# HUS-Graph: I/O-Efficient Out-of-Core Graph Processing with Hybrid Update Strategy

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

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

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

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

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

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

## 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 the better performance, these systems adopt a full I/O model that accesses all edges during the computation to avoid the ineffectiveness of random I/Os. Although this model ensures good I/O access locality, it loads a large number of useless edges when running graph algorithms that only require a small portion of edges in each iteration. A natural 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, since the I/O cost will grow rapidly as the number of active edges increases due to larger amount of random I/Os.

In this paper, 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 amount and I/O access locality. HUS-Graph first adopts a hybrid update strategy including two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP). It can adaptively select the optimal update model for the graph algorithms that have different computation and I/O features, based on an I/O-based performance prediction method. Furthermore, HUS-Graph proposes a dual-block representation to organize graph data, which ensures good access locality. Extensive experimental results show that HUS-Graph outperforms existing out-of-core systems by 1.4x-23.1x.

## CCS CONCEPTS

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

## KEYWORDS

graph computing, out-of-core, hybrid update strategy

**ACM Reference Format:**
Xianghao Xu, Fang Wang, Hong Jiang, Yongli Cheng, Dan Feng, and Yongxuan Zhang. 2018. HUS-Graph: I/O-Efficient Out-of-Core Graph Processing with Hybrid Update Strategy. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225108

## 1 INTRODUCTION

Graph data has been widely used to model and solve many problems in different areas such as social networks, web graphs, chemical compounds and biological structures. 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 [17], PowerGraph [11], GraphX [12] and BlitzG [6]) is a natural choice for handling these large graphs, a recent trend initiated by GraphChi [15] advocates developing out-of-core support to process large graphs on a single commodity PC.

Out-of-core graph processing systems (e.g. GraphChi [15], X-Stream [19] and GridGraph [25]) utilize secondary storage to process very large graphs and achieve scalability without massive hardware. Furthermore, they overcome the challenges faced by distributed systems, such as load imbalance and significant communication overhead. When processing an input graph, these systems divide the vertices of the graph 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

*This author is the corresponding author.
†Also with Shenzhen Huazhong University of Science and Technology Research Institute.


*ICPP 2018, August 13–16, 2018, Eugene, OR, USA*

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225108

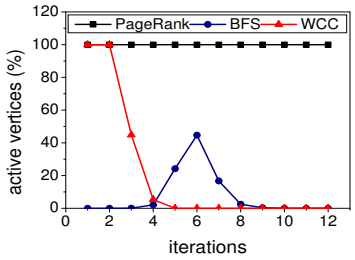



**Figure 1: The percentage of active edges per iteration**

one vertex interval and its associated edge shard at a time. To obtain the 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 are not needed.

Figure 1 shows the percentage of active edges (the edges that have active sources vertices and are needed in current iteration) per iteration for different algorithms on LiveJournal graph [1]. For PageRank[1], all edges are always active as all vertices compute their PR values in each iteration. For BFS and 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 loading the useless edges incurs significant I/O inefficiencies that degrade the overall performance. The natural 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. This dilemma motivates us to propose a new out-of-core system that enables switching between full I/O model and on-demand I/O model adaptively based on the number of active edges.

In this paper, we present HUS-Graph, an I/O-efficient out-of-core graph processing system with hybrid update strategy. Our work is inspired by state-of-art shared-memory graph processing systems ([2], [20]) that utilize an adaptive update 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 the two models, the system can handle different active edges densities with optimal performance. We extend this hybrid solution to 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 vertices 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 model. 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 edges sets, ROP only traverses the active edges and pushes updates to vertices, which fully avoids the loading of useless data. When running algorithms with dense active edges sets, COP streams all edges and update vertices by pulling updates from neighbors, which overcomes the challenges of random disk accesses and eliminates write conflicts among different threads. By utilizing an I/O-based performance prediction method, HUS-Graph can dynamically select the optimal update model based on the I/O loads of current iteration.
- **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 by up to 3.3x-23.1x and 1.4x-11.5x 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 evaluations. We discuss the related works in Section 5 and conclude this paper in Section 6.

## 2 BACKGROUND AND MOTIVATION

In this section, we first 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 update strategy to solve the I/O issues of current out-of-core systems.

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

As many works [15, 19, 25] have shown, out-of-core graph processing systems can efficiently process billion-scale graphs on a single machine. They process large graphs by efficiently using the disk drives. As the major performance bottleneck is disk I/O overhead [21], these systems are usually optimized for the sequential performance of disk drives and eliminate random I/O by scanning the

[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.

