertex in memory. Since each worker is in charge of one partition in GraphMap, only anchor vertices of its assigned partition are loaded in memory on each worker. For read-only edge data (i.e., vertex blocks of the anchor vertices), we need to carefully design its disk representation because otherwise it would be too costly to load vertex blocks from disk in each iteration. To tackle this challenge, we consider two types of access locality in graph algorithms: 1) edge access locality and 2) vertex access locality. By the *edge access locality*, we mean that all edges (out-edges, in-edges or bi-edges) of an anchor vertex are accessed together to update its vertex value. By using the vertex blocks as our building blocks for storage on disk, we can utilize the edge access locality because all edges of an anchor vertex are placed together. By the *vertex access locality*, we mean that the anchor vertices (and their vertex blocks) of a partition are accessed by the same worker in every iteration. To utilize the vertex access locality, for each partition, we store its all vertex blocks into contiguous disk blocks to utilize sequential disk accesses when we read the vertex blocks from disk in each iteration. In addition to the sequential disk accesses, in order to support efficient random accesses for reading the vertex block of a specific vertex, we store the vertex blocks in sorted order by their anchor vertex identifiers and create an index block that stores the start vertex identifier for each data block. In other words, we use range partitioning in which each data block stores vertex blocks of a specific range.

Fig. 2 shows an example of the in-memory and on-disk graph data representation for a partition held by a worker in GraphMap. All anchor vertices of the partition and their current vertex data are stored in a mapping table in memory. Since GraphMap employs the BSP model based on messaging, we keep an incoming message queue that stores all messages sent to this worker and an outgoing message queue for each worker in memory. On disk, eight vertex

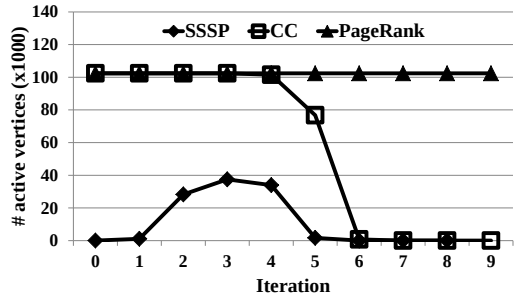


Figure 3: The number of active vertices per iteration

blocks are stored in three data blocks and one index block is created to store the start vertex for each data block.

4. LOCALITY-BASED OPTIMIZATIONS

In GraphMap, two levels of parallel computations can be provided: (1) Workers can process graph partitions in parallel. (2) Within each partition, we can compute multiple vertex blocks concurrently using multi-threading. By combining the graph parallel computations with our locality-based data placement, each parallel task can run independently with minimal non-sequential disk IOs. Since the vertex blocks belonging to the same partition are accessed by the same worker and stored in contiguous disk blocks, we can speed up graph computations in each iteration by combining parallelism with the sequential disk IOs for reading the vertex blocks.

It is interesting to note that different iterative graph algorithms may have different computation patterns and sequential disk accesses are not efficient for all types of graph computation patterns. For example, Fig. 3 shows the number of active vertices per iteration of a worker for three different iterative graph algorithms, Single-Source Shortest Path (SSSP), Connected Components (CC), and PageRank, on the orkut graph [18]. In PageRank, since all vertices are always active during all iterations and thus all vertex blocks of the anchor vertices are required in each iteration, our sequential disk accesses would be always efficient for reading the vertex blocks. On the other hand, in SSSP and CC, the number of active vertices is different for different iterations. When the number of active vertices is small, using the sequential accesses and reading the vertex blocks of all anchor vertices would be far from optimal because we do not need to evaluate the vertex blocks of most anchor vertices.

