Int J Parallel Prog (2017) 45:1259–1272
DOI 10.1007/s10766-016-0468-8

FOG: A Fast Out-of-Core Graph Processing Framework

Zhiyuan Shao[1] · **Jian He**[1] · **Huiming Lv**[1] · **Hai Jin**[1]

Received: 29 September 2016 / Accepted: 26 October 2016 / Published online: 1 November 2016

Abstract In this paper we present FOG, an open source graph processing framework designed for out-of-core (external memory) graph processing (https://github.com/mrshawcode/fog). FOG provides a set of programming interfaces that break down update functions of vertices to their incident edges so as to process the functions with edge-centric manner. By these, FOG gives intuitive and productive programming interfaces, and achieves high main memory utilization rate and processing efficiency at the same time. Moreover, FOG proposes an in-place update shuffling mechanism to improve the performance by dramatically reducing disk I/Os during computing. By extensive evaluations on typical graph algorithms and large real-world graphs, we show that FOG outperforms existing out-of-core graph processing systems, including GraphChi, X-Stream and TurboGraph. By comparing the performances of FOG and those of state-of-art distributed graph processing frameworks, we show that only by using just a commodity PC, FOG achieves comparable or even better performance than the best distributed graph processing framework that uses an Amazon EC2 cluster with 128 nodes.

Keywords Big data · Graph processing · Parallel computing · Performance optimization

✉ Zhiyuan Shao
zyshao@hust.edu.cn

Hai Jin
hjin@hust.edu.cn

[1] Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

1 Introduction

The increasing sizes of real-world graphs and the wide adoption of graph computing (e.g., in social networks, bio-informatics, web graph analysis and many others) have been motivating the building of graph processing systems in the past several years. In order to process large graph data-sets, except for purchasing an expansive huge memory machine, there are two other options: one is to use a cluster and employ distributed graph processing system, such as Giraph [1], GraphLab [14], PowerGraph [7] and many others. The other is to use a single commodity machine and employ out-of-core graph processing system, such as GraphChi [11], X-Stream [16], TurboGraph [9], GridGraph [19] and many others. The first (cluster) option brings in the burden of configuring and maintaining a cluster. The second (out-of-core) option is much cost-effective and easily attracts more attentions.

As the performance is generally decided by the slowest part (straggler), optimizing the accesses to the disk, which is obviously the straggler in out-of-core graph processing systems, is crucial in improving the performance. Existing out-of-core graph processing systems are used to access the whole on-disk graph data sequentially, in order to reach a maximum disk bandwidth. However, as the execution of many graph algorithms only need to access a subset of data, existing systems inevitably result in substantial waste on both disk bandwidth and memory space. The way of using the *limited* main memory space is another crucial factor to the performance of out-of-core graph processing systems. Some systems (e.g., GraphChi) build the sub-graphs in memory to facilitate processing. Besides high graph ingression overheads, the fixed structure of sub-graphs makes it impossible to just access the on-demand vertices or edges. Some other systems (e.g., X-Stream) avoid the graph-ingression by employing the edge-centric processing model. However, the edge-centric processing model introduces a new processing stage named as *shuffling*, which relocates the updates to their destination processors. As the updates need to be flushed to disk and read again for shuffling, it results in high overheads.

FOG addresses these limitations by (1) implementing on-demand data acquisition to reduce waste on memory space and disk bandwidth; (2) employing an in-place update shuffling mechanism to avoid the overhead for shuffling and improve parallelism. This paper introduces FOG and makes following contributions:

- Presents FOG, an open source out-of-core graph processing framework, which provides programming interfaces for the users to program their graph algorithms and process them on big graph data-sets efficiently.
- Proposes a novel in-place update shuffling mechanism that greatly reduces disk I/Os during processing and thus improves performance.
- Extensively evaluates the performance of FOG and compares it with state-of-art out-of-core graph processing systems and distributed graph processing systems.

The rest of the paper is organized as follows: Sect. 2 gives an overview of FOG. Section 3 elaborates update handling. Section 4 addresses load balancing. Section 5 evaluates the performance. Section 6 gives a brief survey on the related works. Section 7 concludes the paper.