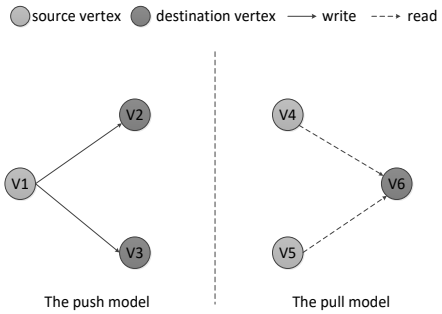



**Figure 2: The Push and Pull models**

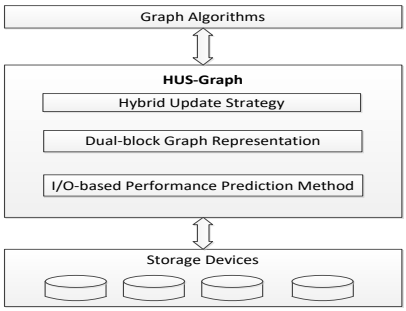



**Figure 3: The HUS-Graph Architecture**

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 Figure 1. On the other hand, the on-demand I/O model that is based on the active edges can fully 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.

## 2.2 Adaptive Push/Pull Model

The push and pull update models are extensively used in graph processing. As shown in Figure 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. Actually, different densities of active edges sets call for different update models. Specifically, sparse active edges set prefers the push model, as the system only traverses the outgoing edges of the active vertices where new updates are made. Contrarily, dense active edges set prefers the pull model, as it significantly reduces the contention in updating vertex states via locks or atomic operations. Inspired by this principle, several shared-memory systems [2, 20] and distributed systems [22, 24] adopt an adaptive update model during graph processing. For example, Ligra [20] proposes a lightweight graph processing framework that adaptively switches between the push and pull models according to the densities of active edges sets in a shared-memory machine. Gemini [24] extends such adaptive design to distributed systems and proposes a sparse-dense signal-slot model.

The current out-of-core systems either use the push model [13, 19, 25] or the pull model [5, 15]. Furthermore, the push model for these systems is based on all vertices rather than the active vertices, since they normally put a higher priority on I/O access locality rather than I/O amount. We argue that the adaptive push/pull model can efficiently solve the I/O issues of out-of-core systems and achieve a good balance between I/O access locality and I/O amount. When running algorithms with sparse active edges 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 edges sets, the pull model sequentially accesses the edges of all vertices, which overcomes the challenges of random disk accesses and enables full parallelism.

# 3 SYSTEM DESIGN

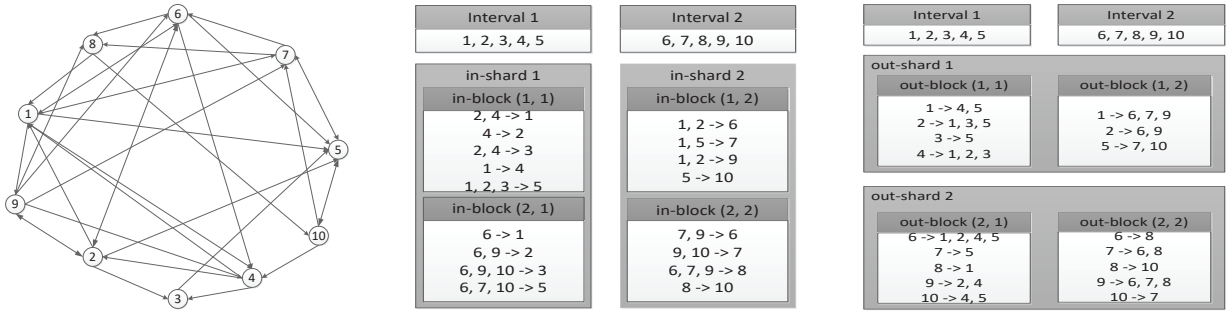
In this section, we first introduce the system overview of HUS-Graph. Then, we present the detail designs including the graph representation, hybrid update strategy and the I/O-based performance prediction method.

## 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. Running a graph algorithm on $G$ is to read and update $V$ and $E$ and updates are propagated from source vertices to destination vertices through the edges. In addition, undirected graph is supported by adding two opposite edges for each pair of vertices.

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 amount 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. Figure 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 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 loads of current iteration.

