CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing

Yu Zhang, Xiaofei Liao, Hai Jin, and Lin Gu, ***Huazhong University of Science and Technology;***
Ligang He, ***University of Warwick;*** **Bingsheng He,** ***National University of Singapore;***
Haikun Liu, ***Huazhong University of Science and Technology***

https://www.usenix.org/conference/atc18/presentation/zhang-yu

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the
2018 USENIX Annual Technical Conference
is sponsored by USENIX.

CGraph: A Correlations-aware Approach for Efficient Concurrent Iterative Graph Processing

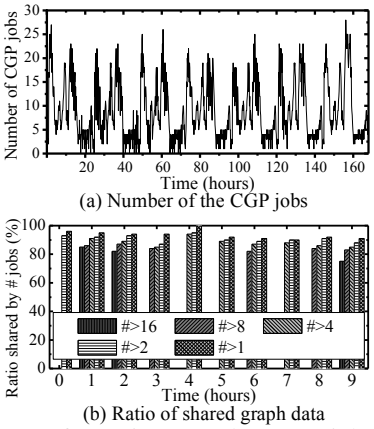
Yu Zhang[†] *Xiaofei Liao*[†*] *Hai Jin*[†] *Lin Gu*[†] *Ligang He*[§] *Bingsheng He*[‡] *Haikun Liu*[†]
[†] *Services Computing Technology and System Lab,*
Big Data Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[§]*Department of Computer Science, University of Warwick, UK*
[‡]*Department of Computer Science, National University of Singapore, Singapore*
{*zhyu, xfliao, hjin, lingu, hkliu*}*@hust.edu.cn* *ligang.he@warwick.ac.uk* *hebs@comp.nus.edu.sg*

Abstract

With the fast growing of iterative graph analysis applications, the graph processing platform has to efficiently handle massive *Concurrent iterative Graph Processing* (CGP) jobs. Although it has been extensively studied to optimize the execution of a single job, existing solutions face high ratio of data access cost to computation for the CGP jobs due to significant cache interference and memory wall, which incurs low throughput. In this work, we observed that there are strong spatial and temporal correlations among the data accesses issued by different CGP jobs because these concurrently running jobs usually need to repeatedly traverse the shared graph structure for the iterative processing of each vertex. Based on this observation, this paper proposes a correlations-aware execution model, together with a core-subgraph based scheduling algorithm, to enable these CGP jobs to efficiently share the graph structure data in cache/memory and their accesses by fully exploiting such correlations. It is able to achieve the efficient execution of the CGP jobs by effectively reducing their average ratio of data access cost to computation and therefore delivers a much higher throughput. In order to demonstrate the efficiency of the proposed approaches, a system called CGraph is developed and extensive experiments have been conducted. The experimental results show that CGraph improves the throughput of the CGP jobs by up to 2.31 times in comparison with the existing solutions.

1 Introduction

In the past decade, iterative graph analysis has become increasingly important in a large variety of domains [7, 26], which need to iteratively handle the graph round by round until convergence. Due to the increasing need of analyzing graph-structured data (e.g., social networks and web graphs), many iterative graph algorithms run as concurrent services on a common platform. These



(a) Number of the CGP jobs
(b) Ratio of shared graph data

Figure 1: Information traced on a social network

Concurrent iterative Graph Processing (CGP) jobs are usually executed on the same graph simultaneously so as to analyze it for various information. For example, facebook [2] uses Giraph [13] to handle a large number of CGP jobs (such as the variants of pagerank [21], SSSP [20], and *k*-means [16]) daily over the same graph (or its different snapshots) to provide various information for different products, respectively. Figure 1(a) gives the number of the CGP jobs traced over a large social network and shows that more than 20 CGP jobs may be submitted to the common platform to concurrently analyze the same graph in an iterative way at the peak time.

Many systems are recently proposed to support large-scale graph analytics. They try to fully utilize high sequential memory bandwidth [17, 22, 23], improve data locality [11, 28, 31, 32], spare the redundant data accesses [6, 25], and reduce the memory consumption [29, 30], etc. Despite of these research efforts, there is a major challenge for the efficient execution of the CGP jobs. When a massive number of CGP jobs are running on the same underlying graph using the existing systems, each individual CGP job separately accesses the shared graph structure along different graph paths, resulting in repeatedly loading of the same data into the cache at different time by different jobs. They suffer from expensive data

*Corresponding Author: Xiaofei Liao

access overhead due to the factors such as serious cache interference and limited bandwidth. As the result of high ratio of data access cost to computation in graph algorithm, the current graph processing systems experience low throughput. This paper investigates whether and how we can improve the throughput of the CGP jobs.

In practice, the CGP jobs usually need to repeatedly traverse the shared graph and iteratively process each vertex for their own purpose. It suggests that there are a large number of intersections among the graph structure data being accessed by these jobs in each iteration, which we call the *spatial correlation* of data accesses. In addition, the partition of the shared graph structure may need to be accessed by multiple jobs within a short time interval, which we call the *temporal correlation* of data accesses. These two correlations indicate that there exist significant redundant data storing and accessing cost in the jobs, which leaves us good opportunities to reduce these unnecessary costs and improve the throughput.

Based on the observation, we propose a data-centric *Load-Trigger-Pushing* (denoted by LTP) model to improve the throughput of CGP jobs by fully exploiting the correlations of their data accesses. It decouples the graph structure data from the vertex state associated with each job. Within each iteration, the graph structure partitions shared by multiple CGP jobs are streamed into the cache and trigger the related jobs to concurrently process the data, followed by vertex state pushing for convergence. In this way, many accesses to the shared graph partitions can be amortized by multiple CGP jobs through handling them along a common order. The consumption of cache/memory is also reduced since a single copy of the graph structure data is used to serve multiple jobs at the same time. It indicates higher throughput thanks to much lower data access cost. To further improve the throughput, a core-subgraph based scheduling algorithm is designed to maximize cache utilization by judiciously arranging the loading order of the partitions.

We conducted the extensive experiments with our system CGraph and compare its performance with those of three cutting-edge graph processing systems, i.e., CLIP [6], Nxgraph [11], and Seraph [29, 30]. Experimental results show that CGraph improves the throughput of the CGP jobs by up to 3.29 times, 4.32 times, and 2.31 times over CLIP, Nxgraph, and Seraph, respectively.

The remainder of this paper is organized as follows. Section 2 discusses the the motivation of this work. Section 3 outlines our approach, followed by experimental evaluation in Section 4. Section 5 gives a survey of related work. Finally, we conclude this paper in Section 6.

2 Problem Presentation and Motivation

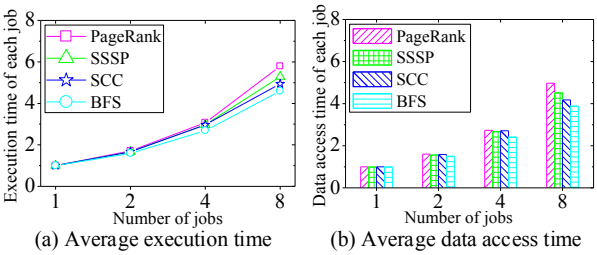
A common characteristic of an iterative graph processing job is that the operations are usually operated on two types of data: graph structure data and vertex state data. The graph structure data contains the edges between vertices and the information associated with each edge, whereas the vertex state data (e.g., ranking scores for PageRank [21], the distances from the source vertex for SSSP [20]) is computed by its tasks in a parallel way within each iteration and typically consumed in the next iteration. The graph structure data always occupies a majority of the memory, as compared with the vertex state data (i.e., job-specific data), and its proportion varies from 71% to 83% for different datasets [30]. As evaluated in Figure 1, the graph structure data is usually shared by multiple CGP jobs. However, in existing graph processing systems, these CGP jobs handle the shared graph in an individual manner along different graph paths, incurring low throughput for many redundant accesses to the shared graph and cache interference.