2 System Overview

2.1 Programing Interfaces

FOG provides a set of programming interfaces as shown in Fig. 1. Two core functions, i.e., *scatter_one_edge* and *gather_one_update*, can be defined by a graph algorithm to scatter values along an incident edge of a vertex or gather an update towards a vertex respectively. *init* is invoked during the initialization phase to give initial value to a vertex. *finalize* is invoked at the convergence of the algorithm. *before_iteration* and *after_iteration* are invoked before or after each iteration respectively. FOG implements two types of engines: global engine and targeted engine. During processing, the global engine traverses the entire graph at each iteration, while the targeted engine traverses only the scheduled vertices of the graph in each iteration. When using targeted engine, the program needs to explicitly initialize scheduling in *init* function, and adds new tasks in *gather_one_update* function.

2.2 On-Disk Data Organization

FOG uses the CSR (*Compressed Sparse Row*) format to organize the on-disk data. Figure 2 illustrates the on-disk data of FOG with an example graph. The adjacency lists (without the source vertex) are stored continuously in an array file (out_edge_array). An index file (out_edge_idx) is generated to record the start offset of vertex's adjacency list. FOG can optionally extend the out_edge_array file with a weight value field, and optionally generate the index and array files by in-edges (i.e., Compressed Sparse Column format).

FOG leverages virtual memory management mechanism of OS to map these on-disk files to virtual addresses. During processing, reads to these virtual addresses is automatically translated to disk reads to achieve on-demand data acquisition, while

Fig. 1 Programming interfaces

```
init (vertex &v)
    initialize the attribute of vertex v
    {initialize scheduling}

before_iteration ()
    invoked before each iteration

scatter_one_edge (vertex v, edge e)
    scatter edge e according to its source vertex v

gather_one_update (vertex &v, update u)
    apply update u to vertex v
    {add v to schedule of next iteration}

after_iteration ()
    invoked after each iteration

finalize ()
    invoked at convergence
```

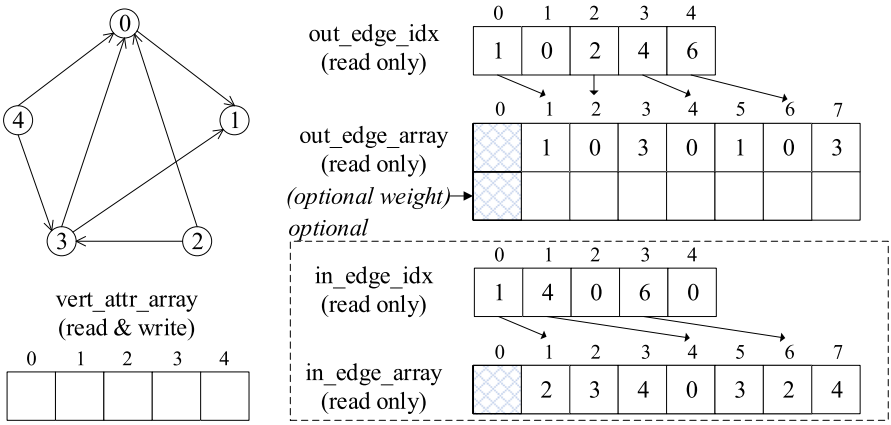

Fig. 2 Organization of on-disk data

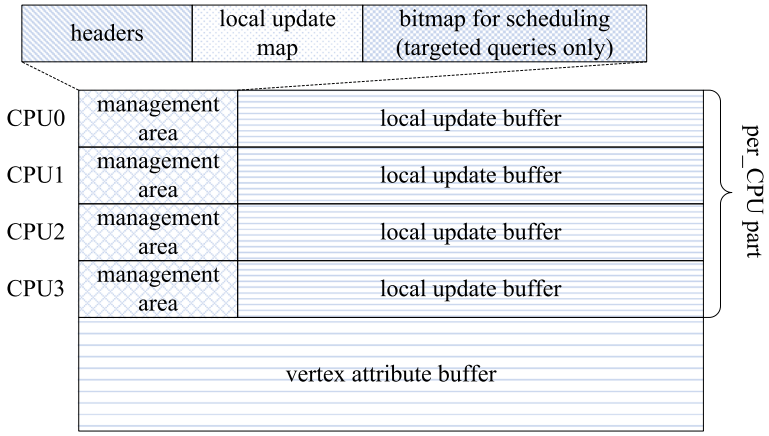

Fig. 3 Organization of in-memory data

writes to vert_attr_array file are governed by mechanisms that will be discussed in Sect. 3.