HUS-Graph is novel in two aspects compared with current out-of-core graph processing systems. First, HUS-Graph handles graph algorithms that have different computation and I/O loads (number of active edges) with hybrid update strategy to achieve a optimal performance. While current out-of-core systems either adopt a push-style update [13, 19, 25] or a pull-style update [5, 15]. Second, HUS-Graph achieves a good balance between I/O amount 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




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

**Figure 4: Illustration of the dual-block representation**

degrades the overall performance especially when the number of active edges is small.

## 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 [20, 24] that use 1-dimensional partitioning to store the out-edges/in-edges, HUS-Graph adopts a 2-dimensional partitioning 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. 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 in-block or out-block and the corresponding vertices can fit in memory, HUS-Graph can ensure good locality when processing each in-block or out-block.

Figure 4 shows the dual-block representation of an example graph for HUS-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 out-blocks (1, 2) since vertex 1 belongs to interval 1 and vertex 6 belongs to interval 2.

The storage format of edges in in-blocks and out-blocks is similar to GraphChi's Shard format [15]. In addition, the indices to the edges for each vertex are also stored. We refer to in-index$(i, j)$ as the vertices indices structure of in-blocks$(i, j)$ and out-index$(i, j)$ as the vertices indices structure of out-blocks$(i, j)$. This enables selective loading of the active edges in ROP and parallel update in COP as introduced in Section 3.3. The dual-block representation is similar to the grid [25] or Destination-Sorted Sub-Shard [10] representations that also use a 2-dimensional partitioning. Differently, the dual-block representation stores both in-edges and out-edges to enable hybrid processing.

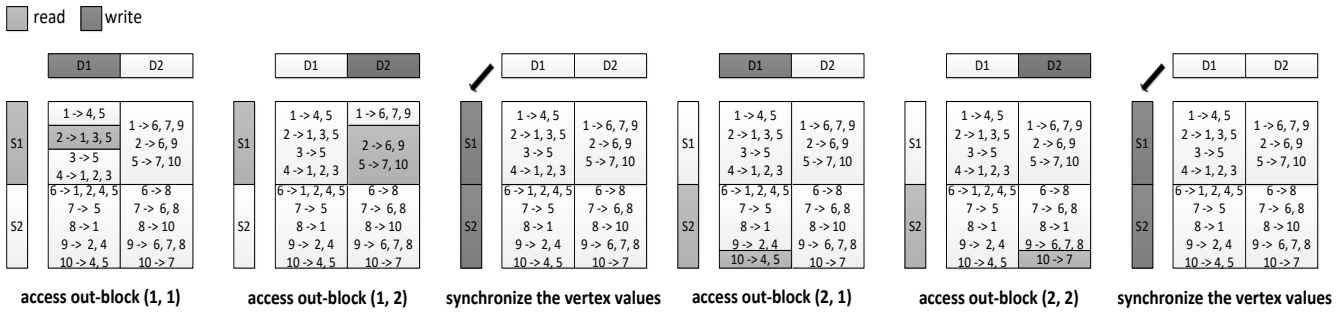
**Algorithm 1** Pseudo code of HUS-Graph execution

```
1:  for each interval i do
2:      Out ← NewActiveVerticesSet
3:      /* Identify the active vertices in interval i*/
4:      V_active ← GetActiveVertices(i)
5:      /* Determine 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
```

## 3.3 Hybrid Update Strategy

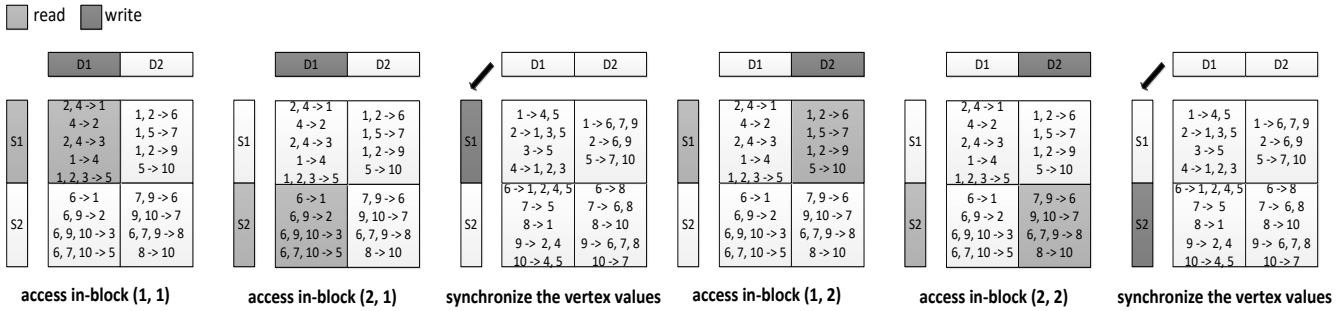
Like many out-of-core systems, HUS-Graph processes the input graph one vertex interval at a time. Algorithm 1 shows the computation procedure of HUS-Graph in one iteration. For the processing of each vertex interval, HUS-Graph proposes two update models, Row-oriented Push (ROP) and Column-oriented Pull (COP), to accommodate different I/O and computation loads of graph algorithms. ROP is applied when the number of active edges is small while COP is applied when the number of active edges is large. The selection of update model is discussed in Section 3.4. 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. Figure 5 and Figure 6 respectively illustrate the executions of ROP and COP with the example graph in Figure 4(a).

