Front. Comput. Sci., 2024, 18(3): 183105
https://doi.org/10.1007/s11704-023-2361-0

RESEARCH ARTICLE

A disk I/O optimized system for concurrent graph processing jobs

Xianghao XU[1,2], Fang WANG[2], Hong JIANG[3], Yongli CHENG (✉)[4,5], Dan FENG[2], Peng FANG[2]

1 School of Computer Science and Engineering, Nanjing University of Science and Technology, Nanjing 210094, China
2 Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China
3 Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, TX 76019, USA
4 College of Computer and Data Science, Fuzhou University, Fuzhou 350108, China
5 Zhejiang Lab, Hangzhou 311121, China

Abstract 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 unnecessary data accesses and severe competition of I/O bandwidth when handling the CGP jobs. In this paper, we propose GraphCP, a disk I/O optimized out-of-core graph processing system that efficiently supports the processing of CGP jobs. GraphCP proposes a benefit-aware sharing execution model to share the I/O access and processing of graph data among the CGP jobs and adaptively schedule the graph data loading based on the states of vertices, which efficiently overcomes above challenges faced by existing out-of-core graph processing systems. Moreover, GraphCP adopts a dependency-based future-vertex updating model so as to reduce disk I/Os in the future iterations. 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 20.5× and 8.9× faster than two out-of-core graph processing systems GridGraph and GraphZ, and 3.5× and 1.7× faster than two state-of-art concurrent graph processing systems Seraph and GraphSO.

Keywords graph processing, disk I/O, concurrent jobs

1 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] or 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] and LUMOS [8]) 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 [9] and significant communication overheads [10]. 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 uses Apache Giraph 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 [11]. Several concurrent graph processing systems such as Seraph [12,13] are proposed to support the execution of CGP jobs based on distributed systems. Unfortunately, when deploying these CGP jobs on out-of-core graph processing systems, it will incur significant performance degradation. As the CGP jobs iteratively traverse the graph along different paths for their own purposes, there are a large

Received June 16, 2022; accepted February 7, 2023

E-mail: chengyongli@fzu.edu.cn

number of intersections among the graph data being accessed by these jobs in each iteration, which produces redundant disk I/Os and memory storage overheads. Moreover, 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. Recent concurrent graph processing systems like CGraph [11] and GraphM [14] can solve above problems and support disk-based processing. However, these systems suffer from suboptimal disk I/O performance as a result of the following reasons. First, they usually incur many unnecessary data accesses due to loading the inactive edges. Second, they do not support future-value or cross-iteration computation, and only allow each edge/vertex to be processed once in each iteration, leaving significant room to further reduce disk I/Os. In addition, existing concurrent graph processing systems 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 a disk I/O optimized out-of-core graph processing system called GraphCP to efficiently handle the CGP jobs. The main contributions of GraphCP are summarized as follows.

- **Benefit-aware sharing execution model** 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. Moreover, it 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 support such effective scheduling, an I/O cost based benefit evaluation model is proposed to guide the system to select the optimal I/O access model with little extra computation overheads.
- **Dependency-based future-vertex updating model** To further reduce disk I/O costs, GraphCP adopts a dependency-based future-vertex updating (DFVU) model that exploits dependencies among vertices to enable future-vertex value updating and reduce disk I/Os in the future iterations. Compared with our previous work [15], the DFVU model can improve overall performance and reduce disk I/Os by 1.6× and 1.5× respectively.
- **Source-Sorted Sub-Block graph representation** GraphCP uses 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.
- **Extensive experiments** We evaluate the performance of GraphCP by comparing with state-of-art graph processing systems including GridGraph, GraphZ, Seraph and GraphSO. The evaluation results show that GraphCP outperforms them by 20.5×, 8.9×, 3.5× and 1.7× on average thanks to significant improvement of I/O performance.

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

2 Background and motivation

A graph problem is usually encoded as a graph $G = (V, E)$, where V and E are the set of vertices and edges respectively. For an 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 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.

2.1 Out-of-core graph processing

Out-of-core (disk-based) graph processing systems are designed to enable users to process large graphs that cannot fit in the DRAM of a machine. When processing a graph, these systems divide the vertex set of the graph into disjoint intervals and create an edge block for each interval which contains the in-edges or out-edges of the vertices in the interval. The partition of the vertices and edges should ensure that the edge block of each vertex interval fit in the available memory. Figure 1 illustrates the graph data organization of an out-of-core graph processing system. In this example, the vertices of the example graph are divided into four intervals (1, 2), (3, 5), (6, 8) and (9, 10), and the edges are partitioned into four sub-blocks according to the four intervals. During the processing of the graph, the system successively loads and processes each vertex interval and the corresponding edge block from disk. Through a disk-friendly graph data organization format and well-designed execution engine, existing out-of-core graph processing systems can significantly reduce random disk accesses, even achieving competitive performance with distributed graph processing systems [5,7,16].

2.2 Concurrent graph processing

Since graph processing becomes more and more popular in the real-world, the need of simultaneously handling many graph processing jobs is increasingly growing. These concurrent graph processing (CGP) jobs run many graph applications on the same graph so as to meet the requirements of users. For example, Taobao handles billions of users' queries on its large commercial product graph every day for product recommendation and personalized search [17]. In addition, many graph applications are essentially composed of concurrent jobs, such as multi-source shortest paths [18]. Therefore, CGP attracts considerable research interests in recent years, and several concurrent graph processing systems [11,12,14] are proposed. These concurrent graph processing systems usually decouple the graph structure data from the

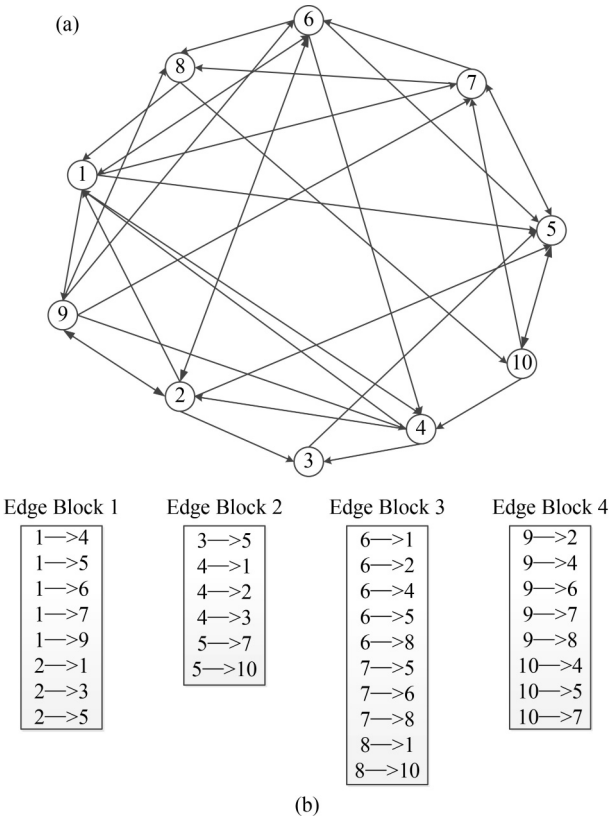


Fig. 1 Example graph and the partitioned four edge blocks that reside on disk. (a) An example graph; (b) Edge blocks

CGP jobs and only store one copy of graph structure data in memory, while each CGP job maintains its own application-specific vertex attributes. In this way, multiple CGP jobs can share the access of the same graph structure data, significantly reducing the redundant data accesses and storage costs.

2.3 Motivation

Although existing concurrent graph processing systems can efficiently handle multiple CGP jobs, they are usually deployed on a distributed or shared-memory system, which suffers from the problems such as significant communication overheads and poor scalability. The enormous amount of intermediate update messages produced by the CGP jobs significantly exacerbate these problems. Several works like CGraph [11] and GraphM [14] can support processing CGP jobs from secondary storage. However, these systems usually suffer from the following challenges that hinder them to achieve optimal disk I/O performance.

Inactive data access. During the processing, there may exist some edge blocks containing very few active edges. Unfortunately, to purse the sequential disk bandwidth, existing systems like CGraph and GraphM still load the whole edge blocks even though the number of active edges in the edge blocks is very small, leading to massive unnecessary I/Os. Recent work GraphSO [19] solves this problem by using a repartition-centric model that only loads the active chunks (i.e., containing active edges) and constructs new logical partitions with these chunks. However, it can incur extra overheads for the repartition procedure.

No support for future-value updating. In order to reduce disk I/Os, several out-of-core graph processing systems [8,16] enable future-value or cross-iteration computation. In other words, these systems perform multi-iteration of computation in one round of graph loading, which speeds up the convergence of graph algorithms and reduces disk I/Os. Unfortunately, existing concurrent graph processing systems do not support this updating model and allow each edge/vertex to be processed only once in each iteration, leaving significant room to further reduce disk I/Os.

Limited processing capacity and scalability. Existing concurrent graph processing systems have to maintain many copies of vertex values for different CGP jobs in memory. This will consume enormous memory footprints and limit the processing capacity and scalability of the system when the graph is very large. Moreover, if the vertex values cannot fit in memory, the system will incur a large amount of random disk accesses that greatly degrade the overall performance.

The above challenges motivate us to propose a disk I/O optimized (out-of-core) concurrent graph processing with the following design principles. First, the system should avoid loading inactive edges in each iteration to reduce unnecessary disk accesses. Second, the system should enable future-vertex value updating to further reduce disk I/O costs. Third, the system should adopt efficient graph representation to improve the processing capacity and scalability when the graph is large.

