GraphCP: An I/O-Efficient Concurrent Graph Processing Framework

Xianghao Xu*, Fang Wang*, Hong Jiang †, Yongli Cheng ‡, Dan Feng*, Yongxuan Zhang*, Peng Fang*
*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, China
†Department of Computer Science & Engineering, University of Texas at Arlington, USA
‡College of Mathematics and Computer Science, FuZhou University, China
Email: {xianghao, wangfang, dfeng, fangpeng}@hust.edu.cn, hong.jiang@uta.edu, chengyongli@fzu.edu.cn

***Abstract*—Big data applications increasingly rely on the analysis of large graphs. In order to analyze and process the large graphs with high cost efficiency, researchers have developed a number of out-of-core graph processing systems in recent years based on just one commodity computer. On the other hand, with the rapidly growing need of analyzing graphs in the real-world, graph processing systems have to efficiently handle massive concurrent graph processing (CGP) jobs. Unfortunately, due to the inherent design for single graph processing job, existing out-of-core graph processing systems usually incur redundant data accesses and storage and severe competition of I/O bandwidth when handling the CGP jobs, thus leading to very long waiting time experienced by users for the computing results. In this paper, we propose an I/O-efficient out-of-core graph processing system, GraphCP, to support the processing of CGP jobs. GraphCP proposes a benefit-aware sharing execution model that shares the I/O access and processing of graph data among the CGP jobs and adaptively schedules the loading of graph data, which efficiently overcomes above challenges faced by existing out-of-core graph processing systems. In addition, GraphCP organizes the graph data with a Source-Sorted Sub-Block graph representation for better processing capacity and I/O access locality. Extensive evaluation results show that GraphCP is 10.3x and 4.6x faster than two state-of-the-art out-of-core graph processing systems GridGraph and GraphZ respectively, and 2.1x faster than a CGP-oriented graph processing system Seraph.**

***Index Terms*—graph processing, I/O, concurrent processing**

I. Introduction

Graph is a powerful data structure to model and solve many real-world problems. There are various modern big data applications relying on graph computing, including social networks, Internet of things, and neural networks. However, with the real-world graphs growing in size and complexity, processing these large and complex graphs in a scalable way has become increasingly more challenging. While a distributed system (e.g., Pregel [1], GraphLab [2], PowerGraph [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], GridGraph [7], LUMOS [8] and HUS-Graph [9]) efficiently use the secondary storage (e.g., hard disk, SSD) to process large graphs in a single compute node. As we know, the secondary storage has much larger capacity and lower price than the DRAM. Therefore, the out-of-core graph processing systems can scale to very large graphs without expensive hardware, serving as a promising alternative to distributed solutions. Furthermore, they overcome the challenges faced by distributed systems, such as load imbalance problem [10] and significant communication overheads [11]. For an input graph, out-of-core graph processing systems divide the vertices of the graph into disjoint intervals and break the large edge list into smaller blocks containing edges with source or destination vertices in corresponding vertex intervals so that each edge block can fit in memory. When processing the graph, they load and process each vertex interval and its associated edge block from disk at a time.

On the other hand, with the increasing demand for graph analytics, many iterative graph algorithms run as concurrent services on a common platform. These concurrent iterative graph processing (CGP) jobs are usually executed on the same graph simultaneously so as to analyze it for various information. For example, Facebook [12] uses Apache Giraph [13] that runs different graph algorithms (e.g., label propagation, variants of Pagerank, k-means clustering) simultaneously to provide various information for their many products and services. Some graph processing systems such as Seraph [14], [15] are proposed to support the execution of CGP jobs. However, these systems usually rely on a distributed or shared-memory system, which incurs significant communication overheads among compute nodes or poor scalability when processing large-scale graphs due to the limited memory capacity. The enormous amount of intermediate messages produced by the CGP jobs significantly exacerbate these problems. This motivates us to use the cost-effective out-of-core systems that have better scalability to handle these CGP jobs.

Unfortunately, although existing out-of-core graph processing systems can efficiently process a single graph processing job, they suffer from poor performance when handling the CGP jobs. As the CGP jobs iteratively traverse the graph along different paths for their own purposes, there are a large number of intersections among the graph data being accessed by these jobs in each iteration, which produces redundant disk I/O and memory storage overheads. Moreover, since each individual job initiates the I/O request to the disk separately,

Fang Wang is the corresponding author.

978-0-7381-3207-5/21/$31.00 ©2021 IEEE

it incurs severe competition for the limited I/O bandwidth, which greatly reduces the I/O throughput and leads to very long waiting time experienced by users for the computing results, significantly reducing the quality of service. Some CGP-oriented graph processing systems like CGraph [16] and GraphM [17] can solve above problems and support disk-based processing. However, they mainly focus on in-memory processing and improving cache performance and ignore improving the disk I/O performance. For example, they can not skip loading the useless data when the number of active edges is very small. In addition, they maintain many copies of vertex values (for different CGP jobs) in memory, which limits the processing capacity and scalability as the size of graph dataset continues to grow.

Based on the above analysis, we present GraphCP, an I/O-efficient graph processing system to handle the CGP jobs. The main contributions of GraphCP are summarized as follows.

- GraphCP proposes a benefit-aware sharing execution model that shares the I/O accesses of CGP jobs by loading and processing a graph partition in a common order for all CGP jobs, which greatly reduces the redundant accesses and avoids the competition of I/O bandwidth. In addition, this model adaptively schedules the loading of graph data by skipping loading and processing inactive edges in each iteration whenever such skipping can bring performance benefit, to further improve the disk I/O performance.
- GraphCP proposes a Source-Sorted Sub-Block graph representation that adopts a 2-dimensional partitioning method to partition the graph into several sub-blocks. By restricting data access to each sub-block and corresponding source and destination vertices, GraphCP can improve the processing capacity for very large graphs and ensure good I/O access locality.
- We evaluate the performance of GraphCP by comparing with state-of-art graph processing systems including GridGraph, GraphZ and Seraph. Extensive evaluation results show that GraphCP outperforms GridGraph, GraphZ and Seraph by 10.3x, 4.6x and 2.1x on average thanks to a great improvement of I/O performance.