2.1 Data Access Problems of the CGP Jobs

In order to investigate the level of the inefficiency of the individual data accessing manner of the CGP jobs, we conducted the benchmark experiments to evaluate the execution time of different number of CGP jobs over Seraph [29, 30] on uk-union [3]. The hardware platform and benchmarks are the same as those described in Section 4.

From Figure 2(a), we made two observations. First, the concurrent way performs better than the sequential way of executing the jobs one by one. As observed in the experiments, it is because that the execution time of graph processing job is dominated by the data access cost and the CPU is always underutilized. Seraph is able to utilize the CPU in a better way by concurrently executing the jobs and also allowing them to share the in-memory graph structure data for less average data access cost. When there are eight jobs, the total execution time of the concurrent execution way is about 60% of the sequential way. Note that the total execution time of the concurrent way is the maximum value of these jobs' execution time, while it is the sum of those of all jobs for the sequential way. Second, the average execution time of each job, however, is significantly prolonged as the number of jobs increases. It is almost doubled when the number of jobs increases from four to eight, because of higher data access cost for each vertex processing.

Figure 2(b) shows the average data access time of the jobs over Seraph when the number of jobs increases. We can observe that the increment of the number of jobs leads to the significant rise of data access cost. It is because that the shared graph partitions are handled by the CGP jobs in an individual manner along different graph paths. As the number of the CGP jobs increases, more copies of the same data need to be created and loaded into the cache by the jobs at different time. Thus, more redundant data accesses are issued by the CGP jobs and



(a) Average execution time (b) Average data access time

Figure 2: Normalized performance of each CGP job over Seraph against that of the sequential way

it also incurs more serious cache interference due to the fact that more redundant data are stored into the cache for different jobs at different time. It eventually leads to low system throughput, since the data accessing cost typically represents a major proportion of the total execution time for an iterative graph algorithm.

Take Figure 3 as an example and assume that the cache can only store a partition for the CGP jobs. With the existing solutions, the SSSP job may firstly access partition 1 and then partition 2, whereas the PageRank job may firstly access partition 2 and then partition 1. Besides, the processing of each partition is various for different jobs, making their accesses more irregular. As a result, the partition 1 and partition 2 need to be repeatedly loaded into the cache. It leads to serious contention among the jobs for the data access channel, the cache and so on.

2.2 Correlations between the CGP Jobs

It is common that a set of CGP jobs involve in the analysis of the same graph. Figure 1(b) shows the ratios of a graph shared by different number of CGP jobs at various time sampled from the real trace. We discover that there are strong temporal and spatial correlations between the data accesses of the CGP jobs due to the repeated traverse of the graph shared by them. It indicates that many redundant accesses are issued by the CGP jobs and much cache space is also wasted to store several copies of the same graph structure data for the jobs at different time.

As described in Figure 1(b), the intersections of the set of graph partitions to be handled by different CGP jobs in each iteration are large (more than 75% of all active partitions on average). This is called the *spatial correlation*. However, the CGP jobs in existing systems access the shared graph partitions in different order individually, inducing much redundant overhead. Ideally, these CGP jobs should consolidate the accesses to the shared graph structure and store a single copy of the shared data in the cache to serve multiple CGP jobs at the same time.

In addition, some graph partitions may be accessed by multiple CGP jobs (may be more than 16 jobs) within a short time duration. This is called the *temporal correlation*. The traced results show that the number of CGP jobs to access each partition is skewed at any time. The current solutions, e.g., *Least Recently Used* (LRU) algorithm, may load the infrequently-used data into the cache when it is needed. It not only incurs the cost to load the data, but also swaps out the frequently-used data. A better solution should take into account the temporal correlations, e.g., the usage frequency of the graph partitions, when loading them into the cache.

These observations motivate us to develop a solution for efficient use of cache/memory and the data access channel to achieve a higher throughput by fully exploiting the spatial/temporal correlations discussed above.

3 Our Proposed Approach

Although we have identified the correlations between the CGP jobs, there are still several challenges that need to be tackled in order to exploit them. First, the shared vertices and edges may be individually handled by different jobs along different graph paths. Second, the CGP jobs have different properties (e.g., the rounds for convergence and the submission time), which reduce the chance of sharing the accesses to the graph structure data within a short time interval. For example, some graph structure partitions may be accessed by some jobs (e.g., PageRank) much more frequently than the others (e.g., BFS). Besides, the CGP jobs that share the same graph structure may be put into execution at different time. Third, it is a non-trivial task to design an efficient partition-loading order that can achieve a high cache utilization ratio.

Thus, we propose a data-centric *Loading-Trigger-Pushing* (LTP) model to fully exploit the spatial/temporal correlations between the CGP jobs, aiming to minimize the redundant accessing and storing cost of the shared graph structure data. In our LTP model, the shared graph is divided into a set of partitions. These partitions are loaded into the cache in sequence and in the same order for all jobs, where each partition is concurrently handled by the related CGP jobs. By such means, the accessing and storing of most graph structure partitions can be shared by multiple CGP jobs, thus significantly reducing the data access cost. When loading the graph partitions, a scheduling algorithm is further developed to specify the loading order of graph partitions (as well as the related job-specific data). The scheduler aims to maximize the cache utilization by fully exploiting the temporal correlations among the jobs' data accesses.

3.1 Data-centric LTP Execution Model

Assume that the data for an iterative graph algorithm is expressed as $D = (V, S, E, W)$, where V is the set of vertices, S is the set of states for the vertices, E is the set of edges, W is the set of weights associated with the edges. In our LTP model, the data of each job is decoupled as the graph structure data, i.e., $G = (V, E, W)$, and job-specific vertex states, i.e., S, where $G = \cup_i G^i$ is shared by different jobs and G^i is the i^{th} partition of the graph G. Each

job has its own S, and repeatedly updates its S through its processing iterations until the calculated results converge. The processing of each iteration is divided into three stages: graph loading, parallel trigger, and pushing stage, which are formalized as follows.

Graph Loading. In each iteration, the shared graph structure partitions, e.g., G^i, are sequentially loaded for the CGP jobs along an order. It performs the following operation to load a graph partition: $L^i \leftarrow L(G^i, \cup_{j\in J} S^i_j)$, where $L(*)$ denotes an operator that loads the data specified in the parameter list "*" into the cache, J is the job set, S^i_j denotes the states of the vertices in G^i associated with the j^{th} job, and L^i is the data loaded into the cache. $S_j = \cup_i S^i_j$ is the set of vertex states related to the j^{th} job. In this way, it only needs to load a copy of each shared graph partition, e.g., G^i, for multiple CGP jobs and the partitions are also loaded for these jobs along a common order to provide opportunity to the sparing of redundant accesses by fully utilizing the correlations of these jobs.

