# Towards Fast Large-scale Graph Analysis via Two-dimensional Balanced Partitioning

Shuai Lin
University of Science and Technology of China
HeFei, China
shuailin@mail.ustc.edu.cn

Rui Wang
Zhejiang University
Hangzhou, China
rwang21@zju.edu.cn

Yongkun Li
University of Science and Technology of China
HeFei, China
ykli@ustc.edu.cn

Yinlong Xu
Anhui Province Key Laboratory of High Performance Computing, USTC
HeFei, China
ylxu@ustc.edu.cn

John C.S. Lui
The Chinese University of Hong Kong
Hong Kong, China
cslui@cse.cuhk.edu.hk

Fei Chen
Huawei
Nanjing, China
chenfei57@huawei.com

Pengcheng Wang
Huawei
Nanjing, China
wangpengcheng25@huawei.com

Lei Han
Huawei
Nanjing, China
phoebe.han@huawei.com

## ABSTRACT

Distributed graph systems often leverage a cluster of machines by partitioning a large graph into multiple small-size subgraphs. Thus, graph partition usually has a significant impact on the performance of distributed graph systems. However, existing widely used partition schemes in practical graph systems can realize a good balance only in one dimension, e.g., either the number of vertices or the number of edges, and they may also incur lots of edge cuts. To address the problem, we develop BPart, which adopts a two-phase partition scheme to realize two-dimensional balance for both vertices and edges. Its core idea is to first partition the original graph into more small pieces than the cluster scale, and combine the partition to realize desired properties, then selectively combine the small pieces to construct larger subgraphs to generate two-dimensional balanced partition. We implement BPart into two open-source distributed graph systems, Gemini [58] and KnightKing [57]. Results show that BPart realizes good balance in both dimensions, and also significantly reduces the number of edge cuts. As a result, BPart reduces the total running time of various graph applications by 5% - 70%, compared to multiple existing partition schemes, e.g., Chunk-V, Chunk-E, Fennel, and Hash.

*ICPP '22, August 29-September 1, 2022, Bordeaux, France*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545060

**ACM Reference Format:**
Shuai Lin, Rui Wang, Yongkun Li, Yinlong Xu, John C.S. Lui, Fei Chen, Pengcheng Wang, and Lei Han. 2022. Towards Fast Large-scale Graph Analysis via Two-dimensional Balanced Partitioning. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545060

## 1 INTRODUCTION

Graph analysis has received a lot of attentions in both academia and industry in recent years, and many graph analytic algorithms have been proposed to explore useful information over graphs. Examples of widely studied graph algorithms include personalized PageRank [14, 31], SimRank [26], Deepwalk [40], Node2vec [19], and so on. To support the efficient execution of graph algorithms, various graph systems are developed recently. In particular, as graph sizes increase, e.g., many web graphs already contain billions of vertices and hundreds of billions of edges, it is inefficient to analyze such big graphs in a single machine, so distributed graph systems based on a cluster of machines have been developed to handle large graphs, such as PowerGraph [17], GraphX [18], G-Miner [7], Gemini [58], and KnightKing [57], etc.

Distributed graph systems usually partition a large graph into multiple subgraphs and store each subgraph in a single machine of the cluster, then each machine only processes the local subgraph and transmits the analysis tasks to other machines if needed. To coordinate the analysis tasks between machines, the bulk synchronous parallel (BSP) model is often used [5, 17, 18, 37, 46, 49, 57, 58]. In detail, the graph analysis tasks are processed in an iterative fashion, and in each iteration, all machines perform the analysis on the local subgraphs and transmit the computing data to other machines in parallel. When all machines finish the computation and communication, then they go to the next iteration. Note that this BSP model is widely used in many distributed graph systems to support general graph analytic algorithms, including Gemini [58] and distributed random walk system KnightKing [57].