The rest of the paper is organized as follows. Section II presents the background and related works. Section III describes the detailed system designs of GraphCP. Section IV presents extensive performance evaluations. We conclude this paper in Section V.

II. Background And Related Work

A. Out-of-Core Graph Processing

Recently, many out-of-core (disk-based) graph processing systems have been proposed to enable users to analyze, process and mine large graphs in a single PC by efficiently using secondary storage. GraphChi [5] is a pioneering single-PC-based out-of-core graph processing system that supports vertex-centric computation model [1] 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 method 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. Dynamic Shards [18] removes unnecessary I/O of out-of-core graph processing by employing dynamic partitions whose layouts are dynamically adjustable. GraphZ [19] supports out-of-core graph analytics by adopting two innovations. One is degree-ordered storage, a new storage format that dramatically lowers book-keeping overhead when graphs are larger than memory. The other is ordered dynamic messages which update their destination immediately, reducing both the memory required for intermediate storage and IO pressure. CLIP [20] and Lumos [8] adopt an out-of-order execution model to make full use of the loaded blocks to avoid loading the corresponding graph portions in future iterations. Their cross-iteration value propagation method can significantly speedup the convergence of graph algorithms and reduce disk I/O.

Although these out-of-core graph processing systems can efficiently process a single graph processing job, they are faced with two key challenges when handling the CGP jobs. First, since the CGP jobs iteratively traverse the graph along different paths, there are a large number of intersections among the graph data being accessed by these jobs in each iteration, which means some portions of the graph are usually accessed many times in each iteration. This causes redundant data accesses and storage overheads. Second, since each individual job initiates the I/O request to the disk separately, it incurs severe competition for the limited I/O bandwidth, which greatly reduces the I/O throughput and performance.

B. Concurrent Graph Processing

With the increasing demand for graph analytics, many iterative graph algorithms run as concurrent services on a common platform. These concurrent iterative graph processing (CGP) jobs are usually executed on the same graph simultaneously so as to analyze it for various information. For example, Facebook [12] uses Apache Giraph [13] to process various graph algorithms across their many products, including typical algorithms such as label propagation, variants of Pagerank, k-means clustering, etc. Figure 1 depicts the number of CGP jobs over a large Chinese social network [16]. The stable distribution shows that more than 83.4% of the time has at least two CGP jobs executed simultaneously. The average number of concurrent jobs is 8.7. At the peak time, over 20 CGP jobs are submitted to the same platform. This indicates that the need of concurrent graph processing is increasingly growing in the real world.

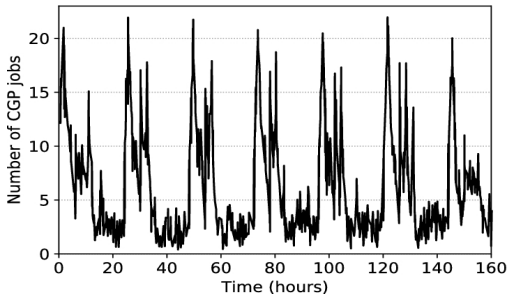


Fig. 1: The number of CGP jobs

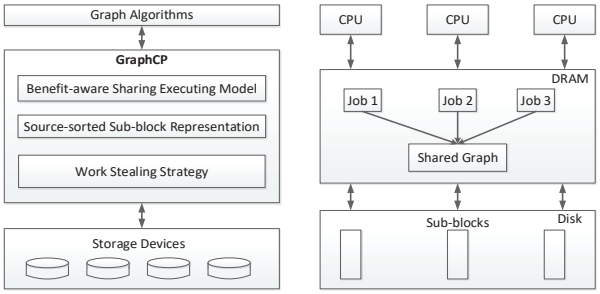


Fig. 2: The GraphCP Architecture

To handle multiple concurrent graph processing and queries, several CGP-oriented graph processing and querying systems are developed in recent years. Seraph [14], [15] decouples graph data into graph structure data and application-specific data, and enables massive CGP jobs to correctly share one copy of the in-memory graph structure data. Congara [21] schedules a group of concurrent queries to fully utilize the memory bandwidth while preventing contention between different queries. It relies upon off-line profiling with different number of threads to determine the scalability and memory bandwidth consumption of different graph algorithms on different input graphs. C-Graph [22] is an edge-set based concurrent graph traversal framework that achieves both high concurrency and efficiency for k-hop reachability queries. Unfortunately, these systems usually rely on a distributed or shared-memory system, which suffers from the problems such as significant communication overheads and poor scalability. The enormous amount of update messages produced by the CGP jobs significantly exacerbate these problems.

CGraph [16] proposes a correlations-aware execution model together with a core-subgraph based scheduling algorithm to efficiently share the graph structure data in memory and their accesses by fully exploiting such correlations. GraphM [17] is an efficient storage system that can be integrated into the existing graph processing systems to efficiently support concurrent iterative graph processing jobs for higher throughput by fully exploiting the similarities of the data accesses between these concurrent jobs. Although they can process CGP jobs from secondary storage, they mainly focus on in-memory processing and improving cache performance and ignore improving the disk I/O performance. For example, they do not consider the usability of loaded graph data. In addition, they maintain many copies of vertex values for different CGP jobs in memory, which limits the processing capacity and scalability when the graph is very large.

III. System Design

In this section, we first present the system overview of GraphCP. Then, we introduce the detail designs such as the Source-Sorted Sub-Block graph representation and benefit-aware sharing execution model. Finally, we present the programming model of GraphCP.

