GraphTune: An Efficient Dependency-Aware Substrate to Alleviate Irregularity in Concurrent Graph Processing

JIN ZHAO, Zhejiang Lab & Huazhong University of Science and Technology, China
YU ZHANG, Huazhong University of Science and Technology, China
LIGANG HE, University of Warwick, UK
QIKUN LI, XIANG ZHANG, XINYU JIANG, HUI YU, XIAOFEI LIAO, HAI JIN, LIN GU, and HAIKUN LIU, Huazhong University of Science and Technology, China
BINGSHENG HE, National University of Singapore, Singapore
JI ZHANG, University of Southern Queensland, Australia
XIANZHENG SONG, LIN WANG, and JUN ZHOU, ANT Group, China

With the increasing need for graph analysis, massive *Concurrent iterative Graph Processing* (CGP) jobs are usually performed on the common large-scale real-world graph. Although several solutions have been proposed, these CGP jobs are not coordinated with the consideration of the inherent dependencies in graph data driven by graph topology. As a result, they suffer from redundant and fragmented accesses of the same underlying graph dispersed over distributed platform, because the same graph is typically irregularly traversed by these jobs along different paths at the same time.

In this work, we develop *GraphTune*, which can be integrated into existing distributed graph processing systems, such as D-Galois, Gemini, PowerGraph, and Chaos, to efficiently perform CGP jobs and enhance system throughput. The key component of GraphTune is a dependency-aware synchronous execution engine in conjunction with several optimization strategies based on the constructed cross-iteration dependency graph of chunks. Specifically, GraphTune transparently regularizes the processing behavior of the CGP jobs in a novel synchronous way and assigns the chunks of graph data to be handled by them based on the topological order of the dependency graph so as to maximize the performance. In this way, it can transform the irregular accesses of the chunks into more regular ones so that as many CGP jobs as possible can fully share the data

This is a new article, not an extension of a conference paper.

This article is supported by National Key Research and Development Program of China under grant No. 2022YFB2404202, NSFC (No. 62072193), Major Scientific Research Project of Zhejiang Lab No. 2022PI0AC03, CCF-AFSG Research Fund No. RF20220211, and the Young Top-notch Talent Cultivation Program of Hubei Province.

Authors' addresses: J. Zhao, Zhejiang Lab & National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, 430074, China; email: zjin@hust.edu.cn; Y. Zhang (corresponding author), Q. Li, X. Zhang, X. Jiang, H. Yu, X. Liao, H. Jin, L. Gu, and H. Liu, National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, Hubei, 430074, China; emails: {zhyu, lqk2021, xiangz, xinyujiang, huiy, xfliao, hjin, lingu, hkliu}@hust.edu.cn; L. He, University of Warwick, Coventry, CV4 7AL, UK; email: ligang.he@warwick.ac.uk; B. He, National University of Singapore, 21 Lower Kent Ridge Road, 119077, Singapore; email: hebs@comp.nus.edu.sg; J. Zhang, University of Southern Queensland, West Street, Toowoomba Qld, 4350, Australia; email: zhangji77@gmail.com; X. Song, L. Wang, and J. Zhou, ANT Group, Xihu District, Hangzhou, Zhejiang, 310000, China; emails: {xianzheng.sxz, fred.wl, jun.zhoujun}@antgroup.com.

1544-3566/2023/07-ART37
https://doi.org/10.1145/3600091

accesses to the common graph. Meanwhile, it also efficiently synchronizes the communications launched by different CGP jobs based on the dependency graph to minimize the communication cost. We integrate it into four cutting-edge distributed graph processing systems and a popular out-of-core graph processing system to demonstrate the efficiency of GraphTune. Experimental results show that GraphTune improves the throughput of CGP jobs by 3.1~6.2, 3.8~8.5, 3.5~10.8, 4.3~12.4, and 3.8~6.9 times over D-Galois, Gemini, PowerGraph, Chaos, and GraphChi, respectively.

CCS Concepts: • **Computer systems organization → Special purpose systems**; **Parallel architectures;**

ACM Reference format:
Jin Zhao, Yu Zhang, Ligang He, Qikun Li, Xiang Zhang, Xinyu Jiang, Hui Yu, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, Ji Zhang, Xianzheng Song, Lin Wang, and Jun Zhou. 2023. GraphTune: An Efficient Dependency-Aware Substrate to Alleviate Irregularity in Concurrent Graph Processing. *ACM Trans. Arch. Code Optim.* 20, 3, Article 37 (July 2023), 24 pages.
https://doi.org/10.1145/3600091

1 INTRODUCTION

Large-scale graph analysis is a crucial task for many enterprises, and they typically perform massive ***Concurrent iterative Graph Processing*** (**CGP**) jobs on their distributed graph processing systems daily to exploit various properties of the same underlying graphs. For example, Facebook [1], Google [2], Twitter [7], and Tencent [6] use Giraph [22], Pregel [36], GraphJet [46], and Plato [54, 62] to handle many graph algorithms for different applications (e.g., the variants of PageRank [42] for content recommendation, ***Weakly Connected Component*** (**WCC**) [23] for social media monitoring, and *k*-means [21] for customer segmentation), respectively. An experiment on a large graph processing platform of a real Chinese social network also shows that a massive number of jobs concurrently and periodically run on the same graph every day and there can be up to 45 CGP jobs [31, 61]. However, existing distributed systems [17, 19, 24, 28, 44, 62] are primarily proposed to process a single graph processing job. To enable efficient execution of multiple CGP jobs on existing graph processing systems, GraphM [31, 61] introduces a graph storage system that can be plugged into these systems to reduce the data access and storage cost of CGP jobs. Although several solutions [15, 31, 52, 53, 57, 60, 61] are proposed to serve the execution of CGP jobs, challenges still remain for high throughput of them.

A major challenge is induced by the irregular data accesses that occur when the CGP jobs process the same graph in different patterns. Specifically, these jobs are usually submitted with various parameters and executed by the graph algorithms with different computing complexities, making these jobs traverse the same graph along different paths (patterns). It leads to the following two types of irregularities. ***(1) Irregular memory access***: These jobs usually access different graph data simultaneously, although most proportion of the graph data are the same for them. Thus, the same graph data may have to be transferred from the memory to the ***Last-Level Cache*** (**LLC**) repeatedly. ***(2) Irregular communication***: The jobs conduct their communications independently to exchange information between the hosts. Our studies show that it incurs many small messages and extra communication cost, leading to underutilized network bandwidth. These irregularities induce the inefficient use of data access channels due to the redundant and fragmented data accesses, eventually resulting in low system throughput.

In this work, we analyse the data access patterns of CGP jobs that process a common graph, and find that the irregular data accesses of different CGP jobs can be regulated by controlling the order in which the graph vertices and edges are handled following the graph topology. In particular, the vertex states are inherently propagated along the state propagating dependencies intrinsic to the graph topology, thus an inactive chunk of graph data needs to be accessed and processed by a job only when its neighbor chunks propagate their vertices' new states to it to activate it. Based on the analyses and findings, we develop an effective and lightweight runtime system, called ***GraphTune***, which can be integrated into existing distributed graph processing systems. The programmers only need to make a minor extension to existing system via a few APIs provided by GraphTune, and then the system can gain performance improvement brought by GraphTune. Fundamentally different from existing solutions, GraphTune proposes an efficient dependency-aware synchronous execution model to fully alleviate the data access irregularities for CGP jobs and enable multiple jobs to fully share the accesses to the same chunks of graph data. Specifically, GraphTune constructs a dependency graph of the chunks to describe the topology of possible vertex state propagations between the chunks when graph algorithm runs through different iterations. Then, GraphTune transparently regulates the traversal paths of the chunks of different jobs to follow the topological order of the dependency graph and triggers the related CGP jobs to concurrently handle each loaded chunk in a synchronous way. By such means, the common chunks can be processed by the related jobs at the same time, and the accesses to them can be fully shared by these jobs. Based on the dependency graph, a coalesced communication scheme is also designed to minimize communication overhead for multiple jobs.

The contributions of this article are summarized as follows.

- We investigate the problem of irregular data access when executing multiple CGP jobs with existing distributed graph processing systems and find that it is the primary cause of low throughput in the CGP jobs.
- We propose an efficient dependency-aware synchronous execution model for CGP jobs to regularize their traversal paths along a common order. It enables multiple CGP jobs to fully share the access to the same graph, significantly reducing data access and storage cost.
- We propose several optimization techniques to improve the performance of running multiple CGP jobs in distributed environments, including (1) an efficient strategy to generate the cross-iteration dependency graph; (2) a load balancing scheme to achieve high resource utilization; (3) a coalesced communication method to reduce the communication cost.
- We integrate GraphTune with five systems, i.e., D-Galois [17], Gemini [62], PowerGraph [19], Chaos [44], and GraphChi [30], and conduct extensive experiments to evaluate the performance improved by GraphTune. The results show that GraphTune improves the throughput of CGP jobs by 3.1~6.2, 3.8~8.5, 3.5~10.8, 4.3~12.4, and 3.8~6.9 times over D-Galois, Gemini, PowerGraph, Chaos, and GraphChi, respectively.

2 BACKGROUND AND MOTIVATION

In existing distributed systems [17, 19, 36, 44, 62], a graph is usually divided into chunks, which are evenly allocated to the hosts for parallel processing and the chunk is the processing unit. The data of each job consist of the job-specific data, i.e., S, which is the set of vertex states (such as the ranking score for PageRank [42] and the distance from source vertex for SSSP [39]) calculated by this job, and the graph structure data, i.e., G=(V, E, W). To efficiently store the graph structure data, the edges of each vertex are stored in the edge array, and the offsets of the beginning and end of the edges in the edge array for each vertex are maintained in the vertex array. The weight of each edge is stored in the weight array. During the execution, each job on the system propagates its vertex states along the edges to update the states of other vertices. In fact, multiple jobs usually process a common graph along different graph paths due to their different submission times, parameters, and computing complexities. Without proper coordination, these CGP jobs exhibit irregular data accesses, causing low system throughput.

