A Hybrid Update Strategy for I/O-Efficient Out-of-Core Graph Processing

Xianghao Xu, Fang Wang, Hong Jiang, *Fellow, IEEE*, Yongli Cheng, Dan Feng, *Member, IEEE*, and Yongxuan Zhang

Abstract—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, due to their high cost efficiency. To obtain a better performance, these systems adopt a full I/O model that scans all edges during the computation to avoid the inefficiency of random I/Os. Although this model ensures good I/O access locality, it leads to a large number of useless edges to be loaded when running graph algorithms that only access a small portion of edges in each iteration. An intuitive method to solve this I/O inefficiency problem is the on-demand I/O model that only accesses the active edges. However, this method only works well for the graph algorithms with very few active edges, since the I/O cost will grow rapidly as the number of active edges increases due to the increasing amount of random I/Os. In this article, we present HUS-Graph, an efficient out-of-core graph processing system to address the above I/O issues and achieve a good balance between I/O traffic and I/O access locality. HUS-Graph adopts a hybrid update strategy including two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP). It supports switching between ROP and COP adaptively, for the graph algorithms that have different computation and I/O features. For traversal-based algorithms, HUS-Graph also provides an immediate propagation-based vertex update scheme to accelerate the vertex state propagation and convergence speed. Furthermore, HUS-Graph adopts a locality-optimized dual-block representation to organize graph data and an I/O-based performance prediction method to enable the system to dynamically select the optimal update model between ROP and COP. To save the disk space and further reduce I/O traffic, HUS-Graph implements a space-efficient storage format by combining several graph compression methods. Extensive experimental results show that HUS-Graph outperforms two existing out-of-core systems GraphChi and GridGraph by 1.2x-52.8x.

Index Terms—Graph processing, out-of-core, I/O, hybrid update strategy

1 Introduction

Graph data has been widely used to model and solve many problems in application areas ranging from social networks to web graphs, from chemical compounds to biological structures, etc. 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. While a distributed system (e.g., Pregel [1], PowerGraph [2], GraphX [3] and Gemini [4]) is a natural choice for handling these large graphs, a recent trend initiated by GraphChi [5] advocates developing out-of-core support to process large graphs on a single commodity PC.

Out-of-core graph processing systems (e.g., GraphChi [5], X-Stream [6] and GridGraph [7]) utilize secondary storage to process very large graphs and achieve scalability without massive, costly hardware. Furthermore, they overcome the challenges faced by distributed systems, such as load imbalance and significant communication overhead. When processing an input graph, out-of-core systems divide the vertices into disjoint intervals and break the large edge list into smaller shards containing edges with source or destination vertices in corresponding vertex intervals. They process one vertex interval and its associated edge shard at a time. To obtain a better performance, these systems adopt a full I/O model to utilize the sequential bandwidth of disk and minimize the random I/Os. In this way, each edge shard is loaded entirely into memory, even though a large number of edges in the shard may not be needed.

Fig. 1 shows the percentage of active edges (the edges that have active sources vertices and are accessed in current iteration) per iteration for different iterative graph algorithms on LiveJournal graph [8]. For PageRank,[1] all edges are always active as all vertices compute their PR values in each iteration. For Breadth-first Search (BFS) and Weakly Connected Components (WCC), the number of active edges is small in most iterations. In this case, the pursuit of high I/O bandwidth overshadows the usefulness of data accesses. Repeatedly

- *X. Xu, F. Wang, D. Feng, and Y. Zhang are with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: {xianghao, wangfang, dfeng, zyx}@hust.edu.cn.*
- *H. Jiang is with the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019. E-mail: hong.jiang@uta.edu.*
- *Y. Cheng is with the College of Mathematics and Computer Science, Fuzhou University, Fuzhou, Fujian 350116, China, and also with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: chengyongli@fzu.edu.cn.*

Manuscript received 28 Dec. 2018; revised 1 Feb. 2020; accepted 5 Feb. 2020. Date of publication 11 Feb. 2020; date of current version 23 Mar. 2020. (Corresponding author: Fang Wang.) Recommended for acceptance by P. Sadayappan. Digital Object Identifier no. 10.1109/TPDS.2020.2973143

1. This is a standard implementation of PageRank. There is another implementation of PageRank (PageRank-Delata) where vertices are active in an iteration only if they have accumulated enough change in their PR value.

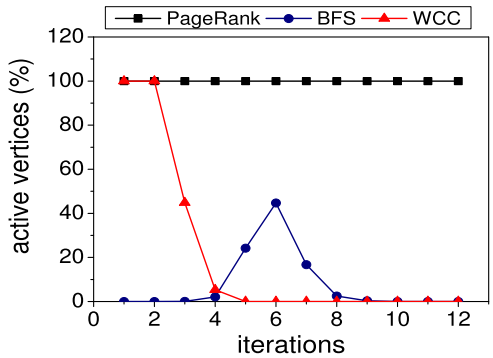


Fig. 1. The percentage of active edges per iteration.

loading the useless edges incurs significant I/O inefficiencies that degrade the overall performance. An intuitive method to solve this problem is the on-demand I/O model that only accesses the active edges. However, this method only works well for the graph algorithms with very few active edges. It incurs a large amount of small random disk accesses when the number of active edges is large due to the discontinuous distribution of active edges on disk. Dynamic Shards proposed in [9] eliminate this issue of random disk accesses by developing dynamic partitions whose layouts change during the runtime to compactly capture the set of active edges. It achieves this in light-weight manner by marking the active edges during compute phase and writing those marked edges back in separate partitions on disk during write-back phase. While [9] significantly eliminates loading of inactive edges, it requires writing back active edges. This dilemma motivates us to propose a new out-of-core system that achieves a good balance between I/O traffic and I/O access locality.

In this paper, we present HUS-Graph, an I/O-efficient out-of-core graph processing system with a hybrid update strategy. Our work is inspired by state-of-art shared-memory graph processing systems [10], [11] that utilize an adaptive push-pull model to handle graphs with different densities of active edges (percentage of active edges in all edges). When the density of active edges is sparse, these systems adopt a push-style model to only traverse the active edges and push updates to their destination vertices, which skips the processing of useless edges. When the density of active edges is dense, these systems adopt a pull-style model where each vertex collects data from its neighbors through its incoming edges and then updates its own value with the collected data, which eliminates atomic operations and enables full parallelism. By adaptively switching between these two update models, the system can handle different active edges densities with optimized performance. We extend this hybrid solution to the disk-based scenario. However, this is non-trivial work due to the following reasons. First, in both push and pull models, the vertices access their neighbors to update or obtain data. This can cause a large number of random and frequent disk accesses in out-of-core systems when the vertex values are too large to be cached in memory, which impacts the overall performance. Second, in order to gain optimal performance, it requires us to explore an effective performance prediction mechanism that guides the system to adaptively switch between push and pull models. HUS-Graph solves these problems by adopting a locality-optimized graph representation and an I/O-based performance prediction method. The main contributions of our work are summarized as follows.

- *Locality-optimized graph representation*. HUS-Graph proposes a dual-block representation to organize the graph data. It divides the vertices into several disjoint intervals and groups the outgoing and incoming edges of a vertex interval in out-blocks and in-blocks according to the source and destination vertices respectively. By restricting data access to each out-block or in-block and corresponding source and destination vertices, access locality can be ensured under the dual-block representation.
- *Hybrid update strategy*. HUS-Graph proposes two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP), to accommodate different computation and I/O loads. When running algorithms with sparse active edge sets, ROP only traverses the active edges and pushes updates to vertices, which avoids the loading of useless data. When running algorithms with dense active edge sets, COP streams all edges and updates vertices by pulling updates from neighbors, which overcomes the challenges of random disk accesses. For traversal-based algorithms, HUS-Graph also provides an immediate propagation-based vertex update scheme to accelerate the vertex states propagation and convergence speed. By utilizing an I/O-based performance prediction method, HUS-Graph can dynamically select the optimal update model based on the I/O load of the current iteration.
- *Space-efficient storage format*. To save the disk space and further reduce I/O traffic, HUS-Graph implements a space-efficient storage format by combining several graph compression methods. Compared with our previous work [12], the space-efficient storage format can reduce disk consumption by up to 71 percent, which further significantly reduces I/O traffic and improves system performance with very little extra preprocessing and decompression time.
- *Extensive experiments*. We evaluate HUS-Graph on several real-world graphs with different algorithms. Extensive evaluation results show that HUS-Graph outperforms GraphChi and GridGraph significantly, from 4.1x to 58.2x and from 1.2x to 29.3x respectively due to its efficient hybrid update strategy that brings a great improvement of I/O performance.

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

2 Background and Motivation

In this section, we first introduce the I/O issues of out-of-core graph processing systems. Then we present the prominent features of the adaptive push-pull model and its applying in shared-memory and distributed systems. This helps motivate us to propose a new out-of-core system that utilizes the hybrid solutions to solve the I/O issues of current out-of-core systems.

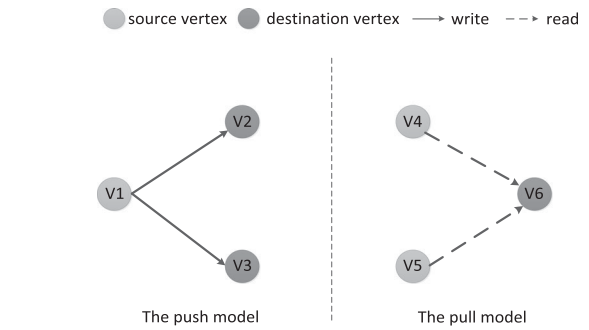


Fig. 2. The push and pull models.

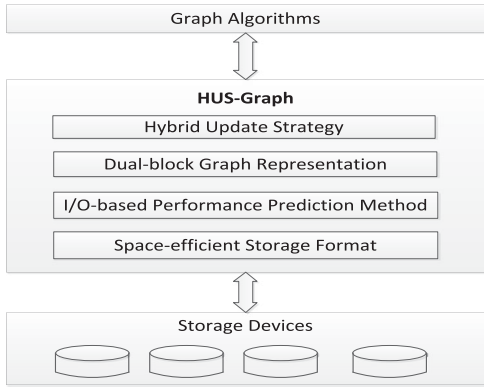


Fig. 3. The HUS-Graph architecture.

