GraphScSh: Efficient I/O Scheduling and Graph Sharing for Concurrent Graph Processing

Shang Liu[1], Zhan Shi[1](✉), Dan Feng[1], Shuo Chen[1], Fang Wang[1], and Yamei Peng[2]

[1] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
{upup,zshi,dfeng,shuochen,wangfang}@hust.edu.cn
[2] Didi, Inc., Beijing, China
1014280613@qq.com

Abstract. With the increasing need for analyzing graph data, graph systems have to efficiently deal with concurrent graph processing (CGP) jobs. However, existing platforms are inherently designed for a single job, they incur the high cost when CGP jobs are executed. In this work, we observed that existing systems do not allow CGP jobs to share graph structure data of each iteration, introducing redundant accesses to same graph. Moreover, all the graphs are real-world graphs with highly skewed power-law degree distributions. The gain from extending multiple external storage devices is diminishing rapidly, which needs reasonable schedulings to balance I/O pressure into each storage. Following this direction, we propose GraphScSh that handles CGP jobs efficiently on a single machine, which focuses on reducing I/O conflict and sharing graph structure data among CGP jobs. We apply a CGP balanced partition method to break graphs into multiple partitions that are stored in multiple external storage devices. Additionally, we present a CGP I/O scheduling method, so that I/O conflict can be reduced and graph data can be shared among multiple jobs. We have implemented GraphScSh in C++ and the experiment shows that GraphScSh outperforms existing out-of-core systems by up to 82%.

Keywords: Graph processing · CGP jobs · Graph sharing · I/O scheduling

1 Introduction

In the past decade, graph analysis has become important in a large variety of domains. Due to the increasing need to analyze graph structure data, it is common that Concurrent Graph Processing (CGP) jobs are executed on same processing platforms, in order to acquire different information from same graphs.

© IFIP International Federation for Information Processing 2019

Published by Springer Nature Switzerland AG 2019
X. Tang et al. (Eds.): NPC 2019, LNCS 11783, pp. 3–15, 2019.
https://doi.org/10.1007/978-3-030-30709-7_1

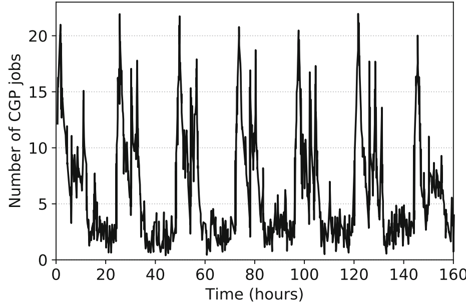


Fig. 1. The number of CGP jobs.

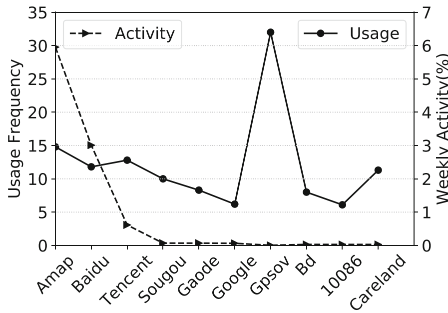


Fig. 2. The utilization of map apps.

For example, Facebook uses Apache Giraph [6] to execute various graph algorithms, such as the variants of PageRank [12], SSSP [10], etc. Figure 1 depicts the number of CGP jobs over a large Chinese social network [17]. The stable distribution shows that more than 83.4% of the time has at least two CGP jobs executed simultaneously. At the peak time, over 20 CGP jobs are submitted to the same platform. Also, Fig. 2 shows the usage of Chinese map Apps in a week of 2017. We can observe that each map App is used by each user more than five times within a week. Particularly, Amap App [2] ranks the first and handles over 10 billion route plannings every week, that is to say, it is used more than 60 thousand times per minute on average.

The existing processing systems can process a single graph job efficiently. 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, like GraphChi [8], X-Stream [13], GridGraph [20] and Graphene [9], PreEdge [11], etc. However, these systems are usually designed for a single graph processing job, which are much more inefficient when executing multiple CGP jobs. The inefficiencies include I/O conflict and repeated access to same graph structure data.