Based on this observation, we develop another locality-based optimization in GraphMap, which can dynamically choose between the sequential disk accesses and the random disk accesses based on the computation loads of the current iteration for each worker. This dynamic locality-based adaptation enables us not only to progressively filter out non-active vertices in each of the iterations for the iterative graph algorithms but also to avoid unnecessary and thus wasteful sequential disk IOs as early as possible. Recall that we store the vertex blocks of a partition in sorted order by their vertex identifiers and create an index block to support efficient random accesses. Given a query vertex, we need one (when the index block resides in memory) or two (when the index block is not in memory) disk accesses to read its vertex block. Specifically, in each iteration of a worker, if the number of active vertices is less than a system-defined

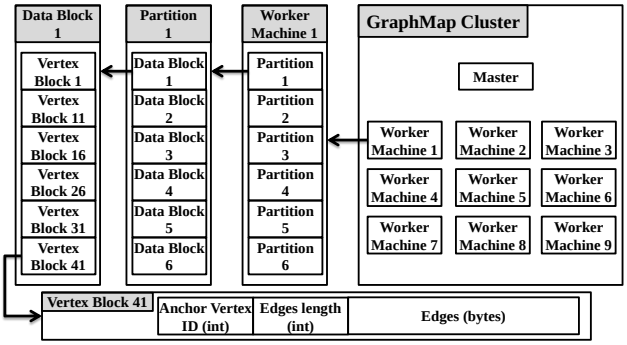


Figure 4: Hierarchical Disk Representation in GraphMap

(and user-modifiable) threshold θ, we choose the random disk accesses as our access method and read the index block from disk first, if not in memory. Based on the block information (i.e., start vertex for each data block) stored in the index block, we select and read only those data blocks that include the vertex block of active vertices. If the number of active vertices is equal to or larger than θ, we choose the sequential disk accesses and read the vertex blocks of all anchor vertices regardless of the current active vertices.

Because different clusters and different worker machines have different disk access performance, we also dynamically change the value of θ for each worker machine. Conceptually, by monitoring the current processing time of each random disk access and one full scan (i.e., sequential disk accesses for reading all vertex blocks), we calculate the break-even point, in which one full scan time is equal to the time of r random disk accesses, and use r as the value of θ.

The algorithm of updating the value of θ is formally defined as follows. Let θ_{iw}, s_{iw}, r_{iw}, and a_{iw} denote the threshold, one full scan time, total random disk access time, and the number of active vertices in iteration i on worker w respectively. If the full scan (or random disk access) is not used in iteration i on worker w, s_{iw} (or r_{iw}) is not defined (i.e., not a valid number). We use m and n, initially having 0 (zero), to denote the IDs of the last iteration where the full scan and random disk access was used respectively. θ_{0w} is the initial threshold on worker w and calculated empirically (e.g., random disk access time for 2% of all anchor vertices is similar to sequential disk access time for all anchor vertices on worker w). In iteration i ($i > 0$), before running the vertex program for each active vertex, we calculate the new threshold as follows:

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

In addition, when we store the vertex block of each anchor vertex in a data block, we bundle all out-edges (or in-edges or bi-edges) of the anchor vertex and store them together, as shown in Fig. 4, to utilize the edge access locality.

5. EXPERIMENTAL EVALUATION

In this section, we report the experimental evaluation results of the first prototype of GraphMap for various real-world graphs and iterative graph algorithms. We first explain the characteristics of graphs we used for our evaluation

and the experimental settings. We categorize the experimental results into four sets: 1) We show the execution time of various iterative graph algorithms in GraphMap and compare it with that of an Pregel-like system. 2) We present the effects of our dynamic access methods for various graph datasets. 3) We evaluate the scalability of GraphMap by increasing the number of workers in the cluster. 4) We compare GraphMap with other state-of-the-art graph systems.

