LOSC: Efficient Out-of-Core Graph Processing with Locality-optimized Subgraph Construction

Xianghao Xu
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
xianghao@hust.edu.cn

Fang Wang*†
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
wangfang@hust.edu.cn

Hong Jiang
Department of Computer Science & Engineering, University of Texas at Arlington
hong.jiang@uta.edu

Yongli Cheng‡
College of Mathematics and Computer Science, FuZhou University
chengyongli@fzu.edu.cn

Yu Hua
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
csyhua@hust.edu.cn

Dan Feng
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
dfeng@hust.edu.cn

Yongxuan Zhang
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
zyx@hust.edu.cn

ABSTRACT

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). On the other hand, the vertex-centric computing model is extensively used in graph processing thanks to its good applicability and expressiveness. Unfortunately, when implementing vertex-centric model for out-of-core graph processing, the large number of random memory accesses required to construct subgraphs lead to a serious performance bottleneck that substantially weakens cache access locality and thus leads to very long waiting time experienced by users for the computing results. In this paper, we propose an efficient out-of-core graph processing system, LOSC, to substantially reduce the overhead of subgraph construction without sacrificing the underlying vertex-centric computing model. 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 6.9x and 3.5x faster than GraphChi and GridGraph, two state-of-the-art out-of-core systems.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**; • **Hardware → External storage**;

KEYWORDS

graph computing, out-of-core, subgraph construction

ACM Reference Format:
Xianghao Xu, Fang Wang, Hong Jiang, Yongli Cheng, Yu Hua, Dan Feng, and Yongxuan Zhang. 2019. LOSC: Efficient Out-of-Core Graph Processing with Locality-optimized Subgraph Construction. In *IEEE/ACM International Symposium on Quality of Service (IWQoS '19), June 24–25, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3326285.3329069

1 INTRODUCTION

Graph is a powerful data structure to model and 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

*This author is the corresponding author.
†Also with Shenzhen Huazhong University of Science and Technology Research Institute.
‡Also with Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology.

IWQoS '19, June 24–25, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6778-3/19/06…$15.00
https://doi.org/10.1145/3326285.3329069

update function for a specific graph application without considering the underlying execution details. The quality of service of the graph processing frameworks depends on the processing performance of their graph processing jobs. To obtain a better performance, many systems adopt a large cluster to deploy their their large graph processing jobs, such as Pregel [17], PowerGraph [9], GraphX [10], etc. 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 [7] because of the significant amount of communication and coordination required among a large number of computing nodes when processing large graphs.

In recent years, several out-of-core graph processing systems such as GraphChi [13], X-Stream [19], GridGraph [26], etc. have been proposed to process large graphs on a single compute node, by efficiently using the secondary storage (e.g., hard disk, SSD). As we know, the secondary storage has much larger capacity and lower price than the DRAM. Therefore, the out-of-core systems can scale to very large graphs, serving as a promising alternative to distributed solutions. These 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.

Vertex-centric systems. 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. Most graph processing systems [9, 13, 17] are based on the vertex-centric model as it is intuitive for users to express many graph algorithms. GraphChi [13] is a widely-used out-of-core graph processing system that supports vertex-centric computation 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. By using a novel parallel sliding windows method to reduce random I/O accesses, GraphChi is able to process large-scale graphs in reasonable time. Based on GraphChi, other systems like BSPP [18], VENUS [6] and ODS [21] also adopt the vertex-centric model to further improve the performance of out-of-core graph processing systems.

Edge-centric systems. The edge-centric computing model explicitly factors computation over edges instead of vertices and takes the edge as the processing unit. X-Stream [19] is a typical one that uses an 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. Similar to X-Stream, GridGraph [26] uses a 2-Level hierarchical partition and a streaming-apply model to reduce the amount of data transfer, enable streamlined disk access, and maintain locality.

Compared with the vertex-centric model, the edge-centric model can leverage high disk bandwidth 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 [6]. Furthermore, for some algorithms such as community detection, it is difficult to implement them in an edge-centric model [6]. Therefore, in this paper, we mainly focus on the vertex-centric out-of-core graph processing systems for their better applicability and expressiveness.