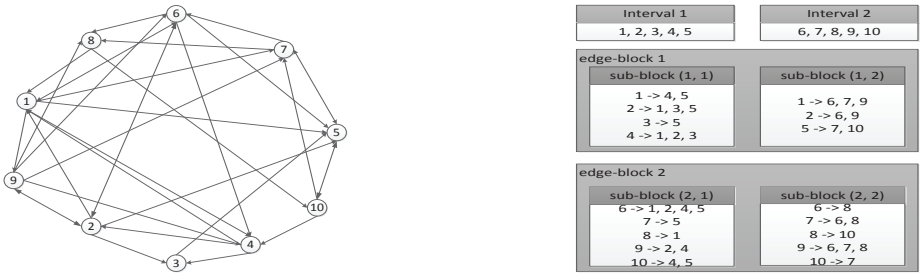
A. System Overview

A graph problem is usually encoded as a directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. For a directed edge $e = (u, v)$, we refer to e as v's in-edge, and u's out-edge. Additionally, u is an in-neighbor of v, v is an out-neighbor of u. The computation of a graph G is usually organized in several iterations where V and E are read and updated. Updating messages are propagated from source vertices to destination vertices through the edges. The computation terminates after a given number of iterations or when it converges. Like previous works [23], [24], we treat all vertices as mutable data and edges as read-only data. Furthermore, this optimization does not result in any loss of expressiveness as mutable data associated with edge $e = (u, v)$ can be stored in vertex u [23]. Therefore, only the vertex values are updated during the computation.

GraphCP is a novel out-of-core graph processing system that can efficiently execute CGP jobs. Figure 2 presents the system architecture of GraphCP. GraphCP aims to solve two key problems faced by existing works when handling CGP jobs. One is the redundant accesses and I/O competition problem. The other is the limited processing capacity and scalability problem. For the former, GraphCP proposes a benefit-aware sharing execution model that shares the I/O accesses of CGP jobs by loading the graph in a common order for all CGP jobs. In addition, this model adaptively schedules the loading of graph data to further improve the disk I/O performance. For the latter, GraphCP uses a Source-Sorted Sub-Block graph representation that adopts a 2-dimensional partitioning method that partitions the graph into several sub-blocks. This graph representation can improve the processing capacity and scalability as well as ensure the I/O access locality. In addition, a work stealing strategy is utilized to address the loads imbalance problem that stems from the skewed computation loads of different CGP jobs.

B. Graph Representation

In order to efficiently support the processing of CGP jobs under a limited memory capacity, GraphCP adopts a 2-dimensional partitioning method and implements a Source-Sorted Sub-Block (SSSB) graph representation. Like many out-of-core graph processing systems, GraphCP first splits the vertices V of graph G into P disjoint vertex intervals. Then,



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

Fig. 3: Illustration of the dual-block representation

each vertex interval associates an edge block to store the out-edges of the vertices within the interval. Furthermore, each edge block is further divided into P sub-blocks according to their destination vertices. Inside each sub-block, edges are sorted by their source vertices. In this graph representation, the edges are partitioned into $P \times P$ sub-blocks. Each sub-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 sub-block and the corresponding vertices can fit in memory, the SSSB representation can improve the processing capacity for very large graphs and ensure good I/O access locality when processing each sub-block.

Figure 3 shows the SSSB representation of an example graph. The vertices are divided into two intervals (1, 5) and (6, 10), the edges are partitioned into four sub-blocks according to the two intervals. For example, the out-edge (1, 6) is assigned to sub-block (1, 2) since vertex 1 belongs to interval 1 and vertex 6 belongs to interval 2. When processing sub-block (1, 2), edges in sub-block (1, 2) and vertices values in interval 1 will be read and used to calculate new values for interval 2. Here, interval 1 is called the source interval as all source vertices reside in it and interval 2 is called the destination interval. In addition, SSSB representation also maintains the index to the edges for each vertex in each sub-block. We refer to index (i, j) as the vertex index of sub-block (i, j). This enables selective data access as shown in Section III-C.

Note that, some systems 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 SSSB representation. However, the SSSB representation is different from GridGraph's grid format from the following aspects. First, the SSSB representation sorts the out-edges by the source vertices, so that out-edges with the same source vertex are stored contiguously. This is beneficial to the compression of edges and efficient parallel processing. While GridGraph can not fully utilize the parallelism without sorted edges. Second, the SSSB representation creates a vertex index structure to enable the selective loading of edges.

C. Benefit-aware Sharing Execution Model

In order to overcome the challenges faced by existing out-of-core graph processing systems when handling CGP jobs as well as improve disk I/O performance, GraphCP adopts a benefit-aware sharing execution model. In this model, the graph data is decoupled as graph structure data (i.e., sub-block) and application-specific vertex attributes (i.e., PageRank values) like previous works [14], [16]. Graph structure data is shared by different CGP jobs. Each CGP job has its own vertex attributes that are repeatedly updated until the job converges. When processing CGP jobs, the sub-blocks are loaded into memory in sequence and in a common order for all CGP jobs, where each edge block is concurrently handled by the related CGP jobs. In this way, the accessing and storing of most edge blocks can be shared by the CGP jobs, which greatly reduces the redundant accesses and storage and avoids the competition of I/O bandwidth. During the processing, GraphCP dynamically schedules the loading of edges, i.e., skipping loading and processing inactive edges in each iteration whenever such skipping can bring performance benefit. In the benefit-aware sharing execution model, GraphCP processes the input graph one vertex interval at a time. For each vertex interval, GraphCP processes each sub-block at a time. The processing of each sub-block can be divided into two steps: benefit-aware sub-block loading, concurrent processing of the sub-block.

1) Benefit-aware Sub-block Loading: Current out-of-core graph processing systems [5], [6], [7] are usually optimized for the sequential performance of disk drives and eliminate random I/Os by scanning the entire graph data in all iterations of graph algorithms. However, for many graph algorithms (e.g., Breadth-first Search, Weak Connected Components, Single Source Shortest Path) that only access small portions of data during each iteration, this full I/O access model can be wasteful. For example, Breadth-first Search only visits vertices in a frontier in each iteration. For concurrent graph processing, there may exist some sub-blocks that have very few or no active edges for all CGP jobs. In this case, sequentially loading all sub-blocks will lead to suboptimal I/O performance. On the other hand, the on-demand I/O access 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 accesses to disk drives deliver much less bandwidth than sequential accesses. Therefore, only accessing the useful data for out-of-core graph processing is an overkill

when the number of active vertices is large. To address this dilemma and improve disk I/O performance, GraphCP adopts a benefit-aware scheduling scheme when loading each sub-block to skip loading and processing inactive edges in each iteration whenever such skipping can bring performance benefit.