2.1 I/O Issues in Out-of-Core Systems

As many works [5], [6], [7] have shown, out-of-core graph processing systems can efficiently process billion-scale graphs on a PC. They process large graphs by efficiently using the disk drives. As the major performance bottleneck is disk I/O overhead [9], these systems are usually optimized for the sequential performance of disk drives and eliminate random I/Os by scanning the entire graph data in all iterations of graph algorithms. This full I/O model can be wasteful for algorithms that access only small portions of data during each iteration, such as BFS in Fig. 1. On the other hand, the on-demand I/O model that is based on the active edges can avoid loading the useless data. Unfortunately, it incurs a large amount of small random disk accesses due to the randomness of the active vertices. As we know, random access to disk drives delivers much less bandwidth than sequential access. Therefore, only accessing the useful data for out-of-core graph processing is an overkill when the number of active vertices is large. [9] solves this problem by employing dynamic partitions whose layouts are dynamically adjustable in each iteration. It achieves this in light-weight manner by marking the active edges during compute phase and writing those marked edges back in separate partitions on disk during write-back phase. Although [9] can eliminate random disk accesses, it requires writing back active edges.

2.2 Adaptive Push-Pull Model

The push and pull update models are extensively used in graph processing [4], [10], [11]. As shown in Fig. 2, in the push model, each vertex passes the updates to its neighbors through its outgoing edges. In the pull model, each vertex collects data from its neighbors through its incoming edges, and then updates its own value with the collected data. Obviously, push and pull are suitable for different scenarios, which is determined by the number of active edges. Specifically, sparse active edge set prefers the push model, since the system only traverses the outgoing edges of the active vertices where new updates are made. Contrarily, dense active edge set prefers the pull model, since it significantly reduces the contention in updating vertex states via locks or atomic operations. Inspired by this principle, several shared-memory systems [10], [11] and distributed systems [4], [13] adopt an adaptive update model during graph processing. For example, Ligra [11] proposes a lightweight graph processing framework that adaptively switches between the push and pull models according to the densities of active edge set in a shared-memory machine. Gemini [4] extends such adaptive design to distributed systems and proposes a sparse-dense signal-slot model.

However, current out-of-core systems either use the push model [6], [7] or the pull model [5], [14]. Furthermore, the push model for these systems is based on all vertices rather than the active vertices, since they put a higher priority on I/O access locality rather than I/O traffic. Actually, the adaptive push-pull model works well for out-of-core systems as well. When running algorithms with sparse active edge sets, the push model enables selective data access that only traverses the active edges, which fully avoids the loading of useless data. When running algorithms with dense active edge sets, the pull model sequentially accesses the edges of all vertices, which overcomes the challenges of random accesses and enables full parallelism. This motivates us to extend this hybrid solution to disk-based scenario to solve the I/O issues of current out-of-core systems.

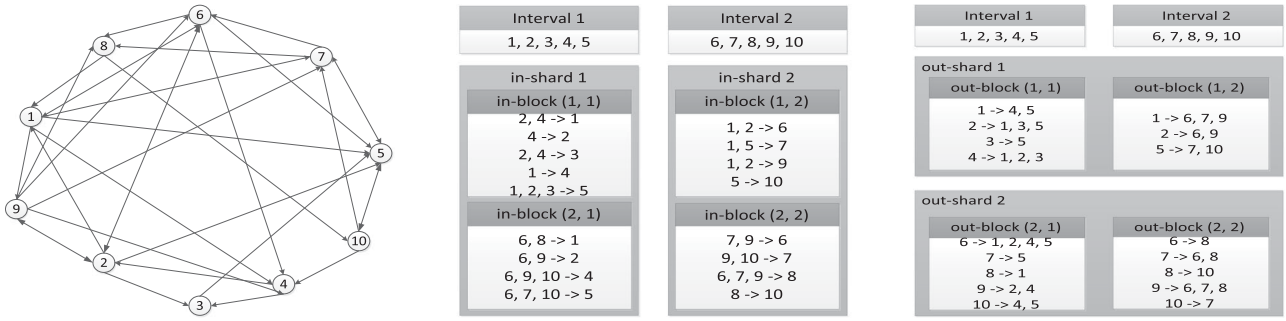
3 System Design

In this section, we first introduce the system overview of HUS-Graph. Then, we present the detail designs including the graph representation, space-efficient storage format, hybrid update strategy and the I/O-based performance prediction method.

3.1 System Overview

A graph $G = (V, E)$ is composed of its vertices V and edges E. 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 out-of-core graph processing systems, HUS-Graph focuses on maximizing the I/O performance as well. It improves the I/O performance by achieving a good balance between I/O traffic and I/O access locality. To achieve this, it adopts a hybrid update strategy including Row-oriented Push (ROP) and Column-oriented Pull (COP), inspired by shared-memory systems. Fig. 3 presents the system architecture of HUS-Graph. To efficiently support the hybrid update strategy, HUS-Graph adopts a dual-block representation to organize the graph data, which provides



(a) Example graph (b) In-blocks (c) Out-blocks

Fig. 4. Illustration of the dual-block representation.

fast loading of out-edges (in ROP) and scanning of in-edges (in COP). By restricting data access to each out-block or in-block and corresponding source and destination vertices, locality can be ensured under the dual-block representation. In addition, HUS-Graph implements an I/O-based performance prediction method that enables the system to dynamically select the optimal update model based on the I/O load of the current iteration.

Compared with current graph processing systems, HUS-Graph is different in the following two aspects.

Compared with current out-of-core systems, HUS-Graph handles graph algorithms that have different computation and I/O loads (number of active edges) with hybrid update strategy to achieve an optimal performance. While current out-of-core systems either adopt a push-style update [6], [7], [15] or a pull-style update [5], [14]. Second, HUS-Graph achieves a good balance between I/O traffic and I/O access locality, while current out-of-core systems improve the I/O access locality at the expense of larger amount of disk I/O, which degrades the overall performance especially when the number of active edges is small.

Compared with shared memory or distributed systems that use the adaptive push/pull model, HUS-Graph makes great efforts and proposes different techniques to overcome several new challenges when extending adaptive push/pull model to out-of-core systems. First, in both push and pull models, the vertices frequently access their neighbors to update the latter or obtain data for their own update. This can cause a large number of random disk accesses in out-of-core systems when the vertex values are too large to be cached in memory, notably reducing system performance. Second, in order to gain optimal performance, some important issues such as the proper switching time and effective performance prediction model must be explored, which is not easy in an out-of-core system whose performance depends highly on disk I/O overhead. To address these challenges, HUS-Graph adopts a locality-optimized graph representation and an I/O-based performance prediction method.

3.2 Graph Representation

The hybrid update strategy requires the system to store both out-edges and in-edges of vertices to support the adaptive processing. Unlike previous systems [4], [11] that use 1-dimensional partitioning to store the out-edges/in-edges, HUS-Graph adopts a 2-dimensional partitioning method and implements a dual-block representation to improve the locality of vertex access.

Like many out-of-core systems, HUS-Graph first splits the vertices V of graph G into P disjoint intervals. Each interval associates two edge shards, *in-shard* and *out-shard*, to respectively store the in-edges and out-edges of the vertices within the interval. Moreover, each in-shard is further partitioned into P *in-blocks* according to their source vertices. Similarly, each out-shard is partitioned into P *out-blocks* according to their destination vertices. Edges inside each in-block and out-block are respectively sorted by the destination and source vertices. In this way, both in-edges and out-edges are partitioned into $P \times P$ edge blocks. Each in-block (i, j) or out-block (i, j) contains edges that start from vertices in interval i and end in vertices in interval j. By selecting P such that each edge-block and the corresponding vertices can fit in memory, HUS-Graph can ensure good locality when processing each edge-block.

Fig. 4 shows the dual-block representation of an example graph. The vertices are divided into two equal intervals (1, 5) and (6, 10), the in-edges and out-edges are respectively partitioned into four in-blocks and four out-blocks according to the two intervals. For example, the out-edge (1, 6) is partitioned into out-blocks (1, 2) since vertex 1 belongs to interval 1 and vertex 6 belongs to interval 2.

In addition, we also maintain the index to the edges for each vertex. We refer to in-index (i, j) as the vertex index of in-block (i, j) and out-index (i, j) as the vertex index of out-block (i, j). This enables selective loading of the active edges in ROP and parallel update in COP.

3.3 Space-Efficient Storage Format

Since the dual-block representation stores both in-edges and out-edges plus the vertex index of each in-block and out-block, the storage size of of a graph is $2(|E| + P|V|)$ where P is the number of vertex intervals. This consumes more disk space than the existing graph representations such as Compressed Sparse Row (CSR) [5] whose storage size of a graph is $|E| + |V|$. To solve this problem, we implement a more space-efficient storage format by combining several graph compression methods, i.e., compression of undirected graph and ID compression.

Compression of Undirected Graph. For undirected graphs, HUS-Graph stores each edge twice, one for each of the two directions. Assuming that the example graph in Fig. 4a is undirected, its dual-block representation is shown in Fig. 5. 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, in-block (1, 1) and out-block (1, 1) are a duplicate

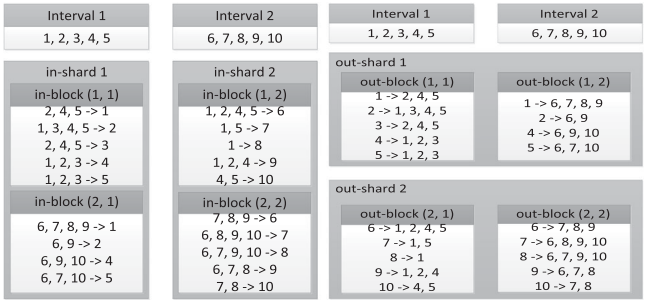


Fig. 5. Illustration of the dual-block representation when storing an undirected graph.

of each other as shown in Fig. 5, since the in-edges of an interval can also work as the out-edges. Similarly, in-block (1, 2) and out-block (2, 1), in-block (2, 1) and out-block (1, 2), in-block (2, 2) and out-block (2, 2) duplicate each other as well. Generally, for any two vertex intervals i and j in an undirected graph G with dual-block representation, if $i = j$, in-block (i, j) is a duplicate of out-block (i, j). Otherwise, in-block (i, j) is a duplicate of out-block ($P - i$, $P - j$), where $1 \leq i \leq P, 1 \leq j \leq P$.

