Journal of Parallel and Distributed Computing 172 (2023) 51–68

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

LOSC: A locality-optimized subgraph construction scheme for out-of-core graph processing

Xianghao Xu[a,b,*], Fang Wang[b], Hong Jiang[c], Yongli Cheng[d,e], Yu Hua[b], Dan Feng[b], Yongxuan Zhang[b]

[a] School of Computer Science and Engineering, Nanjing University of Science and Technology, China
[b] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China
[c] Department of Computer Science Engineering, University of Texas at Arlington, USA
[d] College of Mathematics and Computer Science, FuZhou University, China
[e] Zhejiang lab, China

A R T I C L E   I N F O

Article history:
Received 1 February 2021
Received in revised form 3 October 2022
Accepted 7 October 2022
Available online 14 October 2022

Keywords:
Graph processing
Out-of-core
Subgraph construction

A B S T R A C T

Big data applications increasingly rely on the analysis of large graphs. In recent years, a number of out-of-core graph processing systems have been proposed to process graphs with billions of edges on just one commodity computer, by efficiently using the secondary storage (e.g., hard disk, SSD). Unfortunately, these graph processing systems continue to suffer from poor performance, despite of many solutions proposed to address the disk I/O bottleneck problem, a commonly recognized root cause. However, our experimental results show that another root cause of the poor performance is the subgraph construction phase of graph processing, which induces a large number of random memory accesses that substantially weaken cache access locality and thus greatly degrade the performance. In this paper, we propose an efficient out-of-core graph processing system, LOSC, to substantially reduce the overheads of subgraph construction. LOSC proposes a locality-optimized subgraph construction scheme that significantly improves the in-memory data access locality of the subgraph construction phase. Furthermore, LOSC adopts a compact edge storage format and a lightweight replication of vertices to reduce I/O traffic and improve computation efficiency. Extensive evaluation results show that LOSC is respectively 9.4× and 5.1× faster than GraphChi and GridGraph, two representative out-of-core systems. In addition, LOSC outperforms other state-of-art out-of-core graph processing systems including FlashGraph, GraphZ, G-Store and NXGraph. For example, LOSC can be up to 6.9× faster than FlashGraph.

1. Introduction

Graph is a powerful data structure to solve many real-world problems. There exist various modern big data applications relying on graph computing, including social networks, Internet of things, and neural networks.

However, with the real-world graphs growing in size and complexity, processing these large and complex graphs in a scalable way has become increasingly more challenging. To tackle this challenge, a number of graph-specific processing frameworks have been proposed. With these graph processing frameworks, users can write an update function for a specific graph application without considering the underlying execution details. To obtain a better performance, many systems adopt a large cluster to deploy their large graph processing jobs, such as Pregel [24], PowerGraph [13] and GraphX [14]. These systems distribute a large graph into the compute nodes of a cluster by constructing node-resident subgraphs from the original graph, which enables them to utilize the aggregate memory of a cluster to achieve good scalability. Unfortunately, they usually suffer from high hardware and communication/synchronization costs because of the significant amount of communication and coordination required among a large number of computing nodes when processing large graphs.

In recent years, many out-of-core graph processing systems such as GraphChi [18], X-Stream [26] and GridGraph [43], have been proposed to process large graphs on a single compute node, by efficiently using the secondary storage (e.g., hard disk, SSD). They overcome the challenges faced by distributed systems, such as load imbalance and significant communication overheads. Many recent works [18,43,5] have shown that these systems can achieve a competitive performance compared with distributed systems without massive hardware. When processing an input graph, the

* Corresponding author.
E-mail address: xianghao@njust.edu.cn (X. Xu).

https://doi.org/10.1016/j.jpdc.2022.10.005
0743-7315/© 2022 Elsevier Inc. All rights reserved.

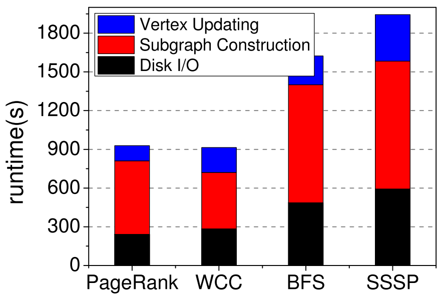


Fig. 1. Breakdown time of several algorithms on Twitter for GraphChi. To exactly measure the execution time of different phases, we have modified GraphChi's code to enable the sequential executions of the subgraph construction and disk I/O phases, since these two phases are overlapped in GraphChi's source code.

out-of-core systems divide a large graph into many partitions and load and process one partition from disk at a time. Current out-of-core graph processing systems mainly adopt two computing models, i.e., vertex-centric and edge-centric.

The vertex-centric computing model takes the vertex as the processing unit and each vertex can invoke a user-defined function to update its own value in parallel. As it is intuitive for users to express many graph algorithms, many graph processing systems [24,13,18] are implemented based on the vertex-centric model. The edge-centric computing model factors computation over edges instead of vertices and takes the edge as the processing unit. Compared with the vertex-centric model, the edge-centric model can leverage high disk bandwidth with fully sequential accesses. However, as vertex-centric model is properly designed to distribute and parallelize large graph analytics [12], traditional iterative graph computation is naturally expressed in a vertex-centric manner. Furthermore, for some algorithms such as community detection, it is difficult to implement them in an edge-centric model [5]. Therefore, in this paper, we mainly focus on the vertex-centric out-of-core graph processing systems for their better applicability and expressiveness.

Although out-of-core graph processing systems can be a cost-effective solution to handle large-scale graphs, there are two main problems that severely impact the performance of out-of-core graph processing systems.

First, because the secondary storage delivers much less bandwidth and much longer latency than DRAM, the disk I/O overheads become an inevitable performance bottleneck. Most of current out-of-core graph processing systems are designed to reduce the disk I/O overheads, by proposing various techniques. For example, GraphChi [18] exploits a novel method called parallel sliding windows (PSW) to reduce random disk accesses as much as possible. CLIP [3] adopts a reentry technique to make full use of the loaded blocks to avoid loading the corresponding graph portions in the future iterations and speedup the convergence of graph algorithms.

Second, we have discovered another performance bottleneck of the out-of-core graph processing systems is the inefficient subgraph construction that causes frequent random memory accesses. Fig. 1 shows the runtime breakdown of several algorithms on Twitter graph executed by GraphChi. We observe the subgraph construction phase is responsible for at least 48% of the whole execution time. In fact, when implemented in an out-of-core system to process a graph partition, the vertex-centric model requires all edges of the partition to be loaded from the disk and assigned to their source and destination vertices to construct an in-memory vertex-centric subgraph structure, before updating the vertices of the partition. This is the phase of subgraph construction. Since the vertex data structures are stored sequentially by the vertex ID in memory, the assignments of edges will incur many random memory accesses as the source or destination vertices of the edges usually have non-sequential vertex IDs. Random memory accesses greatly weaken cache access locality and thus degrade performance by increasing cache miss rate. Table 1 shows the cache misses of different execution phases when running BFS on Twitter. We can see that the last-level cache (LLC) misses rate of the subgraph construction phase is much higher than those of other execution phases, which explains both the random memory accesses and high execution time caused by the subgraph construction phase, considering the much higher miss penalty of LLC than other levels of cache.

Table 1
Cache misses of different execution phases.

Execution Phase	LLC Miss (Read)	LLC Miss (Write)
Disk I/O	1.8%	1.5%
Subgraph Construction	14.1%	31.1%
Vertex Updating	2.3%	9.1%
Overall	4.3%	6.8%

In this paper, we present LOSC, an efficient out-of-core graph processing system that not only optimizes disk I/O performance but also significantly reduces the overheads of subgraph construction without sacrificing the underlying vertex-centric computing model. The main contributions of LOSC are summarized as follows.

- **Locality-optimized subgraph construction scheme**. LOSC proposes a locality-optimized subgraph construction scheme that improves the locality of memory access to greatly reduce the overheads of constructing subgraphs. The locality-optimized subgraph construction scheme ensures that the vertices required for adding incoming and outgoing edges are stored sequentially in memory when constructing subgraphs, which significantly improves the memory access locality. Moreover, to further improve the performance of subgraph construction, the locality-optimized subgraph construction scheme fully utilizes the parallelism of multi-threaded CPU when constructing subgraphs by assigning different worker threads to take charge of different vertices and their associated in-edges and out-edges.
- **Benefit-aware scheduling scheme**. To reduce disk I/O overheads, LOSC adopts a benefit-aware scheduling scheme to improve I/O performance by skipping loading and processing inactive edges in each iteration whenever such skipping can bring performance benefit. To achieve this, the benefit-aware scheduling scheme incorporates the designs of vertices indices, bitmap operation and I/O-based benefit evaluation model. Compared with our previous work [34], the benefit-aware scheduling scheme can reduce the amount of I/O traffic by up to 75%, which further reduces the subgraph construction overheads and improves system performance with negligible computing overheads.
- **Compact edge storage format**. LOSC implements a compact edge storage format by combining several graph compression methods, i.e., compression of undirected graph, delta compression and ID compression, to save storage size and reduce I/O traffic.
- **Lightweight replication of interval vertices**. LOSC adopts a lightweight replication of interval vertices (vertices within an interval) to improve computation efficiency by enabling full thread-level parallelism.
- **Extensive experiments**. We evaluate LOSC on several real-world graphs with different algorithms. Extensive evaluation results show that LOSC outperforms GraphChi and GridGraph by 9.4× and 5.1× on average respectively due to its locality-optimized subgraph construction and reduced disk I/Os.

Note that this paper is based on our prior work presented at the 2019 IEEE/ACM International Symposium on Quality of Service (IWQoS'19) [34]. We briefly provide the new contents beyond the prior conference version as follows.

- A benefit-aware scheduling scheme that further reduces disk I/O overheads.
- Parallelized locality-optimized subgraph construction scheme that further improves the performance of subgraph construction.
- Added experiments that make the performance evaluation of LOSC more convincing and more adequate.
- Substantial new contents and some revised old contents that strengthen motivation, evaluation and clarity, and help the reader better understand how LOSC works.

The rest of the paper is organized as follows. Section 2 presents the background and motivation. Section 3 describes the detailed system design of LOSC. Section 4 presents extensive performance evaluations. We discuss the related works in Section 5 and conclude this paper in Section 6.

2. Background and motivation

In this section, we first present the computing models of existing graph processing systems. Then, we introduce the state-of-art out-of-core graph processing systems. Finally, we take GraphChi as an example to demonstrate the process and performance impact of subgraph construction. This helps motivate us to propose a new out-of-core system that significantly improves system performance by reducing the overheads of subgraph construction.

2.1. Computing models of graph processing systems

Existing out-of-core graph processing systems mainly adopt two computing models, i.e., vertex-centric and edge-centric. The vertex-centric computing model establishes a “think like a vertex” idea [24] that can express a wide range of applications, for example, graph mining, data mining, machine learning and sparse linear algebra, as shown by many researchers [13,24,21,18,30]. This model consists of a sequence of iterations and a user-defined update function executed for all vertices in parallel. In each iteration of computation, each vertex gathers data from its incoming edges; then it uses the gathered data to update its own value by invoking the user-defined update function; finally, it propagates its new value along its outgoing edges to its neighbors.

Unlike the vertex-centric model, the edge-centric computing model [26,43] explicitly factors computation over edges instead of vertices and takes the edge as the processing unit. In this model, the system processes each edge by gathering an update from its source vertex and applying the update to its destination vertex. Therefore, the edge-centric model can stream edges from the storage and high disk bandwidth can be achieved with fully sequential accesses. However, as traditional iterative graph computation is naturally expressed in a vertex-centric manner, users must re-implement many algorithms in edge-centric API [5,42]. Furthermore, for some algorithms such as community detection, it is difficult to implement them in an edge-centric model [5].

2.2. Out-of-core graph processing

GraphChi [18] is an extensively-used out-of-core graph processing system that supports the vertex-centric computation model and is able to express many graph algorithms. It divides the vertices into disjoint intervals and breaks the large edge list into smaller shards containing edges with destinations in the corresponding vertex intervals. For a given vertex interval, its incoming edges are stored in its associated shard called memoryshard, while its outgoing edges are distributed among other shards called sliding shards. In addition, edges in a shard are sorted by their source vertices. GraphChi exploits a novel method of parallel sliding windows (PSW) to process all intervals. For each interval, PSW loads the incoming edges of the interval from memoryshard and loads the outgoing edges from sliding shards. Updating messages with their destination vertices in the working interval will be applied instantly, while other updates will be written to the rest of the sliding shards on the disk.