GraphCP adaptively schedules the edge loading based on the number of active edges. When the number of active edges is small, the system only traverses the active edges to avoid the loading of useless data, which improves I/O efficiency. When the number of the active edges is large, the system loads the whole sub-block to eliminate random disk accesses. To achieve this, GraphCP incorporates the designs of bitmap operation and I/O-based benefit evaluation model.

Bitmap operation. Selective scheduling of the active edges needs to scan all vertices to identify the active vertices. To efficiently support concurrent graph processing, GraphCP maintains a job-specific state bitmap whose storage size is $|V|/8$ bytes for each CGP job to record whether a vertex is active or not for the job. In each iteration, GraphCP scans all job-specific state bitmaps and generates a shared state bitmap that records whether a vertex is active or not for any job in all CGP jobs. Specifically, a vertex is marked as active whenever it will be processed by any CGP job in current iteration. By identifying the number of active vertices of each sub-block, GraphCP can compute the I/O loads (active edges) when processing the sub-block and decides whether sequentially load the whole sub-block or only load the active edges of the sub-block.

I/O-based benefit evaluation model. To evaluate the performance benefit of loading the active edges, the key is to compare the I/O costs between sequentially loading all edges and randomly loading the active edges. 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*, *N*, *W* respectively be the size of an edge structure value, the size of a vertex value record and the size of an edge weight value. In addition, T_{rr}, T_{rw}, T_{sr} and T_{sw} represent random read, random write, sequential read and sequential write throughput (MB/s) respectively. For the ease of expression, we assume the number of vertices in each interval is equal to $|V|/P$.

For easy reference, we list the notations in Table I. When sequentially loading the whole sub-block, GraphCP loads all edges and vertex values of all CGP jobs into memory. In addition, only vertex values are updated since we store mutable data in vertices. Therefore, the I/O costs of GraphCP C_s when processing sub-block(i, j) can be stated constantly as:

$$C_s = \frac{\frac{|V|}{P} \times N \times J + S_{i,j}}{T_{sr}} + \frac{\frac{|V|}{P} \times N \times J}{T_{sw}}$$

When selectively (randomly) loading the active edges, we suppose that the active vertex set in current iteration is *A*, so the I/O amount of the active edges is equal to the size of all edges of vertices in *A*. Moreover, GraphCP also loads the vertex index so as to locate the active edges and compute the number of active edges, in addition to the vertex values. Therefore, the I/O cost C_r can be stated as:

TABLE I: Notations

Notation	Definition
G	the graph $G = (V, E)$
V	vertices in *G*
E	edges in *G*
P	number of intervals
J	number of CGP jobs
A	active vertex set in current iteration
M	size of an edge structure value
N	size of a vertex value
W	size of an edge weight value
$S_{i,j}$	size of an sub-block(i,j)
T_{rr}	random read throughput
T_{rw}	random write throughput
T_{sr}	sequential read throughput
T_{sw}	sequential write throughput

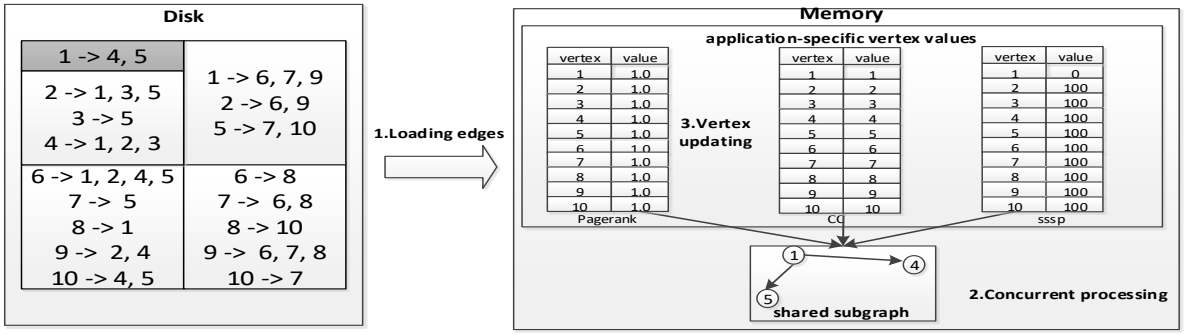
$$C_r = \frac{\sum_{v \in A}(index(i,j)[v+1] - index(i,j)[v]) \times (M+W)}{T_{rr}} + \frac{\frac{|V|}{P} \times N \times (J+1)}{T_{sr}} + \frac{\frac{|V|}{P} \times N \times J}{T_{sw}}$$

If $C_r \leq C_s$, the system selectively loads the active edges to avoid the loading of useless data. Otherwise, the system just loads all in-edges and out-edges to eliminate random disk accesses. The disk access throughput T_{rr}, T_{rw}, T_{sr} and T_{sw} can be measured by using several measurement tools such as fio [6] before we conduct the experiments. And other parameters such as A and $S_{i,j}$ can be directly collected and computed in the runtime. This provides an accurate performance prediction that enables efficient scheduling.

