Two-Dimensional Balanced Partitioning and Efficient Caching for Distributed Graph Analysis

Shuai Lin, Rui Wang, Yongkun Li, Yinlong Xu, and John C. S. Lui, *Fellow, IEEE*

***Abstract*—Distributed graph analysis usually partitions a large graph into multiple small-sized subgraphs and distributes them into a cluster of machines for computing. Therefore, graph partitioning plays a crucial role in distributed graph analysis. However, the widely used existing graph partitioning schemes balance only in one dimension (number of edges or vertices) or incur a large number of edge cuts, so they degrade the performance of distributed graph analysis. In this article, we propose a novel graph partition scheme BPart and two enhanced algorithms BPart-C and BPart-S to achieve a balanced partition for both vertices and edges, and also reduce the number of edge cuts. Besides, we also propose a neighbor-aware caching scheme to further reduce the number of edge cuts so as to improve the efficiency of distributed graph analysis. Our experimental results show that BPart-C and BPart-S can achieve a better balance in both dimensions (the number of vertices and edges), and meanwhile reducing the number of edge cuts, compared to multiple existing graph partitioning algorithms, i.e., Chunk-V, Chunk-E, Fennel, and Hash. We also integrate these partitioning algorithms into two popular distributed graph systems, KnightKing and Gemini, to validate their impact on graph analysis efficiency. Results show that both BPart-C and BPart-S can significantly reduce the total running time of various graph applications by up to 60% and 70%, respectively. In addition, the neighbor-aware caching scheme can further improve the performance by up to 24%.**

***Index Terms*—Distributed graph systems, graph partition, graph algorithms, graph processing.**

I. Introduction

GRAPHS are efficient to represent the relationships among entities in various domains, and numerous graph applications have been developed to extract useful information from graphs, such as personalized PageRank [2], [3], SimRank [4], Node2vec [5], etc. However, large-scale graphs pose a challenge for a single machine's memory resources. For instance, many web graphs are several terabytes in size, which necessitates frequent disk I/O operations on a single machine, resulting in poor performance. Therefore, various distributed graph systems have been proposed to leverage a cluster of machines for graph analysis [6], [7], [8], [9], [10], [11], [12], [13], [14].

These distributed graph systems partition the large graph into multiple subgraphs, each on a cluster machine. Each machine only analyzes the local subgraph and transfers the tasks to other machines if needed. To achieve efficient communication and synchronization between cluster machines, these systems usually adopt a BSP model [7], which analyzes the graph iteratively. In each iteration, all machines first perform analysis on local subgraphs and then transmit the communication data to other machines in parallel. When all machines finish the current iteration, then they synchronously go to the next iteration. Due to the simplicity and efficiency of the synchronous framework, this BSP model is widely used in many distributed graph systems to support general graph algorithms, including the state-of-the-art systems KnightKing [6] and Gemini [14].

We highlight that the graph partitioning strategy is vital in distributed graph systems, as it affects the load balancing and communication overhead. Specifically, graph partitioning affects the balance of computing loads among machines, which depends on the distribution of vertices and edges in each subgraph. For example, for many random walk based algorithms, the computing loads in each machine are decided by the number of walkers and the number of steps that each walker can move over the local subgraph, which are dependent on the number of vertices and edges, respectively. On the other hand, graph partition also affects the communication traffic since there may be many edge cuts (i.e., edges between partitioned subgraphs). And there would be a data transfer if a walker visited an edge cut. These two factors greatly affect the time cost in each iteration for BSP based distributed graph systems, and ultimately influence the graph analysis efficiency. This motivates us to develop a two-dimensional balanced graph partitioning scheme, which partitions the large-scale graph into equal sizes on both vertices and edges, meanwhile minimizing the number of edge cuts between subgraphs.

However, the graph partitioning strategies that are widely used in the existing distributed graph systems can usually balance one dimension, either the number of vertices or edges. For example, the common chunk-based graph partitioning strategy is to chunk the vertices array into multiple disjoint intervals, and by coordinating the chunking boundaries to get balanced vertices or edges between intervals, we have Chunk-V and

Received 27 May 2023; accepted 5 November 2024. Date of publication 18 November 2024; date of current version 12 December 2024. This work was supported in part by NSFC under Grant 62172382 and in part by Youth Innovation Promotion Association CAS. An earlier version of this paper was presented at the International Conference on Parallel Processing (ICPP 2022) [DOI:10.1145/3545008.3545060]. Recommended for acceptance by D. Tiwari. *(Corresponding author: Yongkun Li.)*

Shuai Lin is with the University of Science and Technology of China, Hefei 230026, China (e-mail: Shuailin@mail.ustc.edu.cn).

Rui Wang is with the Zhejiang University, Hangzhou 310027, China (e-mail: rwang21@zju.edu.cn).

Yongkun Li and Yinlong Xu are with the Anhui Province Key Laboratory of High Performance Computing, University of Science and Technology of China, Hefei 230026, China (e-mail: ykli@ustc.edu.cn; ylxu@ustc.edu.cn).

John C. S. Lui is with the The Chinese University of Hong Kong, Ma Liu Shui, Hong Kong (e-mail: cslui@cse.cuhk.edu.hk).

Digital Object Identifier 10.1109/TPDS.2024.3501292

Chunk-E, respectively. Both of them have been adopted in recent distributed graph systems, such as Gemini [14] and GridGraph [15] use Chunk-V, and KnightKing [6] uses Chunk-E. Nevertheless, these algorithms can only balance one dimension, while the other dimension is quite imbalanced. This is because many real-world graphs have a scale-free nature [16], i.e., the vertex degrees follow a power-law distribution. Thus, if the vertices (or edges) are balanced, then the edges (or vertices) are often highly imbalanced as some high-degree vertices are easily gathered in the same subgraph [17].

For instance, when using the vertex-balanced Chunk-V to partition the Twitter [18] graph into eight subgraphs, the number of edges varies from 61 to 737 million. Likewise, for the edge-balanced Chunk-E, the number of vertices varies from 744 thousand to 14 million. This imbalance of vertices or edges causes imbalanced computing loads among machines and high synchronization overhead for the BSP model as it takes a long time to wait for the slowest machine. When using KnightKing [6] to run Deepwalk [19] on eight machines, the average waiting time is up to 40% of the total time (see Section IV-C).

Another simple graph partitioning strategy is to do hashing on vertices by randomly assigning them to a subgraph according to their hash results, and we call this partitioning strategy as Hash. Hash is also widely used in distributed graph systems like Giraph [11] and Pregel [7]. Due to the randomness, this hash-based design can balance both vertices and edges, but it introduces many edge cuts and thus incurs high communication costs in distributed graph systems. For example, even we use Hash to partition the Twitter or Friendster graph into eight subgraphs, there are around 88% of edges cuts, and these edge cuts cause an expensive communication time as high as 40% of the total computing time (see Section IV-E). In summary, none of the existing graph partitioning strategies can achieve efficient distributed graph processing with load balancing and lightweight communication overhead.

In this paper, we develop a two-dimensional balanced graph partition scheme BPart,[1] which realizes the balanced graph partition for both vertices and edges to achieve load balancing, and minimizes the edge cuts between subgraphs to decrease the communication overhead for distributed graph systems. The main idea of BPart is to adopt a two-phase partition, which first splits the large graph into many small pieces with regular distributions of vertices and edges, and few edge cuts, and then elaborately combine these small pieces into larger subgraphs to achieve two-dimensional balanced partitioning. In the first phase, we use a weighted graph partitioning policy to integrate the effects of both vertices and edges, aiming to simultaneously relax the imbalanced degree of both dimensions meanwhile making them inversely proportional, so that the pieces can be combined to form better-balanced subgraphs. In the second phase, we exploit the inversely proportional feature of the numbers of vertices and edges of the partitioned pieces, and combine them into larger subgraphs to improve the balance of the two dimensions. Through multiple rounds of the two-phase partitioning process, we can balance both vertices and edges, thus reducing the synchronization overhead, and meanwhile minimizing the edge cuts and decreasing the communication overhead. Eventually, we can achieve high-performance distributed graph computing with BPart. Our main contributions are summarized as follows.

- We study the impact of graph partition in distributed graph systems on the balanced computing loads between a cluster of machines, and the communication overhead of minimizing the number of edge cuts. We also propose a weighted policy to measure the effects of both vertices and edges.
- We propose a two-dimensional balanced partition scheme BPart, via a two-phase partition, which uses a weighted policy to partition the graph into many pieces and then combine two pieces into a larger one. By combining the pieces for multiple rounds, we can realize the desired balance for both vertices and edges and decrease the number of edge cuts.
- Based on BPart, we propose two versions of graph partitioning algorithms, i.e., BPart-C and BPart-S. BPart-C chunks the vertices into disjoint intervals by a weighted policy and combines intervals into larger subgraphs. It enables fast and two-dimensional balanced partitions, and brings a few edge cuts. BPart-S partitions the vertex stream and assigns each candidate vertex to a piece by quantifying the impacts on pieces, e.g., the impacts on edge cuts and the weighted policy, then it combines these pieces into larger subgraphs. It achieves two-dimensional balanced partitioning and minimizes edge cuts for efficient graph processing.
- We also propose a caching strategy to allow each machine to retain some redundant graph data to further reduce the communication overhead. Specifically, we extend BPart to support overlapped partitioning by allowing the partitioned subgraphs to have some overlaps. This scheme enables each machine to cache some relevant vertices and edges, thus reducing the communication overheads between machines for distributed graph computation.
- We implement BPart-C and BPart-S into two popular distributed graph systems, Gemini [14] and KnightKing [6], and conduct extensive experiments to demonstrate their effectiveness and efficiency. The results show that BPart-C and BPart-S can balance both vertices and edges among subgraphs, thus balancing computing loads between machines. Moreover, the neighbor-aware caching strategy can further decrease the communication costs.

The rest of this paper is organized as follows. In Section II, we first introduce the framework of distributed graph computation and analyze the impact of graph partition, then motivate the design of BPart by analyzing the limitations of existing graph partition schemes. In Section III, we present the main idea and the design details of BPart and evaluate its performance in Section IV. Finally, Section V reviews related work and Section VI concludes this paper.

II. Background and Motivation

In this section, we first present the computation framework of distributed graph systems, and analyze how graph partition

[1]The source codes of BPart: *https://github.com/ustcadsl/BPart*.

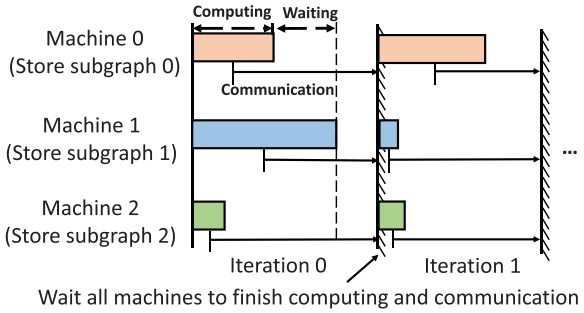


Fig. 1. Illustration on the iteration-based BSP model.

affects computing load balance in a cluster, then we review existing graph partitioning algorithms and discuss their limitations for distributed graph computing.