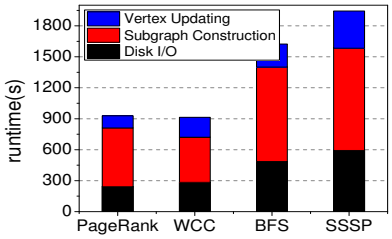


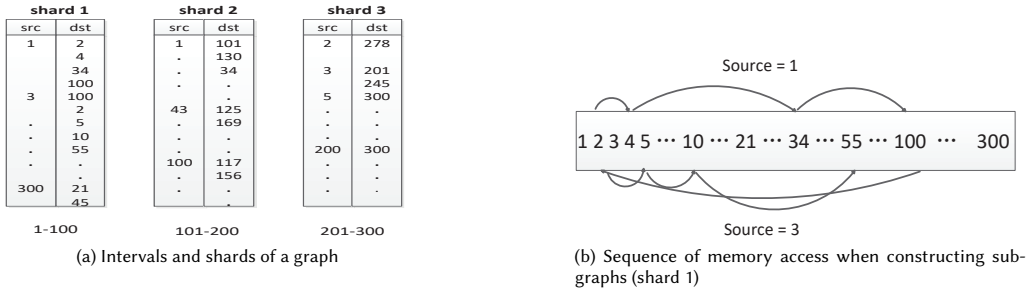
Figure 1: Breakdown time of several algorithms on Twitter for GraphChi

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%

Unfortunately, our experimental results show that the performance of vertex-centric out-of-core systems is usually limited by the inefficient subgraph construction that causes frequent random memory accesses. 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. Figure 1 shows the runtime breakdown of several algorithms on Twitter graph [12] executed by GraphChi. We observe the subgraph construction phase is responsible for at least 48% of the whole execution time, which leads to very long waiting time experienced by users for the computing results, significantly reducing the quality of service. Table 1 shows the cache misses of different execution phases when running BFS on Twitter. We can see that the last-level cache (LLC) miss 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.

In this paper, we present LOSC, an efficient out-of-core graph processing system that significantly reduces the overhead of subgraph construction without sacrificing the underlying vertex-centric computing model. The main contributions of LOSC are summarized as follows.



(a) Intervals and shards of a graph

(b) Sequence of memory access when constructing subgraphs (shard 1)

Figure 2: The example of constructing subgraphs

- 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 improve the memory access locality.
- LOSC implements a compact edge storage format by combining several graph compression methods to save storage size and reduce I/O traffic.
- LOSC adopts a lightweight replication of interval vertices (vertices within an interval) to improve computation efficiency by enabling full thread-level parallelism.
- We evaluate LOSC on several real-world graphs with different algorithms. Extensive evaluation results show that LOSC outperforms GraphChi and GridGraph by 6.9x and 3.5x on average respectively due to its locality-optimized subgraph construction and reduced disk I/Os.

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 an 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 a brief introduction to the vertex-centric computing model. 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 overhead of subgraph construction.

2.1 Vertex-centric Computing Model

The vertex-centric computing model establishes a "think like a vertex" idea [17] that can express a wide range of graph algorithms, for example, graph mining, data mining, machine learning and sparse linear algebra, as shown by many researchers [9, 13, 16, 17]. 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.

2.2 Out-of-Core Graph Processing

GraphChi [13] is an extensively-used out-of-core graph processing system that supports vertex-centric computation 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. GraphChi can process large graphs with a reasonable performance while using much fewer hardware resources than a distributed system. However, as shown in Figure 1, the subgraph construction phase significantly degrades the overall performance.

Other studies such as X-Stream [19] and GridGraph [26] utilize different computing models that can eliminate the overhead of subgraph construction. For example, X-Stream adopts an edge-centric computing model and avoids constructing vertex-centric subgraphs. However, this edge-centric model usually has poorer applicability and expressiveness than the vertex-centric model. And some algorithms such as community detection can not be implemented in an edge-centric model [6]. Therefore, most graph processing systems implement the vertex-centric computing model.

