GraphSD: A State and Dependency aware Out-of-Core Graph Processing System

Xianghao Xu
School of Computer Science and Engineering, Nanjing University of Science and Technology
Nanjing, China
xianghao@njust.edu.cn

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

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

Yongli Cheng*
College of Computer and Data Science, FuZhou University
Fuzhou, China
Zhejiang Lab
Hangzhou, China
chengyongli@fzu.edu.cn

Peng Fang
Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
Wuhan, China
fangpeng@hust.edu.cn

ABSTRACT

In recent years, system researchers have proposed many out-of-core graph processing systems to efficiently handle graphs that exceed the memory capacity of a single machine. Through disk-friendly graph data organizations and well-designed execution engines, existing out-of-core graph processing systems can maintain sequential locality on disk access and greatly reduce disk I/Os during processing. However, they have not fully explored the characteristics of graph data and algorithm execution to further reduce disk I/Os, leaving significant room for performance improvement. In this paper, we present a novel out-of-core graph processing system called GraphSD, which optimizes the I/O traffic by simultaneously capturing the state and dependency of graph data during computation. At the heart of GraphSD is a state- and dependency-aware update strategy that includes two adaptive update models, selective cross-iteration update (SCIU) and full cross-iteration update (FCIU). These two update models are dynamically triggered at runtime to enable active-vertex aware processing and cross-iteration vertex value computation, which avoid loading inactive edges and reduce disk I/Os in the future iterations. Moreover, an efficient sub-block based buffering scheme is proposed to further minimize I/O overheads. Our evaluation results show that GraphSD outperforms two state-of-the-art out-of-core graph processing systems HUS-Graph and Lumos by up to 2.7× and 3.9× respectively.

*Corresponding author

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545039

CCS CONCEPTS

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

KEYWORDS

graph processing, I/O-efficient, state and dependency of graph data

ACM Reference Format:
Xianghao Xu, Hong Jiang, Fang Wang, Yongli Cheng, and Peng Fang. 2022. GraphSD: A State and Dependency aware Out-of-Core Graph Processing System. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545039

1 INTRODUCTION

Graph processing plays an important role in many big data applications, such as social networks [16], fraud detection [12] and Internet of things [3]. Real-world graphs usually exhibit enormous sizes (i.e., billions of vertices and edges) and complex structures, which makes very challenging to handle them in a scalable way. To efficiently handle these large-scale graphs, researchers have developed a series of out-of-core (disk-based) graph processing systems [2, 6, 11, 17, 20, 21, 29] in recent years.

Out-of-core graph processing systems effectively use the external storage to process large graphs beyond the available memory of a single compute node. Due to the large capacity and low price of the external storage, out-of-core graph processing systems are more easy and cheaper to scale to process very large graphs, compared with distributed (memory) graph processing systems [13, 14, 27]. Moreover, they overcome the challenges of high communication costs [8] and load imbalance [18] faced by distributed systems. Before processing a graph, out-of-core graph processing systems first divide the vertex set of the graph into disjoint intervals and then partition the edge set into several smaller edge blocks, where each edge block contains the incoming or outgoing edges of the vertices in corresponding vertex interval. In this way, they can ensure each edge block fits in the available memory. During the processing of

the graph, the system successively loads and processes each vertex interval and the corresponding edge block from disk. Through a disk-friendly graph data organization format and well-designed execution engine, existing out-of-core graph processing systems can significantly reduce random disk accesses, even achieving competitive performance with distributed graph processing systems [2, 11, 29]. However, they have not fully explored the characteristics of graph data and algorithm execution for opportunities to further reduce disk I/Os.

On one hand, many graph algorithms are organized into several iterations and they usually access a small portion of graph data in each iteration. Moreover, as iterations progress, the number of active vertices continuously shrinks [15]. For example, Breadth-first Search (BFS) only visits neighbors of vertices in the current frontier in each iteration, and the number of unvisited vertices becomes very small at the end of the search. Unfortunately, existing out-of-core graph processing systems are usually designed for sequential accessing of graph data, which loads the whole graph into memory in each iteration. They are unaware of the states of vertices (active or not) during the algorithm executions, which incurs many unnecessary accesses of inactive edges.

On the other hand, most graph processing systems follow the processing semantics of the Bulk Synchronous Parallel (BSP) [19] model, to ensure the consistency and correctness of algorithm executions. In this model, the value of a vertex in the current iteration is computed based on the values of its neighbors in the previous iteration. Therefore, this model specifies the dependencies among vertices in graph computing [20]. Specifically, for an edge $e = (u, v)$, the value of v in iteration t is dependent on the value of u in iteration $t - 1$. In fact, the dependencies among vertices provide an opportunity to proactively compute the vertex values in the future iterations by making full use of the loaded edge blocks, so as to avoid the I/Os of the corresponding graph portions in the future iterations. For instance, after vertex u is updated in iteration t, we can use u's latest value to update the values of u's neighbors in iteration $t+1$ based on u's edges that are loaded into memory. Therefore, the I/Os of loading u's edges can be avoided when executing iteration $t + 1$. Unfortunately, most of the current out-of-core graph processing systems have not exploited this future dependency, which allows each edge/vertex to be processed only once in each iteration.

Although some recent works make efforts to explore above characteristics to improve I/O efficiency by performing active vertex aware processing [21, 22] or future-value computation [2, 20], none of them can simultaneously exploit the states and dependencies of graph data, leaving significant room to further reduce disk I/Os. In this paper, we develop GraphSD, an I/O efficient out-of-core graph processing system fully exploiting such characteristics of graph data. The main contributions of GraphSD are summarized below.

- GraphSD proposes a state- and dependency-aware update strategy that simultaneously captures the state and dependency of graph data during computation to significantly reduce I/O traffic. In this update strategy, GraphSD first selects a proper I/O access model including the on-demand I/O model and the full I/O model through a state-aware I/O scheduling strategy. Then, based on the selected I/O access model, GraphSD proposes two adaptive update models, selective cross-iteration update (SCIU) and full cross-iteration update (FCIU). Specifically, when selecting the on-demand I/O model, SCIU is triggered to selectively process the active edges, avoiding loading unnecessary data. When selecting the full I/O model, FCIU is triggered to load and process all edges. Both SCIU and FCIU further executes cross-iteration vertex value computation by efficiently exploiting the dependencies among vertices to reduce I/O traffics in the future iterations.
- To further reduce I/O overheads, GraphSD adopts an efficient sub-block based buffering scheme to judiciously buffer sub-blocks based on their priorities in the FCIU model. The priority of a sub-block is determined by the number of active edges in the sub-block.
- We conduct extensive experiments to demonstrate GraphSD's efficacy. Evaluation results show that GraphSD achieves a speedup up to 2.7× and 3.9× respectively over HUS-Graph [22] and Lumos [20], two state-of-the-art out-of-core graph processing systems, thanks to GaphSD's improved I/O performance.