A. Distributed Graph Computation Framework

A bulk synchronous parallel (BSP) model is used by many distributed graph systems to coordinate analysis tasks between machines in a cluster [6], [7], [8], [9], [10], [11], [12], [13], [14]. As shown in Fig. 1, the BSP model executes the graph computation task in an iteration-based way, and each iteration consists of two phases: computing and communication, i.e., each machine first updates the local subgraph until no more changes can be made, then sends and receives data to and from other machines for further updates. All machines synchronize at the end of each iteration, i.e., when they are all done with the analysis tasks and the communication tasks of the current iteration, they can go to the next iteration. Some systems process computation and communication phases in a pipelined fashion to amortize the communication cost.

The computation process of the BSP model shows that both the balanced computing load and the amount of communication tasks can greatly affect the performance of distributed graph systems. First, if the computing loads are imbalanced, some machines wait for data from slower machines, which brings the waiting time and degrades the system's performance. The waiting time of a machine depends on when it and the slowest machine finish their computation tasks and the total waiting time of all machines, which we call *synchronization overhead*, depends on the load balance of all machines. Ideally, if all machines have equal workloads, they can quickly receive and send data to other machines and proceed to the next iteration. Second, the time for sending communication tasks, which we call *communication overhead*, depends on the number of communication tasks. If the number is small, all machines can finish the communication phase fast and proceed to the next iteration. Next, we analyze how graph partition affects the *synchronization overhead* and the *communication overhead*.

B. Impact of Graph Partitioning

We observe that graph partition can greatly influence the performance of distributed graph systems from two aspects. First, the balanced degree of the number of vertices and the number of edges between the partitioned subgraphs directly influences the balance of the computing workloads between machines. The main reason is that both the number of vertices and the number of edges in a partitioned subgraph directly affect the load of the computation tasks over subgraphs. To further illustrate this, we take a typical kind of graph application, i.e., random walk, as an example. On the one hand, many random walk based algorithms usually start random walks from each vertex or each random walk randomly selects a vertex to start, e.g., Deepwalk [19], node2vec [5], and random walk with domination [20], so the number of walkers in a subgraph usually depends on the number of vertices in this subgraph. On the other hand, according to the BSP model, each random walk continues to walk until it requires the data of other subgraphs, so the number of steps that the walkers in a subgraph can move depends on the number of edges in this subgraph. As a result, the total computing load in a machine is determined by both the number of vertices and the number of edges of the subgraph stored in this machine. Mathematically, the computing load over a subgraph G_i $(i = 0, 1, \ldots, n-1$, where n is the number of machines or subgraphs) can be roughly estimated with the following formula.

$$L(G_i) = f\left(g\left(|V_i|\right), h\left(|E_i|\right)\right),$$

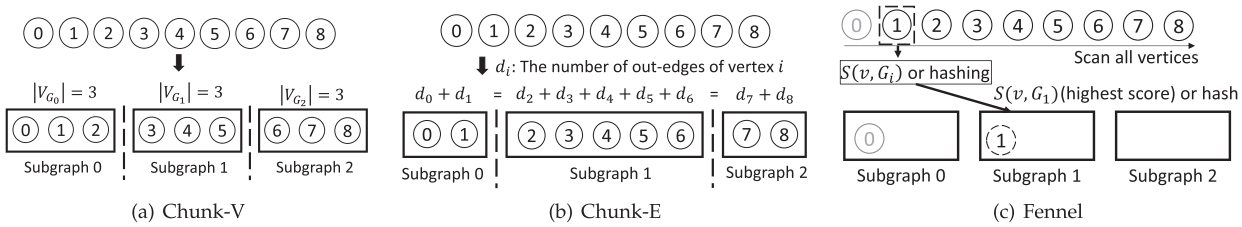
where V_i and E_i denote the vertex set and the edge set of subgraph G_i, respectively.

Second, the number of edge cuts, i.e., edges spanning two partitioned subgraphs, can directly influence the amount of communication tasks. To illustrate this, we also take the example of random walk. For each random walk, they repeatedly choose an edge of the current vertex, and visit the destination vertex of the chosen edge. If a random walk chooses an edge cut, it can not continue to update in the local machine, as the visited vertex is in another machine. Further, the system will generate a communication walk, and send it to the corresponding machine. So, if the number of edge cuts is high, then the communication overhead is large in distributed graph system.

To meet the high-performance requirements of distributed graph systems, it is very crucial to adopt a balanced graph partition while minimizing the number of edge cuts. In particular, it is necessary to achieve a balanced partition in both the number of vertices and the number of edges. With such a partition strategy, the synchronization overhead and communication overhead can be minimized, thereby improving the efficiency of graph systems.

C. Existing Graph Partitioning Algorithms

Chunk-based graph partitioning: A simple and common graph partitioning method is the chunk-based strategy, which simply cuts the vertices array into disjoint intervals. Fig. 2(a) and (b) show the workflows of Chunk-V and Chunk-E, respectively. Chunk-V chunk all vertices into multiple disjoint intervals by grouping vertices with adjacent IDs into the same subgraph, and balance the number of vertices by coordinating the chunking boundaries. Due to the simple partition scheme, Chunk-V is used in multiple systems, such as Gemini [14] and GridGraph [15] Similarly, Chunk-E balances the number of edges by chunking



(a) Chunk-V (b) Chunk-E (c) Fennel

Fig. 2. Illustration on the chunk-based and stream-based partition algorithms: Chunk-V, Chunk-E, and Fennel.

each subgraph with the same number of edges, and is also used in multiple systems, such as the state-of-the-art distributed random walk system KnightKing [6]. Note that Chunk-V and Chunk-E can balance either the number of vertices or edges, but they do not consider the number of edge cuts, i.e., the edges between subgraphs. The number of edge cuts for the chunk-based algorithms depends on the locality of vertices with adjacent IDs. For example, if we assign adjacent IDs to vertices in the same community, then the chunk-based algorithms have better locality and fewer edge cuts. On the contrary, if we assign random IDs to vertices, then they have poor locality and more edge cuts.

Stream-based graph partitioning: In order to balance the number of vertices and meanwhile reduce the number of edge cuts, the stream-based partitioning algorithms are proposed and widely used in large-scale graph partition. They treat all vertices as a vertex stream, and each time take vertices and their belonging edges from the stream to decide which subgraph they belong to. For example, the Fennel algorithm [21] each time takes one vertex from the vertex stream, and decides which subgraph it should belong to by computing a score for each subgraph, then adds this vertex and its associated edges to the subgraph with the highest score. Fig. 2(c) illustrates the process. The score is defined as follows:

$$S\left(v, G_i\right) = |V_i \cap N(v)| - \alpha\gamma|V_i|^{\gamma-1},$$

where $N(v)$ is vertex $v's$ neighbor set, V_i is the vertex set of subgraph G_i, α and γ are constants. The first term $|V_i \cap N(v)|$ denotes the number of common vertices between $v's$ neighbors and subgraph G_i. We should add vertex v to the subgraph G_i with the highest number of common vertices, so as to minimize the edge cuts. The second term $\alpha\gamma|V_i|^{\gamma-1}$ denotes the number of vertices already assigned to G_i with a weighted factor, so it is similar to a penalty factor to avoid a large subgraph continuing to have more vertices, which penalizes large subgraphs and balances the number of vertices.

Hash-based graph partitioning: Another simple graph partitioning design is to use Hash, which randomly assigns each vertex to a subgraph. Its workflow is similar to Fennel as shown in Fig. 2(c), but instead of computing a complicated score, it simply generates a hash value for each vertex to decide which subgraph to assign. Note that the randomness of hash-based algorithms can achieve a balanced graph partitioning in the two dimensions of the number of vertices and edges, but the cost is obvious, the randomness also breaks the locality of the subgraph, resulting in plenty of edge cuts and high communication overhead.

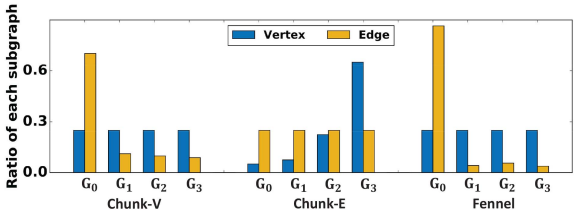


Fig. 3. The ratios of the number of vertices and the number of edges in subgraphs $G_0 - G_3$.

D. Limitations of Existing Solutions

Limitation #1: Inefficiency of 2D balanced partition: It is pointed out that existing graph partitioning algorithms except for Hash can only balance one dimension, i.e., the number of vertices or the number of edges, while the other dimension is quite imbalanced due to the scale-free nature of real-world graphs. Specifically, Chunk-V and Fennel only balance the number of vertices and Chunk-E only balances the number of edges. To demonstrate the above inferences experimentally, we run the above-mentioned three graph partitioning algorithms, i.e., Chunk-V, Chunk-E, and Fennel, to show the distributions of the number of vertices and the number of edges. We take the real-world graph Twitter graph as an example, and partition it into four subgraphs to distribute them in a four-machine cluster, and show the ratios of the number of vertices and the number of edges in each subgraph in Fig. 3. We also observe similar results under more graph datasets (see Section IV). From the results, we can see that Chunk-V and Fennel can realize a balanced partition for the number of vertices, that is, for all four subgraphs $G_0 - G_3$, the ratio of the number of vertices is close to 0.25. However, the number of edges is quite imbalanced, and the distribution is highly skewed, and the difference between the maximum and the minimum numbers of edges, can reach up to $8\times$. In contrast, for Chunk-E, all four subgraphs contain a balanced number of edges, but the numbers of vertices are quite imbalanced, and the difference between the maximum and the minimum numbers of vertices can reach up to $13\times$.

We point out that such highly skewed distributions of vertices or edges greatly affects the distribution of computing loads in distributed graph processing, which leads to high synchronization overhead. To further demonstrate, we take a random walk application as an example, and run experiments to illustrate the impact of imbalanced partition on the distribution of computing loads. We start five random walks from each vertex over the

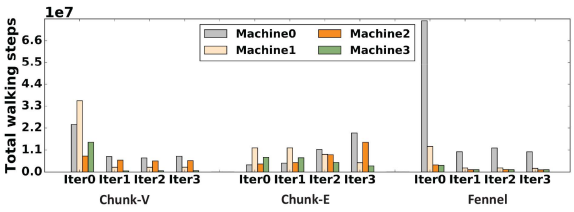
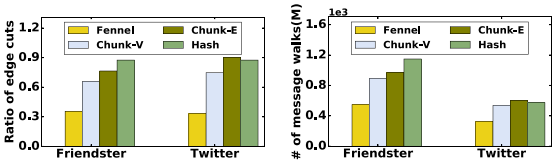


Fig. 4. The distribution of the computing loads between machines in different iterations (Iter0 - Iter3).



(a) Edge cuts (b) Total message walks

Fig. 5. The ratio of edge cuts and total message when using different partition algorithms.

Twitter graph and let each walk move four steps. Fig. 4 shows the computing load (i.e., the number of total walking steps) of each machine. We see that for the three partition algorithms, which only balance one dimension, either the number of vertices or edges, the computing loads between machines are imbalanced in each iteration of the BSP model. In particular, for Chunk-V and Fennel, which balance the number of vertices, the computing loads are still highly imbalanced even though the initial number of walks is balanced in the first iteration. This is because the walkers move different numbers of steps due to the imbalanced number of edges. For Chunk-E, the imbalanced number of vertices causes imbalanced number of random walks and computing loads between machines. From these results, we conclude that existing chunk-based and stream-based partition algorithms can not realize the balanced computing loads among a cluster of machines due to the imbalanced graph partition, which finally leads to low performance of distributed graph processing.