2.3 In-Memory Data Organization

FOG statically allocates a block of memory to organize in-memory data structures. Figure 3 illustrates such organization on a four-processor platform. The allocated memory is divided into two parts: the *vertex attribute buffer* and the *per_CPU part*. The vertex attribute buffer is dedicated to store the vertex attributes. The per_CPU part is evenly divided among all participating processors, and each of processor's space is divided into two parts: the *management area* and the *local update buffer*. The local update buffer stores the updates generated during processing. The management area is further divided into three parts: the *headers* part stores the management data structures,

Algorithm 1 Computes S_{attr_buf}, $N_{segment}$ and $C_{segment}$

```
1: S_attr_buf ← γ × M                                   ▷ tentative size
2: if S_attr_buf ≥ S_attr_arr then                      ▷ file is small
3:     S_attr_buf ← S_attr_arr                          ▷ shrink the buffer
4:     N_segment ← 1
5:     C_segment ← V
6: else                                                 ▷ file is large, divide the file into segments
7:     N_segment ← ⌈S_attr_arr/(S_attr_buf/2)⌉
8:     C_segment ← ⌊(S_attr_buf/2)/S_attr⌋
9: end if
```

the *local update map* is used to track the usage of local update buffer, and the *bitmap for scheduling* area is used to store the bitmap of the scheduler employed by targeted engine, and does not exist in global engine.

Instead of applying the updates to the memory mapped vert_attr_array file, FOG uses the vertex attribute buffer to absorb these small and random writes. However, the size of the vert_attr_array file (denoted as S_{attr_arr}) may be too big to be fully loaded into the vertex attribute buffer. In such case, the vertex attribute buffer is divided into two and used as dual buffer, while the vert_attr_array file is divided into multiple *segments* to be loaded one at a time. Denote the size of the allocated memory as M, the number of vertices of the graph as V, and the size of a vertex attribute as S_{attr}, the size of vertex attribute buffer (as S_{attr_buf}). The number of segments ($N_{segment}$) and the segment capacity ($C_{segment}$, in unit of vertex attributes) are computed by Algorithm 1, where γ ($\gamma = 0.4$) is an empirically chosen constant parameter.

Denote the size of management area as S_{mgr_area}. The size of the local update buffer (denoted as $S_{local\_update\_buf}$, in unit of bytes) of each processor is computed by Eq. 1, where N_{CPU} denotes the number of participating CPUs.

$$S_{local\_update\_buf} = \frac{M - S_{attr\_buf}}{N_{CPU}} - S_{mgr\_area} \tag{1}$$

2.4 Flowchart

FOG organizes the computation of a graph algorithm into multiple scatter-gather iterations as illustrated in Fig. 4. In the beginning of an iteration, each participating processor will be given a set of tasks (in vertex IDs), and the iteration ends when all processors finish their assigned tasks. During scatter phase, a processor places the updates in its local update buffer. However, as the space of local update buffer is limited as computed by Eq. 1, when handling huge amount of tasks in one iteration, there may not be enough space to store all the updates. In such case, the processor terminates the scatter phase, and enters gather phase to consume the updates so as to clear the local update buffer. After that, the processor will come back to scatter phase again to continue handling the unaccomplished tasks. To facilitate discussion, we call one of such scatter-and-gather loops within an iteration as a *sub-iteration*, and the mechanism as *in-place update shuffling*.