The rest of the paper is organized as follows. Section 2 introduces the background and related works. Section 3 presents the system overview of GraphSD. Section 4 describes the detailed designs of the state- and dependency-aware update strategy. Section 5 demonstrates our experiment results and Section 6 concludes this paper.

2 BACKGROUND AND RELATED WORKS

Out-of-core graph processing systems are designed for handling large graphs beyond the available memory of a machine. GraphChi [11] is a pioneering out-of-core graph processing system using a vertex-centric computing model. It adopts an interval-shard structure to store a graph. The vertices are divided into disjoint intervals and a shard structure is created for each interval to store the incoming edges. Furthermore, GraphChi proposes a parallel sliding windows model to reduce random disk I/Os. Following GraphChi, a number of out-of-core graph processing systems have been proposed. Based on the adopted optimizations, these systems can be classified into three categories, as summarized in Table 1.

Eliminating Random Accesses. To fully avoid random accesses, X-Stream [17] uses an edge-centric approach. It streams the raw edge list during each iteration to update the destination vertices of each edge. GridGraph [29] further improves X-Stream by using a 2-Level hierarchical partition method and dual sliding window model, which avoids writing the intermediate results to disk. PathGraph [24] proposes a path-centric abstraction, which stores a graph as a collection of tree-based partitions and iteratively processes along the path in each partition, aiming to take full advantage of access locality. VENUS [6] proposes two I/O-efficient algorithms that enable the system to stream the graph data while performing vertex updating. NXgraph [7] adopts three adaptive update strategies according to different memory budgets so as to ensure the locality of graph data access.

Avoiding Inactive Data. Recent works exploit the running characteristics of graph algorithms to perform active vertex based

Table 1: Main optimizations adopted by existing out-of-core graph processing systems and GraphSD

System	Eliminating Random Accesses	Avoiding Inactive Data	Future-Value Computation
GraphChi [11]	✗	✗	✗
X-Stream [17]	✓	✗	✗
GridGraph [29]	✓	✗	✗
PathGraph [24]	✓	✗	✗
VENUS [6]	✓	✗	✗
NXgraph [7]	✓	✗	✗
GraphZ [26]	✓	✗	✗
DynamicShards [21]	✓	✓	✗
HUS-Graph [22]	✓	✓	✗
MultiLogVC [15]	✓	✓	✗
CLIP [2]	✓	✗	✓
Wonderland [25]	✓	✗	✓
Lumos [20]	✓	✗	✓
GraphSD	✓	✓	✓

processing. These systems further reduce I/O traffic by identifying the active vertices and avoiding loading the inactive data. Dynamic Shards [21] removes unnecessary I/Os by creating a dynamic shard for each interval in each iteration, which only contains the active edges in the current iteration. Therefore, the loading of inactive edges can be skipped. It also delays the updating of several vertices that cannot be performed due to missing edges. HUS-Graph [22] proposes a hybrid update strategy that adaptively selects I/O access and computing model based on the number of active vertices. MultiLogVC [15] uses a combination of the CSR graph format, and message logging to reduce read amplification caused by reading inactive vertices and edges.

Future-Value Computation. Moreover, several systems enable future-value computation to perform multi-iteration of computation in one round of graph loading, which significantly speeds up the convergence of graph algorithms and reduces disk I/Os. CLIP [2] uses a reentry method to make full use of the loaded edge blocks, enabling more efficient algorithms that require fewer total disk I/Os. Wonderland [25] allows users to extract effective abstractions from the original graph to capture certain graph properties. The abstractions can be used as a bridge to propagate information across different graph partitions. Lumos [20] adopts an out-of-order execution model to proactively compute vertex values in the future iterations while simultaneously ensuring the processing guarantees of BSP.

All these optimizations can dramatically improve I/O performance. However, as shown by Table 1, none of these systems can perform future-value computation while simultaneously capturing the states of vertices to avoid loading inactive graph data, leaving significant space to further reduce disk I/Os. Motivated by this observation, we propose a novel out-of-core graph processing system called GraphSD that can simultaneously support active-vertex aware processing and future-value computation by capturing the states and dependencies of graph data during computation.

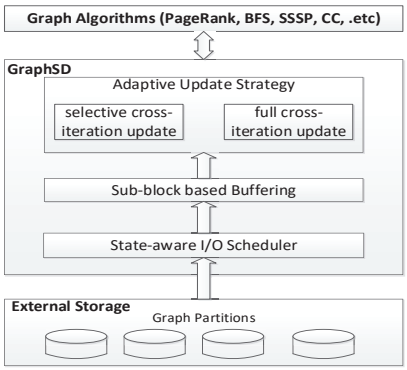
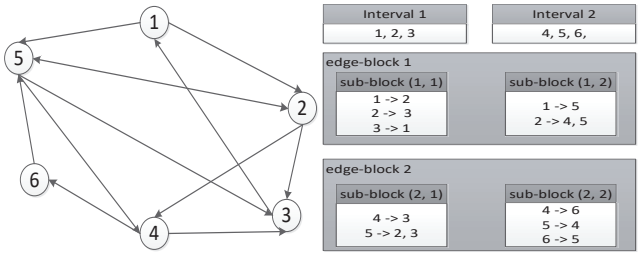


Figure 1: The GraphSD Architecture

3 SYSTEM OVERVIEW

3.1 The GraphSD Architecture

GraphSD aims to reduce the disk I/Os of out-of-core graph processing systems by fully exploiting the characteristics of graph data and algorithm execution. Figure 1 shows the architecture overview of GraphSD. To efficiently exploit the states of vertices and avoid the loading of inactive data, GraphSD adopts a state-aware I/O scheduler to schedule the loading of graph partitions. The state-aware I/O scheduler first identifies the number of active vertices in the current iteration, and then performs a benefits evaluation to select the proper I/O access model. Intuitively, when the number of the active vertices is small, the system tends to only load the active edges of a graph partition since it skips many unnecessary I/Os. When the number of the active vertices is large, the system tends to load the whole graph partition to avoid the expensive random disk accesses. Depending on different I/O access models, the system proposes two adaptive update models, selective cross-iteration update (SCIU) and full cross-iteration update (FCIU). When only loading the active edges, GraphSD uses SCIU model to selectively update the destination vertices of the active vertices. When loading the



(a) Example graph (b) Intervals and sub-blocks

Figure 2: fig/Graph representation of GraphSD

whole graph partition, GraphSD uses the FCIU model to update all vertices. Both SCIU and FCIU models can not only update the vertex values of the current iteration but also exploit the dependencies among vertices to enable cross-iteration vertex value computation, which reduces I/O traffics in the next iteration. In addition, several sub-blocks (graph partitions) are selected and buffered in DRAM to efficiently utilize the memory resource and further reduce the disk I/Os.