Limitation #2: High communication traffic: On the other hand, the hash-based algorithms can balance both the number of vertices and edges due to their randomness in assigning vertices to subgraphs. However, the randomness also breaks the locality of vertices in the same subgraph, bringing plenty of edge cuts between subgraphs, which lead to high communication traffic due to the high chance of visiting cut edges. To demonstrate this, we use different partition algorithms, e.g., Chunk-V, Chunk-E, Fennel, and Hash, to partition the Friendster and Twitter graphs into eight subgraphs, and show the ratio of edge cuts in Fig. 5(a). We can see that Chunk-E and Hash contain around 90% edge cuts, that is, around 90% edges are crossing edges between partitioned subgraphs. Fennel significantly reduces the edge cuts to only around 30%, due to the efficient score computation. To further show the impact of edge cuts on the communication traffic, we also run a random walk application as an example, by starting five random walks from each vertex and letting each walk move four steps, and show the number of message walks, i.e., walks being transmitted, in Fig. 5(b). The results show that Chunk-E and Hash have more than 2× message walks than Fennel as they have more edge cuts, which hurts the performance of distributed graph processing.

III. Design of BPart

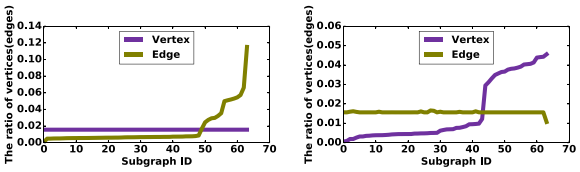
In this section, we first introduce our key observation on partition constraints and the main idea of our new partition scheme, which aims to achieve the two-dimensional balanced partition by using a two-phase partition scheme. After that, we present the design details in each phase. Baed on BPart, we further propose two enhanced algorithms, BPart-C and BPart-S. We then develop intersected partitioning, which allows each partitioned subgraph to have some redundancy, i.e., the subgraph stored on each machine also has related graph data of other subgraphs, so that the communication cost between machines during distributed processing can be reduced.

A. Observation & Main Idea

To support high-performance distributed graph processing with balanced computing loads, the graph partition algorithms should be balanced in two dimensions, i.e., the number of vertices and the number of edges. In the following, we first present an observation by analyzing existing partition schemes, then show the main idea of BPart.

Observation: Existing graph partitioning algorithms usually target balancing the measure in one dimension, i.e., either the number of vertices or the number of edges, which often results in a highly imbalanced distribution in another dimension, especially for real-world natural graphs. Specifically, if the number of vertices is evenly distributed, then the number of edges may be highly imbalanced between partitioned subgraphs, and vice versa. The rationale of this observation is that natural graphs like social networks often have a scale-free property, e.g., the distribution of vertex degrees (i.e., the numbers of edges connected to each vertex) follows a power-law distribution, with a small fraction of vertices containing the large majority of edges. As a result, in order to decrease the edge cuts, existing one-dimensional balanced graph partitioning algorithms tend to gather high-degree vertices in the same subgraphs, because there are many edge connections between these high-degree vertices. For example, if the numbers of vertices are evenly partitioned among the subgraphs, then subgraphs containing high-degree vertices often have significantly more edges than the other subgraphs. That is, the number of edges among the partitioned subgraphs is highly imbalanced.

To further validate this observation, we conduct experiments on real-world graphs to show the distributions of the numbers of vertices and edges, and to illustrate the skewness of the imbalanced dimension. We partition the Twitter [18] graph into 64 small subgraphs with the chunk-based partition algorithms, i.e., Chunk-V and Chunk-E, by using either the vertex-based balance indicator or the edge-based balance indicator, respectively. Note that, we also observe similar trends on other graphs like Friendster [22] and LiveJournal [23] and other one-dimensional balanced graph partitioning algorithms like Fennel. Fig. 6(a)



(a) Partitioning with Chunk-V (b) Partitioning with Chunk-E

Fig. 6. The distribution of $|V_i|$'s and $|E_i|$'s of the subgraphs.

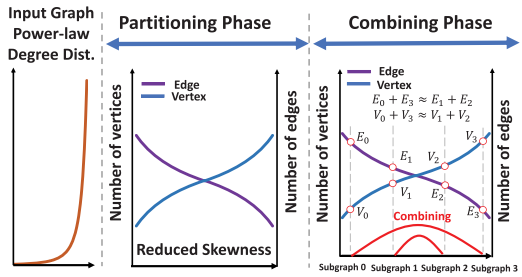


Fig. 7. Main idea of BPart with two-phase partition.

and (b) show the ratio of the number of vertices (i.e., $|V_i|/|V|$) and the ratio of the number of edges (i.e., $|E_i|/|E|$) of each subgraph, by using Chunk-V and Chunk-E, respectively. Here V_i and E_i denote the vertex set and the edge set of subgraph G_i, V and E denote the vertex set and the edge set of the original graph. The results show that Chunk-V balances $|V_i|$, while $|E_i|$ is extremely imbalanced with highly skewed distribution. Among them, the largest subgraph with 1.56% vertices even contains more than 11% of the total edges, which is 119× higher than the number of edges contained in the smallest subgraph with only 0.1% of the total edges. Similarly, Chunk-E balances $|E_i|$, while $|V_i|$ is highly imbalanced, and the number of vertices in the largest subgraph is 75× that of the smallest subgraph.

Remark: This observation of the highly skewed distribution in the imbalanced dimension (i.e., either the number of vertices or the number of edges), also implies that it is difficult to achieve two-dimensional balance through simple subgraph composition.

Main idea: Two-phase partition: To realize a two-dimensional balanced partition, and meanwhile minimize the edge cuts, we develop a new partition scheme, BPart, and its key idea is to adopt a *two-phase partition*, which includes a partitioning phase and a combing phase, as shown in Fig. 7. Specifically, in the partitioning phase, instead of aiming to realize a perfect balance in one dimension while making the other dimension highly imbalanced among partitioned subgraphs, our goal is to first partition the original graph into many small pieces, and reduce the skewness of the distributions in both dimensions, thereby removing the pow-law distributions among these small pieces in both dimensions. The principle is that by generating small pieces without extremely large numbers of vertices or edges, these small pieces can be combined into larger subgraphs with balanced numbers of vertices and edges, thereby achieving balance in both dimensions. To achieve this goal, BPart leverages a weighted policy that takes into account both the number of vertices and the number of edges in the balance indicator, thus reflecting the impacts of both dimensions during the partition. Through this weighted policy, we can adjust the distributions of both the numbers of vertices and edges, and coordinate the two distributions to make them inversely proportional, that is, the partitioned small pieces with fewer vertices must have more edges, and vice versa. In the subsequent combining phase, we aim to use the results of the partitioning phase to achieve a better balance in both dimensions. To achieve this, we can selectively combine two small pieces into a larger subgraph based on the distributions of the numbers of vertices and edges, so that the newly combined subgraphs have more balanced distributions in both dimensions.

There are two challenges that need to be addressed to realize the idea of above two-phase graph partitioning. First, in the partitioning phase, how to design a weighted policy that considers the impacts of vertices and edges at the same time, so that the distribution of the number of vertices and edges among the partitioned pieces can be adjusted to be inversely proportional as desired. Second, in the combining phase, how to combine small pieces into larger subgraphs to achieve a balanced partition in both dimensions. We may need to perform multiple rounds of combination to finally achieve the ideal balance of vertices and edges. We carefully introduce the design details of BPart to address these two challenges, and base on the design of BPart, we also propose two versions of graph partitioning algorithms BPart-C and BPart-S to accommodate different scenarios. Then, we also propose a caching strategy to allow each machine to retain some redundant graph data to further reduce the communication overhead. Next, we introduce them in detail.

B. Partitioning Phase Design

Two-dimensional weighted balance indicator: Traditional graph partitioning strategies use either the number of vertices $|V_i|$ or the number of edges $|E_i|$ in subgraph G_i as the balance indicator, e.g., Chunk-V and Fennel use $|V_i|$ as the balance indicator, and Chunk-E uses $|E_i|$ instead. As a result, one can only get balanced partitions in one dimension, either the number of vertices or the number of edges, while the other dimension would be highly imbalanced due to the scale-free nature of real-world graphs. Therefore, we need to design a new two-dimensional balance indicator to guide the graph partition process. The key idea is to use a weighted approach to integrate the influences of both $|V_i|$ and $|E_i|$. Mathematically, we design the two-dimensional weighted balance indicator $\mathcal{W}_i$ for subgraph G_i as follows:

$$\mathcal{W}_i = c \times |V_i| + (1 - c) \times |E_i|/\bar{d}, \tag{1}$$

where $c (0 \leq c \leq 1)$ is a weighting factor to control the influence ratio of $|V_i|$ and $|E_i|$, and $\bar{d}$ is the average degree of the graph. With this two-dimensional weighted balance indicator, the graph partition goal is to make $\mathcal{W}_i$ to be equal. In particular, $c = 0$ corresponds to the edge balance indicator, which achieves the balanced distribution of $|E_i|$, and $c = 1$ corresponds to the vertex balance indicator, which achieves the balanced distribution of $|V_i|$. For the weighting factor, we recommend using equal weights of $|V_i|$ and $|E_i|$ based on our empirical study, i.e., we set

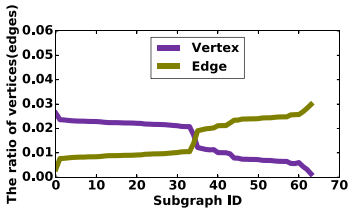


Fig. 8. The ratios of $|V_i|$'s and $|E_i|$'s with the weighted policy.

$c = \frac{1}{2}$ by default. One can also set different weighting factors for different graph partition situations to choose different balances between the number of vertices and the number of edges.

Chunk-based weighted graph partitioning: Based on the weighted balanced indicator $\mathcal{W}_i$, we first follow the chunk-based partition workflow to propose a lightweight chunk-based weighted graph partitioning algorithm BPart-C. The core idea of BPart-C is to decide the chunking boundaries, which are decided by the weighted balanced indicator $\mathcal{W}_i$. Specifically, we sequentially add the adjacent vertex IDs and their corresponding edges to the same subgraph, until $\mathcal{W}_i$ reaches $\bar{\mathcal{W}}$, where $\bar{\mathcal{W}}$ denotes the average value of the balanced indicator, and defines as follows:

$$\bar{\mathcal{W}} = \frac{1}{P} \times \left(\frac{1}{2} \times |V_i| + \frac{1}{2} \times |E_i|/\bar{d} \right),$$

where P is the number of partitioned subgraphs. Using the weighted policy in the partitioning phase can reduce the skewness of the distribution of $|V_i|$ and $|E_i|$, and in particular, the number of vertices could be proportional to the inverse of the number of edges. The rationale is that as $\mathcal{W}_i$'s are equal, then a subgraph G_i containing fewer vertices (i.e., smaller $|V_i|$) must have more edges (i.e., larger $|E_i|$). We also run experiments to further demonstrate this result. We partition the Twitter graph into 64 small subgraphs with the chunk-based partition algorithm by using the weighted balance indicator, and show the ratio of the number of vertices and edges of each subgraph in Fig. 8. We can see that neither $|V_i|$ nor $|E_i|$ is balanced among subgraphs, while the skewness is greatly decreased compared with the results in Fig. 6(a) and (b), and the two distributions of $|V_i|$ and $|E_i|$ are inversely proportional to each other. This implies that we could realize the desired balance for both vertices and edges through appropriate combinations of these small subgraphs.