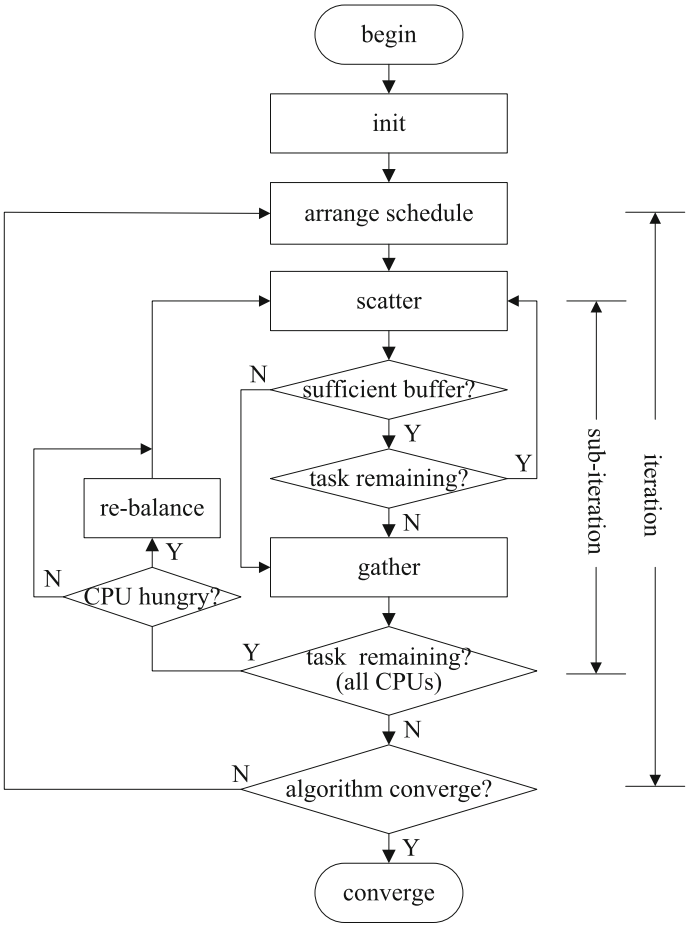

Fig. 4 Flowchart of FOG

When there are multiple processors participating processing, FOG synchronizes the processing on them and enforces barriers at each phase-changing. In order to maximize the parallelism and achieve balanced processing, FOG distributes the tasks by modulo-based fashion and re-balances the tasks among the processors when some processors are hungry. Such mechanism will be discussed in Sect. 4.

3 Update Handling

When an update U, whose destination vertex is dst_vert, is generated during scatter phase on CPU_i, the processor (say CPU_j) that will consume this update is decided by dst_vert. FOG places U at an indexable memory location in CPU_i's local update buffer to help CPU_j to find it during gather phase. In order to keep track of the memory location where updates are placed, the local update buffer of a processor is divided into multiple equal-sized *stripes* according to the number of segments of the graph.

Algorithm 2 Place update U

```
1: r ← dst_vert/C_segment                          ▷ row number in LUM_i
2: c ← dst_vert%N_CPU                              ▷ column number in LUM_i
3: offset ← LUM_i[r][c] × N_CPU + c
4: B ← CPU_i's local update buffer start address
5: stripe_addr ← B + C_stripe × r × S_update
6: if offset < C_stripe then
7:     LUM_i[r][c] ← LUM_i[r][c] + 1
8:     p ← stripe_addr+ (offset ×N_CPU + c) ×S_update
9:     Place U at position p
10: else
11:     quit scatter phase
12: end if
```

Algorithm 3 Consume updates at stripe r on CPU_j

```
1: for x ∈ [0, N_CPU) do                           ▷ browse all processors
2:     t ← LUM_x[r][j]
3:     B ← CPU_x's local update buffer start address
4:     stripe_addr ← B + C_stripe × r × S_update
5:     for t > 0 do                                ▷ t updates to be consumed
6:         t ← t − 1
7:         p ← stripe_addr + (t × N_CPU + j) × S_update
8:         Consume update at position p
9:     end for
10:    LUM_x[r][j] ← 0
11: end for
```

The capacity (denoted as C_{stripe}, in unit of updates) of one stripe buffer is computed by Eq. 2, where S_{update} is the size of the update structure.

$$C_{stripe} = \lfloor S_{local\_update\_buf}/(N_{segment} \times S_{update}) \rfloor \tag{2}$$

Each processor also maintains a local update map (denoted as LUM), which is a matrix with its element values recording the number of updates, its rows indicating the corresponding stripes and its columns indicating the processors. Algorithm 2 places U into CPU_i's local update buffer. Note that in Algorithm 2, CPU_i will quit the scatter phase when there is no space to place U. However, at that moment, its local update buffer may *not* be actually full, since there may be remaining space in other stripes (i.e., inter-stripe imbalance), or inside stripe r for other processors except for $CPU_{dst\_vert}\%N_{CPU}$ (i.e., intra-stripe imbalance). We will address the inter-stripe imbalance and discuss the intra-stripe imbalance in Sect. 4.

During developing, we find that the intra-stripe imbalance is common when processing real-world graphs due to their skewed nature. Hence, it will be wasteful to load all the segments into memory to consume all the updates in the local update buffers at each sub-iteration. In order to reduce overhead and guarantee the correctness of computing results, FOG decides whether to load a segment to consume the updates in gather phase by following principles:

(1) If a segment's corresponding stripe is considered to be full by at least one processor, the segment will be loaded;

(2) If the average utilization rate of the segment's corresponding stripes among *all processors* is beyond a predefined threshold (80%), the segment will be loaded;
(3) If the sub-iteration is at the end of an iteration (i.e., all processors had finished their tasks of the iteration), all segments will be loaded one after another.