3.2 Preprocessing

We consider two principles when designing GraphSD's graph representation. First, to enable efficient access of active vertices and edges, the edges should be sorted by the source vertices so as to support fast query of active vertices and edges. Second, to support cross-iteration vertex value propagation for a vertex, the vertex should be fully updated in the current iteration. Therefore, the edges should also be organized by the destination vertices. In this way, a graph partition stores the incoming edges of a vertex interval and the vertices belong to an interval are fully updated after processing the interval.

Based on these two principles, GraphSD uses a 2-D partitioning method [7, 22, 29] to organize and partition the graph in the preprocessing phase. Specifically, the edges are first partitioned into P (P is the number of vertex intervals) edge blocks according to the source vertices. Then, each edge block is further partitioned into P sub-blocks based on the intervals of the destination vertices in the edge block. Intuitively, the 2-D partitioning method partitions the input graph into $P \times P$ grid. Figure 2 shows the data layout of an example graph in GraphSD's representation. The vertex set are partitioned into two intervals (1, 3) and (4, 6), and edges are organized into four sub-blocks. Each sub-block (i, j) stores edges whose source and destination vertices belong to interval i and interval j respectively. Therefore, interval i is called the source interval and interval j is called the destination interval for sub-block (i, j). For the processing of each sub-block, vertices in the source interval are read to calculate new values for vertices the destination interval. In addition, GraphSD creates an index structure index (i, j) to store the offset of each edge in sub-block (i, j), which enables fast accesses of active vertices and edges.

Table 2: Notations

Notation	Definition
G	the graph $G = (V, E)$
V	vertex set in G
E	edge set G
P	number of intervals
A	active vertex set
M	size of an edge structure
N	size of a vertex value
W	size of an edge weight value
B_{rr}	random read bandwidth
B_{rw}	random write bandwidth
B_{sr}	sequential read bandwidth
B_{sw}	sequential write bandwidth

4 GRAPHSD UPDATE STRATEGY

After the preprocessing phase, GraphSD can execute different graph algorithms programmed by the users based on the partitioned sub-blocks. In this section, we introduce the detailed designs of GraphSD's update strategy and system designs.

4.1 State-aware I/O Scheduling Strategy

To accommodate different I/O access characteristics of graph algorithms, GraphSD adopts a state-aware I/O scheduling strategy. Specifically, GraphSD dynamically selects the proper I/O access model by identifying the states of vertices (active or not) and capturing the number of active edges. If the number of active edges in the current iteration is small, GraphSD selectively load the active edges to avoid the unnecessary I/Os (the on-demand I/O model). Otherwise, GraphSD sequentially loads the whole sub-blocks to eliminate random disk I/Os (the full I/O model).

To accurately switch and select between these two I/O access models, GraphSD should evaluate the performance benefits of these two I/O access models and choose the best one. The performance benefit is estimated based on the I/O cost of each I/O access model, which can be computed by the total size of data accessed divided by the bandwidth of disk access. We analyze the I/O cost of each I/O access model as follows. For easy reference, we list the notations in Table 2. M, N, W represent the size of an edge structure, the size of a vertex value record and the size of an edge weight value respectively. For the disk bandwidth, B_{rr}, B_{rw}, B_{sr} and B_{sw} represent random read, random write, sequential read and sequential write bandwidth respectively.

When adopting the full I/O model, GraphSD sequentially loads all sub-blocks and vertex values into memory. For data writing, GraphSD only needs to write back vertex values, since only vertex values are updated during the processing. Therefore, the I/O cost C_s when executing an iteration can be constantly stated as:

$$C_s = \frac{|V| \times N + |E| \times (M + W)}{B_{sr}} + \frac{|V| \times N}{B_{sw}}$$

When adopting the on-demand I/O model, we suppose that the active vertex set is A. Note that reading the active edges is not totally random. There are some vertices with large degrees or vertices with contiguous IDs. The reading of their edge lists can be sequential. Therefore, we should calculate the sizes of edge lists that are sequentially and randomly read respectively, which are stated as S_{seq} and S_{ran}. Through one pass of A and the corresponding vertex degrees, S_{seq} and S_{ran} can be computed with the time complexity of $O(|A|)$. We skip the computing details here due to the page limitation. In addition, GraphSD also loads the vertex index structure in order to locate the active edges. Therefore, the I/O cost C_r when randomly loading the active edges in an iteration can be stated as:

$$C_r = \frac{S_{ran}}{B_{rr}} + \frac{S_{seq}}{B_{sr}} + \frac{2|V| \times N}{B_{sr}} + \frac{|V| \times N}{B_{sw}}$$

If $C_r \leq C_s$, the system selects the on-demand I/O model. Otherwise, it selects the full I/O model. The disk access bandwidths B_{rr}, B_{rw}, B_{sr} and B_{sw} can be measured by some measurement tools such as fio [17] conducting the experiments. Other parameters such as A, S_{seq} and S_{ran} can be directly collected and computed in the runtime. This performance benefit evaluation method provides an accurate performance prediction that enables efficient scheduling of edges, as shown in Section 5.4.

4.2 Update Models

According to different I/O access models, GraphSD proposes two adaptive update models, selective cross-iteration update (SCIU) and full cross-iteration update (FCIU), as shown in Algorithm 1. These two update models can not only update the vertex values of the current iteration but also exploit the dependencies among vertices to enable cross-iteration vertex value computation, so as to to reduce disk I/Os in the future iterations.

Algorithm 1 The GraphSD execution

```
1:  Out ← NewActiveVertexSet
2:  OutNI ← NewActiveVertexSetFortheNextIteration
3:  for each iteration iter do
4:      V_active ← Out
5:      Out ← OutNI
6:      OutNI ← Empty
7:      model ← UpdateModelSelection(V_active)
8:      if model = SCIU then
9:          /* Implement SCIU model, using Alg. 2*/
10:         SCIU(V_active, Out, OutNI)
11:     else
12:         /* Implement FCIU model, using Alg. 3*/
13:         FCIU(iter, Out, OutNI)
14:     end if
15: end for
```

Selective Cross-iteration Update. When selecting the on-demand I/O model, SCIU is triggered to selectively load the edges of the active vertices and update their neighbors. Algorithm 2 shows the execution procedure of SCIU in one iteration. When processing each interval *i*, SCIU successively processes from sub-block(*i*, *0*) to sub-block(*i*, *P-1*). For each sub-block (*i*, *j*), SCIU first locates active edges in the sub-block based on the corresponding vertex index and loads them into memory (Line 7). Then, it updates the destinations of these edges through a user-defined update function (Line 8 ~ 11). If a vertex's value is updated, it will be added to the new active vertex set *Out* and scheduled in the next iteration.