We find that graph partition plays a critical role in distributed graph processing, and it not only affects the balance of computing loads between machines due to the unbalanced vertices or edges between partitioned subgraphs, but also influences the amount of communication traffic due to the large amount of edge cuts (i.e., the edges between partitioned subgraphs). Both of them determine the time cost in each iteration for BSP based graph systems, and finally influence the overall graph processing efficiency. To realize balanced computing loads between machines in each iteration, we observe that the key factor is to enable balanced graph partition in both vertices and edges. The main reason is that the computing loads usually depend on both the number of vertices as well as the number of edges of the subgraph. For example, for many random walk based algorithms, the computing loads are mainly decided by the number of walkers and the number of steps that each walker can move over the subgraph, which are dependent on the number of vertices and edges, respectively. On the other hand, to reduce the communication traffic, the key factor is to minimize the number of edge cuts between subgraphs, as there would be a data transfer between machines if there is an edge cut between the subgraphs stored in the two machines. Thus, it motivates us to develop a two-dimensional balanced graph partition scheme, i.e., to partition the graph into equal sizes on both vertices and edges, and meanwhile, to minimize the the number of edge cuts between partitioned subgraphs.

However, the widely used graph partitioning schemes in existing distributed graph systems usually realizes balance in only one dimension, i.e., either the number of vertices or the number of edges is balanced, or ignore the problem of minimizing edge cuts. For example, one partitioning design is to do chunking in the vertex stream or in the edge stream (see §2.2 for detailed introduction), so it can evenly split the amount of vertices or edges. We call the two design choices Chunk-V and Chunk-E, respectively, and both of them are adopted in recent distributed graph systems, for example, Gemini [58] and GridGraph [60] adopt Chunk-V, and the state-of-the-art distributed random walk system KnightKing [57] adopts Chunk-E. However, these algorithms can realize balance only in one dimension and the other dimension is quite imbalanced. This is because many real-world graphs often exhibit a scale-free nature [42]. That is, the vertex degrees usually follow a power-law distribution and high-degree vertices are easily gathered together in the same subgraph [4]. For example, when partitioning the Twitter [51] graph into 8 subgraphs by using Chunk-V, the number of edges can vary from 61M to 737M. Similarly, for Chunk-E, the number of vertices can differ from 744K to 14M. As a result, the computing loads between machines are highly imbalanced, and this introduces a high synchronization overhead for BSP model as it takes a long time to wait for the slowest machine. For instance, when running KnightKing on an eight-machine testbed, the average waiting time of each machine can be 20%-40% of the total computation time for graph algorithms like Deepwalk (see §4.3). Another simple partition design used in practical systems, e.g., Giraph and Pregel system [5, 37], is to do hashing by randomly assigning each vertex to a subgraph. Even though hash-based design can achieve balanced partition in both dimensions, but it introduces a very large number of edges cuts and thus incurs a large amount of communication traffic. For example, there are around 88% of edges cuts even we only partition into 8 subgraphs under Twitter and Friendster dataset.

To realize balanced graph partition for both vertices and edges, and meanwhile, reduce the number of edge cuts, we develop a two-dimensional balanced graph partition scheme, BPart, and its main idea is to adopt a two-phase partitioning scheme, which first partitions the original graph into many small pieces with desired properties, and then selectively combines them to form the final partition. Specifically, in the partition phase, we aim for relaxing the unbalanced degree in both dimensions simultaneously. To realize it, we follow the idea of stream-based partition in Fennel [50], and when deciding the assignment of each candidate vertex to which subgraph, we leverage a weighted policy to quantify its impact on each subgraph in multiple aspects, e.g., its impact on the increased number of vertices and edges, as well as the number of edge cuts. With the weighted policy, we can reduce the skewness of the distribution of the number of vertices and the distribution of the number of edges of the partitioned subgraphs, and we can also adjust the two distributions to make them be inversely proportional, so that the partitioned small subgraphs can be combined to form larger ones with better balance. In the second phase, we combine small subgraphs into a larger one by leveraging the inversely proportional feature to improve the balance. By applying the two-phase partition process for multiple rounds, we can finally realize the desired balance for both vertices and edges and also limit the number of edge cuts.