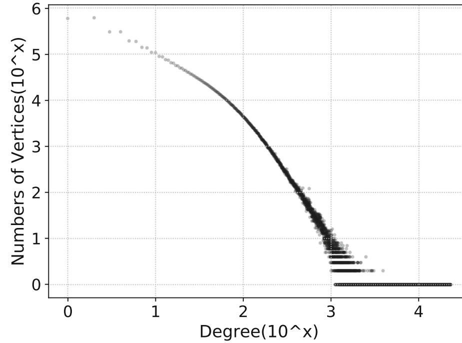


Fig. 3. Power-law degree distribution.

I/O Conflict: When multiple CGP jobs are executed over same graph, it is commonplace that these jobs visit same partition data, resulting in I/O conflict among multiple jobs. Fortunately, extending multiple external storage devices is possible to reduce this conflict, which can distribute multiple I/O of CGP jobs to multiple external storage devices. However, graphs derived from real-world phenomena, like social networks and the web, typically have highly skewed power-law degree distributions [1], which implies that a small subset of vertices connects to a large fraction of the graph. Figure 3 depicts the pow-degree distribution of graph from LiveJournal [14], which is a free online community with almost 10 million members. The highly skewed characteristic of graph challenges the above assumption and make it more difficult. Although using multiple storage devices reduces I/O conflict, this conflict is still the bottleneck of overall performance.

Data Access Problems: Graph processing jobs are usually operated on two types of data [9]: graph structure data and graph state data. The graph structure data mainly consists of vertices, edges, and the information associated with each edge. The graph state data, such as ranking scores for PageRank, is computed within each iteration and consumed in the next iteration. The graph structure data usually occupies a large volume of the memory, whose proportions are varying from 71% to 83% for different datasets [19]. However, existing graph platforms do not allow CGP jobs to share the graph structure data in memory, resulting in redundant access to the graph from external storage. Furthermore, existing out-of-core systems leverage various mechanisms to utilize the sequential usage of memory bandwidth and achieve a better data locality, such as PSW in GraphChi, Edge-Centric in X-Stream and 2-level hierarchical partitioning in GridGraph, etc. Unfortunately, CGP jobs destroy these optimized mechanisms above, increasing overhead of randomized access significantly.

In this paper, we propose GraphScSh, a graph processing system based on multiple external storage devices. Our design concentrates on reducing I/O conflict and sharing the graph structure data among CGP jobs. Specifically, the graph structure data is divided into multiple external storage devices evenly by CGP balanced partition method. The subgraph of each partition can match the

size of memory well, which reduces the overhead of frequent swap operations. Furthermore, we present a new CGP I/O scheduling method based on multiple external storage and graph sharing, so that I/O conflict can be reduced and the graph can be shared among multiple CGP jobs.

The system GraphScSh has been implemented in C++. To demonstrate the efficiency of our solutions, we conducted extensive experiments with our system GraphScSh and compared its performance with state-of-the-art systems GridGraph over different combinations of CGP jobs. The experiments show that overall performance of GraphScSh outperforms GridGraph by up to 82%.

The rest of this paper is organized as follows. The design details of GraphScSh are presented in Sect. 2, including CGP balanced partition schema, and CGP I/O scheduling method. Section 3 gives the specific implementation of our system GraphScSh, followed by experimental evaluation in Sect. 4. We then describe related work in Sect. 5 and conclude in Sect. 6.

2 Our Proposed Approach

To reduce the I/O conflict and the redundant access to graph efficiently, we propose GraphScSh based on multiple external storage devices, which is designed to reduce I/O conflict and share the graph structure data among CGP jobs.

2.1 CGP Balanced Partition

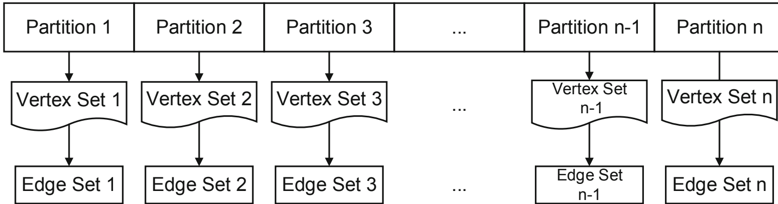


Fig. 4. Partitioning schema of GraphScSh.