Trigger and Parallel Execution. For each loaded graph partition G^i, the related CGP jobs, which are the jobs that need to process the vertices in the partition G^i and have not yet obtained the convergent results, are triggered to concurrently execute the following operator: $S^{i_{new}} \leftarrow \cup_{j\in J} T_j(G^i, S^i_j)$. The function $T_j(G^i, S^i_j)$ denotes the specific graph processing operations performed by the activated job j on the loaded data (i.e., G^i and S^i_j) towards its own objectives. Its outputs (denoted by $S^{i_{new}}_j$) are the new states related to the vertices in G^i and are associated with the j^{th} job. $S^{i_{new}} = \cup_{j\in J} S^{i_{new}}_j$ is the new vertex states that are related to the vertices in G^i for all CGP jobs. When the processing of G^i is finished for all related jobs, the next partition then can be loaded. By such means, it enables multiple jobs to regularly and concurrently process the set of shared graph partitions for their own goals along the same order and efficiently share the accesses to them for lower overhead.

State Pushing. If a job, e.g., j, has processed all its active partitions in an iteration, its new calculated results, i.e., $S^{new}_j = \cup_i S^{i_{new}}_j$, at this iteration are pushed for the state synchronization between the vertices of its different partitions (stored in its own job-specific space) for convergence. Then, the job starts a new iteration. Note that a CGP job will move to the next iteration once it has processed all active partitions in its current iteration and therefore different CGP jobs may be in different iterations of their graph processing. For example, BFS [10] may only need to handle a few active partitions in each iteration, while other algorithms, e.g., PageRank [21], may have to go through all partitions to complete an iteration.

Figure 3 gives an example to illustrate the LTP model. In this example, the graph in Figure 4(a) is divided into two partitions, which need to be handled by two jobs, i.e., a PageRank job and a SSSP job. The graph structure data is stored in the global space and is shared by these two CGP jobs, while the job-specific space is provided for each CGP job to store its own vertex states. It can load the two partitions along the order of partition 1 then 2. When the partition 1 and the related job-specific data are loaded into the cache, the related jobs (i.e., the PageRank job and the SSSP job) are triggered to concurrently handle it and update their own vertex states. When the two jobs have handled the partition 1, the partition 2 can be loaded for processing. When both the two partitions are handled by the jobs, the new iteration of each job begins.

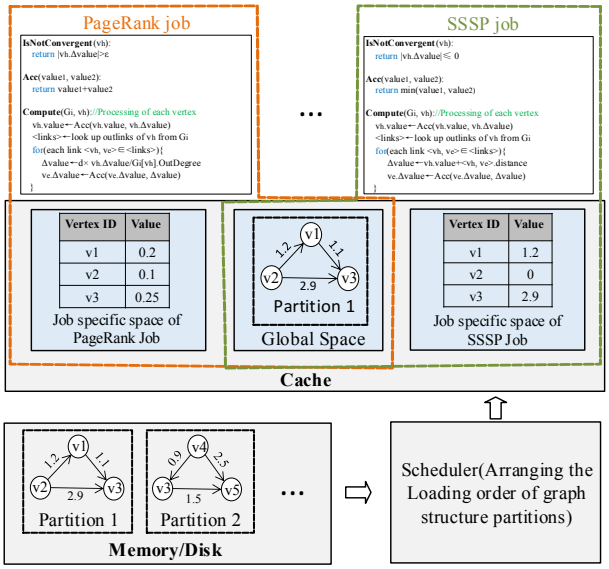


Figure 3: Illustration of our data-centric LTP model

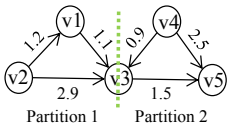
3.2 Correlations-aware Execution of Jobs

This section discusses how to efficiently implement our LTP model for the execution of multiple CGP jobs.

3.2.1 Graph Storage for Multiple CGP Jobs

We first show how to efficiently store the graph for the CGP jobs in our approach.

Data Structure of Graph Partition. For parallel processing, large-scale graph needs to be divided into partitions. However, the real-world graph usually has highly skewed power-law degree distributions [12], incurring imbalanced load among the partitions. Thus, our system also uses existing vertex-cut partitioning method [31], and evenly divides the edges of the graph into same-sized partitions in terms of the number of edges. Note that a vertex may have multiple replicas (e.g., v_3 in Figure 4(b)), where one of the replicas is nominated as the master vertex and the others are regarded as the mirror vertices. In this way, it not only gets balanced load for the partitions, but also does not incur communication cost when handling each partition. The communication only occurs when the replicas of the same vertex in different



(a) Example graph

PageRank Job

Vertex ID	Value	Vertex ID	Value
v1	0.2	v3	0.05
v2	0.1	v4	0.1
v3	0.25	v5	0.3

Private Table Partitions

SSSP Job

Vertex ID	Value	Vertex ID	Value
v1	1.2	v3	∞
v2	0	v4	∞
v3	2.9	v5	∞

Private Table Partitions

Vertex ID	Edge List	Flag	Master Location	Information Associated with Its Edges
v1	v3	Master	Partition 1	1.1
v2	v1, v3	Master	Partition 1	1.2, 2.9
v3	Ø	Master	Partition 1	Ø

Vertex ID	Edge List	Flag	Master Location	Information Associated with Its Edges
v3	v5	Mirror	Partition 1	1.5
v4	v3, v5	Master	Partition 2	0.9, 2.5
v5	Ø	Master	Partition 2	Ø

Graph Structure Partitions

(b) Details for the related tables

Figure 4: An example to show how to store data for multiple jobs, where the graph is divided into two partitions.

partitions synchronize their states. In order to effectively store the graph partitions for the CGP jobs, multiple key-value tables are established. In detail, a single global table is created to store the graph structure data for all CGP jobs. Multiple private tables are used to store the vertex states of the jobs, i.e., one table for each job.

Each global table entry represents a graph structure partition indexed by its key and with three other fields to describe corresponding information. The first two fields indicate the location of this graph structure partition and the number of its vertices, respectively. The third field stores the IDs of the jobs to process it at the next iteration (along with the locations of the related private tables associated with these jobs). The information of each graph structure partition is also stored in a key-value table and each of its data item indicates a vertex and contains four fields: vertex ID, edges assigned in this partition, flag, master location and the information associated with its edges, e.g., priority. Each private table entry represents a vertex state and has two fields, i.e., vertex ID and its state. Figure 4 gives an example to illustrate how to store the data for multiple jobs.

Suitable Size of Graph Partition. In order to efficiently use the parallelism of CPU and ensure good cache locality, the cache is expected to be just fully loaded when each core has data to handle. Therefore, the suitable size of each graph structure partition, i.e., P_g, is determined by the number of CPU cores, i.e., N, and the size of the cache, i.e., C. The value of P_g is expected to be the maximum integer such that $P_g + \frac{P_g}{s_g} \times s_p \times N + b \leq C$, where

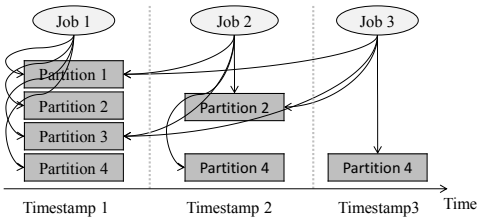


Figure 5: An example to illustrate how to store evolving graph structure for the CGP jobs submitted at different timestamps, where partitions 2 and 4 are changed at timestamp 2, and partition 4 is changed at timestamp 3.

s_g is the average size of each graph structure partition's item, s_p is the size of each private table's item, and b is the size of reserved buffer.