**Row-oriented Push**. ROP processes the input graph by pushing updates through the out-edges. Algorithm 2 shows the procedure




(a) Processing vertex interval 1 (b) Processing vertex interval 2

**Figure 5: Execution procedure of ROP**




(a) Processing vertex interval 1 (b) Processing vertex interval 2

**Figure 6: Execution procedure of COP**

of ROP to execute a vertex interval *i*. ROP successively accesses the out-edges of active vertices and updates their out-neighbors from out-block (*i*, 0) to out-block (*i*, *P*-1) (Line 2 ~ 16). For the processing of each out-block ($i$ , $j$) ($0 \leq j \leq P-1$), ROP first loads vertex values of $S_i$ and $D_j$ as well as the corresponding out-index. Then, each active vertex locates their out-edges in the out-block based on the corresponding out-index and loads them into memory (Line 7). 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 (Line 8 ~ 14). In this process, the vertex values in $S_i$ are read-only while the vertex values in $D_j$ are write-only. If the value of any out-neighbor is changed, this out-neighbor is added to the new active vertices set and will be scheduled in the next iteration. After the active edges in all out-blocks of interval $i$ (in the $i_{th}$ row) are processed, ROP synchronizes the vertex values by replacing the values of source intervals with the values of destination intervals for subsequent computation (Line 17 ~ 19).

In the example of Figure 5, ROP iterates over out-block (1, 1) to out-block (1, 2) when processing interval 1. The active vertices of current iteration are vertex 2, 5 and 10. Therefore, ROP successively loads and processes the out-edges of vertex 2 and 5 when executing each out-block. 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.

**Algorithm 2** Pseudo code of *RowOrientedPush* function

```
1:  LoadfromDisk(S_i)
2:  for j from 0 to P-1 do
3:      LoadfromDisk(D_j)
4:      OutIndex ← out − index(i, j)
5:      for each active vertex v in V_active do
6:          out − degree ← OutIndex(v + 1) − OutIndex(v)
7:          edges ← LoadOutEdges(OutIndex(v), out −
    degree, out − block(i, j))
8:          for each edge e in edges do
9:              neighbor ← e.dst
10:             UserDefinedFunction(v, neighbor)
11:             if IsChanged(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 do
18:     Swap(S_j, D_j)
19: end for
```

ROP puts a higher priority on I/O amount rather than I/O access locality when processing a graph. It enables selective disk I/O and

minimizes the amount of data transfer. 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, since the destination intervals of different out-blocks in a row are disjoint.

**Column-oriented Pull.** COP processes the input graph by pulling updates through the in-edges and update the vertices. Algorithm 3 shows the procedure of COP when executing a vertex interval *i*. COP streams the in-edges from in-block (0, *i*) to in-block (*P-1*, *i*) (Line 2 ~ 19). For the processing of each in-block (*j*, *i*) ($0 \le j \le p-1$), COP streams all in-edges of the in-block and load the vertex 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 8). Then, it collects data from the in-neighbors by reading $S_j$ and updates its own value with a user-defined update function (Line 9 ~ 12). If the value of any vertex in $D_i$ is updated, this vertex is added to the new active vertices set. After all in-blocks of the interval $i$ (in the $i_{th}$ column) are processed, COP replaces the $S_i$ with $D_i$ to synchronize the vertex values (Line 20).

**Algorithm 3** Pseudo code of *ColumnOrientedPush* function

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

In the example of Figure 6, 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 the next column.

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. 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 access locality.