Following GraphChi, a number of out-of-core graph processing systems are proposed. Through a disk-friendly graph data organization format and well-designed execution engine, out-of-core graph processing systems can process large graphs with a reasonable performance while using much fewer hardware resources than a distributed system. However, there are two main problems that severely impact the performance of out-of-core graph processing systems. One is the expensive disk I/O overheads. As we know, the secondary storage delivers much less bandwidth and much longer latency than DRAM. Reducing disk I/O overheads has become the heart of the system designs for most current out-of-core graph processing systems. The other is the inefficient subgraph construction, which exists in the vertex-centric out-of-core graph processing systems. As shown in Fig. 1, the subgraph construction phase significantly degrades the overall performance. Other studies such as X-Stream [26] and GridGraph [43] utilize the edge-centric model where the computation is based on the edges and the system needs not to construct a vertex-centric subgraph in memory before processing. However, as mentioned in 2.1, these systems exhibit limited expressivity and programmability.

2.3. Subgraph construction in out-of-core graph processing

In fact, for vertex-centric graph processing systems (e.g., GraphChi), they usually create an in-memory data structure for each vertex during the processing, which usually includes vertex values, in-edges and out-edges (or messages) [18]. For out-of-core graph processing systems where all edges are stored on the disk, when implementing the vertex-centric computation on a graph partition, all edges of the partition should be first loaded into memory and then assigned to the memory structures of corresponding vertices. This is the phase of subgraph construction. During this phase, each edge is added to the out-edge and in-edge array of its source and destination vertex. Based on the in-memory vertex-centric subgraph, each vertex can perform the vertex-centric computation in parallel.

Fig. 2 illustrates an example of constructing subgraphs in GraphChi. As shown in Fig. 2(a), the vertices of the example graph are split into three intervals: 1-100, 101-200 and 201-300. Each interval is associated with a shard containing incoming edges of vertices in the interval. When constructing the subgraph of shard 1, GraphChi first processes the edge (1, 2) and accesses the memory address of vertex 2, then it writes the edge to the incoming edge array of vertex 2. Afterwards, it accesses the memory address of vertex 4 and writes the edge (1, 4) into the incoming edge array of vertex 4 until all edges in the shard are added. The non-sequential destination vertices of the edges cause many random reads and writes in memory to add the incoming edges when constructing subgraphs as shown in Fig. 2(b). It is a known fact that random memory accesses tend to weaken cache locality and result in high cache miss rate, thus degrading memory access performance.

Based on the above observations and analysis, the subgraph construction phase significantly impacts the performance of out-

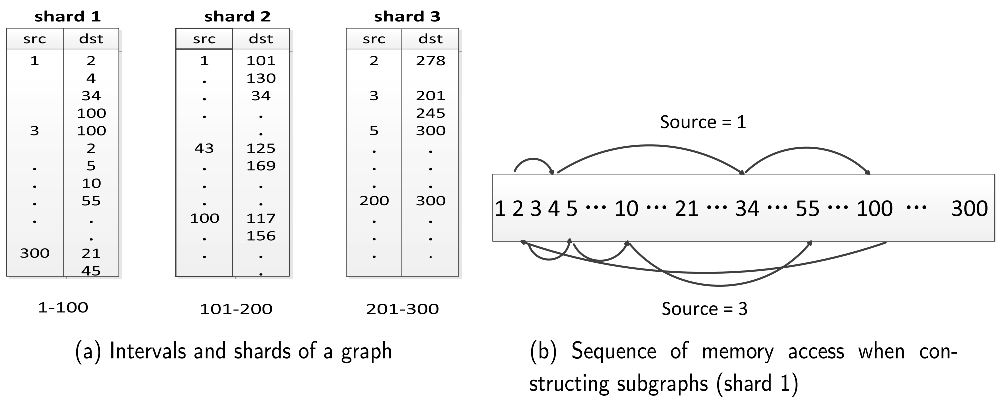


Fig. 2. The example of constructing subgraphs.

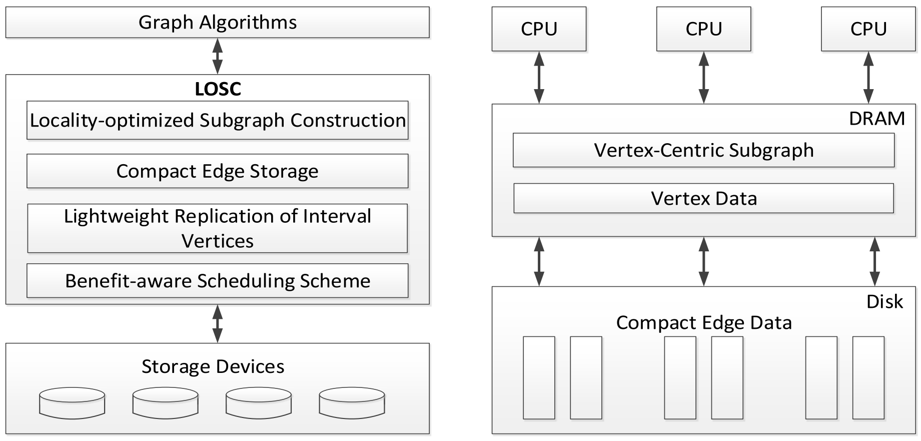


Fig. 3. The LOSC architecture.

of-core graph processing systems due to massive random reads and writes in memory. This motivates us to seek a design that minimizes random memory accesses to reduce the overheads of subgraph construction. In addition, the disk I/O overheads remain a severe performance bottleneck of out-of-core graph processing systems, I/O-related optimizations should also be considered to improve the overall performance.

3. System design

A graph problem is usually encoded as a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. For a directed edge e = (u, v), we refer to e as v's *in-edge*, and u's *out-edge*. Additionally, u is an *in-neighbor* of v, v is an *out-neighbor* of u. The computation of a graph G is usually organized in several iterations where V and E are read and updated. Updating messages are propagated from source vertices to destination vertices through the edges. The computation terminates after a given number of iterations or when it converges. Like previous works [5,41], we treat all vertices as mutable data and edges as read-only data.

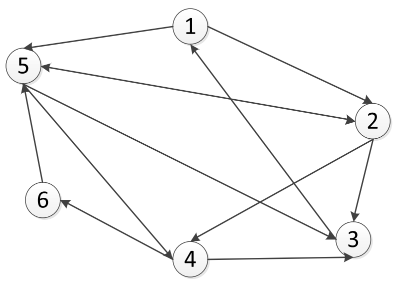
In this section, we first present the system overview of LOSC. Then, we illustrate the design of locality-optimized subgraph construction scheme, compact edge storage format, benefit-aware scheduling scheme and lightweight replication of interval vertices. Finally, we describe the main workflow of LOSC in detail with an example.

3.1. System overview

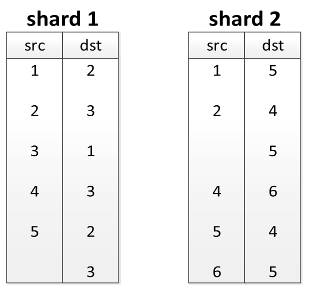
LOSC is an efficient out-of-core graph processing system supporting vertex-centric computing model. Fig. 3 presents the system architecture of LOSC. In order to reduce the overheads of subgraph construction, LOSC uses a novel locality-optimized subgraph construction scheme that maximizes the sequential accesses to vertices and edges during the subgraph construction phase. To further improve the system performance, LOSC also adopts a compact edge storage format to reduce I/O traffic and a lightweight replication of interval vertices to improve computation efficiency. In addition, a benefit-aware scheduling scheme is applied to skip loading and processing the inactive edges, which further improves the I/O performance.

To support the efficient subgraph construction scheme, LOSC organizes the graph data with a dual-shard representation. Like GraphChi, LOSC splits the vertices V of graph G into P disjoint intervals and edges E into P shards with source or destination vertices in corresponding vertex intervals. Differently, it associates two edge shards for each interval, in-shard and out-shard. In-shard (n) contains all in-edges of the vertices in interval (n), sorted by the destination vertices. Out-shard (n) contains all out-edges of the vertices in interval (n), sorted by the source vertices. By this way, the system can ensure the sequential accesses to the source or destination vertices when processing the out-edges or in-edges. The number of intervals, P, is chosen to ensure that the in-shard and out-shard of each interval can fit in memory. We illustrate the contrast between the representation of GraphChi and LOSC of an example graph in Fig. 4. The example graph has six vertices, which have been divided into two equal intervals: 1-3 and 4-6. While GraphChi only stores the in-edges of an interval in the corresponding shard, LOSC stores in-edges and out-edges of each interval in the corresponding in-shard and out-shard respectively.

Discussion. While the dual-shard representation that adopts 1-D partitioning and maintains both in-edges and out-edges is simi-



(a) Example graph



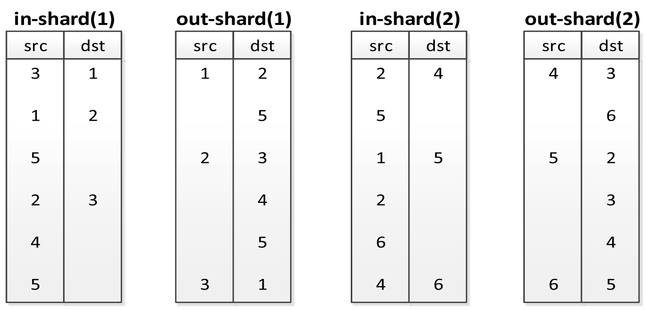
shard 1

src	dst
1	2
2	3
3	1
4	3
5	2
	3

shard 2

src	dst
1	5
2	4
	5
4	6
5	4
6	5

(b) GraphChi shard representation



in-shard(1)

src	dst
3	1
1	2
5	
2	3
4	
5	

out-shard(1)

src	dst
1	2
	5
2	3
	4
	5
3	1

in-shard(2)

src	dst
2	4
5	
1	5
2	
6	
4	6

out-shard(2)

src	dst
4	3
	6
5	2
	3
	4
6	5

(c) LOSC's dual-shard representation

Fig. 4. Illustration of graph representation.

lar to the CSR/CSC format that many previous works [28,41,44,35] use, we use this representation for a totally different purpose. Specifically, previous works use this representation to support different computing models and graph algorithms that need both in-edges and out-edges. For example, Ligra [28] stores both in-edges and out-edges to enable the adaptive EDGEMAPSPARSE/EDGEMAP-DENSE update models. FlashGraph [41] maintains both in-edges and out-edges with the CSR/CSC format to support different graph algorithms that require in-edges, out-edges or both in-edges and out-edges. On the other hand, for LOSC, it uses this graph representation to solve the inefficient subgraph construction problems of out-of-core systems. Based on the representation, LOSC proposes a locality-optimized subgraph construction scheme that improves the locality of memory access to greatly reduce the overheads of constructing subgraphs, which is the key contribution of the paper.

3.2. Locality-optimized subgraph construction

As mentioned in Section 2.3, the subgraph construction phase significantly degrades the overall performance of out-of-core graph processing systems due to a large amount of random memory accesses. To solve this problem, LOSC implements a locality-optimized subgraph construction scheme that significantly reduces the number of random memory accesses during the subgraph construction phase.

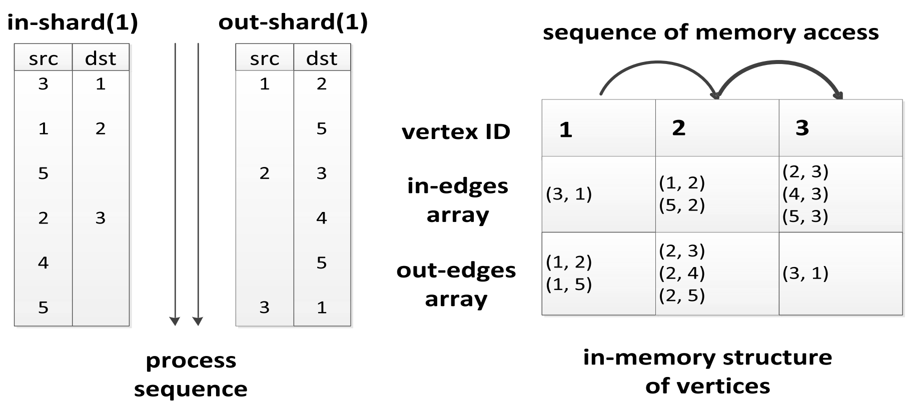
Algorithm 1 presents the procedure of locality-optimized subgraph construction scheme. The procedure of subgraph construction is to add in-edges/out-edges to the in-edge array/out-edge array of each vertex. For each in-edge, LOSC first accesses the memory address of its destination vertex, and then adds the edge to the in-edge array of the vertex. Similarly, for each out-edge, LOSC accesses the memory address of its source vertex, and adds the edge to the out-edge array of the vertex. Based on the dual-shard representation, the in-edges in the in-shards are sorted by the destination vertices and the out-edges in the out-shards are sorted by the source vertices. In this case, LOSC maximizes sequential memory access when adding the in-edges/out-edges to

Algorithm 1 Locality-optimized subgraph construction.

