Manuscript version: Author’s Accepted Manuscript
The version presented in WRAP is the author’s accepted manuscript and may differ from the published version or Version of Record.

Persistent WRAP URL:
http://wrap.warwick.ac.uk/130159

How to cite:
Please refer to published version for the most recent bibliographic citation information. If a published version is known of, the repository item page linked to above, will contain details on accessing it.

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions.

Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher’s statement:
Please refer to the repository item page, publisher’s statement section, for further information.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk.
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  
hkluo@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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

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

...
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 planning [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 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].
GraphM: An Efficient Storage System for Concurrent Graph Processing

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.

2 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.

2.1 Redundant Data Access Overhead

Figure 4(a) depicts the percentage of the graphs that are shared by different number of concurrent jobs traced from a 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.
### 3.1 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 [30], 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: \( P_j^i \leftarrow \text{Sharing}(G, \text{Load}) \). \( G \) is the name of the graph to be loaded and is used to identify the range of shared memory which contains the shared graph structure partition. \( \text{Load}() \) 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 \( P_j^i \) shared by the \( j^{th} \) job. When the jobs need to be concurrently executed (the step \( 3 \)), 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, \( \text{Init}() \) is used to initialize GraphM by preprocessing the graph as described in Section 3. \( \text{Sharing}() \) 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 \( \text{Sharing}() \) is various for different graph processing systems, e.g., the parameter is the function \( \text{Load}() \) for GridGraph and is the function \( \text{LoadSubgraph}() \) for GraphChi.

---

**Figure 5: System architecture of GraphM**

**Table 1: GraphM Programming Interface**

<table>
<thead>
<tr>
<th>APIs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{Init}() )</td>
<td>Initialization of GraphM</td>
</tr>
<tr>
<td>( \text{GetActiveVertices}() )</td>
<td>Get active vertices in each iteration</td>
</tr>
<tr>
<td>( \text{Sharing}() )</td>
<td>Load the shared graph data</td>
</tr>
<tr>
<td>( \text{Start}() / \text{Barrier}() )</td>
<td>Notify GraphM to start or end fine-grained synchronization</td>
</tr>
</tbody>
</table>
Algorithm 1 Partition Labelling Algorithm

1. function LABEL($P^i$, $Set^i_j$)
2.  $edge_num \leftarrow 0$
3.  $c_{table} \leftarrow \emptyset$
4.  for each edge $e \in P^i$ do
5.     if $e_i \in c_{table}$ then
6.       $c_{table}.N^+(e_i) \leftarrow c_{table}.N^+(e_i) + 1$
7.     else
8.       $c_{table}.InsertEntry((e_i, 1))$
9.     end if
10.    if $edge_num \times \frac{S_G}{S_c} \geq S_c$ or $P^i$ is visited then
11.       $Set^i_j(\text{Store}(c_{table}))$
12.       /*Prepare to store information of next chunk*/
13.       $Clear(c_{table}, edge_num)$
14.    end if
15.  end for
16. end function

Figure 6: An example to illustrate how to integrate GraphM into existing graph processing systems

Meanwhile, two notification functions (i.e., $\text{Start}()$ and $\text{Barrier}()$) 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 $\text{GetActiveVertices}()$ 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 $\text{Load}(G)$ 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 chunk 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 chunk 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 $C_{LLC} \times N$ 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 side 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 the cache line for better locality.

$$S_c \times N + \frac{S_c \times N}{S_G} \times |V| \times U_V + r \leq C_{LLC} \quad (1)$$

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 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 $\text{chunk.table}$. Each entry of the $\text{chunk.table}$ 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., $e_i$) is incremented by one in the corresponding entry of $\text{chunk.table}$ (i.e., $e_{table}$) (Line 6). If the source vertex of the edge is not found in $e_{table}$, 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 $e_{table}$ 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 $e_{table}$ is stored as an element of an array, i.e., $Set^i_j$ (it holds the information of all chunks in $P^i$) (Line 12). $e_{table}$ 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 $\text{should_access_shard}$ 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 Pi that needs to be loaded*/
3: Pi ← GetActivePartition()
4: /*Get the set of jobs that need to handle Pi*/
5: J' ← GetJobs(Pi)
6: Resume(J') /*Resume the suspended jobs in J'*/
7: if j ∉ J' then /*j does not need to handle Pi*/
8: Suspend(j) /*Suspend the job j*/
9: end if
10: if Pi is not in the memory then
11: /*Create a shared buffer to store Pi*/
12: Buf ← CreateMemory(G, Pi)
13: else /*Job j gets Pi in the shared buffer*/
14: Buf ← Load(Pi) /*Load Pi into Buf*/
15: end if
16: if Pi is not in the memory then
17: else /*Job j gets Pi in the shared buffer*/
18: Buf ← Attach(G) /*Attach Buf to j*/
19: end if
20: Remove(j, J') /*Remove j from J'*/
21: return Buf
22: end function