In order to further reduce the overheads resulted by frequent segment loading and writing, FOG employs the least recently used (*LRU*) algorithm to manage the replacement of the segments inside the (dual) vertex attribute buffer. Moreover, when the system detects only a few updates (e.g., 2% of total update buffer space usage rate) to be gathered at the end of an iteration, the updates will be applied *directly* to the mapped vert_attr_array file. When the system decides to consume updates towards vertex attributes in segment r, Algorithm 3 is employed to guide a participating processor (say CPU_j) to find and consume all the updates in stripe buffer r. Note that Algorithm 3 is for all participating processors, which means *each* participating processor will access the local update buffers of *all* participating processors (including itself) during gather phase to retrieve and consume the updates.

4 Load Balancing

Since the phase-changings are synchronized among all participating processors, FOG designs two mechanisms: modulo-based task assignment and re-balancing. Suppose there is a set of tasks $S = \{0, 1, ..., V-1\}$ to be processed at the beginning of an iteration. There are two possible manners to assign them to the processors: range manner and modulo-based manner. By range manner (GridGraph adopts this), CPU_i will handle task subset $S_i = \{v | v \in [(V/N_{CPU}) \times i, (V/N_{CPU}) \times (i+1) - 1]\}$, while by modulo-based manner, it will handle subset $S_i = \{v | v \in S, \textit{and } v\%N_{CPU} = i\}$. We evaluate these two manners by conducting one-iteration SpMV algorithm [4] on Twitter graph [10], and show the stripe buffer usage rates and the relative standard deviations (indicating intra-stripe imbalance) on the number of updates that are to be consumed on different processors in Fig. 5. It can be observed from Fig. 5 that the

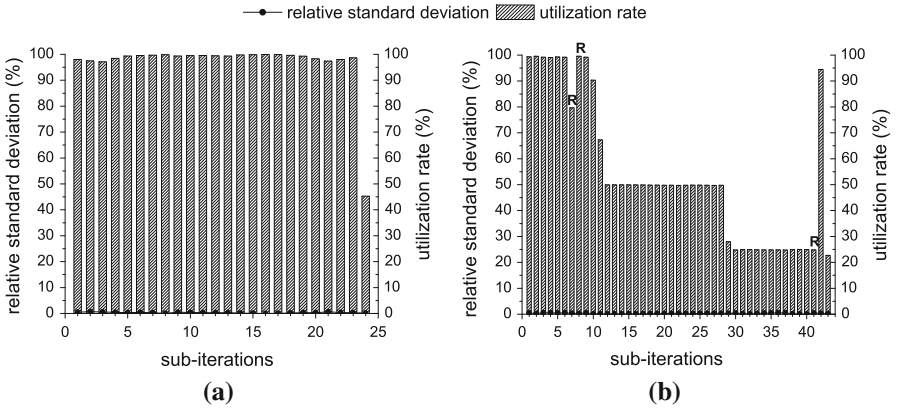

Fig. 5 The local update buffer utilization rate and metrics of update distribution during the sub-iterations of SpMV on Twitter graph ("R" above the columns stands for re-balancing). **a** Modulo-based, **b** range

Table 1 Real-world graphs used in the experiments

Graph	LiveJournal	Twitter	UK	YahooWeb
Vertices	4.8M	41.6M	105M	1.4B
Edges	69M	1.5B	3.7B	6.6B

modulo-based manner results in less sub-iterations and few re-balancing operations. The reason is that the modulo-based manner task distribution resembles partitioning a graph by hash algorithm, which generally leads to well-balanced partitions [15], and it converts to the high update buffer usage rates and balanced load distribution during gather phase. As we observe similar buffer usage patterns when conducting other algorithms on real-world graphs, FOG chooses to employ the modulo-based manner task assignment.

For scatter phases, FOG designs a simple re-balancing mechanism to cope with the "in-case" situation that some processors accomplished their assigned tasks faster than the others: if it happens, FOG will browse the processors (one at a time) with remaining tasks and distribute its remaining tasks (with range manner at this time) among all processors.

5 Performance Evaluation

5.1 Experiment Setups

The test-bed for experiments is a commodity PC with one Intel Core i7-2600 processor, which has 4 cores running at 3.4 GHz, 4 × 256 KB L2 cache and 8 MB L3 cache. The hyper-threading feature of the CPU is disabled. The PC is configured with 12 GB of DDR3 RAM (scales to 28 GB in Sect. 5.3), one 7200 RPM Seagate 1TB HDD and one 500 GB Samsung 840 EVO-Series SSD with SATA3 interface. We use 64-bit Ubuntu server 12.04 to conduct all experiments in Linux, and 64-bit Windows7 with SP1 to conduct experiments in Windows.