Details to Store Evolving Graph Structure. In practice, the graph structure may evolve with time. Thus, we also maintain a series of snapshots for it, where the graph updates, e.g., the adding/deleting of vertices and edges, are only visible to the jobs submitted later than the updates. In this way, different jobs are able to correctly handle the related snapshots of the graph, respectively. Because the changes of graph structure are usually very small at each time, the most part of these snapshots is the same. Thus, the series of snapshots can be stored in an incremental way for low overhead. For each snapshot, it creates a new global table and labels it with a timestamp, where this table only stores the new version of the partition with changes. The newly submitted job handles the graph partitions with the highest timestamp yet less than its arrival time. Figure 5 gives an example to illustrate it. Note that most graph structure partitions, e.g., the partitions 1 and 3, are usually shared by the jobs when they handle different snapshots, respectively.

3.2.2 Loading of Graph Partition

In practice, a partition is to be handled by a job in the next iteration only when its vertices are activated by the other ones of this job at the current iteration. Therefore, it is easy for each partition to identify the set of CGP jobs to process it within the next iteration through tracing the partitions activated within the current iteration. This profiled information, i.e., the temporal correlations of the jobs, is stored in the global table for each partition. The spatial correlations between the data accesses issued by the CGP jobs can be gotten by calculating the intersection of the set of graph partitions to be processed by different jobs. After that, it is able to load the shared graph partitions for exactly once along a common order to serve multiple CGP jobs within each iteration, amortizing the data access cost. Note that the correctness will not be affected by any loading order and the runtime loads the partitions in a round-robin way by default.

For each job, the states of most vertices may have converged at the early iterations, although some vertices

Algorithm 1 Details of each trigger

```
1: procedure TRIGGER(G^i, S^i_j)
2:     for each v_h ∈ S^i_j ∧ IsNotConvergent(v_h) do
3:         Compute(G^i, v_h)
4:         if v_h is a mirror vertex then
5:             D ← v_h.MasterLocation
6:             S^new_j.Insert(v_h, i, D, v_h.Δvalue)
7:         end if
8:     end for
9: end procedure
```

need hundreds of iterations for convergence. The loading and processing of the inactive vertices can be skipped for the related job for low overhead. In detail, when a graph structure partition G^i is loaded into the cache, it only loads the related job-specific private partitions, e.g., S^i_j, of the CGP jobs which need to process G^i. It does not load G^i when there is no job to handle G^i, i.e., the states of the vertices in G^i are inactive for all jobs. Specifically, when a graph structure partition is not used by any job at the next iteration, this graph structure partition is labeled as an inactive one so as to skip its loading. Similarly, it is relabeled as an active one when it needs to be processed by some jobs at the next iteration.

3.2.3 Parallel Processing of Graph Partition

After loading a graph partition G^i into the cache, it triggers the related CGP jobs (e.g., j) to concurrently handle their private vertex states (e.g., S^i_j) associated with this partition, respectively. Note that any newly submitted job only needs to register the partitions to be processed by it at its first iteration and waits to be triggered to handle them. It is possible that the number of CGP jobs is more than the number of CPU cores, i.e., N. Assume a partition is shared by $|J|$ number of jobs. When the value of $|J|$ is larger than N, these CGP jobs are assigned to be processed as different batches, where the shared graph structure partition is fixed in the cache and only the job-specific partitions are replaced. A graph structure partition is swapped out of the cache only when it has been processed by the related jobs within the current iteration. Otherwise, the unprocessed jobs need to load it again.

For the processing of each partition, the computation load of different CGP jobs is usually skewed, leading to low utilization ratio of hardware. In order to tackle this problem, it identifies the straggler, i.e., the job with the most number of unprocessed vertices in its private table for this partition. Note that the number of unprocessed vertices can be easily gotten, because the number of active vertices for each job in each partition is known as this partition is handled by the jobs at the previous iteration. Then, as described in Figure 6, it logically divides the unprocessed vertices in the private partition of the straggler into pieces and assigns them to the free cores to assist its processing.

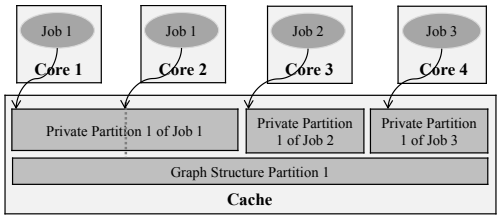


Figure 6: An example to illustrate how to get balanced load, where the core 1 and the core 2 are handling the partition 1 of the private table of the job 1 together.

The processing details for a job are given in Algorithm 1, where each job only computes the new state for its vertices in S^i_j according to their local neighbors recorded in the graph structure partition G^i (See Lines 2-8). Therefore, there is no cache miss when handling a partition, because no communication occurs between the vertices on different partitions. Obviously, the vertex with replicas on different partitions needs to synchronize their states. The mirror vertex needs to push its new state to its master vertex to get this vertex's final state at the current iteration. The new calculated state on the master vertex needs to be pushed to its mirrors. As a result, for such a vertex state synchronization, many partitions of private table are frequently loaded into the cache and incur high cache miss rate. In order to tackle this problem, for each mirror vertex, its new state is directly buffered in S^{new}_j (See Line 6), which will be implicitly sent to the master replica for batched vertex state synchronization at the data synchronization stage.

3.2.4 Data Synchronization

When there are multiple CGP jobs to synchronize vertex state, it is done for the jobs one by one to reduce resource contention, because there is no data sharing between the jobs. For efficient vertex state synchronization among replicas, as depicted in Algorithm 2, they are done together in batches at this stage for each job, aiming to avoid the frequent load of private table's partitions at runtime. The items buffered in the queue S^{new}_j (with the new states of the mirror vertices, e.g., v_h) are firstly sorted according to the IDs (e.g., $v_h.MasterLocation$, which is described in Figure 4(b)) of the partitions with the related master vertices (See Line 2), before pushing them.