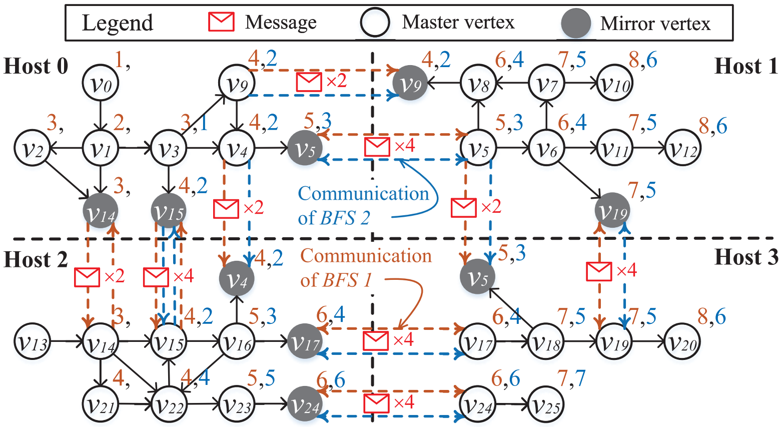


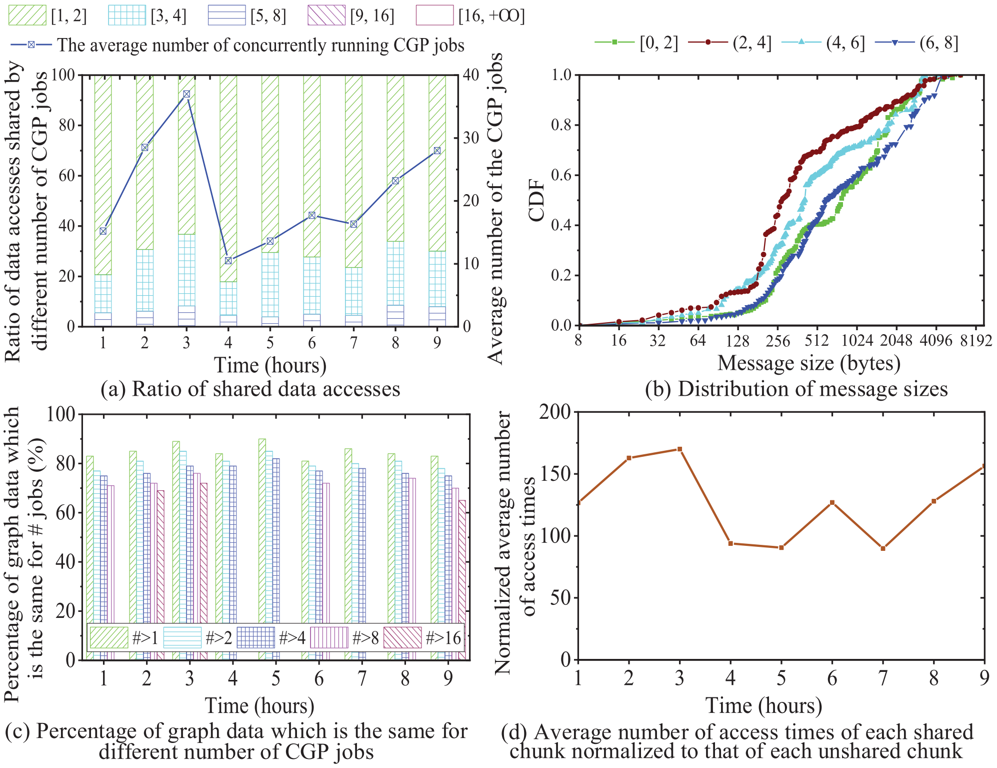
Fig. 1. Execution of two BFS jobs on a distributed platform with four hosts.

2.1 Irregular Data Access of the CGP Jobs

Several approaches have been proposed in literature [15, 52, 53, 57, 60, 61] to enable the accesses associated with the common graph data to be locally shared by CGP jobs in each iteration. Specifically, these approaches explore the graph structure data that each job needs to process before each iteration and enable the accesses associated with the intersections of the explored data to be shared by different CGP jobs. Nevertheless, different CGP jobs usually traverse the same graph along different graph paths, because these jobs typically own various characteristics, e.g., submitting with different parameters and executing with different computing complexities. As a result, the accesses to the same graph structure data are not coordinated among different CGP jobs, inducing significant unnecessary overhead for data storage and access.

Let us employ Figure 1 to illustrate the problem. Two ***Breadth-First Search*** **(BFS)** [3] jobs traverse the graph from different vertices, i.e., *BFS* 1 starting from v_0 and *BFS* 2 from v_3. In the figure, the orange and blue numbers above the vertices represent the iteration number in which the vertices are handled by *BFS* 1 and *BFS* 2, respectively. The traversing order of the vertices (i.e., the traversal path) can also be reflected by the iteration numbers.

Observation 1: Most proportion of the same graph data are accessed repeatedly by various CGP jobs at different times, resulting in low ratio of shared data accesses. We can see from Figure 1 that most vertices need to be handled by the two BFS jobs, but in different iterations. When *BFS* 1 and *BFS* 2 start the first iteration, they first access v_0 and v_3, respectively. Although v_3 also needs to be accessed by *BFS* 1 at its third iteration, with existing methods, *BFS* 1 and *BFS* 2 access v_3 individually. The graph data associated with v_3 have to be reloaded again to serve *BFS* 1 at its third iteration. In this example, due to such *irregular memory accesses*, these two jobs only access v_{22}, v_{23}, v_{24}, and v_{25} at the same time and can share the accesses of these vertices, while the accesses of other vertices cannot be shared. It causes low ratio of shared data accesses. Figure 2(a) shows the ratio of accesses shared by the CGP jobs executed on a cutting-edge system called D-Galois-M (i.e., D-Galois [17] integrated with GraphM [61]) on uk-union [11], where the jobs are submitted according to the trace profiled from a real Chinese social network. The details of the platform and benchmark are presented in Section 5. From Figure 2(a), we can find that the accesses to the same graph data are only shared by a small portion of jobs. Less than 8% of the accesses are shared by five or more jobs. It indicates that multiple copies of the same graph data have to be repeatedly swapped into the LLC to support the execution of various jobs at different time.



(a) Ratio of shared data accesses

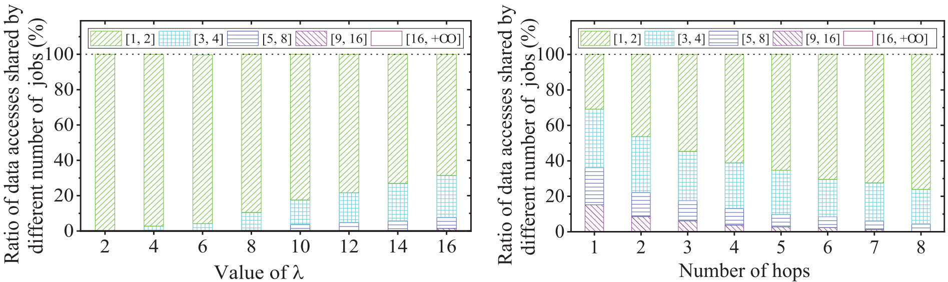
(b) Distribution of message sizes

(c) Percentage of graph data which is the same for different number of CGP jobs

(d) Average number of access times of each shared chunk normalized to that of each unshared chunk

Fig. 2. Performance of the CGP jobs over uk-union: (a) the ratio of accesses shared by the jobs, where those legends (e.g., the first legend [1, 2]) for the columns mean the ratio of data accesses shared by the range of the number of CGP jobs (i.e., ranging from 1 to 2); (b) the cumulative distribution of the sizes of the messages generated in D-Galois-M every two hours (e.g., the legend [0, 2] means the time interval between 0 hour and 2 hour); (c) the percentage of graph data which is the same for different number of CGP jobs; (d) the average number of access times of each shared chunk normalized to that of each unshared chunk in each hour.

Observation 2: The state synchronization of the same vertex is independently performed by various CGP jobs at different time, incurring many small messages and much redundant communication cost. After graph partitioning [17, 19, 32, 44, 62], a vertex may have multiple mirrors distributed over many hosts. For example, the shaded vertices in Figure 1 are the mirror vertices. When a mirror vertex's state is updated, this state needs to be synchronized with its master vertex in another host, and then the master's new state also needs to be broadcasted to its mirrors on other hosts. With existing solutions [52, 53, 57, 60, 61], state synchronization of the same vertex handled by various jobs are triggered independently at different time, resulting in significant *irregular communication.* For example, the state of mirror vertex v_{15} on host 0 is updated by *BFS* 1 after the third iteration but by *BFS* 2 after the first iteration. Thus, multiple messages are generated for the two jobs to synchronize their states of the same vertex. More importantly, most of them are small messages and are unable to be fully consolidated. It is because, after a few iterations, only a few vertices in most chunks are active and need to send their new states, due to the power-law property [19]. In Figure 2(b), more than 73% of the messages have the size less than ***Maximum Transmission Unit*** (**MTU**) [34, 35], i.e., 2 KB, of the network. These small messages eventually cause underutilized network bandwidth and also high cost of packet header. In addition, although many messages



(a) Ratio of shared data accesses for different λ (b) Ratio of shared data accesses for different hops

Fig. 3. The ratio of the data accesses shared by different number of BFS jobs: (a) when submitting 64 BFS jobs following the Poisson process [20] with different λ; and (b) when the starting vertices (but the number of hops between the starting vertices of different jobs are controlled increasing from 1 to 8) are randomly selected for the 64 BFS jobs, respectively.

are sent by the jobs to synchronize the same vertex, many graph data, e.g., the same vertex ID, have to be duplicated in these messages [17, 19, 44, 62]. In the experiments, the extra cost (i.e., the packet headers and redundant graph data) occupies up to 56.5% of the total communication volume.

2.2 Motivation

Figures 2(c) and (d) show the data access statistics of the jobs in the above experiments. We have two findings. *First, as depicted in Figure 2(c), a large ratio (e.g., more than 76%) of the graph data to be processed by CGP jobs (e.g., more than four jobs) are the same. Second, as depicted in Figure 2(d), the number of access times to different chunks is skewed. In each hour, the average number of access times of each shared chunk is more than 89.4 times that of each unshared chunk, because the shared chunks are repeatedly accessed by many CGP jobs and many of them also need more iterations to converge than the unshared chunks.* It means that more than 76% of all accesses are allowed to be shared by five or more jobs, which is more than 8% gotten by D-Galois-M as depicted in Figure 2(a) due to the irregular access behavior of the CGP jobs. Besides, the sharing ratio of D-Galois-M becomes lower when the access behavior is more irregular. Figures 3(a) and (b) show the impact of arrival rate and starting vertices of 64 BFS jobs on the sharing of accesses on D-Galois-M for uk-union. In Figure 3(a), when the interval between the starting time of the jobs becomes longer (i.e., smaller value of λ), the accesses are shared by fewer jobs. In Figure 3(b), when the starting vertices of the jobs are farther away from each other (i.e., the number of hops is larger), the accesses are shared by fewer jobs, although 60.2%–85.8% of the accessed graph data are the same for all BFS jobs due to graphs' power-law property. *These findings motivate us to develop GraphTune to fully exploit the hidden similar data access patterns among CGP jobs and coordinate the execution of multiple jobs, thereby efficiently consolidating and sharing the data accesses of different jobs.*

We still use the example of Figure 1 to illustrate our basic idea in coordinating the execution of various jobs. In fact, an inactive vertex, e.g., v_1, needs to be processed by a job only if its precursors, e.g., v_0, propagate their new states to it. Thus, in this example, we can generate a topology order for the vertices on host 0 as v_0, v_1, v_2, v_3, v_9, and v_4. We then coordinate *BFS 1* and *BFS 2* to process the vertices following this order. Since the starting vertex of *BFS 1* is v_0, v_0 is handled first by *BFS 1*, and then v_1 is activated. After v_1 is handled by *BFS 1*, v_2 and v_3 are activated. After v_2 is processed by *BFS 1*, v_3 then can be appointed to be processed by *BFS 1* and *BFS 2* at the same time. It allows these two jobs to share the access to v_3. Similarly, v_9 and v_4 can be handled simultaneously by these two jobs, also allowing to share their accesses. Moreover, when v_3 is concurrently handled by them, the state of mirror vertex v_{15} will be updated by the two jobs. Thus, the state synchronization of v_{15} between host 0 and host 2 are triggered by these two jobs at the same time. Likewise, the communications associated with v_4, v_5, and v_9 are also performed by these two jobs concurrently. In this way, there are more opportunities for coalesced communication for these two jobs.