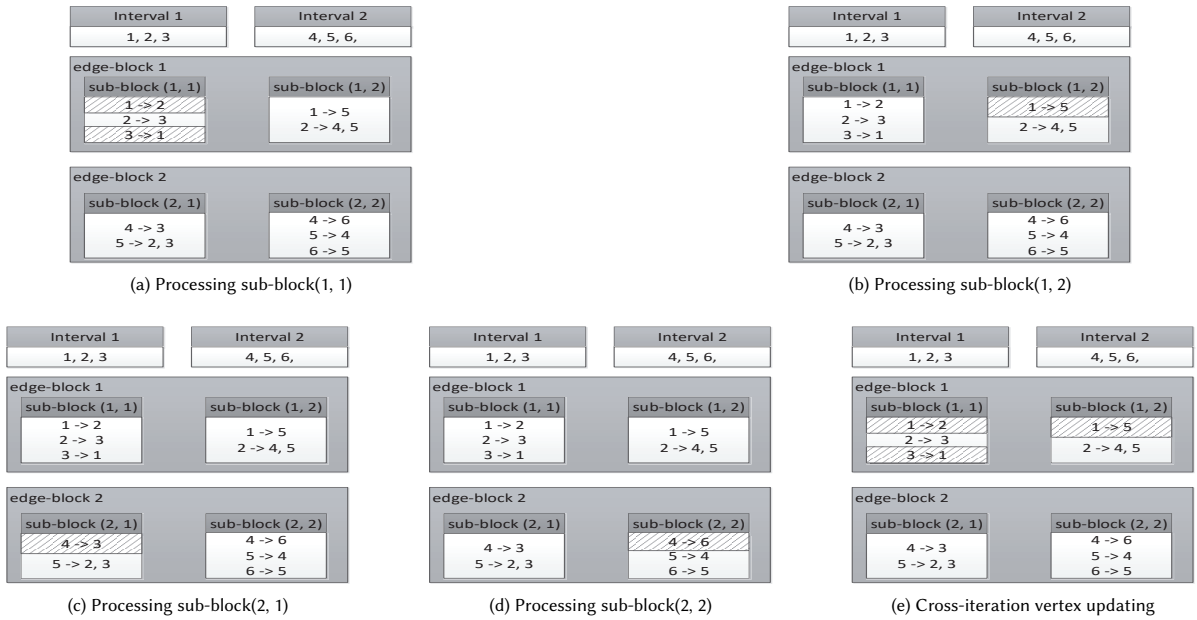
Algorithm 2 $SCIU(V_{active}, Out, OutNI)$

```
1:  for i from 0 to P − 1 do
2:      /* Identify active vertices in interval i*/
3:      V^i_active ← GetActiveVertices(V_active, i)
4:      for j from 0 to P − 1 do
5:          VertexIndex ← index(i, j)
6:          for each active vertex v in V^i_active do
7:              v.edges ← LoadEdges(v, VertexIndex, sub-block(i,
    j))
8:              for each edge e in v.edges do
9:                  neighbor ← e.dst
10:                 UserFunction(v, neighbor, Out)
11:             end for
12:         end for
13:     end for
14: end for
15: for each new activated vertex v_new in Out do
16:     if v_new ∈ V_active then
17:         Out.Remove(v_new)
18:         for each edge e′ in v_new.edges do
19:             neighbor ← e′.dst
20:             CrossIterUpdate(v, neighbor, OutNI)
21:         end for
22:     end if
23: end for
```

Different from existing works that support active-vertex aware processing [21, 22], SCIU further performs cross-iteration vertex value computation based on the loaded active edges (Line 15 ~ 23). Specifically, if the edges of the new activated vertices are already loaded into memory (i.e., the new activated vertices are also active in the current iteration), SCIU can directly update their neighbors' values in the next iteration in advance. Consequently, these vertices are removed from *Out* and their edges need not to be loaded in the next iteration.

Figure 3 illustrates one iteration of execution of SCIU with the example graph in Figure 2(a). Supposing the active vertices in the current iteration are vertex 1, 3 and 4. When processing vertex interval 1, SCIU successively accesses sub-block (1, 1) to sub-block (1, 2) to load the edges of vertex 1 and 3 and update their destinations. Then, SCIU moves to interval 2 to process sub-block (2, 1) and sub-block (2, 2). After the processing of this iteration is finished, vertex 1 and 3 are activated again and their edges are already loaded into memory. Therefore, SCIU performs cross-iteration vertex value computation for the neighbors of vertex 1 and 3, and the corresponding edges will not be scheduled in the next iteration.

Full Cross-iteration Update. When selecting the full I/O model, FCIU is triggered to load all edges and update the vertices. FCIU executes two consecutive iterations to enable cross-iteration vertex



(a) Processing sub-block(1, 1)

(b) Processing sub-block(1, 2)

(c) Processing sub-block(2, 1)

(d) Processing sub-block(2, 2)