By such means, it only needs to load fewer partitions of private table for the state updates of master vertices, since many updates become successive accesses to the same partition. Besides, when the successive updates for a master vertex are done (See Line 7), the final state of this vertex for the current iteration is gotten. Then, such a new value can be directly buffered for batched state updating of mirror vertices as well (See Lines 10-12). Note that, with the traditional solutions, it is impossible to

Algorithm 2 Details of data synchronization

```
1:  procedure PUSH(j, S_j^new)
2:      SortD(S_j^new) /*Sort the items recorded in S_j^new*/
3:      for <v_h, i, MasterLocation, Δvalue> ∈ S_j^new do
4:          D ← S_j^new[v_h].MasterLocation
5:          value ← S_j^new[v_h].Δvalue
6:          S_j^D[v_h].Δvalue ← Acc(S_j^D[v_h].Δvalue, value)
7:          if Last update of S_j^D[v_h].Δvalue is end then
8:              val ← S_j^D[v_h].value
9:              S_j^D[v_h].value← Acc(val, S_j^D[v_h].Δvalue)
10:             for each S_j^new[v_h].MasterLocation=D do
11:                 S_j^new[v_h].Δvalue ← S_j^D[v_h].Δvalue
12:             end for
13:         end if
14:     end for
15:     SortS(S_j^new) /*Sort the items recorded in S_j^new*/
16:     for <v_h, i, MasterLocation, Δvalue> ∈ S_j^new do
17:         i ← S_j^new[v_h].i
18:         S_j^i[v_h].Δvalue ← S_j^new[v_h].Δvalue
19:     end for
20: end procedure
```

know whether the final state is gotten for a master vertex until all updates are done. Then, the master vertex needs overhead to be reloaded for accessing, because it may have been swapped out of the cache. After that, it is done in a similar way to update mirror vertices' states according to the related master vertices' states (See Lines 15-19), where the items are sorted according to the IDs of the partitions with the mirror vertices (See Line 15).

3.3 Scheduling Based on Core-subgraph

With existing solutions, the partitions loaded into the cache may be underutilized. First, some vertices need more iterations to converge than the others for much higher degree. They make the partitions containing them repeatedly loaded into the cache and incur high overhead to load and store the early convergent vertices in the same partition. Second, the usage frequency of different partitions is also skewed and also evolving with time. In detail, the same partition of different jobs and different partitions in the same job all may need various iterations to converge. Besides, a graph partition is only visible to the jobs with the arrival time later than its timestamp. As a result, a loaded partition may need to be processed by very few (even one) jobs when the partition is arbitrarily loaded into the cache, inducing poor performance.

In order to maximize the utilization ratio of each partition loaded into the cache, we propose a scheduling algorithm based on core-subgraph partitioning. The key idea is to firstly put the core vertices (with degree higher than a given threshold) together and then make the loaded partition shared by as many jobs as possible on average via arranging the loading order of graph partitions. In detail, it firstly identifies a core subgraph, consisting of the core vertices and the edges on the paths between them, from the graph. Then it evenly divides the graph based on such a subgraph, where the edges of this subgraph are put together into several same-sized partitions and the remaining edges are divided into the other same-sized partitions. By such means, the frequently loading and processing of core vertices incur less cost to load the early convergent vertices in the same partition, sparing the consumption of bandwidth and the cache space.

After that, it gives each partition P a priority $Pri(P)$ and schedules the loading order of them based on the dynamically profiled priorities of them. The partition with the highest priority is firstly loaded into cache for the CGP jobs to handle, so as to improve the cache utilization ratio. The basic scheduling rules are as follows:

- First, a partition should be given the highest priority and be firstly loaded into the cache when it is needed by the most number of jobs for processing.
- Second, a partition should be set with a higher priority when it has a higher average vertex degree or a larger average vertex state changes, because more vertex states will be propagated to others through them. Then, most vertices need less iterations (also less consumption of the cache) to absorb other vertices' states for convergence.

The above rules are captured by such an equation:

$$Pri(P) = N(P) + \theta \cdot \overline{D}(P) \cdot C(P) \quad (1)$$

where $N(P)$ is the number of jobs to process P and is used to capture the temporal correlations for the CGP jobs. $\overline{D}(P)$ is the average degree of the vertices in P, and $C(P)$ is the average state changes of the vertices in P for all its jobs at the previous iteration. The initial values of $N(P)$ and $C(P)$ and the value of $\overline{D}(P)$ are gotten at preprocessing time, while $N(P)$ and $C(P)$ are incrementally updated at the execution time. There, $0 \leq \theta < \frac{1}{\overline{D}_{max} \cdot C_{max}}$ is the scaling factor set by the runtime system at preprocessing time to ensure that a partition with the highest value of $N(P)$ is firstly processed, where $\overline{D}_{max}$ and C_{max} are the maximum values of any partition's $D(P)$ and $C(P)$, respectively. By such means, the partition loaded into the cache can serve as many jobs as possible, while the other partitions have more opportunity to be needed by more jobs after a time interval, further improving the throughput via reducing the average data access cost.

3.4 Implementation and Interfaces

The implementation details of CGraph are described in Algorithm 3. It repeatedly loads the unprocessed partitions, e.g., G^i, of the global table into the cache according to the scheduling algorithm (See Line 4). For each G^i,

Algorithm 3 Executor for CGraph

```
1:  procedure EXECUTOR(G, S_Jobs)
2:      while the job set S_Jobs is not empty do
3:          while G has unprocessed G^i for some jobs do
4:              G^i ← LoadPartition(G) /*Load G^i*/
5:              /*Get the set of jobs to handle G^i*/
                J ← GetJobs(G^i, S_Jobs)
6:              for each j ∈ J do
7:                  /*Trigger the job j to handle G^i*/
                    ParallelTrigger(j, G^i, S^i_j)
8:              end for
9:              for each j ∈ J and S^new_j are gotten do
10:                 /*Vertex state synchronization for j*/
                    Push(j, S^new_j)
11:                 if vertex states of j are inactive then
12:                     /*Remove j from the set S_Jobs*/
                        Remove(S_Jobs, j)
13:                 end if
14:             end for
15:         end while
16:     end while
17: end procedure
```

the job-specific partitions, e.g., S^i_j, of the related CGP jobs are also loaded and these jobs are triggered to concurrently handle the loaded data (See Lines 5-8), where each job calculates the new states of its vertices according to the states of their local neighbors. When all active partitions of G have been handled for a job, e.g., j, at the current iteration (See Line 9), this job synchronizes the states of the vertices with several replicas distributed over different partitions (See Line 10). Then, its new iteration begins. Each job is repeatedly triggered until all its vertex states are inactive (See Lines 11-13). Note that it allows to add new jobs into S_{Jobs} at runtime.

For programming, a user only needs to instantiate three functions, i.e., IsNotConvergent(), Compute(), and Acc(), which are used in existing systems [23, 30, 31]. The first one indicates whether a vertex is convergent. Compute() is employed to update a vertex state and calculate the contributions of a vertex for the new states of its neighbors, and Acc() is utilized for a vertex to accumulate the contributions of its neighbors. Figure 7 gives two examples to show how to implement iterative graph algorithm on CGraph. Within each iteration, each vertex updates its state according to the accumulated contributions of its neighbors. After that, it calculates and sends its contributions to its neighbors for their state updates.