2.3 Subgraph Construction in Out-of-Core Graph Processing

For out-of-core 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 loaded into memory and assigned to corresponding vertices of the partition. This is the phase of subgraph construction. During this phase, each edge is added to the edge array of its source or destination vertex.

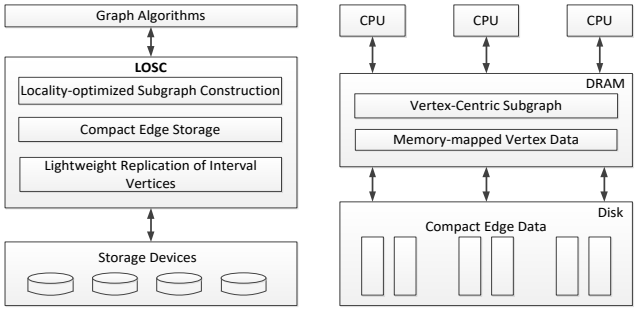


Figure 3: The LOSC Architecture

Figure 2 illustrates an example of constructing subgraphs in GraphChi. As shown in Figure 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 casue many random reads and writes in memory to add the incoming edges when constructing subgraphs as shown in Figure 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. Thus, the subgraph construction phase becomes a severe performance bottleneck when processing large real-world graphs for out-of-core graph processing systems. This motivates us to seek a design that minimizes random memory accesses to reduce the overhead of subgraph construction.

3 SYSTEM DESIGN

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

3.1 System Overview

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 [6, 24], we treat all vertices as mutable data and edges as read-only data. Furthermore, this optimization does not result in any loss of expressiveness as mutable data associated with edge $e = (u, v)$ can be stored in vertex u [6]. Therefore, only the vertex values are updated during the computation. For the large graphs whose vertex data is too large to be cached in memory, we simply mmap all the vertex data into memory as previous works [2, 15], to reduce random disk accesses.

LOSC is an efficient out-of-core graph processing system supporting vertex-centric computing model. Figure 3 presents the system architecture of LOSC. LOSC improves the performance by reducing the overhead of subgraph construction using a novel locality-optimized subgraph construction scheme. This scheme greatly improves the data access locality when constructing subgraphs. Furthermore, LOSC adopts a compact edge storage format to reduce I/O traffic and a lightweight replication of interval vertices to improve computation efficiency.

3.2 Graph Representation

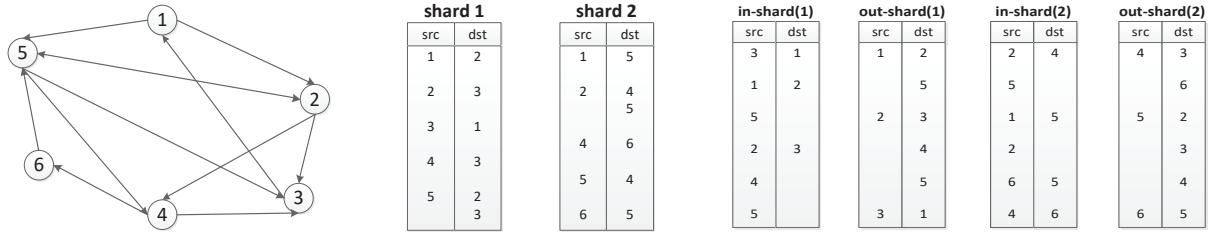
LOSC adopts the similar graph representation as GraphChi, which splits the vertices V of graph G into P disjoint intervals and edges E into P shards with source or destination vertices in corresponding vertex intervals. It differs from GraphChi in that it associates two edge shards for each interval: 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. We illustrate the contrast between the representation of GraphChi and LOSC of an example graph in Figure 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.

Although maintaining both in-edges and out-edges is not a novel design [14, 20, 22, 25], we use this to solve a totally different problem than previous works [14, 20, 25]. Specifically, previous works use this design to support different computing models and graph algorithms that need both in-edges and out-edges. For example, Ligra [20] stores both in-edges and out-edges to enable the adaptive push/pull update model. Moreover, these systems are designed for in-memory graph processing and do not need subgraph construction. On the other hand, LOSC uses this graph representation to solve the inefficient subgraph construction problems of out-of-core systems by fully utilizing the memory access locality. In addition, by storing in-edges and out-edges of an interval separately, it requires only two non-sequential disk reads to fully process an interval subgraph, rather than the P non-sequential reads required by GraphChi (P is the number of intervals), which substantially improves the disk accesses locality.