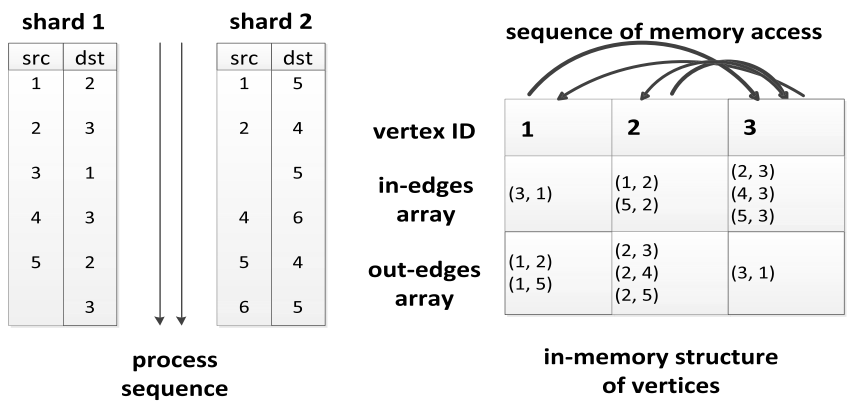
```
Input: Interval index number p
Output: Subgraph of vertices in interval p
 1: /* Initialization */
 2: $a \leftarrow interval(p).start$
 3: $b \leftarrow interval(p).end$
 4: $G \leftarrow InitializeMemoryForSubgraph(a, b)$
 5: /* Load in-edges in in-shard */
 6: $Inegdes \leftarrow in-shard(p).readfully()$
 7: /* Parallel in-edges adding */
 8: for each edge e in Inedges do
 9:     $G.vertex[e.dest].addInEdge(e.source)$
10: end for
11: /* Load out-edges in out-shard */
12: $Outegdes \leftarrow out-shard(p).readfully()$
13: /* Parallel out-edges adding */
14: for each edge e in Outedges do
15:     $G.vertex[e.source].addOutEdge(e.dest)$
16: end for
17: return G
```

the destination/source vertices and data access locality is exploited as much as possible when constructing subgraphs.

Fig. 5 provides an example to compare the locality-optimized subgraph construction with GraphChi's PSW subgraph construction. Both LOSC and GraphChi construct a subgraph for interval 1 of the graph in Fig. 4(a). As we see in Fig. 5(a), for interval 1, the access order of vertices to construct subgraph is 1, 2, 3 and these vertices are stored sequentially in memory. When LOSC executes the construction program, it first accesses the address of vertex 1, then it adds edge (3, 1) to the in-edge array of vertex 1 and adds edge (1, 2) and edge (1, 5) to the out-edge array in parallel. Afterwards, it successively accesses the memory addresses of vertex 2, vertex 3, and adds their in-edges and out-edges. However, for GraphChi, it requires many random memory accesses to add in-edges for vertices in interval 1 as shown in Fig. 5(b). When processing real-world graphs that have a large number of vertices and edges and complex structures, the inefficiency of GraphChi's



(a) LOSC's locality-optimized subgraph construction



(b) GraphChi's subgraph construction

Fig. 5. Comparison of constructing subgraphs.

PSW subgraph construction will become a severe problem for system performance.

3.3. Exploiting parallelism in subgraph construction

In order to further improve the performance of subgraph construction, LOSC fully utilizes the parallelism of multi-thread CPU when constructing subgraphs. Specifically, LOSC creates several worker threads that take charge of different vertices and their associated in-edges or out-edges. Once the in-shard or out-shard of a vertex interval is loaded into memory, these worker threads can fetch their own edges and add them to the in-edge arrays and out-edge arrays of the corresponding vertices in parallel. To this end, two edge index structures are created to indicate the in-edges and out-edges for each vertex as introduced in Section 3.6, so that worker threads can easily locate their own edges in the in-shards and out-shards. Since the in-edges in an in-shard and the out-edges in an out-shard are sorted by the destination and source vertices respectively, there is no write conflicts between these worker threads and no thread locks or atomic operations are required to maintain consistency. This enables high degree of parallelism when an in-shard/out-shard is large enough. Note that memory accesses and writes are still sequential as long as each thread reads and writes its own data.

Moreover, LOSC overlaps subgraph construction with edge loading as much as possible to make better use of parallelism. Specifically, the worker threads periodically check the loading progress of edges. As soon as the edges belong to a worker thread are loaded, the worker thread can immediately process these edges. For example, when loading in-shard (1) from disk in Fig. 5(a), the worker thread that takes charge of vertex 1 can add the edge (3, 1) into the in-edge array of vertex 1 once edge (3, 1) is loaded, while the I/O thread continues to load other edges from in-shard (1) simultaneously.

While for GraphChi, when parallelizing the subgraph construction, it will cause many write conflicts since in-edges in the memoryshard are sorted by the source vertices and the destination vertices are non-sequential. Due to a great reduction of random memory accesses and effective utilization of parallelism, the locality-optimized subgraph construction scheme significantly improves the system performance. We will quantitatively evaluate the efficiency of locality-optimized subgraph construction in Section 4.3.

3.4. Compact edge storage format

Although our graph representation improves the performance of subgraph construction, it takes more storage space than the existing graph representations since it stores both in-edges and out-edges. To solve this problem, we implement a compact edge storage format by combining several graph compression methods, i.e., compression of undirected graph, delta compression and ID compression.

Compression of undirected graph. For undirected graphs, LOSC stores each edge twice, one for each of the two directions. Actually, for an undirected edge $e = (u, v)$, e can be regarded as the in-edge and out-edge of u and v simultaneously. Therefore, for a vertex interval i, in-shard (i) and out-shard (i) are a duplicate of each other. To avoid this redundant storage, LOSC only maintains one

Table 2
Notations.

Notation	Definition
A	active vertex set in current iteration
M	size of an edge structure value
N	size of a vertex value
W	size of an edge weight value
C	average size of a compressed adjacency list
T_{rr}	random read bandwidth
T_{rw}	random write bandwidth
T_{sr}	sequential read bandwidth
T_{sw}	sequential write bandwidth

copy of edges for undirected graphs, i.e., only storing in-edges or out-edges of an interval.

Delta compression. In fact, each in-shard or out-shard consists of all adjacency lists of the vertices in an interval. The adjacency list of a vertex consecutively stores the vertex IDs of the vertex's neighbors. We can compress the adjacency lists by utilizing the delta values of vertex IDs. This is motivated by the locality and similarity in web graphs [4] where most links contained in a page lead the user to some other pages within the same host. In this case, the neighbors of many vertices may have similar vertex IDs. Instead of storing all vertex IDs in an adjacency list, LOSC stores the vertex ID of the first neighbors and the delta values of the vertex IDs of remaining neighbors.

ID compression. Current systems always store the ID as an integer of four-byte or eight-byte length. However, this can be wasteful if the IDs are of small values. LOSC adopts a variable-length integer [37] to encode each vertex ID. Thus, a minimum number of bytes are used to encode a given integer. Furthermore, the most significant bit of each compressed byte is used to indicate different IDs and the remaining seven bits are used to store the value. For example, considering an adjacency list of vertex *v1*, $adj(v1) = \{v2, v3, v4\}$. Supposing that the IDs of *v2*, *v3* and *v4* are 2, 5, and 300, the adjacency list of vertex *v1* is stored as “00000010 10000011 00000010 00100111”. The first byte is the id of 2, and the second byte is the delta value between 2 and 5 (removing the most significant bit). The third byte and the fourth byte have the same most significant bit, which means that they are used to encode the same ID. 00000100100111 (after removing the most significant bit of the third and fourth byte) is the delta value between 300 and 5.

By combining these compression techniques, the compact edge storage format can significantly reduce disk storage consumption, which further reduces I/O traffic and improves system performance, as shown in the evaluation results in Section 4.4.

3.5. I/O cost analysis

The I/O cost can be calculated by the total size of data accessed divided by the random/sequential bandwidth of disk access. Let M, N, W, C respectively be the size of an edge structure value, the size of a vertex value record, the size of an edge weight value and the average size of the compressed adjacency list of a vertex. In addition, T_{rr}, T_{rw}, T_{sr} and T_{sw} respectively represent random read, random write, sequential read and sequential write bandwidth (MB/s).

For easy reference, we list the notations in Table 2. For LOSC, during one pass of the whole graph, each edge is accessed twice from the disk, once in each direction. So the total I/O amount of edges is $2|V|C$. Furthermore, the I/Os of edge weights are avoided and only vertex values are updated since we store mutable data in vertices. Therefore, the I/O cost of LOSC C_{LOSC} can be stated constantly as:

$$C_{LOSC} = \frac{|V| \times (2C + N)}{T_{sr}} + \frac{|V| \times N}{T_{sw}}$$

Note that, in above I/O analysis, we do not consider the random accesses of vertices when the vertices are too large to be cached in memory, for the following reasons. First, the random accesses of vertices depend on different algorithms, graphs and iterations, which makes it very hard to quantify the I/O costs. Second, as shown by [3], the memory mapping method we used can significantly mitigate the issue of random accesses of vertices. Moreover, since the size of vertices is usually much smaller than the size of edges, the memory capacity of modern machines can easily fit in the vertices of most graphs [3].

While for GraphChi, in the best case, both endpoints of each edge belong to the same vertex interval, it is read only once from disk, otherwise, it is read twice. If the update function modifies edges in both directions, the number of writes is exactly the same; if in only one direction, the number of writes is half as many. Therefore, the I/O cost of GraphChi C_{GC} can be stated as:

$$\frac{|E| \times (M + W) + |V| \times N}{T_{sr}} + \frac{|E| \times W + |V| \times N}{T_{sw}} \leq C_{GC}$$
$$\leq \frac{2|E| \times (M + W) + |V| \times N}{T_{sr}} + \frac{2|E| \times W + |V| \times N}{T_{sw}}$$

Due to use of the compact edge storage format, the storage size of edges of LOSC is much smaller than that of GraphChi, which means $|V| \times C \ll |E| \times M$. Moreover, GraphChi has to write a large amount of intermediate data (edge weights) to disk for subsequent computation. Based on these analyses, we show that LOSC exhibits much higher I/O efficiency than GraphChi.

3.6. Benefit-aware scheduling scheme

Current out-of-core graph processing systems [18,26,43] are usually optimized for the sequential performance of disk drives and eliminate random I/Os by scanning the entire graph data in all iterations of graph algorithms. However, for many graph algorithms (e.g., Breadth-first Search, Weak Connected Components, Single Source Shortest Path) that access only small portions of data during each iteration, this full I/O model can be wasteful. For example, Breadth-first Search only visits vertices in a frontier in each iteration. On the other hand, the on-demand I/O model that is based on the active edges (the edges that have active sources vertices and impact the subsequent computation) can avoid loading the useless data. Unfortunately, it incurs a large amount of small random disk accesses due to the randomness of the active vertices. As we know, random access to disk drives delivers much less bandwidth than sequential access. Therefore, only accessing the useful data for out-of-core graph processing is an overkill when the number of active vertices is large. To address this dilemma and improve I/O performance, we adopt a benefit-aware scheduling scheme that skips loading and processing inactive edges in each iteration whenever such skipping can bring performance benefit.

The benefit-aware scheduling scheme adaptively schedules the edge loading based on the number of active edges. When the number of active edges is small, the system only traverses the active edges to avoid the loading of useless data, which improves I/O efficiency. Specifically, LOSC only loads the out-edges of active vertices and adopts a push-style processing to update their out-neighbors. Furthermore, LOSC enables atomic operations when updating vertices to ensure the consistency. When the number of active edges is large, the system simply loads all in-edges and out-edges to eliminate random accesses, and uses pull-style processing like GraphChi. To achieve this dynamic scheduling, the benefit-aware scheduling scheme incorporates the designs of vertices indices, bitmap operation and I/O-based benefit evaluation model.

Vertices indices. To enable the fast loading of the active edges, we maintain the indices to the edges for each vertex. Specifically, we associate two index structures, in-index and out-index. In-index (v) points to the position where the first in-edge of v stores, and out-index (v) points to the position where the first out-edge of v stores.

Bitmap operation. Selective scheduling of the active edges needs to scan all vertices to identify the active vertices. To achieve this, we associate two bitmaps, a state bitmap and an interval bitmap. The state bitmap whose storage size is $|V|/8$ bytes records whether a vertex is active or not and is divided into P intervals like the vertices. The interval bitmap records whether an interval contains at least one active vertex, which has a storage size of $P/8$ bytes.