4 Experimental Evaluation

The hardware platform used in our experiments is a server containing 4-way 8-core 2.60 GHz Intel Xeon CPU E5-2670 and each CPU has 20 MB last-level cache,

```
IsNotConvergent(vh):
  return |vh.Δvalue|>ε

Acc(value1, value2):
  return value1+value2

Compute(Gi, vh)://Processing of each vertex
  vh.value←Acc(vh.value, vh.Δvalue)
  <links>←look up outlinks of vh from Gi
  for(each link <vh, ve>∈<links>){
    Δvalue←d× vh.Δvalue/Gi[vh].OutDegree
    ve.Δvalue←Acc(ve.Δvalue, Δvalue)
  }
```

(a) PageRank

```
IsNotConvergent(vh):
  return |vh.Δvalue|⩽ 0

Acc(value1, value2):
  return min(value1, value2)

Compute(Gi, vh)://Processing of each vertex
  vh.value←Acc(vh.value, vh.Δvalue)
  <links>←look up outlinks of vh from Gi
  for(each link <vh, ve>∈<links>){
    Δvalue←vh.value+<vh, ve>.distance
    ve.Δvalue←Acc(ve.Δvalue, Δvalue)
  }
```

(b) SSSP

Figure 7: Instantiation of graph algorithms on CGraph

Table 1: Properties of data sets

Data sets	Vertices	Edges	Sizes
Twitter [3]	41.7 M	1.4 B	17.5 GB
Friendster [4]	65 M	1.8 B	22.7 GB
uk2007 [3]	105.9 M	3.7 B	46.2 GB
uk-union [3]	133.6 M	5.5 B	68.3 GB
hyperlink14 [5]	1.7 B	64.4 B	480.0 GB

running a Linux operation system with kernel version 2.6.32. Its memory is 64 GB and the secondary storage for it is a disk with 1TB. It spawns a worker for each core to run benchmarks. The program is compiled with cmake version 2.6.4 and gcc version 4.7.2.

In experiments, four popular iterative graph algorithms from web applications and data mining are employed as benchmarks: (1) PageRank [21]; (2) *single-source shortest path* (SSSP) [20]; (3) *strongly connected component* (SCC) [14]; (4) *breadth-first search* (BFS) [10]. The datasets used for these graph algorithms are real-world graphs existing on the websites [3, 4, 5] as described in Table 1. The performance of CGraph is compared with three cutting-edge graph processing solutions, i.e., CLIP [6], Nxgraph [11], and Seraph [29, 30], implemented by us on GridGraph [32]. Seraph is the state-of-the-art system optimized to support the efficient execution of multiple CGP jobs. Note that the jobs (e.g., PageRank, SSSP, SCC, and BFS) in the experiments are submitted to each system simultaneously.

4.1 Performance of Scheduling Strategy

First, we discuss the contributions of our scheduling algorithm on the performance of CGraph. In order to get this goal, PageRank, SSSP, SCC and BFS are executed as four CGP jobs to evaluate the total execution time of them over CGraph (with our scheduler described in Section 3.3) and CGraph-without (without our scheduler), respectively. Note that the graph partitions in CGraph-without are loaded in a round-robin way. As shown in Figure 8, the execution time of CGraph-without is more than that of CGraph under any circumstances. The execution time of CGraph is even only 60.5% of CGraph-without over hyperlink14. It is because that the scheduling scheme is able to maximize the utilization ratio of each partition in the cache via firstly loading the most important partition for the jobs.

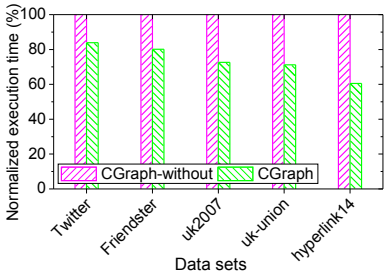


Figure 8: Execution time for the four jobs without/with our scheduler

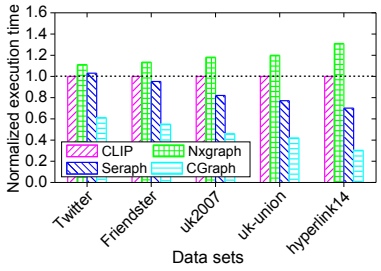


Figure 9: Total execution time for the four jobs with different solutions

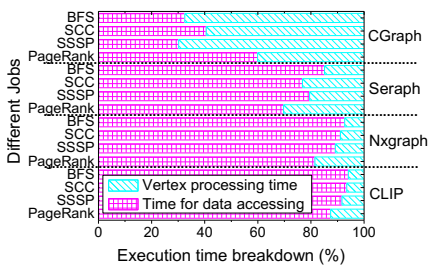


Figure 10: Execution time breakdown of different jobs on hyperlink14

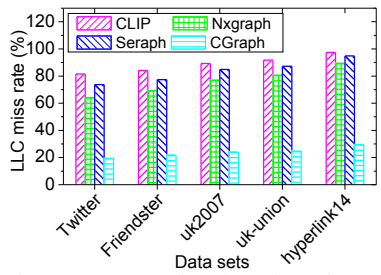


Figure 11: Last-level cache miss rate for the four jobs

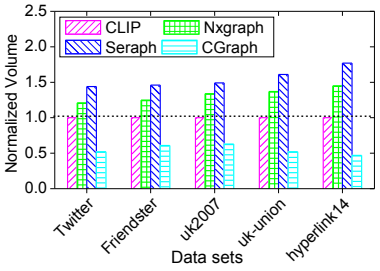


Figure 12: Volume of data swapped into the cache for the four jobs

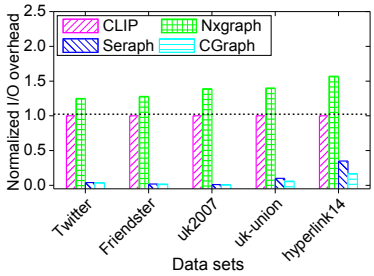


Figure 13: I/O overhead for the four jobs with different solutions

4.2 Overall Performance Comparison

To compare CLIP, Nxgraph, Seraph, and CGraph, we simultaneously submit PageRank, SSSP, SCC, and BFS as four jobs to each of these systems. Figure 9 shows the total execution time of the four jobs over different systems. We find that the four jobs over CGraph are able to converge with less time, which indicates higher throughput than the other systems. For example, over hyperlink14, CGraph can improve the throughput by 2.31 times, 3.29 times, and 4.32 times in comparison with Seraph, CLIP, and Nxgraph, respectively. We identify that the highest throughput of CGraph mainly comes from much lower average data access cost to computation ratio than them.

Figure 10 depicts the execution time breakdown of different jobs evaluated on hyperlink14 with different solutions. We can observe that the pure vertex processing time of the job over CGraph occupies the most ratio of its total execution time, while the ratio is very low in CLIP, Nxgraph, and Seraph. There are two reasons leading to lower average data access cost to computation ratio for CGraph than the other solutions. First, through efficiently exploiting the data access correlations between the CGP jobs, CGraph needs to store less data into the cache, getting a lower cache miss rate. Second, CGraph needs to access less volume of data due to efficient share of data accesses for the jobs, which means less consumption of bandwidth for main memory and the disk.

In order to demonstrate it, we firstly evaluate the last-level cache miss rates of CLIP, Nxgraph, Seraph, and CGraph using Cachegrind [1]. The miss rates of the above four jobs over them are given in Figure 11. As described, the cache miss rate of CLIP is larger than that of Nxgraph, because CLIP tries to trade off locality for the reduction of the total amount of data accesses while Nxgraph uses destination-sorted sub-shard structure to store a graph for better locality. However, the cache miss rate of Nxgraph is still more than that of CGraph. For example, the cache miss rate of Nxgraph is 89.5% for hyperlink14, while the rate is only 29.6% for CGraph. It is mainly because that a single copy of graph structure data in the cache is able to serve multiple jobs of CGraph.