3.3 Locality-optimized Subgraph Construction

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

Algorithm 1 presents the procedure of high performance subgraph construction scheme. The procedure of subgraph construction is to add in-edges/out-edges to the in-edge array/out-edge



(a) Example graph (b) GraphChi shard representation (c) LOSC's representation

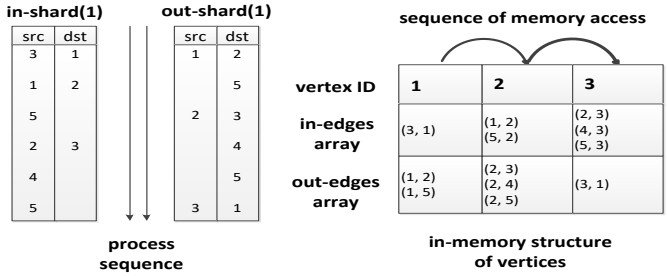
Figure 4: Illustration of graph representation

array of vertices. For each in-edge, LOSC first accesses the memory address of its destination vertex, and then adds the edge into 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 into the out-edge array of the vertex. Since 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 the destination/source vertices and data access locality is exploited as much as possible when constructing subgraphs.

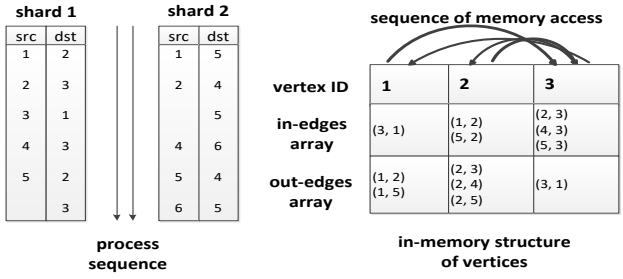
Algorithm 1 Locality-optimized Subgraph Construction

```
1:  for each interval p do
2:      /* Initialization */
3:      a ← interval(p).start
4:      b ← interval(p).end
5:      G ← InitializeMemoryForSubgraph(a, b)
6:      /* Load in-edges in in-chunk */
7:      Inegdes ← in − chunk(p).readfully()
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-chunk */
12:     Outegdes ← out − chunk(p).readfully()
13:     for each edge e in Outedges do
14:         G.vertex[e.source].addOutEdge(e.dest)
15:     end for
16:     return G
17: end for
```

Figure 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 Figure 4(a). As we see in Figure 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 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



(a) LOSC's locality-optimized subgraph construction



(b) GraphChi's subgraph construction

Figure 5: Comparison of Constructing Subgraphs

vertices in interval 1 as shown in Figure 5(b). When processing real-world graphs that have large numbers of vertices and edges and complex structures, the inefficiency of GraphChi's PSW subgraph construction will become an extremely serious problem for system performance.

Due to a great reduction of random memory accesses, 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 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 [5] 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 [23] to encode each vertex ID (including the delta values). 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 with very little extra preprocessing and decompression time, as shown in the evaluation results in Section 4.4.

3.5 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 Figure 4(c), they can not 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 values of the Latest-copy and the Old-copy and deletes the Latest-copy.

3.6 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. Figure 6 shows an example of processing on interval 1 of the graph in Figure 4(a).

Load edges. The loading phase of LOSC is very simple but I/O-efficient. As we see in Figure 6, 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). Unlike PSW of GraphChi, it maximizes the sequential access and requires only two non-sequential disk reads to process an interval subgraph, rather than the P non-sequential reads in GraphChi (P is the number of intervals).

Construct subgraph. When the edges are loaded into memory, LOSC starts the locality-optimized subgraph construction for the interval as described in Section 3.3. LOSC sequentially accesses the memory addresses of vertices 1, vertex 2, vertex 3, and adds their in-edges and out-edges. In fact, LOSC overlaps subgraph construction with edge loading as much as possible to make better use of parallelism.