**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 vertices set of interval i |
| $d_i$ | out-degree of vertex i |
| $M$ | size of an edge |
| $N$ | size of a vertex value |
| $T_{sequential}$ | sequential disk access throughput |
| $T_{random}$ | random disk access throughput |

## 3.4 I/O-based Performance Prediction Method

The key to gain optimal performance is to select the fastest update model between ROP and COP. Shared-memory systems [2, 20] select between push and pull model by empirically setting a threshold $\theta$ and comparing the threshold with the computing loads of current iteration (number of edges processed). For example, Ligra [20] sets $\theta$ to $|E|/20$ in all its applications. If the number of active edges is smaller than $\theta$, the push model is applied. Otherwise, the pull model is applied. Unlike shared-memory systems [2] or distributed systems [8, 9] whose performance depend on CPU performance or communication cost, the performance of an out-of-core system is mainly up to the I/O costs of loading edges from disk [21]. Instead of finding a threshold that indicates the system to select proper update model, 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 loads of current iteration.

Concretely, we refer to $C_{rop}$ as the time cost of loading the active edges in ROP model, and refer to $C_{cop}$ as the time cost of loading all in-edges in COP model. The time cost can be calculated by the total size of data accessed divided by the random/sequential throughput of disk access. Supposing the active vertices set of vertex interval $i$ is $A_i$, the number of the active edges is $\sum_{v \in A_i} d_v$, where $d_v$ denotes the out-degree of vertex $v$. Let $M$ be the size of an edge structure and $N$ be the size of a vertex value record. For the ease of expression, we assume the number of vertices and edges in each interval are equal to $|V|/P$ and $|E|/P$ respectively. In addition, $T_{random}$ and $T_{sequential}$ respectively represent the random access and sequential access throughput. For easy reference, we list the notations in Table 1.

In ROP, HUS-Graph only loads the out-edges of the active vertices. In addition, corresponding source and destination vertices as well as the vertices indices are also loaded into memory. Therefore, $C_{rop}$ of vertex interval $i$ can be stated as:

$$C_{rop} = \frac{\sum_{v \in A_i} d_v \times M + (\frac{2|V|}{P} + |V|) \times N}{T_{random}}$$

In COP, HUS-Graph loads the in-edges of all vertices within an interval. Thus $C_{cop}$ of vertex interval i is constant and can be stated as:

$$C_{cop} = \frac{\frac{|E|}{P} \times M + (\frac{2|V|}{P} + |V|) \times N}{T_{sequential}}$$

If $C_{rop} \leq C_{cop}$, we have

$$\frac{\sum_{v \in A_i} d_v \times M + (\frac{2|V|}{P} + |V|) \times N}{\frac{|E|}{P} \times M + (\frac{2|V|}{P} + |V|) \times N} \leq \frac{T_{random}}{T_{sequential}}$$

The parameters $A_i$, $d_v$, $|E|$ and $|V|$ can all be collected and computed in the runtime. Furthermore, the disk access throughput $T_{random}$ and $T_{sequential}$ can be measured by using several measurement tools such as fio [19] before we conduct the experiments. This provides an accurate performance estimate that enables the system to select the optimal 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 $\alpha$. In our experiments, $\alpha$ is empirically set to 5% of all vertices. If the number of active vertices is larger than $\alpha$, HUS-Graph selects COP model regardless of the current active vertices.

### 3.5 Fine-grained Parallelism in HUS-Graph

HUS-Graph provides fine-grained parallelism to improve the system performance in both ROP and COP. It exploits the power of multi-thread CPU as follows.

For ROP, it enables overlapped processing of different out-blocks when executing a vertex interval. Since the destination intervals of different out-blocks in a row are disjoint, worker threads that takes charge of different out-blocks can overlap with each other to enable high-degree parallelism. For COP, the processing of the in-blocks in a column can not be overlapped. However, the high parallelism within an in-block is still valid. Specifically, COP creates several worker threads that take charge of different destination vertices and their associated in-edges within an in-block. There is no write conflict between these worker threads and the access of in-edges is sequential since the in-edges are sorted by the destination vertices. Furthermore, CPU processing and disk I/O are overlapped as much as possible thanks to the parallel processing between different out-blocks or within an in-block. For example, the out-edges of the next out-block can be loaded before the processing of current out-block is finished if the memory is sufficient.

## 4 EVALUATION

In this section, we first introduce the evaluation environment and the graph algorithms. Then, we explore the effects of the system designs including hybrid update strategy and I/O-based performance prediction method. Next, we compare HUS-Graph with state-of-art out-of-core systems in terms of runtime of graph algorithms and I/O amount. Finally, we evaluate the scalability of HUS-Graph.