Stream-based weighted graph partitioning: Based on the weighted balanced indicator $\mathcal{W}_i$, we follow the stream-based partition workflow to propose an efficient stream-based weighted graph partitioning algorithm BPart-S. Specifically, BPart-S treats all vertices as a vertex stream and computes a score to decide which subgraph a vertex should belong to. We extend the score designs of Fennel by using a weighted policy. Mathematically, we design the partition score $S(v, G_i)$ of vertex v and subgraph G_i as follows:

$$S(v, G_i) = |V_i \cap N(v)| - \alpha\gamma\mathcal{W}_i^{\gamma-1},$$

where V_i denotes the current vertex set of subgraph G_i, $N(v)$ denotes $v's$ neighbors, $|V_i \cap N(v)|$ denotes the number of common vertices between v's neighbors and V_i, the larger number of $|V_i \cap N(v)|$ denotes fewer edge cuts between v and subgraph G_i. α and γ are just constants used for adjusting the weights of the edge-cut number and the balanced degree, and we set both of them as 1.5 by default.

We point out that the two versions of graph partitioning workflows, BPart-C and BPart-S, have different scenarios. BPart-C can achieve fast graph partitioning by simple chunking, but the number of edge cuts depends on the locality of the adjacent IDs. In contrast, BPart-S requires a larger time cost to compute the scores for all vertices, but it contributes to fewer edge cuts like the Fennel algorithm. Therefore, there is a trade-off between the partition overhead and communication cost for the two kinds of partition algorithms. We also experimentally investigate this trade-off in Section IV-F. One can choose to use the lightweight version, i.e., BPart-C, which has smaller partition overhead, or the high-performance version, i.e., BPart-S, which has fewer edge cuts and thus lower communication cost.

TABLE I
$|V_i|$ AND $|E_i|$ OF THE PARTITIONED 8 SMALL SUBGRAPHS

Subgraph	$\|V_i\|$	$\|E_i\|/d$
0	15264196	1137845
1	12651971	3750121
2	9181822	7220273
3	7079884	9322208
4	6449143	9952990
5	5278939	11123154
6	5115450	11286643
7	4586962	11815131

C. Combining Phase Design

In the combining phase, we target to achieve a two-dimensional balanced partition for both dimensions. The key idea is to combine the small pieces partitioned by the partitioning phase into the final output subgraphs. For example, if we want to partition a graph into N subgraphs, we first partition the graph into $2 \times N$ smaller pieces based on the weighted balance indicator defined in (1). Then, we sort these small pieces by the number of vertices, i.e., $|V_i|$, and according to the inversely proportional nature, the piece with a smaller $|V_i|$ generally has a larger number of edges, i.e., $|E_i|$, and vice versa. As a result, we can combine the piece with the fewest vertices (also with the most edges) and the piece with the most vertices (also with the fewest edges) into a larger subgraph with moderate numbers of vertices and edges, and continue this combination for the remaining pieces. Finally, we can get N larger subgraphs and expect that these combined subgraphs have more balanced vertices and edges. We further illustrate the above process by using a precise example. Suppose that we intend to partition the Friendster graph into 4 subgraphs, we first partition it into 8 smaller pieces based on the weighted balance indicator defined in (1). Table I shows the distribution of $|V_i|$ and $|E_i|$ of each piece, which is sorted in descending order of $|V_i|$. Then we combine subgraph i with subgraph $7 - i$ $(i = 0, 1, 2, 3)$, and show the results after combination in Table II, in which the combined

TABLE II
$|V_i|$ AND $|E_i|$ OF THE COMBINED 4 LARGE SUBGRAPHS

Subgraph	$\|V_i\|$	$\|E_i\|/d$
0	19851158	12953026
1	17767421	15036765
2	14460761	18343428
3	13529027	19275198

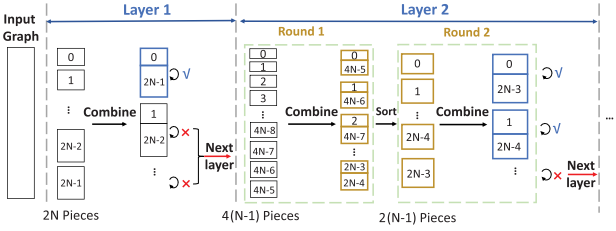


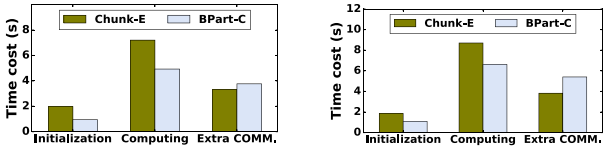
Fig. 9. Multi-layer combination strategy.

subgraph i ($i = 0, 1, 2, 3$) is the combination of the small pieces i and $7 - i$ in Table I.

Multi-layer combination strategy: From the results in Table II, we can see that the distributions of $|V_i|$ and $|E_i|$ are much more balanced than those in Table I, but they are still not perfectly balanced. This example implies that only one shot of combination is usually not enough to achieve our goal. Fortunately, we find that the two distributions of $|V_i|$ and $|E_i|$ of the combined subgraphs still satisfy the desired inversely proportional nature, which guides us to design a *multi-layer combination strategy* that continues the combination in multiple rounds until the balanced condition is satisfied. In detail, as illustrated in Fig. 9, after each round of combination as introduced above, we check the balanced degree of $|V_i|$ and $|E_i|$ for each combined subgraph. If the subgraph reaches the balanced thresholds for both vertices and edges, we take it as a final partitioned subgraph. Otherwise, we re-partition the remaining unsatisfied N_r subgraphs and proceed to the next layer of combination, which implies that $N - N_r$ combined subgraphs have reached the balanced thresholds.

For example in Fig. 9, in the first layer, we first partition the origin graph into $2 \times N$ pieces and combine them into N subgraph, of which only one subgraph reaches the balanced thresholds, and the other $N_r = N - 1$ subgraphs are not balanced. Thus, in the second layer, we partition the remaining subgraphs into $4 \times N_r$ pieces and combine them in two rounds to obtain N_r subgraphs. The first combination combines $4 \times N_r$ pieces into $2 \times N_r$ pieces and the second combination combines these $2 \times N_r$ pieces into N_r subgraphs. Then, we check again how well $|V_i|$ and $|E_i|$ of these combined N_r subgraphs are balanced, and repeat the above process until all the vertices and edges of the combined subgraphs are balanced. Generally speaking, according to our experiments, after two to three rounds of combinations, we can get the desired balanced partition of vertices and edges.

Connectivity of the combined subgraphs: Through this multi-layer combination strategy, the final partitioned subgraphs are composed of multiple small-size pieces. One might be concerned whether these small pieces belonging to the same final subgraph are still well connected, that is, whether there are enough connected edges between these small pieces. We would like to point out that even if we partition the graph into many smaller pieces, there are still massive edge connections between any two of these small pieces. To demonstrate this, we take Friendster as an example, and partition it into 64 small pieces using the partitioning phase of BPart-C and BPart-S, respectively. We find that there are at least 159,000 and 50,000 edge connections between any two pieces of BPart-C and BPart-S, respectively, implying that the combined pieces are well connected and the combination strategy will not make the combined subgraphs disconnected.



(a) Twitter (b) Friendster

Fig. 10. Time cost breakdown.

D. Intersected Partitioning via Neighbor-Aware Caching

In this subsection, we develop a caching scheme that enables the partitioned subgraphs to have certain redundancy of related graph data. We first perform an empirical study to justify the need of such an intersected partitioning design. To achieve this, we first implement our graph partition algorithms BPart-C and BPart-S on top of the state-of-the-art distributed graph system KnightKing [6], which also supports random walks, and study their impacts on distributed graph processing. For illustration, we show one experiment which simultaneously starts $5 \times |V|$ random walks, and lets each walk move 10 steps on Friendster and Twitter graphs. For the detailed experiment settings, please refer to Section IV-A. We show the time cost breakdown in Fig. 10, which includes three parts: initialization of random walks, computing, and extra communication. Note that KnightKing uses pipelining to reduce the total cost of the computation and communication (refer to Fig. 1). So we do not count the overlapped part of the communication time which has already been amortized by the computation, and we only record the communication time after computing, which we call the extra communication time (abbreviated as Extra COMM). We show the results of Chunk-E, which is the default partition strategy used in KnightKing, and BPart-C, for fair comparison as they are both chunk-based graph partition algorithms with comparable partition overhead.

We can see that BPart-C largely decreases the computation time by around 23.9% and 31.6% on Friendster and Twitter, respectively, due to the two-dimensional balanced partition, compared with Chunk-E. However, BPart-C increases the extra communication time by around 41.3% and 13.6% on Friendster and Twitter, respectively, because there is less overlapped communication time that can be amortized by the computation as we have reduced the computation time. Therefore, we also

propose a caching strategy to allow each machine to keep certain redundant graph data so as to further reduce the number of edge cuts and the communication overhead.

Given a fixed memory budget, the key issue of the caching design is to decide which data should be chosen to cache to improve the cache efficiency. We adopt a *neighbor-aware caching strategy* to cache the more appropriate vertices according to their neighborhood information. Specifically, when we consider caching vertices for the machine storing subgraph G_i, we classify the neighbors of the candidate vertex v ($v \in$ subgraph G_j), i.e., $N(v)$, into three categories: (1) the set of neighbors that belong to subgraph G_i, which is denoted as $N_i(v)$, (2) the set of neighbors that belong to subgraph G_j, which is denoted as $N_j(v)$, and (3) the set of neighbors that belong to the rest subgraphs, which is denoted as $\sum_{k\in P\setminus\{G_i,G_j\}} N_k(v)$, where P denotes the subgraph set. Note that $\frac{\sum_{k\in P}|N_k(v)|}{d(v)} = 1$, where $d(v)$ denotes the degree of vertex v. Based on this classification, we should first preferentially cache the vertices in $N_i(v)$ with a larger ratio, because two times of communication may be saved between the machines storing subgraphs G_i and G_j, respectively. Then, we should preferentially cache the vertices in $\sum_{k\in P\setminus\{G_i,G_j\}} N_k(v)$ with a larger ratio, because one time of communication may be saved between the machines storing subgraphs G_i and G_j, respectively. Based on the above observations, we compute a caching score for each candidate vertex according to its neighbors' information, then cache the vertices with the highest score values. The score of vertex v ($v \in$ subgraph G_j) in the machine storing the subgraph G_i, which is denoted as $C(v, G_i)$, is defined as follows:

$$C(v, G_i) = \frac{|N_i^{in}(v)| \times \left(2 \times \frac{|N_i(v)|}{d(v)} + \sum_{k\in P\setminus\{G_i,G_j\}} \frac{|N_k(v)|}{d(v)}\right)}{d(v)},$$

where $N_i^{in}(v)$ denotes the number of in-edges of vertex v connected with subgraph i, which is used for indicating the visit frequency of vertex v during graph computation. $2 \times \frac{|N_i(v)|}{d(v)}$ and $\sum_{k\in P\setminus\{G_i,G_j\}} \frac{|N_k(v)|}{d(v)}$ indicate the contribution in reducing the communication of vertex v. Finally, we divide $d(v)$ to give a weight on the storage cost for vertex v. We compute $C(v, G_i)$ for each candidate vertex, and sequentially store the graph data of these vertices in an ascending order of $C(v, G_i)$ in a cache file. After that, with a given memory budget, we can preferentially cache the vertices in the front of the cache file, i.e., cache the vertices with the highest scores.