5.1 Datasets and Graph Algorithms

We evaluate the performance of GraphMap using real-world graphs of different sizes and different characteristics for three types of iterative graph algorithms. Table 2 gives a summary of the datasets used for our evaluation. The first type of graph algorithms is represented by PageRank. In these algorithms, all vertices are always active during all iterations. The second type of graph algorithms is represented by Connected Components (CC), in which all vertices of the graph are active in the first iteration and then the number of active vertices starts to decrease as the computation progresses towards convergence. The third type of graph algorithms is represented by Single-Source Shortest Path (SSSP), where only the start vertex is active in the first iteration and the number of active vertices increases in early iterations and decreases in later iterations. We choose these three types of graph applications because they display different computation characteristics as we have shown earlier in Fig. 3.

Dataset	#vertices	#edges
hollywood-2011 [4]	2.2M	229M
orkut [18]	3.1M	224M
cit-Patents [15]	3.8M	16.5M
soc-LiveJournal1 [3]	4.8M	69M
uk-2005 [4]	39M	936M
twitter [12]	42M	1.5B

Table 2: Datasets

5.2 Setup and Implementation

We use a cluster of 21 machines (one master and 20 worker machines) on Emulab [27]: each node is equipped with 12GB RAM, one quad-core Intel Xeon E5530 processor, and two 7200 rpm SATA disks (500GB and 250GB), running CentOS 5.5. They are connected via a 1 GigE network. We run three workers on each worker machine and each worker is a JVM process with a maximum heap size of 3GB, unless otherwise noted. When we measure the computation time, we perform five runs under the same setting and show the *fastest* time to remove any possible bias posed by OS and/or network activity.

In order to compare with distributed memory-based graph systems, we use Apache Hama (Version 0.6.3), an open source implementation of Pregel. Another reason we choose Hama is that we implement the BSP engine and the messaging engine of GraphMap workers by adapting the BSP module and the messaging module of Apache Hama for the first prototype of GraphMap. This allows us to compare GraphMap with Hama's graph package more fairly.

To implement our vertex block-based data representation on disk, we utilize Apache HBase (Version 0.96), an open source wide column store (or two-dimensional key-value store), on top of Hadoop Distributed File System (HDFS) of Hadoop (Version 1.0.4). We choose HBase for the first prototype of GraphMap because it has several advantages. First, it provides a fault-tolerant way of storing graph data in the cloud. Since HBase utilizes the data replication of HDFS for fault-tolerance, GraphMap will continue to work even though some worker machines fail to perform correctly. Second, since HBase row keys are in sorted order and adjacent rows are usually stored in the same HDFS block (a single file in the file system), we can directly utilize HBase's range scans for implementing sequential disk accesses. Third, we can place all the vertex blocks of a partition in the same worker machine (called a region server) by using the HBase regions and renaming vertex identifiers. Specifically, we first pre-split the HBase table for the input graph into a set of regions in which each region is in charge of one hash partition. Next, we rename each vertex identifier by adding its partition identifier as a prefix of its new vertex identifier, such as "11-341" in which "11" and "341" represent the partition identifier and the original vertex identifier respectively. Thus all vertex blocks of a partition are stored in the same region. In other words, our hash partitioning is implemented by renaming vertex identifiers and using the pre-split regions and our range partitioning on each partition is implemented by HBase, which stores rows in sorted order by their identifier. Fourth, to implement our edge access locality-based approach, we bundle all edges of a vertex block and store the bundled data in a single column because the data is stored together on disk. Another possible technique is to use a column for each edge of a vertex block using the same column family because all column family members are stored together on disk by HBase. However, to eliminate the overhead of handling many columns, we implement the former technique for our edge access locality-based approach.

5.3 Iterative Graph Computations

We first compare the total execution time of GraphMap with that of Hama's graph package for the three iterative graph algorithms. Table 3 shows the results for different real-world graphs. For SSSP, we report the execution time when we choose the vertex having the largest number of out-edges as the start vertex except the uk-2005 graph in which we choose the vertex having the third largest number of out-edges because only about 0.01% vertices are reachable from each of the top two vertices. For PageRank, we report the execution time of 10 iterations. The result clearly shows that GraphMap outperforms Hama significantly on all datasets for all algorithms (PageRank, SSSP, and CC). For large graphs such as uk-2005 and twitter, GraphMap considerably reduces the memory requirement for iterative graph algorithms. However, Hama fails for all algorithms because it needs not only to load all vertices and edges of the input graph but also to hold all intermediate results and messages in memory. Thus Hama cannot handle those large graphs, such as uk-2005 (936M edges) and twitter (1.5B edges) datasets, where the number of edges is approaching or exceeding one billion.