3 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, benefit-aware sharing execution model and dependency-based future-vertex updating model. Next, we illustrate the main workflow of GraphCP with an example. Finally, we present the programming model of GraphCP.

3.1 System overview

Figure 2 presents the architecture overview of GraphCP. For graph preprocessing, GraphCP uses a Source-Sorted Sub-Block graph representation that adopts a 2-dimensional partitioning method to partition the input graph into several sub-blocks. This graph representation can improve the processing capacity and scalability as well as ensure the I/O access locality.

After preprocessing, the graph data needs to be loaded into the memory to serve the execution of concurrent jobs. To improve the disk I/O efficiency, GraphCP adopts 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 this way, the CGP jobs concurrently access the shared graph to update the vertex values for their own purposes, which solves the redundant accesses and I/O competition problem. Moreover, it adaptively schedules the loading of graph data based on the states of vertices (active or not) to skip the I/O accesses of inactive data. An I/O cost based benefit evaluation model is proposed to guide the system to select the optimal I/O access model with little overheads.

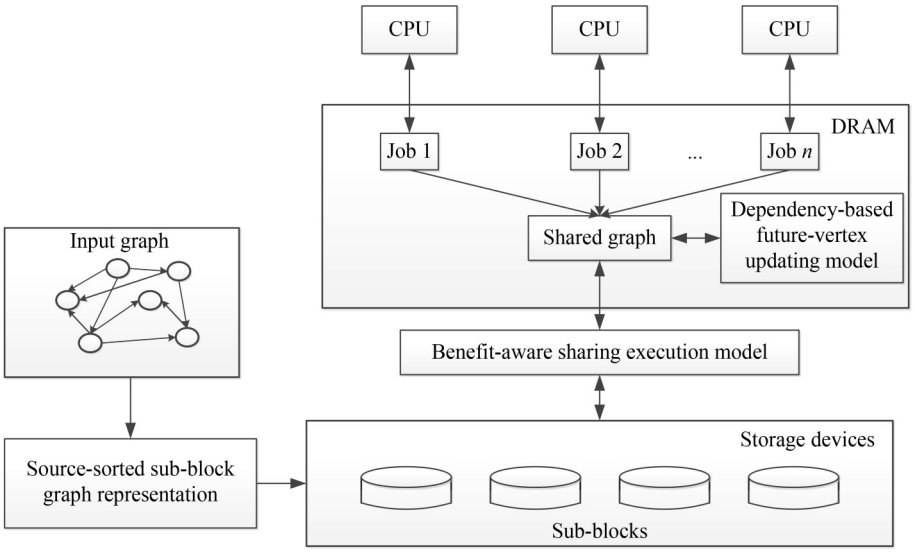


Fig. 2 The GraphCP architecture

In addition, during the vertex updating phase, GraphCP proposes a dependency-based future-vertex updating model that exploits the dependencies among vertices to enable cross-iteration vertex updating based on the loaded edges, so as to reduce disk I/Os in the future iterations.

3.2 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, 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.

Note that, due to the irregularity and skewed degree distributions of the real-world graphs, the SSSB representation does not necessarily guarantee the number of edges in each sub-block balanced. Fortunately, since GraphCP processes each sub-block at a time instead of processing many sub-blocks in parallel, this possible imbalance among sub-blocks may not affect the overall performance. In addition, it requires at least two times of scanning of the graph to generate the balanced 2-D sub-blocks [20], incurring much more preprocessing overheads.

Fig. 3 shows the SSSB representation of the example graph in Fig. 1(a). 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 to enable selective data access. We refer to index (i, j) as the vertex index of sub-block (i, j).

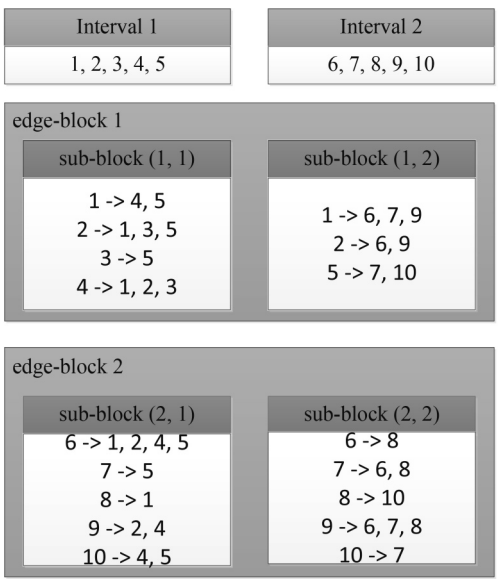


Fig. 3 Illustration of the Source-Sorted Sub-Block representation

3.3 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 concurrent graph processing systems [11,12]. The 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 sub-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 graph data loading, GraphCP dynamically schedules the loading of edges based on the number of active vertices. 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.

3.3.1 Benefit-aware sub-block loading

Current out-of-core graph processing systems [5–7] 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 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. 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, which adaptively schedules the edge loading based on the states of vertices and the number of active edges. When the number of the active edges is small, the system selectively loads the active edges in each sub-block to avoid the loading of useless (inactive) 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 accurately switch between these two kinds of I/O access models, GraphCP adopts an I/O-based benefit evaluation model (Section 3.3.2) that guides the system to skip loading and processing inactive edges in each iteration whenever such skipping can bring performance benefit.

3.3.2 I/O cost based benefit evaluation model

In order to accurately switch and select between sequentially loading the whole sub-block and selectively loading the active edges in each sub-block, the system should evaluate the performance benefits of these two kinds of I/O access models and choose the best one. To this end, GraphCP proposes an I/O cost based benefit evaluation model. In this model, The performance benefit is estimated based on the I/O cost of each I/O access model, which can be calculated by the total size of data accessed divided by the bandwidth of disk access. Let M, N, W respectively be the size of an edge structure, the size of a vertex value record and the size of an edge weight value. B_{rr}, B_{rw}, B_{sr} and B_{sw} represent random read, random write, sequential read and sequential write bandwidth respectively. For easy reference, we list the notations in Table 1.

When sequentially loading the whole sub-blocks, 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 executing an iteration can be stated as:

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

When selectively (randomly) loading the active edges, we suppose that the active vertex set in the current iteration is A. Note that reading the active edges is not totally random. There may exist vertices with large degrees or vertices with contiguous IDs. The reading of their edge lists can be sequential. Therefore, we should estimate the sizes of edge lists that are sequentially and randomly read respectively, which are stated as S_{seq} and S_{ran}. Algorithm 1 shows the procedure of calculating S_{seq} and S_{ran}. Specifically, GraphCP successively examines the active vertices in A. When handling the first active vertex, GraphCP checks if the size of the edge list exceeds the I/O block size B to decide whether sequentially load the edge list (Line 2−8). When handling other vertices in A, GraphCP first checks if vertex v has the contiguous vertex ID with the previous active vertex ($v0$) (Line 11). If so, GraphCP can load the edge lists of v and $v0$ together (Line 12−17). The merged edge list can be loaded sequentially if the size of the merged edge list exceeds B. If v has no contiguous ID with the previous active vertex, GraphCP checks the size of its edge list like the first vertex in A to decide how to load the edge list (Line 25−30).

As we can see, the time complexity of Algorithm 1 is $O(|A|)$, which is acceptable considering the number of active vertices. After calculating S_{seq} and S_{ran}, the I/O cost C_r when randomly loading the active edges in an iteration can be stated

Table 1 Notations

Notation	Definition
V	The vertex set
E	The edge set
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_{seq}	Size of edges that are sequentially read
S_{ran}	Size of edges that are randomly read
T_{rr}	Random read bandwidth
T_{rw}	Random write bandwidth
T_{sr}	Sequential read bandwidth
T_{sw}	Sequential write bandwidth

Algorithm 1 Calculating S_{seq} and S_{ran}

Input: A

Output: S_{seq}, S_{ran}

```
for each vertex v in A do
    if v is the first vertex in A then
        if size(edgelist(v)) > B then
            S_seq+ = size(edgelist(v))
            L(v) = seq. // sequentially load the edge list
        else
            L+ = size(edgelist(v))
        end if
        v0 = v
    else
        if v.id == v0.id+1 then // contiguous vertex IDs
            if L(v0) == seq. then
                S_seq+ = size(edgelist(v))
                L(v) = seq.
            else
                L+ = size(edgelist(v))
            end if
        else
            if L > B then
                S_seq+ = L
            else
                S_ran+ = L
            end if
            L = 0
            if size(edgelist(v)) > B then
                S_seq+ = size(edgelist(v))
                L(v) = seq.
            else
                L+ = size(edgelist(v))
            end if
        end if
        v0 = v
    end if
end for
```

as (the vertex index is also loaded to locate the active edges and compute the number of active edges):

$$C_r = \frac{S_{ran}}{B_{rr}} + \frac{S_{seq}}{B_{sr}} + \frac{|V| \times N \times (J+1)}{B_{sr}} + \frac{|V| \times N \times J}{B_{sw}}.$$

If $C_r \leqslant C_s$, GraphCP selectively loads the active edges to avoid the loading of useless data. Otherwise, GraphCP loads the whole sub-blocks to eliminate random disk accesses. Before executing graph algorithms on GraphCP, several parameters should be determined for the I/O cost based benefit evaluation model. Specifically, the disk access bandwidths B_{rr}, B_{rw}, B_{sr} and B_{sw} can be measured by using several measurement tools such as fio [6]. The vertex size N, edge size M, edge weight W and the number of jobs J are specified by the users according to different applications or workloads. Other parameters such as A, S_{seq} and S_{ran} can be directly collected and computed in the runtime. Based on these parameters, this performance benefit evaluation method can provide an accurate performance prediction that enables efficient scheduling of edges.