IV. Evaluation

BPart aims to provide a two-dimensional balanced graph partition and meanwhile minimize the edge cuts, thus realizing a balanced computing load among a cluster of machines and decreasing the communication traffics for distributed graph processing systems. To demonstrate the effectiveness and efficiency of our partition scheme BPart, we implement both the chunk-based version BPart-C, and the stream-based version BPart-S, and compare them with four commonly used graph partition algorithms Chunk-V, Chunk-E, Fennel and Hash, as well as the state-of-the-art shared memory graph partitioning algorithm Mt-KaHIP [24]. We first compare the balance degree in both dimensions, i.e., the vertex balance degree and the edge balance degree. Then we compare the balance of the computing load and the total running time for various graph algorithms based on the different partition algorithms. Here, we take both KnightKing [6], which is the state-of-the-art distributed graph system for running random walk algorithms, and Gemini [14], which supports other graph algorithms, as the code bases, and integrate all partition algorithms into the systems for comparison. After that, we also compare with the hash-based partition algorithm, including the comparison of the number of edge cuts and the total running time for graph applications. We also evaluate the partition overhead and study the impact of the caching strategy.

TABLE III
Statistics of the Graph Datasets

Graphs	# of Vertices	# of Edges	Average Degree
LiveJournal [23]	7.5M	225M	29.99
Twitter [18]	41.39M	1.2B	35.72
Friendster [22]	65.60M	3.6B	21.54

A. Experiment Setup

Testbed: Our testbed uses a cluster of eight machines connected with a 56Gbps Ethernet. Each machine has two 24-core Intel Xeon CPU E5-2650 v4 processors and 64GB DRAM. In the experiments, we may vary the number of machines being really used to study the impact of the cluster scale.

Datasets: Table III shows the statistics of the three graph datasets with different scales used in our experiments, which are all real-world social networks and are widely used to evaluate many graph systems [6], [8], [25], [26], [27].

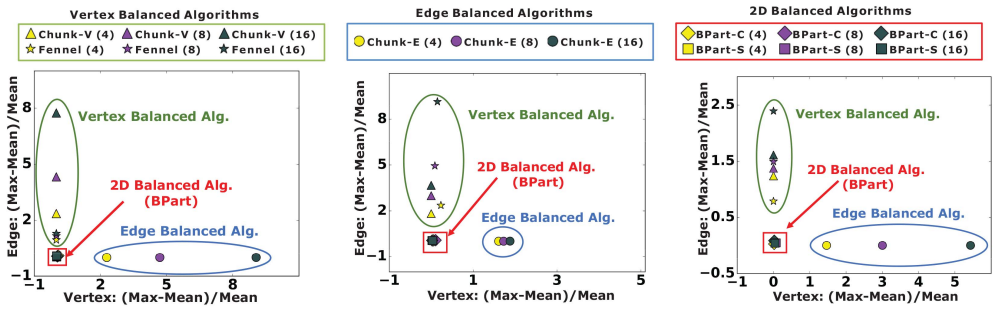
Metrics of balanced degree: We use the following two metrics to study the balanced degree of the partition results:

1) *Bias:* We define *bias* as the difference between the maximum value and the mean value, normalized by the mean value, mathematically, for a set of n values $\{x_i | i = 0, 1, \ldots, n-1\}$, the bias is defined as

$$\text{Bias:} \quad B = \frac{\max(x_i) - \text{mean}(x_i)}{\text{mean}(x_i)},$$

where $\max(x_i)$ denotes the maximum value of $\{x_i\}$, and $\text{mean}(x_i)$ denotes the mean value. We choose the bias metric because the *synchronization overhead* of the BSP model is determined by the difference between the maximum and mean computing loads, that is, all other machines need to wait for the slowest machine (i.e., the machine with the maximum computing load) to finish its computation before going to the next iteration. The computing load of each subgraph is further determined by the number of vertices and the number of edges.

2) *Fairness*: we also use another commonly used [28], [29], [30] fairness measurement, i.e., the Jain's fairness index [31], to characterize the balanced degree of the partitioned subgraphs,



(a) LiveJournal (b) Twitter (c) Friendster

Fig. 11. Balanced degrees of the number of vertices and the number of edges estimated with the bias metric. Note that the numbers in the parentheses denote the number of partitioned subgraphs.

which is defined as follows:

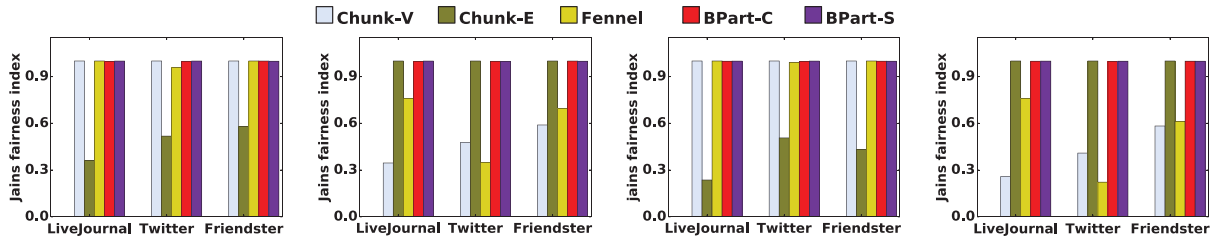
$$\text{Fairness:}\quad F = \frac{\left(\sum_{i=0}^{n-1} |x_i|\right)^2}{n \times \sum_{i=0}^{n-1} |x_i|^2}.$$

Note that the value of Jain's fairness index ranges from $\frac{1}{n}$ to 1. $F = \frac{1}{n}$ means that the partition is completely imbalanced, i.e., one subgraph contains all the vertices or edges, while $F = 1$ means that the partitioned subgraphs are completely balanced, i.e., all subgraphs contain the same number of vertices or edges.

Graph algorithms: Besides evaluating the balanced degree, we also evaluate the balance of the computing loads among the cluster of machines and the total running time for different algorithms. We consider seven widely used graph applications, e.g., Deepwalk [19], personalized PageRank (PPR) [2], random walk with domination (RWD) [20], random walk with jump (RWJ) [32], node2vec [5], PageRank (PR) [33] and Connected Components (CC) [34]. The first five algorithms are random walk algorithms, and we start $|V|$ walks for them by using the same setting as in KnightKing [6]. In each step of a walk, PPR terminates with probability 0.1 and RWJ jumps to a random vertex with probability 0.2. Deepwalk, RWD, RWJ, and Node2vec terminate with a fixed number of steps. The last two algorithms are iteration-based algorithms, and we run them on Gemini [14]. We run PR for ten iterations and CC until convergence. PR and CC are iteration-based algorithms, which are executed for ten iterations and until convergence, respectively, and we run them on Gemini [14].

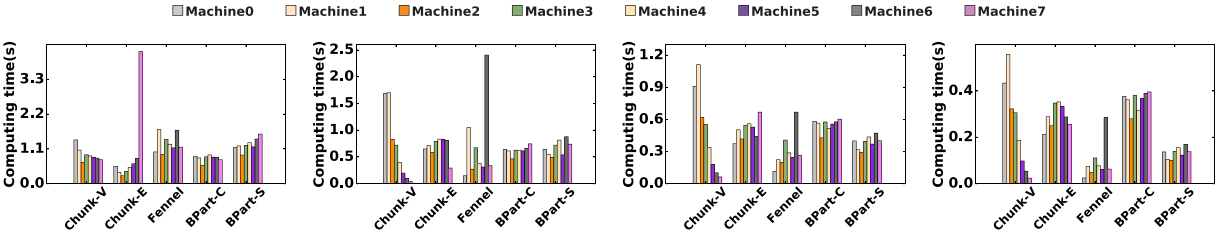
B. Balanced Degree of the Partitioned Subgraphs

We compare the balanced degree of the partition results of the above-mentioned five graph partition algorithms on three real-world graphs. We first show the bias metric of $\{V_i\}$ and $\{E_i\}$, when partitioning the large graph into 4, 8, and 16 subgraphs. As shown in Fig. 11, the x-axis denotes the bias of $\{V_i\}$ and the y-axis denotes the bias of $\{E_i\}$. The results show that Chunk-V, Chunk-E, and Fennel can only achieve a one-dimensional balanced partition. In particular, Chunk-V and Fennel can achieve the vertex balanced partition, but the number of edges is highly imbalanced, e.g., the bias of $\{E_i\}$ can reach up to 7.74 for Chunk-V and 9.15 for Fennel. Chunk-E can achieve the edge balanced partition, however, the distribution of the number of vertices is highly imbalanced, e.g., the bias of $\{V_i\}$ can reach up to 9.06. Furthermore, as we partition the graph into more subgraphs, the bias of $\{V_i\}$ and $\{E_i\}$ gets larger. Our proposed BPart-C and BPart-S can realize a two-dimensional balanced partition, and the bias for different experiments is always very small, within 0.1. In addition to the above graph partitioning algorithms commonly used in the distributed graph systems, we also compare the state-of-the-art shared memory graph partitioning algorithm Mt-KaHIP [24]. Mt-KaHIP uses a multi-level approach, which first coarsens the original graph, and groups multiple vertices into a hyper-vertex by using a label propagation method. The coarsened graph is then partitioned into multiple subgraphs, and finally, these subgraphs are uncoarsened to recover vertices from super-vertices. Note that Mt-KaHIP needs to load the whole graph data into memory and traverse it multiple times to conduct graph partitioning, so it is too memory-consuming and time-consuming to be used in large-scale distributed graph processing. Even if it takes a lot of memory and time, it still can only achieve balanced partitioning of vertices, because the balanced index only considers the number of vertices. To further experimentally demonstrate its balance degrees in vertices and edges, we use Mt-KaHIP to partition three real-world datasets into eight subgraphs and show the bias of $\{V_i\}$ and $\{E_i\}$. The bias of $\{V_i\}$ for different datasets is small than 0.03, but the bias of $\{E_i\}$ are 2.5853, 2.5622, and 0.7046 for LiveJournal, Twitter, and Friendster, respectively. The result implies that Ma-KaHIP can only achieve balanced partition in the number of vertices, and the distribution of the number of edges is quite imbalanced. In the next experiments, we will not talk about Mt-KaHIP as it requires more partitioning time but has a similar partitioning result compared with Fennel. We then show the result of Jain's fairness index to show the balanced degree of the partitioned subgraphs, Fig. 12(a) and (b) show the values of the fairness index for the number of vertices and the number of edges, respectively. In this experiment, we partition the whole graph into 4 subgraphs. Fig. 12(c) and (d) further show the results when partitioning the whole graph into 8 subgraphs. We can see that when using our proposed BPart-C and BPart-S, the fairness index is always very close to 1in both dimensions of



(a) Balance of vertices (4 subgraphs) (b) Balance of edges (4 subgraphs) (c) Balance of vertices (8 subgraphs) (d) Balance of edges (8 subgraphs)

Fig. 12. Balanced degree estimated with Jain's fairness index: (a) and (b) show the balance trade-off when partitioning to 4 subgraphs, and (c) and (d) show the trade-off when partitioning to 8 subgraphs.