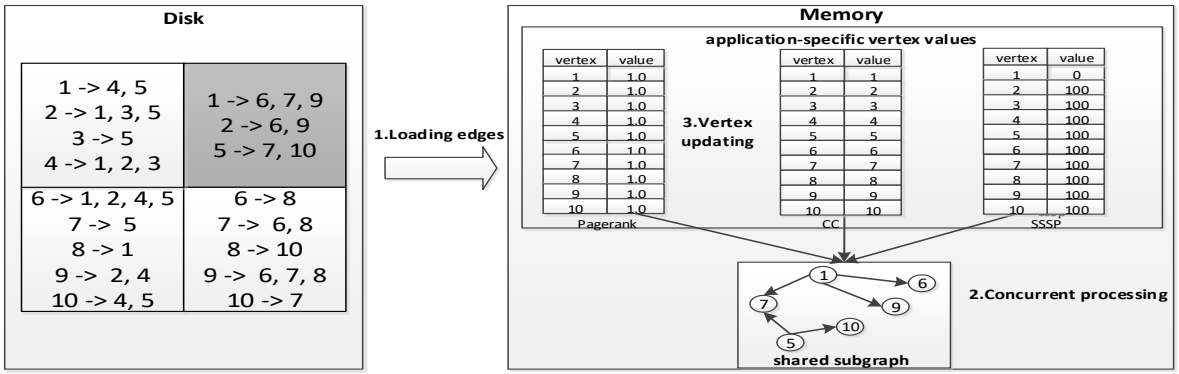
2) Concurrent Processing of the Sub-block: After loading a sub-block, the related CGP jobs that have unprocessed vertices and edges in the sub-block will concurrently access the sub-block and update their application-specific vertex values. When the processing of the sub-block is finished for all related jobs, the next sub-block then can be loaded. The CPU cores are assigned to the CGP jobs evenly when the processing starts. When the number of jobs is larger than the number of CPU cores, these CGP jobs are assigned to be processed as different batches. During the processing, there may exist some jobs that have fewer computation loads in the sub-block and finish more quickly than other jobs, leaving some worker threads idle. To tackle this problem, GraphCP adopts a work stealing strategy that enables the worker threads whose jobs have been finished to process the unfinished jobs (Section III-D).

For the processing of each sub-block, each CGP job accesses the edges in the sub-block and pushes updates from the source to the destination vertices with a user-defined update function. Once a CGP job has processed all sub-blocks in current iteration, it synchronizes its own vertex values with the latest updated values in the iteration.

3) Workflow Example: Figure 4 illustrates an example of the workflow of the benefit-aware sharing execution mod-



(a) Processing of Sub-block (1,1)



(b) Processing of Sub-block (1,2)

Fig. 4: Illustration of Benefit-aware Sharing Execution Model

el with the graph in Figure 3(a). In this example, system processes vertex interval 1 and its associated sub-blocks, i.e., sub-block (1, 1) and sub-block (1, 2). The graph needs to be handled by three CGP jobs, i.e., a PageRank job, a Connected Components (CC) job and a Single-Source Shortest Path (SSSP) job. Supposing there is only one active vertex 1 when processing sub-block (1, 1) (Figure 4(a)), only the out-edges of vertex 1 are loaded into memory. Then the CGP jobs concurrently access and process these edges as the shared subgraph. Each job updates its own application-specific vertex values using its own update function. After processing sub-block (1, 1), vertex 4 and 5 are activated since their values are updated. Therefore, there are 5 active edges (i.e., out-edges of vertex 1 and 5) when processing sub-block (1, 2) (Figure 4(b)). According to the I/O-based benefit evaluation model, the whole sub-block should be loaded into memory. After the three jobs finish the processing of sub-block (1, 2), system starts to process vertex interval 2 and their associated sub-blocks (i.e., sub-block (2, 1) and sub-block (2, 2)). When all sub-blocks are processed by all CGP jobs, the CGP jobs move to the next iteration of processing.

D. Work Stealing Strategy

Due to the different computation features of different graph applications, the computation loads (active edges) of each CGP job are usually skewed when processing each sub-block. In this case, some jobs may finish much more quickly than other jobs, which causes some worker threads idle after some jobs are finished and reduces the utilization ratio of hardware. For example, BFS may only need to process a few edges when processing a sub-block, while PageRank may have to go through all edges to complete the processing.

In order to address this load imbalance problem, GraphCP adopts a work stealing strategy that reassigns the threads whose jobs have been finished to process the unfinished jobs. To this end, GraphCP first calculates the computation loads of each job, which can be easily computed based on the number of newly activated vertices. Then, it identifies several ”straggler jobs” whose computation loads are larger than a user-defined threshold. For these jobs, the sub-blocks are logically divided to several shards according to the source vertices of the edges and are processed from the beginning to the end. For threads whose jobs have been finished (assisting threads), they will check the number of unprocessed shards for all straggler jobs and assist to process the straggler job with most number of unprocessed shards. When handling straggler jobs, the assisting threads process the unfinished shards from the last shard until all unfinished shards are processed.

E. Programming Model

We have fully implemented GraphCP in C++. The main execution procedure of GraphCP is described in Algorithm

1. When processing each vertex interval, GraphCP identifies the shared active vertices by merging the active vertices set of all CGP jobs in the interval (Line 8 $\sim$ 13). For the processing of each sub-block, GraphCP adaptively selects the I/O access model based on the I/O-based benefit evaluation model, to selectively load the active edges in the sub-block or sequentially load the whole sub-block (Line 15 $\sim$ 21). Then, each CGP job concurrently accesses the edges and executes the vertices updating for its own purposes (Line 22 $\sim$ 26).

Algorithm 1 Pseudo code of GraphCP execution

```
1: procedure Executor
2: for each CGP job j in J do
3:     A_j ← ActiveVerticesSet
4:     S_j ← VertexValues
5:     Out_j ← NewActiveVerticesSet
6: end for
7: for each interval i do
8:     for each j in J do
9:         /* Identify the active vertices in interval i*/
10:        A_j^i ← GetActiveVertices(i, S_j)
11:        /* Identify the shared active vertices set of all CGP
       jobs*/
12:        A_s ← ⋃ A_j^i
13:    end for
14:    for each sub − block in interval i do
15:        /* Select the I/O access model*/
16:        IOModel ← Selection(sub − block, A_s)
17:        if IOModel = selective then
18:            edges ← SelectiveLoad(sub − block, A_s)
19:        else
20:            edges ← SequentialLoad(sub − block)
21:        end if
22:        for each CGP job j in J do
23:            if A_j^i ← NotEmpty then
24:                ParallelUpdate(edges, j, A_j^i, S_j, Out_j)
25:            end if
26:        end for
27:    end for
28: end for
29: end procedure
```

In our programming model, only function *ParallelUpdate* is user-defined, while the others are provided by runtime. Users can modify the *ParallelUpdate* function to write their own graph algorithms for the CGP jobs. Algorithm 2 shows the implementation of *ParallelUpdate* with the example of Connected Components (CC) algorithm. In this algorithm, the vertex values and the active vertex set of the CC job are updated for the subsequence computation.