To avoid the above redundant storage, HUS-Graph only maintains one copy of edges for undirected graphs, i.e., only storing in-edges or out-edges of an interval. For directed graphs, it still stores both in-edges and out-edges to enable different update models.

ID Compression. We further compress the ID of each vertex. To achieve the compression of vertex ID, we combine two methods as follows.

1)*Delta-based compression*. In fact, each edge block in the dual-block representation 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 [16] 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, HUS-Graph stores the vertex ID of the first neighbors and the delta values of the vertex IDs of remaining neighbors.

2)*Variable-length ID encoding*. 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. HUS-Graph adopts a variable-length integer [17] 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, our space-efficient storage format can reduce disk consumption by up to 71 percent, which further significantly 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.

3.4 Hybrid Update Strategy

Like many out-of-core systems, HUS-Graph processes the input graph one vertex interval at a time. Algorithm 1 shows the computation procedure of HUS-Graph in one iteration. For the processing of each vertex interval, HUS-Graph proposes two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP), to accommodate different I/O and computation loads of graph algorithms. The selection between ROP and COP is based on the number of active vertices with the I/O-based performance prediction method that is further discussed in Section 3.7. For both ROP and COP, we maintain two copies of vertex values for each interval i, source interval S_i and destination interval D_i. S_i stores the vertex values of previous iteration, serving as the source vertices. D_i stores the vertex values of current iteration, serving as the destination vertices.

Algorithm 1. Pseudo Code of HUS-Graph Execution

```
1: for each interval i do
2:    Out ← NewActiveVerticesSet
3:    /* Identify the active vertices in interval i*/
4:    V_active ← GetActiveVertices(i)
5:    /* Select the update strategy, using the I/O-based performance prediction method*/
6:    model ← UpdateModelSelection(V_active)
7:    if model = ROP then
8:       /* Implement ROP model, using Alg. 2*/
9:       RowOrientedPush(i, Out, V_active)
10:   else
11:      /* Implement COP model, using Alg. 3*/
12:      ColumnOrientedPull(i, Out)
13:   end if
14: end for
```

1)*Row-oriented Push. Execution*. ROP processes the input graph by pushing updates through the out-edges with the row-oriented process order. Algorithm 2 shows the procedure of ROP to execute a vertex interval i. ROP successively processes the out-blocks in a row from out-block (i, 1) to out-block (i, P) (Lines 2 $\sim$ 16). For the processing of each out-block (i , j) ($1 \leq j \leq P$), ROP first loads vertex values of S_i and D_j as well as the corresponding out-index. Then, it locates the out-edges of the active vertices in the out-block based on the corresponding out-index and loads them into memory (Line 8). As soon as the edges are loaded, ROP traverses the loaded edges and pushes the updates to their out-neighbors with a user-defined update function (Lines 9 $\sim$ 14). In this process, the vertex values in S_i are read-only while the vertex values in D_j are write-only. If there is new vertex activated, it is added to the new active vertex set and will be scheduled in the next iteration (Lines 11 $\sim$ 12). After the active edges in all out-blocks of interval i (in the ith row) are processed, ROP synchronizes the vertex values by replacing the values of source intervals with the values of destination intervals for

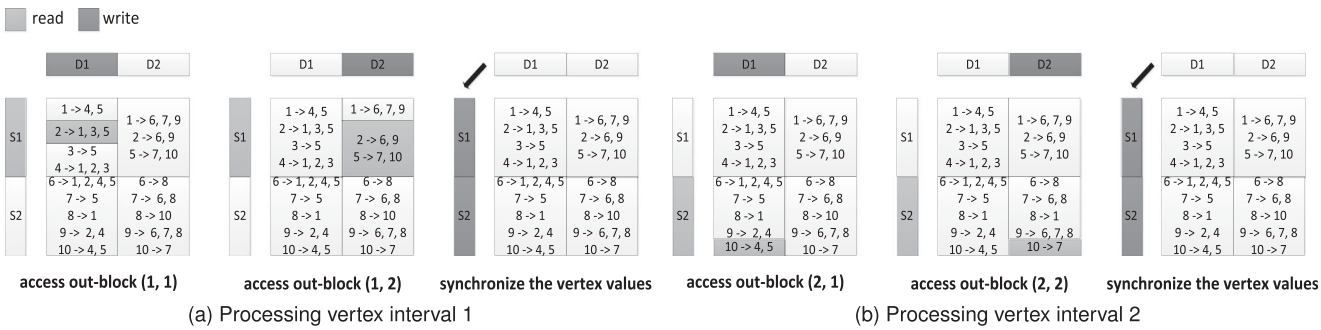


Fig. 6. Execution procedure of ROP.

subsequent computation (Lines 17 $\sim$ 19). Fig. 6 illustrates the execution procedure of ROP with the example graph in Fig. 4.

Algorithm 2. Pseudo Code of *RowOrientedPush* Function

```
1: LoadfromDisk(S_i)
2: /*parallel loops that overlap the processing of out-blocks of
   one row*/
3: for j from 0 to P-1 do
4:    LoadfromDisk(D_j)
5:    OutIndex ← out − index(i, j)
6:    for each active vertex v in V_active do
7:       out − degree ← OutIndex(v + 1) − OutIndex(v)
8:       edges ← LoadOutEdges(OutIndex(v), out − degree,
         out − block(i, j))
9:       for each edge e in edges do
10:         neighbor ← e.dst
11:         if UserUpdateFunction(v, neighbor) then
12:            Out.add(neighbor)
13:         end if
14:      end for
15:   end for
16: end for
17: for j from 0 to P-1 then
18:    S_j ← D_j
19: end for
```

Example. In the example of Fig. 6, ROP iterates over out-block (1, 1) to out-block (1, 2) when processing interval 1. Supposing that the active vertices of interval 1 in current iteration are vertex 2, 5 and 10. ROP successively loads and processes the out-edges of vertex 2 and 5 and updates their destinations when executing each out-block. Furthermore, processing of each out-block does not conflict with others due to the disjoint destination intervals. ROP can overlap the processing of out-blocks to fully exploit the multi-threading. When the processing of the first row of out-blocks is finished, ROP synchronizes the vertex values of all source intervals and destination intervals and moves to the next row to process out-block (2, 1) and out-block (2, 2).

I/O Cost Analysis. The I/O cost can be calculated by the total size of data accessed divided by the random/sequential throughput of disk access. Let M be the average size of the compressed adjacency list of a vertex and N be the size of a vertex value record. The active vertex set for each vertex interval i is A_i. In addition, T_{rr}, T_{rw}, T_{sr} and T_{sw} respectively represent random read, random write, sequential read and sequential write throughput (MB/s). For easy reference, we list the notations in Table 1.

For ROP, when processing vertex interval i, HUS-Graph only loads the out-edges of the active vertices, so the size of the active edges is $|A_i| \times M$ since the edge blocks consist of the compressed adjacency lists of vertices, as mentioned in Section 3.3. In addition, the vertex values of source interval (S_i) and destination intervals ($D_1 \sim D_P$) as well as the vertex indices are also loaded into memory. For the data writes, since the mutable attributes are stored in vertices, only vertex values of the destination intervals are written back to disk. For the ease of expression, we assume that the number of vertices in each interval is equal to $|V|/P$. Therefore, the I/O cost of processing vertex interval i, C_{rop}, can be expressed as

$$C_{rop} = \frac{|A_i| \times M}{T_{rr}} + \frac{(\frac{|V|}{P} + 2|V|) \times N}{T_{sr}} + \frac{|V| \times N}{T_{sw}}.$$

Summary. ROP puts a higher priority on I/O traffic rather than I/O access locality when processing a graph. It avoids the loading of useless data by only loading the active edges. Furthermore, with the row-oriented process order, ROP can overlap the processing of out-blocks to fully exploit the multi-threading when executing a vertex interval (Lines 2 $\sim$ 3), since the destination intervals of different out-blocks in a row are disjoint.

2) *Column-oriented Pull*. *Execution*. COP processes the input graph by pulling updates through the in-edges and updating the vertices with the column-oriented process order. Algorithm 3 shows the procedure of COP when executing a vertex interval i. COP successively processes the in-blocks in a column from in-block (1, i) to in-block (P, i) (Lines 2 $\sim$ 19). For the processing of each in-block (j, i) ($1 \leq j \leq P$), COP streams all in-edges of the in-block and loads the vertex

TABLE 1
Notations

Notation	Definition
G	the graph $G = (V, E)$
V	vertices in G
E	edges in G
P	number of intervals
A_i	active vertex set of interval i
M	average size of a compressed adjacency list
N	size of a vertex value
T_{rr}	random read throughput
T_{rw}	random write throughput
T_{sr}	sequential read throughput
T_{sw}	sequential write throughput

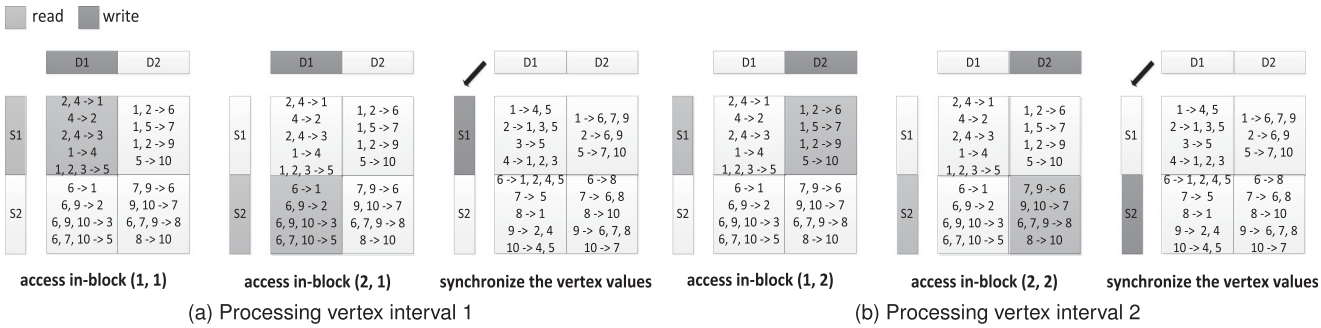


Fig. 7. Execution procedure of COP.

values of S_j and D_i. For each vertex in interval i , it locates its own in-edges and accesses its in-neighbors based on the corresponding in-index of the in-block (Line 9). Then, it collects data from the in-neighbors by reading S_j and updates its own value with a user-defined update function (Lines 9 $\sim$ 16). If there is new vertex activated, it is added to the new active vertex set. After all in-blocks of the interval i (in the ith column) are processed, COP replaces the S_i with D_i to synchronize the vertex values (Line 20). Fig. 7 illustrates the execution procedure of COP with the example graph in Fig. 4.

