GraphM: An Efficient Storage System for High Throughput of Concurrent Graph Processing

Jin Zhao
Huazhong University of Science and
Technology, China*
zjin@hust.edu.cn

Yu Zhang
Huazhong University of Science and
Technology, China*
zhyu@hust.edu.cn

Xiaofei Liao
Huazhong University of Science and
Technology, China*
xfliao@hust.edu.cn

Ligang He
University of Warwick, United
Kingdom
ligang.he@warwick.ac.uk

Bingsheng He
National University of Singapore,
Singapore
hebs@comp.nus.edu.sg

Hai Jin
Huazhong University of Science and
Technology, China*
hjin@hust.edu.cn

Haikun Liu
Huazhong University of Science and
Technology, China*
hkliu@hust.edu.cn

Yicheng Chen
Huazhong University of Science and
Technology, China*
yichengchen@hust.edu.cn

ABSTRACT

With the rapidly growing demand of graph processing in the real world, a large number of iterative graph processing jobs run concurrently on the same underlying graph. However, the storage engines of existing graph processing frameworks are mainly designed for running an individual job. Our studies show that they are inefficient when running concurrent jobs due to the redundant data storage and access overhead. To cope with this issue, we develop an efficient storage system, called GraphM. It can be integrated into the existing graph processing systems to efficiently support concurrent iterative graph processing jobs for higher throughput by fully exploiting the similarities of the data accesses between these concurrent jobs. GraphM regularizes the traversing order of the graph partitions for concurrent graph processing jobs by streaming the partitions into the main memory and the *Last-Level Cache* (LLC) in a common order, and then processes the related jobs concurrently in a novel fine-grained synchronization. In this way, the concurrent jobs share the same graph structure data in the LLC/memory and also the data accesses to the graph, so as to amortize the storage consumption and the data access overhead. To demonstrate the efficiency of GraphM, we plug it into state-of-the-art graph processing systems, including GridGraph, GraphChi, PowerGraph, and Chaos. Experiments results show that GraphM improves the throughput by 1.73~13 times.

CCS CONCEPTS

• **Computer systems organization** → *Multicore architectures*; • **Information systems** → *Hierarchical storage management*; • **Computing methodologies** → *Parallel computing methodologies*.

KEYWORDS

Iterative graph processing; concurrent jobs; storage system; data access similarity

* Jin Zhao, Yu Zhang (Corresponding author), Xiaofei Liao, Hai Jin, Haikun Liu, and Yicheng Chen are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

1 INTRODUCTION

A massive number of concurrent iterative graph processing jobs are often executed on the same cloud platform, e.g., the Facebook Cloud [2] and the Huawei Cloud [3], to analyze their daily graph data for different products and services. For example, Facebook [2] adopts Apache Giraph [16] to support many different iterative graph algorithms (e.g., the variants of PageRank [29] and label propagation [8]) that are used by various applications running on the same underlying graph, e.g., social networks. However, existing solutions [13, 20–22, 25, 27, 37, 38] mainly focus on optimizing the processing of individual graph analysis jobs. In order to achieve the efficient execution of concurrent iterative graph processing jobs, the following two key challenges need to be addressed.

First, there is much unnecessary data access cost when the concurrent jobs running on the same underlying graph do not take into account the similarities of their data accesses. It eventually induces low throughput. Specifically, concurrent iterative graph

Jin Zhao, Yu Zhang, Xiaofei Liao, et al.

(a) Existing systems   (b) Existing systems integrated with GraphM
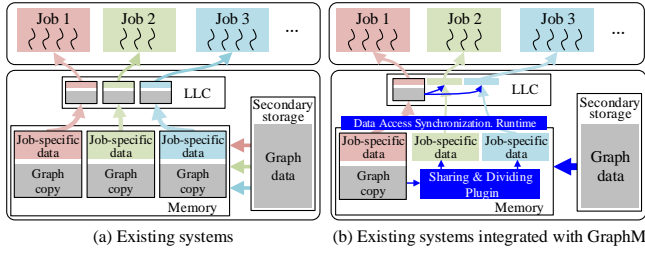
Figure 1: Execution of concurrent iterative graph processing jobs on (a) existing graph processing systems and (b) the ones integrated with GraphM

processing jobs usually traverse the same graph structure repeatedly and a large proportion of the graph data accessed by them is actually the same. However, as shown in Figure 1 (a), with the graph storage engines highly-coupled with existing graph processing frameworks [13, 20, 22, 25], multiple copies of the shared graph data are maintained in the *Last-Level Cache* (LLC)/memory and are individually accessed by the concurrently running jobs. It results in inefficient use of data access channels and storage resources (e.g., the LLC/memory). There is a clear trend of running more and more iterative graph processing applications on the same platform. For example, DiDi [1] carries out more than 9 billion path planing [7] daily in 2017. The highly redundant overhead discussed above incurs low throughput of concurrent iterative graph processing jobs.

Second, diverse graph processing systems, which are highly coupled with their own storage engines, are developed, because it is important to employ suitable graph processing schemes for better performance according to their own requirements [26]. It is desired to decouple the graph storage system from graph processing to allow different graph processing systems to share a single optimized graph storage system, i.e., one storage system for all. Then, an optimized storage system can integrate with these graph processing engines to enable the concurrent and efficient execution of existing iterative graph processing applications while imposing little programming burden on the users.

To address these challenges, a novel and efficient storage system, called **GraphM**, is proposed in this paper. It is a lightweight runtime system which can be run in any existing graph processing system and enables the system to support the concurrent execution of iterative graph processing jobs. In GraphM, we design a novel *Share-Synchronize* mechanism to fully exploit the similarities in data access between concurrently running jobs. The graph structure data is decoupled from the job-specific data to be shared by multiple graph processing jobs, while only the job-specific data is maintained for each individual job. Then, GraphM regularizes the traversal paths of the graph partitions for the concurrent jobs by streaming the partitions into the LLC/memory in a common order and concurrently processing multiple jobs related to a common graph partition in novel fine-grained synchronization. Then, there is only a single copy of the graph structure data in the LLC/memory for multiple concurrent jobs, and the data access cost is amortized by them. More importantly, the existing graph processing systems residing above GraphM can still run with their own execution model, because the traversal path of the systems are regularized transparently in each iteration by GraphM. The idea is illustrated in Figure 1 (b), where only one copy of the common graph (rather
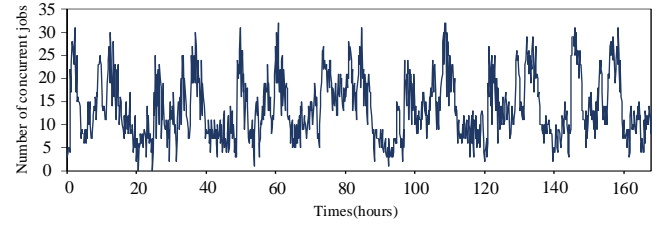

Figure 2: Number of jobs traced on a social network

than several copies in the existing systems) is maintained to serve multiple concurrent jobs and the concurrent jobs can share the storage of the common graph and the data access to it. When writing the graph processing applications, the programmers only need to call a few APIs provided by GraphM to achieve higher performance for the concurrent execution of these applications. Moreover, in order to further improve the throughput, a scheduling strategy is designed in GraphM to specify the loading order of graph partitions to maximize the utilization ratio of the graph partitions loaded into the main memory.

This paper has the following main contributions:

- The redundant data access overhead is revealed when existing graph processing system handles multiple concurrent jobs over a common graph, and the similarity between data accesses of the jobs is investigated.
- A novel and efficient storage system is developed to improve the throughput of existing graph processing systems for handling concurrent jobs while little programming burden is imposed on programmers.
- An efficient scheduling strategy is developed to fully exploit the similarities among the concurrent jobs.
- We integrate GraphM into existing popular graph processing systems, i.e., GridGraph [50], GraphChi [24], PowerGraph [14], and Chaos [32], and conduct extensive experiments. The results show that GraphM improves their performance by 1.73~13 times.

The rest is organized as follows. Section 2 discusses our motivation. GraphM is presented in Section 3 and the scheduling strategy is described in Section 4, followed by experimental evaluation in Section 5. The related work is surveyed in Section 6. This paper is concluded in Section 7.