3 OVERVIEW OF GRAPHTUNE

We propose GraphTune to address the above-mentioned limitations and opportunities in CGP jobs. In GraphTune, a cross-iteration dependency graph is first generated to describe the topology of possible vertex state propagations between the chunks at different iterations. Then, it proposes a dependency-aware execution model to regularize the traversal paths of the chunks for CGP jobs along the topology order of the dependency graph such that the accesses to the same graph can be shared by more jobs. Next, several optimization strategies are designed to improve resource utilization and reduce communication cost. Some APIs are also provided, through which GraphTune can be integrated into existing distributed systems to transparently support high-throughput execution of the CGP jobs.

3.1 Dependency-Aware Synchronous Execution

According to our findings, the irregular data accesses can be transformed into more regular ones when the processing order of the graph data is tuned for the jobs based on the graph topology. Thus, in Figures 4(a) and (b), a cross-iteration dependency graph G' is first constructed for the chunks of existing systems to depict the dependencies between these chunks. Namely, G'=(V',E',W'), where V'=$\{C^m|C^m \in G\}$ and E'=$\{<C^m, C^n>|\ \exists\, C^m \in G \wedge \exists\, C^n \in G$ s.t. $<v_a, v_b> \in E \wedge v_a \in C^m \wedge v_b \in C^n\}$. C^m and C^n denote the *m*th and *n*th chunks of the graph G, respectively. W' is the set of weights associated with E', representing the number of edges between two chunks. Each directed edge $<C^m, C^n>$ means that C^m is a precursor of C^n, and the states of C^m will be propagated to C^n to activate its processing at the next iteration. After that, the topological order of the chunks designated in this dependency graph is expected to be used to guide the jobs to efficiently access and process these chunks along a regularized traversal path.

However, the dependencies between some chunks may constitute a cyclic path (e.g., $C^0 \rightarrow C^1 \rightarrow C^0$ in Figure 4(b)) such that the states of a chunk will be propagated back to this chunk itself. To generate the topological order for the chunks, a ***Directed Acyclic Graph*** (**DAG**) sketch [49] (described in Figure 4(c)) of the dependency graph G' is generated by contracting ***Strongly Connected Components*** (**SCCs**), i.e., the components form a cyclic path, of G' into vertices.

Each *SCC-vertex*, e.g., SCC_x, of the DAG is a set of chunks, and each edge of the DAG represents the state propagation topology between two SCC-vertices, which in turn describes the topological order between the chunks belonging to the SCC-vertices. Following this DAG's topological order, the active chunks (i.e., the chunks need to be handled) can be handled by the jobs more regularly, allowing the accesses to these chunks to be shared by these jobs. In detail, the chunks are dispatched to be processed along the topological order of the SCC-vertices in the DAG. The chunks contained in the same SCC-vertex are directly assigned with an order, e.g., the order of them when traversing them according to their dependencies in a breadth-first order. Then, for each dispatched chunk, all related jobs (i.e., the ones need to process this chunk) are triggered to synchronously process it, reducing the storage and access overhead corresponding to this chunk. The next chunk is assigned to be handled only when the current chunk has been processed by all related jobs. Note that the active chunks belonging to independent SCC-vertices can be dispatched in parallel.

Figure 5 compares the data access patterns of existing solutions and our model by taking the execution of the two jobs in Figure 1 as an example, where *BFS* 1 is assumed to be submitted at the end of the first iteration of *BFS* 2. Existing solutions exploit the similar data access patterns locally

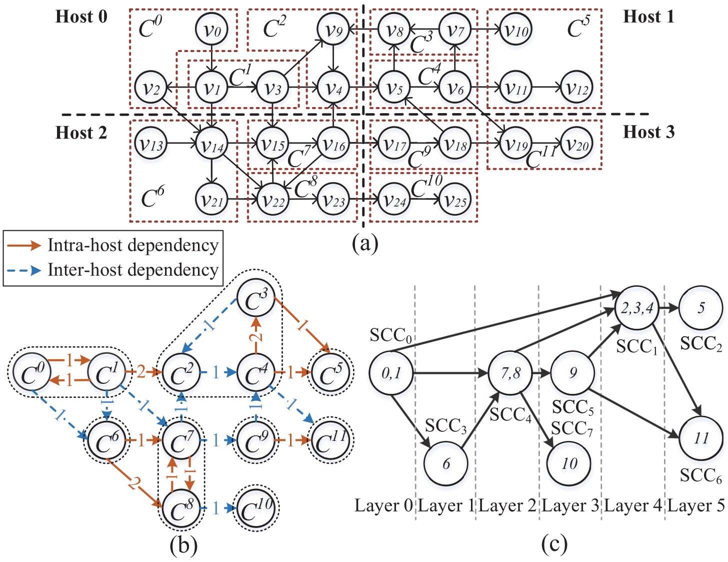

Host 0
Host 1
Host 2
Host 3
C^0 C^1 C^2 C^3 C^4 C^5 C^6 C^7 C^8 C^9 C^{10} C^{11}
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8 v_9 v_{10} v_{11} v_{12} v_{13} v_{14} v_{15} v_{16} v_{17} v_{18} v_{19} v_{20} v_{21} v_{22} v_{23} v_{24} v_{25}
(a)
Intra-host dependency
Inter-host dependency
(b)
0,1 2,3,4 5 7,8 9 6 10 11
SCC_0 SCC_1 SCC_2 SCC_3 SCC_4 SCC_5 SCC_7 SCC_6
Layer 0 Layer 1 Layer 2 Layer 3 Layer 4 Layer 5
(c)

Fig. 4. Generation of dependency graph: (a) the graph is divided into 12 chunks; (b) the state propagation dependencies between the chunks; (c) the DAG of the dependency graph, and (0,1) represents that the *SCC-vertex* consists of C^0 and C^1.

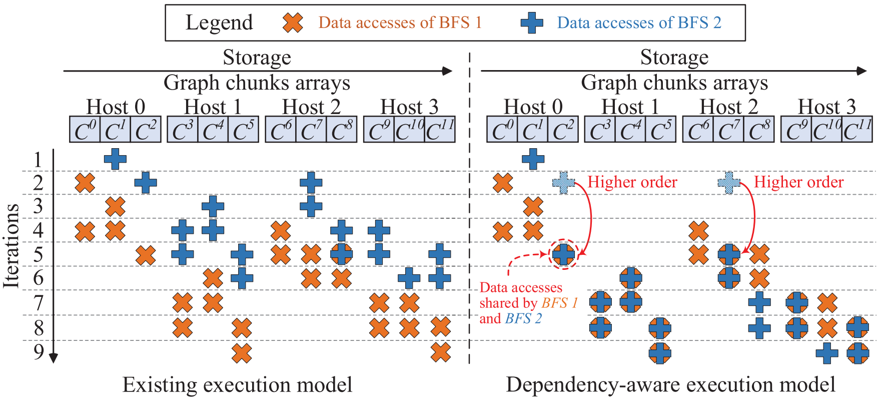


Fig. 5. Data access patterns of the CGP jobs on the existing execution model (left) and our dependency-aware execution model (right).

at each iteration. Thus, only a small portion of chunks' accesses (e.g., C^8 at the fifth iteration) are shared by these jobs. In our model, the active chunks are handled based on the topological order. For example, at the second iteration, the active chunks consist of C^0, C^2, and C^7. Because the topological order of SCC_0 (with C^0) is lower than that of SCC_1 (with C^2) and SCC_4 (with C^7), C^0 is handled first by *BFS* 1 and then C^1 is activated. Similarly, C^0, C^1, and C^6 are handled at the third and forth iterations, because SCC_0 (with C^0 and C^1) and SCC_3 (with C^6) own lower topological order than SCC_1 (with C^2) and SCC_4 (with C^7). After that, C^2 and C^7 are activated by *BFS* 1. Hence, the accesses to C^2 and C^7 can be shared by *BFS* 1 and *BFS* 2 at the fifth iteration. It shows that more CGP jobs can directly retrieve the shared chunks from the LLC, enabling more accesses to be shared by the CGP jobs than existing solutions.

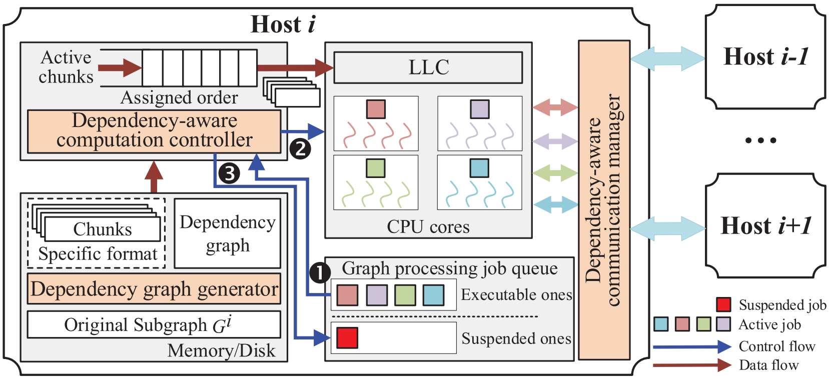


Fig. 6. Architecture of GraphTune.

3.2 System Architecture

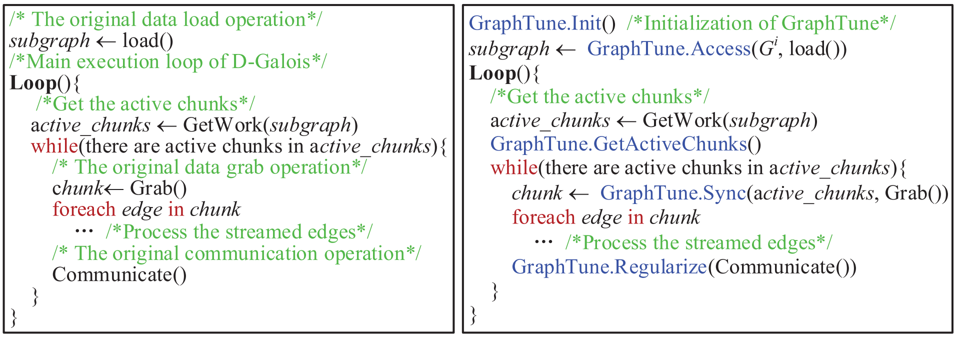
The architecture of GraphTune is depicted in Figure 6. Original graph is partitioned into several subgraphs, i.e., $G = \cup_i G^i$. GraphTune on each host maintains a subgraph, e.g., G^i, and has the following three main components.