To demonstrate the effectiveness and efficiency of BPart, we also implement BPart into two distributed graph systems, Gemini [58] and KnightKing [57], which support general iteration-based graph algorithms such as PageRank [39] and Connected Component, and random walk algorithms, e.g., personalized PageRank (PPR) [14], random walk with jump (RWJ) [23], random walk with domination (RWD) [34], Deepwalk [40] and node2vec [19], respectively. Experiments show that BPart can achieve well balanced partition for both vertices and edges among subgraphs, and also significantly reduces the number of edge cuts compared to hash-based design, e.g., hash generates around 87.5% edge cuts, while BPart reduces this number to 50% in Friendser [15]. As a result, BPart reduces the total running time of various graph applications by 5% - 70%, compared to multiple existing partition schemes, Chunk-V, Chunk-E, Fennel, and Hash (see §4). We will release the source codes in the final paper.

The rest of this paper is organized as follows. In §2, we first introduce the framework of distributed graph computation, then introduce existing graph partitioning schemes, and analyze their limitations to motivate the design of BPart. In §3, we present the main idea and the design details of BPart, and evaluate its performance in §4. Finally, §5 reviews related work and §6 concludes this paper.

## 2 BACKGROUND AND MOTIVATION

In this section, we first introduce the computation framework of distributed graph systems, then introduce the widely used graph partition algorithms, and analyze their limitations to motivate the design of two dimensional balanced partition.

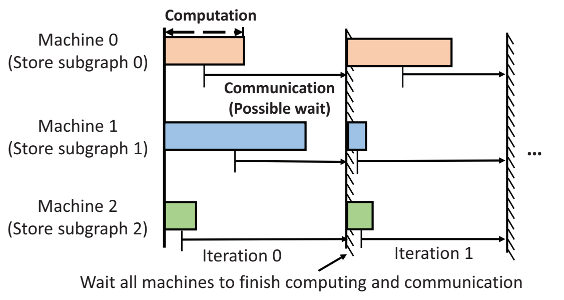

Computation
Machine 0 (Store subgraph 0)
Communication (Possible wait)
Machine 1 (Store subgraph 1)
Machine 2 (Store subgraph 2)
...
Iteration 0
Iteration 1
Wait all machines to finish computing and communication

**Figure 1: Iteration-based computation with BSP.**

## 2.1 Distributed Graph Computation

Distributed graph systems usually adopt an iteration-based computation framework with a bulk synchronous parallel (BSP) model to coordinate the analysis tasks between machines [5, 17, 18, 37, 46, 49, 57, 58]. Figure 1 shows the general idea of the iteration-based BSP model. Specifically, as the graph is partitioned into multiple subgraphs which are stored in different machines, a graph computation task is executed in an iteration-based way, and in each iteration, each machine first processes the local subgraph and then synchronizes with other machines. In particular, for each machine, the execution process within one iteration consists of two phases: computation, and communication. In the computation phase, the analysis task is executed over the local subgraph stored in the machine until no updates can be made, and this greedy-like strategy is to fully utilize the local subgraph for minimizing the communication frequency between machines. When the computation phase ends, the machine enters into the communication phase, in which it needs to first send its own computing data to other corresponding machines for further updates, and then waits to receive the data transmitted from all other machines after they finish their computation. This phase lasts until all machines finish the computation and communication in the current iteration. After that, they go to the next iteration. Note that the computation and communication phases may be processed in a pipelined fashion in some systems, thus amortizing part of the communication cost.

According to the above computation process, it is clear that if the computing loads in an iteration are not balanced, some machines may need to wait for receiving data from other slow machines which are still doing computation, and this significantly degrades system performance and they should be minimized as much as possible. In fact, the waiting time really depends on when the machine itself and the slowest one finish their computation tasks. In other words, the total waiting time of all machines, which we call *t*he synchronization overhead, depends on the balance of the computing loads of all machines. In the ideal case, if all machines process the same amount of computing workloads, then they can immediately receive the data from other machines after finishing their own computation and the send of their data to other machines, then all machines can go to the next iteration.