The existing partitioning methods are usually designed for a single job. When CGP jobs are executed, we cannot make sure that partitioning size of all jobs match the size of memory, resulting in frequently swap-in and swap-out operations. We propose a new partitioning method to process CGP jobs, as shown in Fig. 4.

The graph is divided into n partitions, and each partition includes a vertex set and an edge set. Within a vertex set, the index id of vertices is continuous. The edge set of a partition consists of all edges whose source vertex is in the partition's vertex set. When GraphScSh executes graph algorithms, each partition size depends on both memory configuration and number of CGP jobs, so

that data of each vertex set can be fit into memory. Additionally, GraphScSh leverages multiple external devices to store the graph data. For the load balance, different partitions are stored in multiple storage devices and the number of edges for each partition is same. The position *disk_id* of each partition in multiple external storage can be described as,

$$disk_id = partition_id \% disk_num \tag{1}$$

where *partition_id* is the id of graph partition, *disk_num* is the number of external storage.

2.2 CGP I/O Scheduling

Based on the above partitioning method, we break graph structure data into multiple partitions evenly which are stored in multiple external storage devices. To reduce the I/O conflict and share the graph among CGP jobs, we propose a CGP I/O scheduling method based on CGP Balanced Partition method. The scheduling method includes two strategies for load balance and graph sharing.

First, we count the total number of jobs in each external storage and select one external storage that has the fewest jobs as the target, for loading balance. During execution of CGP jobs, system records *partition_id* that each job visits. The position of graph partition is computed according to the mapping between partitions and the external storage. For example, there are n jobs executed, where m jobs visit the first external storage for graph, and $(n-1-m)$ jobs access the second external storage. If $m > (n-1)/2$, the second one will be selected as the target, otherwise the first will be targeted. Assume that the number of external storage is k, where the number of jobs is $n-1, n-2, ..., n-k$, the storage with the fewest jobs will be targeted.

Second, we leverage synchronous field to reduce total number of I/O as much as possible to share the same graph, as Fig. 5 shows. The sync field mainly records information about the mapping from graphs to memory, including mapping address *mmap_addr* [18], the number *edge_num* of edges, and the descriptor fd of file. In addition, the field must include the total number *unit_num* of jobs and determines whether to remove the mapping of partition according to it. Specifically, according to *unit_num*, the system decides if partition data has been mapped into the memory according to the sync field. If $unit_num = 0$, the partition is not visited by jobs and should be filled into memory through mapping. Otherwise, the partition has been loaded into memory by other jobs, and the current job visits partition by the address of field.

unit_num	mmap_addr	edge_num	fd

Fig. 5. Sync field of graph.

The specific process of CGP I/O scheduling method includes several steps. Suppose that the number of the external storage is k, the concurrent graph job is A, the I/O scheduling of CGP jobs contains the following steps:

- According to synchronous information of CGP jobs and mapping information between partition and disk, the system counts the number of jobs executed in each external storage as n_1, n_2, ..., n_k, respectively.
- According to synchronous information of CGP jobs and mapping information between partition and disk, the system counts the number of partitions in each external storage, as s_1, s_2, ..., s_k, respectively, and records *partition_id*.
- The system sorts the external storage according to the values of n_1, n_2, ..., n_k. Then the corresponding *id* of the external storage is added into set U, where the number of jobs in each external storage is in ascending order.
- The system decides each external storage of U one by one. If the set s_i of one external storage i contains a partition that has not been accessed, the external storage i is selected as the target.
- If the partition data in memory has been processed by job A, A will visit each storage in U to find the data which has not been used. If the data exists, the corresponding external storage will be as the target and the current iteration ends.