Dependency Graph Generator. Graph formats and partition strategies usually vary on different distributed graph processing systems [17, 19, 28, 36, 44, 62]. Thus, before starting the execution of an existing system, the subgraphs of the original graph stored in GraphTune are first converted to the graph format specific to this system (such as the edge list format for Chaos, and the CSR/CSC format for D-Galois, Gemini, and PowerGraph) using *Convert*(). After that, the subgraphs are divided into chunks by the existing system for parallel processing using its own partitioning scheme. A dependency graph is created to describe the dependencies between these chunks. A DAG sketch is also generated for this dependency graph.

Dependency-aware Computation Controller. It regularizes the processing behavior of the CGP jobs. The subgraph G^i is loaded into the memory by GraphTune and is allowed to be shared by all related jobs. At execution time, for each job, it traces the chunks (i.e., the chunks to be processed by this job in the next iteration) activated in current iteration. Thus, at the end of the current iteration, it can obtain the set of jobs to handle a chunk in the next iteration. This information about the chunks on each host is maintained in its local table, where each entry contains a list of the IDs of the jobs to handle the corresponding chunk. Then, the jobs synchronously grab the chunks to be handled by them. In detail, the processing order of the chunks is assigned based on the dependency graph of chunks. Each chunk is loaded into the LLC for the related jobs by one of them (step ❶), and is also allowed to be shared by these jobs. After that, it activates these jobs to concurrently process this chunk (step ❷). The other non-related jobs are suspended and wait for their chunks to be grabbed (step ❸).

Dependency-aware Communication Manager. It assembles and disassembles the messages on each host to eliminate the problems of small messages and redundant communication cost. Specifically, it tries to ensure coalesced communication for as many CGP jobs as possible based on the dependency graph of the chunks handled by them.

Programming APIs. To use GraphTune, the user only requires to simply insert the provided APIs (Table 1) into existing distributed graph processing systems and no modification is needed for graph applications. Figure 7 shows how to integrate a distributed system D-Galois with GraphTune using these APIs. *Init*() is used to initialize GraphTune. To efficiently load the shared graph data, *Access()* replaces the original data load operation of existing system. *Sync()* replaces its original



```
/* The original data load operation*/
subgraph ← load()
/*Main execution loop of D-Galois*/
Loop(){
    /*Get the active chunks*/
    active_chunks ← GetWork(subgraph)
    while(there are active chunks in active_chunks){
        /* The original data grab operation*/
        chunk← Grab()
        foreach edge in chunk
            ··· /*Process the streamed edges*/
        /* The original communication operation*/
        Communicate()
    }
}
```

(a) Pseudocode of D-Galois

```
GraphTune.Init() /*Initialization of GraphTune*/
subgraph ← GraphTune.Access(G^i, load())
Loop(){
    /*Get the active chunks*/
    active_chunks ← GetWork(subgraph)
    GraphTune.GetActiveChunks()
    while(there are active chunks in active_chunks){
        chunk ← GraphTune.Sync(active_chunks, Grab())
        foreach edge in chunk
            ··· /*Process the streamed edges*/
        GraphTune.Regularize(Communicate())
    }
}
```

(b) Pseudocode of D-Galois integrated with GraphTune

Fig. 7. Illustration of integrating GraphTune into D-Galois.

Table 1. Programming Interfaces of GraphTune

APIs	Description
Init()	Initializing GraphTune
Convert()	Converting original graph to specific format
GetActiveChunks()	Obtaining active chunks in each iteration
Access()/*Sync*()	Loading the subgraph and synchronously grabbing its shared chunks for processing
Regularize()	Assembling and disassembling the communication messages

grab operation for synchronous processing of the chunks. *Regularize()* replaces its communication operation to achieve the optimized communication. Note that, to obtain the active chunks before each iteration, *GetActiveChunks()* is also provided, because this operation is used by some systems (e.g., D-Galois) to skip the processing of inactive chunks.

4 IMPLEMENTATION OF GRAPHTUNE

4.1 Generation of Dependency Graph

Generation of Dependency Graph. GraphTune generates the dependency graph in parallel. Each host first gets the dependencies between the chunks of its subgraph G^i. As shown in Algorithm 1, for each chunk C^m in G^i, it traverses its edges to collect the set of their source and destination vertices, i.e., src_set and dst_set (lines 2-3). Then, it stores src_set and dst_set into Set^i, where $Set^i[m]$ corresponds to the sets of the chunk C^m (line 4). Next, it obtains the intersection of src_set and dst_set between C^m and the set of chunks (i.e., C^i_*) which have collected their src_set and dst_set (line 5). If the intersection of the dst_set of C^m and the src_set of C^n is not empty (line 6), it creates a directed edge, i.e., $<C^m, C^n>$ (line 7). The number of edges between them is regarded as this edge's weight (line 8). For example, v_3's new state in C^1 is propagated to v_4 and v_9 in C^2 along the edges $<v_3, v_4>$ and $<v_3, v_9>$. Thus, the weight of the edge $<C^1, C^2>$ is 2. Similar operations are performed on the opposite direction (lines 10–13). When all dependencies in each subgraph have been created, the dependency graph G' is finally generated by merging these dependencies. Meanwhile, as shown in Figure 4(b), the dependencies between the chunks on the same host are marked as *intra-host dependencies*, while the others are marked as *inter-host dependencies*, so as to guide the message batching (discussed in Section 4.4). The DAG sketch can be generated for this dependency graph using Tarjan algorithm [49]. Note that, when the graph has changes, i.e., the vertices/edges in some chunks are added/deleted, their dependencies are also updated, and then a new DAG will be constructed to obtain their new topological order.

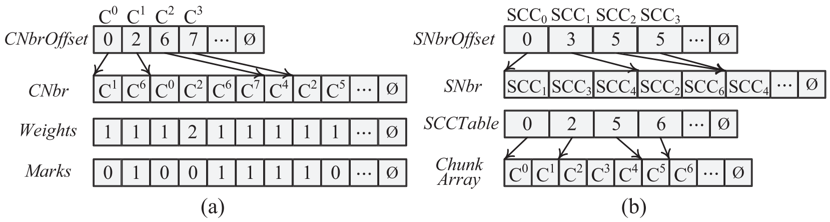


Fig. 8. Representation of (a) the dependency graph and (b) its DAG sketch for the example graph shown in Figure 4.

ALGORITHM 1: Dependency Graph Generation in GraphTune

```
function GENERATE($C^m$, $Set^i$, $E'$, $W'$)
    $src\_set \leftarrow$ CollectSourceVertices($C^m$)
    $dst\_set \leftarrow$ CollectDestinationVertices($C^m$)
    $Set^i[m]$.Store($src\_set$, $dst\_set$)
    for each chunk $C^n$ (excepts $C^m$) in $C^i_*$ do
        if $Set^i[m].dst\_set \cap Set^i[n].src\_set \neq \emptyset$ then
            $E'$.CreateEdge(<$C^m$, $C^n$>)
            $W'$.<$C^m$, $C^n$> $\leftarrow$ GetWeight($C^m$, $C^n$)
        end if
        if $Set^i[n].dst\_set \cap Set^i[m].src\_set \neq \emptyset$ then
            $E'$.CreateEdge(<$C^n$, $C^m$>)
            $W'$.<$C^n$, $C^m$> $\leftarrow$ GetWeight($C^n$, $C^m$)
        end if
    end for
end function
```

Storage of Dependency Graph. To efficiently store the dependency graph (Figure 8(a)), an array *CNbrOffset* is employed to store the offset of the beginning and the end of the neighbors for each chunk, while *CNbr* stores all neighbors for them. Moreover, two arrays, i.e., *Weights* and *Marks*, are used to store the *weight* and the *mark* of each dependency, where the value 0 or 1 of a *mark* indicates the related dependency is an intra-host one or inter-host one. To store the DAG sketch (Figure 8(b)), *SNbrOffset* stores the offset of the beginning and the end of the neighbors for each SCC, and *SNbr* stores all neighbors for them. *SCCTable* and *ChunkArray* are employed to record the set of chunks contained in each SCC-vertex.

4.2 Dependency-Aware Computation

To make the jobs efficiently share their data accesses using our approach, several challenges need to be tackled. First, as shown in Figure 2(d), because of the power-law property [19], a few chunks may need to be frequently accessed by more jobs than the others and many chunks may also depend on such a small set of chunks. When the processing of these important chunks is delayed, many jobs and many chunks cannot be activated, inducing low ratio of shared data accesses. Second, the same chunk's computational load is usually skewed in various jobs due to the difference in computing complexity and the different number of active vertices. It may incur high synchronization cost when synchronizing the execution of these jobs. We now discuss how to address these challenges.

ALGORITHM 2: Iteration Number Prediction in GraphTune

```
1: function Predict($C^{root}$, $SCC_x$, $I$)
2:     Set the chunk $C^{root}$ to be visited and $I^{root}$ to zero
3:     while $SCC_x$ has unvisited chunks do
4:         for each visited chunk $C^m$ in $SCC_x$ do
5:             for each successor $C^n$ of $C^m$ in $SCC_x$ do
6:                 if $C^n$ is unvisited then
7:                     $I^n \leftarrow I^m + 1$
8:                     Set the chunk $C^n$ to visited
9:                 end if
10:            end for
11:        end for
12:    end while
13: end function
```

Mining of Topological Order. The DAG sketch is first divided into layers. Each SCC-vertex is assigned with a layer number as shown in Figure 4(c), so that the SCC-vertices in a layer can only be activated by the SCC-vertices at the lower layer. To give an order for each chunk in the same SCC-vertex, it performs the propagation operation (e.g., breadth-first search [14]) to follow the dependencies between these chunks, and approximately predicts the iteration number to update this chunk. In Algorithm 2, for each SCC-vertex, one of its chunks (i.e., C^{root}) is randomly selected as the root and C^{root}'s iteration number is set to zero (line 2). Then, it repeatedly marks the iteration numbers (e.g., I^n) of the successors (e.g., C^n) of each visited chunk C^m, and the marked chunks are set to be visited (lines 5–8). When all chunks have been visited, the order can be obtained. Specifically, the chunks of the SCC-vertex with lower layer number is assigned with a lower topological order. Each chunk in the same SCC-vertex is assigned with a lower order as its iteration number is lower.