GraphMap not only reduces the memory requirement for iterative graph algorithms but also significantly improves the iterative graph computation performance compared to Hama. For SSSP, CC, and PageRank, GraphMap is 2x-6x, 1.8x-4.5x, and 1.7x-2.6x faster than Hama respectively. Given that both GraphMap and Hama use the same messaging and BSP engines, the difference in terms of the number of messages in Table 3 is due to the effect of the combiner.

Dataset	total execution time (sec)						the number of messages					
	SSSP		CC		PageRank		SSSP		CC		PageRank	
	Hama	Graph Map	Hama	Graph Map	Hama	Graph Map	Hama	Graph Map	Hama	Graph Map	Hama	Graph Map
hollywood-2011	108.776	18.347	177.854	39.365	268.474	111.466	229M	80M	1.2B	348M	2.1B	2.1B
orkut	108.744	21.345	195.841	54.383	286.054	111.46	224M	123M	1.3B	548M	2.0B	2.0B
cit-Patents	27.693	12.337	24.646	12.335	30.688	18.353	219K	212K	17M	15M	149M	149M
soc-LiveJournal1	48.697	18.346	60.734	33.357	75.76	39.369	68M	51M	359M	243M	616M	616M
uk-2005	Fail	156.49	Fail	706.329	Fail	573.964	Fail	450M	Fail	5.2B	Fail	8.3B
twitter	Fail	150.486	Fail	303.653	Fail	1492.966	Fail	585M	Fail	1.5B	Fail	13.2B

Table 3: Total execution time and the number of messages

Dataset	total execution time (sec)					
	SSSP			CC		
	GraphMap-Sequential	GraphMap-Random	GraphMap-Dynamic	GraphMap-Sequential	GraphMap-Random	GraphMap-Dynamic
hollywood-2011	24.354	27.345	18.347	45.383	81.405	39.365
orkut	27.35	33.356	21.345	57.384	126.455	54.383
cit-Patents	15.34	12.34	12.337	18.34	12.332	12.335
soc-LiveJournal1	24.348	36.357	18.346	36.361	120.447	33.357
uk-2005	1225.637	225.555	156.49	2033.522	2898.407	706.329
twitter	252.598	267.622	150.486	712.085	721.089	303.653

Table 4: Effects of dynamic access methods

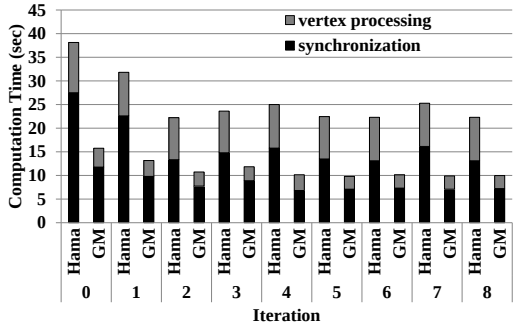


Figure 5: Comparison with Hama (PageRank on orkut)

The GraphMap's messages are counted after the combiner is executed and, on the other hand, Hama reports only the numbers, which are measured before the combiner is executed. For PageRank, the number of messages is almost the same for both systems because no combiner is used.