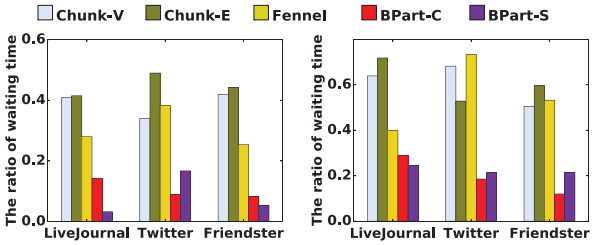
(a) Iteration 0 (b) Iteration 1 (c) Iteration 2 (d) Iteration 3

Fig. 13. The computing time of each machine in different iterations: imbalanced partition leads to imbalanced computing time distribution.

the number of vertices and edges, which indicates that they can always achieve a very good balance in both dimensions. While the fairness index values of the other three partition algorithms only tend to 1in one dimension of vertices or edges, and the value is only around 0.4in the other dimension, indicating that the results are highly imbalanced.

C. Balanced Degree of Computing Loads

We now evaluate how well the computational load is balanced to illustrate the impact of balanced graph partitioning on distributed graph systems. In this experiment, we adopt KnightKing as a code base and integrate various partition algorithms, including Chunk-V, Chunk-E, Fennel, BPart-C, and BPart-S, into it for experiments. Note that we do not modify the computation process of KnightKing. Since KnightKing is optimized for random walks, we run five simple random walks starting from each vertex, i.e., a total of $5|V|$ walks, letting each walk terminate at the fourth step, so the system runs four iterations in total. We first show the ratio of the total waiting time, which is defined as the total waiting time divided by the total running time of all machines. As shown in Fig. 14, for Chunk-V, Chunk-E, and Fennel, the ratio of waiting time can reach up to 70%, which means that 70% of the total running time of the machines in the cluster needs to wait for the slowest machine to finish the computation. This huge ratio of waiting time is caused by the imbalanced partition of the number of vertices or the number of edges. On average, the ratios of waiting time are 45% and 55% with 4 machines and 8 machines, respectively. While for BPart-C and BPart-S, the ratio of waiting time is quite small,

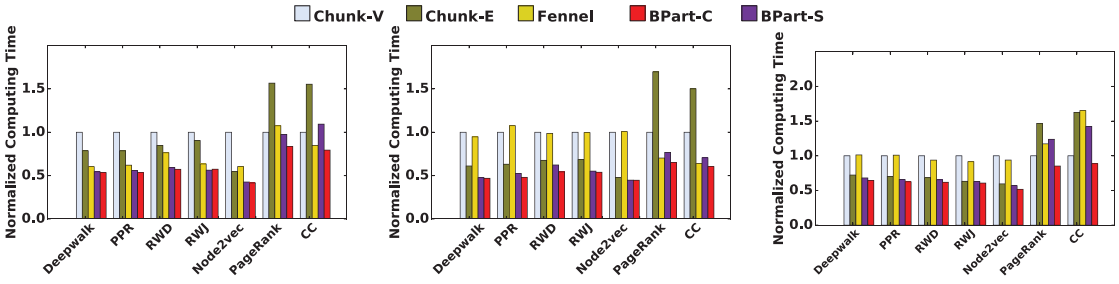


(a) Four machines (b) Eight machines

Fig. 14. The ratio of the total waiting time of all machines to the total running time of random walks: balanced partition algorithms significantly reduce the waiting time.

for example, in the case of using 4 machines and 8 machines, the proportion of waiting time is only 10% and 20%, respectively. This is because BPart-C and BPart-S can achieve a balanced partition in both dimensions and as a result, in each iteration, each machine has a similar amount of computing load, which ultimately contributes to the smaller synchronization cost.

To further illustrate why imbalanced partitions incur a large amount of synchronization cost in waiting, we also show the distribution of the computation time of each machine in every iteration. As shown in Fig. 13, each sub-figure represents the results of one iteration, the x-axis denotes different graph partition algorithms and the y-axis denotes the computation time of each machine. Here we only show the results of Friendster on a cluster of eight machines, and the results are similar for other datasets or cluster scales. For Chunk-V, Chunk-E, and Fennel, we can observe a highly imbalanced distribution of computation time



(a) LiveJournal (b) Twitter (c) Friendster

Fig. 15. The normalized running time of different graph application algorithms with different graph partitioning schemes.

among different machines in almost all iterations, in particular, the machine with the longest computation time is two to three times of the mean computation time, which causes other machines to have to wait for the slowest machine, leading to a high synchronization cost in waiting. In contrast, For BPart-C and BPart-S, the distribution of computation time is more balanced in each iteration due to the two-dimensional balanced partition, which saves a lot of waiting time and improves the efficiency of the distributed graph systems.

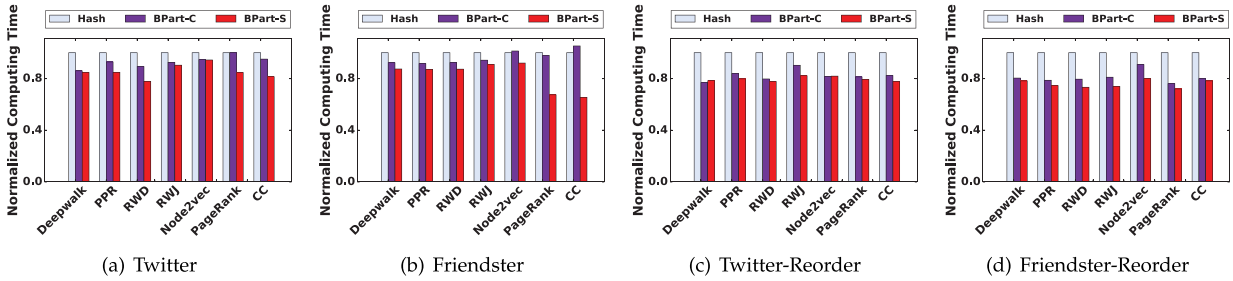
D. Total Running Time of Graph Applications

We now evaluate the total running time of various graph applications when using different graph partitioning schemes. Here we consider seven widely studied graph application algorithms: Deepwalk, PPR, RWD, RWJ, node2vec, PageRank, and Connected Components. The first five algorithms are random walk based graph algorithms, so we use the state-of-the-art distributed random walk system KnightKing as the code base to execute them. The last two algorithms are general iterative graph algorithms, so we use the popular distributed graph system Gemini as the code base to run them. We show the normalized computing time in Fig. 15. The results show that for the first five random walk based graph algorithms, our BPart-C and BPart-S outperform the other three partition algorithms in all situations. Specifically, we can reduce 20%-70% of the total running time compared with Chunk-V and Fennel, and reduce 10%-30% of the total running time compared with Chunk-E. For the other two general iterative graph algorithms, BPart-S always outperforms other partition algorithms in all situations, which can reduce 5%-70% of the total running time compared with Fennel and Chunk-V, and reduce 10%-60% of the total running time compared with Chunk-E. These results show that by balancing both the number of vertices and the number of edges in each subgraph, our BPart-C and BPart-S balance the computing loads, contributing to better performance for distributed graph processing. We would like to point out that BPart-S always outperforms BPart-C. This is because BPart-S generates fewer edge cuts than BPart-C, so that it costs less time for transmitting the computing data. However, this benefit is gained with the cost of a relatively higher partition overhead, and we study the trade-off between the partition overhead and the number of edge cuts in Section IV-F.

E. Comparsion With Hash

Note that the hash-based graph partitioning algorithm, by assigning each vertex to a subgraph according to a randomly generated hash value, can also achieve a certain degree of balance in the two dimensions of the number of vertices and the number of edges, due to the randomness, but it causes lots of edge cuts which may result in high communication overhead. Therefore, we also compare our BPart-C and BPart-S with Hash. We compare the computation time of the seven graph applications when using Hash, BPart-C, and BPart-S for partitioning, and other settings are the same as those in Section IV-D. We show the results under Twitter and Friendster in Fig. 16(a) and (b), respectively, where the x-axis denotes different graph applications, and the y-axis denotes the normalized computation time when using Hash as one. From the results, we can see that even though all three partitioning algorithms achieve two-dimensional balanced partitioning, their computation times are different. Specifically, BPart-S always outperforms Hash, e.g., for the first five random walk based algorithms, BPart-S can decrease 5% to 20% of the total computation time, while for other two iteration based algorithms, BPart-S can reduce the computation time by 20% to 35%. The reduction of the computation time mainly comes from the decrease in the number of edge cuts, because fewer edge cuts incur smaller communication costs during the computation process. On the other hand, BPart-C outperforms Hash in most cases, but it only decreases 2% to 10% of the total computation time for Twitter and Friendster, this is because the number of edge cuts between the partitioned subgraphs for the chunk-based partitioning algorithm heavily depends on the locality among vertex IDs.

To further demonstrate this, we also reorder the graph datasets and make the adjacent IDs closer to each other. To be specific, we first run the Fennel algorithm to partition the original graph into 128 subgraphs, and each subgraph has a good locality. Then we reorder the vertex IDs by making the IDs in each subgraph adjacent. We reorder two datasets, i.e., Twitter and Friendster datasets, and denote them as Twitter-reorder (Twi-R) and Friendster-Reorder (FS-R), respectively. We still compare the computation time of the seven graph applications under the Twitter-reorder and Friendster-reorder datasets, and show the results in Fig. 16(c) and (d), respectively. We can see that BPart-C can always outperform Hash as long as the adjacent



(a) Twitter (b) Friendster (c) Twitter-Reorder (d) Friendster-Reorder

Fig. 16. The normalized computation time of various graph applications when using Hash, BPart-C and BPart-S for graph partition.

TABLE IV
THE TIME OVERHEAD OF PARTITION ALGORITHMS (IN SECONDS)

	LiveJournal	Twitter	Friendster
Chunk-V	0.1739	0.9849	1.572322
Chunk-E	0.1738	1.0045	1.572702
Hash	1.8463	9.5549	15.2458
Fennel	6.4711	55.4845	179.0585
BPart-C	0.4287	2.0680	3.105893
BPart-S	17.1727	89.6942	210.3751

TABLE V
THE RATIO OF THE NUMBER OF EDGE CUTS (I.E., THE EDGES BETWEEN PARTITIONED SUBGRAPHS) TO THE TOTAL NUMBER OF EDGES

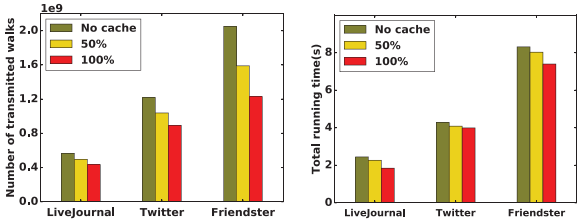
	LJ	Twi	FS	Twi-R	FS-R
Chunk-V	0.5758	0.7475	0.6592	0.6767	0.5774
Chunk-E	0.9033	0.9026	0.7645	0.7075	0.6012
Fennel	0.6491	0.3338	0.3565	0.3209	0.3839
Hash	0.8750	0.8749	0.8750	0.8732	0.8731
BPart-C	0.9007	0.8447	0.8224	0.7185	0.6087
BPart-S	0.7331	0.6226	0.5301	0.6650	0.5641

IDs of the datasets have a better locality, for example, BPart-C can decrease 20% to 25% of the total computation time comparing with Hash for Twitter-Reorder and Friendster-Reorder. In particular, the performance of BPart-C and BPart-S are similar for Twitter-Reorder and Friendster-Reorder, since both can achieve a two-dimensional balanced partition and have a similar number of edge cuts. Therefore, if the adjacent IDs of the graph datasets have a good locality, we recommend using BPart-C as the partitioning algorithm for distributed graph systems, because BPart-C has a small partition overhead. We also investigate the trade-off between the partition overhead and the number of edge cuts in the next subsection.