**Table 2: Datasets used in evaluation**

| Dataset | Vertices | Edges | Type |
|---|---|---|---|
| LiveJournal [1] | 4.8 million | 69 million | Social Graphs |
| Twitter2010 [14] | 42 million | 1.5 billion | Social Graphs |
| SK2005 [4] | 51 million | 1.9 billion | Social Graphs |
| UK2007 [3] | 106 million | 3.7 billion | Web Graphs |
| UKunion [3] | 133 million | 5.5 bilion | Web Graphs |

### 4.1 Experiment Setup

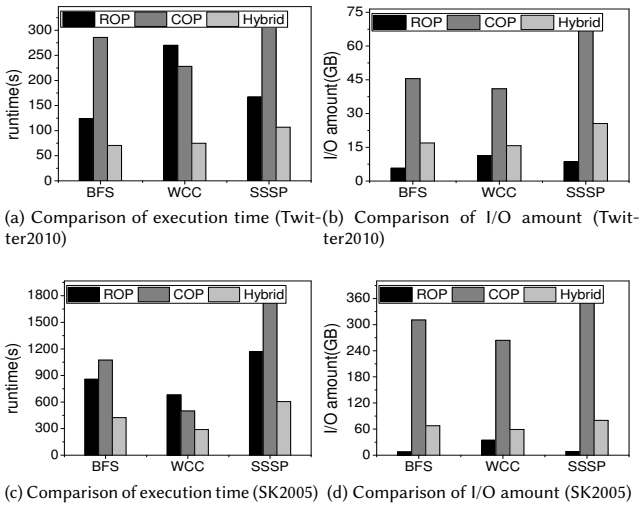
All experiments are conducted on a 16-core commodity machine equipped with 16GB main memory and 500GB 7200RPM HDD, running Ubuntu 16.04 LTS. In addition, a 128GB SATA2 SSD is installed to evaluate the scalability. The datasets for our evaluation are all real-world graphs with power-law degree distributions, summarized in Table 2. LiveJournal, Twitter2010 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 four graphs are respectively 1.5x, 2.1x, 3.9x and 6.1x larger than available memory.

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.

We compare HUS-Graph with two state-of-art out-of-core systems, GraphChi [15] and GridGraph [25]. 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 8GB memory budget, 16 execution threads for the executions of all algorithms.

### 4.2 Effect of Hybrid Update Strategy

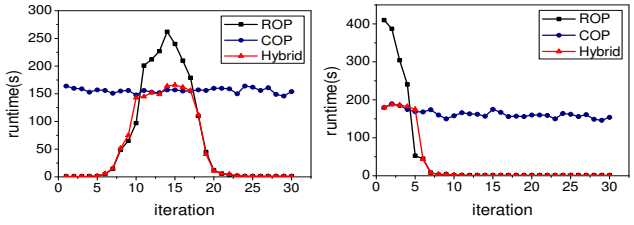
Figure 7 shows the effect of the hybrid update strategy of HUS-Graph, by comparing the Hybrid model that adaptively switches between ROP and COP with two baseline approaches that respectively implement ROP and COP in all iterations. Figure 7(a) and Figure 7(c) shows the comparisons of runtime with different models on Twitter2010 and SK2005. Figure 7(b) and Figure 7(d) shows the corresponding comparisons of I/O amount. 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




(a) Comparison of execution time (Twitter2010) (b) Comparison of I/O amount (Twitter2010)

(c) Comparison of execution time (SK2005) (d) Comparison of I/O amount (SK2005)

**Figure 7: Comparison of different update strategies when runtime BFS, WCC and SSSP on Twitter2010 and SK2005**




(a) BFS (b) WCC

**Figure 8: Effect of the I/O-based performance prediction model**

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 amount, 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 amount for the Hybrid model is moderate.

## 4.3 Effect of I/O-based Performance Prediction Method

To evaluate the effectiveness of the I/O-based performance prediction method, we run BFS and WCC on Ukunion with three different update models, ROP, COP and Hybrid, and report the runtime of different models in each iteration (30 iterations) in Figure 8.