2 BACKGROUND AND MOTIVATION

Most existing systems [6, 13, 20, 22, 25] are inherently designed to optimize the performance of individual iterative graph processing job. In fact, however, with the increasing demand for graph analytics, various iterative graph algorithms are often run concurrently on a common platform. Figure 2 shows the variation of the number of concurrent graph processing jobs within one week traced from a real Chinese social network. We find that more than 30 jobs are executed concurrently at the peak time, and the average number of concurrent jobs is about 16. In particular, these concurrent jobs are often handled on a common underlying graph. The data accesses related to the graph occupy a large proportion of their memory overhead during the execution, which varies from 71% to 83% for different datasets [43].
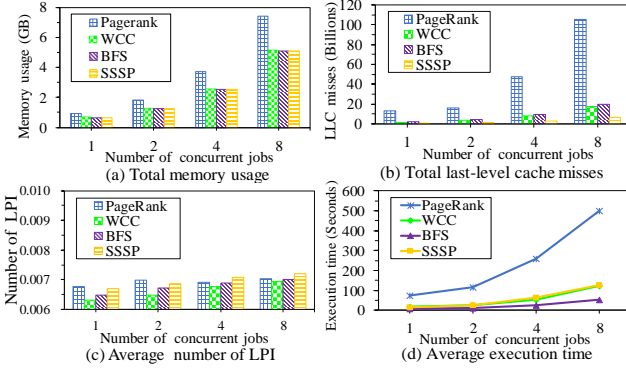
Figure 3: Performance evaluation of concurrent iterative graph processing jobs executed on GridGraph

2.1 Redundant Data Access Overhead

With the storage engines highly-coupled with existing graph processing systems [13, 20, 22, 25], the common graph is individually accessed by concurrent iterative graph processing jobs. It generates excessive unnecessary overhead in storage resource and data access channel, and thus significantly increases the data access cost. This eventually results in low system throughput because the data access cost usually dominates the total execution time of iterative graph processing [36]. To demonstrate it, we evaluated the performance of concurrent jobs on GridGraph [50] over Twitter [23], where the platform is the same as that introduced in Section 5.

We observe that more data access cost is generated with the increase of the number of concurrent jobs. It is because that multiple copies of the common underlying graph are loaded into the storage by its storage engine for the concurrent jobs. For example, as depicted in Figure 3(a), the total amount of memory usage for the processing of each partition significantly increases due to redundant memory usage as the number of jobs increases. Figure 3(b) describes the total number of LLC misses for different number of concurrent graph processing jobs over GridGraph [50], which represents the size of the graph data loaded into the LLC. It can be seen that much redundant graph data is also swapped into the LLC.

In addition, with more concurrent jobs, more serious contention for storage resources and data access channels occurs in a resource-limited machine, thereby causing more page faults, LLC misses, etc. As shown in Figure 3(c), the average number of *LLC misses per Instruction* (LPI) increases when more concurrent jobs are executed over GridGraph [50], due to the intense cache interference caused by the fact that multiple copies of the same graph partition are being individually loaded into the LLC. For example, when there are eight concurrent jobs, the average number of the LPI of these jobs increases by about 10% comparing with that of one job, because the graph data required for the execution of the instructions of different jobs is usually the same for these concurrent jobs. It exacerbates the above challenges. To show the impact of resource contention, Figure 3(d) shows the average execution time of each job as the number of jobs increases. It can be observed that the execution time of each job significantly increases as the number of jobs increases.

2.2 Our Motivation

Figure 4(a) depicts the percentage of the graphs that are shared by different number of concurrent jobs traced from a social network,
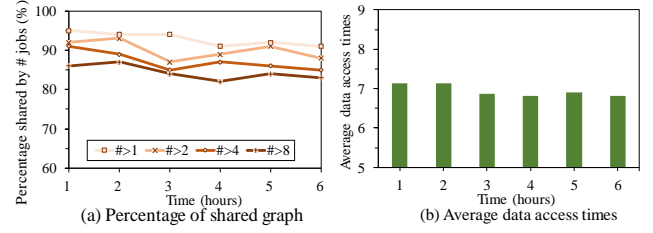

Figure 4: Information traced on the social network

and Figure 4(b) indicates the average number of accessed times of the graph partitions which have been repeatedly accessed by different jobs in each time period (one hour). It can be observed from Figure 4 that there are strong similarities between the data accesses of many concurrent jobs, because the same graph is repeatedly traversed by them.

Observation 1. Most proportion of the same graph is shared by multiple concurrent jobs during the traversals, which is called the spatial similarity. As shown in Figure 4(a), more than 82% of the same underlying graph is concurrently processed by the concurrent jobs during the traversals. Unfortunately, in most existing systems, the intersection of the graph data handled by the concurrent jobs is not shared in the LLC/memory, and it is accessed by these jobs along different graph paths individually, which results in a large amount of redundant data access overhead. Ideally, it only needs to maintain a single copy of the same graph data in the LLC/memory to serve the concurrent jobs in each traversal.

Observation 2. The same graph data may be accessed by different concurrent jobs over a period of time, which is called the temporal similarity. In detail, since the same underlying graph is individually handled by the concurrent jobs, the shared graph data may be frequently accessed by multiple jobs within their repeated traversals (about 7 times on average as shown in Figure 4(b)). However, the existing systems are not aware of this temporal similarity, so that the graph data frequently accessed by different jobs may be swapped out of the LLC/memory, which leads to the rise of the data access cost. Therefore, the accesses to the shared graph data for the concurrent jobs should be consolidated so that the same graph data is only loaded into the LLC/memory once to be handled by the concurrent jobs in each traversal for once.

The strong spatial and temporal similarities motivate us to design a storage system, which can integrate with existing graph processing engines to manage the data accesses of concurrent iterative graph processing jobs for higher throughput while imposing little programming burden on the users.

3 OVERVIEW OF GRAPHM

The storage system developed in this work is called GraphM. It is designed as a runtime system and can be plugged into the existing graph processing systems to manage the data access to the shared graph stored in the storage resources (e.g., the memory and disk). The key idea of our system is to enable concurrent jobs to share a single copy of the common graph and traverse the graph synchronously along the same graph path. To achieve this goal, an efficient *Share-Synchronize* mechanism and the corresponding pre-processing method are proposed in GraphM. Further, several lightweight APIs are provided for the existing graph processing systems to use.
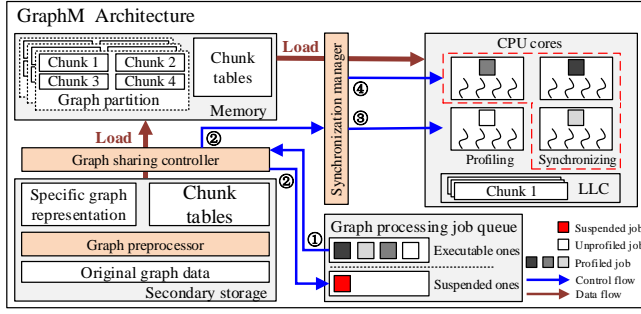
Figure 5: System architecture of GraphM

3.1 System Architecture

Generally, the data needed by an iterative graph processing job is composed of the graph structure data (i.e., the graph represented by $G=(V, E, W)$), job-specific data (e.g., ranking scores for PageRank [29], and the component ID for Connected Components [19]), marked as S. During the execution, each job needs to update its S through traversing the graph structure data until the calculated results converge. Specifically, in existing systems [12, 24, 44, 50], G and S are stored separately. GraphM enables G to be shared by the concurrent jobs, thereby fully exploiting the strong spatial and temporal similarities between these jobs. Figure 5 shows the architecture of GraphM. It consists of three main components: graph preprocessor, graph sharing controller, and synchronization manager, which are overviewed in the following subsections.

Graph Preprocessor. The graph formats and the preprocessing methods can be different for various graph processing systems. Thus, before the graph processing, the original graph stored in GraphM needs to be converted to the graph representation format specific to the graph processing system (which needs to handle this graph) using user defined function *Convert()*. For example, the original graph data is converted to the grid format for GridGraph [50], the shard format for GraphChi [24], the CSR/CSC format for PowerGraph [14], and the edge list format for Chaos [32]. After that, as existing graph processing systems [12, 24, 44, 50], the graph is divided into partitions for parallel processing and the operations of the concurrent jobs are still performed on the specific graph representation of the related system. Meanwhile, the graph structure partitions are further logically divided and labelled as a series of chunks according to the common traversing order of the graph of the jobs for the purpose of fine-grained synchronization as the following described. In addition, it can exploit the cache locality because the chunks can be fit in the LLC. When dividing a partition, a *chunk_table* array is generated to describe the key information of each logical chunk for the purpose of regular accessing of the graph partition shared by multiple jobs, where the specific graph representation is not modified.

Graph Sharing Controller. After graph preprocessing, the specific graph structure data needs to be loaded into the memory to serve concurrent jobs. This functional module is used to assign the loading order and also load the graph structure partitions, which will be shared by concurrent jobs. The module is designed as a thin API to be plugged into the existing graph processing systems. The API can be expressed as: $P_j^i \leftarrow Sharing(G, Load())$. G is the name of the graph to be loaded and is used to identify the range of shared memory

Table 1: GraphM Programming Interface

APIs	Description
$Init()$	Initialization of GraphM
$GetActiveVertices()$	Get active vertices in each iteration
$Sharing()$	Load the shared graph data
$Start()/Barrier()$	Notify GraphM to start or end fine-grained synchronization

which contains the shared graph structure partition. *Load()* is the original load operation of the graph processing system integrated with GraphM for the loading of graph data, and P_j^i denotes a loaded graph structure partition P^i shared by the jth job. When the jobs need to be concurrently executed (the step ①), the loaded graph structure data is only shared by active jobs, while inactive jobs are suspended and wait for their active graph vertices/edges to be loaded into the memory (the step ②). In addition, the mutations and updates of the shared graph structure data are isolated among concurrent jobs to ensure the correctness of the processing. In this way, only one copy of the shared graph structure data needs to be loaded and maintained in the memory to serve concurrent jobs. Thus, the redundant cost of the memory resource and the amount of disk data transfers are reduced.