(e) Cross-iteration vertex updating

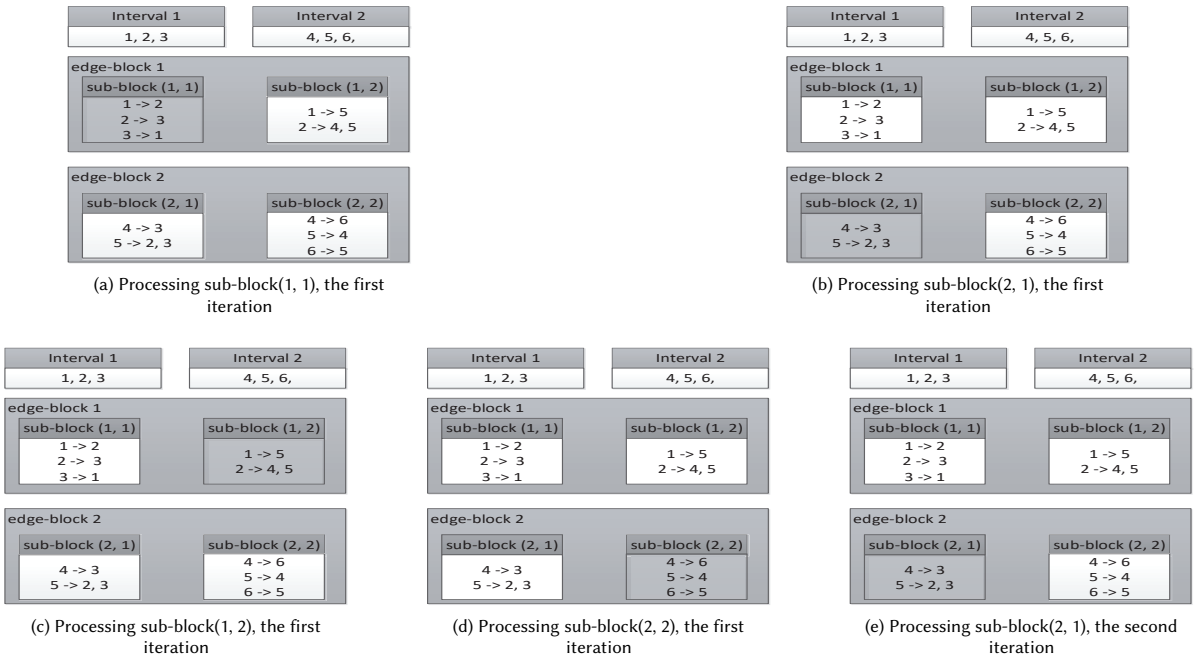
Figure 3: Execution procedure of the SCIU model

value computation. Algorithm 3 shows the execution procedure of FCIU. In the first iteration, FCIU loads the whole sub-blocks and iterates over all vertices to update the vertex values in the current iteration (Line 3 ~ 6). Then, it updates the values of several vertices in the next iteration based on the loaded edges (Line 7 ~ 11). The vertices chosen for cross-iteration values propagation should satisfy the following two conditions. First, they have been updated in the current iteration. Second, their edges should be loaded into memory. In this way, the vertices can exploit the dependencies among vertices to update the values of their neighbors in the next iteration with their own values in the current iteration, according to the BSP model. Unlike previous works [20] that create secondary partitions to store these edges, GraphSD can easily capture these edges with its graph representation. Specifically, for a sub-block(i, j), it can satisfy the dependencies among vertices and enable cross-iteration vertex value computation if $i < j$, since vertices in interval i are updated before vertices in interval j. In addition, sub-block (i, i) can also perform cross-iteration vertex value computation after all vertices in interval i are updated (Line 13 ~ 16). FCIU holds sub-block (i, i) in memory until all vertex updating for interval i have finished. In the second iteration (Line 18~ 26), FCIU only needs to load the sub-blocks that do not perform cross-iteration vertex value computation in the first iteration (i.e., $i > j$), and performs normal vertex updating (we called these sub-blocks as *secondary sub-blocks*).

Figure 4 shows the execution procedure of FCIU for two consecutive iterations. FCIU iterates over sub-block (1, 1) to sub-block (2, 1) when updating the vertices in interval 1, and sub-block (1, 2) to sub-block (2, 2) when updating the vertices in interval 2 in the first iteration. During the processing, sub-block (1, 1), sub-block (1, 2) and sub-block (2, 2) can enable cross-iteration vertex value computation. For instance, when processing edge (1, 5) in sub-block (1, 2), FCIU can use the value of vertex 1 in the first iteration (since

Algorithm 3 $FCIU(iter, Out, OutNI)$

```
1:  for j from 0 to P-1 do
2:      for i from 0 to P-1 do
3:          edges ← LoadEdges(sub-block(i, j))
4:          for each edge e in edges do
5:              UserFunction(e.src, e.dst, Out)
6:          end for
7:          if int(e.src) <int(e.dst) then
8:              for each edge e in edges do
9:                  CrossIterUpdate(e.src, e.dst, OutNI)
10:             end for
11:         end if
12:     end for
13:     /*sub-block(i, i) is held in memory*/
14:     for each edge e in sub-block(i, i) do
15:         CrossIterUpdate(e.src, e.dst, OutNI)
16:     end for
17: end for
18: iter = iter + 1
19: for i from 1 to P-1 do
20:     for j from 0 to i-1 do
21:         edges ← LoadEdges(sub-block(i, j))
22:         for each edge e in edges do
23:             UserFunction(e.src, e.dst, Out_n)
24:         end for
25:     end for
26: end for
```



(a) Processing sub-block(1, 1), the first iteration

(b) Processing sub-block(2, 1), the first iteration

(c) Processing sub-block(1, 2), the first iteration

(d) Processing sub-block(2, 2), the first iteration

(e) Processing sub-block(2, 1), the second iteration

Figure 4: Execution procedure of the FCIU model (two consecutive iterations)

vertex 1 has been updated before processing sub-block (1, 2)) to update the value of vertex 5 in the second iteration. Consequently, edge (1, 5) does not need to be loaded in the second iteration. And so do other edges in sub-block (1, 2). While for sub-block (2, 1), it has to be loaded in the second iteration since their source vertices are updated behind their destination vertices and cannot support cross-iteration vertex value computation in the first iteration.

In our programming model (Algorithms 1, 2 and 3), only function *UserFunction* and *CrossIterUpdate* are user-defined, while the others are provided by runtime. *UserFunction* is applied to the edges to conduct the user-defined update function, and generate new active vertices in the current iteration. Differently, *CrossIterUpdate* is executed for cross-iteration vertex values computation, which updates the values of vertices in next iteration in advance. Compared with other computing model such as block-centric model [23], our programming model focuses on disk-based graph processing and enables both selective and sequential access of edges by combining vertex-centric and edge-centric models. In addition, some systems that adopt asynchronous execution model like Giraph [9] can also support future-value computation by relaxing the BSP barrier. While our update strategy can not only enable future-value computation, but also guarantee synchronous processing semantics that ensures the consistency and correctness of algorithm executions.

4.3 Buffering of Secondary Sub-blocks

As mentioned above, the secondary sub-blocks are loaded into memory twice in the FCIU model. Moreover, their structures are unchanged during the computation. Therefore, it provides an opportunity to further reduce disk I/Os by buffering these secondary sub-blocks in memory. However, it may not be efficient to simply cache all secondary sub-blocks. First, there is not enough memory to cache all secondary sub-blocks for very large graphs. Second, after finishing the first iteration of processing, there may only exist very few active vertices, which means the secondary sub-blocks contain very few active edges. Hence, simply caching all the secondary sub-blocks can result in low utilization of memory.