Assume that the total execution time of a graph job is T, its computation time is T_c and its I/O wait time is T_w. When N jobs are executed on the same graph, the computation time of jobs is $T_{C1}, T_{C2}...T_{CN}$ respectively, and I/O wait time is T_w. The total execution time of existing systems can be described as,

$$T_o = max(T_{C1}, T_{C2}, ...T_{CN}) + NT_w \tag{2}$$

where $T_{C-MAX} = max(T_{C1}, T_{C2}, ...T_{CN})$. So the total time can be described as,

$$T_o = T_{C-MAX} + NT_w \tag{3}$$

Suppose that the number of external storage devices is D. Based on loading balancing, the I/O pressure is balanced into each external storage. Therefore, the number of jobs running on each device is N/D. The new total execution time can be described as,

$$T_{multi-disks} = T_{C-MAX} + T_w * N/D \tag{4}$$

the total number of I/O is from NT_W to $N/D * T_W$. The new total execution time is described as,

$$T_G = max(T_{C-MAX}, T_w) \tag{5}$$

We can see that the new I/O Scheduling outperforms the existing methods by up to $(N - N/D)$ theoretically.

3 GraphScSh Implementation

We have implemented our system GraphScSh in C++. Figure 6 illustrates the modules of GraphScSh, including graph management, mapping management, data structure, operation module, and graph algorithms. We mainly focus on two parts in this section: operation module and graph algorithms.

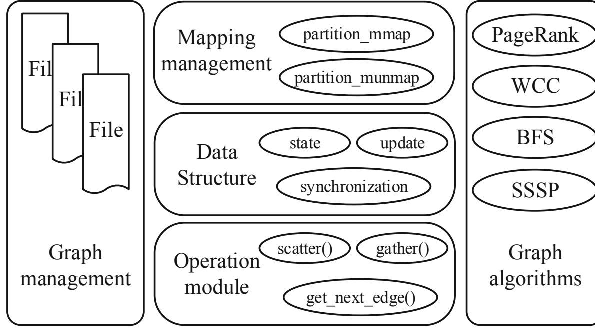


Fig. 6. Modules of GraphScSh.

3.1 Operation Module

The function of this module is achieved by operations of Scatter and Gather. In *Scatter* phase, it accesses to graph in streaming way by function $get_next_edge()$ and generates the updated information according to state data. In *Gather* phase, it read updated data and updates the state data. The Traversal operation is the kernel operation and implements by the function $get_next_edge()$. First, the function needs to determine partitions of graph whether to be visited. If false, the next edge data will be accessed. Then, $get_next_edge()$ decides all partition of this iteration whether to be visited. If true, the next iteration will be started. If false, the function findNextPartition() will be activated to find the next partition to visit. The implementation details of FindNextPartition are described in Algorithm 1.

Algorithm 1. Details of FindNextPartition

Input:
The partition set of graph $unaccess_partition$;
The set of external storage U;
The visited partition set of graph $s_1, s_2, ..., s_k$;
Output:
The next partition to be visited $partition_index$;
1: **for** i in U **do**
2:    **if** $\exists p \in unaccess_partition, p \in s_i$ **then**
3:       $partition_index = p$;
4:       return;
5:    **else**
6:       continue;
7:    **end if**
8: **end for**
9: **for** p in $unaccess_partition$ **do**
10:    $partition_index = p$;
11:    break;
12: **end for**

3.2 Implementations of Graph Algorithm

We define *Graph* as the base class, which provides a programming interface for graph algorithms. Class *Graph* defines five virtual functions, including *initUnit*() for initialization, *output*() for outputting result, *reset*() for cleaning after one iteration, *Scatter*(), and *Gather*(). The function *initUnit*() initializes the related work of graph algorithms, for example, the out-degree of each vertex in PageRank. The function *reset*() resets partition sets that workers have visited, and the number of partitions that each external storage has accessed. Algorithms 2 and 3 give examples to show how to implement graph algorithms on GraphScSh, which uses edge-centric Scatter-Gather model to run graph algorithms.