In each iteration, LOSC first scans the interval bitmap to skip the intervals without any active vertices. This can significantly reduce the scanning time for the algorithms with very few active edges in each iteration. At the end of each iteration, LOSC updates the interval bitmap based on the newly updated state bitmap for the next iteration. The bitmap operation is also used in other systems like G-Store [17] for selective scheduling. However, LOSC maintains two bitmaps, which can not only achieve a more fine-grained scheduling but also significantly reduce the scanning time.

I/O-based benefit evaluation model. To evaluate whether only loading the active edges can bring performance benefit, the key is to compare the I/O costs between sequentially loading all edges and randomly loading the active edges. The former can be stated as C_{LOSC} as analyzed in Section 3.5. For the latter, supposing that the active vertex set in current iteration is A, so the I/O amount of the active edges is equal to the size of all out-edges of vertices in A. Moreover, LOSC also loads the out-index of each active vertex so as to locate the active edges, in addition to the vertex values. Therefore, the I/O cost $C_{LOSC'}$ can be stated as:

$$C_{LOSC'} = \frac{\sum_{v \in A}(out-index(v+1) - out-index(v))}{T_{rr}} + \frac{2|V| \times N}{T_{sr}} + \frac{|V| \times N}{T_{sw}}$$

If $C_{LOSC'} \le C_{LOSC}$, the system selectively loads the active edges to avoid the loading of useless data. Otherwise, the system just loads all in-edges and out-edges to eliminate random disk accesses. The disk access bandwidth T_{rr}, T_{rw}, T_{sr} and T_{sw} can be measured by using several measurement tools such as fio [26] before we conduct the experiments. And other parameters can be directly collected and computed in the runtime. This provides an accurate performance prediction that enables efficient scheduling.

In summary, the benefit-aware scheduling scheme brings two advantages to LOSC. First, by intelligently skipping loading the inactive edges, the system can significantly reduce the I/O traffic and improve I/O performance with negligible computing overheads. Second, the reduced I/O traffic can also lead to lower overheads of subgraph construction since less edges are loaded into memory to construct subgraphs. We will quantitatively evaluate the benefits and computing overheads of the benefit-aware scheduling scheme in Section 4.5.

3.7. *Lightweight replication of interval vertices*

As shown in Section 2.1, each vertex computes its new value in parallel in the vertex-centric computing model. However, if two vertices in the same vertex interval have a common edge, e.g., vertex 1 and vertex 2 in Fig. 4(c), they cannot be updated in parallel as update sequences of these vertices have an influence on the computing result. For example, when vertex 2 is updated, it reads the value of vertex 1. If vertex 1 is updated earlier, vertex 2 will obtain the latest value of vertex 1. Otherwise, it will obtain the value of the last iteration. To solve this problem, GraphChi implements a deterministic parallelism in which vertices of the same interval are updated sequentially if they share a common edge. Although this method eliminates race conditions, it limits the utilization of CPU parallelism and reduces the computation efficiency.

To solve the above problem, LOSC adopts a lightweight replication of interval vertices to eliminate race conditions while enabling full CPU parallelism. Concretely, LOSC maintains two copies of the interval vertices, Latest-copy and Old-copy, when executing a vertex interval. Latest-copy stores the latest values and is updated during the computation. Old-copy serves as the in-neighbors and is read by other vertices, storing the values of the last iteration. Consequently, all vertices in an interval can access their read-only in-neighbors and execute update function in parallel, and the update sequence of vertices will not affect the computing result. Since LOSC just replicates the vertices in an interval, it will not cause much memory pressure. After a vertex interval is processed, LOSC synchronizes the vertex values by replacing the Old-copy with the Latest-copy.

Note that, the lightweight replication of interval vertices is applied only when LOSC loads all edges and adopts the pull-style vertex updating. When LOSC selectively loads the active edges and adopts the push-style vertex updating, LOSC needs not to copy the interval vertices since it only involves write operations.

3.8. *Workflow example*

We now use an example to illustrate the main workflow of LOSC in detail. LOSC processes the input graph one vertex interval at a time. The processing of an interval consists of four steps: 1) load edges; 2) construct subgraph; 3) parallel update; 4) synchronize vertex values. Fig. 6 shows an example of processing on interval 1 of the graph in Fig. 4(a) when all vertices are active and only vertex 2 is active.

Load edges. The loading phase of LOSC is very simple but I/O-efficient. As we see in Fig. 6(a), LOSC concurrently loads the in-edges from the in-shard and out-edges from the out-shard for interval 1 (shards in shaded color are loaded into memory), which maximizes the sequential disk accesses.

Construct subgraph. When the edges are loaded into memory, LOSC starts the locality-optimized subgraph construction for the interval as described in Section 3.2. LOSC sequentially accesses the memory addresses of vertices 1, vertex 2, vertex 3, and adds their in-edges and out-edges.

Parallel update. After the subgraph is constructed, LOSC executes a user-defined update program for each vertex of the current interval in parallel. When a vertex is updated, it first reads the values of its in-neighbors and produces an aggregated value. Then, the user-defined update program takes this value as input and updates the value of the vertex. Algorithm 2 shows an example update program that computes PageRank of an input graph. In addition, for the interval vertices, e.g., vertex 1, 2, 3 in Fig. 6(a), LOSC maintains two types of values (Latest-copy and Old-copy) to enable full CPU parallelism while ensures the consistency of computation.

Synchronize vertex values. When all vertices of an interval have been updated, LOSC directly updates the values of the Old-copy of interval vertices (e.g., vertex 1, 2, 3) with the values of the Latest-copy. Unlike previous systems [18,26] that write the updated edge attributes back to the disk for subsequent processing, synchronization of vertices significantly reduces disk I/Os and improves the system performance.

In Fig. 6(b) where only vertex 2 is active, the benefit-aware scheduling scheme only loads the out-edges of vertex 2 into memory and pushes updates to its neighbors (vertex 3, 4, 5). In this

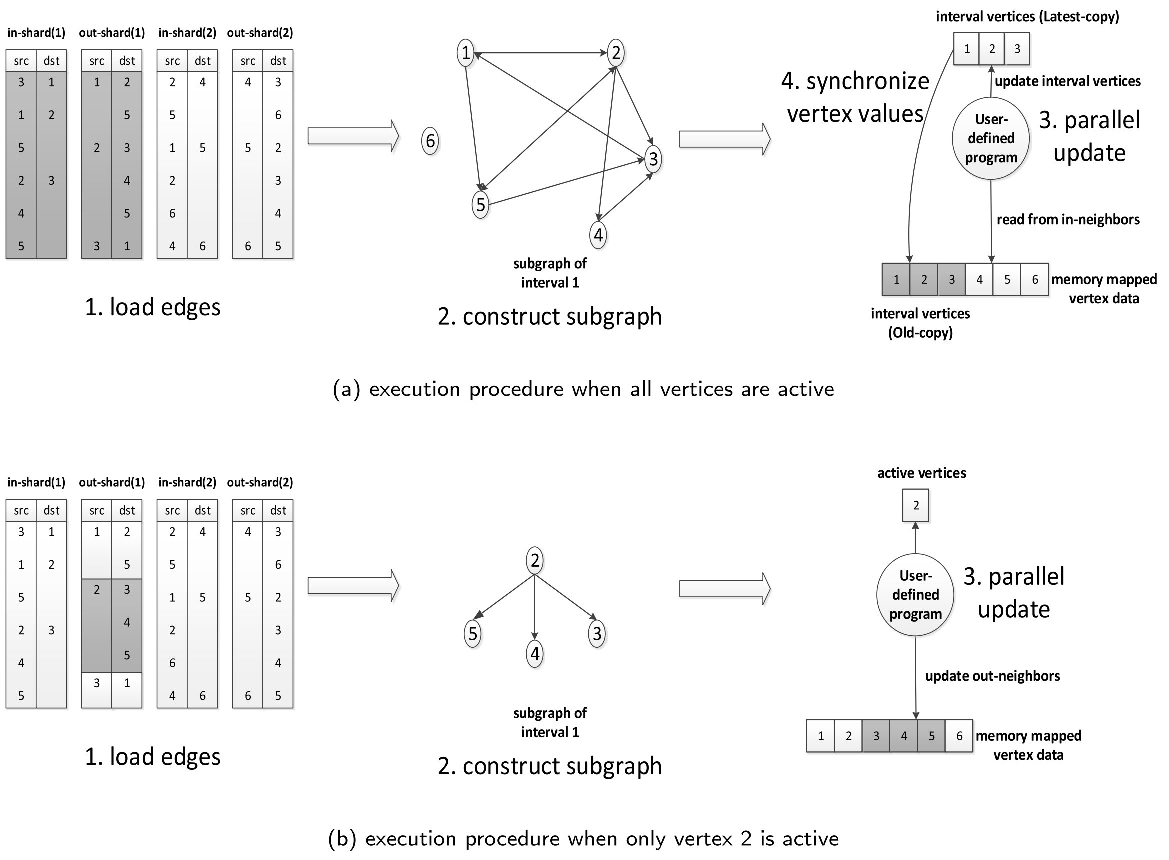


Fig. 6. An Example of the LOSC workflow.

Algorithm 2 Update function (v): PageRank.

```
1: Procedure PageRank
2: /* gather: read values from in-neighbors */
3: for each edge e of v.inedges() do
4:     $src \leftarrow e.source$
5:     $sum \leftarrow sum + src.value/src.outdegree$
6: end for
7: /* apply: update the value */
8: $pagerank \leftarrow 0.15 + 0.85 \times sum$
9: $v.value \leftarrow pagerank$
10: End Procedure
```

case, LOSC can avoid loading the redundant edges of inactive vertices that have no impact on the computation. This results in fewer disk I/Os and further reduces the overheads of subgraph construction and vertex updating.

4. Evaluation

In this section, we present experimental evaluation of our system LOSC in comparison with state-of-the-art out-of-core graph processing systems.

4.1. Experiment setup

Platform and Datesets. The hardware platform used in our experiments is an 8-core commodity machine equipped with 12 GB main memory and 600 GB 7200 RPM HDD, running Red Hat 4.8.5. In addition, a 128 GB SATA2 SSD is installed for evaluating the scalability.

Datasets used for the evaluation are summarized in Table 3. LiveJournal, Twitter2010 and Friendster are social graphs, showing the relationship between users within each online social network. UK2007 and Ukunion are web graphs that consist of hyperlink

Table 3
Datasets used in evaluation.

Dataset	Vertices	Edges	Type
LiveJournal	4.8 million	69 million	Social Network
Twitter2010	42 million	1.5 billion	Social Network
Friendster	66 million	1.8 billion	Social Network
UK2007	106 million	3.7 billion	Web Graph
UKunion	133 million	5.5 billion	Web Graph
Kron30	1 billion	32 billion	Synthetic Graph

relationships between web pages, with larger diameters than social graphs. Kron30 is generated with the Graph500 generator [2]. The in-memory graph LiveJournal is chosen to evaluate the performance of in-memory processing and the scalability of LOSC. The other five graphs Twitter2010, Friendster, UK2007, UKunion and Kron30 are larger than memory by 2.1×, 2.6×, 5.2×, 7.9× and 21.3× respectively.

We use several benchmarks algorithms in our evaluation to show the applicability of LOSC: PageRank (PR), Sparse Matrix Vector Multiply (SpMV), Breadth-first search (BFS), Weak Connected Components (WCC), and Single Source Shortest Path (SSSP). These algorithms exhibit different I/O access and computation characteristics, which provides a comprehensive evaluation of LOSC. For PageRank, we run five iterations on each graph. For SpMV, we run one iteration to calculate the multiplication result. For BFS, WCC and SSSP, we run them until convergence.

Systems for Comparison. We mainly compare LOSC with two representative out-of-core systems that use the vertex-centric and edge-centric model respectively, GraphChi [18] and GridGraph [43]. We also compare LOSC against the first version of LOSC (we name it as LOSC-v1) [34] to evaluate the effects of the newly proposed optimizations. In addition, we add the comparisons with other state-of-art out-of-core graph processing systems including Flash-

Table 4
Execution time (in seconds).