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 Figure 6, 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 [13, 19] that write the updated edge attributes back to the disk for subsequent processing, synchronization of vertices significantly reduces disk IOs and improves system performance.

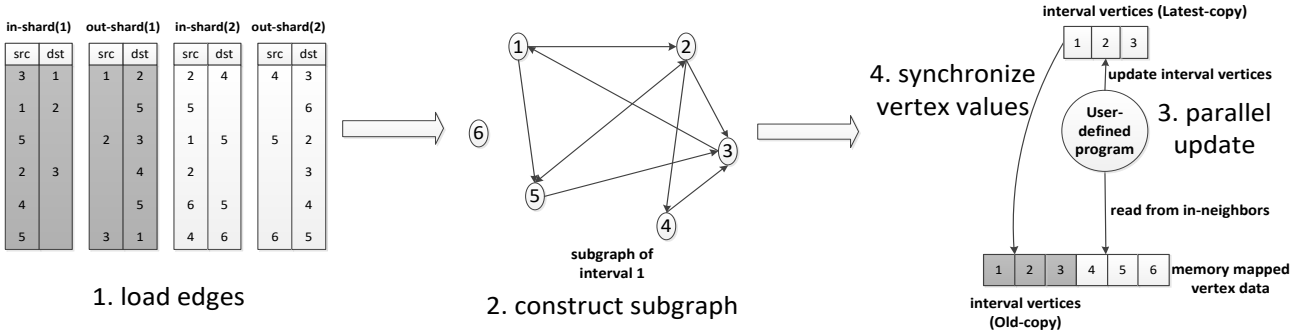


Figure 6: An Example of the LOSC workflow

Algorithm 2 Update Function (v): PageRank

```
Procedure PageRank
/* read values from in-neighbors */
for each edge e of v.inedges() do
    src ← e.source
    sum ← sum + src.value/src.outdegree
end for
/* update the value */
pagerank ← 0.15 + 0.85 × sum
v.value ← pagerank
End Procedure
```

In addition, LOSC supports selective scheduling to skip inactive vertices like GraphChi by representing the current schedule as a bit-array (we assume enough memory to store $|V|/8$ bytes for the schedule). This enables LOSC to focus on computation only where it is needed and improve the efficiency of computation.

4 EVALUATION

In this section, we first introduce our evaluation environment and the algorithms we used. Then, we evaluate LOSC by measuring the overall performance, the effects different system optimizations and the scalability.

4.1 Experiment Setup

All experiments are conducted on an 8-core commodity machine equipped with 12GB main memory and 600GB 7200RPM HDD, running Red Hat 4.8.5. In addition, a 128GB SATA2 SSD is installed for evaluating the scalability. Datasets used for the evaluation are all real-world graphs with power-law degree distribution, summarized in Table 2. 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 relationships between web pages, with larger diameters than social graphs. The in-memory graph LiveJournal is chosen to evaluate the performance of subgraph construction and the scalability of LOSC. The other three graphs Twitter2010, Friendster, UK2007 and UKUnion are larger than memory by 2.1x, 2.6x, 5.2x and 7.9x respectively.

Table 2: Datasets used in evaluation

Dataset	Vertices	Edges	Type
LiveJournal [3]	4.8 million	69 million	Social Network
Twitter2010 [12]	42 million	1.5 billion	Social Network
Friendster [5]	66 million	1.8 billion	Social Network
UK2007 [4]	106 million	3.7 billion	Web Graph
UKunion [4]	133 million	5.5 bilion	Web Graphs

We implement several different graph algorithms 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.

We compare LOSC with two state-of-art out-of-core systems that use the vertex-centric and edge-centric model respectively, GraphChi (introduced in Section 2.2) and GridGraph [26]. For all compared systems, we provide 8GB memory budget, 8 execution threads for the executions of all algorithms.