To address these problems, we propose an efficient buffering scheme to judiciously buffer secondary sub-blocks based on their priorities. Specifically, the priority of a secondary sub-block is determined by the number of active edges in the secondary sub-block. The buffering scheme is implemented as follows. In the first iteration of the FCIU model, for each loaded secondary sub-block, if it is not cached and the buffer space is not full, it will be inserted into the buffer space. If the loaded sub-block has not been cached and there is no buffer space, the cached secondary sub-block with the lowest priority will be evicted. The priority of a secondary sub-block will be automatically updated after the processing of this secondary sub-block in the first iteration of the FCIU model. In this way, the memory resources can be efficiently utilized, which further reduces I/O traffic.

5 EVALUATION

In this section, we first present the evaluation environment including the hardware platform, graph dataset, graph algorithms and compared systems. Then, we evaluate GraphSD in terms of overall performance, I/O traffic and preprocessing time. Finally, we evaluate the effects of the design choices.

Table 3: Datasets used in evaluation

Dataset	Vertices	Edges	Type
Twitter2010 [10]	42 million	1.5 billion	Social network
SK2005 [5]	51 million	1.9 billion	Social network
UK2007 [4]	106 million	3.7 billion	Web graph
UKUnion [4]	133 million	5.5 billion	Web graph
Kron30 [1]	1 billion	32 billion	Synthetic graph

Table 4: Execution time (in seconds) of GraphSD

	PR	PR-D	CC	SSSP
Twitter2010	167.6	231.5	163.6	239.1
SK2005	170.1	140.9	766.3	1489.1
UK2007	370.1	337.6	892.1	1223.8
UKUnion	563.1	567.4	1962.7	4121.6
Kron30	3378.6	4370.1	7563.1	22181.8

5.1 Experiment Setup

Our experiments are conducted on a server with two 8-core 2.10 GHz Intel Xeon E5-2620 CPU, 32GB main memory and two 500GB HDDs, running Ubuntu 16.04 LTS. The dataset used for the experiments as summarized in Table 3.

We use four graph algorithms: PageRank (PR), PageRank-delta (PR-D), Connected Components (CC), and Single Source Shortest Path (SSSP). PR computes the rank value of each page by several rounds of iterations, in order to evaluate the impact of a Web page on a Web graph. PR-D is a variant of PageRank where vertices are activated in an iteration only if they have accumulated enough changes in their PR values. CC is an important graph mining algorithm that is usually implemented based on Label Propagation [28]. SSSP computes the shortest paths from a root vertex to other vertices, commonly used for navigation and traffic planing. These algorithms exhibit different I/O access and computation characteristics, which provides a comprehensive evaluation of GraphSD. For PR and PR-D, we run five iterations and twenty iterations on each graph respectively. For CC and SSSP, we run them until convergence. We use 16 execution threads for all algorithms. To better evaluate the out-of-core processing performance, we limit the memory budget to 5% of the graph data. In addition, for fair comparison and evaluation of the I/O optimizations, we disable the pagecache and use direct I/O in the experiments.

We compare GraphSD with two state-of-the-art out-of-core graph processing systems that support active vertex aware processing and future-value computation respectively, HUS-Graph [22] and Lumos [20]. HUS-Graph proposes a hybrid update strategy that captures the number of active vertices to skip unnecessary I/Os. Lumos proactively computes vertex values in future iterations to reduce I/O traffics across iterations. Therefore, they become a natural baseline to evaluate the efficiency of GraphSD's state- and dependency-aware update strategy.

5.2 Comparison to Other Systems

We first evaluate the overall performance of the algorithm executions. To intuitively compare the performance of different systems on different graphs, we show the normalized results in Figure 5. We also report the absolute execution time of GraphSD in Table 4. GraphSD finishes the executions more quickly than other two systems in all cases. On average, GraphSD outperforms HUS-Graph and Lumos by 1.7× and 2.7× respectively (up to 2.7× and 3.9×).

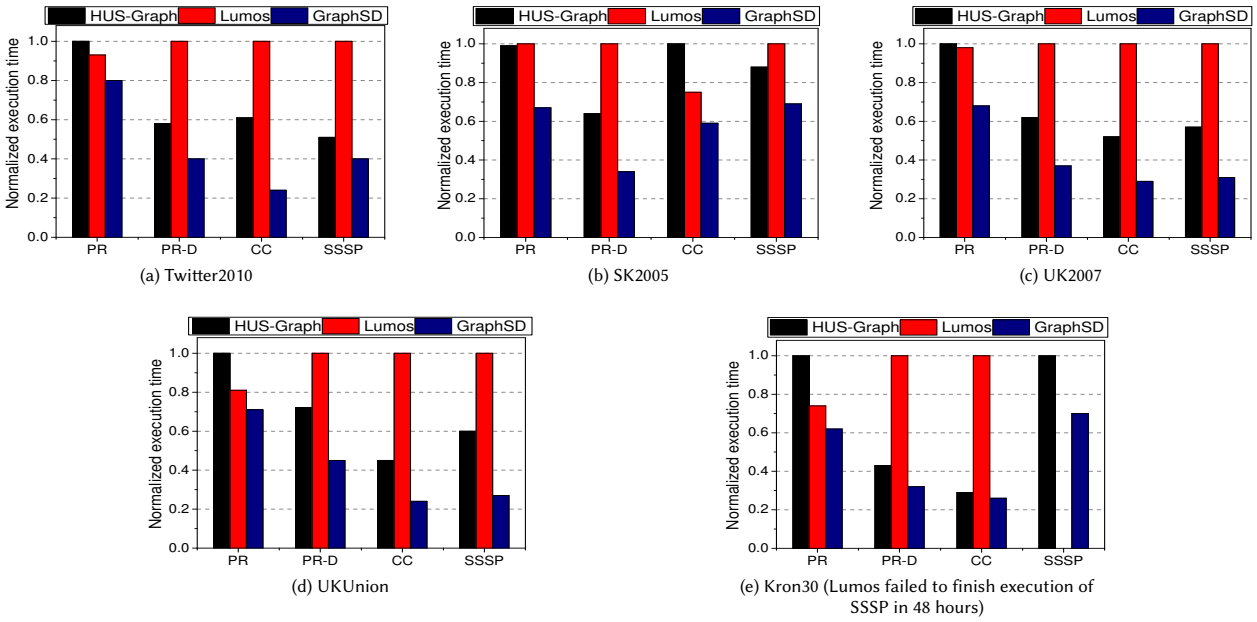
HUS-Graph adaptively schedules the loading of graph data to skip inactive I/Os. However, it cannot support cross-iteration computation. Lumos performs out-of-order execution model to reduce I/Os across future iterations but it loads many inactive vertices and edges. While for GraphSD, performance benefits come from the fact that it can not only perform active-vertex aware processing but also cross-iteration processing, leading to much less disk I/Os. Therefore, GraphSD is more useful for graph algorithms where the number of active vertices is small, and datasets whose structures can enable many cross-iteration propagations. Specifically, for the PR-D, CC and SSSP where the number of active vertices is small, GraphSD outperforms HUS-Graph by 1.8×, 1.5× and 1.7× respectively thanks to the reduced I/Os result from cross-iteration vertex-value computation. It outperforms Lumos by 2.8×, 3.1× and 2.7× respectively since it avoids the loading of inactive edges. For PR where all vertices are active and active vertex aware processing may not bring benefits, GraphSD still outperforms Lumos by 1.4× due to the efficient buffering of sub-blocks. The performance speedup over HUS-Graph is less for Kron30 because this synthetic graph exhibits different graph structure, which may produce fewer cross-iteration propagations than other datasets.