IV. Evaluation

In this section, we first introduce our evaluation environment and graph algorithms. Then, we compare GraphCP with state-of-art graph processing systems in terms of overall performance, I/O traffic and scalability. Finally, we evaluate the effects of different system optimizations.

Algorithm 2 $ParallelUpdate(edges, j, A_j^i, S_j, Out_j)$: CC

```
1: procedure CC
2: for each edge e in edges do
3:     if e.src ∈ A_j^i then
4:         if S_j(e.src) < S_j(e.dst) then
5:             S_j(e.dst) ← S_j(e.src)
6:             Out_j.add(e.dst)
7:         end if
8:     end if
9: end for
10: end procedure
```

TABLE II: Datasets used in evaluation

Dataset	Vertices	Edges
LiveJournal [25]	4.8 million	69 million
Twitter2010 [26]	42 million	1.5 billion
SK2005 [27]	51 million	1.9 billion
UK2007 [28]	106 million	3.7 billion
Kron30 [29]	1 billion	32 billion

A. Experiment Setup

The hardware platform used in our experiments is a commodity server equipped with two 8-core 2.10 GHz Intel Xeon CPU E5-2620, 16GB main memory and 600GB 7200RPM HDD, running Ubuntu 16.04 LTS. In addition, a 128GB SATA2 SSD is installed to evaluate the scalability.

We use different types of graphs for the evaluation as summarized in Table II. LiveJournal, Twitter2010 and SK2005 are social graphs, showing the relationship between users within each online social network. UK2007 is a web graph that consists of hyperlink relationships between web pages. Kron30 is a synthetic graph generated with the Graph500 generator [29]. The small graph LiveJournal is chosen to evaluate the in-memory processing performance of GraphCP. The other four graphs are respectively larger than memory capacity by 1.6x, 1.9x, 3.9x and 16.0x.

Note that, although our hardware platform is not a very powerful platform, it is sufficient to show the problem we focus on and evaluate the efficiency and benefit of our system, since most graphs used in the evaluation are larger than the memory capacity. When deploying our system in a more powerful platform and using much larger graphs for evaluation, the problem still remains as the large graphs may not fit in memory, and our evaluation results will not be changed.

We run four graph algorithms as concurrent jobs: PageRank (PR), Breadth-first search (BFS), Weak Connected Components (WCC), and Single Source Shortest Path (SSSP). These algorithms exhibit different I/O access and computation characteristics, which provides a comprehensive evaluation of GraphCP. For PageRank, we run five iterations on each graph. For BFS, WCC and SSSP, we run them until convergence.

TABLE III: Execution time (in seconds)

System	GridGraph	GraphZ	Seraph	GraphCP
liveJournal	16.9	9.4	9.6	6.3
Twitter2010	3115.4	1639.6	328.2	205.1
SK2005	3807.1	1586.3	768.4	349.3
UK2007	6911.2	2303.7	1573.3	561.9
Kron30	-	-	130944.5	56932.4

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

We compare GraphCP with three baseline systems. Two of them are state-of-art out-of-core graph processing systems, GridGraph [7] and GraphZ [19]. The other is Seraph [14], [15], which is a state-of-art graph processing system optimized for efficient execution of CGP jobs, implemented by us on GridGraph.

B. Overall Performance

To compare the overall performance, we run all the CGP jobs (PageRank, BFS, WCC, and SSSP) simultaneously for each of these four systems. Note that, for GraphCP and Seraph, we only need to run a single GraphCP and Seraph instance to execute these CGP jobs thanks to their sharing access of the graph. While for GridGraph and GraphZ, each CGP job needs to initiate a GridGraph or GraphZ instance for parallel execution. Table III shows the total execution time of the CGP jobs for different systems. We can see that GraphCP finishes the CGP jobs much more quickly than other three systems. On average, GraphCP outperforms GridGraph, GraphZ and Seraph by 10.3x, 4.6x and 2.1x respectively.

GridGraph uses a 2-Level hierarchical partition scheme and a streaming-apply model to reduce the amount of data transfer, enable streamlined disk access, and maintain locality. However, it is optimized for the processing of individual graph processing job. When handling CGP jobs, it exhibits poor performance due to the reasons shown in Section II-A. GraphZ can greatly reduce the I/O costs and improve performance by using the degree-ordered storage and dynamic messages. Unfortunately, it also suffers from the problems as other out-of-core graph processing systems when processing concurrent jobs. While for GraphCP, it shares the accesses of sub-blocks among the CGP jobs with the benefit-aware sharing execution model, which leads to high performance when executing the CGP jobs. Note that, when executing the CGP jobs on LiveJournal, GraphCP only outperforms GridGraph and GraphZ by 2.7x and 1.5x. This is because GridGraph and GraphZ can avoid the I/O conflicts when processing the in-memory graph.

For Seraph, although it is also able to spare the data accesses via shared in-memory graph structure, it has not efficiently scheduled the computation loads of different CGP jobs, which causes load imbalance and significant synchronization overheads for the iterative processing. GraphCP alleviates these issues by adopting a work stealing strategy. Moreover, GraphCP dynamically schedules the loading of edges, which further improves the I/O performance.