Furthermore, to efficiently identify the active vertices and support the I/O cost based benefit evaluation model, 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 the current iteration. By identifying the number of active vertices of each sub-block, GraphCP can compute the I/O loads (number of active edges) when processing the sub-block so as to enable I/O costs estimation.

3.3.3 Efficient parallel 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. Specifically, each CGP job accesses their active edges in the sub-block and updates the vertices with a user-defined update function. When the processing of the sub-block is finished for all related jobs, the next sub-block then can be loaded. When all sub-blocks are processed by all CGP jobs, GraphCP moves to the next iteration of processing.

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. 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. To tackle this load imbalance problem, GraphCP efficiently allocates and schedules the computation resources for the CGP jobs. Specifically, when the processing starts, GraphCP assigns the worker threads to each CGP job based on its computation load. The computation load is proportional to the number of active edges of each CGP job in a sub-block. If the number of CGP jobs is larger than the number of worker threads, these CGP jobs are assigned to be processed as different batches with different computation loads. Moreover, GraphCP supports the work stealing scheme [21,22] that enables the worker threads whose jobs have been finished to process the unfinished jobs.

3.4 Dependency-based future-vertex updating

For most graph processing systems, they follow the processing semantics of the Bulk Synchronous Parallel (BSP) model [23]. In this model, the value of a vertex in the current iteration is computed based on the values of its neighbors in the previous iteration. Therefore, this model specifies the dependencies between vertices in graph computing. Specifically, for an edge $e = (u, v)$, the value of v in iteration t is dependent on the value of u in iteration $t - 1$. Such clear separation between values being generated and values being used can ensure the consistency and correctness of algorithm executions. In fact, the dependencies among vertices provide an opportunity to proactively compute the vertex values in the future iterations by making full use of the loaded edge blocks, so as to avoid

loading the corresponding graph portions in the future iterations. For instance, after vertex u is updated in iteration t, we can use u's latest value to update the values of u's neighbors in iteration $t+1$ based on u's edges that are loaded into memory. Therefore, the disk reads of u's edges can be avoided when executing iteration $t+1$. Unfortunately, existing concurrent graph processing systems have not exploited this future dependency, which allows each edge/vertex to be processed only once in each iteration.

Motivated by the above idea, GraphCP adopts a dependency-based future-vertex updating (DFVU) model to reduce disk I/Os in the future iterations. To enable future-vertex updating, DFVU should identify the vertices that satisfy the following two conditions. First, they have been updated in the current iteration. Second, their edges should be loaded into memory. In this way, the vertices can exploit the dependencies among vertices to update the values of their neighbors in the next iteration with their own values in the current iteration, according to the BSP model. We explain how the DFVU model performs future-vertex value updating for different I/O access models as follows.

When only loading the active edges, the DFVU model is executed after all active edges have been processed. Specifically, for each CGP job, DFVU is triggered for the vertices that are updated or activated in the current iteration. If the edges of these vertices are already loaded into memory (i.e., these vertices are also active in the current iteration), DFVU can directly update their neighbors' values in the next iteration in advance based on their edges. Consequently, these vertices are removed from the frontier and their edges need not to be loaded in the next iteration.

When loading the whole sub-block, since all edges are loaded into memory, the vertices in the sub-block can enable future-vertex value updating only when they have been already updated in the current iteration before processing this sub-block. Thanks to the SSSB representation, GraphCP can easily capture the sub-blocks that satisfy the above condition. Specifically, for a sub-block (i, j), it can enable future-vertex value updating if $i < j$, since vertices in interval i are already updated before updating vertex values of interval j. In addition, for sub-block (i, i) whose source interval and destination interval are identical, it can also perform future-vertex value updating after all vertices in interval i are updated. GraphCP holds sub-block (i, i) in memory until all vertex updating for interval i have finished. Therefore, to execute future-vertex value updating for sub-block (i, j), GraphCP first checks if $i \leqslant j$. If so, DFVU is triggered for all vertices in the sub-block. In the next iteration, GraphCP only needs to load the sub-blocks whose source intervals are behind the destination intervals (i.e., $i > j$) since they do not perform future-vertex value updating in the previous iteration.

3.5 Workflow example

To better illustrate the benefit-aware sharing execution model and dependency-based future-vertex updating model, we use an example to show the main execution flow of GraphCP with the graph in Fig. 3, as shown in Figs. 4 and 5. 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.

Figure 4 shows one iteration of execution when the number of active vertices is small. Supposing there is only two active

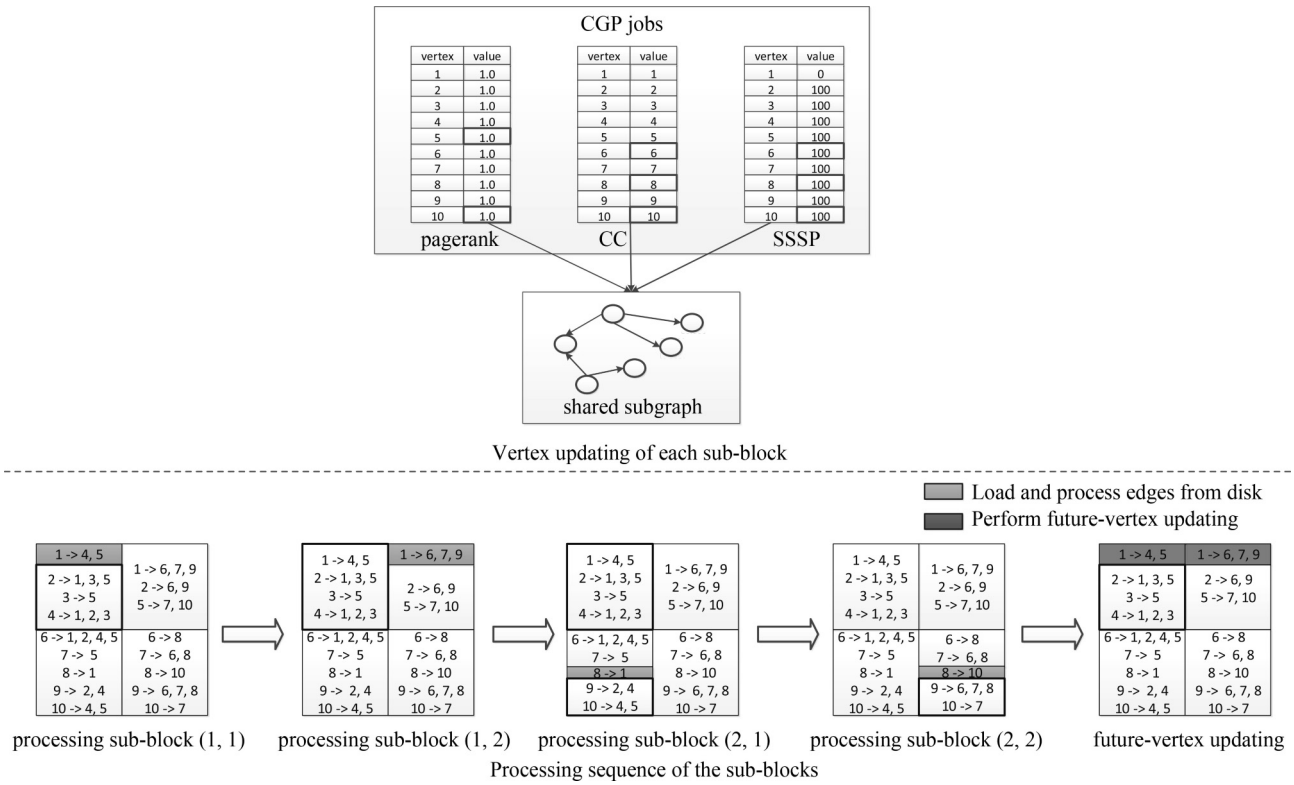


Fig. 4 One iteration of execution for GraphCP when the number of active vertices is small

vertices, i.e., vertex 1 and 8, in the current iteration. Based on the I/O cost based benefit evaluation model, only active edges need to be loaded. Specifically, GraphCP loads the edges of vertex 1 and 8 and generate a shared subgraph in memory for each sub-block (from sub-block (1, 1) to sub-block (2, 2)). During the vertex updating phase of each sub-block, the CGP jobs concurrently access the shared subgraph, and updates their own application-specific vertex values using their own update function. After the processing of this iteration is finished, vertex 1 is activated again and its edges are already loaded into memory. Therefore, the DFVU model is triggered for vertex 1 to enable future-vertex updating for its neighbors (i.e, vertex 4, 5, 6, 7 and 9), and the corresponding edges will not be scheduled in the next iteration. Note that, GraphCP holds the loaded active edges in memory until the vertex updating of the iteration is finished to support the DFVU model.