## 2.2 Graph Partition Algorithms

One simple and widely used graph partition is to treat all vertices or edges as a vertex stream, and then takes vertices or edges from the stream one by one to decide which subgraph it should belong to. Figure 2(a) illustrates the work flow of a simple design, which we call Chunk-V and is used in multiple systems [58, 60]. It sequentially adds the adjacent vertex IDs and their corresponding edges to the same subgraph, until it reaches the balanced indicator, after that it adds the rest of the vertices to another subgraph. It continues the above process until we add all vertices to the subgraphs. Similarly, we can also treat the edges as a stream to do a similar partition, and we call it Chunk-E, which is also used in multiple systems [32, 57], and it is illustrated in Figure 2(b). Note that Chunk-V and Chunk-E are designed to balance either the number of vertices or the number of edges, while they do not take into consideration the number of edge cuts, i.e., the edges between partitioned subgraphs.

To reduce edge cuts, the Fennel algorithm [50] takes each vertex from the vertex stream, and computes a score for each partition, then adds this vertex and its associated edges to the partition with the highest score. Figure 2(c) illustrates the process. The score is defined as follows:

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

where $N(v)$ is the neighbor set of vertex $v$, $V_i$ is the vertex set of the subgraph $G_i$, $\alpha$ and $\gamma$ are two constants. The first term $|V_i \cap N(v)|$ denotes the number of common vertices between $v's$ neighbors and $V_i$, if this number is large, then adding $v$ to the subgraph $G_i$ could minimize the number of 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 like a penalty factor to avoid a large partition continuing to have more vertices, and this penalty factor helps to balance the number of vertices in different partitions.

Another simple partition design is to use hash, and its key idea is to randomly assign each vertex to a subgraph. Its work flow is similar to that of Fennel as shown in Figure 2(c), and the difference is that instead of computing a complicated score, it simply generates a hash value for each vertex to decide which subgraph to assign.

## 2.3 Limitations

**Limitation #1: Inefficiency of 2D balanced partition.** We note that existing algorithms except for hash can only achieve balanced partition in one dimension, either the number of vertices or the number of edges. For example, Chunk-V and Fennel can only balance the number of vertices, however, the number of edges is quite imbalanced; Chunk-E can balance the number of edges, However, the number of vertices is quite imbalanced.

To further demonstrate the result, we run the above introduced three partition algorithms, i.e., Chunk-V, Chunk-E, and Fennel, to show the distributions of the number of vertices and edges. In the interest of space, we take only the Twitter graph as an example, and partition the graph into four subgraphs to run on a four-machine cluster. We observe similar results under more graph datasets (see §4). Figure 3 shows that Chunk-V and Fennel can realize a balanced partition for the number of vertices, but the numbers of edges are highly imbalanced, and the gap, i.e., the difference between the maximum and the minimum numbers of edges, can reach up to 8×. Chunk-E can realize a balanced partition for the number of edges, but the numbers of vertices are highly imbalanced, and the gap can reach up to 13×.