Algorithm 2. PageRank Scatter

```
1: for each edge e of graph do
2:    update_t upt;
3:    if update_bitset[e.dst]= false then
4:       upt.id = e.dst;
5:       upt.value = e.src.value/e.src.degree;
6:       add upt to update_buf;
7:    end if
8: end for
```

Algorithm 3. PageRank Gather

```
1: for each update u of upt_buffer do
2:    if update_bitset[u.id]= false then
3:       _aux[upt.id].tmp+ = u.value;
4:    end if
5: end for
6: for  each element ele of _aux  do
7:    if update_bitset[ele.index − start]=false then
8:       tmp = init_wgt + 0.85 ∗ ele.tmp;
9:       if fabs(ele.tmp − tm) < 0.00000001  then
10:         update_bitset[ele.index − start]=true;
11:      else
12:         ele.res = tmp;
13:      end if
14:      ele.tmp = 0;
15:   end if
16: end for
```

4 Experimental Evaluation

4.1 Experiment Environment and Datasets

The hardware platform used in our experiments is a single machine containing 6-core 1.60 GHz Intel(R) Xeon(R) CPU E5-2603. Its memory is 8 GB and has two SSDs with 300 GB. The program is compiled with g++ version 11.0.

In our experiments, four popular graph algorithms are employed as benchmarks: (1) breadth-first search (BFS) [3]; (2) PageRank (PR) [12]; (3) weakly connected component (WCC) [7]; (4) single-source shortest path (SSSP) [10]. The datasets used for these graph algorithms are real-world graphs and generated graphs described in Table 1. Where Twitter [14] is from online social networks and edges represent interactions between people. R-MAT, SW, and ER are generated based on power-law [4], small-world model [15] and ER model [5] respectively.

Table 1. Data sets properties

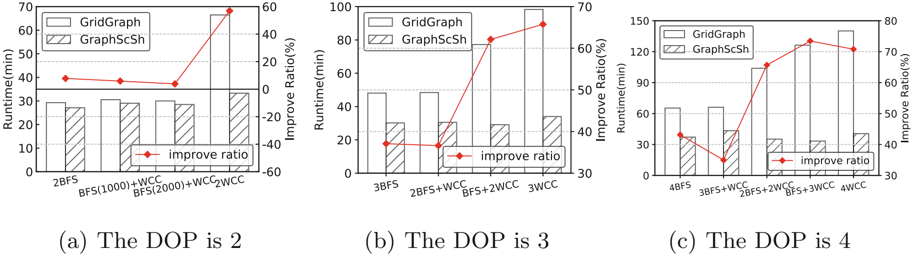
DataSets	Vertexes	Edges	Average degree	Description
Twitter	61.6 M	1.5 B	23.8	Social networks from Twitter
R-MAT26	67.1 M	1.1 B	16	Power-law degree distributions
ER26	67.1 M	1.1 B	16	Random degree distributions

Table 2. Execution time of algorithms on GridGraph(s)

Data Sets	BFS	WCC	PageRank
Twitter	768.84	883.07	3630.38
R-MAT26	409.27	349.27	2389
ER26	482.22	223	596.33

4.2 Comparison with GridGraph

To compare the performance of GridGraph and GraphScSh, we simultaneously submit multiple jobs to each system. The partition number of GraphScSh is set same as GridGraph, and different datasets have a different number of partitions. The execution time of various graph processing algorithms has been computed, as Table 2 depicted. For better comparing the performance of systems, CGP jobs consist of two graph algorithms with same converge speed based on different datasets. To acquire better integrity, experiments are designed under different degree of parallelism (DOP) [16].



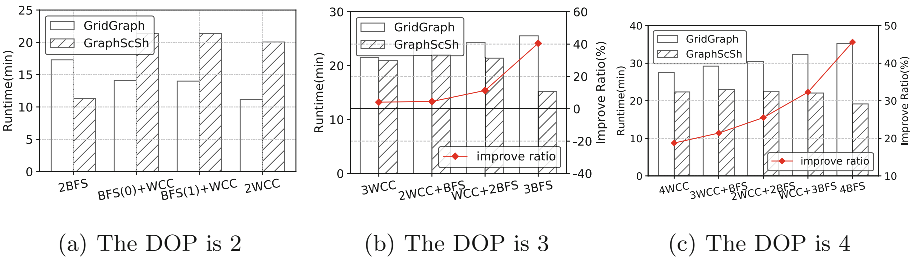
(a) The DOP is 2 (b) The DOP is 3 (c) The DOP is 4