4.2 Overall Performance

We first report the execution time of the chosen algorithms on different graphs and systems in Table 3. We can see that LOSC significantly outperforms GraphChi and GridGraph. On average, LOSC outperforms GraphChi and GridGraph by 6.9x and 3.5x respectively.

The speedup over GraphChi mainly derives from the significant reduction in time spent on subgraph construction and in number of disk I/Os. PR and SpMV are computation-intensive algorithms in which subgraph construction dominates the execution time. For these algorithms, on average LOSC speeds up graph processing by 9.3x and 10.1x respectively, compared with GraphChi. BFS, WCC and SSSP are I/O-intensive algorithms and the disk I/O costs become

Table 3: 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	1.7	0.9	3.7	4.1	4.0
Twitter2010					
GraphChi	928.6	371.4	1624.3	913.7	1913.9
GridGraph	451.9	197.2	598.9	522.5	660.4
LOSC	126.5	57.6	230.1	176.3	249.2
Friendster					
GraphChi	2562.8	568.8	2294.5	2612.3	1802.4
GridGraph	1009.4	371.4	578.6	526.8	708.6
LOSC	230.2	70.8	473.4	481.3	376.2
UK2007					
GraphChi	2812.5	1160.7	7154.5	6862.8	7495.8
GridGraph	1242.2	511.2	6025.2	4783.8	7029.4
LOSC	265.1	121.7	1172.2	864.7	1171.4
Ukunion					
GraphChi	3376.6	1620.8	24062.3	5665.8	56650.9
GridGraph	1829.3	810.5	18929.2	13265.1	25554.2
LOSC	390.1	178.9	13022.5	3513.9	18171.4

the key factor on the system performance for these algorithms. Thanks to the significant reduction in disk I/Os due to the compact edge storage format, LOSC still outperforms GraphChi by 5.5x, 4.7x and 5.7x respectively on these three algorithms.

For GridGraph, although it processes the input graph without non-sequential disk reads and avoids constructing vertex-centric subgraphs since the computation is based on the edge lists, it has a worse performance than LOSC. This attributes to LOSC's compact edge storage format that leads to much fewer disk I/Os than GridGraph. Moreover, GridGraph disables full CPU parallelism and incurs a significant overhead to ensure consistency when updating vertices.

Furthermore, to analyze the performance gains obtained by using each optimization, we compare the system performance by respectively applying each optimization to LOSC. The optimizations include the locality-optimized subgraph construction scheme (LSCS), the compact edge storage format (CESF), the lightweight replication of interval vertices (LRIV). As the baseline, we disable all optimizations of LOSC. As shown in Figure 7, we can see that LSCS contributes to the most performance improvement while LRIV contributes to the least. This is because the vertex update phase has relatively less impact on the overall performance, compared with subgraph construction and disk I/Os.

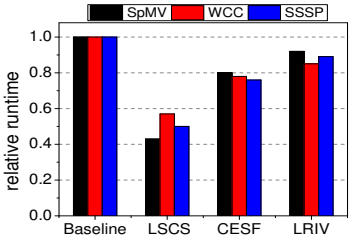


Figure 7: Varying performance by applying different optimizations to LOSC

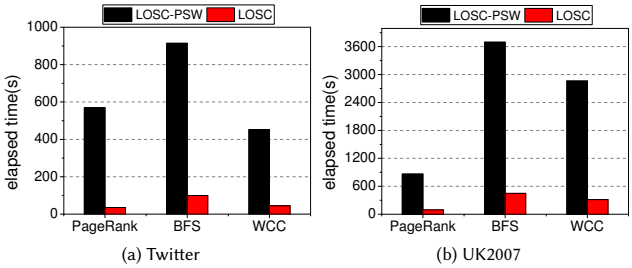


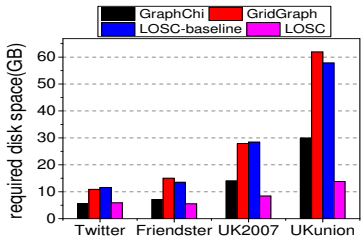
Figure 8: Time cost of subgraph construction