Algorithm 3. Pseudo Code of *ColumnOrientedPush* Function

```
1:  LoadfromDisk(D_i)
2:  for j from 0 to P-1 do
3:    LoadfromDisk(S_j)
4:    InIndex ← in − index(j, i)
5:    edges ← LoadInEdges(in − block(j, i))
6:    /*each vertex pulls updates in parallel*/
7:    for each vertex v in D_i do
8:      in − degree ← InIndex(v + 1) − InIndex(v)
9:      inedges ← edges.locate(InIndex(v), in − degree)
10:     for each edge e in inedges do
11:       neighbor ← e.src
12:       if IsActive(neighbor) then
13:         if UserUpdateFunction(neighbor, v) then
14:           Out.add(v)
15:         end if
16:       end if
17:     end for
18:   end for
19: end for
20: S_i ← D_i
```

Example. In the example of Fig. 7, COP iterates over in-block (1, 1) to in-block (2, 1) when processing interval 1. During the computation, each vertex in interval 1 can pull updates from their in-neighbors through the in-edges to update its own value in parallel. After the processing of the first column of in-blocks, COP replaces S_1 with D_1 and moves to process the next column.

I/O Cost Analysis. For COP, when processing vertex interval i, HUS-Graph loads all in-edges within an interval. Furthermore, the vertex values of source intervals ($S_1 \sim S_P$) and destination interval (D_i) as well as the vertex indices are also loaded into memory. For the data writes, only vertex values of the destination interval are written back to disk. Thus C_{cop} of vertex interval i is constant and can be expressed as

$$C_{cop} = \frac{\frac{|V|}{P} \times M + (\frac{|V|}{P} + 2|V|) \times N}{T_{sr}} + \frac{\frac{|V|}{P} \times N}{T_{sw}}.$$

Summary. Contrary to ROP, COP improves the I/O access locality at the expense of larger amount of disk I/O. Although it can not overlap the processing of the in-blocks in a column due to the write conflicts, the parallelism within each in-block can be achieved (Lines 6 $\sim$ 7) since the edges are sorted by the destination vertices. Note that ROP only accesses the out-edges stored in out-blocks while COP only accesses the in-edges stored in in-blocks. By restricting data access to each out-block or in-block and corresponding source and destination vertices, both ROP and COP ensure the locality when accessing vertices.

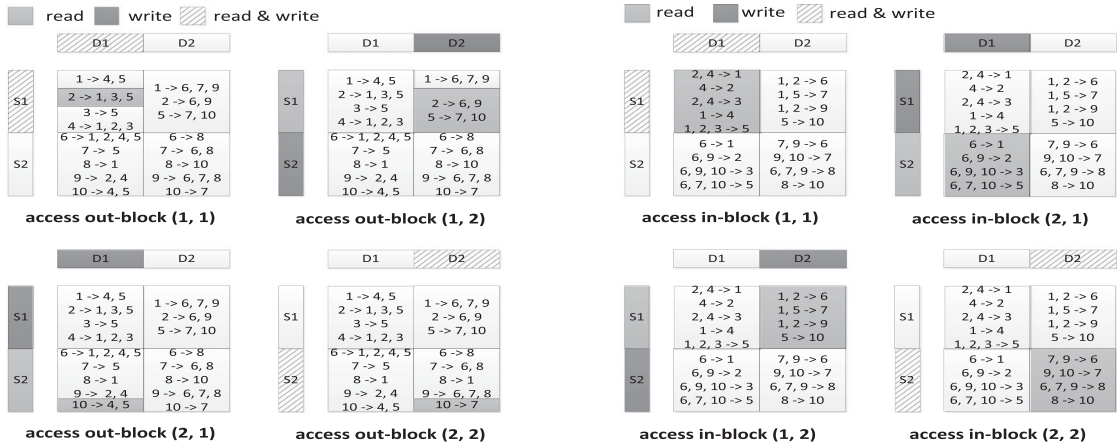
Note that our programming model is similar to Ligra's dense/sparse model [11]. In our programming model (Algorithms 1, 2 and 3), only function UserUpdateFunction is user-defined, while the others are provided by runtime. UserUpdateFunction is applied to the edges to conduct the user-defined programs, and identify if the destination vertex is activated. Algorithm 4 shows the implementation of UserUpdateFunction of Connected Components (CC).

Algorithm 4. *UserUpdateFunction (s, d)*: CC

```
1: Procedure CC
2: if s.label < d.label then
3:   d.label = s.label
4:   Return 1
5: else
6:   Return 0
7: end if
8: End Procedure
```

3.5 Comparison With GridGraph-Like Systems

Note that, some works like GridGraph [7] also use a 2-dimensional partitioning method and present a grid-like format to improve the I/O performance, which is similar to the dual-block representation of HUS-Graph. Moreover, GridGraph also supports column-oriented and row-oriented processing as well as selective scheduling. However, HUS-Graph adopts some new designs in both graph representation and graph processing execution model.



(a) Execution procedure of ROP (b) Execution procedure of COP

Fig. 8. Execution procedure of ROP and COP when using immediate propagation-based vertex update scheme.

In the graph representation, first, HUS-Graph stores both in-edges and out-edges to enable hybrid processing. Second, unlike GridGraph, HUS-Graph sorts the in-edges and out-edges by the destination and source vertices respectively, so that in-edges with the same destination vertex and out-edges with the same source vertex are stored contiguously. This is beneficial for the compression of edges and efficient parallel processing. Third, HUS-Graph creates a vertex index structure to enable the selective loading of edges.

In the graph processing execution model, HUS-Graph differs from GridGraph by adopting the following new designs based on the graph representation.

Hybrid Processing. Although GridGraph can also support column-oriented and row-oriented processing, it processes each edge in one direction, i.e., top-down update (push-style update) [11], shown in GridGraph's programming model. For HUS-Graph, it provides different update models for different edges and processing orders. For out-edges (ROP), HUS-Graph pushes updates to the destination vertices, which is a top-down update. For in-edges (COP), HUS-Graph adopts a pull-style model where each vertex pulls updates from the source vertices, which is a bottom-up update. By combining the two update models, HUS-Graph can enable selective loading of edges when using the push-style update and avoid the atomic operations when using the pull-style update, as described in Section 2.2.

Efficient Edge Storing and Parallel Processing. Due to the sorting of in-edges and out-edges, in-edges with the same destination vertex and out-edges with the same source vertex are stored contiguously. Therefore, HUS-Graph can enable edge compression and space-efficient storage format to reduce disk I/O. Moreover, fine-grained parallel processing can be achieved as shown in Section 3.4. While for GridGraph, it can not enable edges compression and fully utilize the parallelism without sorting edges [18].

Fine-Grained Selective Scheduling. By efficiently using the vertex index structure, HUS-Graph enables the selective loading of the active edges and avoids the loading of useless data. While for GridGraph, it can only skip processing the sub-blocks without any active-edges. This means that it loads and processes a sub-block even though there is only one active edge, which is a much more coarse-grained selective scheduling.

3.6 Immediate Propagation-Based Vertex Update

In both ROP and COP, the vertex update messages are propagated by synchronizing the vertex values of source and destination intervals. Concretely, HUS-Graph will replace the values of source intervals with the values of destination intervals until all out-blocks of one row are processed in ROP or until all in-blocks of one column are processed in COP. We call this an accumulation-based vertex update scheme. This scheme is particularly suitable for sparse matrix multiplication algorithms such as PageRank. These algorithms will not propagate update messages to vertices until all the update messages from their source vertices are accumulated. For traversal-based algorithms, they need not to accumulate all update messages of source vertices to update the states of vertices. For example, when running BFS, any active in-neighbor of a vertex can send an update message to it and make it active. Therefore, the accumulation-based vertex update scheme is suboptimal for these algorithms since the updated messages can not be applied instantly, which slows down the speed of vertex state propagation.

Based on this observation, HUS-Graph provides an immediate propagation-based vertex update scheme to accelerate the vertex state propagation and convergence speed of traversal-based algorithms. In this model, HUS-Graph applies the vertex updates immediately after receiving the update messages. Fig. 8 shows the executions of ROP and COP in the immediate propagation-based vertex update scheme with the example graph in Fig. 4. Here, we only maintain one copy of vertex values for each interval, which means that the source intervals work as the destination intervals at the same time. Therefore, the update messages are propagated to the destination vertices instantly with no need for vertex values synchronization, and the newly updated vertices can propagate update messages to their destination vertices in the subsequent processing in the same iteration. For example, when processing in-block(1, 1) in COP, as shown in Fig. 8b, vertex 1 that is newly updated by vertex 2 and vertex 4 is treated as a newly activated vertex from which vertex 4 and vertex 5 pull updates in the subsequent processing. While in the accumulation-based vertex update scheme as shown in Fig. 7a, vertex 4 and vertex 5 can only read vertex 1's value of the last iteration, which slows down the vertex state propagation and convergence speed of algorithms. The first source interval S1 and the first

destination interval D1 are identical physically and will be read and updated simultaneously, although they are logically separated.