Synchronization Manager. When the graph structure partition is shared by concurrent jobs, it is individually accessed by them in a consistent logical order according to its programming model. However, since some jobs may skip the inactive vertices for them and the computational complexity of the processing of the streamed data is usually different for various jobs, these jobs may process the shared graph partitions in different orders. Hence, the shared graph data is irregularly streamed into the LLC by concurrent jobs, resulting in unnecessary data access cost. To solve this problem, we use a novel and efficient fine-grained synchronization way to fully exploit the temporal similarity between these jobs.

This module enables the chunks of the shared graph data to be regularly streamed into the LLC by traversing the same graph path in fine-grained synchronization. In detail, each job needs to be profiled to determine the computational load of each chunk before each iteration (the step ③). The computing resources are then unevenly allocated to the jobs for their concurrent execution based on the skewed computational load of these jobs (the step ④), so as to synchronize the graph traversals with low cost. By such means, each chunk typically only needs to be loaded into the LLC once and be reused by concurrent jobs in each iteration. Thus, it significantly reduces the data access cost by fully exploiting the similarities of these jobs.

Programming APIs. To invoke GraphM in graph analysis programs, the user only needs to insert our APIs shown in Table 1 into existing graph processing systems. Note that it does not need to change the graph applications above these graph processing systems. In detail, $Init()$ is used to initialize GraphM by preprocessing the graph as described in Section 3. *Sharing()* function is inserted in existing graph processing systems to replace the original data load operation for the efficient load of the shared graph data. Note that the parameter in the function *Sharing()* is various for different graph processing systems, e.g., the parameter is the function $Load()$ for GridGraph and is the function $LoadSubgraph()$ for GraphChi.

(a) Pseudocode of GridGraph   (b) Pseudocode of GridGraph integrated with GraphM
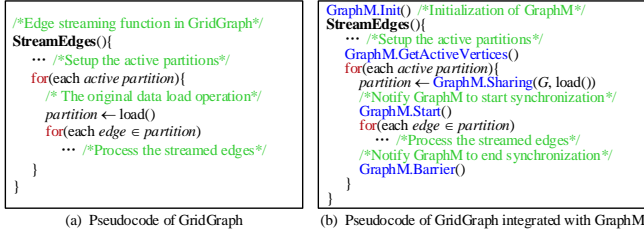
Figure 6: An example to illustrate how to integrate GraphM into existing graph processing system

Meanwhile, two notification functions (i.e., *Start()* and *Barrier()*) are inserted at the beginning and the end of the procedure that traverses the shared graph structure partition for the graph processing systems, respectively. Note that *GetActiveVertices()* is also provided to get the active vertices before each iteration, because some graph processing systems (e.g., GridGraph [50]) allow to use this operation to skip the processing of inactive vertices. Figure 6 takes GridGraph [50] as an example to show how to integrate existing graph processing systems with GraphM to efficiently support concurrent graph processing jobs, where *Load()* is the graph loading operation of GridGraph [50].

3.2 Graph Preprocessing

The CPU utilization ratio and the cache locality may be influenced by the chunk size, denoted by S_c. Setting it too large may increase the data access cost. This is because when only a part of a chunk can be loaded into the LLC, this part has to be swapped out when the rest of the chuck is loaded into the LLC. Since a chunk will be accessed by different concurrent jobs, the part of the chunk that has been swapped out has to be loaded into the LLC again, which increases the overhead. On the contrary, setting the chunk size too small may lead to frequent synchronization among the concurrent jobs that are processing this chunk since only when concurrent jobs have finished processing this chuck can they move to process the next one.

The suitable chunk size S_c is determined in the following way. N denotes the number of CPU cores, and C_{LLC} denotes the size of the LLC. S_c is set to be such a maximum integer that satisfies Formula 1, where S_G is the size of the graph data, $|V|$ is the number of vertices in the graph, U_v is the data size of each vertex, and r is the size of the reserved space in the LLC. The first term on the right of the formula represents the LLC size required to accommodate the chunks which are concurrently processed by the threads of a running job (the number of threads usually equals to the number of CPU cores in the computer, hence we have $S_c \times N$). The second term represents the size required to store the job-specific data in the LLC. Note that the size of a chunk is also a common multiple of the size of an edge and the size of a cache line for better locality.

$$S_c \times N + \frac{S_c \times N}{S_G} \times |V| \times U_v + r \leq C_{LLC} \tag{1}$$

With this setting, the same chunk only needs to be loaded into the LLC once and is then reused by all concurrent jobs with low synchronization cost. Only the job-specific data need to be replaced by different jobs, where the jobs are triggered to handle the loaded data in a round-robin way.

Note that the graph is not physically divided into the chunks of the size discussed above. Rather, in the preprocessing phase, the

---

Algorithm 1 Partition Labelling Algorithm

1: **function** LABEL(P^i, Set_c^i)
2:      $edge\_num \leftarrow 0$
3:      $c\_table \leftarrow null$
4:      **for** each edge $e \in P^i$ **do**
5:          **if** $e_s \in c\_table$ **then**
6:              $c\_table.N^+(e_s) \leftarrow c\_table.N^+(e_s) + 1$
7:          **else**
8:              $c\_table.\text{InsertEntry}(\langle e_s, 1 \rangle)$
9:          **end if**
10:          /*Count the number of edges labelled in c_table*/
             $edge\_num \leftarrow edge\_num + 1$
11:          **if** $edge\_num \times \frac{S_G}{|E|} \geq S_c$ or P^i is visited **then**
12:              $Set_c^i.\text{Store}(c\_table)$
13:              /*Prepare to store information of next chunk*/
             $\text{Clear}(c\_table, edge\_num)$
14:          **end if**
15:      **end for**
16: **end function**

---

graph is traversed so that each graph partition is labelled as a series of chunks in the order in which the graph data is streamed into the LLC. The labelling information of each chunk is stored in a key-value table, called *chunk_table*. Each entry of the *chunk_table* is a key-value pair, in which the key is the ID of a source vertex in the chunk (denoted by v) and the value is the number of this vertex's outgoing edges in the chunk (denoted by $N^+(v)$).

Algorithm 1 shows how to label a partition, e.g., P^i. When an edge is traversed, $N^+(v)$ of the edge's source vertex (i.e., e_s) is incremented by one in the corresponding entry of *chunk_table* (i.e., c_table) (Line 6). If the source vertex of the edge is not found in c_table, a new entry (i.e., a key-value pair) is created with the value being 1 and inserted into the table (Line 8). When the number of edges in c_table makes the chunk size to be the value determined by Formula 1 or all edges of P^i are visited (Line 11), these edges are treated as a chunk and the c_table is stored as an element of an array, i.e., Set_c^i (it holds the information of all chunks in P^i) (Line 12). c_table is then cleared and used to store the information of the next labelled chunk, where the value of $edge\_num$ is reset to zero (Line 13). Note that this procedure only runs once for a graph processing system, although it incurs extra cost.

3.3 Memory Sharing of Graph Structure

3.3.1 Sharing the Graph Structure. Generally, the graph structure data is divided into a series of partitions for parallel processing. A partition is loaded into the memory only when it is needed by some jobs (we call such partitions active partitions). Note that the active partitions for different jobs are usually different, because the jobs may traverse the graph along different paths. Such information can be directly obtained from the graph processing system (e.g., the *should_access_shard* array in the GridGraph [50]) because the set of partitions to be processed by each job in the next iteration can be gotten through tracing the partitions activated within the current iteration. A global table is created to gather this information. Each entry in the global table is a linked list to store the process

Algorithm 2 Graph Sharing Algorithm

1: **function** SHARING(G, LOAD()) /*Triggered by job j*/
2:     /*Get an active partition P^i that needs be loaded*/
    $P^i \leftarrow$ GetActivePartition()
3:     /*Get the set of jobs that need to handle P^i*/
    $J^i \leftarrow$ GetJobs(P^i)
4:     Resume(J^i) /*Resume the suspended jobs in J^i*/
5:     **if** $j \notin J^i$ **then** /*j does not need to handle P^i*/
6:         Suspend(j) /*Suspend the job j*/
7:     **end if**
8:     **if** P^i is not in the memory **then**
9:         /*Create a shared buffer to store P^i*/
        $Buf \leftarrow$ CreateMemory(G, P^i)
10:         $Buf \leftarrow$ Load(P^i) /*Load P^i into Buf*/
11:     **else** /*Job j gets P^i in the shared buffer*/
12:         $Buf \leftarrow$ Attach(G) /*Attach Buf to j*/
13:     **end if**
14:     Remove(j, J^i) /*Remove j from J^i*/
15:     **return** Buf
16: **end function**

Figure 7: Consistent snapshots for concurrent jobs

IDs (PIDs) of the active jobs of the corresponding graph partition. Each job needs to update the global table in real time. Particularly, the order of the entries in the global table is determined by the loading order of their corresponding graph partitions by default. After that, a lightweight API (i.e., the function $Sharing()$ described in Algorithm 2) is designed to extend the graph loading operation in the existing systems, allowing the job (e.g., the job j) to share the loaded graph partitions.