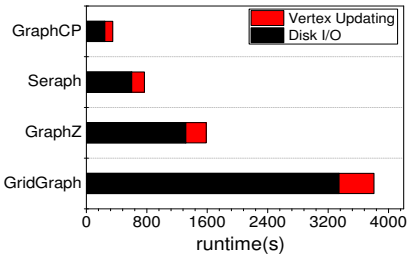


Fig. 5: Runtime breakdown on SK2005

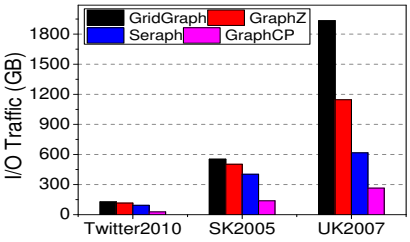


Fig. 6: I/O traffic comparison

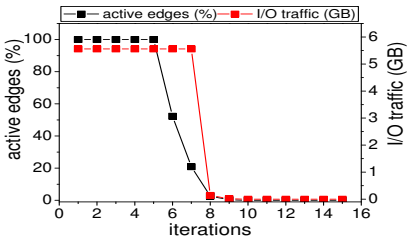


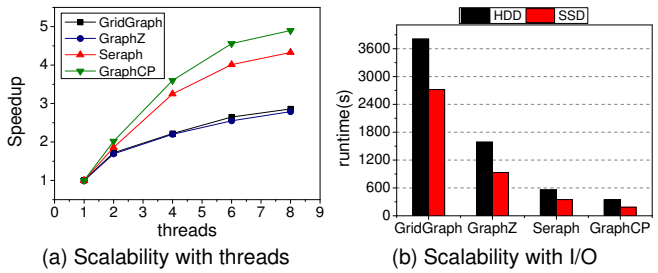
Fig. 7: I/O traffic per iteration

To further demonstrate the efficiency of GraphCP, we also report the runtime breakdowns of the executions on SK2005 for all compared systems, as shown in Figure 5. From the results, we can see that GraphCP outperforms other systems in both disk I/O performance and computational performance.

C. I/O Traffic

Then we compare the total volume of I/O traffic for different systems. Figure 6 shows the results. From the results, the volume of I/O traffic of GraphCP is 5.3x, 4.1x and 2.7x less than that of GridGraph, GraphZ and Seraph respectively. This is mainly attributed to GraphCP's benefit-aware loading scheme that avoids loading the useless data when there are very few active edges for the CGP jobs to access. On the other hand, GridGraph and GraphZ load more data than Seraph. This is because they suffer from many repeated data accesses due to ignoring the data access correlations among the CGP jobs. While Seraph can share the accesses among the CGP jobs.

To further demonstrate GraphCP can adaptively schedule the I/O access according to the number of active edges, we report the I/O traffic and percentage of active vertices in each iteration on Twitter, as shown in Figure 7. When the number of active edges is large, GraphCP traverses all graph data for sequential access performance, so the amount of I/O traffic is large. When the number of active edges decreases to the



(a) Scalability with threads (b) Scalability with I/O

Fig. 8: Evaluation of scalability

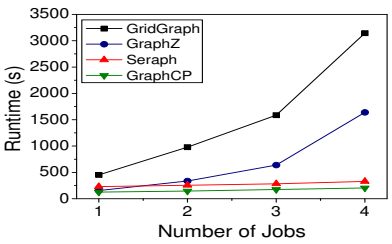
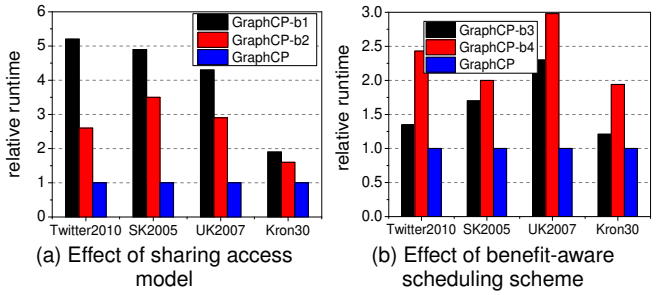
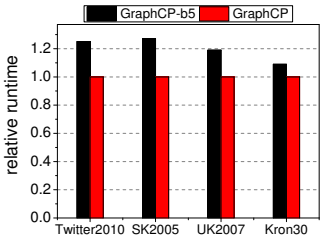


Fig. 9: Performance variation with different number of jobs



(a) Effect of sharing access model (b) Effect of benefit-aware scheduling scheme



(c) Effect of work-stealing strategy

Fig. 10: Evaluation of different system optimizations

extent that selectively accessing the active edges outperforms sequentially accessing all edges, GraphCP only loads the active edges.

D. Scalability

The scalability of GraphCP is evaluated by observing the performance improvement when more hardware resource is added. Figure 8(a) shows the effect of the thread number on execution time when executing CGP jobs on LiveJournal. We observe that GraphCP and Seraph achieve better scalability than GridGraph and GraphZ. This stems from efficient share of data accesses. Figure 8(b) shows the performance improvement of CGP jobs on SK2005 when using different I/O devices. Compared with disk performance, GridGraph, GraphZ, Seraph and GraphCP respectively achieve a speedup of 1.4x, 1.6x, 1.6x and 1.9x when using SSD. This indicates that GraphCP can benefit more from the utilization of SSD, since GraphCP enables selective (random) data access to load the active edges, which works well on SSD.

We also evaluate the performance variation when increasing the number of CGP jobs, as shown in Figure 9. The number of CGP jobs is increased in the order of PageRank, BFS, WCC and SSSP. When the number of CGP jobs is increased from 1 to 4, the execution time of GridGraph, GraphZ, Seraph and GraphCP increases by 6.9x, 10.6x, 1.4x and 1.6x respectively. Obviously, Seraph and GraphCP achieve a better scalability than the other two systems. The worse scalability of GridGraph and GraphZ is attributed to the severe competition for the limited I/O bandwidth, which significantly reduces the system throughput and performance.

E. Effects of System Optimizations

To analyze the performance gains obtained by using different system optimizations, we compare the system performance by disabling each optimization to GraphCP. The system optimizations include the sharing access model, benefit-aware scheduling scheme and working stealing strategy.