Table 4: Memory access and cache miss

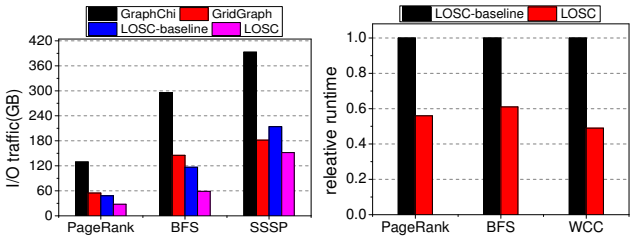
System		Read	Write
LOSC-PSW	mem. refs	416278519	212376955
	LLC misses	32053445	49059076
	LLC miss rate	7.7%	23.1%
LOSC	mem. refs	410852346	205426173
	LLC misses	1608991	6183666
	LLC miss rate	0.4%	3.0%

4.3 Effect of Locality-optimized Subgraph Construction

To evaluate the effect of the locality-optimized subgraph construction scheme, we compare LOSC with its baseline implementation (LOSC-PSW) that constructs subgraphs by using the PSW method [13] of GraphChi. Figure 8 shows the time cost of subgraph construction. We find that LOSC exhibits high efficiency of subgraph construction and achieves an average speedup of 10.3x compared with LOSC-PSW. This is mainly attributed to the locality-optimized subgraph construction scheme that makes better use of the locality of memory access during the subgraph construction phase. To further demonstrate that LOSC significantly improves the memory access locality, we measure the number of memory reads/writes and cache misses during the subgraph construction phase using Cachegrind [1], a tool capable of simulating 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



(a) Disk space consumption when storing different data sets



(b) I/O Traffic (c) Runtime

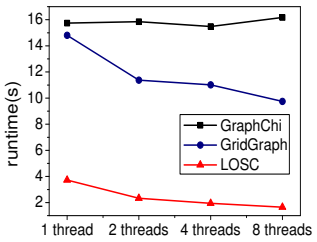
Figure 9: Evaluating the benefits of compact edge storage format

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 4. 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.

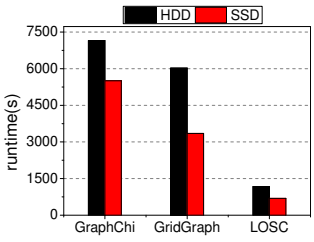
4.4 Effect of Compact Edge Storage Format

We evaluate the benefits of the compact edge storage format on storage space, I/O traffic and runtime of algorithms. Figure 10 shows the evaluation results. We compare LOSC with GraphChi, GridGraph and LOSC-baseline [1] in the evaluations. As shown in Figure 10(a), we observe that the storage of LOSC is much more efficient even though it stores two copies of each edge. Specially, the storage usages of GraphChi and GridGraph are respectively 1.4x and 3.1x higher than that of LOSC on average. Compared with LOSC-baseline, the compact edge storage format can save storage size by up to 76%. Figure 10(b) and Figure 10(c) shows the effects of the compact edge storage format on I/O traffic and runtime of algorithms. We can see that the compact edge storage format can significantly reduce the amount of I/O traffic, leading to better algorithm performance.

[1] implement LOSC without using the compact edge storage format.



(a) Scalability with threads



(b) Scalability with I/O

Figure 10: Evaluation of scalability

4.5 Scalability

We evaluate the scalability of LOSC by observing the improvement when more hardware resource is added. Figure 11(a) shows the execution time of PageRank on LiveJournal when 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 [26]. For LOSC, it makes full use of parallelism by using a lightweight replication of interval vertices during the computation as introduced in Section 3.5. 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 [13].

Figure 11(b) shows the performance improvement of BFS on UK when using different I/O devices. Compared with disk performance, GraphChi, GridGraph and LOSC achieves a speedup of 1.3x, 1.8x and 1.7x 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.

5 RELATED WORK

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 [11] 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 [6]. Bishard Parallel Processor [18] also separates in-edges and out-edges to reduce non-sequential IOs. However, it sorts all edges by source vertices and still suffers the inefficiency of constructing subgraphs. VENUS [6] 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. [21] provides a general optimization for out-of-core graph processing, which removes unnecessary I/O by employing dynamic partitions whose layouts are dynamically

adjustable. However, it incurs significant extra computation overheads.