In detail, when the job j calls the function $Sharing()$, it first gets the ID of an active partition P^i assigned by GraphM (Line 2), where the loading order of the partitions to be loaded is assigned by GraphM along the order of their corresponding entries in the global table. The set of jobs (e.g., J^i) that need to handle P^i are then obtained from the global table (Line 3). After that, the active jobs recorded in J^i are popped from $suspended\_queue$ and their executions are resumed (Line 4), if these jobs are suspended. If the current job j is not in J^i (i.e., j does not need to handle P^i), it is suspended to release its computing resources for the execution of the jobs that need to handle P^i, and is pushed into $suspended\_queue$ (Lines 5-7). If P^i is not in the shared memory, a buffer Buf is created to store it (Lines 9-10). Otherwise, the job j is assigned to share the loaded graph partition P^i in the memory with the other concurrent jobs (Line 12). Then, the PIDs of the job j is removed from the J^i (Line 14) and the job j begins to handle the graph partition P^i referenced by the returned results (Line 15).

3.3.2 Ensuring of Consistent Snapshots. The shared graph may be updated or mutated by some jobs over time, resulting in incorrect results or runtime errors. Hence, a storage management technique is further designed to ensure the consistency of snapshots among the concurrent jobs.

When the shared graph structure data needs to be modified, e.g., removing or adding edges/vertices, users need to call our API to handle the evolving graph. In detail, GraphM first copies the corresponding chunks of the graph data that need to be modified to
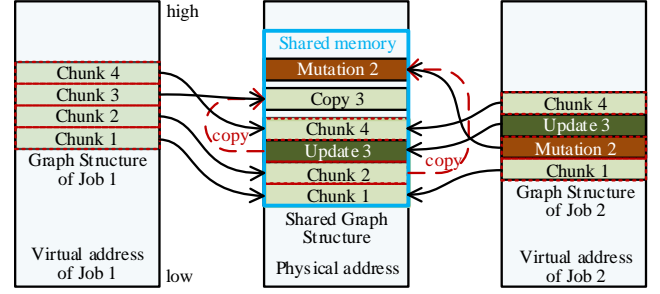
other shared memory space. If it is mutated by a job, the modification will be applied to the copied chunks, and alter the mapping of the virtual address of the corresponding chunks in the job to the copied chunks. Thus, the shared graph structure is not changed, it can be shared by other jobs. Besides, the copied chunks will be released when the corresponding job is finished. Different from graph mutation, which is only visible to its corresponding job, the graph update is only available to the jobs submitted after the update. Therefore, the shared graph structure will be updated to serve the newly submitted jobs, and previous jobs can refer to the copied chunks to continue their calculation. Note that when all previous jobs are completed, these copied chunks will be released. By doing so, the shared graph structure is always visible and shared by the newly submitted jobs. Note that Set_c^i also needs to be updated accordingly when the shared graph is updated.

As shown in Figure 7, two jobs are submitted, where *job* 1 is submitted before *job* 2. If a graph update arrives after the submission of *job* 1, it will create a copy (e.g., *copy* 3) for the corresponding graph structure data (e.g., *chunk* 3) for *job* 1 to use, and *chunk* 3 is going to be updated. Besides, the copied data will be released when *job* 1 is finished. Then, a new graph structure chunk (i.e., *update* 3) is constructed before *job* 2 is submitted. If *job* 2 needs to modify a chunk (e.g., *copy* 2) of the graph structure, the mutation is applied to the copied data to generate *mutation* 2, which is only visible to *job* 2. Note that the graph mutations and updates usually only happen to a small fraction of graph data, and thus a majority of the graph structure data can be shared by concurrent jobs and the update cost of the Set_c^i is also small.

3.4 Fine-grained Synchronization for Regular Streaming

This section discusses the details of the fine-grained synchronization way for efficient execution of concurrent jobs.

3.4.1 Mining the Similarities between Concurrent Jobs. This fine-grained synchronization scheme mines the chunks of the shared graph that can be concurrently handled by the jobs in each iteration. Moreover, the similarities are dynamically changed because the vertices in the chunks may be activated or converged in some jobs during the iteration, which therefore needs to be dynamically updated before each iteration.

First, we monitor the chunks that need to be processed by each job in the current iteration, i.e., some vertices in these chunks are active for the job, especially the job that needs to skip the useless streaming. For example, in each iteration, SSSP [28] may only need to process a part of the graph data, whereas PageRank [29] usually has to traverse the entire graph structure. This information can be

procured by tracing the change of vertices states after each iteration. In general, a vertex needs to be processed within the current iteration only when its value has been updated by the neighbours within the previous iteration. Note that the active vertices in the first iteration are designated by the user for each job. To express the active vertices succinctly, a bitmap is created for each job. If some jobs do not skip the useless streaming, all of their vertices are active by default. Then, the active chunks of concurrent jobs can be obtained by their bitmaps and the *chunk_table* arrays. Finally, the similarities between the data accesses of the concurrent jobs can be rapidly obtained based on the intersection of their active chunks.

3.4.2 Fine-grained Synchronization of Traversals. To fully exploit the temporal similarity between the data access of the concurrent jobs, it enables the chunks loaded into the LLC to be processed by these jobs in a regular way. In detail, the computing resources are unevenly allocated to the concurrent jobs to synchronize their data accesses, because the computational loads of different jobs are usually skewed when processing each chunk. Generally, the load of each job j for a chunk is determined not only by the amount of edges that need to be processed, but also by the computational complexity of the edge processing function of this job, denoted as $T(F_j)$. In addition, the average data access time for each edge, indicated as $T(E)$, affects the execution time of the jobs. Thus, for each job, the fine-grained synchronization has two phases, i.e., the profiling phase and the syncing phase.

Profiling Phase. This phase is to profile the needed information (i.e., $T(F_j)$ and $T(E)$) of the jobs. When a new job (e.g., the job j) is submitted, the profiling phase of this job traverses the shared graph partition (e.g., P^i) and captures its execution time, denoted by T_j^i, which is composed of the graph processing time and the graph data access time. Thus, T_j^i is represented as the following formula:

$$T(F_j) \times \sum_{k \in C^i} \sum_{v \in V_k \cap A_j} N_k^+(v) + $$
$$T(E) \times \sum_{k \in C^i} \sum_{v \in V_k} N_k^+(v) = T_j^i \tag{2}$$

where C^i is the set of chunks in the partition P^i, and V_k is the set of vertices in the kth chunk. A_j is the set of active vertices for the job j within the current iteration, which can be easily obtained via its bitmap. $N_k^+(v)$ is the number of out-going edges of the vertex v in the kth chunk. V_k and $N_k^+(v)$ are stored in the corresponding Set_c^i. According to Formula 2, after the processing of the first two active partitions of each job j, the needed information, i.e., $T(F_j)$ and $T(E)$, of the job j can be obtained, where $T(E)$ is a constant for the same graph and only needs to be profiled once for different jobs.