Next, we evaluate the total volume of data swapped into the cache for the above four jobs over different systems. The normalized results of them against CLIP are depicted in Figure 12. We can find that CLIP needs to swap much smaller volume of data into the cache than Nxgraph and Seraph, because it is able to reduce the number of iterations for convergence via reentry of loaded data and beyond-neighborhood accesses. Note that the method employed by CLIP can also be used in Seraph as well as CGraph, rather than Nxgraph.

Besides, from Figure 12, we can observe that the volume of CGraph is much less than those of the other solutions. For example, the value of CGraph is only 47.1% of CLIP over hyperlink14, because CGraph does not need to load and to store the shared graph structure data for each job, separately. However, CLIP suffers from many redundant data accesses due to ignoring the data access correlations between the CGP jobs. Although Seraph can spare some data accesses from the disk to the main memory via sharing in-memory data, each job loads data into the cache in an individual way, incurring high data access overhead as well. It also means that Seraph is only suitable to out-of-core computation.

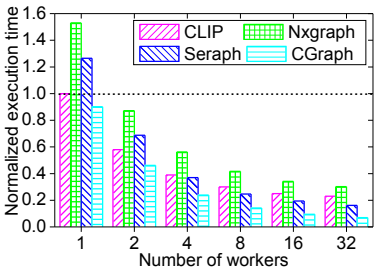


Figure 14: Scalability of the four jobs on hyperlink14

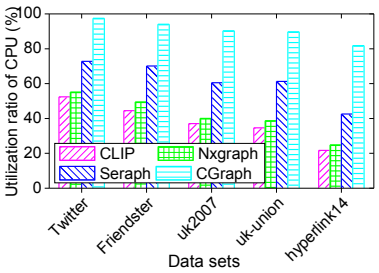


Figure 15: Utilization ratio of CPU for the four jobs

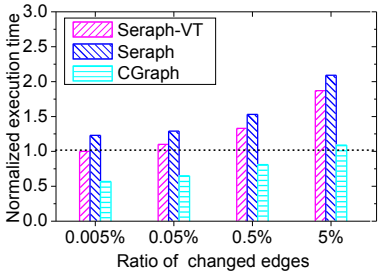


Figure 16: Execution time of the eight jobs on hyperlink14 with changes

Finally, the I/O overhead of the above four jobs is also evaluated over different systems. As shown in Figure 13, the jobs on the first three graphs almost not incur I/O overhead for both CGraph and Seraph, because they only needs to store one copy of the graph structure data in the main memory and these graphs can be totally loaded. When the graph size is larger than the memory size, CGraph needs less I/O overhead than Seraph through consolidating data accesses for the jobs. It also indicates a better performance of CGraph for out-of-core computation, because the data access time dominates the total execution time under such circumstances.

4.3 Scalability of CGraph

The scalability of CGraph is evaluated via executing the above four jobs on hyperlink14 and increasing the number of workers. Figure 14 gives the results relative to that of CLIP with only one worker. We can observe that CGraph has much better scalability than the other ones. The best scalability of CGraph mainly comes from efficient share of data accesses, while the other systems suffer from limited bandwidth for main memory and magnetic disk. Meanwhile, such a limited bandwidth also induces low utilization ratio of CPU for them.

In Figure 15, we evaluate the average utilization ratio of CPU for the vertex processing of the four jobs over different systems. As observed, existing systems suffer from low CPU utilization ratio, because the long data access time leads to the waste of CPU for them. Besides, from Figure 14 and Figure 15, we can find that the CPU cores of CGraph are almost fully utilized due to balanced load and low data access cost to computation ratio. It indicates that the limited computation ability of the CPU cores becomes the bottleneck of CGraph. GPGPU may be a suitable accelerator to help CGraph to process the CGP jobs for its powerful computing ability.

4.4 Performance on Graph with Changes

In real-world applications, several snapshots may be created for the graph with changes, and multiple CGP jobs are generated to handle them, respectively. In this part, we evaluate CGraph for the graph with structure changes. In the following experiments, we repeatedly generate the CGP jobs in the order of PageRank, SSSP, SCC, and BFS until a given number of jobs are created, where the CGP jobs are executed over a series of snapshots, respectively. Note that Seraph-VT is the version of Seraph incorporating multi-version switching approach [15].

First, we evaluate the total execution time of eight CGP jobs over different systems for hyperlink14 with the graph change ratio ranging from 0.005% to 5%. In detail, the change on the successive two snapshots ranges from 0.005% to 5%. Figure 16 gives the results relative to the execution time of Seraph-VT when the ratio of changed edges is 0.005%. We can observe that CGraph always gets the best performance under different graph change ratios. It is because CGraph still gets a low average data access cost to computation ratio, although the snapshots have differences in graph structure. Besides, we can also find that CGraph needs longer execution time to handle the graph when the graph change ratio is larger, because of less data access correlations between the jobs.

In the following experiments, we take a series of snapshots of hyperlink14 as datasets, where the change ratio between any successive two snapshots is 5% and each job handles a snapshot. Figure 17 depicts the execution time breakdown of the jobs over different systems on hyperlink14 when the the number of jobs increases. We can find that the jobs over CGraph have a lower average data access cost to computation ratio with the increase of the number of jobs, because there are more jobs to amortize the data access cost. However, for Seraph-VT and Seraph, the condition with more jobs leads to much more volume of data loaded into the cache and makes them suffer from serious cache interference and limited bandwidth. Thus, CGraph performs much better than Seraph-VT and Seraph when the number of jobs is larger.

The last-level cache miss rate is also evaluated for them on hyperlink14. As depicted in Figure 18, the cache miss rate of CGraph is significantly reduced when the number of jobs is increased, because the data in the cache can be repeatedly used by different CGP jobs. For example, in CGraph, the cache miss rate of the condition with eight jobs is even only 32.8% of the condition with one job. However, in the other solutions, the cache miss rate is significantly increased when the number of jobs is more, because of serious cache interference.

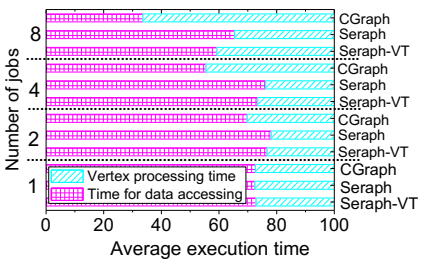


Figure 17: Execution time breakdown of different solutions on hyperlink14

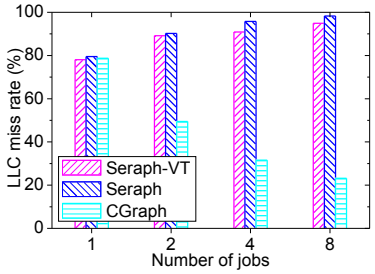


Figure 18: Last-level cache miss rate of different solutions on hyperlink14

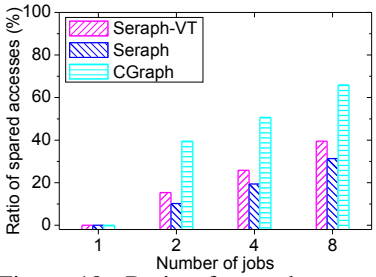


Figure 19: Ratio of spared accessed data on hyperlink14

Figure 19 gives the ratio of the total accessed data (including the data from the disk to the main memory and the main memory to the cache) spared by different solutions on hyperlink14 in comparison with the way sequentially executing the jobs over Seraph. As expected, the number of data accesses spared by CGraph is much more than the other solutions. For example, when the number of jobs is eight, the ratio is even up to 65.9% for CGraph, while the ratios of Seraph-VT and Seraph are only 39.5% and 31.3%, respectively. Besides, as observed, CGraph spares much more data accesses when the number of jobs increases, due to more opportunity to share data accesses between different jobs.