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 Pi 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') that need to handle Pi are then obtained from the global table (Line 3). After that, the active jobs recorded in J' are copied 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.e., j does not need to handle Pi), it is suspended to use computing resources for the execution of the jobs that need to handle Pi, and is pushed into suspended_queue (Lines 5-7). If Pi 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 Pi in memory with the other concurrent jobs (Line 12). Then, the PIDs of the job j is removed from the J' (Line 14) and the job j begins to handle the graph partition Pi 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 Set2 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 Set2 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 \( T(E) \). In addition, the average data access time for each edge, indicated as \( T(V) \), 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., \( P^j \)) and captures its execution time, denoted by \( T^j_1 \), which is composed of the graph processing time and the graph data access time. Thus, \( T^j_1 \) is represented as the following formula:

\[
T(F_j) \times \sum_{k \in C^j} \sum_{v \in V_k}^+ N^+_k(v) + T(E) \times \sum_{k \in C^j} \sum_{v \in V_k} N^-_k(v) = T^j_1
\]

(2)

where \( C^j \) is the set of chunks in the partition \( P^j \), and \( V_k \) is the set of vertices in the \( k \)th 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 \( k \)th chunk. \( V_k \) and \( N^+_k(v) \) are stored in the corresponding Set\(^+_k\). 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 \( j \)th job for the processing the \( k \)th chunk (i.e., \( L^k_j \)) can be determined by the following equation:

\[
L^k_j = T(F_j) \times \sum_{v \in V_k \cap A_j} N^+_k(v)
\]

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 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 \( t^j_k \) for the \( k \)th chunk can be obtained by the following equation:

\[
f^j_k = L^j_k \times T(E) \times \sum_{v \in V_k} N^+_k(v)
\]

(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
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 \( \text{Pri}(P^j) \) of each partition \( P^j \) is set using Equation 5, where \( f^j \) denotes the set of jobs to handle \( P^j \) in the next iteration, \( N_j(P) \) denotes the number of active partitions of the \( j \)-th job (i.e., a job of the set \( f^j \)), and \( N(N^j) \) denotes the number of jobs in the set \( N^j \).

\[
\text{Pri}(P^j) = \max_{j \in f^j} \frac{1}{N_j(P)} \times N(N^j) 
\]

The values of \( N_j(P) \) and \( N(N^j) \) 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>
<thead>
<tr>
<th>Datasets</th>
<th>Vertices</th>
<th>Edges</th>
<th>Data sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LiveJ [5]</td>
<td>4.8 M</td>
<td>69 M</td>
<td>526 MB</td>
</tr>
<tr>
<td>Orkut [5]</td>
<td>3.1 M</td>
<td>117.2 M</td>
<td>894 MB</td>
</tr>
<tr>
<td>Twitter [23]</td>
<td>41.7 M</td>
<td>1.5 B</td>
<td>10.9 GB</td>
</tr>
<tr>
<td>UK-union [9]</td>
<td>133.6 M</td>
<td>5.5 B</td>
<td>40.1 GB</td>
</tr>
<tr>
<td>Clueweb12 [4]</td>
<td>978.4 M</td>
<td>42.6 B</td>
<td>317 GB</td>
</tr>
</tbody>
</table>

Table 2: Graph datasets used in the experiments

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...
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 GridM 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 43.3% and 43.3% for GridGraph-S and GridGraph-C, respectively. It is mainly because
The results show that higher speedup is obtained by GraphM when the jobs is more frequently submitted (i.e., larger \( \lambda \)), especially for the real trace shown in Figure 2, where different schemes, where the experiments for different schemes of PowerGraph and Chaos are done on a cluster with 128 nodes.
as described in Section 5.1. We can observe that all systems get faster 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 logic 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

We would like to thank our shepherd and anonymous reviewers for their constructive comments. Yu Zhang (zyhu@hust.edu.cn) is the corresponding author of this paper. This paper is supported by National Key Research and Development Program of China under grant No. 2018YFB1003500, National Natural Science Foundation of China under grant No. 61832006, 61825202, 61702022, and 61929103.

REFERENCES


GraphM: An Efficient Storage System for Concurrent Graph Processing