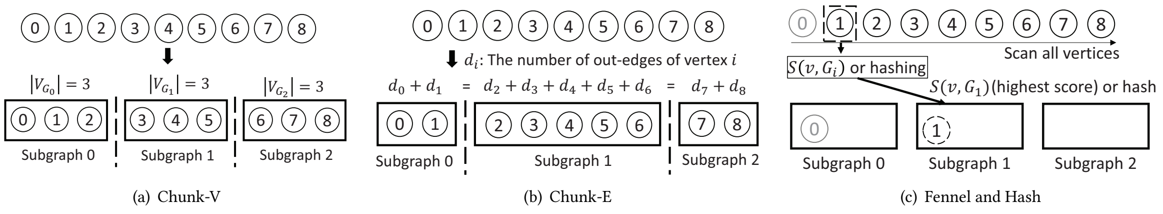


**Figure 2: Illustration on graph partition algorithms: Chunk-V, Chunk-E, Fennel and Hash.**

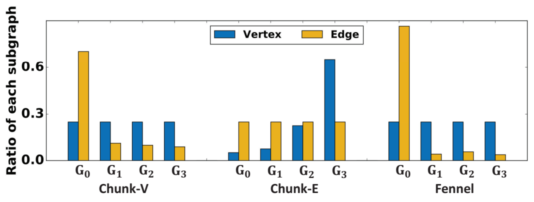


**Figure 3: The ratios of the number of vertices and the number of edges in subgraphs $G_0 - G_3$.**

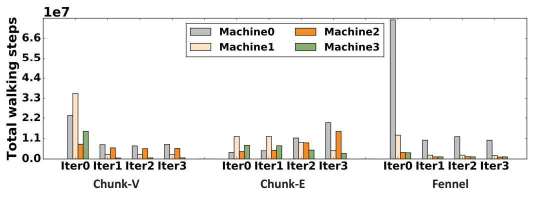


**Figure 4: The distribution of the computing loads between machines in different iterations (Iter0 - Iter3).**

Due to the imbalance of the number of vertices or the number of edges, it inevitably leads to an imbalanced computing loads between machines. To further demonstrate, we take a random walk application as an example, and run experiments to show the computing load distribution. We start five random walks from each vertex over the Twitter graph, and let each walk move four steps. Figure 4 shows the computing load of each machine, which is characterized by using the number of walking steps. We see that the computing loads between machines are highly imbalanced. In particular, for Chunk-V and Fennel, even though the initial numbers of walks started at each machine are balanced in the first iteration due to the balanced vertices, the computing loads are still highly imbalanced, because the walkers can move different numbers of steps due to the imbalanced number of edges.

**Limitation #2: High communication traffic.** Hash based algorithm can realize the balanced partition in both dimensions, however, it faces high edge cuts between subgraphs due to the randomness of assigning vertices to subgraphs. As a result, it causes high communication traffic due to the higher probability of visiting an edge cut. To demonstrate this, we first show the ratio of edge cuts when using different partition algorithms, e.g., Chunk-V, Chunk-E, Fennel, and Hash. As shown in Figure 5(a), when partitioning into 8 subgraphs, we can see that Chunk-E and Hash contain around 90% edge cuts, that is, around 90% edges are crossing edges between partitioned subgraphs. We also see that Fennel significantly reduces the number of edges cuts, which is only around 30%. To further show the impact of edge cuts, we also run a random walk application as an example to show the communication traffic, which is defined as the number of message walks, i.e., the number of walks being transmitted. We also start five random walks from each vertex and let each walk move four steps. Figure 5(b) shows that Chunk-E and Hash have more than 2× walks being transmitted compared with Fennel as they contain more edge cuts.

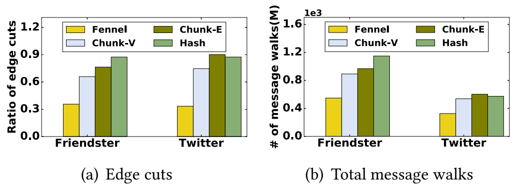


**Figure 5: The ratio of edge cuts and total message when using different partition algorithms.**

## 3 DESIGN OF BPART

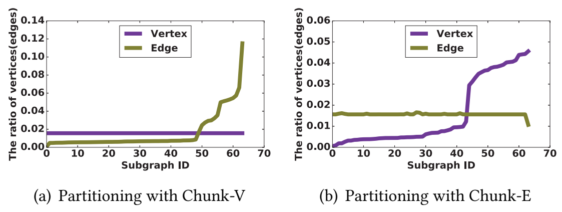
In this section, we first introduce a key observation on the balanced degree of graph partition, then present the main idea of our new partition scheme BPart, which targets for two-dimensional balanced partition by using a two-phase partition scheme. After that, we present the design details in each phase.

### 3.1 Observation & Main Idea

Note that for graph partition, the balanced degree of the partitioned subgraphs is characterized in two dimensions: the number of vertices and the number of edges of the subgraphs.