To provide in-depth analysis, we further divide the total execution time into the vertex processing time and the synchronization (global barrier) time per iteration as shown in Fig. 5. The synchronization time includes not only the message transfer time among workers but also the waiting time until the other workers complete their processing in the current iteration. The vertex processing time includes the vertex update time (the core part defined in the vertex-centric program), received message processing time, and message queuing time for messages to be sent during the next synchronization. For GraphMap, it also includes the disk (HBase) access time. It is interesting to note that, even though Hama is the in-memory system, its vertex processing time is much slower than that of GraphMap, which accesses HBase to read the vertex blocks stored on disk, for all iterations. This result shows that a carefully designed framework based on the access locality of iterative graph algorithms can be competitive with and in some cases outperform the in-memory framework in terms of the total execution time even though it requires disk IOs in each iteration for reading a part of graph data.

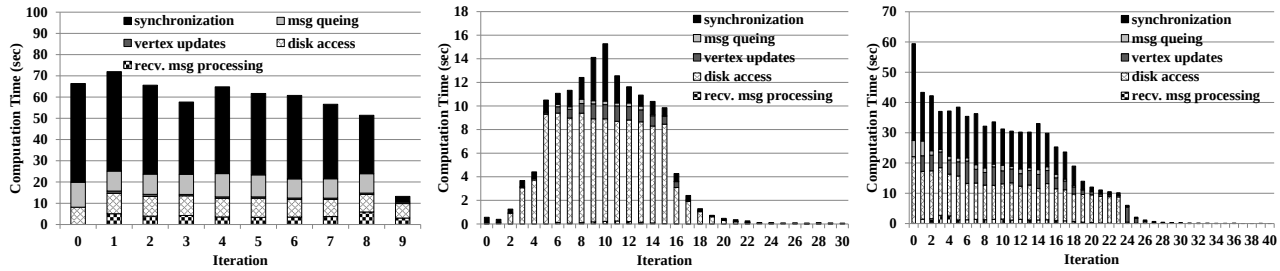
We split the vertex processing time of GraphMap for further details as shown in Fig. 6. We could not find measurement points to gather such numbers for Hama. For PageRank, all iterations have similar results except the first iteration, in which there is no received message, and the last iteration, in which no message is sent. Note that the vertex update time, the core component for the vertex-centric model, is only a small part in the total execution time. For SSSP, the disk IOs from iteration 5 to iteration 15 are almost the same because GraphMap chooses the sequential accesses based on our dynamic access methods. From iteration 16 to iteration 30, the disk IOs continue to drop until the algorithm converges thanks to our dynamic locality-based optimization.

5.4 Effects of Dynamic Access Methods

Table 4 shows the effects of the dynamic access methods of GraphMap, compared to two baseline approaches that use only sequential accesses or only random accesses in all iterations for SSSP and CC. For PageRank, GraphMap always chooses the sequential accesses because all vertices are active during all iterations. The experimental results clearly show that GraphMap with the dynamic access methods offers the best performance because it chooses the optimal access method for each worker and in each iteration based on the current computation loads, such as the ratio of active vertices to total vertices in a partition.

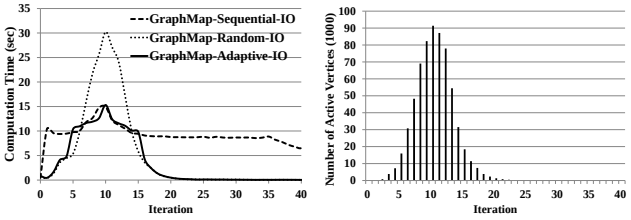
Table 4 also shows that for the cit-Patents graph dataset, GraphMap always chooses the random accesses because only 3.3% vertices are reachable from the start vertex and thus the number of active vertices in each iteration is very small. For SSSP on the uk-2005 graph, the baseline approach using only sequential accesses is 8x slower than GraphMap. This is because it takes 198 iterations for SSSP on the uk-2005 graph to converge and the baseline approach always runs with the sequential disk accesses even though the number of active vertices is very small in most iterations.