5 Related Work

With the explosion of graph scale, many systems [18, 19, 27] have focused on achieving high efficiency for iterative graph analysis. However, most of them focus on supporting single graph processing job. They improve the efficiency either by fully utilizing the sequential usage of memory bandwidth, or by achieving a better data locality and less redundant data accesses, which consequently reduces the volume of the accessed data.

GraphChi [17] achieves sequential storage access by employing parallel sliding windows. X-Stream [23] and Chaos [22] improve GraphChi by using streaming partitions for better sequential access of out-of-core data. Xie etc. [28] propose a novel block-oriented computation model, in which computation is iterated locally over blocks of highly connected nodes, which improves the amount of computation per cache miss. PathGraph [31] models a large graph using a collection of tree-based partitions for better locality. GridGraph [32] proposes 2-Level hierarchical partitioning scheme to improve the locality and reduce the amount of I/Os. NXgraph [11] uses destination-sorted sub-shard structure to store graph for better locality and adaptively chooses the fastest strategy to fully utilize memory and reduce data transfer. Instead of targeting a better locality, CLIP [6] proposes to reduce the total data access cost through the reentry of the loaded data and beyond-neighborhood accesses.

Although these systems can support efficient execution of a single iterative graph processing job, multiple separate copies of the same graph need to be created in the main memory by them for the CGP jobs. Following on this direction, Seraph [29, 30] is designed to allow multiple jobs to correctly share one copy of the in-memory graph structure data. However, in Seraph, the accesses to the same graph partitions are performed separately by the jobs along different graph paths, incurring redundant accesses and wasting the cache as well. Note that graph databases [8] are recently proposed to support concurrent queries over a graph. For example, TAO [9] provides a simple data model and APIs to store and query graph data. Wukong [24] uses a RDMA-based approach to provide low-latency concurrent queries over large graph-based RDF datasets. However, these graph database solutions can not efficiently support the execution of the CGP jobs because they are dedicated to graph queries which usually only touch different small subsets of a graph for exactly once, instead of iteratively processing the entire graph for many rounds.

6 Conclusion

This paper discovers that many redundant data accesses exist in the CGP jobs for their strong temporal and spatial correlations. A novel data-centric LTP model and an efficient scheduling algorithm is then proposed to exploit our observed data access correlations in these jobs and allows multiple CGP jobs to efficiently amortize the data access cost for higher throughput. Experimental results show that our approach significantly improves the throughput for the CGP jobs against the state-of-the-art solutions. This work mainly focuses on static graph processing. In the future, we will research how to further optimize our approach for evolving graph analysis and also extend it to distributed platform and also heterogeneous platform consisting of GPUs so as to get higher throughput for the CGP jobs.

Acknowledgments

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. 61702202 and 61628204, China Postdoctoral Science Foundation Funded Project under grant No. 2017M610477 and 2017T100555.

References

[1] Cachegrind. http://www.valgrind.org/, 2017.

[2] facebook. http://www.facebook.com/, 2017.

[3] Law. http://law.di.unimi.it/datasets.php, 2017.

[4] Snap. http://snap.stanford.edu/data/index.html, 2017.

[5] Wdc. http://webdatacommons.org/hyperlinkgraph/, 2017.

[6] AI, Z., ZHANG, M., WU, Y., QIAN, X., CHEN, K., AND ZHENG, W. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 125–137.

[7] BALUJA, S., SETH, R., SIVAKUMAR, D., JING, Y., YAGNIK, J., KUMAR, S., RAVICHANDRAN, D., AND ALY, M. Video suggestion and discovery for youtube: Taking random walks through the view graph. In *Proceedings of the 17th International Conference on World Wide Web* (2008), pp. 895–904.

[8] BORNEA, M. A., DOLBY, J., KEMENTSIETSIDIS, A., SRINIVAS, K., DANTRESSANGLE, P., UDREA, O., AND BHATTACHARJEE, B. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 121–132.

[9] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., AND LI, H. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), pp. 49–60.

[10] BULUÇ, A., AND MADDURI, K. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–12.

[11] CHI, Y., DAI, G., WANG, Y., SUN, G., LI, G., AND YANG, H. Nxgraph: An efficient graph processing system on a single machine. In *Proceedings of the 2016 IEEE International Conference on Data Engineering* (2016), pp. 409–420.

[12] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th Usenix Conference on Operating Systems Design and Implementation* (2012), pp. 17–30.

[13] HAN, M., AND DAUDJEE, K. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment 8*, 9 (2015), 950–961.

[14] HONG, S., RODIA, N. C., AND OLUKOTUN, K. 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* (2013), pp. 1–11.

[15] JU, X., DAN, W., JAMJOOM, H., AND KANG, G. S. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 523–536.

[16] KUMAR, R., KUMAR, R., LU, K., VASSILVITSKII, S., AND VASSILVITSKII, S. Local search methods for k-means with outliers. *Proceedings of the VLDB Endowment 10*, 7 (2017), 757–768.

[17] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (2012), pp. 31–46.

[18] MAASS, S., MIN, C., KASHYAP, S., KANG, W., KUMAR, M., AND KIM, T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the 12th European Conference on Computer Systems* (2017), pp. 527–543.

[19] MALICEVIC, J., LEPERS, B. J. E., AND ZWAENEPOEL, W. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *Proceedings of the 2017 USENIX Annual Technical Conference* (2017), pp. 631–643.

[20] MEYER, U. 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* (2001), pp. 797–806.

[21] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[22] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 410–424.

[23] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), pp. 472–488.

[24] SHI, J., YAO, Y., CHEN, R., CHEN, H., AND LI, F. Fast and concurrent RDF queries with rdma-based distributed graph exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation* (2016), pp. 317–332.

[25] VORA, K., XU, G., AND GUPTA, R. Load the edges you need: A generic i/o optimization for disk-based graph processing. In *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), pp. 507–522.

[26] WANG, K., HUSSAIN, A., ZUO, Z., XU, G., AND AMIRI SANI, A. 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* (2017), pp. 389–404.

[27] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. Gram: Scaling graph computation to the trillions. In *Proceedings of the 6th ACM Symposium on Cloud Computing* (2015), pp. 408–421.

[28] XIE, W., WANG, G., BINDEL, D., DEMERS, A., AND GEHRKE, J. Fast iterative graph computation with block updates. *Proceedings of the VLDB Endowment 6*, 14 (2013), 2014–2025.

[29] XUE, J., YANG, Z., HOU, S., AND DAI, Y. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers 66*, 5 (2017), 876–890.

[30] XUE, J., YANG, Z., QU, Z., HOU, S., AND DAI, Y. Seraph: An efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing* (2014), pp. 227–238.

[31] YUAN, P., ZHANG, W., XIE, C., JIN, H., LIU, L., AND LEE, K. Fast iterative graph computation: A path centric approach. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), pp. 401–412.

[32] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference* (2015), pp. 375–386.