Syncing Phase. After obtaining $T(F_j)$ of the concurrent jobs, the computational load of the jobs in each chunk is easily acquired before each iteration. In detail, the computational load of the jth job for the processing the kth chunk (i.e., L_j^k) can be determined by the following equation:

$$L_j^k = T(F_j) \times \sum_{v \in V_k \cap A_j} N_k^+(v) \tag{3}$$

Each partition may be handled by the threads of different concurrent jobs. To achieve fine-grained synchronization and better locality, the threads of different jobs handling the same partition need to be migrated to the same CPU core to synchronize their
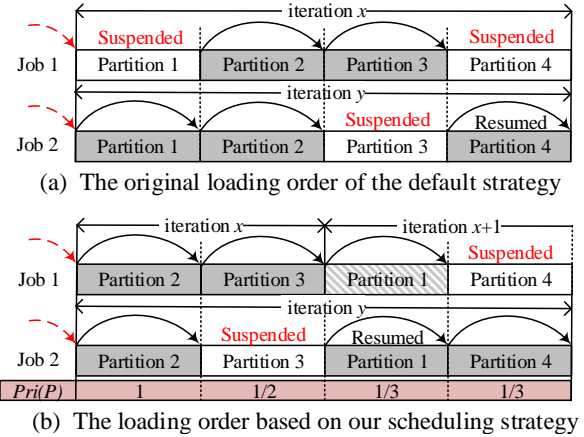

(a) The original loading order of the default strategy

(b) The loading order based on our scheduling strategy

Figure 8: An example to illustrate the scheduling of loading order of graph partitions, where *partition* 1 is activated by the other partitions of *job* 1 and can be handled at the $(x + 1)^{th}$ iteration for *job* 1

data access by scheduling their CPU time slices. Usually, only a small amount of migrations are generated, because these threads access a series of chunks synchronously each time in the partition. After that, the computing resources need to be allocated unevenly to the threads of the concurrent jobs according to the skewed computational load. Each thread monopolizes the CPU time to finish processing the current chunk. Apparently, the execution time of the chunk for each thread can be represented by its corresponding *computational load* according to Formula 3, except the thread that first processes the chunk. It is because that the graph data needs to be loaded into the LLC by this thread and its execution time F_j^k for the kth chunk can be obtained by the following equation:

$$F_j^k = L_j^k + T(E) \times \sum_{v \in V_k} N_k^+(v) \tag{4}$$

After the corresponding execution times of all threads have elapsed, the jobs will process the next chunk concurrently with the reallocated computing resources.

4 THE SCHEDULING STRATEGY FOR OUT-OF-CORE GRAPH ANALYSIS

The loading order of graph partitions in the out-of-core graph processing systems may cause the similarities between the data access of concurrent jobs not to be fully exploited due to the following reasons. First, some jobs may only need to handle a few of partitions in the current iteration, but more partitions will be activated in the next iteration. For example, in BFS [11] and SSSP [28] only one or a few vertices are active at the beginning, but then a large number of vertices will be activated by these vertices. Second, the activated partitions may be accessed by other jobs in the current iteration, e.g., PageRank [29] and WCC [35]. Hence, a partition may be repeatedly loaded into the memory to serve different jobs in contiguous iterations, resulting inefficient usage of the partitions that are loaded into the memory.

Because loading the partitions in different orders does not influence the correctness of the final results [44], a scheduling strategy is proposed to fully exploit the similarities across different iterations. The key idea is to load the partitions that are handled by the jobs

Table 2: Graph datasets used in the experiments

Datasets	Vertices	Edges	Data sizes
LiveJ [5]	4.8 M	69 M	526 MB
Orkut [5]	3.1 M	117.2 M	894 MB
Twitter [23]	41.7 M	1.5 B	10.9 GB
UK-union [9]	133.6 M	5.5 B	40.1 GB
Clueweb12 [4]	978.4 M	42.6 B	317 GB

with the least number of active partitions. Other partitions may be activated in these jobs, which can then advance to next iteration to process the activated partitions, as shown in Figure 8. In this way, the strategy enables the partitions loaded into the memory to serve more concurrent jobs, further amortizing the data access cost, especially when the size of the graph is very large.

To achieve the goal described above, each partition is assigned a priority. The partitions with the higher priority are loaded first into the memory to serve the related jobs, so that these jobs can complete current iteration as quickly as possible to activate other partitions. Two rules are applied when setting the priority. First, the partitions are given a higher priority when they are handled by the jobs with fewer active partitions. Second, a partition is given the highest priority when it is processed by most jobs. In summary, the priority $Pri(P^i)$ of each partition P^i is set using Equation 5, where J^i denotes the set of jobs to handle P^i in the next iteration, $N_j(P)$ denotes the number of active partitions of the jth job (i.e., a job of the set J^i), and $N(J^i)$ denotes the number of jobs in the set J^i.

$$Pri(P^i) = MAX_{j \in J^i} \frac{1}{N_j(P)} \times N(J^i) \qquad (5)$$

The values of $N_j(P)$ and $N(J^i)$ are directly obtained from the global table. The priority is calculated before each complete traversal over all the partitions. After that, the entries in the global table are sorted according to the priority of their corresponding partitions and determine the loading order of the partitions. From Figure 8, we can observe that the partition 1 can serve more concurrent jobs when it has been loaded into the memory via this scheduling strategy. Then, the similarities between concurrent jobs are fully exploited.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

The experiments are conducted on a server with two 8-core Intel Xeon E5-2670 CPUs (each CPU has 20 MB last-level cache) operating at the clock frequency of 2.6 GHz, a 32 GB memory and a 1 TB hard drive, running Linux kernel 2.6.32. All codes are compiled with cmake version 3.11.0 and gcc version 4.9.4. Table 2 shows the properties of the five real-world graphs used in our experiments, where LiveJ, Orkut, and Twitter can be stored in the memory, while the size of UK-union and Clueweb12 are larger than the memory size. Four representative graph processing algorithms are used as benchmarks, including *weakly connected component* (WCC) [35], PageRank [29], *single source shortest path* (SSSP) [28], *breadth-first search* (BFS) [11]. These algorithms have different characteristics in the data access and resource usage. For example, PageRank and WCC are network-intensive [40], which need to frequently traverse the majority of the graph structure, whereas SSSP and BFS only traverse a small fraction of the graph at the beginning.

Table 3: Preprocessing time (in seconds)

	LiveJ	Orkut	Twitter	UK-union	Clueweb12
GridGraph	20.89	35.07	439.59	2,312.11	19,267.28
GridGraph-M	21.86	35.76	463.65	2,681.04	22,401.90

To evaluate the performance, we submit WCC, PageRank, SSSP, and BFS in turn in a sequential or concurrent manner until the specific number of jobs are generated, where the parameters are randomly set for different jobs although these jobs may be the same graph algorithm. In detail, the damping factor is randomly set by a value between 0.1 and 0.85 for each PageRank job. The root vertices are randomly selected for the BFS jobs and the SSSP jobs. The total number of iterations is a randomly selected integer between one and the maximum number of iterations for each WCC job. For the concurrent manner, the time interval between successive two submissions follows the poisson distribution [15] with $\lambda = 16$ by default. All benchmarks are run for ten times and the experimental results are the average value.

To evaluate the advantages of GraphM, we integrate GridGraph [50] with GraphM (called *GridGraph-M* in the experiments) to run multiple concurrent graph processing jobs. We then compare *GridGraph-M* with two execution schemes of the original GridGraph, called *GridGraph-S* and *GridGraph-C*. *GridGraph-S* sequentially processes the jobs, while *GridGraph-C* concurrently handles the jobs (but each job runs independently without sharing the underlying graph structure data as in *GridGraph-M*). In *GridGraph-C*, the concurrent jobs are managed by the operating system. We choose GridGraph [50] since it is a state-of-the-art one and outperforms other out-of-core graph processing systems [24, 33].

In addition, we also finally integrate GraphM into the other popular systems (i.e., GraphChi [24], PowerGraph [14], and Chaos [32]) and evaluate their performance. There, Eigen (version 3.2.10) is needed by GraphChi. OpenMPI (version 2.1.6), boost (version 1.53.0), zookeeper (version 3.5.1), bzip2 (version 1.0.6), gperftools (version 2.0), hadoop (version 1.0.1), and libevebt (version 2.0.18) are required by PowerGraph. Boost (version 1.53.0) and zmq (version 4.3.1) are needed by Chaos. The experiments of PowerGraph and Chaos are done on a cluster with 128 nodes, which is connected via 1-Gigabit Ethernet. Each node is the same as the above described. Because PowerGraph and Chaos may not get the best performance due to high communication cost when all nodes are used to handle all jobs for some graphs, the nodes are divided into groups and each group of nodes are used to handle a subset of jobs so as to make the jobs executed over PowerGraph and Chaos in a high throughput mode, where the newly submitted jobs are assigned to the groups in turn. Note that, when some jobs need to be executed on Power-Graph/Chaos over a group of nodes, the graph is only loaded into the distributed shared memory consisting of the memory of this group of nodes. In the experiments, for high throughput of 64 jobs over LiveJ, Orkut, Twitter, UK-union, and Clueweb12, the suitable number of groups is set to 8, 8, 4, 1, 1 for PowerGraph and 8, 4, 2, 1, 1 for Chaos, respectively.