Fig. 7 shows the effects of GraphMap's dynamic access methods per iteration, for the first 40 iterations, on a single worker using the uk-2005 graph. The result shows that GraphMap chooses the optimal access method in most of the iterations based on the number of active vertices. It is

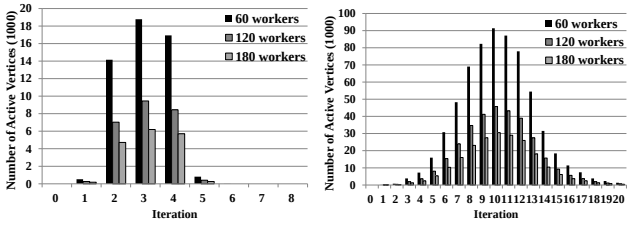


(a) PageRank (uk-2005) (b) SSSP (uk-2005) (c) CC (uk-2005)

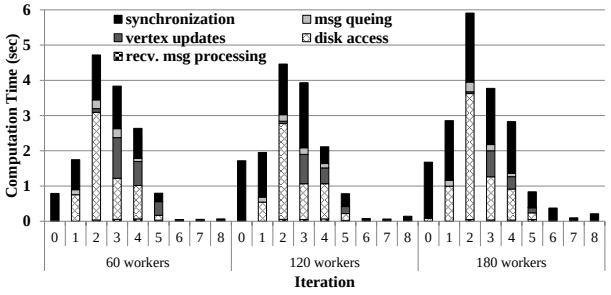
Figure 6: Breakdown of execution time per iteration (single worker)



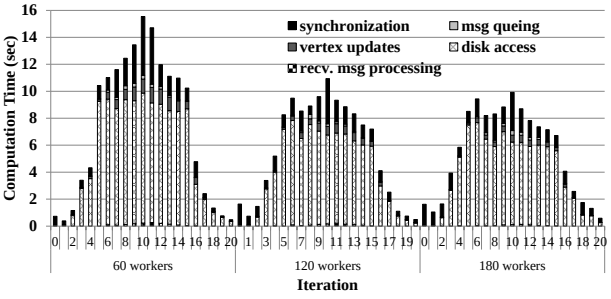
(a) Computation Time (b) #active vertices



(a) #active vertices (orkut) (b) #active vertices (uk-2005)



(c) computation time (orkut)



(d) computation time (uk-2005)

Figure 7: Effects of dynamic access methods

Figure 8: Scalability (varying #workers)

interesting to note that GraphMap chooses the sequential accesses in iteration 5 and 15 even though random accesses are faster. This indicates that GraphMap's performance can be improved further by fine-tuning the threshold θ value. In these experiments, θ is empirically set to 2% of all vertices in each partition.

5.5 Scalability

To evaluate the scalability of GraphMap, we report the SSSP execution results with varying numbers of workers from 60 to 180 using the same cluster, as shown in Table 5. For this set of experiments, we use 1GB as the maximum heap size of each worker for both GraphMap and Hama. The results show that GraphMap needs fewer workers than Hama to run the same graph because it reduces the memory requirement of graph computations. If we run more workers, each worker handles fewer active vertices proportionally, as shown in Fig. 8(a) and Fig. 8(b), because the worker is in charge of a smaller partition. However, by increasing the number of workers, the cost of inter-worker communication will increase significantly, which increases the total computation time even with a smaller number of active vertices on each worker. As shown in Fig. 8(c) and Fig. 8(d), the vertex update time reduces as we increase the number of workers but at the cost of increased synchronization time for coordinating more workers.

5.6 Comparison with Existing Systems

In this section we compare GraphMap with existing representative in-memory graph systems, including GraphX, GraphLab (PowerGraph), Giraph, Giraph++ (with hash partitioning), and Hama, in Table 6. We compare the performance of PageRank and CC on twitter and uk-2005/uk-2007 datasets. Given that GraphX requires Spark and larger memory to run, we extract the performance results of GraphX, GraphLab, and Giraph from [8], annotated with their respective system configurations for the same graph datasets. The results of Giraph++ are extracted from [28]. We offer a number of observations.