Four real-world graphs [i.e., LiveJournal [2], Twitter [10], uk-2007-05 (UK for short) [5] and YahooWeb [17]] as listed in Table 1 and five representative graph algorithms: PageRank [12], Sparse Matrix-Vector Multiplication (*SpMV*) [4], Weakly Connected Components (*WCC*), Breadth-First Search (*BFS*) [3], and Single Source Shortest Path (*SSSP*) are conducted to measure the performances of GraphChi C++ version 0.2, X-Stream version 0.9, and our latest release (i.e., version 0.2) of FOG. As TurboGraph is not an open source software, we only use its PageRank and BFS performances during comparisons.

5.2 Comparisons with Other Out-of-Core Systems

5.2.1 Performance Comparisons

Figure 6 compares the execution times of the chosen algorithms conducted in GraphChi, X-Stream, TurboGraph, and FOG on graphs stored on both SSD and HDD.

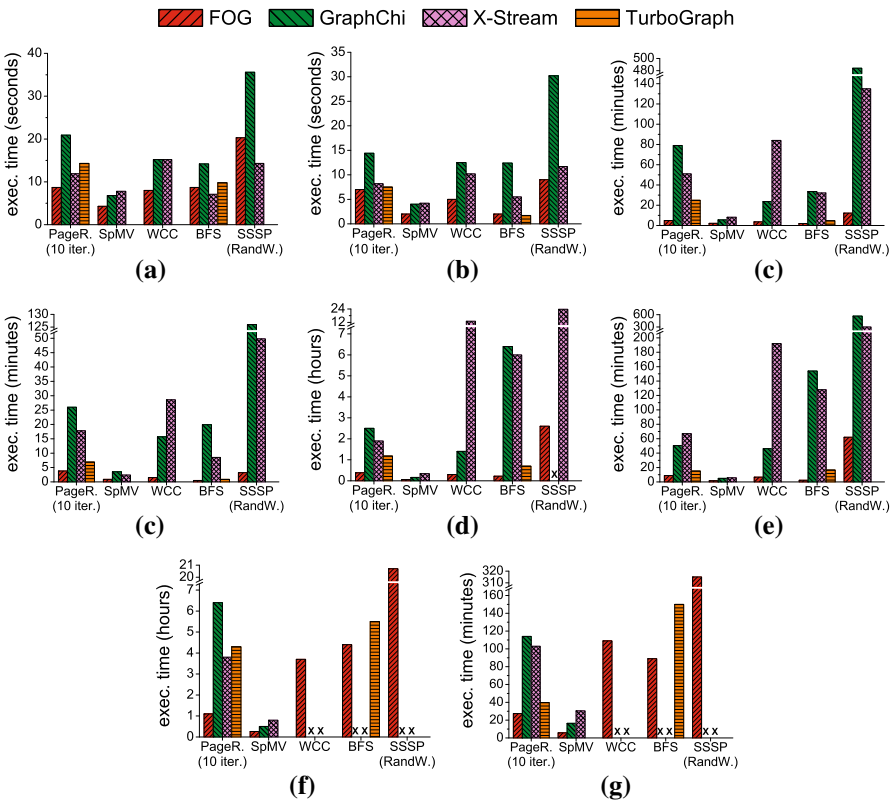

Fig. 6 Average execution times of the graph algorithms in the chosen systems ("X" stands for an execution time bigger than 24 h). **a** LiveJournal on HDD, **b** LiveJournal on SSD, **c** Twitter on HDD, **d** Twitter on SSD, **e** UK on HDD, **f** UK on SSD, **g** YahooWeb on HDD, **h** YahooWeb on SSD

It can be observed from Fig. 6 that for small graphs (i.e., LiveJournal), where all systems work by in-memory mode, FOG's performances are only surpassed by X-Stream in BFS and SSSP on HDD. The reason is that FOG and X-Stream have the same I/O requirements ($E + V$ for reading and V for writing) for the in-memory mode executions. However, FOG implements BFS and SSSP with targeted engine, which randomly reads the data and thus results in suboptimal performance on HDD. The situation changes when running these two algorithms on SSD, since it has much higher IOPS (I/Os per second) than HDD. In other cases, FOG outperforms all other systems. This is because FOG also sequentially reads the data from disk, and does not have the time-consuming graph-ingress phase (GraphChi) or dedicated shuffling phase (X-Stream). When graphs become larger, FOG greatly outperforms the other three systems. For example, considering BFS on UK graph stored on SSD, its performance in FOG is about 68x faster than in GraphChi, 57x faster than in X-Stream and 7.3x faster than in TurboGraph.