Effective Processing Scheduling of Chunks. To maximize the ratio of shared data accesses, each active chunk C^m is given a priority $Pri(C^m)$ to determine its processing order. As discussed above, the active chunks should be handled by the CGP jobs following the topological order. Besides, when an active chunk is to be handled by more CGP jobs, or has more out-going dependencies, or owns more active vertices, it also should be handled preferably, because it can activate more jobs and chunks. These rules can be expressed as follows:

$$Pri(C^m) = \theta \cdot N(J^m) \cdot N(C^m) \cdot \sum W'_+(C^m) - \frac{I^m}{N_x} - L^m, \tag{1}$$

where N_x is the number of chunks in SCC_x containing C^m and $N(J^m)$ is the number of jobs to handle C^m. $N(C^m)$, I^m ($0 \le I^m < N_x$), $\sum W'_+(C^m)$, and L^m are the number of active vertices, the iteration number, the total weights of out-going dependencies, and the layer number of C^m, respectively. $\theta = \frac{1}{N(J) \cdot N_{max}(C) \cdot \sum_{max} W'_+(C)}$ is the scaling factor to expect that the chunk with the lowest topological order is first handled. $N(J)$ is the number of CGP jobs. $N_{max}(C)$ and $\sum_{max} W'_+(C)$ are the maximum number of vertices and the maximum total weights of out-going dependencies of any chunk, respectively. The values of $\sum W'_+(C^m)$, I^m, N_x, $N_{max}(C)$, $\sum_{max} W'_+(C)$, L^m, and the initial values of $N(C^m)$, $N(J^m)$, $N(J)$ are obtained at the preprocessing stage, while $N(C^m)$, $N(J^m)$, and $N(J)$ are updated incrementally during the execution. The priority needs to be updated before the start of each iteration of any job. Then, the processing order of the chunks on each host can be gotten by sorting them based on their priorities.

Efficient Synchronous Processing of Chunks. When an active chunk is grabbed based on the above scheduled order, GraphTune triggers the related jobs to concurrently process this chunk. The next chunk is loaded when these jobs have completed the current chunk's processing. To efficiently synchronize the graph traversals of the jobs in this way, for the processing of each chunk, the computing resources will be unevenly assigned to these jobs based on their skewed computational loads, to make them finish this chunk's processing almost at the same time. For a job j, the load $O_j(C^m)$ to handle a chunk C^m can be calculated by $O_j(C^m) = \alpha_j \cdot E_j(C^m)$, where α_j is its average time to handle an edge at the previous iterations, and $E_j(C^m)$ represents the number of edges in C^m that need to be processed by the job j at the next iteration. In this way, the chunks can be regularly loaded into the LLC with low synchronization cost.

4.3 Efficient Inter-Host Load Balancing

During the execution, the load of the hosts may be skewed. However, existing load-balancing schemes [19, 44, 62] assume that each host's load is determined by the number of edges of its active chunks, and do not take into account the load differences between various jobs. As a result, they either face much runtime cost to balance load dynamically or experience imbalanced load between the hosts. For example, although the number of edges of active chunks on the hosts is skewed, the load may be even across these hosts.

Thus, based on the above profiled load, i.e., $O_j(C^m)$, of each job j to handle each chunk C^m, GraphTune first gets the total load of this chunk, i.e., $O(C^m)$, by summing the load of the related jobs for the processing of this chunk. After that, GraphTune incrementally balances the chunks between the hosts in advance based on the predicted total load of each chunk. To ensure data locality, for each overloaded host, its chunk with the lowest priority $Pri(C^m)$ is tried to be migrated to another underloaded host that contains the most number of chunks depended by this chunk. In detail, when migrating a chunk C^m in the overloaded host (i.e., $host_{over}$) to another underloaded host (i.e., $host_{under}$), $host_{over}$ needs to migrate the graph data (e.g., the graph structure data and the job-specific data of the CGP jobs) associated with C^m to $host_{under}$. Then, $host_{over}$ updates the vertices of other dependent chunks that have vertices in this migrated chunk C^m as the mirror vertices, while $host_{under}$ denotes the vertices in this migrated chunk C^m as the master vertices. Note that the chunk C^m is migrated only when its migration time is less than $O_s = O_H - MAX\{O_H - O(C^m), O_L + O(C^m)\}$. The migration time is the profiled average time to send this chunk at the previous iterations. O_H and O_L are the total load of the overloaded host and the underloaded host, respectively. After the migration, the related jobs then can handle these migrated chunks at the next iteration.

4.4 Dependency-Aware Communication

When the chunks are synchronously handled by multiple jobs with the above approach, it provides the opportunity for fully consolidated communication for these jobs to synchronize their vertex states. Specifically, the jobs also access their job-specific data in a similar way, because the shared chunks are regularly traversed by them along a common path. However, with existing solutions [17, 19, 44, 62], the jobs send their messages independently, incurring high communication cost.

Dependency-aware Message Batching. To exploit the above characteristic of the communications issued by the CGP jobs, GraphTune creates several queues to efficiently batch the messages for consolidated communications. Each queue (e.g., $Dest_i$) is used to buffer the messages to be sent to a destination host (e.g., host i), and $Dest_i$ is generated only when there is inter-host dependency from the local host to the host i, because the communication can be only triggered by the state propagation dependency between them. In Figure 9, two queues (i.e., $Dest_1$ and $Dest_2$) are created on host 0 to buffer the messages to be sent to hosts 1 and 2, respectively. Each queue is associated

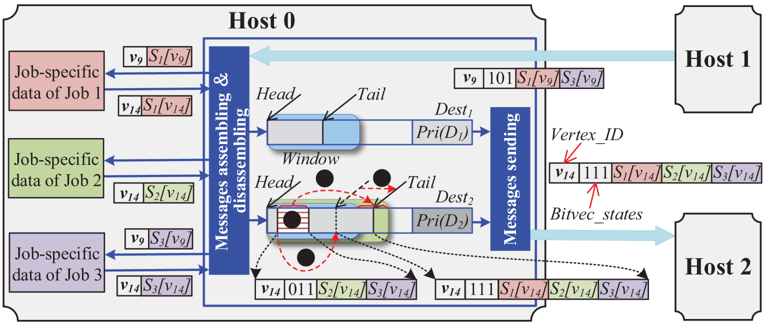

Host 0
Host 1
Host 2
Job-specific data of Job 1
Job-specific data of Job 2
Job-specific data of Job 3
v_9 $S_1[v_9]$
v_{14} $S_1[v_{14}]$
v_{14} $S_2[v_{14}]$
v_9 $S_3[v_9]$
v_{14} $S_3[v_{14}]$
Messages assembling & disassembling
Head
Tail
Window
$Dest_1$
$Pri(D_1)$
Head
Tail
$Dest_2$
$Pri(D_2)$
Messages sending
v_9 101 $S_1[v_9]$ $S_3[v_9]$
Vertex_ID
v_{14} 111 $S_1[v_{14}]$ $S_2[v_{14}]$ $S_3[v_{14}]$
Bitvec_states
v_{14} 011 $S_2[v_{14}]$ $S_3[v_{14}]$
v_{14} 111 $S_1[v_{14}]$ $S_2[v_{14}]$ $S_3[v_{14}]$

Fig. 9. Consolidated communication between hosts.

with two pointers, i.e., *Head* and *Tail*, along with a *Window* (which defines the packet size to be sent and is determined by MTU size of the network). When the CGP jobs have handled a chunk, their messages are pushed into the related queues based on their destination hosts and then the *Tail* pointer is updated. Once the buffered messages' size reaches the *Window* size, the operation of sending the full packet is triggered. The *Head* pointer is then updated when the send operation has been completed. Then, the messages of different jobs destined to the same host can be sent together in a batch. In this way, it not only reduces the cost of sending many packet headers for small messages but also provides the opportunity for effective coalesced communication as follows.

Assembling and Disassembling Messages. As discussed above, in GraphTune, multiple jobs usually propagate their new states regarding the same vertex to another host simultaneously. It allows us to efficiently assemble the messages of these jobs to a large one and also only keep one copy of the vertex ID for these messages for less communication cost. In detail, the messages are coalesced when there exist the messages sent by other jobs to the same vertex in the queue. For example, assume the messages for jobs 2 and 3 to update v_{14} have been assembled and buffered in $Dest_2$. Then, when $S_1[v_{14}]$ of job 1 is pushed into $Dest_2$, the messages of jobs 1, 2, and 3 are coalesced and the coalesced message is again stored in $Dest_2$ (step ❶), because they are destined to the same vertex v_{14}. The *Tail* of $Dest_2$ is also updated accordingly (step ❷). Next, the original message sent by jobs 2 and 3 to v_{14} is marked as empty to skip its sending (step ❸), and the marked message's size is added to the current *Window size* of $Dest_2$ (step ❹). When the coalesced message arrives at the destination host, it is disassembled to several messages for different jobs. Then, these jobs can utilize the network resources more efficiently. To distinguish vertex states of different jobs, each vertex state needs to be sent along with the related job ID. To spare the cost of sending the IDs of the jobs, GraphTune assembles different jobs' vertex states following a specific order. It uses a field-specific bit-vector (i.e., *Bitvec_states*) for each sending vertex to indicate the jobs that send these messages. In Figure 9, the messages for jobs 1, 2, 3 to update v_{14} are assembled to one larger message, where the states of different jobs for the same vertex are successively stored along with a bit-vector. In this way, it can distinguish different jobs' vertex states although the IDs of these jobs are not sent.

Dependency-aware Message Sending. During the execution, the volume of data to be transferred from one host to another, i.e., communication workload, are usually skewed for different queues because of the power-law property [19]. The queue with higher communication workload has the opportunity to coalesce more messages. Thus, this queue is expected to be delayed to be sent, aiming to allow more messages to be pushed into this queue to be fully coalesced. In practice, the inter-host dependency with higher weight usually incurs more messages, and the chunks with more active vertices or handled by more jobs would also generate more messages. Thus, each queue $Dest_i$ can be assigned with a priority $Pri(D_i)$ as follows:

$$\begin{cases} L_i(C^m) = \dfrac{N(C^m)}{|V_{C^m}|} \cdot N(J^m) \cdot \displaystyle\sum_{e \in E'_i(C^m)} W'(e) \\ Pri(D_i) = \dfrac{\sum_{C^m \in A^*} L_i(C^m)}{\sum_{C^m \in A} L_i(C^m)} \end{cases} \tag{2}$$

$L_i(C^m)$ is the predicted communication workload of a chunk C^m in the queue $Dest_i$. $N(C^m)$ and $|V_{C^m}|$ are the number of active vertices and the total number of vertices of C^m, respectively. $E'_i(C^m)$ is the set of inter-host dependencies from C^m to the chunks on the host i. $N(J^m)$ is the number of jobs to handle C^m. $W'(e)$ is the weight associated with e. A^* is the set of chunks that have buffered their messages in the queue $Dest_i$. A consists of the ones in A^* and the active chunks to push their messages into $Dest_i$. Each queue's priority is updated after processing each chunk or sending its packet. When the network is available, the packets generated in the queue with the highest priority are first sent, because less opportunity is left for it to coalesce messages than other queues.