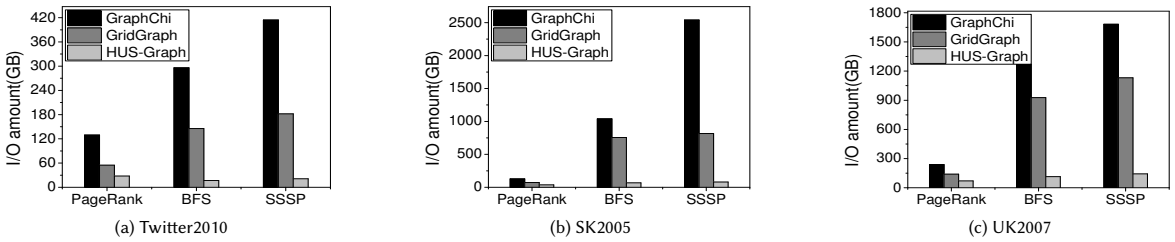
**Table 3: Execution time (in seconds)**

| | PageRank | BFS | WCC | SSSP |
|---|---|---|---|---|
| **LiveJournal** | | | | |
| GraphChi | 16.6 | 20.9 | 24.4 | 21.4 |
| GridGraph | 10.9 | 5.2 | 5.1 | 6.1 |
| HUS-Graph | 2.2 | 3.9 | 3.5 | 4.5 |
| **Twitter2010** | | | | |
| GraphChi | 928.6 | 1624.3 | 913.7 | 1913.9 |
| GridGraph | 451.9 | 598.9 | 522.5 | 660.4 |
| HUS-Graph | 230.3 | 70.6 | 74.8 | 106.8 |
| **SK2005** | | | | |
| GraphChi | 970.3 | 4973.6 | 2769.1 | 5415.8 |
| GridGraph | 669.1 | 4066.3 | 3338.7 | 4180.7 |
| HUS-Graph | 291.3 | 424.5 | 289.2 | 605.3 |
| **UK2007** | | | | |
| GraphChi | 2774.8 | 7154.5 | 6862.8 | 11495.8 |
| GridGraph | 1242.2 | 6025.2 | 4783.8 | 7029.4 |
| HUS-Graph | 600.5 | 1278.2 | 1068.3 | 1750.7 |
| **Ukunion** | | | | |
| GraphChi | 3376.6 | 24062.3 | 15665.8 | 56650.9 |
| GridGraph | 1829.3 | 18929.2 | 13265.1 | 25554.2 |
| HUS-Graph | 922.9 | 1897.9 | 1223.6 | 2797.9 |

As shown in Figure 8, 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 ~ 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, 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 Comparison to Other Systems

We report the execution time of the chosen algorithms on different datasets and systems in Table 3. We can see that HUS-Graph achieves a significant speedup over GraphChi and GridGraph. Specifically, HUS-Graph outperforms GraphChi by 3.3x-23.1x and GridGraph by 1.4x-11.5x.




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

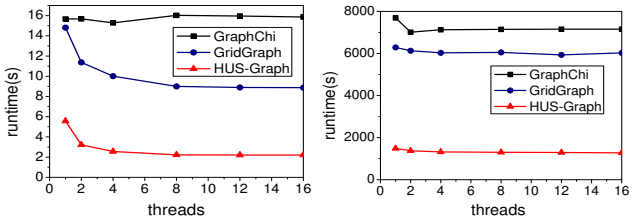
**Figure 9: I/O amount comparison**

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 [19]. GridGraph combines the scatter and gather phases into one streaming-apply phase to avoid writing the intermediate results to disk. Furthermore, it supports the selective scheduling to skip the edge blocks that are not scheduled. Nevertheless, these systems both load the entire graph in each iteration, which is wasteful for the propagation-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 propagation-based algorithms BFS, WCC, and SSSP, HUS-Graph respectively outperforms GraphChi and GridGraph by 11.2x and 6.4x 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 and more fine-grained utilization of parallelism, HUS-Graph remains outperforming GraphChi and GridGraph by 4.6x and 3.2x respectively.

We compare the amount of I/O traffic of HUS-Graph versus Graphchi and GridGraph in Figure 9. For PageRank, the I/O amount of HUS-Graph is respectively 3.9x and 1.9x smaller than that of GraphChi and GridGraph. This is attributed to HUS-Graph's storage format that is more space-efficient than the edge list format that GridGraph uses. For the two propagation-based algorithms BFS and SSSP, the I/O amount of HUS-Graph is respectively 18.4x and 8.8x smaller than that of GraphChi and GridGraph, thanks to the selective data access of HUS-Graph. 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.