Fig. 7. The comparison of runtime on Twitter for GraphScSh and GridGraph

Twitter: First, for graph dataset Twitter, we evaluate the total execution time and the speed-up ratio of various CGP jobs (e.g. the DOP is 2, 3 and 4, respectively, as Fig. 7(a), (b) and (c). In general, for different combinations of CGP jobs, the execution time of GraphScSh is less than that of GridGraph, and the speed-up ratio grows up as DOP increases. Under the same DOP but a different combination, the longer execution time of CGP jobs is, the greater GraphScSh outperforms GridGraph. Specifically, when two systems are executed on dataset Twitter, the combinations of 2WCC, 3WCC, and 4WCC are accelerated by 56.93%, 65.75%, and 70.8% respectively. Because CGP jobs are executed on the GridGraph, resulting in the I/O conflict greatly.

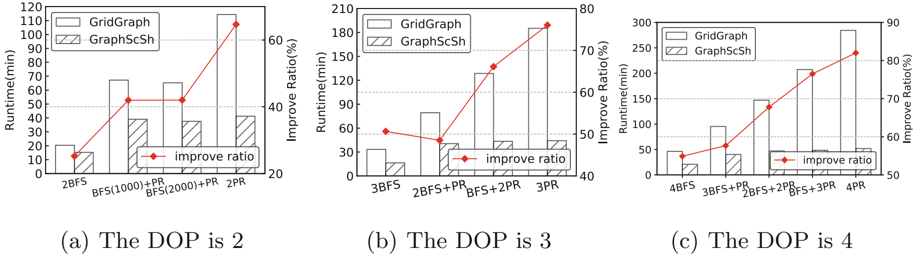
RMAT26: Next, we execute different combinations of CGP jobs on RMAT26 to compare GridGraph and GraphScSh, as Fig. 8(b) and (c) show. When the DOP is 3 or 4, the performance of GraphScSh is better than that of GridGraph. In particular, with the increase DOP, the speed-up ratio grows up gradually. For example, GraphScSh outperforms GridGraph by 34%, 40.5% and 45.6% under the combinations of 2BFS, 3BFS, and 4BFS, respectively.

ER26: Besides, from Fig. 9(a), (b) and (c), we can observe that the total execution time of GraphScSh is much less than those of GridGraph over dataset ER26. For example, for the combinations of 2PR, 3PR and 4PR, GraphScSh outperforms GridGraph by 64.67%, 76.03% and 82%, respectively. Under the same DOP, the difference that GraphScSh executes different combinations of CGP jobs is smaller than that of GridGraph. It also means that GraphScSh with GSSC and MSGL is suitable to cope with CGP jobs.



(a) The DOP is 2 (b) The DOP is 3 (c) The DOP is 4

Fig. 8. The comparison of runtime on R-MAT26 for GraphScSh and GridGraph



(a) The DOP is 2 (b) The DOP is 3 (c) The DOP is 4

Fig. 9. The comparison of runtime on ER26 for GraphScSh and GridGraph

5 Related Work

With the explosion of graph scale, lots of graph processing systems are created to achieve high efficiency for graph analysis. They improve the efficiency either by a prefetcher for graph algorithms, or by full utilizing the sequential usage of memory bandwidth.

PrefEdge [11] is a prefetcher for graph algorithms that parallelises requests to derive maximum throughput from SSDs. PrefEdge combines a judicious distribution of graph state between main memory and SSDs with an innovative read-ahead algorithm to prefetch needed data in parallel. GraphChi [8] a disk-based system for computing efficiently on graphs with billions of edges. By using a novel parallel sliding windows method, GraphChi is able to execute several advanced data mining, graph mining, and machine learning algorithms on very large graphs, using just a single consumer-level computer. X-Stream [13] is novel in using an edge-centric rather than a vertex-centric implementation of this model, and streaming completely unordered edge lists rather than performing random access. GridGraph [20] is an out-of-core graph engine using a grid representation for large-scale graphs by partitioning vertices and edges to 1D chunks and 2D blocks respectively, which can be produced efficiently through a lightweight range-based shuffling.