	PageRank	SpMV	BFS	WCC	SSSP
LiveJournal					
GraphChi	16.6	14.1	20.9	24.4	21.4
GridGraph	10.9	5.1	5.2	5.1	6.1
LOSC-v1	**2.7**	1.9	**3.7**	**4.1**	**4.0**
LOSC	2.8	**1.9**	3.9	4.3	4.1
Twitter2010					
GraphChi	928.6	371.4	1624.3	913.7	1913.9
GridGraph	451.9	197.2	598.9	522.5	660.4
LOSC-v1	126.5	57.6	230.1	176.3	249.2
LOSC	**107.8**	**49.2**	**108.6**	**103.3**	**149.4**
Friendster					
GraphChi	2562.8	568.8	2294.5	2612.3	1802.4
GridGraph	1009.4	371.4	578.6	526.8	708.6
LOSC-v1	230.2	70.8	473.4	481.3	376.2
LOSC	**214.4**	**59.1**	**197.3**	**300.8**	**171.2**
UK2007					
GraphChi	2812.5	1160.7	7154.5	6862.8	7495.8
GridGraph	1242.2	511.2	6025.2	4783.8	7029.4
LOSC-v1	265.1	121.7	1172.2	864.7	1171.4
LOSC	**233.7**	**111.1**	**459.7**	**411.2**	**616.5**
Ukunion					
GraphChi	3376.6	1620.8	24062.3	5665.8	56650.9
GridGraph	1829.3	810.5	18929.2	13265.1	25554.2
LOSC-v1	390.1	178.9	13022.5	3513.9	18171.4
LOSC	**363.2**	**162.3**	**6593.7**	**2437.8**	**8653.1**
Kron30					
GraphChi	42770.5	24731.6	-	-	-
GridGraph	20935.8	11883.5	-	72781.2	-
LOSC-v1	6278.2	3510.8	55772.2	16173.6	66932.4
LOSC	**3923.9**	**2065.2**	**37181.8**	**9513.9**	**41832.6**

"-" indicates that the system failed to finish execution in 48 hours.

Graph [41], GraphZ [42], G-Store [17] and NXGraph [7] to further evaluate the performance of LOSC. Since LOSC's compact storage format may fit whole graph data into memory for several datasets like Twitter2010, and makes it unfair to compare with other systems. We provide 8 GB memory budget, 8 execution threads for the executions of all algorithms for fair comparison. Under 8 GB memory, only the LiveJournal graph can be fit into memory, while other graphs require access to disks.

4.2. Overall performance

We first report the execution time of the chosen algorithms on different graphs in Table 4. Here, we compare LOSC with GraphChi and GridGraph, two representative and widely-used out-of-core graph processing systems that use vertex-centric and edge-centric computing model respectively. In addition, we compare LOSC with LOSC-v1 to see how many performance improvements the newly proposed optimizations can bring. On average, LOSC outperforms GraphChi, GridGraph and LOSC-v1 by 9.4×, 5.1× and 1.5× respectively.

The speedup over GraphChi mainly derives from the significant reduction in time spent on subgraph construction and disk I/Os. PR and SpMV are based on standard matrix-vector multiplication in which all vertices participate in the computation and the I/Os are sequential. So they are computation-intensive algorithms, which makes subgraph construction dominate the execution time. For these algorithms, on average LOSC speeds up graph processing by 10.1× and 10.6× respectively, compared with GraphChi. BFS, WCC and SSSP are traversal algorithms that produce many random I/O accesses. Furthermore, they require less computation since only a portion of the whole vertex set participates in the computation. Therefore, they are I/O-intensive algorithms in which the disk I/O costs become the key factor on the system performance. Thanks to the significant reduction in disk I/Os due to the compact edge storage format and benefit-aware scheduling scheme, LOSC outperforms GraphChi by 10.2×, 7.5× and 8.9× on these three algorithms respectively.

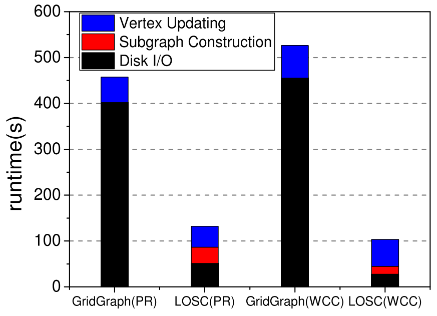


Fig. 7. Runtime breakdown on Twitter2010.

For GridGraph, although it does not need to construct vertex-centric subgraphs since the computation is based on streaming the edge lists, it still has a worse performance than LOSC. To further explain the reasons, we also report the runtime breakdowns of PageRank and WCC on Twitter2010 for both LOSC and GridGraph, as shown in Fig. 7. From the results, we can see that the better performance of LOSC mainly stems from its much lower disk I/O overheads. This attributes to LOSC's compact edge storage format and benefit-aware scheduling scheme. Although GridGraph also supports selective scheduling to reduce the loading of inactive edges, it is very coarse-grained since it can only skip processing the edge blocks without any active edges. This means it loads and processes an edge-block even though there is only one active edge, which still produces many redundant data loading.

Moreover, for in-memory settings (LiveJournal), LOSC still outperforms GraphChi and GridGraph. This is because LOSC can make better use of thread-level parallelism by using a lightweight replication of interval vertices, which leads to higher computational efficiency. However, when compared with the all-in-memory shared-memory systems [28,38], out-of-core systems usually deliver a worse performance. For example, Ligra [28] only takes 0.3 s to run BFS on LiveJournal in our experiment platform. The performance gap is attributed to the overheads incurred by disk I/O optimizations of out-of-core systems and the specific optimizations of parallel algorithms adopted by the shared-memory systems. This further reflects that out-of-core graph processing systems are more suitable for handling large-scale graphs beyond the memory. When a machine with large capacity of memory is available, it is more reasonable to use the shared-memory graph processing systems.

When compared with LOSC-v1, the use of the benefit-aware scheduling scheme and parallelized subgraph construction can improve the performance by up to 2.4×. In addition, we can observe that these newly proposed optimizations bring few benefits when processing the small graph LiveJournal. This is because the LiveJournal graph can fit in memory and the vertex updating time dominates the overall runtime.

We also evaluate the preprocessing time of different systems. The preprocessing time consists of loading raw data into memory, partitioning and compressing the graph. As shown in Fig. 8, LOSC takes more preprocessing time than the other systems, since it needs to build two copies of edges and implement the space-efficient storage format. This may limit LOSC's efficiency and make it not appropriate for problems with dynamic graph structures. However, the overhead of the extra preprocessing is more than offset by the significant performance improvement it brings in almost all cases except for the SpMV algorithm, according to results in Table 4. For example, the preprocessing time of LOSC for Twitter2010 is longer than that of GridGraph by 218.7 s, the algorithm execution time of LOSC is less than that of GridGraph by 344.1 s, 148

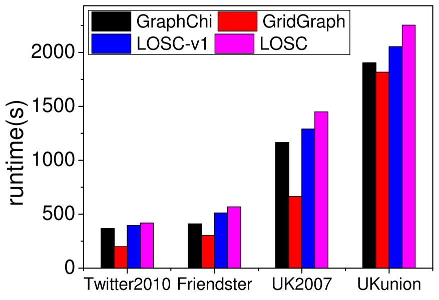


Fig. 8. Preprocessing time.

s, 490.3 s, 419.2 s and 511 s for PR, SPMV, BFS, WCC and SSSP respectively. This indicates the preprocessing time may outweigh the benefits of proposed optimizations when handling the graph algorithms with lightweight computation and I/O loads or few iterations like SpMV. In addition, the preprocessing phase of graphs is usually off-line. Therefore, the graphs can be reused for many times after preprocessing, and the preprocessing overheads can be significantly amortized.

4.3. Effect of locality-optimized subgraph construction

In this subsection, we compare LOSC with GraphChi that also needs subgraph construction in terms of subgraph construction overheads, cache miss rates as well as the effect of thread parallelism. Specifically, we implement LOSC-PSW that constructs subgraphs by using the PSW method [18] of GraphChi. Note that LOSC-PSW also uses GraphChi's graph partition and organization format. For more exact evaluation, we disable the benefit-aware scheduling scheme since it also influences the subgraph construction time. Fig. 9 shows the time cost of subgraph construction on Twitter2010 and UK2007. We find that LOSC exhibits high efficiency of subgraph construction and achieves an average speedup of 28.6× and 2.9× compared with PSW and LOSC-v1. This is attributed to a great reduction of random memory accesses and effective utilization of parallelism during the subgraph construction phase.

To further demonstrate how the locality-optimized subgraph construction scheme improves the performance of subgraph construction, we first measure the number of memory reads/writes and cache misses during the subgraph construction phase using Cachegrind [1], a tool to simulate memory, the first-level and last-level caches etc. Here, we just report the number of memory reads and writes, last-level cache read and write misses (LL misses). The focus on the last-level cache stems from the fact that it has the most influence on the time of subgraph construction, as it masks accesses to main memory and a last-level cache miss can cost as much as 200 cycles [1]. For the ease of measure, we run 1 iteration of BFS on the small graph, LiveJournal, and summarize the results in Table 5. We observe that the LL miss rate of LOSC-PSW is much higher than LOSC. This means that the locality of memory access is exploited better and CPU is able to do more work on data residing in the cache for subgraph construction of LOSC. For LOSC-PSW, CPU has to frequently access memory to read data, which significantly increases the access latency.

Then we evaluate the effect the number of threads on subgraph construction. Fig. 10 shows the results of BFS on LiveJournal and UK2007. When the number of threads increases from 1 to 8, the performance of subgraph construction for LOSC-PSW and LOSC improves by 1.2× and 3.3× on average respectively. This indicates that LOSC makes better use of parallelism, since there are no write conflicts between the threads that take charge of different vertices and edges as introduced in Section 3.3.

Table 5
Memory access and cache miss.

System		Read	Write
LOSC-PSW	mem. refs	416278519	212376955
	LLC misses	32053445	49059076
	LLC miss rate	7.7%	23.1%
LOSC	mem. refs	410852346	205426173
	LLC misses	1608991	6183666
	LLC miss rate	0.4%	3.0%

4.4. Effect of compact edge storage format

We evaluate the effects of the compact edge storage format on storage space, I/O traffic and runtime of algorithms. To evaluate the storage space consumption, We compare LOSC with GraphChi, GridGraph, FlashGraph [41], Ligra+ [29] and G-Store [17]. These systems use different storage formats or compression techniques, which provides a comprehensive evaluation of the effectiveness of LOSC's compact storage format. In addition, we compare LOSC with the baseline implementation without using the compact storage format (LOSC-without). Fig. 11(a) compares the required disk space of these systems. We observe that the storage of LOSC is efficient even though it stores two copies of each edge. Specially, the storage usages of GraphChi and GridGraph that use CSR and edge list to store the graphs respectively are 1.4× and 3.1× higher than those of LOSC on average. Compared with LOSC-without, the compact edge storage format can save storage usages by up to 76%.

FlashGraph also stores both the in-edges and out-edges in the CSR/CSC format, but without compression. Therefore, the storage sizes of FlashGraph are almost equal to those of LOSC-without. Ligra+ and G-Store implement a compact storage by using different compression methods. Ligra+ also enables delta compression. In addition, it utilizes run-length encoded byte codes for vertex encoding. G-Store utilizes the symmetry present in graph data, which is similar to the compression of undirected graph in LOSC. Furthermore, it also enables compression of ID by removing the redundancy of the most-significant-bits (MSBs) of IDs of source and destination vertices within a partition. Unlike these systems, LOSC compresses IDs by using the variable-length integer to encode vertices IDs of different values, as well as storing the delta values of IDs. As shown in Fig. 11(a), the storage sizes of LOSC are respectively 1.1× and 1.5× smaller than those of Ligra+ and G-Store on average.

Fig. 11(b) and Fig. 11(c) shows the benefits of the compact edge storage format on I/O traffic and runtime when running PageRank on different graphs. We can see that the compact edge storage format can significantly reduce the amount of I/O traffic, leading to better algorithm performance. Specifically, the total amount of I/O traffic and runtime can be reduced by 58% and 43% on average when using the compact edge storage format.

4.5. Effect of benefit-aware scheduling scheme

We first evaluate the benefits of the benefit-aware scheduling scheme. To this end, we compare LOSC with a baseline implementation that disables the benefit-aware scheduling scheme (LOSC-d), in terms of I/O traffic, subgraph construction overheads and overall performance of algorithms. The evaluation results are shown in Fig. 12, running BFS and WCC on different graphs. To intuitively show the comparisons of LOSC and LOSC-d, we report the normalized results.

Benefits on I/O traffic. Fig. 12 (a) and (b) show the comparisons of I/O traffic. For BFS and WCC where the number of active edges is small in most iterations, the benefit-aware scheduling scheme can effectively avoid the loading of useless data, significantly reducing

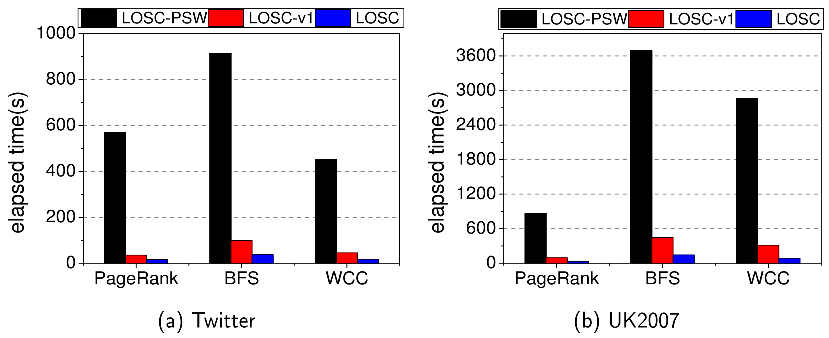


Fig. 9. Time cost of subgraph construction.