Figure 5 shows two consecutive iterations of execution to illustrate the efficiency of our designs when the number of active vertices is large. Supposing that the active vertices in the current iteration are vertex 1, 2, 4, 6, 7 and 8. In this case, GraphCP should load the whole sub-block when processing each sub-block to eliminate random disk accesses. In the first iteration, GraphCP iterates over sub-block (1, 1) to sub-block (2, 1) when updating the values of vertices in interval 1, and sub-block (1, 2) to (2, 2) when updating the values of vertices in interval 2. After the vertex values in the current iteration are updated for each sub-block, the DFVU model is triggered to perform future-vertex value updating. During the processing, sub-block (1, 1), sub-block (1, 2) and sub-block (2, 2) can satisfy the condition of future-vertex value updating since their source vertices can be updated before their destination vertices. For sub-block (1, 2), it performs future-vertex value updating after the processing of it in the first iteration is finished. Specifically, it uses the values of interval 1 in the first iteration to update the value of interval 2 in the second iteration (since interval 1 has been updated before processing sub-block (1, 2)). For sub-block (1, 1) and (2, 2), they can also perform future-vertex value computation after vertices in interval 1 and interval 2 are updated respectively. While for sub-block (2, 1), it has to be loaded in the second iteration, since their source vertices are updated behind their destination vertices and cannot support future-vertex value computation in the first iteration.

3.6 Programming model

We have fully implemented GraphCP in C++. The main execution procedure of GraphCP is described in Algorithm 2.

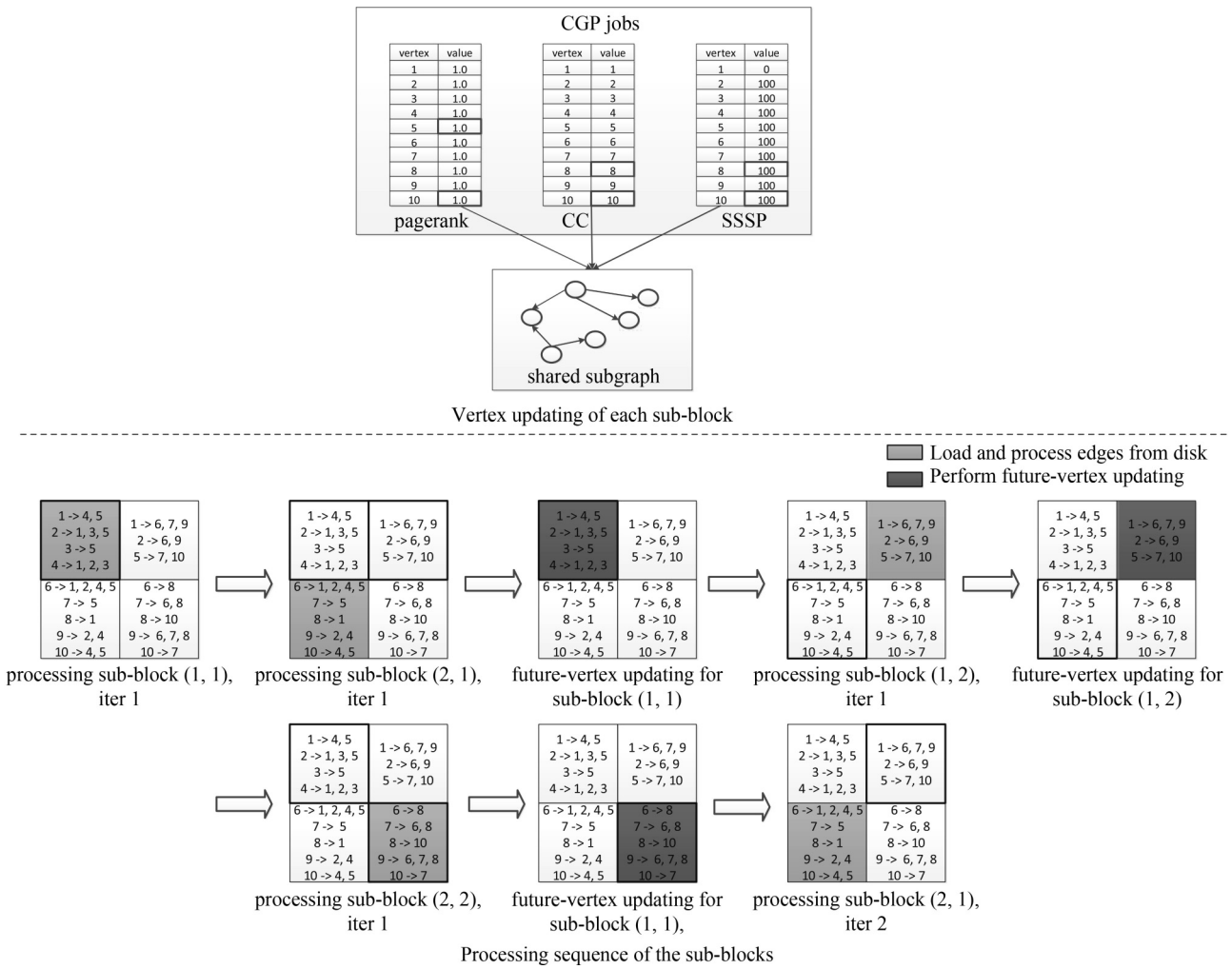


Fig. 5 Two consecutive iterations of execution for GraphCP when the number of active vertices is large

In each iteration, GraphCP first identifies the shared active vertices by merging the active vertex set of all CGP jobs (Lines 12−19). The active vertex set of each CGP job is computed in the previous iteration of processing (i.e., Out_j). Then, GraphCP selects the proper I/O access model based on the shared active vertex set A_s using the I/O cost based benefit evaluation model, and executes different update models based on the selected I/O access model (Line 20−26). We also maintain the vertex values of different iterations (S_j, S_jNI, S_jPI) for vertex updating under the BSP model.

Algorithm 3 shows the updating procedure when GraphCP selectively loads the active edges. For each sub-block, GraphCP first locates the edges of each active vertex in the sub-block according to the vertex index and loads these edges into memory (Line 3). Then, all CGP jobs access these edges and update its own job-specific vertex values in parallel using its own user-defined update function (Lines 4−8). After all CGP jobs finish updating their vertex values in the current iteration, GraphCP performs future-vertex value updating for the new activated vertex based on the DFVU model (Lines 10−17).

Algorithm 4 shows the updating procedure when GraphCP sequentially loads the whole sub-block. For each sub-block, GraphCP loads all edges of this sub-block into memory. Then, each CGP job can implement the user-defined update function (Lines 3−5), and perform future-vertex value updating (DFVU model) if the values of the source interval have been updated before processing the sub-block (Lines 6−10).

Algorithm 2 Pseudo code of GraphCP execution

```
1:  procedure Executor
2:  /*Initialization*/
3:  for each CGP job j in J do
4:      A_j ← ActiveVerticesSet
5:      S_j ← VertexValues
6:      S_jNI ← VertexValuesNextIteration
7:      S_jPI ← VertexValuesPreviousIteration
8:      Out_j ← NewActiveVerticesSet
9:      Out_jNI ← NewActiveVerticesSetNextIteration
10: end for
11: for each iteration iter do
12:     if iter is not the first iteration then
13:         for each j in J do
14:             /* Identify the active vertices*/
15:             A_j ← Out_j
16:         end for
17:     end if
18:     /* Merge the active vertex sets of all CGP jobs*/
19:     A_s ← ⋃ A_j
20:     /* Select the I/O access model*/
21:     IOModel ← Selection(A_s)
22:     if IOModel = selective then
23:         SelectiveLoadAndUpdate /*Algorithm 3*/
24:     else
25:         SequentialLoadAndUpdate /*Algorithm 4*/
26:     end if
27: end for
28: end procedure
```

Algorithm 3 Selective load and update

```
1:  for each sub − block do
2:      for each active vertex v in A_s do
3:          v.edges ← SelectiveLoad(v, Index, sub-block)
4:          for each CGP job j in J do
5:              if v ∈ A_j then
6:                  UserFunction(v.edges, S_j, S_jPI, Out_j)
7:              end if
8:          end for
9:      end for
10:     for each CGP job j in J do
11:         for each new activated vertex v_new in Out_j do
12:             if v_new ∈ A_j then
13:                 Out_j.Remove(v_new)
14:                 DFVU(v_new.edges, S_j, S_jNI, Out_jNI)
15:             end if
16:         end for
17:     end for
18: end for
```

Algorithm 4 Sequential load and update

```
1:  for each sub − block do
2:      edges ← SequentialLoad(sub − block)
3:      for each CGP job j in J do
4:          UserFunction(edges, S_j, S_jPI, Out_j)
5:      end for
6:      if The source interval has been updated then
7:          for each CGP job j in J do
8:              DFVU(edges, S_j, S_jNI, Out_jNI)
9:          end for
10:     end if
11: end for
```

In our programming model, functions *UserFunction* and *DFVU* are user-defined, while the others are provided by runtime. Users can modify the *UserFunction* and *DFVU* to write specific vertex updating function for different CGP jobs. Specifically, *UserFunction* is applied to the edges to update their destination vertices' value in the current iteration. *DFVU* is executed for cross-iteration vertex values computation, which updates the values of vertices in the next iteration in advance. Algorithm 5 shows the implementations of *UserFunction* and *DFVU* with the example of Connected Components (CC).