F. Trade-Off Between Partition Overhead and Edge Cuts

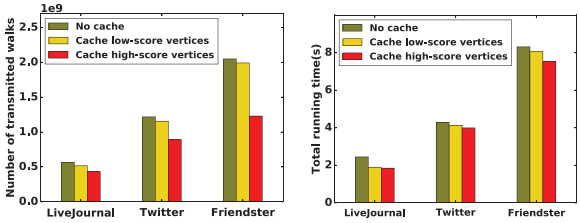
Partition overhead: We first show the partition overheads of different partition algorithms, including Chunk-V, Chunk-E, Hash, Fennel, and our BPart-C, BPart-S. Specifically, we count the time cost of partitioning the three real-world graphs into eight subgraphs, respectively. As shown in Table IV, we can see that the chunk-based partition algorithms, i.e., Chunk-V, Chunk-E, and BPart-C, cost much less time to complete the partitioning process compared with the stream-based partition algorithms, i.e., Fennel and BPart-S. Hash takes less time than the stream-based partition algorithms, but costs more time than the chunk-based partition algorithms. This is because computing the score for each vertex required by the stream-based partition algorithms is more time-consuming than generating a random number by hash, while computing hash is more time-consuming than simply doing chunking according to the number of vertices or edges in the chunk-based partition algorithms. In addition, we can also observe that our proposed BPart-C and BPart-S cost relatively more time than the corresponding baselines, due to the multi-layer combination strategy, which may need multiple rounds of combination to realize a two-dimensional balanced partition, thus bringing a higher partition overhead. We believe that this overhead is acceptable because we can save a lot of running time in graph analytic tasks and improve the efficiency of distributed graph systems by making the computation more balanced. We point out that, the partition is usually executed in preprocess, and it only needs to execute once for all graph analytic tasks. Therefore, it is generally acceptable to cost hundreds of seconds to partition the graph in the distributed graph systems.

Number of edge cuts: Now we study the amount of edge cuts, i.e., the number of edges between different subgraphs, brought by different graph partitioning algorithms, which directly impact the communication cost in distributed graph processing. We count the ratio of the number of edge cuts, i.e., the number of edges between different subgraphs divided by the number of total edges in the whole graph, of the five graph datasets, where Twi-R and FS-R are the reordered versions of Twitter and Friendster as introduced in Section IV-E, by using different graph partitioning algorithms, and show the results in Table V. We can see those stream-based partition algorithms generally have fewer edge cuts. For example, Fennel has only around 35% edge cuts for Twitter and Friendster, but Chunk-V and Chunk-E have 65%–90% edge cuts. Note that BPart-C and BPart-S have more edge cuts compared with their corresponding baselines, because they partition the graph into smaller pieces in the partition phase, and combine these smaller pieces into larger subgraphs. However, BPart-C and BPart-S can achieve two-dimensional balanced partitioning, while their corresponding baselines achieve balance in only one dimension. Hash usually has the most edge cuts compared with other algorithms, and it always has around 87% edge cuts for all datasets, but Fennel and BPart-S have around only 35% and 55% edge cuts, respectively.



(a) Total number of transmitted walks (b) Total running time

Fig. 17. The impact of caching size by varying the number of cached edges (50% or 100% of the number of edges of the subgraph).



(a) Total number of transmitted walks (b) Total running time

Fig. 18. The impact of different caching policies (the number of cached edges is fixed).

On the other hand, for chunk-based partitioning algorithms, including Chunk-V, Chunk-E, and BPart-C, the number of edge cuts largely depends on the locality of vertices with adjacent IDs. For example, in the reordered graphs Twi-R and FS-Rr, the ratio of the number of edge cuts is around 65%, which is much smaller than 83% in the original Twitter and Friendster without reordering.

To sum up, the lower ratio of the number of edge cuts brings smaller communication costs during the computation. Therefore, there is a trade-off between the partition overhead and the communication cost during the computation, so we propose two versions of BPart, namely BPart-C and BPart-S, to meet the needs of different scenarios. Note that both of them can achieve two-dimensional balanced partitioning with low synchronization costs during the distributed graph computation, as illustrated in Section IV-C. Specifically, the chunk-based algorithm BPart-C achieves very fast graph partitioning but has the larger amount of edge cuts, which brings higher communication cost during distributed graph computing. While the stream-based algorithm BPart-S has a larger partition overhead and a lower ratio of the number of edge cuts, which helps to reduce communication costs. In pursuit of efficient graph analysis, we recommend using BPart-S for graph partitioning to minimize synchronization and communication costs. Whereas BPart-C is preferred if the graph already has good locality and well-connected vertices are assigned adjacent IDs.

G. Impact of Caching

We now study the impact of the caching scheme based on our partition algorithms. In the interest of space, we only consider BPart-S as it has fewer edge cuts. We first study the impact of cache size by evaluating the performance when caching different numbers of edges. We load the vertices with higher caching score presented in Section III-D, until the number of edges of all loaded vertices reaches the pre-defined threshold. In this experiment, we consider two cases in which the number of cached edges is set as 50% and 100% of the number of edges of the local subgraph, respectively. We also include the results of no cache for comparison. We start $5 \times |V|$ simple random walks and let each walk move 10 steps. Fig. 17(a) counts the total number of transmitted walks, and the results show that when we cache more edges, the communication overhead is greatly reduced. Fig. 17(b) further shows the corresponding total running time, and we can see that as we cache more edges, the total running time also decreases 6%–24%.

We also study the impact of different caching policies, e.g., by caching different vertices. We fix the number of cached edges as 100% of the number of edges in the local subgraph and consider two different choices. One is to cache the vertices with high scores in the caching file, and the other is to cache the vertices with low scores. Fig. 18(a) shows the total number of transmitted walks, and we can see that caching the edges of the vertices with high scores is beneficial for reducing the communication overhead. Finally, Fig. 18 shows the total transmitted walks and total running time corresponding to different caching policies, and we see that caching the vertices with high scores can significantly reduce the amount of communication and the total running time, this result justifies the rationale of our caching design.

V. Related Work

Graph processing systems: In recent years, many distributed graph systems have been proposed for processing very large graphs [7], [8], [9], [12], [26], [35], [36], which typically leverage a cluster of machines, each of which handles the local analytic tasks and then communicates with other machines to proceed the analytics tasks. Pregel [7] is the first work to focus on distributed graph processing, and proposes a vertex-centric BSP computation model. Since then, many works follow this model to further improve the efficiency of distributed graph systems. For example, PowerGraph [8], PowerLyra [26], and Gemini [14] design new graph partition strategies and computation models to efficiently process graphs with scale free nature. KnightKing [6] proposes a walk-centric BSP computation model to process the random walk applications. LiveGraph [37] optimizes graph storage for transactional graph processing situations. GraphScope [38] exposes a unified programming interface with various graph computations. And Mycelium [39] provided distributed queries with private protection, etc. However, these works pay no attention to the imbalanced computing loads between machines due to the imbalanced graph partitions. Different from them, BPart targets to achieve the balance of computing loads through two-dimensional balanced partitioning, thereby greatly reducing the synchronization overhead and improving the efficiency of distributed graph systems.

In addition to distributed graph systems, a number of out-of-core single-machine graph processing systems are also proposed to handle large graphs [15], [27], [40], [41], [42], [43], [44], [45]. They store the graph data on external storage devices, like SSDs, and iteratively load a subgraph into memory and perform computations associated with that subgraph. Furthermore, some in-memory single-machine graph processing systems have been proposed to solve the problem of random access in graph analytics [46], [47]. The goal of these systems is to effectively utilize high-speed storage devices, such as L1 cache, L2 cache, and L3 cache, to improve the computational efficiency of graph analysis tasks. Besides, some graph systems focus on dynamic graph processing [48], [49], and they adopt hybrid storage to store static graph data and evolving graph data.

Graph partition strategies: Numerous attempts have also been made to enhance the efficiency of graph partitioning and speed up graph processing, which fall into two main types:vertex-cut partitioning algorithms and edge-cut partitioning algorithms The vertex-cut partitioning algorithms [8], [26], [50], [51], [52] split the edge set into multiple disjoint partitions, and cut the vertices that have edge connections with more than one subgraph. Generally, each subgraph will store a copy of these vertices' information to enable computation, thus introducing a lot of redundant data. Edge-cut partitioning algorithms [21], [24], [53], [54], [55], [56], [57], [58], [59], [60], [61], [62], [63], [64], [65], [66], [67] are more commonly used. They split the vertex set into multiple disjoint partitions and cut the edges that connect the vertices of two different partitions. Typically, the graph computations through these edge cuts are transmitted between subgraphs through the network in distributed graph processing systems.

Note that in practical distributed systems, stream-based partitioning, which assigns equal numbers of vertices or edges sequentially as partitions by taking them as streams, is widely used [6], [15], [43]. However, these algorithms can only achieve balanced partitioning in one dimension, i.e., either the number of vertices or edges. By randomly assigning each vertex to a subgraph, hash-based partitioning can achieve balanced partitioning in two dimensions, but it results in many edge cuts and high communication costs in distributed graph computation. Besides, GD [17] uses gradient descent to split a graph into two subgraphs, and it can also achieve balanced partitioning in two dimensions, but it is very time-consuming and can only partition a graph into powers of two subgraphs. Unlike them,BPart aims to divide the graph into any number of subgraphs, while achieving two-dimensional balanced graph partitioning with few edge cuts between subgraphs.

VI. Conclusion

In this paper, we propose a two-dimensional balanced graph partitioning scheme BPart, which realizes balance for both the number of vertices and the number of edges of the partitioned subgraphs. We also propose two graph partitioning algorithms, BPart-C and BPart-S, and integrate them into distributed graph systems. Evaluation results show that the computing load is well-balanced across machines, which greatly reduces the overall running time of graph applications. Besides, we also propose a caching scheme which enables the partitioned subgraphs to have certain redundancies, thereby further reducing the communication cost for distributed graph computation.