Table 2 Total disk I/O amount (GB)

	PageRank		BFS	
	Total read	Total write	Total read	Total write
FOG	125.6	0.4	17.7	0.4
GraphChi	330	157	975	26
X-Stream	613	308	3285	27
TurboGraph	296.5	3.8	132	0.1

5.2.2 Total Amount of Disk I/Os

In order to understand the reasons behind the performance improvements, we collect the amount of disk I/Os during the execution of PageRank and BFS, which represent typical global query and targeted query algorithms respectively, on UK graph. Table 2 lists the total amount of disk I/Os in these systems.

From Table 2, it can be observed that for PageRank, the total amount of disk I/Os (both read and write) in FOG is less than that in GraphChi and X-Stream. The reason is that FOG only needs to read the edges once during each iteration, while besides the edges, GraphChi needs to read the compress edge weight values, and X-Stream needs to read the updates flushed to disk in the previous iteration. Regarding write, since FOG can store the whole vertex array of UK graph in memory, it results in very small quantity of writes, while GraphChi needs to write the compressed edge weight, X-Stream needs to write the updates.

For BFS, it can be observed that the amount of disk I/Os for read in FOG are much smaller than those of other three systems. The reason is that FOG implements on-demand data acquisition by memory mapping, which is very economic in disk reading. At the same time, the execution of BFS only needs partial data of the graph during each iteration. GraphChi and X-Stream still need to read the whole graph as well as extra data (edge weights or updates) from disk at each iteration. The large amount of disk I/Os for reads in TurboGraph (for both PageRank and BFS) suggests that there is high waste on the loaded data in this system.

5.3 Performance Comparisons with Distributed Systems

We now compare the performances of FOG and state-of-art distributed systems. At FOG side, we take the best performances that are achieved on our test-bed (4 CPUs, 28 GB memory and SSD). At distributed systems side, we take the best performances recorded in [8], where the performance data is collected in an Amazon EC2 cluster with up to 128 m1.xlarge instances (each has four virtual processors and 15 GB memory). In order to align with the performance data in [8], we take three algorithms: PageRank (30 iterations), SSSP (unit weight), and WCC. Table 3 compares the performances of the chosen algorithms on Twitter and UK graphs.

From Table 3, it can be observed that the performances of PageRank in FOG is a little worse than those of distributed systems. The reason is that PageRank needs

Table 3 Performance comparison with state-of-art distributed systems (minutes)

Algorithm	FOG	Best distributed system
On Twitter Graph		
PageRank (30 iter.)	8.28	5.84 (GraphLab on 128 instances)
SSSP (unit weight)	0.93	1.49 (GraphLab on 128 instances)
WCC	1.18	2.69 (Giraph on 64 instances)
On UK Graph		
PageRank (30 iter.)	11.67	3.70 (GraphLab on 128 instances)
SSSP (unit weight)	3.02	4.79 (GraphLab on 64 instances)
WCC	5.95	8.72 (GraphLab on 128 instances)

to scan all the graph data at each iteration. In distributed systems, there are multiple disk and memory bus devices, whose aggregated bandwidth can easily surpass the bandwidth of single disk and single memory bus used in FOG.

However, for targeted query algorithms (i.e., SSSP and WCC), the performances in FOG are even better than those of the best distributed systems. The reason is that the frequent communications are the limiting factor to the performance in distributed systems. Meanwhile, for targeted query algorithms, the quantity of data to be accessed at each iteration are generally much smaller than that of global query algorithms. The communications thus occupy the majority of overheads in a distributed system, and limit its scalability. In FOG, however, it is much cheaper for processors to communicate via the shared memory. Moreover, FOG achieves economic use of disk bandwidth and efficient use of main memory space.

6 Related Works

GraphChi [11] proposes a novel Parallel Sliding Window (PSW) method that partitions the graph data into multiple subgraphs. During processing, the subgraphs are loaded into memory and processed in a round-robin fashion at each iteration. After computing on one subgraph, the edge data are written to disk to pass intermediate results to the computation of next subgraph, which results in huge overhead. VENUS [6] improves GraphChi by constraining the writes to vertices so as to avoid the overhead of writing edge data, but only slightly improves the performance. X-Stream [16] partitions graph data into super-partitions and cache-partitions, and employs edge-centric implementation to conduct the computation. Its update shuffling stage, however, results in huge overhead due to the disk reading and writing. Moreover, since a super-partition is treated as an entity to enforce sequential disk access, huge wastes on memory space are resulted when the computation needs only small portion of the vertices.