Unfortunately, when CGP jobs are executed on these systems above, they incur the extra high cost (e.g., inefficient memory use and high fault tolerance cost). Following this observation, Seraph [17] is designed to handle with CGP jobs based on a decoupled data model, which allows multiple concurrent jobs to share graph structure data in memory [19]. Based on this observation 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, CGraph [19] proposed a correlations-aware execution model. Together with a core-subgraph based scheduling algorithm, CGraph enables these CGP jobs to efficiently share the graph structure data in memory and their accesses by fully exploiting such correlations.

6 Conclusion

This paper introduces GraphScSh, a large scale graph processing system that can support CGP jobs running on a single machine with multiple external storage devices. GraphScSh adopts a CGP balanced partition method to break graphs into multiple partitions that are stored in multiple external storage devices. In addition, we present a CGP I/O scheduling method, so that I/O conflict can be reduced and the same graph can be shared among multiple CGP jobs. Experimental results depict that our approach significantly outperforms existing out-of-core systems when running CGP jobs. In the future, we will research to further optimize our solution with a snapshot mechanism for efficient graph processing.

Acknowledgments. This work is supported by NSFC No. 61772216, 61821003, U1705261, Wuhan Application Basic Research Project No. 2017010201010103, Fund from Science, Technology and Innovation Commission of Shenzhen Municipality No. JCYJ20170307172248636, Fundamental Research Funds for the Central Universities.

References

1. Dasgupta, A., Hopcroft, J.E., McSherry, F.: Spectral analysis of random graphs with skewed degree distributions. In: FOCS 2004 (2004)
2. BDR: http://www.bigdata-research.cn/content/201801/635.html
3. Beamer, S., Asanovic, K., Patterson, D.A.: Direction-optimizing breadth-first search. In: SC 2012 (2012)
4. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: a recursive model for graph mining. In: SDM 2004 (2004)
5. Erdös, P., Rényi, A.: On random graphs. Publicationes Mathematicae Debrecen **6**, 290 (1959)
6. Apache Giraph: http://giraph.apache.org/
7. Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D., Kalnis, P.: Mizan: a system for dynamic load balancing in large-scale graph processing (2013)
8. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: OSDI 2012 (2012)
9. Liu, H., Huang, H.H.: Graphene: fine-grained IO management for graph computing. In: FAST 2017 (2017)
10. Maleki, S., Nguyen, D., Lenharth, A., Garzarán, M.J., Padua, D.A., Pingali, K.: DSMR: a parallel algorithm for single-source shortest path problem. In: ICS (2016)
11. Nilakant, K., Dalibard, V., Roy, A., Yoneki, E.: PrefEdge: SSD prefetcher for large-scale graph traversal (2014)
12. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web. Technical report (1999)
13. Roy, A., Mihailovic, I., Zwaenepoel, W.: X-stream: edge-centric graph processing using streaming partitions. In: SOSP 2013 (2013)
14. SNAP: http://snap.stanford.edu/data/index.html
15. Watts, D., Strogatz, S.: Collective dynamics of small world networks. Nature **393**, 440–442 (1998)
16. Wikipedia: https://en.wikipedia.org/wiki

17. Xue, J., Yang, Z., Qu, Z., Hou, S., Dai, Y.: Seraph: an efficient, low-cost system for concurrent graph processing. In: HPDC 2014 (2014)
18. Lin, Z., Kahng, M., Sabrin, K.Md., et al.: MMap: fast billion-scale graph computation on a pc via memory mapping. In: Big Data, pp. 159–164 (2014)
19. Zhang, Y., et al.: CGraph: a correlations-aware approach for efficient concurrent iterative graph processing. In: ATC 2018 (2018)
20. Zhu, X., Han, W., Chen, W.: GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: ATC 2015 (2015)