Edge-centric systems. X-Stream [19] 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. Similar to X-Stream, GridGraph [26] 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 [8] 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. However, as traditional iterative graph computation is naturally expressed in a vertex-centric manner, users need to re-implement many algorithms in edge-centric API.

6 CONCLUSION

In this paper, we present an efficient out-of-core graph processing system called LOSC that aims to reduce the significant overhead of subgraph construction in vertex-centric out-of-core graph processing. LOSC proposes a locality-optimized subgraph construction scheme that improves the in-memory data access locality. LOSC adopts a compact edge storage format and a lightweight replication of vertices to reduce I/O traffic and improve computation efficiency. Our evaluation results show that LOSC can be much faster than GraphChi and GridGraph, two state-of-the-art out-of-core systems.

ACKNOWLEDGMENTS

This work is supported in part by NSFC No.61772216, National Key R&D Program of China NO.2018YFB10033005, National Defense Preliminary Research Project(31511010202), Hubei Province Technical Innovation Special Project (2017AAA129), Wuhan Application Basic Research Project(2017010201010103), Project of Shenzhen Technology Scheme JCYJ20170307172248636, Fundamental Research Funds for the Central Universities. This work is also supported by CERNET Innovation Project NGII20170120. This work is also supported by the Open Project Program of Wuhan National Laboratory for Optoelectronics NO.2018WNLOKF006 and NSFC No.61772212.

REFERENCES

[1] 2018. http://www.valgrind.org/. (2018).
[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *USENIX ATC'17*.
[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *KDD'06*. 44–54.
[4] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. In *ACM SIGIR Forum*. 33–38.
[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW'04*. 595–602.
[6] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. 2015. VENUS: Vertex-centric streamlined graph computation on a single PC. In *ICDE'15*. 1131–1142.
[7] Yongli Cheng, Fang Wang, Hong Jiang, Yu Hua, Dan Feng, and Xiuneng Wang. 2016. LCC-Graph: A high-performance graph-processing framework with low communication costs. In *IWQoS'16*. IEEE, 1–10.
[8] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. 2016. Nxgraph: An efficient graph processing system on a single machine. In *ICDE'16*. IEEE, 409–420.
[9] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12*. 17–30.
[10] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI'14*. 599–613.
[11] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD'13*. 77–85.
[12] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW'10*. 591–600.
[13] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: large-scale graph computation on just a PC. In *OSDI'12*. 31–46.
[14] Kisung Lee, Ling Liu, Karsten Schwan, Calton Pu, Qi Zhang, Yang Zhou, Emre Yigitoglu, and Pingpeng Yuan. 2015. Scaling iterative graph computations with GraphMap. In *SC'15*. 57.
[15] Zhiyuan Lin, Minsuk Kahng, Kaeser Md Sabrin, Duen Horng Polo Chau, Ho Lee, and U Kang. 2014. Mmap: Fast billion-scale graph computation on a pc via memory mapping. In *Big Data'14*. 159–164.
[16] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB* (2012), 716–727.
[17] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD'10*. 135–146.
[18] Kamran Najeebullah, Kifayat Ullah Khan, Waqas Nawaz, and Young-Koo Lee. 2014. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *IJMUE* (2014), 199–212.
[19] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-Stream: edge-centric graph processing using streaming partitions. In *SOSP'13*. 472–488.
[20] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceeding of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming(PPoPP)*, Vol. 48. 135–146.
[21] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC'16*. 507–522.
[22] Xianghao Xu, Fang Wang, Hong Jiang, Yongli Cheng, Dan Feng, and Yongxuan Zhang. 2018. HUS-Graph: I/O-Efficient Out-of-Core Graph Processing with Hybrid Update Strategy. In *ICPP'18*. ACM, 3.
[23] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: a fast and compact system for large scale RDF data. *PVLDB* (2013), 517–528.
[24] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *FAST'15*. 45–58.
[25] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI'16*. 301–316.
[26] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC'15*. 375–386.