3.7 I/O-Based Performance Prediction Method

The key to gain optimal performance is to select the fastest update model between ROP and COP. Moreover, unlike shared-memory systems [10] or distributed systems [19] whose performance depend on CPU performance or communication cost, the major performance bottleneck of out-of-core systems is I/O cost. HUS-Graph proposes a simple I/O-based performance prediction method that enables the system to dynamically select the optimal update model based on the I/O load of the current iteration.

The I/O-based Performance Prediction Method estimates and compares the I/O cost of ROP and COP when executing a vertex interval to guide the system to select the optimal update model. According to the I/O cost analysis in Section 3.4, the I/O traffics of vertex values and vertex indices for both ROP and COP are almost identical, except for the vertex values writes that have a slight impact on the I/O performance. Therefore, if $C_{rop} \leq C_{cop}$, we have

$$\frac{|A_i| \times M}{T_{rr}} \leq \frac{\frac{|V|}{P} \times M}{T_{sr}}.$$

It can be seen that

$$\frac{|A_i| \times P}{|V|} \leq \frac{T_{rr}}{T_{sr}},$$

The parameters A_i, $|V|$ and P can all be collected and computed in the runtime. Furthermore, the disk access throughput T_{rr} and T_{sr} can be measured by using several measurement tools such as fio [6] before we conduct the experiments. This provides an accurate performance estimate that enables the system to select the proper update model.

To reduce the extra computational overhead, HUS-Graph calculates and compares C_{rop} and C_{cop} just when the number of active vertices is less than a user-modified threshold α. In our experiments, α is empirically set to 5 percent of all vertices. If the number of active vertices is larger than α, HUS-Graph selects COP model regardless of the current active vertices.

4 Evaluation

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

4.1 Experiment Setup

Platform and Benchmarks. The hardware platform used in our experiments is a 16-core commodity machine equipped with 16 GB main memory and 500 GB 7200 RPM HDD, running Ubuntu 16.04 LTS. In addition, a 128 GB SATA2 SSD is installed to evaluate the scalability. The program is compiled with gcc version 5.4.0.

The datasets for our evaluation are all real-world graphs with power-law degree distributions, summarized in Table 2. LiveJournal, Twitter2010, Friendster and SK2005 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 scalability of HUS-Graph. The other five graphs are respectively 1.5x, 1.9x, 2.1x, 3.9x and 6.1x larger than available memory.

TABLE 2
Datasets Used in Evaluation

Dataset	Vertices	Edges	Type
LiveJournal [8]	4.8 million	69 million	Social Graphs
Twitter2010 [20]	42 million	1.5 billion	Social Graphs
Friendster [16]	66 million	1.8 billion	Social Graphs
SK2005 [16]	51 million	1.9 billion	Social Graphs
UK2007 [21]	106 million	3.7 billion	Web Graphs
UKunion [21]	133 million	5.5 bilion	Web Graphs

The benchmarks algorithms used in our evaluation include three traversal-based graph algorithms, Breadth-first search (BFS), Weak Connected Components (WCC), and Single Source Shortest Path (SSSP), and a representative sparse matrix multiplication algorithm known as PageRank. BFS, WCC and SSSP vary the number of active vertices in different iterations and can effectively evaluate the hybrid update strategy of HUS-Graph. We run these algorithms until convergence. For PageRank, all vertices are always active as each vertex receives messages from its neighbors to compute the new rank value in all iteration. We run five iterations of PageRank on each dataset. In addition, we use the accumulation-based vertex update scheme for PageRank and use the immediate propagation-based vertex update scheme for other three traversal-based graph algorithms.

Systems for Comparison. We compare HUS-Graph with three baseline out-of-core graph processing systems. One is the first version of HUS-Graph (we name it as HUS-Graph-v1) [12]. We compare HUS-Graph against HUS-Graph-v1 to evaluate the effects of the newly proposed optimizations including the space-efficient storage format and immediate propagation-based vertex update scheme. The other two systems for comparison are GraphChi [5] and GridGraph [7]. GraphChi is an extensively-used out-of-core graph processing system that supports vertex-centric scatter-gather computation model. It exploits a novel parallel sliding windows (PSW) method to minimize random disk accesses. GridGraph 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. For all compared systems, we provide 16 execution threads for the executions of all algorithms.

Evaluation Methodology. We first explore the effects of our system designs including hybrid update strategy, I/O-based performance prediction method, space-efficient storage format and immediate propagation-based vertex update scheme. Then, we compare HUS-Graph against other systems on runtime of algorithms and I/O traffic. Finally, we evaluate the scalability of HUS-Graph by observing the improvement when more hardware resource is added.

4.2 Effect of Hybrid Update Strategy

Fig. 9 shows the effect of the hybrid update strategy of HUS-Graph, by comparing the Hybrid model that adaptively

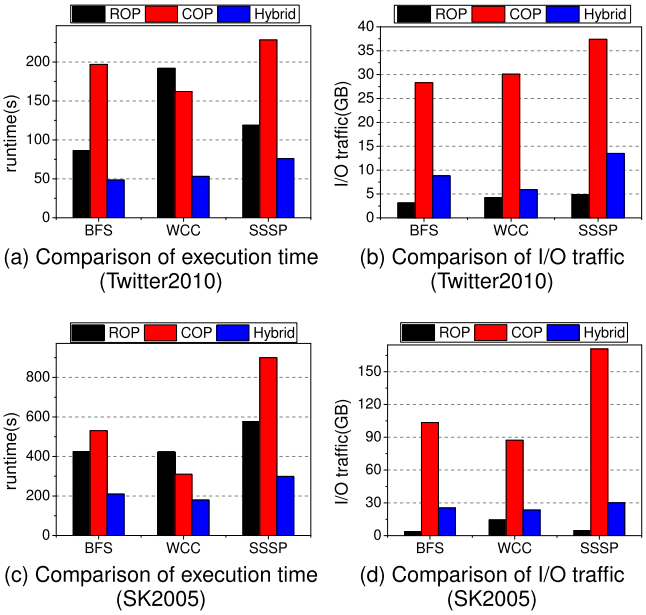


Fig. 9. Comparison of different update strategies.

switches between ROP and COP with two baseline approaches that respectively implement ROP and COP in each iteration. Figs. 9a and 9c shows the comparisons of runtime with different models on Twitter2010 and SK2005. Figs. 9b and 9d shows the corresponding comparisons of I/O traffic. As we see from the results, the Hybrid model always achieves the best performance, as it selects the optimal I/O model and update model for each vertex interval in each iteration. For BFS and SSSP where the number of active vertices is small in most iteration, COP has the worst performance as it loads the entire edges in each iteration. For WCC where most vertices are active in the first few iterations, ROP has the worst performance due to the significant overheads of random disk accesses.

As to I/O traffic, ROP enables selective data access based on the active vertices and accesses the least amount of data for all algorithms. COP streams the whole of data to achieve good I/O access locality. Thus, it accesses the most amount of data. Based on the I/O-based performance prediction method, the Hybrid model dynamically selects between ROP and COP. Therefore, the I/O traffic for the Hybrid model is moderate.

Theoretically, only using ROP can also accommodate different I/O and computation loads of graph algorithms. For sparse active edge sets, ROP only loads the active edges to skip useless data. For dense active edge sets, ROP streams

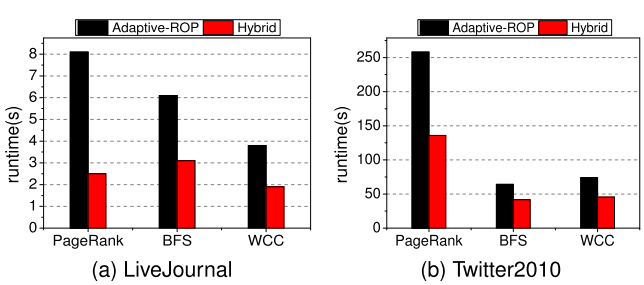


Fig. 10. Adaptive-ROP vs. hybrid.

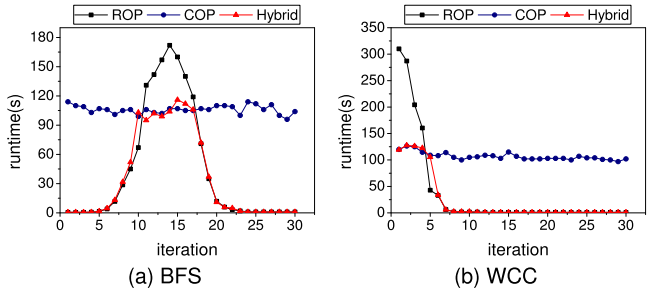


Fig. 11. Effect of the I/O-based performance prediction model.

all edges to avoid random disk accesses. The switch can be easily achieved by using our I/O-based performance prediction method (We referred this execution model as Adaptive-ROP). However, we take COP to handle dense active edge sets for better computational efficiency as follows. First, COP adopts a pull-style update model that can avoid the expensive atomic operations and make better use of parallelism especially when the number of active edges is large, as described in Section 2.2. Second, as many studies [10], [11] have shown, the pull-style update (bottom-up approach) can significantly reduce accesses to vertices that have already been visited and improve the performance of traversal-based algorithms. To further demonstrate the superiority of our Hybrid model (ROP+COP), we compare the performance of these two methods in Fig. 10. Thanks to the better computational efficiency, Hybrid model outperforms Adaptive-ROP in all cases. Specifically, for PageRank, BFS and WCC, Hybrid model respectively outperforms Adaptive-ROP by 2.6x, 1.6x and 1.8x. Actually, the more dense the active edge sets (like PageRank), the more superiority Hybrid has, due to the data contention when updating vertices faced by ROP. On the other hand, when the graph becomes larger, the superiority of Hybrid model becomes smaller due to the increasing disk I/O costs. Even though, Hybrid model still outperforms Adaptive-ROP by up to 1.9x for the Twitter graph.