4 Evaluation

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

4.1 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, 32GB main memory and 1TB HDD,

Algorithm 5 Example: Connected components

```
1: procedure UserFunction(edges, S_j, S_jPI, Out_j)
2: for each edge e in edges do
3:     if S_jPI(e.src) < S_j(e.dst) then
4:         S_j(e.dst) ← S_jPI(e.src)
5:         Out_j.add(e.dst)
6:     end if
7: end for
8: end procedure
9: procedure DFVU(edges, S_j, S_jNI, Out_jNI)
10: for each edge e in edges do
11:     if S_j(e.src) < S_jNI(e.dst) then
12:         S_j(e.dst)NI ← S_j(e.src)
13:         Out_jNI.add(e.dst)
14:     end if
15: end for
16: end procedure
```

running Ubuntu 16.04 LTS. In addition, a 128GB SATA2 SSD is installed to evaluate the scalability. All programs are compiled with gcc version 5.4.0.

We use different types of graphs for the evaluation as summarized in Table 2. LiveJournal (LJ) [24], Twitter2010 (TT) [25] and SK2005 (SK) [26] are social graphs, showing the relationships between users within each online social network. UK2007 (UK) [27] is a web graph that consists of hyperlink relationships between web pages. Kron30 (KR) is a synthetic graph generated by the Graph500 generator. The small graph LiveJournal is chosen to evaluate the in-memory processing performance of GraphCP. For the other four graphs, we limit the memory budget to 10% of the graph data to better evaluate the out-of-core processing performance.

We run four graph algorithms as concurrent jobs: PageRank (PR), Breadth-first search (BFS), Weakly 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.

We compare GraphCP with five baseline systems, including two out-of-core graph processing systems GridGraph [7] and GraphZ [28], two concurrent graph processing systems Seraph [12,13] and GraphSO [29], and the first version of GraphCP (we name it as GraphCP-v1) [15]. GridGraph is a state-of-art out-of-core graph processing system that 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. GraphZ greatly reduces the I/O costs and improves performance by using the degree-ordered storage and dynamic messages. Seraph is a state-of-art graph processing system optimized for efficient execution of CGP jobs through sharing the in-memory graph structure for all jobs. GraphSO is specially designed to support disk-based CGP processing by reducing unnecessary disk I/Os. We implement Seraph and GraphSO based on GridGraph. For all compared systems, we provide 16 execution threads for the executions.

Table 2 Datasets used in evaluation

Dataset	Vertices	Edges	Type
LiveJournal	4.8 million	69 million	Social graph
Twitter2010	42 million	1.5 billion	Social graph
SK2005	51 million	1.9 billion	Social graph
UK2007	106 million	3.7 billion	Web graph
Kron30	1 billion	32 billion	Synthetic graph

4.2 Comparison to other systems

We first compare the overall performance of the algorithm executions for all compared systems. To this end, we run all the CGP jobs (PageRank, BFS, WCC, and SSSP) simultaneously for each of these systems. Note that, for Seraph, GraphSO, GraphCP-v1 and GraphCP, we only need to run a single 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 3 shows the total execution time of the CGP jobs for different systems. We can see that GraphCP finishes the CGP jobs more quickly than other systems in all cases, when processing the four disk-reside graphs (i.e., TT, SK, UK and KR). Specifically, GraphCP outperforms GridGraph, GraphZ, Seraph and GraphSO by 20.5×, 8.9×, 3.5× and 1.7× respectively on average. For LJ, GraphCP outperforms other systems by 2.6×, 1.5×, 1.5×, 1.3× and 1.0× respectively. Obviously, the performance gap is shorten since the disk I/O related optimizations of GraphCP are disabled when processing the in-memory graph.

For GridGraph and GraphZ, they can efficiently process graphs from disk for the individual job. However, when handling CGP jobs, they exhibit poor performance due to many redundant accesses. Moreover, many I/O requests to the disk initiated by different CGP jobs cause severe competition for the limited I/O bandwidth, which greatly reduces the I/O throughput and performance. 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.

For Seraph, although it is also able to spare the data accesses via shared in-memory graph structure, it has not captured the states of vertices and loads a large amount of inactive data during the processing. In addition, it has not efficiently scheduled the computation loads of different CGP jobs, which causes load imbalance and significant synchronization overheads for the iterative processing.

GraphSO can reduce the unnecessary (inactive) disk I/Os by using a repartition-centric execution model that only loads the active chunks and constructs new logical partitions with these

Table 3 Execution time (s)

	LJ	TT	SK	UK	KR
GridGraph	16.9	3115.4	3807.1	6911.2	–
GraphZ	9.4	1639.6	1586.3	2303.7	–
Seraph	9.6	328.2	768.4	1573.3	130944.5
GraphSO	8.5	197.5	326.1	702.5	62354.5
GraphCP-v1	6.3	205.1	349.3	561.9	56932.4
GraphCP	**6.4**	**128.2**	**232.9**	**330.5**	**40666.2**

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

chunks. It also efficiently buffers several frequently accessed chunks in the DRAM to further reduce I/O traffics. However, it incurs extra runtime overheads for the repartition procedure especially when the number of active chunks is large. Furthermore, it has not exploited dependencies among vertices to support future-value updating.

When compared with GraphCP-v1, GraphCP can achieve a speedup by up to 1.7×. The main reason for the performance improvement is the using of the DFVU model that reduces disk I/Os in the future iterations. In addition, the I/O cost based benefit evaluation model adopted in GraphCP enables the system to make a more accurate decision when selecting the I/O access model.

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 Fig. 6. From the results, we can see that the high efficiency of GraphCP mainly stems from its superior disk I/O performance. Specifically, the disk I/O time of GraphCP is less than that of other four systems by 1.5×, 4.5×, 9.8× and 24.9× respectively.

We then compare the I/O traffics of all systems, as shown in Fig. 7. Note that, since GridGraph and GraphZ fail to finish the executions on Kron30 in 48 hours, the corresponding I/O traffic results are not included. Thanks to the benefit-aware sharing execution model and dependency-based future-vertex

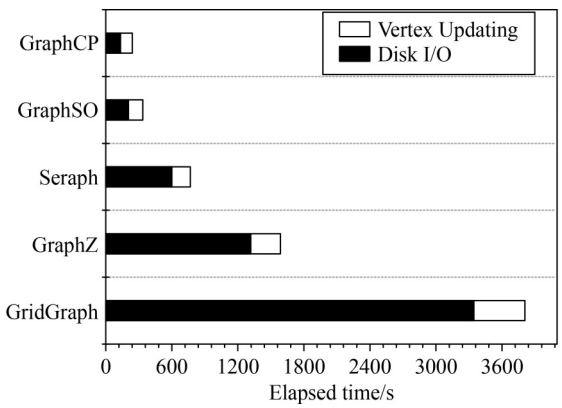


Fig. 6 Runtime breakdown on SK2005

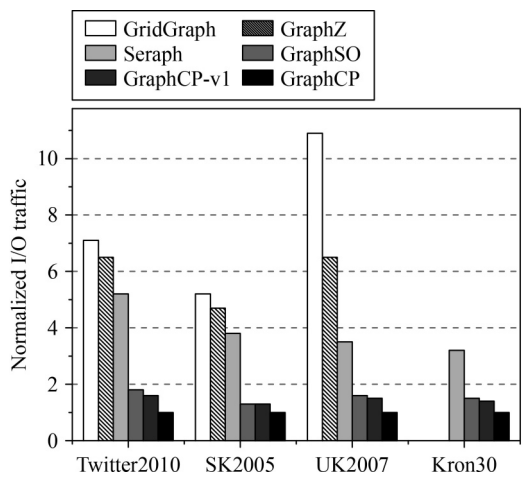


Fig. 7 I/O traffic comparison

updating model, GraphCP can simultaneously exploit the states and dependencies of graph data to reduce the unnecessary disk I/Os, producing much less I/O amount than other systems. On average, the volume of I/O traffic of GraphCP is 7.7×, 5.9×, 4.2×, 1.6× and 1.5× less than that of GridGraph, GraphZ, Seraph, GraphSO and GraphCP-v1 respectively.

4.3 Performance on different workloads

We also evaluate the performance of GraphCP on different workloads (listed in Table 4) to demonstrate its ability to support different types of CGP jobs. The workloads are classified into three categories: Heterogeneous, Homogeneous and Similar Algorithms. Specifically, BFS and SSSP are combined as a Similar-Algorithm workload since they both start processing from a source vertex and traverse the graph in each iteration. PR and WCC are combined as a Similar-Algorithm workload since they both start processing from all vertices. Each workload has 4 concurrent jobs. For BFS and SSSP, the source vertices are randomly selected.

Figure 8 shows the performance comparisons on different workloads for Twitter2010 and UK2007. We can see that GraphCP outperforms other systems in all cases, thanks to its superb I/O performance. On the other hand, the characteristics of the workloads can impact the performance of GraphCP. GraphCP can obtain more performance improvements when the workloads are more homogeneous. This is because similar works usually have similar I/O access patterns, which may augment the advantage of GraphCP's benefit-aware sharing execution model. In addition, we can also observe that the performance improvements on Sim2 are smaller compared with other workloads. This is because the algorithms in Sim2 have a large number of active vertices during the processing, thus the selective data access of GraphCP can only bring limited benefits.