First, the testbeds for other systems have larger RAM and a larger number of CPU cores. For example, GraphX, GraphLab, and Giraph run on a cluster with 1,088GB RAM and 128 cores while GraphMap runs on a cluster with 256GB RAM and 84 cores. For CC on the twitter dataset, GraphMap shows comparable performance to these state-of-the-art in-memory graph systems, even though GraphMap uses much less computing resources (less than two-thirds of cores and less than one-fourth of RAM). For example, GraphMap is only 20% slower than GraphX with

System	Setting	CC (sec)		PageRank (sec per iteration)		Type
		twitter	uk-2005(*uk-2007)	twitter	uk-2005(*uk-2007)	
GraphMap on Hadoop	21 nodes (21x4=84 cores, 21x12=252GB RAM)	304	706	149	57	Out-of-core
Hama on Hadoop	21 nodes (21x4=84 cores 21x12=252GB RAM)	Fail	Fail	Fail	Fail	In-memory
GraphX on Spark	16 nodes (16x8=128 cores 16x68=1088GB RAM)	251	800*	21	23*	In-memory
GraphLab 2.2 (PowerGraph)	16 nodes (16x8=128 cores 16x68=1088GB RAM)	244	714*	12	42*	In-memory
Giraph 1.1 on Hadoop	16 nodes (16x8=128 cores 16x68=1088GB RAM)	200	Fail*	30	62*	In-memory
Giraph++ on Hadoop	10 nodes (10x8=80 cores 10x32=320GB RAM)	No result reported	723	No result reported	89	In-memory

Table 6: System Comparison

Total execution time (sec)				
		#Workers		
Dataset	Framework	60	120	180
hollywood-2011	Hama	Fail	114.801	114.926
	GraphMap	18.352	21.351	27.356
orkut	Hama	Fail	99.784	102.883
	GraphMap	21.36	24.359	30.355
cit-Patents	Hama	27.678	39.738	54.799
	GraphMap	9.348	15.369	18.348
soc-LiveJournal1	Hama	45.683	54.736	75.837
	GraphMap	18.357	21.368	27.356
uk-2005	Hama	Fail	Fail	415.239
	GraphMap	159.517	135.486	138.481
twitter	Hama	Fail	Fail	Fail
	GraphMap	Fail	141.485	126.468

Table 5: Scalability (SSSP)

a much more powerful cluster. GraphMap is even faster than Giraph++ on the uk-2005 dataset. Through our dynamic access methods, GraphMap achieves competitive performance for CC even though it accesses the disk for each iteration. For PageRank, GraphMap is slower than GraphX, GraphLab, and Giraph because it reads all the edge data from disk in each iteration with only two-thirds of CPU cores. This comparison demonstrates the effectiveness of the GraphMap approach to iterative computations of large graphs.

6. RELATED WORK

We classify existing systems for iterative graph algorithms into two categories. The first category is the distributed solution that runs the iterative graph computations using a cluster of commodity machines, represented by distributed memory-based systems like Pregel. The second category of graph systems is the disk-based solution on a single machine, represented by GraphChi [13] and X-Stream [20].

Distributed memory-based systems typically require to load the whole input graph in memory and to have sufficient memory to store all intermediate results and all messages in order to run iterative graph algorithms [17, 7, 16, 22, 8]. Apache Hama and Giraph are the popular open-source implementations of Pregel. GraphX [8] and Pregelix [5] implement a graph processing engine on top of a general-purpose distributed dataflow framework. They represent the graph data as tables and then use database-style query execution techniques to run iterative graph algorithms.

Unlike Pregel-like distributed graph systems, GraphLab [16] and PowerGraph [7] replicate a set of vertices and edges using a concept of ghosts and mirrors respectively. GraphLab is based on an asynchronous model of computations and PowerGraph can run vertex-centric programs both synchronously and asynchronously. Trinity [22] handles both online and offline graph computations using a distributed memory cloud (an in-memory key-value store). In addition, several techniques for improving the distributed memory-based graph systems have been explored, such as dynamic workload balancing [21, 11] and graph-centric view [25].