4.3 Effect of I/O-Based Performance Prediction Method

To evaluate the effectiveness of the I/O-based performance prediction method, we run two algorithms (BFS and WCC) that exhibit different I/O features on Ukunion with three different update models, ROP, COP and Hybrid, and report the runtime of different models in each iteration (30 iterations) in Fig. 11. The two algorithms produce different numbers of active vertices in each iteration, which is appropriate to evaluate the accuracy of the the I/O-based performance prediction method.

As shown in Fig. 11, the performance gap between ROP and COP is quite significant. The performance of COP is relatively constant since it loads all graph data in per iteration, while the performance of ROP depends on the number active vertices. For BFS, ROP outperforms COP in most iterations, except in few iterations (iteration 11 $\sim$ 17) where there are a large number of active vertices that cause frequent random disk accesses. For WCC, COP performs better in the first few iterations when most vertices remain active, while ROP performs better when many vertices reach convergence. HUS-Graph is able to select the optimal update model based on the I/O-based performance prediction method in most iterations,

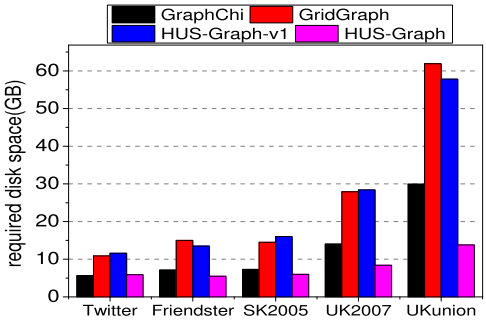


Fig. 12. Comparison of disk space consumption.

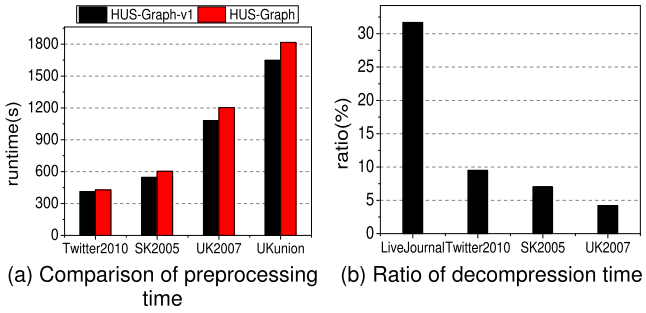


Fig. 14. Evaluating the overheads of space-efficient storage format.

except iteration 10 of BFS and iteration 5 of WCC. These wrong predictions are usually around the intersection of the ROP′ and COP′ performance curves. This indicates that we can implement a more accurate and fine-grained performance evaluation and prediction method to find the critical point where the update strategy switches between ROP and COP.

4.4 Effect of Space-Efficient Storage Format

Fig. 12 compares the required disk space of HUS-Graph with GraphChi, GridGraph and HUS-Graph-v1. Note that we only report the storage size of edges as the storage sizes of vertices and degrees are negligible. Compared with HUS-Graph-v1, HUS-Graph that incorporates the space-efficient storage format can reduce disk consumption by up to 71 percent. For GraphChi and GridGraph, they respectively store the edges in CSR and edge list format. Although they only maintain one copy of edges, the storage usages of them are 1.5x and 2.8x larger than that of HUS-Graph on average. This further shows the efficiency of the space-efficient storage format that adopts several graph compression methods.

Fig. 13 shows the required disk space and runtime when we store all the graph data without any compression (Base), using compression of undirected graph (Undirected), and when using all compression methods (All). We choose the undirected Friendster graph to conduct this experiment since it can efficiently evaluate the the effects of different compression methods. We can see that the Undirected compression can halve the storage size but has no impact on performance. This is because HUS-Graph loads either the in-edges or the out-edges for each vertex interval, and only reducing the redundant storage of edges can not reduce the I/O traffic and improve performance. When using all compression methods, the storage sizes of both in-edges and out-edges reduce, leading to reduced I/O traffic and improved algorithmic performance.

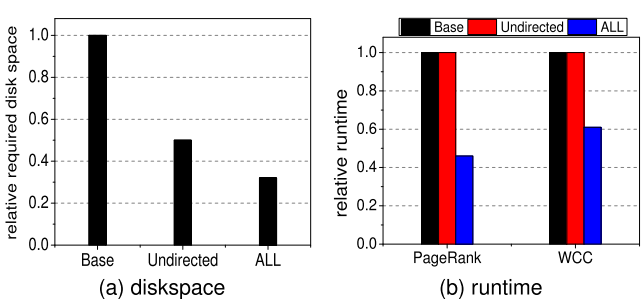


Fig. 13. Evaluating the effects of different compression methods.

We also evaluate the overheads of space-efficient storage format as it involves compression and decompression of graphs. The overheads include two aspects: more preprocessing (compression) time and extra decompression time. The decompression time refers to the time spent on transforming the compressed adjacency lists into initial adjacency lists that the system can directly process. Fig. 14 shows the evaluation results. In Fig. 14a, we compare the preprocessing time of HUS-Graph and HUS-Graph-v1 to evaluate the compression overheads. Due to the using of graph compression techniques, the preprocessing time of HUS-Graph is longer than that of HUS-Graph-v1 by 4.1-11.6 percent. Considering the benefits that the space-efficient storage format brings, these extra overheads in preprocessing are acceptable. Moreover, the graphs can be reused for many times after preprocessing. Fig. 14b shows the percentage of decompression time in overall runtime when running PageRank. The ratio of decompression time ranges from 4.2 to 31.7 percent. We can see that when the graph is larger, the ratio of decompression time is lower, since the disk I/O time dominates the overall runtime. Therefore, the space-efficient storage format may not work well for small graphs but can bring significant benefits for large graphs. In addition, the decompression procedure can be overlapped with vertex updating to further reduce the extra overheads.

4.5 Effect of Immediate Propagation-Based Vertex Update Scheme

We conduct our evaluation on the traversal-based algorithms (BFS, WCC and SSSP) with two vertex update schemes, accumulation-based vertex update (AVU) and immediate propagation-based vertex update (IPVU) schemes. Fig. 15 shows the evaluation results. With the use of the IPVU model, the performance of algorithms can improve by up to 39 percent,

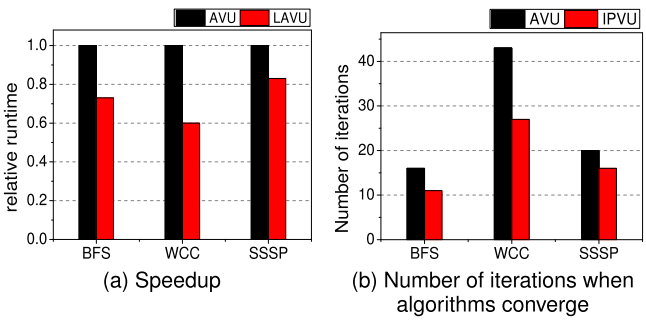


Fig. 15. Evaluating the benefits of immediate propagation-based vertex update scheme.

TABLE 3
Execution Time (in Seconds)

	Data sets	LiveJournal	Twitter2010	Friendster	SK2005	UK2007	UKunion
BFS	GraphChi	20.9	1624.3	2294.5	4973.6	7154.5	24062.3
	GridGraph	5.2	598.9	578.6	4066.3	6025.2	18929.2
	HUS-Graph-v1	3.9	70.6	133.9	424.5	1278.2	1897.9
	HUS-Graph	3.1	41.6	54.8	172.9	737.5	874.2
WCC	GraphChi	24.4	913.7	2612.3	2769.1	6862.8	15665.8
	GridGraph	5.1	522.5	526.8	3338.7	4783.8	13265.1
	HUS-Graph-v1	3.5	74.8	103.5	289.2	1068.3	1223.6
	HUS-Graph	1.9	45.6	44.9	113.9	674.1	734.2
SSSP	GraphChi	21.4	1913.9	1802.4	5415.8	11495.8	56650.9
	GridGraph	6.1	660.4	708.6	4180.7	7029.4	25554.2
	HUS-Graph-v1	4.5	106.8	164.8	605.3	1750.7	2997.9
	HUS-Graph	5.3	56.9	75.1	319.1	924.8	1461.4
PageRank	GraphChi	16.6	928.6	2562.8	970.3	2774.8	3376.6
	GridGraph	10.9	451.9	1009.4	669.1	1242.2	1829.3
	HUS-Graph-v1	2.2	230.3	349.3	291.3	600.5	922.9
	HUS-Graph	2.5	135.8	143.5	148.7	398.3	668.6

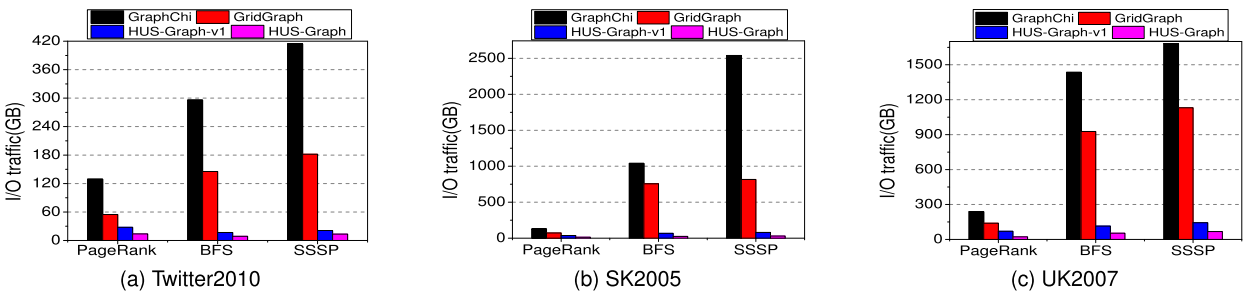


Fig. 16. I/O traffic comparison.

as shown in Fig. 15a. Fig. 15b explains why the IPVU model can improve the performance. It accelerates the vertex states propagation and convergence speed of traversal-based algorithms, reducing the number of iterations. The less iterations mean less computation overheads and unnecessary I/Os when running the algorithms.