5.2 Preprocessing Cost

Table 3 shows the preprocessing cost of GridGraph and GridGraph-M. We can observe that GridGraph-M takes little additional time than the original system GridGraph, so as to create the *chunk_table* and label the graphs by traversing the graphs once. When the size
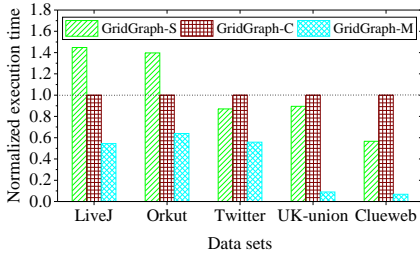
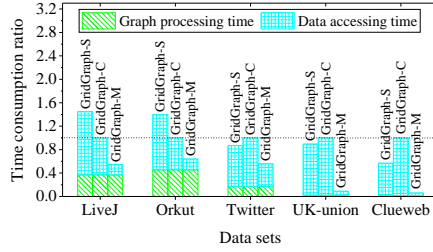**Figure 9: Total execution time for the 16 jobs with different schemes**

Figure 10: Execution time breakdown of jobs with different schemes
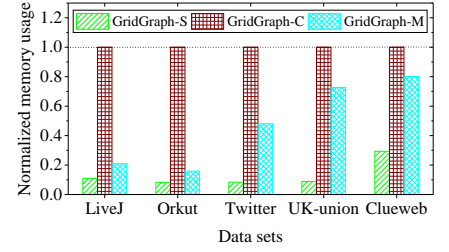

Figure 11: Memory usage for the 16 jobs with different schemes

of graph is larger than the memory size, the labelling procedure of the graph increases the preprocessing time by an average of 16.1%, because these graph needs to be reloaded into the memory. When the graph can be stored in the memory, the labelling procedure of the graph only increases the preprocessing time by an average of 4%. As evaluated, the extra storage cost of GraphM is also small and occupies 5.5%-19.2% of the space overhead of the original graph, i.e., 70.6 MB (13.4%), 49.2 MB (5.5%), 2.09 GB (19.2%), 4.5 GB (11.2%), and 19.9 GB (6.3%) for LiveJ, Orkut, Twitter, UK-union, and Clueweb12, respectively. In general, when the graph has larger maximum out-degree and lower average out-degree, the ratio of its extra space overhead to the space overhead of the original graph is higher. It is because that the vertices with larger out-degree have more replicas stored in different chunks and the extra space overhead is also usually proportional to the ratio of the number of vertices to the number of edges. For example, the maximum out-degree and the average out-degree are 2,997,469 and 35 for Twitter, respectively, while they are 7,447 and 48 for Clueweb12, respectively. Thus, the space overhead ratio of Twitter is higher than that of Clueweb12. Note that, although GraphM needs such extra space overhead, more storage overhead can be spared by GraphM because only one copy of the graph structure data (instead of multiple copies) needs to be maintained by existing systems for multiple jobs when they are integrated with GraphM.

5.3 Overall Performance Comparison

Figure 9 shows the total execution time of 16 concurrent jobs with different schemes. It can be observed that GridGraph-M achieves shorter execution time (thus higher throughput) than the other two schemes for all graphs. Comparing with GridGraph-S and GridGraph-C, GridGraph-M improves the throughput by about 2.6 times and 1.73 times on average respectively when the graphs can be stored in the memory, and by 11.6 times and 13 times on average respectively in the case of out-of-core processing. The throughput improvement is achieved for the lower data access cost in GridGraph-M.

To evaluate data access cost, we further break down the total execution time in Figure 10. It can be observed from this figure that less graph data accessing time is required in GridGraph-M compared with the other two schemes, especially when the size of the graph is very large. For example, for UK-union, the data accessing time is reduced by 11.48 times and 13.06 times in GridGraph-M in comparison with GridGraph-S and GridGraph-C. The reasons for the lower data access cost of GraphM are two-fold: *1)* only a single copy of the same graph data needs to be loaded and maintained in the

memory to serve the concurrent jobs, reducing the consumption of memory and disk bandwidth and the intense resource contention; *2)* the graph data is regularly streamed into the LLC to be reused by the jobs, which avoids unnecessary memory data transfer by reducing LLC miss rate and minimizes the volume of data swapped into the LLC.

Figure 11 shows the usage of main memory during the execution. As observed, GridGraph-M consumes less memory than GridGraph-C, but more than GridGraph-S. This is because the graph structure data is shared in the memory for concurrent jobs by GraphM (thus GridGraph-M consumes less memory than GridGraph-C), and the job-specific data of all concurrent jobs as well as the *chunk_table* of the loaded graph data is loaded into the memory at the same time (thus GridGraph-M consumes more memory than GridGraph-S). Note that as the number of vertices in the graph increases, the job-specific data for concurrent jobs need more memory resource. For example, the memory usage of GridGraph-M over UK-union is 8.2 times bigger than that of GridGraph-S because the job-specific data of the 16 jobs is stored in the memory. However, it is still only 71% of the memory usage of GridGraph-C. Hence, the memory resource is efficiently utilized in GraphM since redundant memory consumption regarding the common graph data is eliminated.

In Figure 12, we evaluate the total I/O overhead of these 16 jobs over three schemes. As observed, the I/O overhead is significantly reduced in GridGraph-M when the size of the graph data is larger than the memory size. It is because that the same graph data only needs to be loaded into the memory once in each iteration for concurrent jobs. However, when the graph can be fitted in the memory, there is no much difference in the I/O overhead among these three schemes, since this graph is cached in the memory via memory mapping and only needs to be read from disks once. Thus, GraphM brings better performance improvement for the out-of-core graph processing for less I/O cost. More specifically, when processing UK-union, the I/O overhead is reduced by 9.2 times and 10.1 times compared with GridGraph-S and GridGraph-C. In addition, GridGraph-C usually performs more I/O operations than GridGraph-S, because there is intense contention for using the memory resource among the jobs, which causes the graph data to be swapped out of the memory.

Next, we evaluate the LLC utilization of the different schemas and show the results in Figure 13. As observed, the LLC miss rate of GridGraph-M is lower than that of both GridGraph-S and GridGraph-C. For example, for UK-union, the LLC miss rate of GridGraph-M is only 15.69%, while the rates are 45.3% and 43.3% for GridGraph-S and GridGraph-C, respectively. It is mainly because
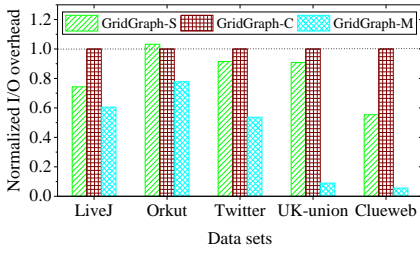
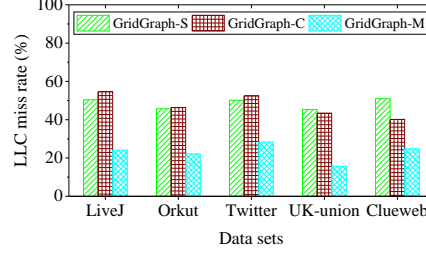**Figure 12: Total I/O overhead for 16 jobs with different schemes**

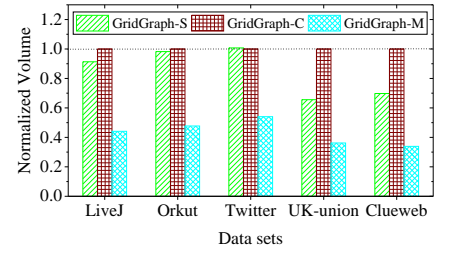**Figure 13: LLC miss rate for 16 jobs with different schemes**

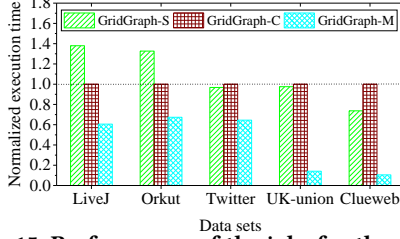**Figure 14: Volume of data swapped into the LLC for 16 jobs**
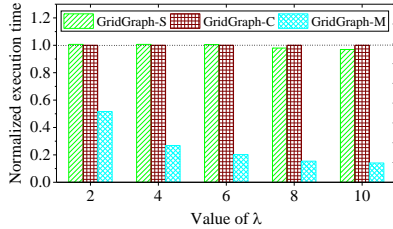

Figure 15: Performance of the jobs for the real-trace

Figure 17: Impacts of the distance between the root vertices of BFS or SSSP jobs

Figure 16: Performance of GraphM for various λ

only a single copy of the graph structure data is loaded into the LLC and the access to this data is shared by the jobs. The graph structure data loaded into the LLC can serve more concurrent jobs in GridGraph-M, resulting in better data locality for these jobs.

Moreover, we traced the total amount of data swapped into the LLC for these 16 jobs. Generally, GridGraph-C needs to swap a larger amount of graph data into the LLC than GridGraph-S, because there is more redundant memory data transfer caused by the intense cache interference among concurrent jobs. As shown in Figure 14, when processing UK-union, the amount of data swapped into the LLC in GridGraph-S is 65% of GridGraph-C. Nevertheless, we observe that the amount of swapped data in GridGraph-M is still much less than GridGraph-S (e.g., only 55% for UK-union). This is because the data access similarities among concurrent jobs are fully exploited by GraphM.