Figure 6 shows the breakdowns of the execution time on Twitter2010. As expected in out-of-core graph processing, the execution time is dominated by disk I/O (56%∼91% of execution time) despite of different computation features of these graph algorithms. From the results, we can observe that the disk I/O time of GraphSD (GS) is 73% and 49% of HUS-Graph (HG) and Lumos (LU) respectively. For vertex updating, HUS-Graph achieves the best performance because of fully utilization of parallelism and no need for cross-iteration values propagation.

Figure 7 compares the I/O traffics on Twitter2010 and UK2007. From the results, the volume of I/O traffic of GraphSD is 1.6× and 5.5× less than that of HUS-Graph and Lumos respectively. This is mainly attributed to GraphSD's state- and dependency-aware update strategy that not only reduces the loading of inactive data but also merges disk I/Os for several iterations. For PageRank where all vertices are active, HUS-Graph produces most I/O amounts since it cannot support cross-iteration computation to reduce I/Os in future iterations. For other three algorithms, Lumos produces most I/O amounts as it has to read many inactive edges.

5.3 Preprocessing Time

The comparisons of preprocessing time of different systems are depicted in Figure 8. The preprocessing procedure includes loading the raw graph data, partitioning and sorting the raw graph data, and writing the preprocessed graph data to disk. As shown in Figure 8, HUS-Graph takes the longest preprocessing time because it has to build two copies of edges and sort these edges. Specifically, the preprocessing time of HUS-Graph is longer than that of Lumos and



(a) Twitter2010 (b) SK2005 (c) UK2007

(d) UKUnion

(e) Kron30 (Lumos failed to finish execution of SSSP in 48 hours)

Figure 5: Comparison of overall execution time

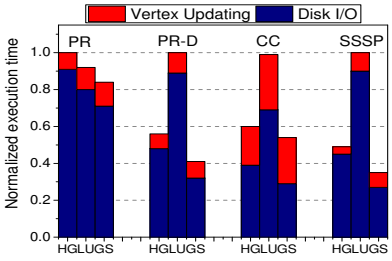
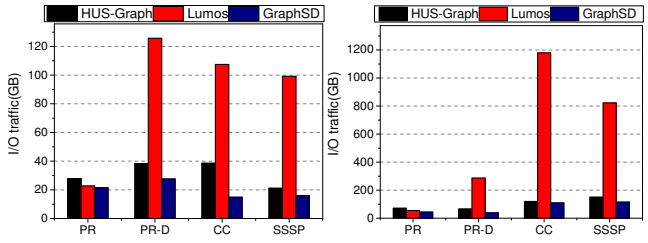


Figure 6: Runtime breakdown on Twitter2010



(a) Twitter2010 (b) UK2007

Figure 7: I/O traffic comparison

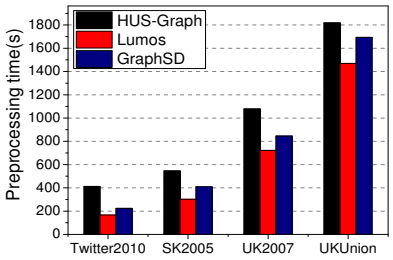
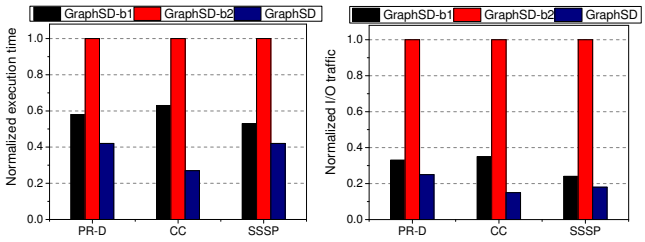


Figure 8: Preprocessing time comparison

GraphSD by 1.8× and 1.4× respectively. Lumos takes the shortest preprocessing time as it only maintains one copy of edges and does not need to sort the edges. For GraphSD, it maintains one copy of edges and needs to sort the edges to construct its representation. Although it takes more preprocessing time than Lumos, the overheads of the extra preprocessing can be offset by the performance improvements it brings. For example, by sorting the edges, GraphSD can enable selective loading of edges to skip many unnecessary I/Os. Furthermore, the preprocessed graph can be reused many times, which significantly amortizes the preprocessing overheads.

5.4 Effects of the Design Choices

We first evaluate the effect of GraphSD's update strategy. We compare GraphSD with two baseline implementations. The first baseline implementation (GraphSD-b1) disables cross-iteration vertex update, which only updates vertex values of the current iteration. The second baseline implementation (GraphSD-b2) disables selective vertex update, which loads all sub-blocks regardless of the number of active vertices. Figure 9 shows the comparison of different update strategies in terms of execution time and I/O traffic on Twitter2010. Thanks to capturing both states and dependencies of graph data during computation, GraphSD outperforms GraphSD-b1 and GraphSD-b2 by 1.7× and 2.8× respectively. In addition, GraphSD-b2 has a worse performance than GraphCP-b1. This indicates that active vertex aware processing can bring more improvements on



(a) Comparison of execution time (b) Comparison of I/O traffic

Figure 9: Effect of different update strategies

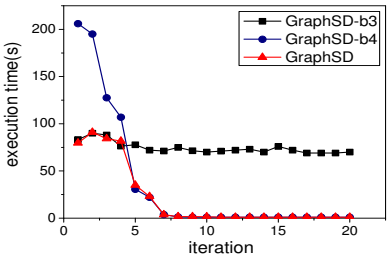


Figure 10: Effect of the state-aware I/O scheduling strategy

I/O performance than cross-iteration processing. As to I/O traffic, the I/O amount produced by GraphSD is 1.6× and 5.4× less than that of GraphSD-b1 and GraphSD-b2 shown in Figure 9(b), which further proves the I/O efficiency of GraphSD's update strategy.