4.6 Comparison to Other Systems

We report the execution time of the chosen algorithms on different datasets and systems in Table 3. The execution time includes the time of graph loading and vertices updating. We can see that HUS-Graph achieves a significant speedup over GraphChi and GridGraph. Specifically, HUS-Graph outperforms GraphChi by 4.1x-58.2x and GridGraph by 1.2x-29.3x.

GraphChi utilizes the vertex-centric scatter-gather processing to maximize sequential disk access. However, it writes a large amount of intermediate updates to disk, which incurs great I/O overheads. In addition, it needs a subgraph construction phase to construct the in-memory vertex-centric data structure, which is a time-consuming process [6]. GridGraph combines the scatter and gather phases into one streaming-apply phase to avoid writing the intermediate results to disk. While both GraphChi and GridGraph support the selective scheduling to reduce the loading of inactive edges, it is very coarse-grained since it can only skip processing the edge blocks without any active edges. This means it loads and processes an edge-block even though there is only one active edge. Therefore, these systems still load many useless data, which is wasteful for the traversal-based algorithms that only have a small number of active vertices in most iterations. And that is just the greatest strength for HUS-Graph that enables selective data access to avoid loading useless data. For the three traversal-based algorithms BFS, WCC, and SSSP, HUS-Graph respectively outperforms GraphChi and GridGraph by 23.9x and 12.3x on average. For the PageRank algorithm where all vertices are always active, HUS-Graph implements COP model and loads the whole of data in each iteration like other systems. Thanks to more compact storage that leads to less amount of I/O, HUS-Graph remains outperforming GraphChi and GridGraph by 8.5x and 3.9x respectively.

When compared with HUS-Graph-v1, the use of the space-efficient storage format and immediate propagation-based vertex update scheme can improve the performance by up to 2.5x. In addition, we can observe that these newly proposed optimizations bring few benefits when processing the small graph LiveJournal. This is attributed to the high decompression overhead as shown in Section 4.4.

We compare the amount of I/O traffic of HUS-Graph versus Graphchi, GridGraph and HUS-Graph-v1 in Fig. 16. For PageRank, the I/O traffic of HUS-Graph is respectively 9.8x and 4.2x smaller than that of GraphChi and GridGraph. This is mainly attributed to HUS-Graph's more compact

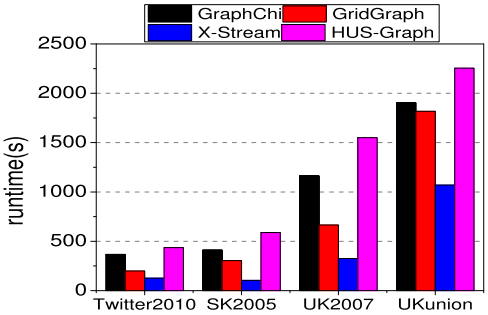


Fig. 17. Preprocessing time for different systems.

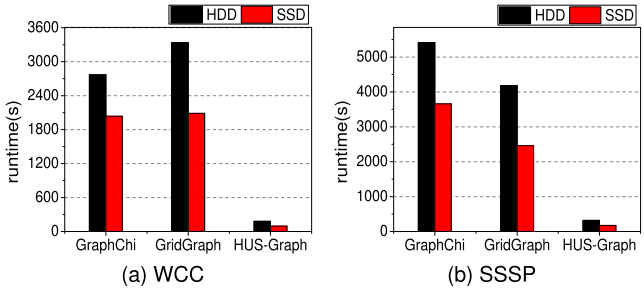


Fig. 19. Effect of I/O devices on performance.

storage format. In addition, GraphChi has to write a large amount of intermediate data (edge values) to disk for subsequent computation, while GridGraph and HUS-Graph only writes vertex values back to disk during the computation. For the two traversal-based algorithms BFS and SSSP, the I/O traffic of HUS-Graph is respectively 39.2x and 20.1x smaller than that of GraphChi and GridGraph, thanks to the combined effort of selective data access and space-efficient storage format. Compared with HUS-Graph-v1, the use of the space-efficient storage format and immediate propagation-based vertex update scheme can reduce I/O traffic by up to 68 percent, since the more compact storage and fewer number of iterations can lead to more smaller amount of I/O traffic.

We also evaluate the processing overhead (preprocessing time) of different systems. The preprocessing procedure converts an input graph into an internal form on the disk, which consists of loading raw data into memory, partitioning and building the graph. In addition to comparing with GraphChi and GridGraph, we also include the preprocessing time of X-Stream [6] that adopts an edge-centric model. As shown in Fig. 17, X-Stream takes the least preprocessing time since it does not need to sort edge lists during preprocessing [6]. It only shuffles the original edge lists to several files according to the streaming partitions [7], [22]. However, this partitioning strategy makes it inefficient for selective scheduling, which largely affects its performance on many algorithms where only a portion of the vertices are used in some iterations [7]. HUS-Graph takes more preprocessing time than other systems, since it needs to build two copies of edges and implement the space-efficient storage format. However, the overhead of the extra preprocessing is more than offset by the significant performance improvement it brings. For example, the space-efficient storage format can greatly reduce I/O traffic, leading to improved algorithmic performance. Moreover, the graphs can be reused for many times after preprocessing, and the preprocessing overheads can be significantly amortized.

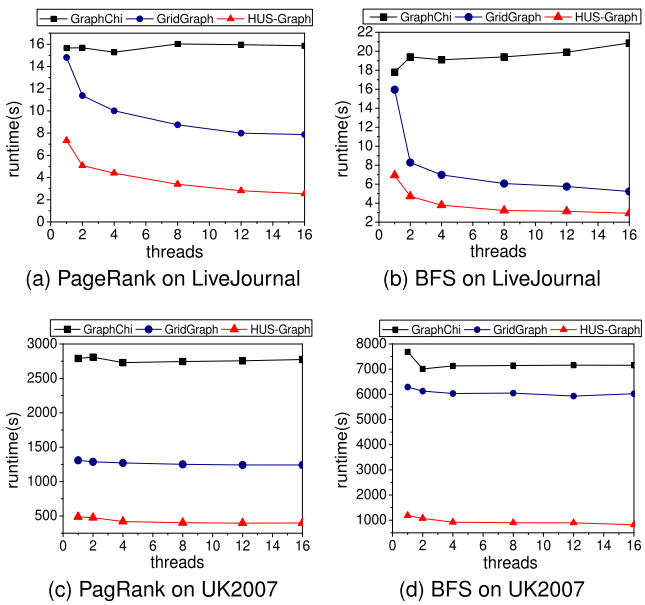


Fig. 18. Effect of the number of thread on performance.

4.7 Scalability

We evaluate the scalability of HUS-Graph by observing the improvement when more hardware resource is added. Fig. 18 shows the effect of the number of threads on system performance. To access the effect of the parallel processing on different algorithms and graphs, we choose two graph algorithms with different computation and I/O features to run on two graphs with different sizes. For the relatively small graph LiveJournal whose data can completely fit in memory, the degree of parallelism has significant impact on system performances of HUS-Graph and GridGraph due to the efficient use of parallelism. Moreover, both the performances of PageRank and BFS improve as the number of threads increases. However, 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 [5]. On the other hand, for the large graph UK2007, system performance is limited by disk I/O. Therefore, thread number has relatively less impact on the performances of the three systems.

Fig. 19 shows the performance improvement of WCC and SSSP on SK2005 when using different I/O devices. Compared with disk cases, GraphChi, GridGraph and HUS-Graph achieve an average speedup of 1.4x, 1.6x and 1.9x respectively when using SSD for WCC and SSSP. This indicates that HUS-Graph can benefit more from the utilization of SSD, since HUS-Graph enables selective (random) data access to load the active edges, which works well on SSD.

5 Related Work

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

5.1 Distributed Systems

Distributed systems usually require to hold the whole input graph and all intermediate results and messages in memory. PEGASUS [23] and GBase [24] are based on MapReduce, which works relatively well for graph algorithms such as PageRank, but performs poorly for graph traversal algorithms. Pregel [1] supports vertex-centric computing model following Bulk-Synchronous Parallel (BSP) message passing model [25]. It abstracts away the complexity of programming in a distributed-memory environment and runs users' code in parallel on a cluster. However, this model usually suffers from expensive synchronization overheads. GraphLab [26] and PowerGraph [2] executes an asynchronous model and uses shared memory for communication among vertices instead of passing messages. Gemini [4] applies multiple optimizations targeting computation performance to build scalability on top of efficiency. Chaos [27], BlitzG [28] and DD-Graph [29] utilize secondary storage to scale distributed graph processing to out-of-core scenery. Gluon [30] introduces a new approach to build distributed memory graph analytics systems that exploit heterogeneity in processor types (CPU and GPU), partitioning policies and programming models.

5.2 Single-Machine Shared-Memory Systems

Single-machine shared-memory systems typically use a high-end server with hundreds or thousands gigabytes of DRAM to hold the whole graph. Ligra [11] is a lightweight shared-memory framework and provides a programming interface optimized for graph traversal algorithms. Ligra+ [31] integrates compression techniques such delta compression into Ligra. Unlike HUS-Graph that compresses the graphs to save the disk space and further reduce I/O traffic, it uses the compression techniques to enable faster in-memory parallel graph processing using less memory footprints. Polymer [32] is a NUMA-aware graph analytics system, which is motivated by a detailed study of NUMA characteristics. Medusa [33] and Garaph [34] fully exploit the power of modern hardware and efficiently support GPU-accelerated graph processing. GraphIt [35] is a novel DSL for graph processing that generates fast implementations for algorithms with different performance characteristics running on graphs with varying sizes and structures.

5.3 Single-Machine Disk-Based Systems

Out-of-core graph processing systems enable users to analyze, process and mine large graphs in a single PC by efficiently using disks. GraphChi [5] is a pioneering single-PC-based out-of-core graph processing system that supports vertex-centric computation and is able to express many graph algorithms. 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.