4.4 Preprocessing time

The comparisons of preprocessing time of different systems are depicted in Fig. 9. For the four graphs, the preprocessing time of the baseline system (i.e., GraphCP) is 224s, 409s, 846s and 28003s. The preprocessing procedure includes loading the raw graph data, partitioning and sorting the raw graph data, and writing the preprocessed graph data to disk. From the results, GraphZ takes the longest preprocessing time because it has to create the degree-ordered storage format. The other two concurrent graph processing systems Seraph and GraphSO implemented based on GridGraph take more preprocessing time than GridGraph since they need to incorporate the CGP-related designs. For example, GraphSO needs to generate the TChunk [19] structure that records the information of chunks. GraphCP takes more preprocessing time than Seraph and

Table 4 Workloads of the CGP Jobs

Workload	Algorithms	Characteristic
Het	{PR, BFS, WCC, SSSP}	Heterogeneous
Hom1	{BFS} × 4	Homogeneous
Hom2	{SSSP} × 4	Homogeneous
Sim1	{BFS, SSSP} × 2	Similar Algorithms
Sim2	{PR, WCC} × 2	Similar Algorithms

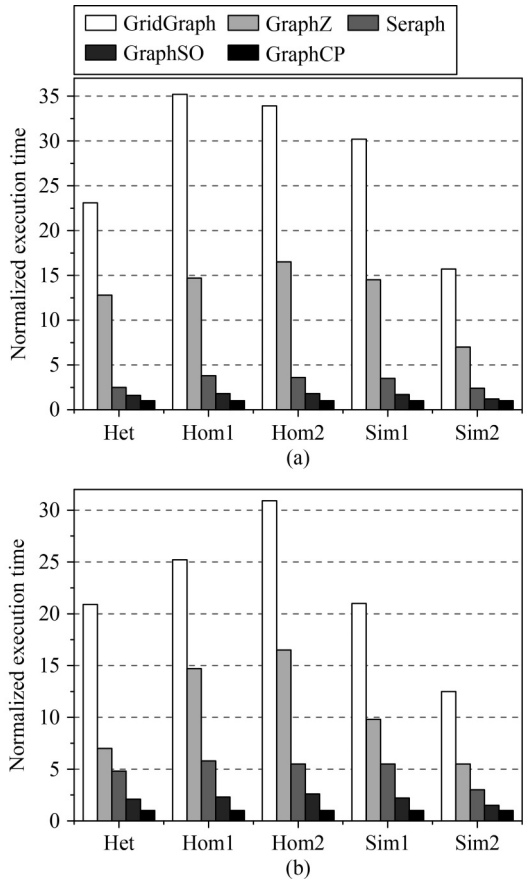


Fig. 8 Performance comparisons on different workloads. (a) Twitter2010; (b) UK2007

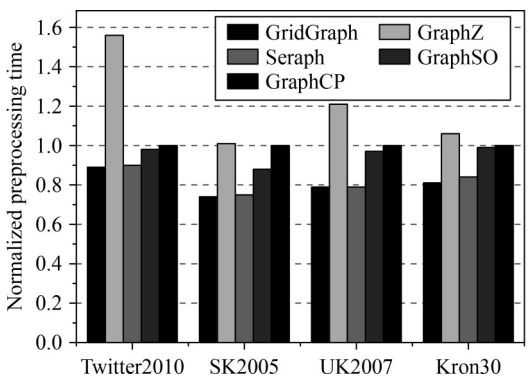


Fig. 9 Preprocessing time comparison

GraphSO as it sorts the edges to construct the source-sorted sub-block representation. Specifically, the preprocessing time of GraphCP is 23% and 6% longer than that of Seraph and GraphSO. Fortunately, the small extra preprocessing overheads can be offset by the I/O performance improvements. For example, by sorting the edges, GraphCP can enable selective loading of edges to skip many unnecessary I/Os. Furthermore, the preprocessed graph can be reused for many times, which significantly amortizes the preprocessing overheads.

4.5 Effects of benefit-aware sharing execution model

We first evaluate the effect of the benefit-aware sharing execution model. The key advantage of this model is that it dynamically schedules the edge loading based on the states of vertices and skips loading inactive edges whenever such skipping can bring performance benefit. We compare GraphCP with two baseline implementations that use different I/O access models, as shown in Fig. 10. The first baseline implementation (GraphCP-b1) sequentially loads the whole sub-block into memory when processing each sub-block. The second baseline implementation (GraphCP-b2) randomly loads the active edges when processing each sub-block regardless of the number of active vertices. The results show that the benefit-aware scheduling scheme can improve the performance by up to 3.0×. On the other hand, GraphCP-b1 outperforms GraphCP-b2 by 1.5× on average. This is because GraphCP-b2 incurs many random disk accesses that significant degrade system performance.

We also evaluate the overheads incurred by the benefit-aware sharing execution model, which refers to the extra computation overheads of performance benefits evaluation. Specifically, we compare the computation overheads and the reduced I/O time overheads optimized by the benefit-aware sharing execution model. As shown in Table 5, the benefit-aware sharing execution model can significantly reduce the I/O time overheads with acceptable extra computation overheads.

4.6 Effects of I/O cost based benefit evaluation model

We then evaluate the efficiency of the I/O cost based benefit evaluation model. To this end, we compare the execution time of GraphCP-b1, GraphCP-b2 and GraphCP in each iteration on Twitter2010, as shown in Fig. 11. For GraphCP-b1, the execution time in each iteration is similar since it loads the whole sub-blocks in each iteration. For GraphCP-b2, the execution time is much longer in the first few iterations when the number of active vertices is large. While for GraphCP, it is able to achieve the optimal performance in each iteration. This

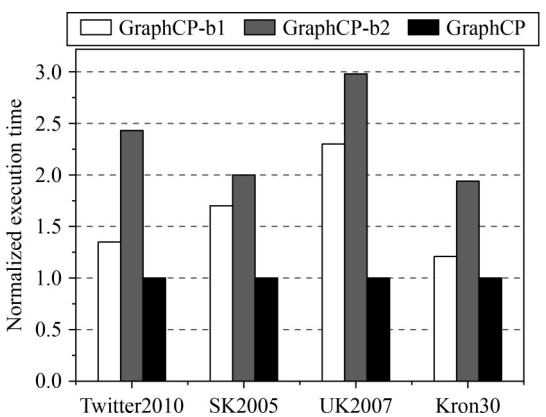


Fig. 10 Comparison of the performance for different I/O access models

Table 5 Extra computation overheads vs. Reduced I/O time overheads

Dataset	Computation overheads/s	Reduced I/O overheads/s
Twitter2010	3.4	71.6
SK2005	12.1	214.5
UK2007	15.2	618.1
Kron30	293.1	11386.5

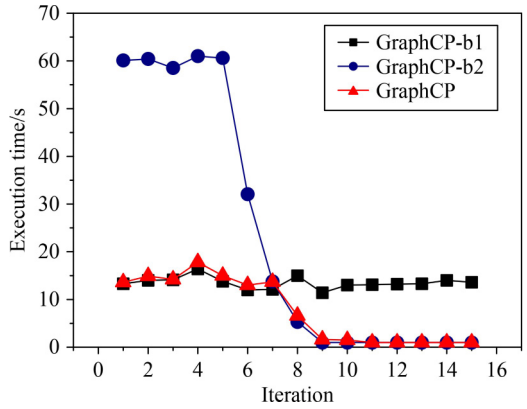


Fig. 11 Comparison of execution time in each iteration for different I/O access models (From 1 to 15 iteration)

verifies the accurate prediction of the I/O cost based benefit evaluation model, which enables GraphCP to select the better I/O access model in each iteration.

4.7 Effects of dependency-based future-vertex updating model

To evaluate the effect of the dependency-based future-vertex updating (DFVU) model, we compare the execution time and I/O traffic of GraphCP with/without using the DFVU model, as shown in Fig. 12. As we see from the results, GraphCP can achieve both better performance and less I/O traffics when using the DFVU model. Specifically, for execution time, GraphCP outperforms GrapCP-without by 1.6×. As for I/O traffic, the amount of disk I/Os of GrahCP is less than that of GraphCP-without by 1.5×. This is attributed to the DFVU model that efficiently exploits the dependencies among vertices to enable future-vertex value updating and merge disk I/Os for several iterations.

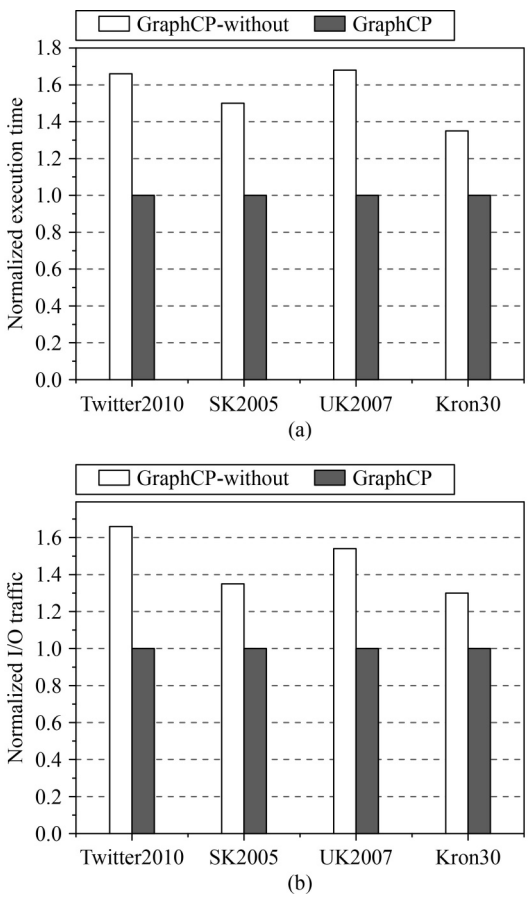


Fig. 12 Performance comparison of GraphCP with/without the DFVU model. (a) Comparison of execution time; (b) comparison of I/O traffic