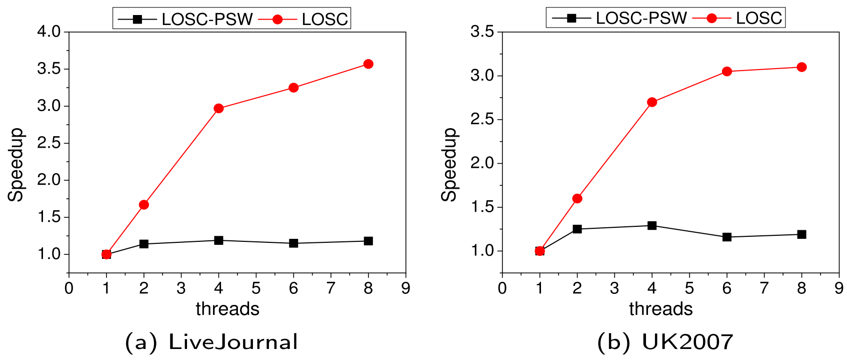


Fig. 10. Effect of threads number on subgraph construction.

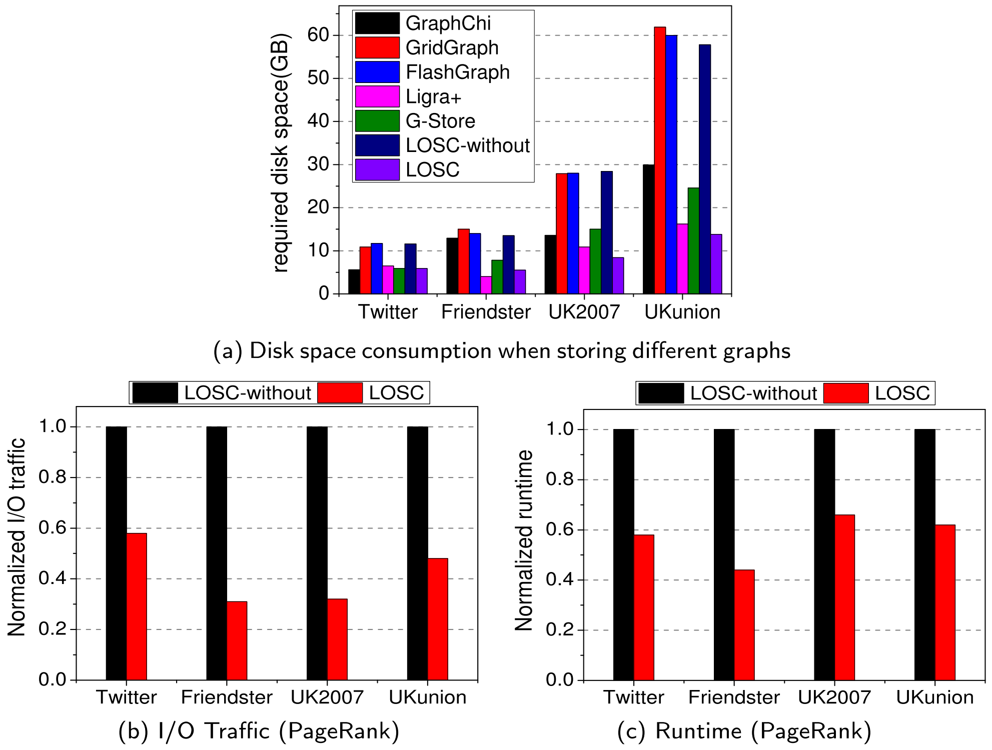


Fig. 11. Evaluating the benefits of compact edge storage format.

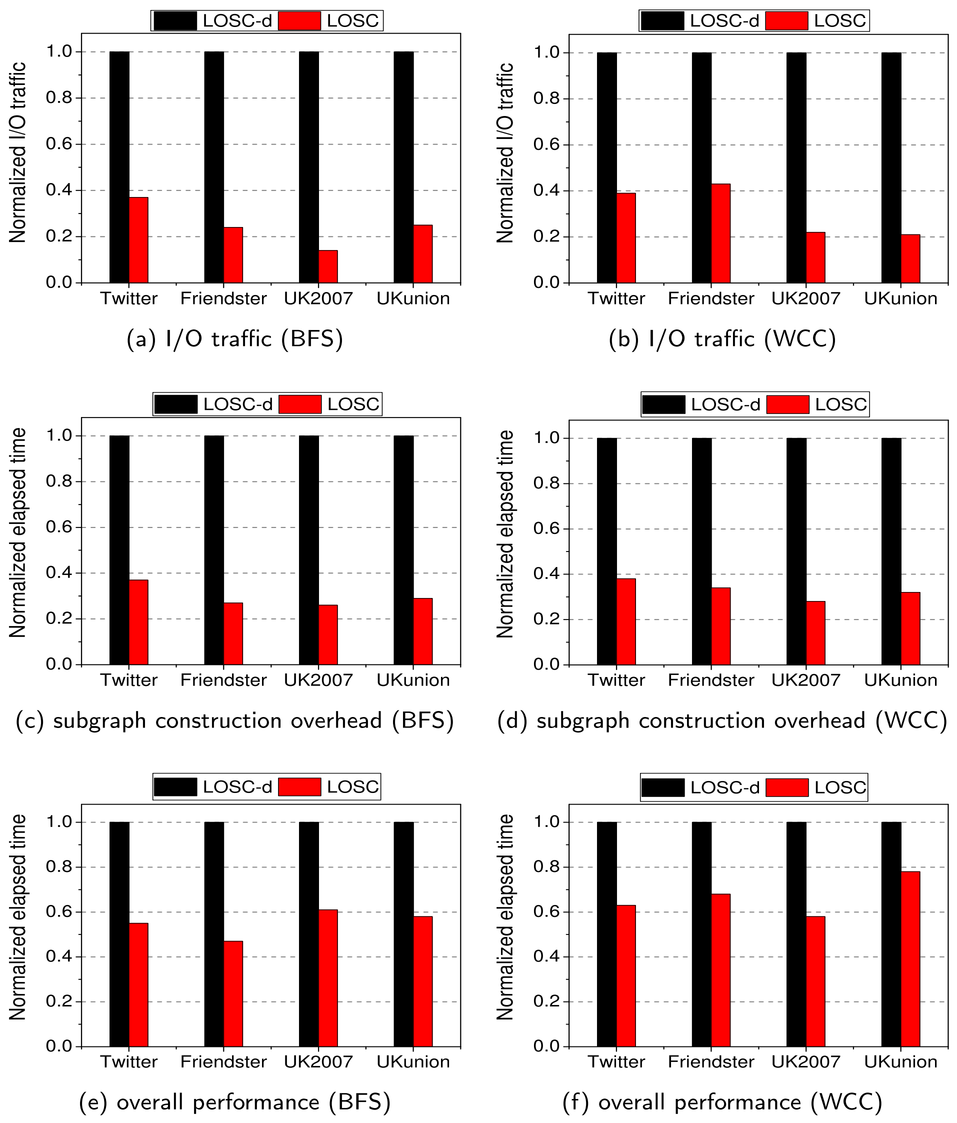


Fig. 12. Evaluating the benefits of benefit-aware scheduling scheme.

the I/O traffic. Specifically, the total amount of I/O traffic can be reduced by 75% and 68% for BFS and WCC respectively.

Benefits on subgraph construction. Fig. 12 (c) and (d) compare LOSC and LOSC-d in term of subgraph construction overheads. The less I/O traffic enabled by the benefit-aware scheduling scheme further reduces the subgraph construction overheads, since less edges are loaded to memory to construct subgraphs. Specifically, the subgraph construction overheads can be reduced by 70% and 66% for BFS and WCC respectively.

Benefits on overall performance. The combined effects of less I/O traffic and faster subgraph construction bring an improvement on overall performance, as shown in Fig. 12 (e) and (f). Specifically, the benefit-aware scheduling scheme can improve the overall performance by 44% and 33% for BFS and WCC respectively.

Then we evaluate the overheads of the benefit-aware scheduling scheme since it will perform the benefit evaluation in each iteration and produce extra computation overheads. Specifically, we compare the computation overheads (benefit evaluation) with the reduced I/O time enabled by the benefit-aware scheduling scheme. As shown in Fig. 13, we can see that the extra computation overheads are negligible. For example, the computation time for benefit evaluation of BFS is only 1.4 s on Twitter2010, while the corresponding reduced I/O time is 58.1 s.

4.6. Memory usage

Fig. 14 shows the maximum memory usage comparison of the three systems when running PageRank, BFS and WCC on Twitter2010 and UK2007. We can see that GridGraph has the minimum memory consumption as it only maintains one copy of edges. For LOSC, due to no need to store edge values, it has less memory consumption than that of GraphChi. In addition, LOSC has less memory consumption when running PageRank, since this algorithm only requires the in-edges to finish computation. Even though, we still plan to seek a more compact in-memory storage structure to mitigate the memory pressure in the future works.

4.7. Scalability

We evaluate the scalability of LOSC by observing the improvement when more hardware resource is added. Fig. 15(a) shows the speedup of different systems when running PageRank on LiveJournal using different numbers of threads. We observe that GridGraph and LOSC improves the performance as the number of threads increases. For GridGraph, it enables parallel processing by overlapping the vertex updating and edge streaming [43]. For LOSC, it makes full use of parallelism by using a lightweight replica-

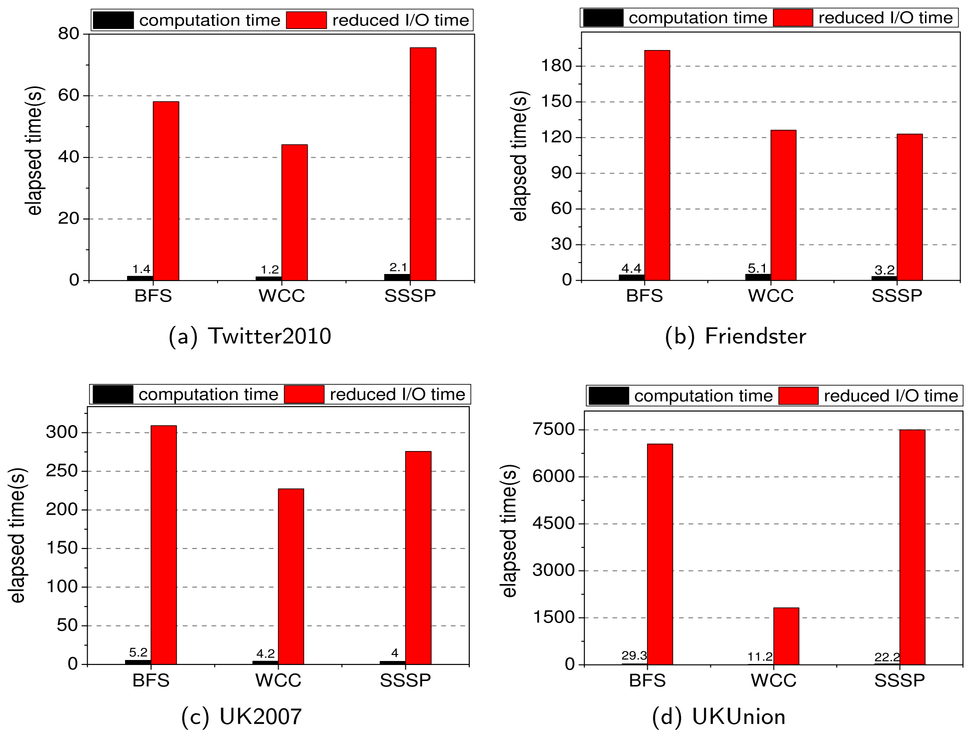


Fig. 13. Evaluating the overheads of benefit-aware scheduling scheme.