Then, we evaluate the efficiency of the state-aware I/O scheduling strategy. To this end, we compare GraphSD with two baseline implementations that adopts the full I/O model (GraphSD-b3) and the on-demand I/O model (GraphSD-b4) for all iterations respectively. Figure 10 shows the comparison of execution time in each iteration when running CC on UKUnion. From the results, GraphSD is able to select the better I/O access model in all iterations, which verifies the accurate prediction of the performance benefits evaluation model of the state-aware I/O scheduling strategy.

We also evaluate the overheads of the state-aware I/O scheduling strategy since it will evaluate the performance benefits in each iteration and produce extra computation overheads. Specifically, we compare the computation overheads with the reduced I/O time overheads optimized by the state-aware I/O scheduling strategy on Twitter2010. As shown in Figure 11, we can see that the extra computation overheads are negligible. For example, the computation time for performance benefits evaluation of PR-D is only 3.4s, while the corresponding reduced I/O time is 158s.

Finally, we evaluate the effect of the buffering scheme of GraphSD. We run all algorithms on UKUnion, and compare the execution time with/without our buffering scheme. As shown in Figure 12, the buffering scheme can improve the performance by up to 21%, since the I/O overheads of the buffered sub-blocks are avoided in the FCIU model.

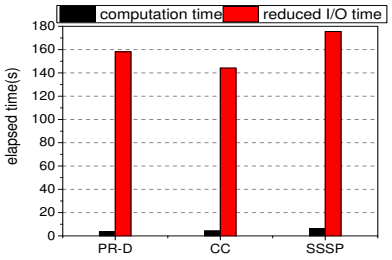


Figure 11: Overheads of the state-aware I/O scheduling strategy

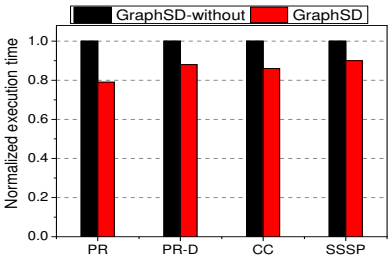


Figure 12: Effect of buffering scheme

6 CONCLUSION

In this paper, we present a novel out-of-core graph processing system called GraphSD that significantly improves disk I/O performance. GraphSD proposes a state- and dependency-aware update strategy that dynamically schedules the I/O accesses of graph data based on active vertices while simultaneously enabling cross-iteration vertex value computation. Moreover, GraphSD adopts an efficient sub-block based buffering scheme to further minimize I/O overheads. Our evaluation results show that GraphSD outperforms two state-of-the-art graph processing systems, HUS-Graph and Lumos. In future works, we will research how to exploit emerging storage devices such as Intel Optane PMM to further improve the I/O performance of GraphSD.

ACKNOWLEDGMENTS

This work was supported by NSFC 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] 2022. http://www.graph500.org/.
[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*. 125–137.
[3] Hisham Alasmary, Aminollah Khormali, Afsah Anwar, Jeman Park, Jinchun Choi, Ahmed Abusnaina, Amro Awad, Daehun Nyang, and Aziz Mohaisen. 2019. Analyzing and detecting emerging internet of things malware: A graph-based approach. *IEEE Internet of Things Journal* 6, 5 (2019), 8977–8988.
[4] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. In *ACM SIGIR Forum*, Vol. 42. ACM, 33–38.
[5] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW'04*. ACM, 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*. IEEE, 1131–1142.

[7] 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.

[8] Hoang-Vu Dang, Roshan Dathathri, Gurbinder Gill, Alex Brooks, Nikoli Dryden, Andrew Lenharth, Loc Hoang, Keshav Pingali, and Marc Snir. 2018. A lightweight communication runtime for distributed graph analytics. In *IPDPS'18*. IEEE, 980–989.

[9] Minyang Han and Khuzaima Daudjee. 2015. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment* 8, 9 (2015), 950–961.

[10] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW'10*. ACM, 591–600.

[11] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. In *OSDI'12*. USENIX, 31–46.

[12] Zhao Li, Haishuai Wang, Peng Zhang, Pengrui Hui, Jiaming Huang, Jian Liao, Ji Zhang, and Jiajun Bu. 2021. Live-Streaming Fraud Detection: A Heterogeneous Graph Neural Network Approach. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3670–3678.

[13] 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.

[14] 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*. ACM, 135–146.

[15] Kiran Kumar Matam, Hanieh Hashemi, and Murali Annavaram. 2021. Multi-LogVC: Efficient Out-of-Core Graph Processing Framework for Flash Storage. In *IPDPS'21*. IEEE, 245–255.

[16] Tuan-Anh Nguyen Pham, Xutao Li, Gao Cong, and Zhenjie Zhang. 2015. A general graph-based model for recommendation in event-based social networks. In *ICDE'15*. IEEE, 567–578.

[17] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP'13*. ACM, 472–488.

[18] Shuang Song, Meng Li, Xinnian Zheng, Michael LeBeane, Jee Ho Ryoo, Reena Panda, Andreas Gerstlauer, and Lizy K John. 2016. Proxy-guided load balancing of graph processing workloads on heterogeneous clusters. In *ICPP'16*. IEEE, 77–86.

[19] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[20] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *USENIX ATC'19*. 429–442.

[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. 2020. A Hybrid Update Strategy for I/O-Efficient Out-of-Core Graph Processing. *IEEE Transactions on Parallel and Distributed Systems* 31, 8 (2020), 1767–1782.

[23] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1981–1992.

[24] Pingpeng Yuan, Wenya Zhang, Changfeng Xie, Hai Jin, Ling Liu, and Kisung Lee. 2014. Fast iterative graph computation: A path centric approach. In *SC'14*. IEEE, 401–412.

[25] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. *ACM SIGPLAN Notices* 53, 2 (2018), 608–621.

[26] Zhixuan Zhou and Henry Hoffmann. 2018. Graphz: Improving the performance of large-scale graph analytics on small-scale machines. In *ICDE'18*. IEEE, 1368–1371.

[27] Xiaowei Zhu, Wenguang Chen, and Weimin Zheng. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI'16*. 301–316.

[28] Xiaojin Zhu and Zoubin Ghahramani. 2002. *Learning from labeled and unlabeled data with label propagation*. Technical Report. Citeseer.

[29] 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.