We first evaluate the effect of the sharing access model. We compare GraphCP with two baseline implementations. The first baseline implementation (GraphCP-b1) handles the CGP jobs by initiating the I/O request of each job to the disk separately like current out-of-core systems. The second baseline implementation (GraphCP-b2) executes the CGP jobs by running one after another. As shown in Figure 10(a), thanks to the efficient sharing accesses, the performance of GraphCP can be accelerated by up to 5.2x. On the other hand, GraphCP-b1 has a worse performance than GraphCP-b2 even though it enables parallel execution. This is because GraphCP-b1 incurs severe competitions for I/O bandwidth, which greatly reduces system performance.

Then we evaluate the effect of the benefit-aware scheduling scheme as shown in Figure 10(b). There are also two baseline implementations. The first baseline implementation (GraphCP-b3) sequentially loads the whole sub-block into memory when processing each sub-block. The second baseline implementation (GraphCP-b4) selectively loads the active edges when processing each sub-block. The results show that the benefit-aware scheduling scheme can improve the performance by up to 3.0x, due to adaptively scheduling the edge loading according to the number of active edges.

Finally, we evaluate the effect of the work stealing strategy by comparing with the baseline implementation that disables the work stealing strategy (GraphCP-b5). Through achieving better load balance among CGP jobs, the work stealing strategy can improve the performance by up to 1.27x, as shown in Figure 10(c). We can find that the work stealing strategy brings fewer performance improvements than the other two

optimizations. This is because the vertex updating phase has relatively less impact on the overall performance for an out-of-core system, compared with the disk I/O phase.

V. Conclusion

In this paper, we present a new out-of-core graph processing system called GraphCP that aims to efficiently handle concurrent graph processing jobs. GraphCP proposes a benefit-aware sharing execution model that shares the accesses of graph data among the CGP jobs and enables selective disk I/O accesses. Moreover, GraphCP adopts a Source-Sorted Sub-Block graph representation to improve processing capacity and ensure I/O access locality. Our evaluation results show that GraphCP can be much faster than GridGraph, GraphZ and Seraph, three state-of-the-art graph processing systems. In future works, we will research how to extend our system to process evolving graphs whose vertices and edges are constantly changing. In addition, we will exploit emerging hardware such as GPU, FPGA, NVM to accelerate data accesses of concurrent jobs for higher throughput.

Acknowledgment

This work was supported by NSFC No. 61832020, 61772216, 61821003, and U1705261, Wuhan application basic research Project No. 2017010201010103, Hubei province technical innovation special Project No. 2017AAA129, National Defense Preliminary Research Project No. 31511010202,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. This work is also supported by the Natural Science Foundation of Fujian Province under Grant No.2020J01493.

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[2] 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," *PVLDB*, pp. 716–727, 2012.

[3] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, 2012, pp. 17–30.

[4] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *12th USENIX Symposium on Operating Systems Design and Implementation*, 2016, pp. 301–316.

[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a pc," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2012, pp. 31–46.

[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 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 *2015 USENIX Annual Technical Conference*, 2015, pp. 375–386.

[8] K. Vora, "Lumos: Dependency-driven disk-based graph processing," in *2019 USENIX Annual Technical Conference*, 2019, pp. 429–442.

[9] X. Xu, F. Wang, H. Jiang, Y. Cheng, D. Feng, and Y. Zhang, "A hybrid update strategy for i/o-efficient out-of-core graph processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 8, pp. 1767–1782, 2020.

[10] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "Powerlyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Transactions on Parallel Computing (TOPC)*, vol. 5, no. 3, pp. 1–39, 2019.

[11] Y. Cheng, F. Wang, H. Jiang, Y. Hua, D. Feng, and X. Wang, "Lcc-graph: A high-performance graph-processing framework with low communication costs," in *IWQoS'16*. IEEE, 2016, pp. 1–10.

[12] http://www.facebook.com/, 2020.

[13] M. Han and K. Daudjee, "Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 950–961, 2015.

[14] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: an efficient, low-cost system for concurrent graph processing," in *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, 2014, pp. 227–238.

[15] J. Xue, Z. Yang, S. Hou, and Y. Dai, "Processing concurrent graph analytics with decoupled computation model," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 876–890, 2017.

[16] Y. Zhang, X. Liao, H. Jin, L. Gu, L. He, B. He, and H. Liu, "Cgraph: A correlations-aware approach for efficient concurrent iterative graph processing," in *2018 USENIX Annual Technical Conference*, 2018, pp. 441–452.

[17] J. Zhao, Y. Zhang, X. Liao, L. He, B. He, H. Jin, H. Liu, and Y. Chen, "Graphm: an efficient storage system for high throughput of concurrent graph processing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[18] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic i/o optimization for disk-based graph processing," in *2016 USENIX Annual Technical Conference*, 2016, pp. 507–522.

[19] Z. Zhou and H. Hoffmann, "Graphz: Improving the performance of large-scale graph analytics on small-scale machines," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 2018, pp. 1368–1371.

[20] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o," in *2017 USENIX Annual Technical Conference*, 2017, pp. 125–137.

[21] P. Pan and C. Li, "Congra: Towards efficient processing of concurrent graph queries on shared-memory machines," in *2017 IEEE International Conference on Computer Design (ICCD)*. IEEE, 2017, pp. 217–224.

[22] L. Zhou, R. Chen, Y. Xia, and R. Teodorescu, "C-graph: A highly efficient concurrent graph reachability query framework," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[23] 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 *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 2015, pp. 1131–1142.

[24] 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 *Proceedings of 13th USENIX Conference on File and Storage Technologies*, 2015, pp. 45–58.

[25] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 44–54.

[26] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*. ACM, 2010, pp. 591–600.

[27] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 595–602.

[28] P. Boldi, M. Santini, and S. Vigna, "A large time-aware web graph," in *ACM SIGIR Forum*, vol. 42, no. 2. ACM, 2008, pp. 33–38.

[29] http://www.graph500.org/, 2020.