TurboGraph [9] organizes the graph data into pages indexed by an in-memory table, and proposes a pin-and-slide method to implement on-demand data acquisition. However, this system still results in high waste on main memory space and suboptimal performance as shown in Sect. 5. GridGraph [19] organizes the graph data into 1D-

partitioned vertex chunks and 2D-partitioned edge blocks, and conducts computation by a novel dual sliding window mechanism. GridGraph proposes an in-place mechanism for update handling, which however, is based on the range partitioning of the vertex ID space, and have to leverage the atomic operations to solve data races.

FlashGraph [18] implements a semi-external memory graph processing system. However, as all the vertices have to be loaded in memory during processing, the system cannot process graphs with huge amount of vertices. Although studies in [13] use memory mapping technique to fast index data during processing as our system, none of them solves the random write problem, which results in drastic thrashing when processing large graph with limited main memory space.

7 Conclusion

In this paper we present FOG, an open source out-of-core graph processing framework. Its programming interfaces break down the update functions over vertices to their incident edges, so as to efficiently use the limited memory space and make on-demand data acquisition possible. Its in-place update shuffling mechanism divides the lengthy iteration of graph processing into multiple sub-iterations, so as to consume the updates without extra disk access. All these combined contribute to reduce the disk I/Os, and thus greatly boost the performance.

Acknowledgements This work is supported by Natural Science Foundation of China under Grant No. 61433019.

References

1. Apache: Apache Giraph. http://giraph.apache.org/ (2012)
2. Backstrom, L., Huttenlocher, D., Kleinberg, J., Lan, X.: Group formation in large social networks: membership, growth, and evolution. In: Proceedings of KDD, pp. 44–54 (2006)
3. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. In: Proceedings of SC, pp. 12:1–12:10 (2012)
4. Bender, M.A., Brodal, G.S., Fagerberg, R., Jacob, R., Vicari, E.: Optimal sparse matrix dense vector multiplication in the I/O-model. In: Proceedings of SPAA, pp. 61–70 (2007)
5. Boldi, P., Rosa, M., Santini, M., Vigna, S.: Layered label propagation: a multiresolution coordinate-free ordering for compressing social networks. In: Proceedings of WWW, pp. 587–596 (2011)
6. Cheng, J., Liu, Q., Li, Z., Fan, W., Lui, J., He, C.: VENUS: vertex-centric streamlined graph computation on a single PC. In: ICDE, pp. 1131–1142 (2015)
7. Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of OSDI, pp. 17–30 (2012)
8. Han, M., Daudjee, K., Ammar, K., Özsu, M.T., Wang, X., Jin, T.: An experimental comparison of pregel-like graph processing systems. Proc VLDB Endow **7**(12), 1047–1058 (2014)
9. Han, W.S., Lee, S., Park, K., Lee, J.H., Kim, M.S., Kim, J., Yu, H.: TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In: Proceedings of KDD, pp. 77–85 (2013)
10. Kwak, H., Lee, C., Park, H., Moon, S.: What is Twitter, a social network or a news media? In: Proceedings of WWW, pp. 591–600 (2010)
11. Kyrola, A., Blelloch, G., Guestrin, C.: GraphChi: large-scale graph computation on just a PC. In: Proceedings of OSDI, pp. 31–46 (2012)
12. Lawrence, P., Sergey, B., Motwani, R., Winograd, T.: The PageRank citation ranking: bringing order to the web. Technical report, Stanford University (1998)

13. Lin, Z., Chau, D.H., U K: Leveraging memory mapping for fast and scalable graph computation on a PC. In: Proceedings of Big Data, pp. 95–98 (2013)
14. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proc VLDB Endow **5**(8), 716–727 (2012)
15. Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., Haridasan, M.: Managing large graphs on multi-cores with graph awareness. In: Proceedings of USENIX ATC, pp. 4–4 (2012)
16. Roy A, Mihailovic I, Zwaenepoel W: X-Stream: edge-centric graph processing using streaming partitions. In: Proceedings of SOSP, pp. 472–488 (2013)
17. Yahoo: Yahoo WebScope. Yahoo! altavista web page hyperlink connectivity graph. http://webscope.sandbox.yahoo.com/ (2002)
18. Zheng, D., Mhembere, D., Burns, R., Vogelstein, J., Priebe, C.E., Szalay, A.S.: FlashGraph: processing billion-node graphs on an array of commodity SSDs. In: Proceedings of FAST, pp. 45–58 (2015)
19. Zhu, X., Han, W., Chen, W.: GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of USENIX ATC, pp. 375–386 (2015)