Disk-based systems focus on improving the performance of iterative computations on a single machine [13, 20, 9, 28, 30]. GraphChi [13] is based on the vertex-centric model. It improves disk access efficiency by dividing a large graph into small shards, and uses a parallel sliding window-based method to access graph data on disk. Unlike the vertex-centric model, X-Stream [20] proposes an edge-centric model to utilize sequential disk accesses. It partitions a large graph into multiple streaming partitions and loads a streaming partition in main memory to avoid random disk accesses to vertices. PathGraph [28] proposes a path-centric model to improve the memory and disk access locality. TurboGraph [9] and FlashGraph [30], based on SSDs, improve the performance by exploiting I/O parallelism and overlapping computations with I/O.

Even though some systems, including Giraph and Pregelix, provide out-of-core capabilities to utilize external memory for handling large graphs, they typically focus only on reducing the memory requirement, not the performance of iterative graph computations. To the best of our knowledge, GraphMap is the first distributed graph processing system, which incorporates external storage into the system design for efficient processing of iterative graph algorithms in a cluster of compute nodes.

7. CONCLUSION

We have presented GraphMap, a distributed iterative graph computation framework, which effectively utilizes secondary storage to maximize access locality and speed up distributed iterative graph computations. This paper makes three unique contributions. First, we advocate a clean separation of those data states that are mutable during iterative computations from those that are read-only in all iterations. This allows us to develop locality-optimized data placement and data partitioning methods to maximize sequential accesses and minimize random accesses. Second, we devise a two-level graph partitioning algorithm to enable balanced workloads and locality-optimized data placement. In addition, we propose a suite of locality-based optimizations to improve computation efficiency. We evaluate GraphMap through extensive experiments on several real graphs and show that GraphMap outperforms an existing distributed memory-based system for various iterative

graph algorithms.

8. ACKNOWLEDGMENTS

This work was performed under the partial support by the National Science Foundation under grants IIS-0905493, CNS-1115375, IIP-1230740, and a grant from Intel ISTC on Cloud Computing.

9. REFERENCES

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

[2] Apache Hama. *https://hama.apache.org/*.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. 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*, KDD '06, pages 44–54, New York, NY, USA, 2006. ACM.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[5] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(Ger) Graph Analytics on a Dataflow Engine. *Proc. VLDB Endow.*, 8(2):161–172, Oct. 2014.

[6] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm. In *PARALLEL PROCESSING FOR SCIENTIFIC COMPUTING. SIAM*, 1997.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[8] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association.

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

[10] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[11] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[12] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM.

[13] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[14] K. Lee and L. Liu. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *Proc. VLDB Endow.*, 6(14):1894–1905, Sept. 2013.

[15] J. Leskovec, J. Kleinberg, and C. Faloutsos. 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, pages 177–187, New York, NY, USA, 2005. ACM.

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

[17] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[18] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[19] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded Asynchronous Graph Traversal for In-Memory and Semi-External Memory. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[20] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[21] S. Salihoglu and J. Widom. GPS: A Graph Processing System. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[22] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[23] I. Stanton and G. Kliot. Streaming Graph Partitioning for Large Distributed Graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[24] S. Suri and S. Vassilvitskii. Counting Triangles and the Curse of the Last Reducer. In *Proceedings of the 20th*

International Conference on World Wide Web, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.

[25] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From" Think Like a Vertex" to" Think Like a Graph. *Proceedings of the VLDB Endowment*, 7(3), 2013.

[26] L. G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, pages 255–270, New York, NY, USA, 2002. ACM.

[28] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast Iterative Graph Computation: A Path Centric Approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 401–412, Piscataway, NJ, USA, 2014. IEEE Press.

[29] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

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