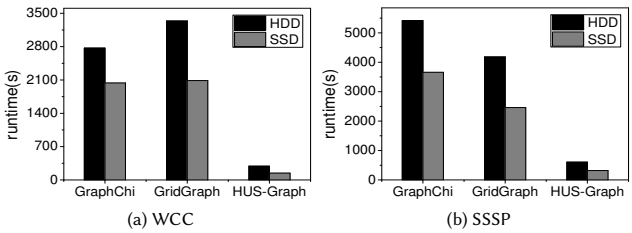
## 4.5 Scalability

We evaluate the scalability of HUS-Graph by observing the improvement when more hardware resource is added. Figure 10 shows the effect of number of threads on system performance. 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. However, GraphChi shows poor scalability as we




(a) PageRank on Twitter2010 (b) BFS on UK2007

**Figure 10: Effect of the number of thread on performance**




(a) WCC (b) SSSP

**Figure 11: Effect of I/O devices on performance**

increase the number of threads. The main blame is GraphChi's deterministic parallelism that limits the utilization of multi-threads [15]. 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.

Figure 11 shows the performance improvement of WCC and SSSP on SK2005 when using different I/O devices. Compared with disk performance, GraphChi, X-Stream and HUS-Graph achieves a speedup of 1.4x, 1.6x and 1.9x respectively when using SSD. 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

Out-of-core graph processing systems enable users to analyze, process and mine large graphs in a single PC by efficiently using disks. GraphChi [15] 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 [19] 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 [25] 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 [5] explores a vertex-centric streamlined computing model that enables streams the graph data while performing computation. Besides the single-machine systems, several distributed systems [7, 18] also support out-of-core processing to improve the scalability. 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.

FlashGraph [23] and Graphene [16] implement a semi-external memory graph engine that stores the vertex values in memory and adjacency lists on SSDs, and closes the performance gap between in-memory and out-of-core graph processing. They both rely on expensive SSD arrays and large memory to provide high IO bandwidth and cache all vertex data. Although these systems enable selective data accesses to only access the useful data, they are designed to match up the performance of in-memory processing, while most of out-of-core systems are HDD-friendly and aim to achieve reasonable performance with low hardware costs. [21] provides a general optimization for out-of-core graph processing, which removes unnecessary IO by employing dynamic partitions whose layouts are dynamically adjustable. However, it incurs significant extra computation overheads to construct the dynamic shards and synchronize the computation.

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

## REFERENCES

[1] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *KDD'06*. ACM, 44–54.
[2] Scott Beamer, Krste Asanović, and David Patterson. 2013. Direction-optimizing breadth-first search. *Scientific Programming* 21, 3-4 (2013), 137–148.
[3] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware web graph. In *ACM SIGIR Forum*, Vol. 42. ACM, 33–38.
[4] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW'04*. ACM, 595–602.
[5] 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.
[6] Yongli Cheng, Hong Jiang, Fang Wang, Yu Hua, and Dan Feng. 2017. BlitzG: Exploiting high-bandwidth networks for fast graph processing. In *INFOCOM'17*. IEEE, 1–9.
[7] YongLi Cheng, Fang Wang, Hong Jiang, Yu Hua, Dan Feng, and XiuNeng Wang. 2016. DD-Graph: A Highly Cost-Effective Distributed Disk-based Graph-Processing Framework. In *HPDC'16*. ACM, 259–262.
[8] Yongli Cheng, Fang Wang, Hong Jiang, Yu Hua, Dan Feng, and Xiuneng Wang. 2016. LCC-Graph: A high-performance graph-processing framework with low communication costs. In *IWQoS'16*. IEEE, 1–10.
[9] Yongli Cheng, Fang Wang, Hong Jiang, Yu Hua, Dan Feng, Jun Zhou, and Lingling Zhang. 2017. A Communication-reduced and Computation-balanced Framework for Fast Graph Computation. In *Frontiers of Computer Science*. https://doi.org/10.1007/s11704-018-6400-1
[10] 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.
[11] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12*. 17–30.
[12] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework.. In *OSDI'14*. 599–613.
[13] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. 2013. TurboGraph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD'13*. ACM, 77–85.
[14] 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.
[15] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a pc. In *OSDI'12*. USENIX, 31–46.
[16] Hang Liu and H Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing.. In *FAST'17*. 285–300.
[17] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD'10*. ACM, 135–146.
[18] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *SOSP'15*. ACM, 410–424.
[19] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP'13*. ACM, 472–488.
[20] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.
[21] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX ATC'16*. 507–522.
[22] Zhigang Wang, Yu Gu, Yubin Bao, Ge Yu, and Jeffrey Xu Yu. 2016. Hybrid pulling/pushing for i/o-efficient distributed and iterative graph computing. In *SIGMOD'16*. 479–494.
[23] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing billion-node graphs on an array of commodity SSDs. In *FAST'15*. 45–58.
[24] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *OSDI'16*. 301–316.
[25] 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.