4.8 Scalability

The scalability of GraphCP is evaluated by observing the performance improvement when more hardware resource is added. Figure 13 shows the effect of the thread number on execution time when executing CGP jobs on LiveJournal and SK2005. For the small graph LiveJournal, when the number of thread increases from 1 to 8, the performances of these systems are improved by 2.9×, 2.7×, 4.3×, 5.1× and 4.9× respectively. This shows that the concurrent systems GraphCP, Seraph and GraphSO achieve better scalability than GridGraph and GraphZ, which stems from their efficient sharing of data accesses. For SK2005, since system performance is limited by disk I/O, thread number has relatively less impact on the performance. Specifically, the performances of these systems are improved by 1.3×, 1.2×, 1.5×, 2.1× and 1.9× respectively, when the number of thread

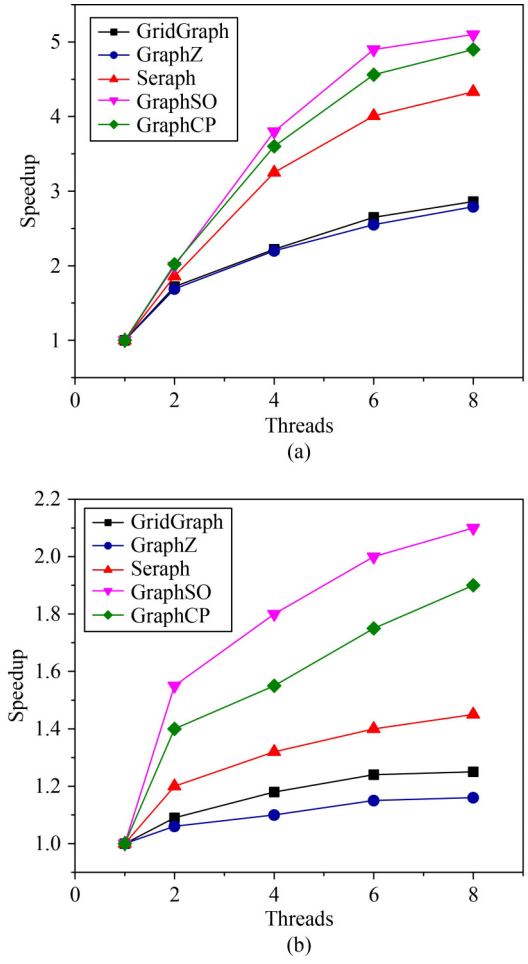


Fig. 13 Performance variation with different number of threads. (a) LiveJournal; (b) SK2005

increases from 1 to 8. Among these systems, GraphSO and GraphCP achieve a better scalability. This is because they have better I/O performance, so the vertex updating time occupies a larger proportion than other systems. In addition, GraphSO achieves a better scalability than GraphCP in these experiments. This is attributed to GraphSO's repartition-centric execution model, which can better assign the computation resources for each job according to the corresponding computational load when processing each chunk. This mitigates the Straggler problem and improves the parallel processing performance. GraphCP allocates the computation resources for the each CGP job only based on the number of active edges in a sub-block, and we leave implementing more advanced computation resources allocation policies as future work.

Figure 14 shows the performance improvement of CGP jobs on SK2005 when using different I/O devices. Compared with performance on HDD, GridGraph, GraphZ, Seraph, GraphSO and GraphCP respectively achieve a speedup of 1.4×, 1.6×, 1.6×, 1.9× and 1.9× when using SSD. This indicates that GraphSO and GraphCP can benefit more from the utilization of SSD, since they enable selective access to load the active edges or chunks, which works well on SSD.

We also evaluate the performance variation when increasing the number of CGP jobs, as shown in Fig. 15. 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, GraphSO and GraphCP increases by 6.9×, 10.6×, 1.4×, 1.2× and 1.3× respectively. Obviously, Seraph, GraphSO and GraphCP achieve a better scalability than the other two out-of-core 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.

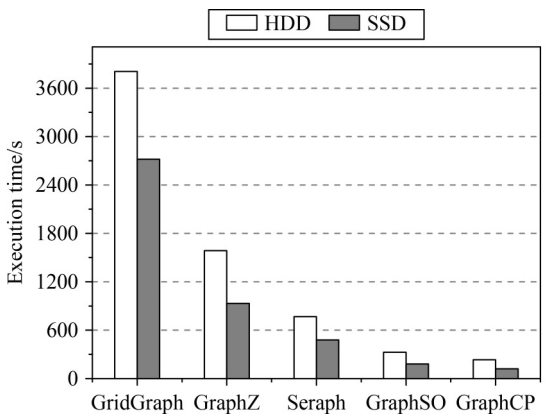


Fig. 14 Performance variation on different I/O devices

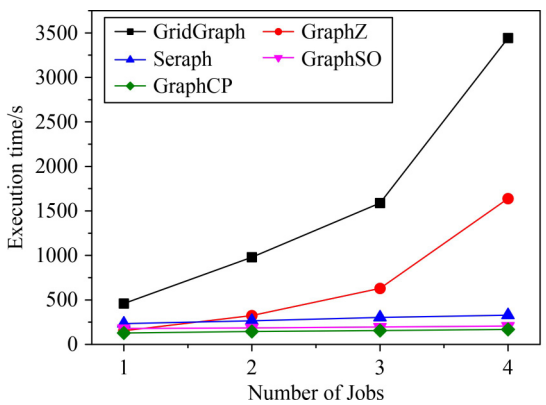


Fig. 15 Performance variation with different number of jobs

5 Related works

5.1 Out-of-core graph processing systems

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. It adopts an interval-shard structure to store a graph. The vertices are divided into disjoint intervals and a shard structure is created for each interval to store the incoming edges. 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 reduce random disk I/Os and 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. VENUS [30] uses a vertex-centric streamlined computing model and proposes a new storage scheme that streams the graph data while performing computation. NXgraph [31] provides three novel update strategies under the Destination-Sorted Sub-Shard (DSSS) structure so as to further ensure locality of graph data access. GraphZ [28] 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 I/O pressure.

In addition to eliminating random disk accesses, some systems further reduce disk I/Os by avoiding loading and processing the inactive graph data. Dynamic Shards [32] removes unnecessary I/Os of out-of-core graph processing by employing dynamic partitions whose layouts are dynamically adjustable. HUS-Graph [33] proposes a hybrid update strategy that adaptively selects I/O access and computing model based on the number of active vertices, which achieves a good balance between I/O traffic and I/O access locality. MultiLogVC [34] uses a combination of the CSR graph format, and message logging to reduce read amplification caused by reading inactive vertices and edges.

Other systems [8,16,35] improve performance by enabling

future-value computation. These systems perform multi-iteration of computation in one round of graph loading, which significantly speeds up the convergence and reduces disk I/Os. CLIP [16] uses a reentry method to make full use of the loaded edge blocks, enabling more efficient algorithms that require fewer total disk I/Os. Lumos [8] adopts an out-of-order execution model to proactively propagate values across iterations while simultaneously providing synchronous processing guarantees, which avoids loading the corresponding graph portions in the future iterations.

5.2 Concurrent graph processing systems

To handle multiple concurrent graph processing and queries, several concurrent graph processing and querying systems are developed in recent years. Seraph [12,13] 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 [36] 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 [37] is an edge-set based concurrent graph traversal framework that achieves both high concurrency and efficiency for k-hop reachability queries.

CGraph [11] 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 [14] 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. GraphSO [19] judiciously loads the required graph data to efficiently construct the logical partitions for the execution of the concurrent jobs, thereby reducing much unnecessary I/O overhead and maximizing the utilization of the loaded graph data. LCCG [38] and Krill [29] unify the data accesses of all CGP jobs by exploring the common traversal paths of the CGP jobs, in order to reduce redundant data accesses and achieve higher throughput of the execution.

6 Conclusion

In this paper, we present an I/O-efficient 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, a dependency-based future-vertex updating model is adopted to enable future-vertex value updating and reduce disk I/Os in the future iterations. To improve processing capacity and ensure I/O access locality, GraphCP adopts a Source-Sorted Sub-Block graph representation to organize graph data. Our evaluation results show that GraphCP can be much faster than GridGraph, GraphZ, Seraph and GraphSO, four state-of-the-art graph processing systems.

Acknowledgements This work was supported by the National Natural Science Foundation of China (Grant Nos. 61832020, 61821003 and U1705261), National Defense Preliminary Research Project (No. 31511010202), the Fundamental Research Funds for the Central Universities, the Open Project Program of Wuhan National Laboratory for Optoelectronics (No. 2022WNLOKF017), the Natural Science Foundation of Fujian Province (No. 2020J01493), Zhejiang provincial “Ten Thousand Talents Program” (No. 2021R52007) and Center-initiated Research Project of Zhejiang Lab (No. 2021DA0AM01).