5 EVALUATION

The platform is a cluster with 32 hosts interconnected by an Infiniband EDR network (with up to 100 Gbps bandwidth) and the MTU size is 2 KB. Each host contains a 2-way 8-core Intel Xeon E5-2670 CPUs (each CPU has 20 MB LLC), a 32 GB physical memory, and an 1 TB hard drive, running the Linux operation system with the kernel version 5.3.18. The program is compiled with cmake 2.8.3 and gcc 4.4.5.

Competitors. To evaluate GraphTune, we integrate it into four popular distributed graph processing systems, i.e., D-Galois (v6.0.0) [17], Gemini (v1.0) [62], Chaos (v1.0) [44], and PowerGraph (v2.2) [19], to run multiple CGP jobs. Due to space, we focus on presenting those on D-Galois because it is a cutting-edge distributed graph processing system and outperforms other solutions [19, 44, 62], and briefly discuss others in Section 5.6. In detail, the D-Galois integrated with GraphTune (called *D-Galois-T* in the experiments) is compared with three execution schemes of the original system, called *D-Galois-S*, *D-Galois-C*, and *D-Galois-M*. *D-Galois-S* sequentially handles the jobs, while *D-Galois-C* and *D-Galois-M* process the jobs concurrently. In *D-Galois-C*, each job runs independently, and the CGP jobs are managed by the operating system. In *D-Galois-M*, the CGP jobs share the underlying graph and are managed by GraphM (v1.0) [61]. It is the same for the experiments of Gemini, PowerGraph, and Chaos in Section 5.6. Besides, to qualitatively and quantitatively evaluate GraphTune, we also integrate it into a popular out-of-core graph processing system, i.e., GraphChi (v1.0) [30]. CGraph [60] is also evaluated to demonstrate the efficiency of GraphTune. Their performance is evaluated with their best-performance settings.

Datasets and Benchmarks. Five real-world graphs (see Table 2) are used. The jobs in the real trace from a real Chinese social network are periodically executed jobs and can be classified into two categories: *all-active algorithms* (all vertices are active at the beginning) and *non-all-active algorithms* (a subset of vertices are active at the beginning). The proportion of the former is about 22.2%, and that of the latter is 77.8%. Specifically, the all-active algorithms are the variants of PageRank [42], WCC [23], *k*-core [26], Label propagation [10], Louvain modularity [33], *k*-means [21], Graph coloring [25], MIS [8], Maximal matching [9], and Degree centrality [27], while the non-all-active algorithms are implemented based on ***Single Source Shortest Path* (SSSP)** [18, 37, 39] or BFS [3, 14, 41]. We also submit the jobs following this real trace. In detail, when the job in the trace is the all-active job, we submit the corresponding algorithm described above. When the job in the trace is implemented based on SSSP or BFS, we submit the SSSP or BFS with a randomly selected starting vertex. Note that the jobs are dynamically submitted along the time of the real trace until

Table 2. Graph Datasets Proprieties ($Dia._{max}$ Denotes the Maximum Diameter, D_{max} Represents the Max Degree, and $\overline{D}$ Represents the Average Degree)

Datasets	*#Vertices*	*#Edges*	Data sizes	$Dia._{max}$	D_{max}	$\overline{D}$
twitter [29]	41.7 M	1.5 B	17.5 GB	24	3,081,112	35.3
friendster [5]	65.6 M	1.8 B	22.7 GB	32	5,214	27.53
uk-2007 [4]	105.9 M	3.7 B	46.2 GB	82	15,402	35.31
uk-union [11]	133.6 M	5.5 B	68.3 GB	147	22,429	41.22
clueweb12 [4]	978.4 M	42.6 B	317 GB	498	7,447	43.51

Table 3. Preprocessing Time (in Seconds)

	twitter	friendster	uk-2007	uk-union	clueweb12
D-Galois	34.20	47.47	79.48	97.40	845.11
D-Galois-T	37.29	53.26	87.60	108.67	961.04

the specified number of jobs have been generated, instead of submitting all jobs at the same time. We run all benchmarks 10 times and the following results are the averaged values.

5.1 Preprocessing Overhead

D-Galois-T's preprocessing cost is shown in Table 3. As observed, D-Galois-T needs little time than the original system D-Galois to generate the dependency graph by traversing the graph once. The preprocessing time is increased 9%, 12.2%, 10.2%, 9.2%, and 13.7% for twitter, friendster, uk-2007, uk-union, and clueweb12, respectively. The extra storage cost for D-Galois-T is also small and occupies 3.6%–5.8% of the space required for the original graph, i.e., 847.2 MB (4.7%), 962.7 MB (4.1%), 1.68 GB (3.6%), 3.7 GB (5.4%), and 18.5 GB (5.8%) for twitter, friendster, uk-2007, uk-union, and clueweb12, respectively. GraphTune spares much redundant data access cost, although it requires this additional cost.

5.2 Performance Comparison

Figure 10 evaluates the execution time of 64 jobs submitted on various schemes normalized to that of D-Galois-C. As shown in Figure 10, the jobs run by D-Galois-T require shorter execution time than other schemes. For different graphs, compared with D-Galois-S, D-Galois-C, and D-Galois-M, the throughput improved by D-Galois-T is 4.1~7.9, 3.5~7.1, and 3.1~6.2 times, respectively, due to lower data access cost. We attribute such acceleration to more regular data access of D-Galois-T.

To verify the above discussion, we have broken down the total execution time into three parts: (1) the time to process graph data, (2) the time to transfer the data from memory to LLC, and (3) the communication time. As shown in Figure 11, D-Galois-T spends fewer data transfer time and communication time than other three schemes. On clueweb12, compared with D-Galois-S, D-Galois-C, and D-Galois-M, the data transfer time is reduced by D-Galois-T by 16.7, 15.2, and 10.9 times, and the communication time is spared by D-Galois-T by 7.3, 6.5, and 6.3 times, respectively. It is because the jobs on D-Galois-T can fully share the data accesses, which is represented in two-fold: (1) these jobs can traverse the common subgraph together, reducing redundant data transferring between the memory and the LLC; (2) the messages issued by the jobs are coalesced, reducing the network cost.

Figure 12 shows the volume of the graph data loaded into LLC when executing 64 jobs. Because of intense cache interference in D-Galois-C, more data are loaded into LLC by D-Galois-C than D-Galois-S. Meanwhile, we can observe that D-Galois-T always swaps less volume of data into LLC

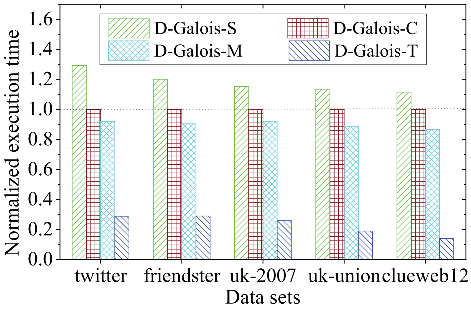


Fig. 10. Normalized execution time of the CGP jobs on various schemes.

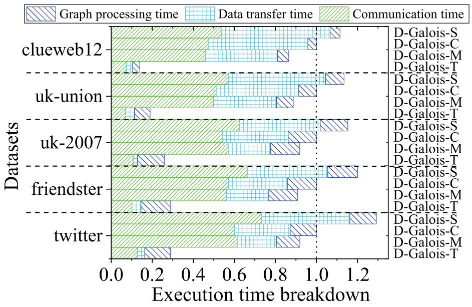


Fig. 11. Execution time breakdown of the CGP jobs on various schemes.

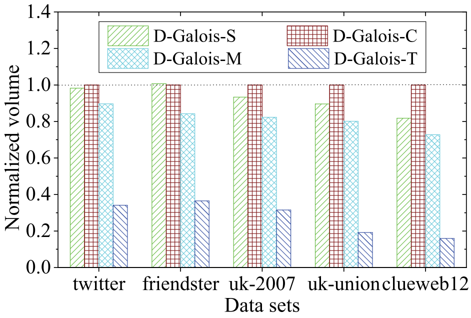


Fig. 12. Normalized volume of data loaded into the LLC on various schemes.

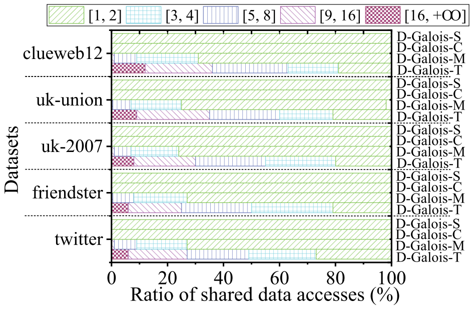


Fig. 13. Ratio of the shared data accesses on various schemes.

than others. It is because the loaded data are enabled to be shared by more CGP jobs on D-Galois-T via regularly streaming them into LLC.

Figure 13 depicts the percentage of data accesses to the same graph data amortized by various number of CGP jobs. D-Galois-T enables a higher ratio of accesses to be shared by more jobs than D-Galois-M. For example, when processing clueweb12, more than 63% of data accesses are shared by more than 4 jobs in D-Galois-T, while only 9% in D-Galois-M. It indicates that less data access cost and storage overhead is required by D-Galois-T to run the CGP jobs.

Figure 14 evaluates the total volume of communication when running the 64 jobs. Obviously, D-Galois-T needs the least volume. For clueweb12, D-Galois-T reduces the volume of D-Galois-M by 54.9%. It is because many small messages initiated by the CGP jobs are combined into large ones, which are further batched efficiently to reduce communication cost.

Figure 15 depicts the ratio of effective communication volume (after removing packet headers and redundant graph data) to the total volume for the 64 jobs. This figure shows that D-Galois-T obtains a higher ratio of effective communication volume compared with the other three schemes. It is because many small messages are merged and repetitive graph data are also removed in D-Galois-T. GraphTune can efficiently utilize the network resource in the case of multiple CGP jobs, potentially improving the system throughput.

5.3 Sensitivity Study

Figure 16 evaluates the job submission frequency's impact on GraphTune for clueweb12, where we submit the jobs following the Poisson process [20] with different values of the arrival rate λ.

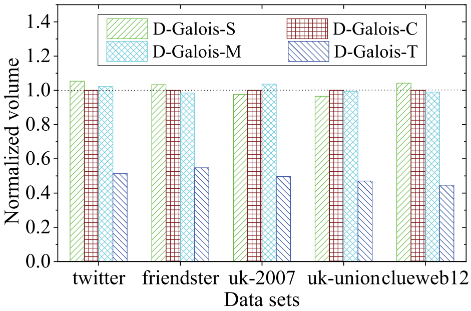


Fig. 14. Communication volume normalized to D-Galois-C.