We also evaluate the performance of GraphM via submitting the jobs according to the real trace shown in Figure 2, where different number of jobs are submitted at various point of time according to the real trace. In Figure 15, the results show that GridGraph-M improves the throughputs of GridGraph-S and GridGraph-C by 1.5–7.1 times, and 1.48–9.8 times, respectively, for the real trace, because of lower graph storage overhead and less data access cost.

In addition, we evaluate the impacts of job submission frequency on GraphM over UK-union in Figure 16 by using different value of λ. The results show that higher speedup is obtained by GraphM when the jobs is more frequently submitted (i.e., larger λ). Figure 17 shows the performance of 16 BFS or SSSP jobs with randomly selected root vertices within the range of different number of hops over LiveJ.
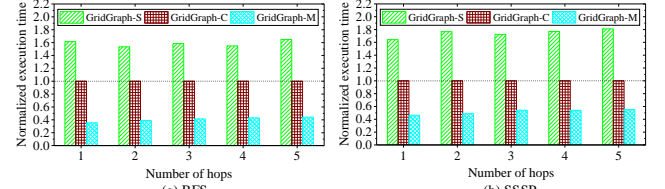
Table 4: Execution time (in seconds) for other systems (i.e., GraphChi [24], PowerGraph [14], and Chaos [32]) integrated with GraphM, where "−" means it cannot be executed for memory errors

	LiveJ	Orkut	Twitter	UK-union	Clueweb12
GraphChi-S	2,348	2,248	43,032	149,352	>1 week
GraphChi-C	776	696	10,580	38,760	>1 week
GraphChi-M	344	468	6,128	12,436	248,840
PowerGraph-S	92	144	1,408	7,183	−
PowerGraph-C	83	111	1,153	6,653	−
PowerGraph-M	43	75	795	3,820	−
Chaos-S	224	159	4,668	29,538	487,272
Chaos-C	516	588	12,011	30,943	>1 week
Chaos-M	121	106	2,261	10,614	156,881

We find that higher speedup is achieved by GraphM for stronger spatial/temporal similarities of the data accesses when the root vertices of the BFS or SSSP jobs are closer to each other.

5.4 Performance of Scheduling Strategy

We also evaluate the impacts of our scheduling strategy on the performance of GraphM when it is integrated with GridGraph. GridGraph-M and GridGraph-M-without are the versions of GridGraph-M with our scheduling strategy (Section 4) and without our scheduling strategy, respectively. In Figure 18, we traced the total execution time of the above 16 jobs on GridGraph-M and GridGraph-M-without. We can observe that GridGraph-M always outperforms GridGraph-M-without. The execution time of GridGraph-M is only 72.5% of GridGraph-M-without over Clueweb12. It is because the graph partitions loaded into the memory can serve as many concurrent jobs as possible, minimizing the data access cost.

5.5 Integration with Other Systems

Table 4 shows the total execution time of 64 concurrent jobs with different schemes, where the experiments for different schemes of both PowerGraph and Chaos are done on a cluster with 128 nodes
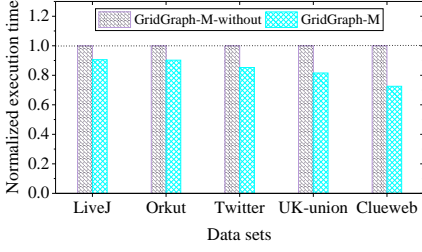
Figure 18: Total execution time with-out/with our scheduling

Figure 19: Total execution time for different number of jobs
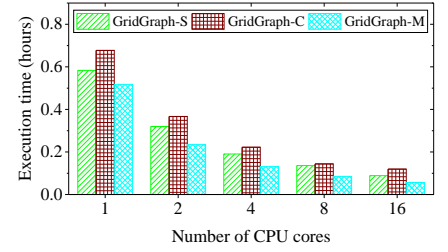

Figure 20: Total execution time on different number of CPU cores
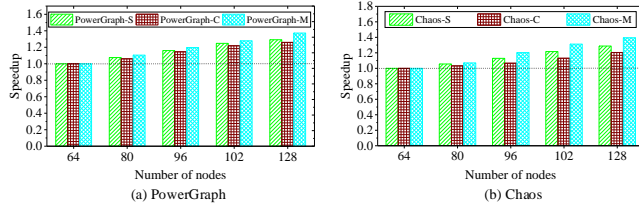

Figure 21: Scalability of different distributed schemes

as described in Section 5.1. We can observe that all systems get better speedups after integrating GraphM into them. Diverse graph processing systems get various performance improvements after using GraphM, because the ratios of the graph access time to the total execution time are different for them. In general, when the ratio of data access time to the execution time is higher for the original system, the greater performance improvement is gotten by GraphM via reducing the redundant graph structure data storage overhead and access cost.

5.6 Scalability of GraphM

Figure 19 shows the performance of various number of concurrent PageRank jobs on different schemes of GridGraph over Clueweb12. Better performance improvement is achieved by GraphM when the number of jobs increases. GridGraph-M gets speedups of 1.79, 3.04, 4.92, and 5.94 against GridGraph-S when the number of jobs is 2, 4, 8, and 16, respectively. It is because that more data access and storage cost is spared by GraphM through amortizing it, as the number of jobs increased. Note that the fine-grained synchronization operation of GraphM does not occur when there is only one job, and thus there is no much difference in the execution time among the three schemes at this moment. The synchronization cost occupies 7.1%–14.6% of the total execution time of the job on GraphM for our tested instances. In addition, as the contention for resources (e.g., memory and bandwidth) gets more serious, the performance of GridGraph-C becomes much worse than that of GridGraph-M, even GridGraph-S. Thus, simply adopting existing graph processing systems to support concurrent jobs may be a terrible choice.

We then evaluate the scaling out performance of GraphM. For this goal, we first evaluate the execution time of 16 jobs on different schemes of GridGraph for Twitter on a single PC by increasing the number of CPU cores. From Figure 20, we find that GridGraph-M performs better than other ones under any circumstances, especially when the number of cores is more, because the storage and access of the graph structure data is shared by the concurrent jobs in GridGraph-M, while the other schemes have a higher data access cost. Second, we evaluate the performance of 64 jobs on different

schemes of PowerGraph and Chaos over UK-union in Figure 21 by increasing the number of nodes from 64 to 128. We can observe that better scalability is also achieved by integrating GraphM with them because of less communication cost. Because the scalability of GraphM is greatly decent in most situations, we believe it can efficiently support concurrent graph processing in industry.

6 RELATED WORK

Recently, many graph processing systems have been proposed. GraphChi [24] and X-Stream [33] achieve efficient out-of-core graph processing through sequentially accessing storage. Hao et al. [39] keep frequently accessed data locally to minimize the cache miss rate. TurboGraph [18] fully exploits the parallelism of multi-core and FlashSSD to overlap CPU computation and I/O operation. FlashGraph [49] adopts a semi-external memory graph engine to achieve high IOPS and parallelism. PathGraph [41] designs a path-centric method to acquire better locality. By using the novel grid format and the streaming-apply model, GridGraph [50] improves the locality and reduces I/O operations. HotGraph [44], FBSGraph [45], DGraph [47], and DiGraph [46] accelerate graph processing via faster state propagation. However, these systems mainly focus on optimizing individual graph processing, which lead to redundant storage consumption and data access cost as handling multiple concurrent graph processing on same graph. Hence, Seraph [40] tries to decouple the data model and computing logics for less consumption of memory. CGraph [43, 48] proposes to reduce the redundant data accesses in the concurrent jobs. Nevertheless, they are tightly coupled to their own programming models and graph processing engines, which cause re-engineering burden of various applications for users while using these engines. Compared with them, GraphM transparently improves the throughput of concurrent jobs on existing graph processing systems.

Meanwhile, some storage optimization methods are also proposed to support efficient iterative graph processing. Graphene [25] supports graph processing on disks via fine-grained I/O management. GraFBoost [20] uses sort-reduce accelerator to reduce the number of I/O accesses to flash storage. V-Part [13] optimizes GraFBoost [20] using a new graph partition scheme. GraphOne [22] proposes a storage system for evolving graph using a data abstraction to enable data access at different granularities with only a small data duplication. However, these storage systems still suffer from high redundant data access cost for concurrent iterative graph jobs without the consideration of the data access similarities. Note that some graph storage and querying systems [10, 17, 30, 31, 34, 42] are recently devised for concurrent graph queries. However, they are

dedicated to graph queries which usually only access different small subsets of a graph for exactly once, and can not efficiently support iterative graph processing which needs to frequently traverse the whole graph.

7 CONCLUSION

This paper proposes a storage system GraphM to integrate with existing graph processing systems for efficient execution of multiple concurrent graph processing jobs. The key mechanisms are an efficient *Share-Synchronize* method to enable multiple jobs to share the graph structure data in the LLC/memory, and a scheduling strategy to maximize the utilization ratio of the graph structure data loaded into the memory. Experimental results show that GraphM brings a performance improvement up to 13 times. In future, we will first exploit the use of new hardware to accelerate data accesses of concurrent jobs for higher throughput. Second, we will further optimize GraphM for distributed platforms. Third, we will also optimize it for evolving graphs. Finally, we will also research the security problems of GraphM because it may be attacked by some jobs when they share a common graph and some private information may also be leaked.