Following GraphChi, a number of out-of-core graph processing systems are proposed to improve the I/O performance. X-Stream [6] uses an edge-centric approach in order to minimize random disk accesses. In each iteration, it streams and processes the entire unordered list of edges during the scatter phase and applies updates to vertices in the gather phase. GridGraph [7] 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 avoids writing updates to disk and enables selective scheduling to skip the inactive edge blocks. VENUS [14] explores a vertex-centric streamlined computing model that enables streams the graph data while performing computation. However, all these systems improve the I/O access locality at the expense of loading all graph data in all iterations, even though a large amount of data is not needed.

GraphQ [36] and Wonderland [37] use abstraction to accelerate graph processing. GraphQ improves the performance of graph queries when queries can be answered using only a few sub-graphs. Wonderland extracts graph abstractions to capture certain graph properties, and then performs abstraction-guided processing to infer better priority processing order and faster information propagation. While this abstraction-based method can accelerate convergence and reduce disk I/O, its scope of applications is limited to path-based monotonic graph algorithms.

Several systems use fast storage devices to accelerate out-of-core graph processing. FlashGraph [38] and Graphene [39] utilize SSD arrays and implement a semi-external memory graph engine to close the performance gap between in-memory and out-of-core graph processing. GraFBoost [40] adopts a sort-reduce accelerator to improve the I/O performance of flash storage. V-Part [41] extends GraFBoost by using a novel to vertex partition scheme to alleviate extra computation overheads. Compared with these systems, HUS-Graph has better adaptability and works well for both SSD and HDD.

[9] removes unnecessary I/O of out-of-core graph processing by employing dynamic partitions whose layouts are dynamically adjustable. These dynamic partitions compactly capture the set of active edges in current iteration. It achieves this by dropping inactive edges across iterations and delaying computations that cannot be performed due to missing edges. Although [9] can avoid the loading of useless data and eliminate random disk accesses, it has to write back the active edges. LUMOS [42] is a dependency-driven out-of-core graph processing system. It performs out-of-order execution to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees.

6 Conclusion

In this paper, we present an I/O-efficient out-of-core graph processing system called HUS-Graph that maximizes the I/O performance by achieving a good balance between I/O traffic and I/O access locality. HUS-Graph adopts a hybrid update strategy including Row-oriented Push (ROP) and Column-oriented Pull (COP), to schedule disk I/O adaptively according to running features of graph algorithms. Furthermore, HUS-Graph adopts a locality-optimized dual-block graph representation to organize the graph data and an I/O-based performance prediction method that enables the system to dynamically select the optimal update model based on the I/O loads of current iteration. To save the disk space and further reduce I/O traffic, HUS-Graph implement a space-efficient storage format by combining several graph compression

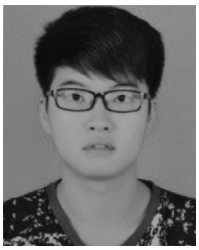
techniques. Our evaluation results show that HUS-Graph can be much faster than GraphChi and GridGraph, two state-of-the-art out-of-core systems.

Acknowledgments

This work was supported by National Defense Preliminary Research Project No. 31511010202, NSFC No. 61832020, 61772216, 61821003, and U1705261, Wuhan application basic research Project No. 2017010201010103, Hubei province technical innovation special Project No. 2017AAA129, and Fundamental Research Funds for the Central Universities. This work was also supported by the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2018WNLOKF006.

References

[1] G. Malewicz *et al.*, "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[2] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[3] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[4] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 301–316.
[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.
[7] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 375–386.
[8] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 44–54.
[9] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2016, pp. 507–522.
[10] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," *Sci. Program.*, vol. 21, no. 3–4, pp. 137–148, 2013.
[11] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 135–146, 2013.
[12] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, and Y. Zhang, "HUS-graph: I/O-efficient out-of-core graph processing with hybrid update strategy," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, Art. no. 3.
[13] Z. Wang, Y. Gu, Y. Bao, G. Yu, and J. X. Yu, "Hybrid pulling/pushing for I/O-efficient distributed and iterative graph computing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 479–494.
[14] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He, "VENUS: Vertex-centric streamlined graph computation on a single PC," in *Proc. IEEE Int. Conf. Data Eng.*, 2015, pp. 1131–1142.
[15] W.-S. Han *et al.*, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 77–85.
[16] P. Boldi and S. Vigna, "The webgraph framework I: Compression techniques," in *Proc. Int. Conf. World Wide Web*, 2004, pp. 595–602.
[17] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu, "TripleBit: A fast and compact system for large scale RDF data," *Proc. VLDB Endowment*, vol. 6, pp. 517–528, 2013.
[18] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "NXgraph: An efficient graph processing system on a single machine," in *Proc. IEEE Int. Conf. Data Eng.*, 2016, pp. 409–420.
[19] Y. Cheng *et al.*, "A communication-reduced and computation-balanced framework for fast graph computation," *Front. Comput. Sci.*, vol. 12, no. 5, pp. 887–907, 2018.
[20] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. Int. Conf. World Wide Web*, 2010, pp. 591–600.
[21] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," *ACM SIGIR Forum*, vol. 42, no. 2, pp. 33–38, 2008.
[22] P. Sun, Y. Wen, D. Ta, and X. Xiao, "GraphMP: I/O-efficient big graph analytics on a single commodity machine," *IEEE Trans. Big Data*, to be published, doi: 10.1109/TBDATA.2019.2908384.
[23] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A peta-scale graph mining system implementation and observations," in *Proc. IEEE Int. Conf. Data Mining*, pp. 229–238.
[24] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "GBASE: A scalable and general graph management system," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 1091–1099.
[25] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, pp. 103–111, 1990.
[26] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, pp. 716–727, 2012.
[27] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel, "Chaos: Scale-out graph processing from secondary storage," in *Proc. ACM Symp. Operating Syst. Princ.*, 2015, pp. 410–424.
[28] Y. Cheng *et al.*, "Using high-bandwidth networks efficiently for fast graph computation," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 5, pp. 1170–1183, May 2019.
[29] Y. Cheng *et al.*, "A highly cost-effective task scheduling strategy for very large graph computation," *Future Gener. Comput. Syst.*, vol. 89, pp. 698–712, 2018.
[30] R. Dathathri *et al.*, "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 752–768.
[31] J. Shun, L. Dhulipala, and G. E. Blelloch, "Smaller and faster: Parallel processing of compressed graphs with Ligra+," in *Proc. Data Compression Conf.*, 2015, pp. 403–412.
[32] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," *ACM SIGPLAN Notices*, vol. 50, no. 8, pp. 183–193, 2015.
[33] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, Jun. 2014.
[34] L. Ma, Z. Yang, H. Chen, J. Xue, and Y. Dai, "Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 195–207.
[35] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. Amarasinghe, "Graphit: A high-performance graph DSL," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, 2018, Art. no. 121.
[36] K. Wang, G. Xu, Z. Su, and Y. D. Liu, "GraphQ: Graph query processing with abstraction refinement—Scalable and programmable analytics over very large graphs on a single PC," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 387–401.
[37] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland: A novel abstraction-based out-of-core graph processing system," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2018, pp. 608–621.
[38] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay, "FlashGraph: Processing billion-node graphs on an array of commodity SSDs," in *Proc. USENIX Conf. File Storage Technol.*, 2015, pp. 45–58.
[39] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing," in *Proc. USENIX Conf. File Storage Technol.*, 2017, pp. 285–300.
[40] S.-W. Jun, A. Wright, S. Zhang, S. Xu, and Arvind, "GraFboost: Using accelerated flash storage for external graph analytics," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit.*, 2018, pp. 411–424.
[41] N. Elyasi, C. Choi, and A. Sivasubramaniam, "Large-scale graph processing on emerging storage devices," in *Proc. USENIX Conf. File Storage Technol.*, 2019, pp. 309–316.
[42] K. Vora, "LUMOS: Dependency-driven disk-based graph processing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2019, pp. 429–442.



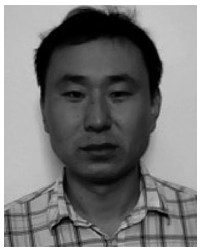
Xianghao Xu received the BE degree in computer science and technology from the LiaoNing University, ShenYang, China, in 2015. He is currently working toward the PhD degree majoring in computer architecture at the Huazhong University of Science and Technology, Wuhan, China. His current research interests include computer architecture and graph processing.

Fang Wang received the BE and master's degrees in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994 and 1997, respectively, and the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2001, where she is currently a professor of computer science and engineering. Her research interests include distribute file systems, parallel I/O storage systems, and graph processing systems. She has more than 50 publications in major journals and conferences, including the *Future Generation Computing Systems*, the *ACM Transactions on Architecture and Code Optimization*, HiPC, ICDCS, HPDC, and ICPP.

Hong Jiang (Fellow, IEEE) received the BE degree from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree from the Texas A&M University, College Station, Texas, in 1991. He is Wendell H. Nedderman endowed professor and chair of the Department of Computer Science and Engineering, University of Texas at Arlington. His research interests include computer architecture, computer storage systems, and parallel/ distributed computing. He serves as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 200 publications in major journals and international Conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *ACM Transactions on Storage*, *ACM Transactions on Architecture and Code Optimization*, *Journal of Parallel and Distributed Computing*, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, and ICPP.

Yongli Cheng received the BE degree from the Chang'an University, Xi'an, China, in 1998, the MS degree from the FuZhou University, FuZhou, China, in 2010, and the PhD degree from the Huazhong University of Science and Technology, Wuhan, China, 2017. He is currently a teacher of the College of Mathematics and Computer Science, FuZhou University currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, the *Future Generation Computing Systems*, *IEEE/ACM Transactions on Networking*, and *Frontiers of Computer Science*.

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



Yongxuan Zhang received the BE degree in computer science and technology from the Nanchang Hangkong University, Nanchang, China, in 2005. He is currently working toward the PhD degree majoring in computer science and technology at the Huazhong University of Science and Technology, Wuhan, China. His current research interests include graph processing and parallel/distributed processing.