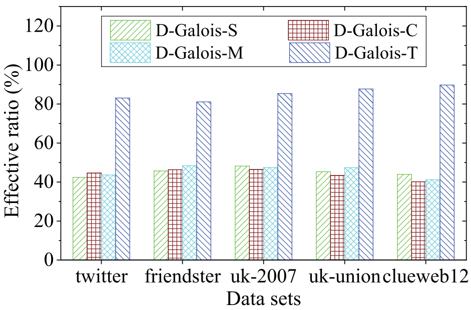


Fig. 15. Ratio of effective communication volume on various schemes.

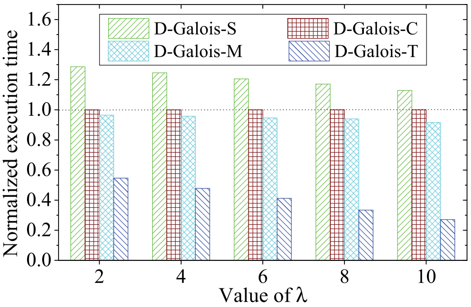


Fig. 16. Execution time normalized to that of D-Galois-C.

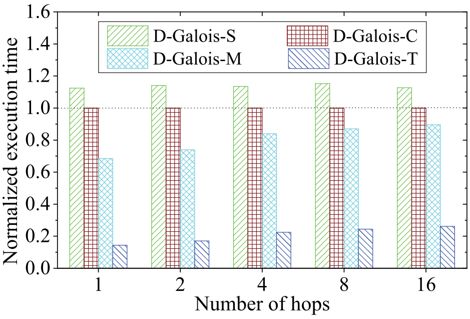


Fig. 17. Impact of the hops between the starting vertices on various schemes.

D-Galois-M and D-Galois-T achieve higher speedup as the jobs are submitted more frequently (i.e., larger λ). However, due to the regular data accesses in D-Galois-T, its performance is always more outstanding than others.

Figure 17 depicts the performance of 64 BFS jobs submitted on clueweb12 with various schemes, where the starting vertices of these BFS jobs are randomly selected but in the range of different number of hops. Under such circumstances, the intersections of the graph data to be handled by the BFS jobs occupy 84.2%–93.6% of the whole graph. Through regularizing their traversal paths, GraphTune enables the accesses to the common graph data to be fully amortized by these jobs. Thus, D-Galois-T improves the throughput by 4.3–7.8, 3.8–6.9, and 3.5–4.8 times in comparison with D-Galois-S, D-Galois-C, and D-Galois-M, respectively. Besides, D-Galois-T obtains much better performance than other schemes when the starting vertices of these jobs are further away from each other.

5.4 Evaluation of Scalability

Figure 18 evaluates the performance of different number of jobs on clueweb12. We observe that D-Galois-T obtains higher performance when submitting more CGP jobs. When there are 4, 8, 16, 32, and 64 jobs, D-Galois-T improves the throughput by 1.37, 1.75, 2.45, 3.71, and 6.21 times, respectively, in comparison with that of D-Galois-M. It is because the data accesses of more CGP jobs are regularized by GraphTune.

Figure 19 evaluates their scale-out scalability over uk-union by submitting the 64 jobs. We observe that D-Galois-T has better scalability than other schemes, because less communication cost

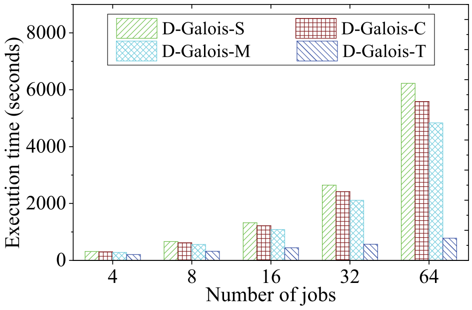


Fig. 18. Total execution time of different number of jobs on various schemes.

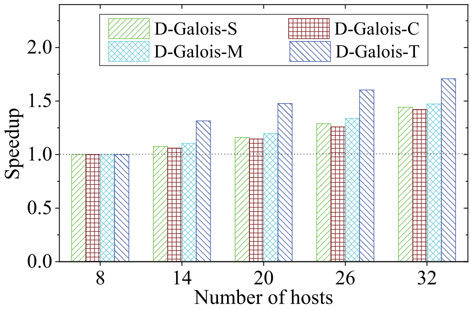


Fig. 19. Speedup on different number of hosts on various schemes.

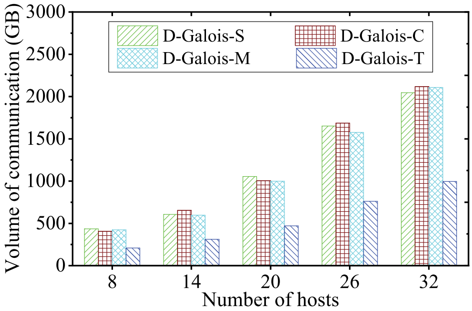


Fig. 20. Communication on different number of hosts over various schemes.

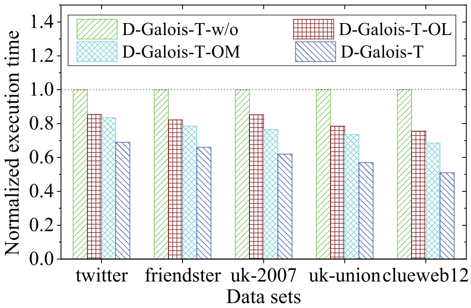


Fig. 21. Normalized execution time of the CGP jobs on various schemes.

is required by D-Galois-T than others. Figure 20 shows the total volume of communication when the 64 jobs are run on uk-union. It shows that, by merging small messages and eliminating redundant communication for multiple CGP jobs, GraphTune generates much less communication volume than other schemes when we increase the number of hosts.

5.5 Hierarchical Evaluation of GraphTune

We also evaluate the performance contributions from different optimization strategies of GraphTune when it is integrated with D-Galois. Figure 21 depicts the normalized execution time of those CGP jobs when running with several levels of optimizations. In D-Galois-T-w/o, the optimization of both load balancing and dependency-aware message sending of the CGP jobs are disabled, whereas the traversal paths of the jobs are regularized based on the dependency graph and the messages of these jobs are coalesced. The optimization strategies of load balancing and message sending are enabled in D-Galois-T-OL and D-Galois-T-OM, respectively. We can observe that D-Galois-T-OL achieves better performance than D-Galois-T-w/o because it efficiently balances the chunks among the hosts for multiple CGP jobs, achieving higher utilization of the hosts' resources. D-Galois-T-OM requires shorter execution time than D-Galois-T-w/o because the communication is further coalesced.

5.6 Integration with Other Systems

Table 4 depicts the execution time of the 64 jobs on other graph processing systems with different schemes. As shown in this table, they achieve high performance improvements after integrating

Table 4. Total Execution Time (in Seconds) of Other Graph Processing Systems (i.e., Gemini, PowerGraph, Chaos, GraphChi, and CGraph) Integrated with GraphTune, Where " – " Means That It Cannot Be Executed Due to Memory Errors and the Experiments of GraphChi Are Performed on a Host of the Used Cluster

	twitter	friendster	uk-2007	uk-union	clueweb12
Gemini-S	56	156	243	478	4,093
Gemini-C	47	138	225	446	3,839
Gemini-M	39	124	192	394	3,359
Gemini-T	9	33	42	59	394
PowerGraph-S	368	576	5,632	28,732	–
PowerGraph-C	332	444	4,612	26,612	–
PowerGraph-M	284	388	4,260	24,296	–
PowerGraph-T	82	105	537	2,256	–
Chaos-S	896	686	18,672	118,152	>1 week
Chaos-C	2,064	2,352	48,044	123,772	>1 week
Chaos-M	788	597	15,148	99,316	>1 week
Chaos-T	172	139	3,089	10,456	82,524
GraphChi-S	38,652	52,365	81,643	132,583	>1 week
GraphChi-C	12,753	19,347	27,734	41,491	>1 week
GraphChi-M	8,241	11,573	18,324	23,863	296,826
GraphChi-T	2,137	2,936	4,708	5,532	42,734
CGraph	32	79	154	269	2,042

with GraphTune. Because different ratios of total execution time are consumed by various systems to access graph data, different performance improvements are obtained by them after using GraphTune, respectively. Generally, more prominent improvements can be obtained by GraphTune, when the original system has a higher ratio. Besides, when integrating GraphTune into the out-of-core graph processing system, i.e., GraphChi, it also gains better performance than other schemes. This is because GraphTune enables the graph data to be regularly loaded from the disk to memory and from memory to cache, ensuring much lower data access cost.

6 RELATED WORK

Single-machine Graph Processing. GridGraph [63] designs a hierarchical partitioning method for better data locality. Galois [40] implements a graph DSLs to support application-specific priority-based scheduling of fine-grained tasks. FBSGraph [59], HotGraph [58], and Wonderland [55] are designed to achieve fast state propagation. GraphBolt [38] and LUMOS [50] track dependencies between vertices' states to proactively propagate vertex states while ensuring synchronous processing semantics. Ascetic [48] exploits data reuse across iterations to reduce data transfer between CPU main memory and GPU memory. Different from them, GraphTune constructs a dependency graph of the chunks to predict the traversal path of them for various jobs, generates a DAG based on this dependency graph aiming to maximize the sharing ratio of the accessed graph data, and adopts a "synchronous execution" scheme to regularize the accessing/processing behavior of the CGP jobs.

Distributed Graph Processing. To ensure balanced load, PowerGraph [19] uses a vertex-cutting method. Chaos [44] scales out X-Stream [45] on multiple machines. Gemini [62] adopts a dual update propagation model and a chunk-based partitioning scheme to build scalability on top of efficiency. D-Galois [17] is a Galois system interfaced with the Gluon runtime that optimizes the communication for distributed graph processing. DH-Falcon [16] presents a graph domain-specific language to express graph algorithms targeting heterogeneous (CPU and GPU) distributed platforms. Nevertheless, these approaches are mostly proposed for a single graph processing job, suffering from redundant data access and storage cost when executing multiple CGP jobs on the common underlying graph.

Concurrent Graph Processing. Seraph [52, 53] decouples the data model of graph processing to reduce memory consumption. CGraph [57, 60] uses a programming model to enable the same graph data to be shared by CGP jobs for less data access overhead. GraphM [31, 61] is a recently developed graph storage system that enables existing graph processing systems to efficiently handle multiple CGP jobs with lower data storage and access cost through sharing the accesses to the same graph data. However, they mainly focus on reducing the data access overhead on a single machine. Importantly, they cannot fully exploit the similar data access patterns among the CGP jobs because there is no consideration about the irregular data accesses between the jobs. In comparison with them, GraphTune can transparently and significantly shorten the execution time of the CGP jobs on existing distributed graph processing systems by maximizing the sharing ratio of data accesses and also minimizing the communication cost.