ACKNOWLEDGMENTS

REFERENCES

[1] 2019. didi. http://www.didiglobal.com/.
[2] 2019. facebook. http://www.facebook.com/.
[3] 2019. huawei. http://www.huawei.com/.
[4] 2019. LAW. http://law.di.unimi.it/datasets.php.
[5] 2019. SNAP. http://snap.stanford.edu/data/index.html.
[6] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and Optimization of the Memory Hierarchy for Graph Processing Workloads. In *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture*. 373–386.
[7] Johannes Blum, Stefan Funke, and Sabine Storandt. 2018. Sublinear Search Spaces for Shortest Path Planning in Grid and Road Networks. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*. 6119–6126.
[8] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web*. 587–596.
[9] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A Large Time-Aware Graph. *ACM SIGIR Forum* 42, 2 (2008), 33–38.
[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*. 49–60.
[11] Aydin Buluç and Kamesh Madduri. 2011. Parallel Breadth-first Search on Distributed Memory Systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.
[12] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John C. S. Lui, and Cheng He. 2015. VENUS: Vertex-centric streamlined graph computation on a single PC. In *Proceedings of the 31st IEEE International Conference on Data Engineering*. 1131–1142.
[13] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. 2019. Large-Scale Graph Processing on Emerging Storage Devices. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 309–316.
[14] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs.

[15] In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. 17–30.
[15] James Goulding, Simon Preston, and Gavin Smith. 2016. Event Series Prediction via Non-Homogeneous Poisson Process Modelling. In *Proceedings of the 16th IEEE International Conference on Data Mining*. 161–170.
[16] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8, 9 (2015), 950–961.
[17] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data*. 1429–1446.
[18] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 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*. 77–85.
[19] Sungpack Hong, Nicole C. Rodia, and Kunle Olukotun. 2013. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *Proceedings of the 2013 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11.
[20] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*. 411–424.
[21] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A Scalable and Fast Graph Analytics System. In *Proceedings of the 2018 International Conference on Management of Data*. 395–410.
[22] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 249–263.
[23] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web*. 591–600.
[24] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.
[25] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained I/O Management for Graph Computing. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. 285–300.
[26] Yi Lu, James Cheng, Da Yan, and Huanhuan Wu. 2014. Large-scale distributed graph computing systems: an experimental evaluation. *Proceedings of the VLDB Endowment* 8, 3 (2014), 281–292.
[27] Jasmina Malicevic, Baptiste Joseph Eustache Lepers, and Willy Zwaenepoel. 2017. Everything You Always Wanted to Know about Multicore Graph Processing but Were Afraid to Ask. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 631–643.
[28] Ulrich Meyer. 2001. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*. 797–806.
[29] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. *The PageRank citation ranking: Bringing order to the web*. Technical Report. Stanford Digital Library Technologies Project.
[30] Peitian Pan and Chao Li. 2017. Congra: Towards Efficient Processing of Concurrent Graph Queries on Shared-Memory Machines. In *Proceedings of the 2017 International Conference on Computer Design*. 217–224.
[31] Vicky Papavasileiou, Ken Yocum, and Alin Deutsch. 2019. Ariadne: Online Provenance for Big Graph Analytics. In *Proceedings of the 2019 International Conference on Management of Data*. 521–536.
[32] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. 410–424.
[33] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 472–488.
[34] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 317–332.
[35] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146.
[36] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Helen Li, and Yiran Chen. 2018. GraphR: Accelerating Graph Processing Using ReRAM. In *Proceedings of the 24th IEEE International Symposium on High Performance Computer Architecture*. 531–543.
[37] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 429–442.

[38] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 389–404.

[39] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[40] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*. 227–238.

[41] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast Iterative Graph Computation: A Path Centric Approach. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*. 401–412.

[42] Qizhen Zhang, Akash Acharya, Hongzhi Chen, Simran Arora, Ang Chen, Vincent Liu, and Boon Thau Loo. 2019. Optimizing Declarative Graph Queries at Large Scale. In *Proceedings of the 2019 International Conference on Management of Data*. 1411–1428.

[43] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 441–452.

[44] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Guang Tan, and Bing Bing Zhou. 2017. HotGraph: Efficient Asynchronous Processing for Real-World Graphs. *IEEE Trans. Comput.* 66, 5 (2017), 799–809.

[45] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating Asynchronous Graph Processing via Forward and Backward Sweeping. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 895–907.

[46] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An Efficient Path-based Iterative Directed Graph Processing System on Multiple GPUs. In *Proceedings of the 2019 Architectural Support for Programming Languages and Operating Systems*. 601–614.

[47] Yu Zhang, Xiaofei Liao, Xiang Shi, Hai Jin, and Bingsheng He. 2018. Efficient Disk-Based Directed Graph Processing: A Strongly Connected Component Approach. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2018), 830–842.

[48] Yu Zhang, Jin Zhao, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, and Ligang He. 2019. CGraph: A Distributed Storage and Processing System for Concurrent Iterative Graph Analysis Jobs. *ACM Transactions on Storage* 15, 2 (2019), 10:1–10:26.

[49] Da Zheng, Disa Mhembere, Randal C. Burns, Joshua T. Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: processing billion-node graphs on an array of commodity SSDs. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 45–58.

[50] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference*. 375–386.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We integrated existing graph processing systems (i.e., GridGraph v1.0, GraphChi v1.0, PowerGraph v2.2, and Chaos v1.0) with GraphM by inserting our provided APIs into them. Then, we ran the four representative graph processing algorithms (i.e., PageRank, WCC, SSSP, and BFS) on these systems with different execution schemes (i.e., the sequential scheme, the concurrent scheme, and the scheme integrating with GraphM), as described in the paper.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:
https://github.com/GraphM-Program/GraphM.git

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon E5-2670 CPUs, lustre 2.1.3

Operating systems and versions: Red Hat 4.8.5-11 running Linux kernel 2.6.32

Compilers and versions: Compile with cmake version 3.11.0 and gcc version 4.9.4

Applications and versions: GridGraph v1.0, GraphChi v1.0, PowerGraph v2.2, and Chaos v1.0

Libraries and versions: Eigen v3.2.10, OpenMPI v2.1.6, Boost v 1.53.0, Zookeeper v3.5.1, Bzip2 v1.0.6, Gperftools v2.0, Hadoop v1.0.1, Libevebt v2.0.18, and Zmq v4.3.1

Key algorithms: PageRank, WCC, BFS, and SSSP

Input datasets and versions: LiveJournal, com-Orkut, twitter-2010, uk-union-2006-06-2007-05, and clueweb12

Paper Modifications: To benefit from GraphM, the users need to insert our APIs into the existing graph processing systems (i.e., GridGraph v1.0, GraphChi v1.0, PowerGraph v2.2, and Chaos v1.0 in our experiments). Note that it does not need to change the graph applications above these graph processing systems. In detail, Init() is used to initialize GraphM by preprocessing the graph as described in Section 3 in our paper. Sharing() function is inserted in existing graph processing systems to replace the original data load operation for the efficient load of the shared graph data. Note that the parameter in the function Sharing() is various for different graph processing systems, e.g., the parameter is the function Load() for GridGraph and is the function LoadSubgraph() for GraphChi. Meanwhile, two notification functions (i.e., Start() and Barrier()) are inserted at the beginning and the end of the procedure that traverses the shared graph structure partition for the graph processing systems, respectively. Note that GetActiveVertices() is also provided to get the active vertices before each iteration.

ARTIFACT EVALUATION

Verification and validation studies: GraphM enables concurrent iterative graph processing jobs to share the graph structure data in the last-level cache/memory and the data accesses to it, so that it can reduce the redundant data storage and access overhead. Therefore, we have evaluated the ratio of data access time to the total execution time, and the amount of the transferred data to ensure the effectiveness of our experiments. Besides, we have also integrated GraphM into different graph processing systems and tested the performance improvements gotten by GraphM to verify the efficiency of GraphM.

Accuracy and precision of timings: To ensure the accuracy of the experimental results, we run all the evaluated cases for 10 times and the experimental results are the average value.

Quantified the sensitivity of results to initial conditions and/or parameters of the computational environment: The performance improvements that GraphM bring are sensitive to the number of concurrent jobs and the size of the graph data. Generally, better performance improvement is achieved by GraphM when the number of jobs increases or the graph structure is larger. Therefore, we have evaluated the performance of various number of concurrent jobs on different schemes (i.e., the sequential scheme, the concurrent scheme, and the scheme integrating with GraphM) of the existing graph processing systems, and evaluated the overall performance improvements achieved by GraphM for different graphs.