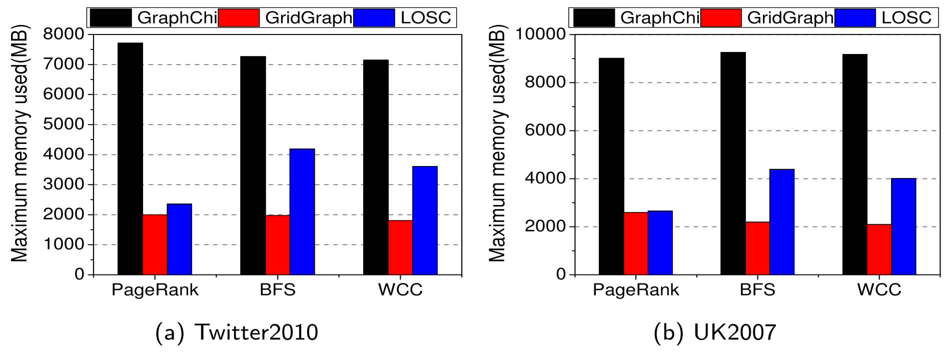


Fig. 14. Maximum memory usage.

tion of interval vertices as introduced in Section 3.7. On the other hand, GraphChi shows poor scalability as we increase the number of threads. The main blame is GraphChi's deterministic parallelism that limits the utilization of multi-threads [18]. Fig. 15(b) shows the results when running BFS on UK2007. Since system performance is limited by disk I/O, thread number has relatively less impact on the performance. Among these systems, LOSC achieves the best scalability. Specifically, when the number of thread increases from 1 to 8, the performance of GraphChi, GridGraph and LOSC are improved by 8%, 12% and 39% respectively. This is because LOSC has the best I/O performance, so that the vertex updating time occupies a larger proportion in overall execution time compared to other two systems and thread parallelism contributes to more performance improvement.

Fig. 15(c) shows the performance improvement of BFS on UK when using different I/O devices. Compared with disk performance, GraphChi, GridGraph and LOSC achieve a speedup of 1.3×, 1.8× and 1.7× respectively when using SSD. This indicates that LOSC and GridGraph can benefit more from the utilization of SSD, since the key performance bottleneck of them is the disk I/O costs. For GraphChi in which the subgraph construction is also a performance bottleneck, using fast I/O devices can only bring limited improvement.

Fig. 15(d) shows the performance variations of PageRank on Twitter graph under different memory budgets. With the memory increased from 2 GB to 12 GB, LOSC achieves a speedup of 6.2×, much higher than the speedups of GraphChi (1.3×) and GridGraph (1.2×). This is because LOSC can fit the whole graph data into memory under 12 GB memory, thanks to the compact storage format.

4.8. Comparison with other systems

In Fig. 16, we compare LOSC with other state-of-art out-of-core graph processing systems including FlashGraph [41], GraphZ [42], G-Store [17] and NXGraph [7] to further evaluate the performance of LOSC. FlashGraph utilizes SSD arrays to implement a semi-external memory graph engine. It supports selective access of the edges requested by the applications in a random manner. GraphZ reduces the I/O costs and improves performance by using the degree-ordered storage and dynamic messages. G-Store is a high-performance graph store engine which delivers high throughput of graph data I/Os from SSDs, combined with judicious use of

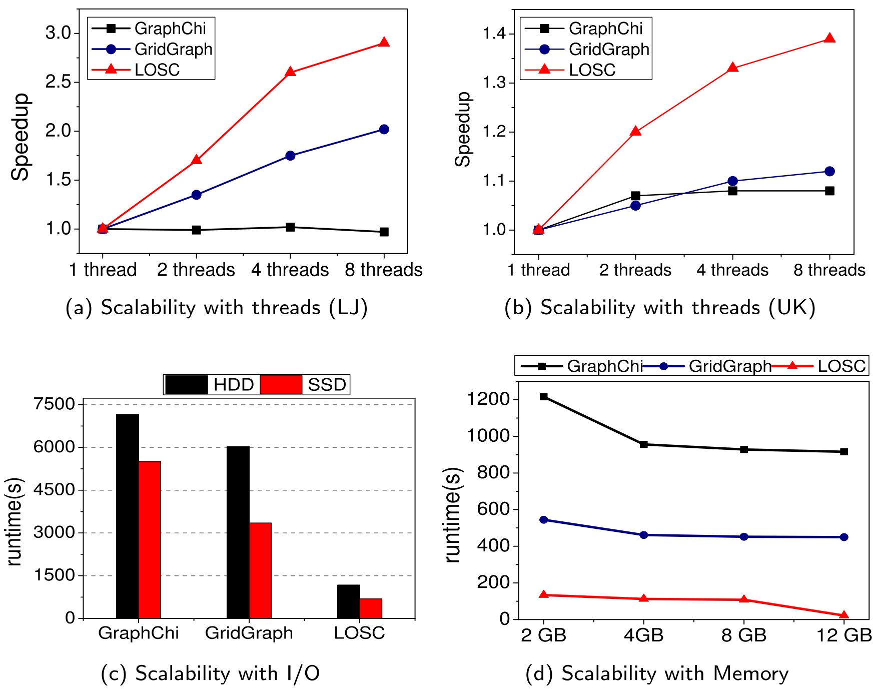


Fig. 15. Evaluation of scalability.

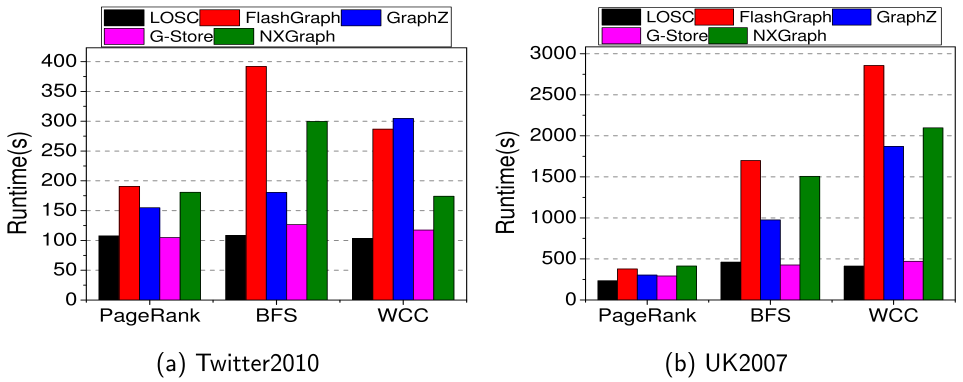


Fig. 16. Execution time of different graph processing systems.

main memory and cache. It also enables compact storage by using techniques such as ID compression. NXgraph provides three novel update strategies under the Destination-Sorted Sub-Shard (DSSS) structure so as to further ensure locality of graph data access. Among them, FlashGraph and GraphZ are vertex-centric graph processing systems and G-Store and NXGraph are edge-centric graph processing systems.

For GraphZ and NXGraph, LOSC achieves average speedups of 2.3× and 2.7× respectively. Specifically, for each algorithm (PageRank, BFS and WCC), LOSC achieves average speedups of 1.4×, 1.9× and 3.8× respectively over GraphZ, and average speedups of 1.7×, 3.1× and 3.4× respectively over NXGraph. In particular, LOSC has a higher speedup when running BFS and WCC. This is because there are very few active edges in most iterations of these algorithms. In this case, the efficiency of LOSC's benefit-aware scheduling is significant. While for PageRank, the benefit-aware scheduling scheme does not take effect. Furthermore, the compact edge storage format also contributes to LOSC's better performance.

FlashGraph supports selective access by loading the active edges in a random manner, relying on expensive SSD arrays that deliver high I/O bandwidth. However, it incurs significant overheads when issuing frequent small random accesses to HDDs or one SSD. While for LOSC, it skips processing inactive edges in each iteration whenever such skipping can bring performance benefit. This results in a higher speedup over FlashGraph when running BFS and WCC, as shown in Fig. 16. Specifically, LOSC outperforms FlashGraph by average speedups of 1.7×, 3.7× and 4.9× for each algorithm respectively.

For G-Store, it enables high-performance storage by utilizing the symmetry present in graph data and removing the redundancy of the most-significant-bits (MSBs) of IDs of source and destination vertices within a partition. Furthermore, G-Store adopts a clever proactive caching strategy to make better use of memory. Thanks to these optimizations, G-Store has a better performance than other three systems. However, LOSC still achieves an average speedup of 1.2× over G-Store. This is attributed to the following two reasons. First, LOSC enables more compact storage by combining several graph compression methods as shown in Section 4.4, which can lead to less I/O traffic. Second, the benefit-aware scheduling scheme avoids more unnecessary I/Os for LOSC. Although G-Store can also support selective fetching, it relies on SSD arrays to achieve this like FlashGraph. While the benefit-aware scheduling scheme has better adaptability and works well for both SSD and HDD.

5. Related work

Many scalable graph processing systems have recently been proposed. In this section, we introduce three categories of existing graph processing systems: distributed systems, single-machine shared-memory systems and single-machine disk-based (out-of-core) systems.

5.1. Distributed systems

Distributed systems usually distribute a large graph into the compute nodes of a cluster by constructing node-resident subgraphs from the original graph, which enables them to utilize the aggregate memory of a cluster to achieve good scalability. Pregel [24] supports vertex-centric computing model following Bulk-Synchronous Parallel message passing model [31]. It abstracts away the complexity of programming in a distributed-memory environment and runs users' code in parallel on a cluster. However, this model usually suffers from expensive synchronization overheads. GraphLab [21] and PowerGraph [13] executes an asynchronous model and uses shared memory for communication among vertices instead of passing messages. Gemini [44] applies multiple optimizations targeting computation performance to build scalability on top of efficiency. Chaos [27], BlitzG [6] and TurboGraph++ [16] utilize secondary storage to scale distributed graph processing to out-of-core scenery. Giraph [8] and Shentu [19] exploit a powerful cluster with high performance interconnects to handle trillion-scale graphs. Gluon [9] introduces a new approach to building distributed memory graph analytics systems that exploit heterogeneity in processor types (CPU and GPU), partitioning policies, and programming models.

5.2. Single-machine shared-memory systems

Single-machine shared-memory systems typically use a high-end server with hundreds or thousands gigabytes of DRAM to hold the whole graph [11]. Ligra [28] is a lightweight shared-memory framework and provides a programming interface optimized for graph traversal algorithms. Polymer [38] is a NUMA-aware graph analytics system, which is motivated by a detailed study of NUMA characteristics. Garaph [22] fully exploits the power of modern hardware and efficiently support GPU-accelerated graph processing. Julienne [10] extends Ligra with an interface for maintaining a collection of buckets under vertex insertions and bucket deletions, to efficiently support bucketing-based graph algorithms. GraphIt [39] is a novel DSL for graph processing that generates fast implementations for algorithms with different performance characteristics running on graphs with varying sizes and structures.

5.3. Out-of-core systems

Out-of-core graph processing systems enable users to analyze, process and mine large graphs in a single PC by efficiently using disks. Current out-of-core graph processing systems mainly adopt two computing models, i.e., vertex-centric and edge-centric.

Vertex-centric systems. TurboGraph [15] inspired by GraphChi focuses on improving parallelism by overlapping the CPU and I/O processing with a novel concept called pin-and-slide, but it is applicable only to certain embarrassingly parallel algorithms [5]. VENUS [5] uses a vertex-centric streamlined computing model and proposes a new storage scheme that streams the graph data while performing computation. Nevertheless, it only loads the in-edges of vertices during computation, which disables selective scheduling and is inappropriate for certain algorithms that also require out-edges of vertices. [33] provides a general optimization for out-of-core graph processing, which removes unnecessary I/O by employing dynamic partitions whose layouts are dynamically adjustable. Although it can avoid the loading of useless data and eliminate random disk accesses, it has to write back the active edges. FlashGraph [41] supports selective access the edges requested by the applications in a random manner. Graphene [20] proposes clever I/O management and scheduling to ease the programming and achieve high IO performance. However, they both rely on expensive SSD arrays that deliver high I/O bandwidth and only support semi-external processing, and will incur significant overheads when issuing frequent small random accesses to HDDs or one SSD. While for LOSC, it has better flexibility and works well for both SSD and HDD, since it skips processing inactive edges in each iteration whenever such skipping can bring performance benefit. MultiLogVC [25] adopts a multi-log update mechanism and an extended compressed sparse row (CSR) format to reduce the loading of inactive vertices and edges. GraphSD [36] simultaneously captures the state and dependency of graph data during computation, so as to significantly improve the disk I/O performance.

Edge-centric systems. X-Stream [26] advocates a novel edge-centric scatter-gather computing model. In the scatter phase, it streams the entire edge list and produces updates. In the gather phase, it propagates these updates to vertices. Although it leverages high disk bandwidth through sequential accessing, it writes a large amount of intermediate updates to disks and disables selective scheduling, which incurs great I/O and computation overhead. GridGraph [43] also uses an edge-centric computing model. Differently, it combines the scatter and gather phases into one "streaming-apply" phase and uses a 2-Level hierarchical partition to break graph into 1D-partitioned vertex chunks and 2D-partitioned edge blocks. It supports selective scheduling by skipping the edge blocks for which vertices in the corresponding chunks are not scheduled. NXgraph [7] proposes destination-sorted subshard structure to store a graph so as to further ensure locality of graph data access. Although these systems can skip the phase of subgraph construction. MOSAIC [23] fully exploits the heterogeneity of modern hardware such as NVMe devices and Xeon Phis, to scale up to one trillion edges using a single machine. However, this may limit its adaptability due to the relying on these high-end hardwares. CLIP [3] and Lumos [32] adopt an out-of-order execution model to make full use of the loaded blocks to avoid loading the corresponding graph portions in future iterations. Their cross-iteration value propagation method can also be used in our work to further speedup the convergence of algorithms and reduce disk I/O.