Concurrent Graph Queries. Some systems [12, 13, 47] are also developed for concurrent graph queries. Wukong+S [56] tries to support real-time consistent queries over fast-evolving data. Wukong+G [51] achieves fast concurrent RDF queries through exploiting the hybrid parallelism of GPU and CPU. Congra [43] adopts a dynamic scheduler to manage concurrent queries for resource efficiency. However, they cannot efficiently handle the CGP jobs due to their diverse traversal characteristics. Usually, they only handle a small subgraph for exactly once, whereas iterative graph processing traverses the entire graph frequently.

7 CONCLUSION

In this work, we investigate data access characteristics of CGP jobs and find that their irregular data access behaviors incur low throughput. We then propose a novel dependency-aware synchronous execution model to fully transform the irregular data accesses of CGP jobs into more regular ones, opening new opportunity of data access sharing on each host and coalesced communication between hosts. Based on this model, we design GraphTune, which can integrate with existing distributed graph processing systems to significantly improve the throughput of CGP jobs over them by up to 12.4 times.

REFERENCES

[1] Facebook. 2023. Retrieved 21 March 2023 from http://www.facebook.com/.

[2] Google. 2023. Retrieved 21 March 2023 from http://www.google.com/.

[3] Graph500. 2023. Retrieved 21 March 2023 from http://graph500.org/?page_id=12#sec-6_1.com/.

[4] LAW. 2023. Retrieved 21 March 2023 from http://law.di.unimi.it/datasets.php.

[5] SNAP. 2023. Retrieved 21 March 2023 from http://snap.stanford.edu/data/index.html.

[6] Tencent. 2023. Retrieved 21 March 2023 from https://www.tencent.com/en-us/about.html/.

[7] Twitter. 2023. Retrieved 21 March 2023 from https://www.twitter.com/.

[8] Ioana Oriana Bercea, Navin Goyal, David G. Harris, and Aravind Srinivasan. 2016. On computing maximal independent sets of hypergraphs in parallel. *ACM Transactions on Parallel Computing* 3, 1 (2016), 5:1–5:13.

[9] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. 2012. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures.* 308–317.

[10] Paolo Boldi, Marco Rosa, Massimo Santini, and Sebastiano Vigna. 2011. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web.* 587–596.

[11] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. 2008. A large time-aware graph. *ACM SIGIR Forum* 42, 2 (2008), 33–38.

[12] Mihaela A. Bornea, Julian Dolby, Anastasios Kementsietsidis, Kavitha Srinivas, Patrick Dantressangle, Octavian Udrea, and Bishwaranjan Bhattacharjee. 2013. Building an efficient RDF store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 121–132.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*. 49–60.

[14] Aydin Buluç and Kamesh Madduri. 2011. Parallel breadth-first search on distributed memory systems. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[15] Hongzheng Chen, Minghua Shen, Nong Xiao, and Yutong Lu. 2021. Krill: A compiler and runtime system for concurrent graph processing. In *Proceedings of the 2021 International Conference for High Performance Computing, Networking, Storage and Analysis*. 51:1–51:16.

[16] Unnikrishnan Cheramangalath, Rupesh Nasre, and Y. N. Srikant. 2017. DH-Falcon: A language for large-scale graph processing on distributed heterogeneous systems. In *Proceedings of the 2017 IEEE International Conference on Cluster Computing*. 439–450.

[17] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 752–768.

[18] Jack Edmonds and Richard M. Karp. 1972. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM* 19, 2 (1972), 248–264.

[19] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 17–30.

[20] James Goulding, Simon Preston, and Gavin Smith. 2016. Event series prediction via non-homogeneous poisson process modelling. In *Proceedings of the 16th IEEE International Conference on Data Mining*. 161–170.

[21] Shalmoli Gupta, Ravi Kumar, Kefu Lu, Benjamin Moseley, and Sergei Vassilvitskii. 2017. Local search methods for k-means with outliers. *Proceedings of the VLDB Endowment* 10, 7 (2017), 757–768.

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

[23] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. 2017. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition* 70, C (2017), 25–43.

[24] Hai Jin, Hao Qi, Jin Zhao, Xinyu Jiang, Yu Huang, Chuangyi Gui, Qinggang Wang, Xinyang Shen, Yi Zhang, Ao Hu, Dan Chen, Chaoqiang Liu, Haifeng Liu, Haiheng He, Xiangyu Ye, Runze Wang, Jingrui Yuan, Pengcheng Yao, Yu Zhang, Long Zheng, and Xiaofei Liao. 2022. Software systems implementation and domain-specific architectures towards graph analytics. *Intelligent Computing* 2022 (2022), 1–32.

[25] Mark T. Jones and Paul E. Plassmann. 1993. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing* 14, 3 (1993), 654–669.

[26] Humayun Kabir and Kamesh Madduri. 2017. Parallel k-Core decomposition on multicore platforms. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops*. 1482–1491.

[27] U. Kang, Spiros Papadimitriou, Jimeng Sun, and Hanghang Tong. 2011. Centralities in large networks: Algorithms and observations. In *Proceedings of the 11th SIAM International Conference on Data Mining*. 119–130.

[28] Seongyun Ko and Wook-Shin Han. 2018. TurboGraph++: A scalable and fast graph analytics system. In *Proceedings of the 2018 International Conference on Management of Data*. 395–410.

[29] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *Proceedings of the 19th International Conference on World Wide Web*. 591–600.

[30] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. 31–46.

[31] Xiaofei Liao, Jin Zhao, Yu Zhang, Bingsheng He, Ligang He, Hai Jin, and Lin Gu. 2022. A structure-aware storage optimization for out-of-core concurrent graph processing. *IEEE Transactions on Computers* 71, 7 (2022), 1612–1625.

[32] Xianzhu Liu, Zhijian Ji, and Ting Hou. 2019. Graph partitions and the controllability of directed signed networks. *SCIENCE CHINA Information Sciences* 62, 4 (2019), 42202:1–42202:11.

[33] Hao Lu, Mahantesh Halappanavar, Ananth Kalyanaraman, and Sutanay Choudhury. 2014. Parallel heuristics for scalable community detection. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 1374–1385.

[34] Patrick MacArthur, Qian Liu, Robert D. Russell, Fabrice Mizero, Malathi Veeraraghavan, and John M. Dennis. 2017. An integrated tutorial on InfiniBand, verbs, and MPI. *IEEE Communications Surveys & Tutorials* 19, 4 (2017), 2894–2926.

[35] German Maglione Mathey, Pedro Yébenes, Jesús Escudero-Sahuquillo, Pedro Javier García, Francisco J. Quiles, and Eitan Zahavi. 2018. Scalable deadlock-free deterministic minimal-path routing engine for infiniband-based dragonfly networks. *IEEE Transactions on Parallel and Distributed Systems* 29, 1 (2018), 183–197.

[36] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 135–146.

[37] Abdullah-Al Mamun and Sanguthevar Rajasekaran. 2016. An efficient minimum spanning tree algorithm. In *Proceedings of the 2016 IEEE Symposium on Computers and Communication*. 1047–1052.

[38] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the 14th EuroSys Conference*. 25:1–25:16.

[39] Ulrich Meyer. 2001. Single-source shortest-paths on arbitrary directed graphs in linear average-case time. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*. 797–806.

[40] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 456–471.

[41] Phuong Nguyen, Paolo Tomeo, Tommaso Di Noia, and Eugenio Di Sciascio. 2015. An evaluation of SimRank and personalized PageRank to build a recommender system for the web of data. In *Proceedings of the 24th International Conference on World Wide Web Companion*. 1477–1482.

[42] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1998. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report. Stanford Digital Library Technologies Project.

[43] Peitian Pan and Chao Li. 2017. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *Proceedings of the 2017 International Conference on Computer Design*. 217–224.

[44] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. 2015. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. 410–424.

[45] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 472–488.

[46] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy J. Lin. 2016. GraphJet: Real-time content recommendations at Twitter. *Proceedings of the VLDB Endowment* 9, 13 (2016), 1281–1292.

[47] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and concurrent RDF queries with RDMA-based distributed graph exploration. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 317–332.

[48] Ruiqi Tang, Ziyi Zhao, Kailun Wang, Xiaoli Gong, Jin Zhang, Wenwen Wang, and Pen-Chung Yew. 2021. Ascetic: Enhancing cross-iterations data efficiency in out-of-memory graph processing on GPUs. In *Proceedings of the 50th International Conference on Parallel Processing*. 41:1–41:10.

[49] Robert Endre Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1, 2 (1972), 146–160.

[50] Keval Vora. 2019. LUMOS: Dependency-driven disk-based graph processing. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 429–442.

[51] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. 2018. Fast and concurrent RDF queries using RDMA-assisted GPU graph exploration. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 651–664.

[52] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. 2017. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers* 66, 5 (2017), 876–890.

[53] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. 2014. Seraph: An efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. 227–238.

[54] Ke Yang, Mingxing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. KnightKing: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 524–537.

[55] Mingxing Zhang, Yongwei Wu, Youwei Zhuo, Xuehai Qian, Chengying Huan, and Kang Chen. 2018. Wonderland: A novel abstraction-based out-of-core graph processing system. In *Proceedings of the 23th International Conference on Architectural Support for Programming Languages and Operating Systems*. 608–621.

[56] Yunhao Zhang, Rong Chen, and Haibo Chen. 2017. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. 614–630.

[57] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Ligang He, Bingsheng He, and Haikun Liu. 2018. CGraph: A correlations-aware approach for efficient concurrent iterative graph processing. In *Proceedings of the 2018 USENIX Annual Technical Conference*. 441–452.

[58] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, Guang Tan, and Bing Bing Zhou. 2017. HotGraph: Efficient asynchronous processing for real-world graphs. *IEEE Transactions on Computers* 66, 5 (2017), 799–809.

[59] Yu Zhang, Xiaofei Liao, Hai Jin, Lin Gu, and Bing Bing Zhou. 2018. FBSGraph: Accelerating asynchronous graph processing via forward and backward sweeping. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 895–907.

[60] Yu Zhang, Jin Zhao, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, and Ligang He. 2019. CGraph: A distributed storage and processing system for concurrent iterative graph analysis jobs. *ACM Transactions on Storage* 15, 2 (2019), 10:1–10:26.

[61] Jin Zhao, Yu Zhang, Xiaofei Liao, Ligang He, Bingsheng He, Hai Jin, Haikun Liu, and Yicheng Chen. 2019. GraphM: An efficient storage system for high throughput of concurrent graph processing. In *Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis*. 3:1–3:14.

[62] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 301–316.

[63] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the 2015 USENIX Annual Technical Conference*. 375–386.

Received 14 October 2022; revised 6 April 2023; accepted 8 May 2023