References

1. Malewicz G, Austern M H, Bik A J C, Dehnert J C, Horn I, Leiser N, Czajkowski G. Pregel: a system for large-scale graph processing. In: Proceedings of 2010 ACM SIGMOD International Conference on Management of Data. 2010, 135−146
2. Low Y, Bickson D, Gonzalez J, Guestrin C, Kyrola A, Hellerstein J M. Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment, 2012, 5(8): 716–727
3. Gonzalez J E, Low Y, Gu H, Bickson D, Guestrin C. PowerGraph: distributed graph-parallel computation on natural graphs. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. 2012, 17−30
4. Zhu X, Chen W, Zheng W, Ma X. Gemini: a computation-centric distributed graph processing system. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. 2016, 301−316
5. Kyrola A, Blelloch G, Guestrin C. GraphChi: large-scale graph computation on just a PC. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. 2012, 31−46
6. Roy A, Mihailovic I, Zwaenepoel W. X-stream: edge-centric graph processing using streaming partitions. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles. 2013, 472−488
7. Zhu X, Han W, Chen W. GridGraph: large-scale graph processing on a single machine using 2-level hierarchical partitioning. In: Proceedings of 2015 USENIX Conference on Usenix Annual Technical Conference. 2015, 375−386
8. Vora K. LUMOS: dependency-driven disk-based graph processing. In: Proceedings of 2019 USENIX Conference on Usenix Annual Technical Conference. 2019, 429−442
9. Chen R, Shi J, Chen Y, Zang B, Guan H, Chen H. PowerLyra: differentiated graph computation and partitioning on skewed graphs. ACM Transactions on Parallel Computing, 2019, 5(3): 13
10. Cheng Y, Wang F, Jiang H, Hua Y, Feng D, Zhang L, Zhou J. A communication-reduced and computation-balanced framework for fast graph computation. Frontiers of Computer Science, 2018, 12(5): 887–907
11. Zhang Y, Liao X, Jin H, Gu L, He L, He B, Liu H. Cgraph: a correlations-aware approach for efficient concurrent iterative graph processing. In: Proceedings of 2018 USENIX Conference on Usenix Annual Technical Conference. 2018, 441−452
12. Xue J, Yang Z, Qu Z, Hou S, Dai Y. 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, 227−238
13. Xue J, Yang Z, Hou S, Dai Y. Processing concurrent graph analytics with decoupled computation model. IEEE Transactions on Computers, 2017, 66(5): 876–890
14. Zhao J, Zhang Y, Liao X, He L, He B, Jin H, Liu H, Chen Y. GraphM: an efficient storage system for high throughput of concurrent graph

processing. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2019, 3
15. Xu X, Wang F, Jiang H, Cheng Y, Feng D, Zhang Y, Fang P. GraphCP: an I/O-efficient concurrent graph processing framework. In: Proceedings of the 29th IEEE/ACM International Symposium on Quality of Service. 2021, 1−10
16. Ai Z, Zhang M, Wu Y, Qian X, Chen K, Zheng W. Squeezing out all the value of loaded data: an out-of-core graph processing system with reduced disk I/O. In: Proceedings of 2017 USENIX Conference on Usenix Annual Technical Conference. 2017, 125−137
17. Zhu R, Zhao K, Yang H, Lin W, Zhou C, Ai B, Li Y, Zhou J. AliGraph: a comprehensive graph neural network platform. Proceedings of the VLDB Endowment, 2019, 12(12): 2094–2105
18. Maleki S, Nguyen D, Lenharth A, Garzarán M, Padua D, Pingali K. DSMR: a parallel algorithm for single-source shortest path problem. In: Proceedings of 2016 International Conference on Supercomputing. 2016, 32
19. Liao X, Zhao J, Zhang Y, He B, He L, Jin H, Gu L. A structure-aware storage optimization for out-of-core concurrent graph processing. IEEE Transactions on Computers, 2022, 71(7): 1612–1625
20. Liu H, Huang H H. Graphene: fine-grained IO management for graph computing. In: Proceedings of the 15th USENIX Conference on File and Storage Technologies. 2017, 285−299
21. Liu W, Liu H, Liao X, Jin H, Zhang Y. Straggler-aware parallel graph processing in hybrid memory systems. In: Proceedings of the 21st IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing. 2021, 217−226
22. Agostini M, O'Brien F, Abdelrahman T. Balancing graph processing workloads using work stealing on heterogeneous CPU-FPGA systems. In: Proceedings of the 49th International Conference on Parallel Processing. 2020, 50
23. Valiant L G. A bridging model for parallel computation. Communications of the ACM, 1990, 33(8): 103–111
24. Backstrom L, Huttenlocher D, Kleinberg J, Lan X. 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. 2006, 44−54
25. Kwak H, Lee C, Park H, Moon S. What is twitter, a social network or a news media? In: Proceedings of the 19th International Conference on World Wide Web. 2010, 591−600
26. Boldi P, Vigna S. The webgraph framework I: compression techniques. In: Proceedings of the 13th International Conference on World Wide Web. 2004, 595−602
27. Boldi P, Santini M, Vigna S. A large time-aware web graph. ACM SIGIR Forum, 2008, 42(2): 33–38
28. Zhou Z, Hoffmann H. GraphZ: improving the performance of large-scale graph analytics on small-scale machines. In: Proceedings of the 34th IEEE International Conference on Data Engineering. 2018, 1368−1371
29. Chen H, Shen M, Xiao N, Lu Y. Krill: a compiler and runtime system for concurrent graph processing. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2021, 51
30. Cheng J, Liu Q, Li Z, Fan W, Lui J C S, He C. VENUS: vertex-centric streamlined graph computation on a single PC. In: Proceedings of the 31st IEEE International Conference on Data Engineering. 2015, 1131−1142
31. Chi Y, Dai G, Wang Y, Sun G, Li G, Yang H. NXgraph: an efficient graph processing system on a single machine. In: Proceedings of the 32nd IEEE International Conference on Data Engineering. 2016, 409−420
32. Vora K, Xu G, Gupta R. Load the edges you need: a generic I/O optimization for disk-based graph processing. In: Proceedings of 2016 USENIX Conference on Usenix Annual Technical Conference. 2016, 507−522
33. Xu X, Wang F, Jiang H, Cheng Y, Feng D, Zhang Y. A hybrid update strategy for I/O-efficient out-of-core graph processing. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(8): 1767–1782
34. Matam K K, Hashemi H, Annavaram M. MultilogVC: efficient out-of-core graph processing framework for flash storage. In: Proceedings of 2021 IEEE International Parallel and Distributed Processing Symposium. 2021, 245−255
35. Zhang M, Wu Y, Zhuo Y, Qian X, Huan C, Chen K. Wonderland: a novel abstraction-based out-of-core graph processing system. ACM SIGPLAN Notices, 2018, 53(2): 608–621
36. Pan P, Li C. Congra: towards efficient processing of concurrent graph queries on shared-memory machines. In: Proceedings of 2017 IEEE International Conference on Computer Design. 2017, 217−224
37. Zhou L, Chen R, Xia Y, Teodorescu R. C-graph: a highly efficient concurrent graph reachability query framework. In: Proceedings of the 47th International Conference on Parallel Processing. 2018, 79
38. Zhao J, Zhang Y, Liao X, He L, He B, Jin H, Liu H. LCCG: a locality-centric hardware accelerator for high throughput of concurrent graph processing. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2021, 45

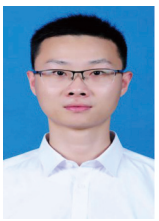
Xianghao Xu received the PhD degree from Huazhong University of Science and Technology, China in 2021. He is currently an assistant professor in School of Computer Science and Engineering, Nanjing University of Science and Technology, China. His current research interests include graph processing, computer architecture and big data analytics. He has several publications in major international conferences and journals, including IEEE-TPDS, JPDC, ICPP, IWQoS and FCS.

Fang Wang received her PhD degree in computer architecture in 2001 from Huazhong University of Science and Technology (HUST), China. She is a professor of computer science and engineering at HUST, China. Her interests include distribute file systems, parallel I/O storage systems and graph processing systems. She has more than 80 publications in major journals and conferences, including IEEE-TC, IEEE-TPDS, IEEE-NSM, ACM TACO, SC, MSST, DATE, HiPC, ICDCS, HPDC, ICCD, ICDE, and ICPP.

Hong Jiang received the BE degree from the Huazhong University of Science and Technology, China, and the PhD degree from the Texas A&M University, USA in 1991. He is Wendell H.Nedderman Endowed Professor & Chair of Department of Computer Science and Engineering, University of Texas at Arlington, USA. His research interests include computer architecture, computer storage systems and parallel/distributed computing. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TPDS, IEEE-TC, ACMTOS, ACM TACO, JPDC, ISCA, MICRO, FAST, USENIX ATC, USENIX LISA, SIGMETRICS, MIDDLEWARE, ICDCS, IPDPS, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP.

Yongli Cheng received the PhD degree from Huazhong University of Science and Technology, China in 2017. He is an associated professor of College of Mathematics and Computer Science at Fuzhou University, China currently. His current research interests include computer architecture and graph computing. He has several publications in major international conferences and journals, including HPDC, IWQoS, INFOCOM, ICPP, FGCS, ToN and FCS.

Dan Feng received the BE, ME, and PhD degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE-TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She is a member of the Association for Computing Machinery and the Chair of the Information Storage Technology Committee, Chinese Computer Academy. She served on the program committees of multiple international conferences, including SC, in 2011 and 2013, and MSST, in 2012 and 2015.



Peng Fang received the BE degree in computer science and technology from Henan Polytechnic University, China in 2014. He is currently a PhD student majoring in computer science and technology in Huazhong University of Science and Technology, China. His current research interests include computer architecture and graph processing.