Acknowledgment

In this journal version, we developed two enhanced partition algorithms BPart-C and BPart-S, and also proposed a neighbor-aware caching scheme to reduce the communication overhead for distributed graph analysis. We added extensive experiments to validate the effectiveness and efficiency of BPart-C, BPart-S and the caching scheme

References

[1] S. Lin et al., "Towards fast large-scale graph analysis via two-dimensional balanced partitioning," in *Proc. 51st Int. Conf. Parallel Process.*, 2022, pp. 1–11.
[2] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós, "Towards scaling fully personalized PageRank: Algorithms, lower bounds, and experiments," *Internet Math.*, vol. 2, no. 3, pp. 333–358, 2005.
[3] A. Kyrola, "DrunkardMob: Billions of random walks on just a PC," in *Proc. 7th ACM Conf. Recommender Syst.*, 2013, pp. 257–264.
[4] G. Jeh and J. Widom, "SimRank: A measure of structural-context similarity," in *Proc. 8th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2002, pp. 538–543.
[5] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 855–864.
[6] K. Yang, M. Zhang, K. Chen, X. Ma, Y. Bai, and Y. Jiang, "KnightKing: A fast distributed graph random walk engine," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 524–537.
[7] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[8] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[9] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.
[10] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "HAMA: An efficient matrix computation with the MapReduce framework," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 721–726.
[11] C. Avery, "Giraph: Large-scale graph processing infrastructure on hadoop," in *Proc. Hadoop Summit*, vol. 11, no. 3, pp. 5–9, 2011.
[12] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "think like a vertex" to "think like a graph"," in *Proc. VLDB Endowment*, vol. 7, no. 3, pp. 193–204, 2013.
[13] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *Proc. 25th Int. Conf. Sci. Statist. Database Manage.*, 2013, pp. 1–12.
[14] X. Zhu, W. Chen, W. Zheng, and X. Ma, "Gemini: A computation-centric distributed graph processing system," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 301–316.
[15] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 375–386.
[16] N. Pržulj, D. G. Corneil, and I. Jurisica, "Modeling interactome: Scale-free or geometric?," *Bioinformatics*, vol. 20, no. 18, pp. 3508–3515, 2004.
[17] D. Avdiukhin, S. Pupyrev, and G. Yaroslavtsev, "Multi-dimensional balanced graph partitioning via projected gradient descent," in *Proc. VLDB Endowment*, vol. 12, no. 8, pp. 906–919, 2019.
[18] Twitter, 2010. [Online]. Available: https://law.di.unimi.it/webdata/twitter-2010/

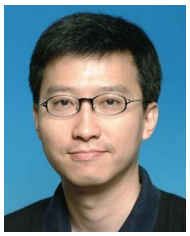
[19] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2014, pp. 701–710.
[20] R.-H. Li, J. X. Yu, X. Huang, and H. Cheng, "Random-walk domination in large graphs," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 736–747.
[21] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342.
[22] Friendster, 2013. [Online]. Available: http://konect.cc/networks/friendster/
[23] LiveJournal, 2006. [Online]. Available: http://konect.cc/networks/livejournal-groupmemberships/
[24] Y. Akhremtsev, P. Sanders, and C. Schulz, "High-quality shared-memory graph partitioning," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2710–2722, Nov. 2020.
[25] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, pp. 496–509, 2011.
[26] R. Chen, J. Shi, Y. Chen, B. Zang, H. Guan, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," *ACM Trans. Parallel Comput.*, vol. 5, no. 3, pp. 1–39, 2019.
[27] R. Wang, Y. Li, H. Xie, Y. Xu, and J. C. Lui, "GraphWalker: An I/O-efficient and resource-friendly graph analytic system for fast and scalable random walks," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2020, Art. no. 38.
[28] I. Rhee, A. Warrier, M. Aia, J. Min, and M. L. Sichitiu, "Z-MAC: A hybrid MAC for wireless sensor networks," *IEEE/ACM Trans. Netw.*, vol. 16, no. 3, pp. 511–524, Jun. 2008.
[29] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, 1989.
[30] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 261–276.
[31] R. K. Jain et al., "A quantitative measure of fairness and discrimination," Eastern Res Lab., Digital Equipment Corporation, Hudson, MA, vol. 21, no. 4, pp. 1–37, 1984.
[32] R. Hussein, D. Yang, and P. Cudré-Mauroux, "Are meta-paths necessary? Revisiting heterogeneous graph embeddings," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, 2018, pp. 437–446.
[33] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web," Stanford InfoLab, Tech. Rep. 1999-66, Nov. 1999. [Online]. Available: http://ilpubs.stanford.edu:8090/422/
[34] L. He, Y. Chao, K. Suzuki, and K. Wu, "Fast connected-component labeling," *Pattern Recognit.*, vol. 42, no. 9, pp. 1977–1987, 2009.
[35] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-Miner: An efficient task-oriented graph mining system," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–12.
[36] A. Khan, G. Segovia, and D. Kossmann, "On smart query routing: For distributed graph querying with decoupled storage," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 401–412.
[37] X. Zhu et al., "LiveGraph: A transactional graph storage system with purely sequential adjacency list scans," 2019, *arXiv: 1910.05773*.
[38] W. Fan et al., "GraphScope: A unified engine for big graph processing," in *Proc. VLDB Endowment*, vol. 14, no. 12, pp. 2879–2892, 2021.
[39] E. Roth, K. Newatia, Y. Ma, K. Zhong, S. Angel, and A. Haeberlen, "Mycelium: Large-scale distributed graph queries with differential privacy," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 327–343.
[40] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.
[41] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.
[42] K. Vora, G. Xu, and R. Gupta, "Load the edges you need: A generic I/O optimization for disk-based graph processing," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2016, pp. 507–522.
[43] H. Liu and H. H. Huang, "Graphene: Fine-grained IO management for graph computing," in *Proc. 15th Usenix Conf. File Storage Technol.*, 2017, pp. 285–299.
[44] Z. Ai, M. Zhang, Y. Wu, X. Qian, K. Chen, and W. Zheng, "Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 125–137.
[45] S. Wang et al., "NosWalker: A decoupled architecture for out-of-core random walk processing," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 466–482.
[46] S. Sun, Y. Chen, S. Lu, B. He, and Y. Li, "ThunderRW: An in-memory graph random walk engine," in *Proc. VLDB Endowment*, vol. 14, pp. 1992–2005, 2021.
[47] K. Yang, X. Ma, S. Thirumuruganathan, K. Chen, and Y. Wu, "Random walks on huge graphs at cache efficiency," in *Proc. ACM SIGOPS 28th Symp. Operating Syst. Princ.*, 2021, pp. 311–326.
[48] P. Kumar and H. H. Huang, "GraphOne: A data store for real-time analytics on evolving graphs," *ACM Trans. Storage*, vol. 15, no. 4, pp. 1–40, 2020.
[49] R. Wang, S. He, W. Zong, Y. Li, and Y. Xu, "XPGraph: XPline-friendly persistent memory graph stores for large-scale evolving graphs," in *Proc. 55th IEEE/ACM Int. Symp. Microarchitecture*, 2022, pp. 1308–1325.
[50] F. Petroni, L. Querzoni, K. Daudjee, S. Kamali, and G. Iacoboni, "HDRF: Stream-based partitioning for power-law graphs," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manage.*, 2015, pp. 243–252.
[51] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 1673–1681.
[52] M. Hanai, T. Suzumura, W. J. Tan, E. S. Liu, G. Theodoropoulos, and W. Cai, "Distributed edge partitioning for trillion-edge graphs," in *Proc. VLDB Endowment*, vol. 12, no. 13, pp. 2379–2392, 2019.
[53] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *Proc. IEEE 30th Int. Conf. Data Eng.*, 2014, pp. 568–579.
[54] F. Bourse, M. Lelarge, and M. Vojnovic, "Balanced graph edge partition," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2014, pp. 1456–1465.
[55] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2012, pp. 1222–1230.
[56] G. Karypis and V. Kumar, "A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," Dept. Comput. Sci. Eng., Univ. Minnesota, Army HPC Res. Center, Minneapolis, MN, vol. 38, no. 2, pp. 1–31, 1998.
[57] W. E. Donath and A. J. Hoffman, "Lower bounds for the partitioning of graphs," in *Selected Papers of Alan J. Hoffman: With Commentary*, Singapore: World Scientific, 2003, pp. 437–442.
[58] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.
[59] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Des. Autom. Conf.*, 1982, pp. 175–181.
[60] M. Holtgrewe, P. Sanders, and C. Schulz, "Engineering a scalable high quality graph partitioner," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.
[61] S. Gong, Y. Zhang, and G. Yu, "HBP: Hotness balanced partition for prioritized iterative graph computations," in *Proc. IEEE 36th Int. Conf. Data Eng.*, 2020, pp. 1942–1945.
[62] W. Fan, M. Liu, P. Lu, and Q. Yin, "Graph algorithms with partition transparency," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 2, pp. 1554–1566, Feb. 2023.
[63] C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *Proc. IEEE 33rd Int. Conf. Data Eng.*, 2017, pp. 1083–1094.
[64] G. Karypis and V. Kumar, "Parallel multilevel series k-way partitioning scheme for irregular graphs," *SIAM Rev.*, vol. 41, no. 2, pp. 278–300, 1999.
[65] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Proc. Eur. Symp. Algorithms*, Springer, 2011, pp. 469–480.
[66] C. Walshaw and M. Cross, "JOSTLE: Parallel multilevel graph-partitioning software–An overview," *Mesh Partitioning Techn. Domain Decomposition Techn.*, vol. 10, pp. 27–58, 2007.
[67] C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, no. 6/8, pp. 318–331, 2008.

Shuai Lin received the bachelor's degree from the University of Electronic Science and Technology of China. He is currently working toward the PhD degree with the School of Computer Science and Technology, University of Science and Technology of China. His research interests in graph processing systems. He also interests in graph algorithms and graph databases.

Rui Wang received the bachelor's degree in computer science from Donghua University in 2016 and the PhD degree in computer science from the University of Science and Technology of China (USTC) in 2021. She is a postdoctoral fellow with the School of Computer Science and Technology, Zhejiang University (ZJU). Her research interests lie in graph computing and storage systems. She is also interested in databases, key-value systems, machine learning systems, etc.

Yongkun Li received the BEng degree in computer science from USTC in 2008 and the PhD degree in computer science and engineering from The Chinese University of Hong Kong in 2012. He is currently an associate professor with the School of Computer Science and Technology, University of Science and Technology of China. His research mainly focuses on memory and file systems, including key-value systems, distributed file systems, as well as memory and I/O optimization for virtualized systems.

Yinlong Xu received the BS degree in mathematics from Peking University in 1983 and the MS and PhD degrees in computer science from the University of Science and Technology of China (USTC) in 1989 and 2004, respectively. He is currently a professor with the School of Computer Science and Technology, USTC, and is leading a research group in doing some storage and high performance computing research. His research interests include network coding, storage systems, etc. He received the Excellent PhD Advisor Award of Chinese Academy of Sciences, in 2006.



John C. S. Lui (Fellow, IEEE) received the PhD degree in computer science from the University of California at Los Angeles. He is currently the Choh Ming Li chair professor with the CSE Department, The Chinese University of Hong Kong. His current research interests include machine learning, online learning (e.g., multiarmed bandit and reinforcement learning), network science, future Internet architectures and protocols, network economics, network/system security, and large-scale storage systems. He is an elected member of the IFIP WG 7.3, and a senior research fellow of the Croucher Foundation. He is a fellow of ACM and Hong Kong Academy of Engineering Sciences. He received various departmental teaching awards and the CUHK Vice-Chancellor Exemplary Teaching Award.