5.4. Graph processing systems with compressed storage

Several systems adopt compressed storage format for efficient storage and better performance. WebGraph [4] presents several compression techniques to compress web and social network graphs, by exploiting the properties of real-world web graphs. However, it mainly focuses on compressing the web graphs and is not used to improve the performance of general graph algorithms or graph processing systems [29]. Ligra+ [29] integrates compression techniques such delta compression into Ligra. In addition, it uses run-length encoded byte codes for vertex encoding. Unlike LOSC that compresses the graphs to save the disk space and further reduce I/O traffic, it uses the compression techniques to enable faster in-memory parallel graph processing using less memory footprints. G-Store [17] utilizes the symmetry present in graph data by storing only the upper triangle (half) of graph data for undirected graphs, which is similar to the compression of undirected graph in LOSC. It also enables compression of ID, but implementing differently by removing the redundancy of the most-significant-bits (MSBs) of IDs of source and destination vertices within a partition. MOSAIC [23] implements the ID compression by mapping a global vertex ID in the original graph to a local vertex ID inside a tile. However, it has to maintain a per-tile meta index structure for the mapping. CIC-PIM [40] proposes a lightweight encoding with chunked index compression, to reduce the memory footprint and the runtime of graph algorithms. It divides index structures into chunks of appropriate size and compress the chunks with a lightweight fixed-length byte-aligned encoding.

6. Conclusion

In this paper, we discover a new performance bottleneck of out-of-core graph processing system other than the disk I/O problem, which is the inefficient subgraph construction caused by a large number of random memory accesses. In order to reduce the significant overheads of subgraph construction, we present a new out-of-core graph processing system called LOSC that supports vertex-centric computing model. LOSC proposes a locality-optimized subgraph construction scheme that improves the in-memory data access locality of subgraph construction phase. LOSC also adopts a compact edge storage format and a lightweight replication of vertices to reduce I/O traffic and improve computation efficiency. Moreover, a benefit-aware scheduling scheme is applied to skip processing the inactive edges, which further improves the I/O performance. Our evaluation results show that LOSC can be much faster than two representative graph processing out-of-core systems GraphChi and GridGraph, and other state-of-the-art out-of-core systems.

CRediT authorship contribution statement

Xianghao Xu: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Fang Wang:** Funding acquisition, Resources, Supervision. **Hong Jiang:** Validation, Writing – review & editing. **Yongli Cheng:** Investigation, Software. **Yu Hua:** Investigation, Validation. **Dan Feng:** Supervision. **Yongxuan Zhang:** Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

Acknowledgment

This work was supported by NSFC No. 61772216, No. 61832020, No. 61821003. This work is also supported by the Natural Science Foundation of Fujian Province under Grant No. 2020J01493, Zhejiang provincial Ten Thousand Talents Program (No. 2021R52007) and Center-initiated Research Project of Zhejiang Lab (No. 2021DA0AM01).

References

[1] http://www.valgrind.org/, 2019.
[2] http://www.graph500.org/, 2020.
[3] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, W. Zheng, Squeezing out all the value of loaded data: an out-of-core graph processing system with reduced disk I/O, in: USENIX ATC'17, 2017, pp. 125–137.
[4] P. Boldi, S. Vigna, The webgraph framework I: compression techniques, in: WWW'04, 2004, pp. 595–602.
[5] J. Cheng, Q. Liu, Z. Li, W. Fan, J.C. Lui, C. He, Venus: vertex-centric streamlined graph computation on a single pc, in: ICDE'15, IEEE, 2015, pp. 1131–1142.
[6] Y. Cheng, H. Jiang, F. Wang, Y. Hua, D. Feng, W. Guo, Y. Wu, Using high-bandwidth networks efficiently for fast graph computation, IEEE Trans. Parallel Distrib. Syst. 30 (5) (2018) 1170–1183.
[7] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, H. Yang, Nxgraph: an efficient graph processing system on a single machine, in: ICDE'16, IEEE, 2016, pp. 409–420.
[8] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: graph processing at Facebook-scale, Proc. VLDB Endow. 8 (12) (2015) 1804–1815.
[9] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, K. Pingali, Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics, in: PLDI'18, ACM, 2018, pp. 752–768.
[10] L. Dhulipala, G. Blelloch, J. Shun, Julienne: a framework for parallel graph algorithms using work-efficient bucketing, in: SPAA'17, ACM, 2017, pp. 293–304.
[11] L. Dhulipala, G.E. Blelloch, J. Shun, Theoretically efficient parallel graph algorithms can be fast and scalable, in: SPAA'18, 2018, pp. 393–404.
[12] N. Elyasi, C. Choi, A. Sivasubramaniam, Large-scale graph processing on emerging storage devices, in: FAST'19, 2019, pp. 309–316.
[13] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, C. Guestrin, Powergraph: distributed graph-parallel computation on natural graphs, in: OSDI'12, 2012, pp. 17–30.
[14] J.E. Gonzalez, R.S. Xin, A. Dave, D. Crankshaw, M.J. Franklin, I. Stoica, Graphx: graph processing in a distributed dataflow framework, in: OSDI'14, 2014, pp. 599–613.
[15] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, H. Yu, Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc, in: KDD'13, 2013, pp. 77–85.
[16] S. Ko, W.-S. Han, Turbograph++ a scalable and fast graph analytics system, in: SIGMOD'18, 2018, pp. 395–410.
[17] P. Kumar, H.H. Huang, G-store: high-performance graph store for trillion-edge processing, in: SC'16, IEEE, 2016, pp. 830–841.
[18] A. Kyrola, G. Blelloch, C. Guestrin, Graphchi: large-scale graph computation on just a pc, in: OSDI'12, 2012, pp. 31–46.
[19] H. Lin, X. Zhu, B. Yu, X. Tang, W. Xue, W. Chen, L. Zhang, T. Hoefler, X. Ma, X. Liu, et al., Shentu: processing multi-trillion edge graphs on millions of cores in seconds, in: SC'18, IEEE, 2018, pp. 706–716.
[20] H. Liu, H.H. Huang, Graphene: fine-grained io management for graph computing, in: FAST'17, 2017, pp. 285–300.
[21] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: a framework for machine learning and data mining in the cloud, in: PVLDB, 2012, pp. 716–727.
[22] L. Ma, Z. Yang, H. Chen, J. Xue, Y. Dai, Garaph: efficient GPU-accelerated graph processing on a single machine with balanced replication, in: USENIX ATC'17, 2017, pp. 195–207.
[23] S. Maass, C. Min, S. Kashyap, W. Kang, M. Kumar, T. Kim, Mosaic: processing a trillion-edge graph on a single machine, in: EuroSys'17, ACM, 2017, pp. 527–543.
[24] G. Malewicz, M.H. Austern, A.J. Bik, J.C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: SIGMOD'10, ACM, 2010, pp. 135–146.
[25] K.K. Matam, H. Hashemi, M. Annavaram, Multilogvc: efficient out-of-core graph processing framework for flash storage, in: IPDPS'21, IEEE, 2021, pp. 245–255.
[26] A. Roy, I. Mihailovic, W. Zwaenepoel, X-stream: edge-centric graph processing using streaming partitions, in: SOSP'13, ACM, 2013, pp. 472–488.
[27] A. Roy, L. Bindschaedler, J. Malicevic, W. Zwaenepoel, Chaos: scale-out graph processing from secondary storage, in: SOSP'15, ACM, 2015, pp. 410–424.
[28] J. Shun, G.E. Blelloch, Ligra: a lightweight graph processing framework for shared memory, in: PPoPP'13, 2013, pp. 135–146.
[29] J. Shun, L. Dhulipala, G.E. Blelloch, Smaller and faster: parallel processing of compressed graphs with ligra+, in: DCC'15, IEEE, 2015, pp. 403–412.
[30] Y. Tian, A. Balmin, S.A. Corsten, S. Tatikonda, J. McPherson, From think like a vertex to think like a graph, Proc. VLDB Endow. 7 (3) (2013) 193–204.
[31] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[32] K. Vora, Lumos: dependency-driven disk-based graph processing, in: ATC'19, 2019, pp. 429–442.
[33] K. Vora, G. Xu, R. Gupta, Load the edges you need: a generic I/O optimization for disk-based graph processing, in: USENIX ATC'16, 2016, pp. 507–522.
[34] X. Xu, F. Wang, H. Jiang, Y. Cheng, Y. Hua, D. Feng, Y. Zhang, Losc: efficient out-of-core graph processing with locality-optimized subgraph construction, in: IWQoS'19, ACM, 2019, p. 34.
[35] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, Y. Zhang, A hybrid update strategy for I/O-efficient out-of-core graph processing, IEEE Trans. Parallel Distrib. Syst. 31 (8) (2020) 1767–1782.
[36] X. Xu, H. Jiang, F. Wang, Y. Cheng, P. Fang, Graphsd: a state and dependency aware out-of-core graph processing system, in: ICPP'22, ACM, 2022.
[37] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, L. Liu, Triplebit: a fast and compact system for large scale rdf data, in: PVLDB, 2013, pp. 517–528.
[38] K. Zhang, R. Chen, H. Chen, Numa-aware graph-structured analytics, ACM SIGPLAN Not. 50 (8) (2015) 183–193.
[39] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, S. Amarasinghe, Graphit: a high-performance graph dsl, Proc. ACM Program. Lang. 2 (OOPSLA) (2018) 121.

[40] Y. Zhang, H. Jiang, F. Wang, Y. Hua, D. Feng, Y. Cheng, Y. Hu, R. Xiao, Cic-pim: trading spare computing power for memory space in graph processing, J. Parallel Distrib. Comput. 147 (2021) 152–165.
[41] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C.E. Priebe, A.S. Szalay, Flashgraph: processing billion-node graphs on an array of commodity ssds, in: FAST'15, 2015, pp. 45–58.
[42] Z. Zhou, H. Hoffmann, Graphz: improving the performance of large-scale graph analytics on small-scale machines, in: ICDE'18, IEEE, 2018, pp. 1368–1371.
[43] X. Zhu, W. Han, W. Chen, Gridgraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning, in: USENIX ATC'15, 2015, pp. 375–386.
[44] X. Zhu, W. Chen, W. Zheng, X. Ma, Gemini: a computation-centric distributed graph processing system, in: OSDI'16, 2016, pp. 301–316.

Xianghao Xu received the PhD degree from Huazhong University of Science and Technology, Wuhan, China, 2021. He is currently an assistant professor in School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing, China. His current research interests include graph processing, computer architecture and big data analytics. He has several publications in major international conferences and journals, including IEEE TPDS, ICPP, IWQoS.

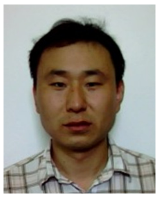
Fang Wang received her BE degree and Master degree in computer science in 1994, 1997, and Ph.D. degree in computer architecture in 2001 from Huazhong University of Science and Technology (HUST), China. She is a professor of computer science and engineering at HUST. Her interests include distribute file systems, parallel I/O storage systems and graph processing systems. She has more than 50 publications in major journals and conferences, including FGCS, ACM TACO, HiPC, ICDCS, HPDC, ICPP.

Hong Jiang received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, and the PhD degree from the Texas A&M University, College Station, in 1991. He is Wendell H. Nedderman Endowed Professor & Chair of Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems and parallel/distributed computing. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACMTOS, ACM TACO, JPDC, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP.

Yongli Cheng received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the FuZhou University, FuZhou, China, in 2010, and PhD degree from Huazhong University of Science and Technology, Wuhan, China, 2017. He is a teacher of College of Mathematics and Computer Science at FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, FGCS, ToN and FCS.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, China, in 2001 and 2005, respectively. He is currently a professor at the Huazhong University of Science and Technology, China. His research interests include computer architecture, cloud computing and network storage. He has more than 80 papers to his credit in major journals and international conferences including IEEE TC, IEEE TPDS, USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP and MASCOTS. He has been on the organizing and program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP, RTSS and IWQoS. He is a senior member of the IEEE, a member of ACM.

Dan Feng received the BE, ME, and PhD degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEETPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012.


Yongxuan Zhang received the B.E. degree in computer science and technology from the Nanchang Hangkong University, China, in 2005. He is currently a Ph.D. student majoring in computer science and technology in Huazhong University of Science and Technology, Wuhan, China. His current research interests include graph processing and parallel/distributed processing.