**Observation.** Balancing the measure in one dimension often results in a highly imbalanced distribution in the other dimension. Specifically, if the distribution of the number of vertices of the partitioned subgraphs is well balanced, then the distribution of the number of edges may be highly skewed, and vice versa. The rationale of this observation can be justified as follows. As natural graphs like social networks usually have a scale-free nature, e.g., the degrees of vertices follow a power-law distribution. As a result, if the number of vertices are evenly partitioned, then the subgraphs containing the high-degree vertices usually have many more edges than other subgraphs. That is, the distribution of the number of edges of the partitioned subgraphs is highly skewed.

To further validate the observation, we conduct experiments to show the skewness of the distribution for the imbalanced dimension on real-world graphs. In the interest of space, we only show the results on Twitter [51], while we also observe similar trend



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

**Figure 6: Distribution of $|V_i|$'s and $|E_i|$'s.**

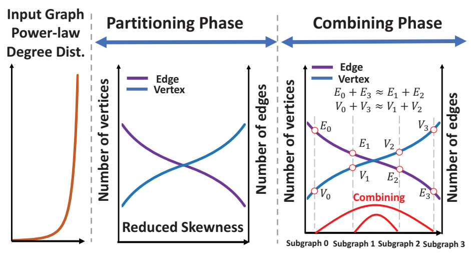


**Figure 7: Idea of two-phase partition in BPart.**

on other graphs like Friendster [15] and LiveJournal [36]. Here we partition the graph into 64 small subgraphs with the vertices balanced algorithm, Chunk-V, and the edges balanced algorithm, Chunk-E. Figure 6 shows the 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. 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. We see that Chunk-V balances the number of vertices, while the number of edges is highly imbalanced. Similarly, Chunk-E balances $|E_i|$'s, while $|V_i|$'s are highly imbalanced.
**Remark:** The observation also implies that due to the highly skewed distribution in the imbalanced dimension, i.e., either the number of vertices or the number of edges, it is also hard to realize balance by simply combining subgraphs.

**Main idea: Two-phase partition.** To realize balanced partition in both dimensions, including the number of vertices and the number of edges of the partitioned subgraphs, we develop a new partition scheme, BPart, and its key idea is to adopt a *t*wo-phase partition, which includes a partitioning phase and a combing phase.

As illustrated in Figure 7, in the partitioning phase, instead of targeting for a balanced partition, which realizes perfect balance in one dimension, but usually makes the other dimension highly imbalanced, our goal is to reduce the skewness of the distributions in both dimensions. The rationale is that by generating subgraphs without extremely large number of vertices or edges, it is possible to combine these subgraphs to realize balance in two dimensions for the final output. To realize it, our idea is to leverage a weighted policy to reflect the influences of both the vertices and edges during the partition. With this weighted policy, we can adjust the distributions of both the number of vertices and edges, and coordinate the two distributions to make them inversely proportional, i.e., the subgraph with fewer vertices has more edges. In the combining phase, we can selectively combine two subgraphs into a larger one according to the distributions of the number of vertices and edges, thus making the newly combined subgraphs have more balanced distributions in both dimensions.

To realize the above two-phase partition, there are two challenges we need to address. First, how to design a weighted policy in the partition process to take into account both the influences of vertices and edges, so that the distribution of the number of vertices and the distribution of the number of edges of the partitioned subgraphs can be adjusted as desired, e.g., to make them be inversely proportional as we need. Second, how to combine small subgraphs into larger ones, and we may need multiple rounds of combinations, so as to finally realize the desired balance for both vertices and edges. In the following, we will introduce the design details of BPart to solve the two challenges.

## 3.2 Partitioning Phase Design

We follow the idea of stream-based partition in Fennel (see §2.2), and design a new balance indicator to guide the graph partition process. The key idea is to leverage a weighted approach to integrate the influences of both vertices and edges, mathematically, we design the 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 weight of $|V_i|$ and $|E_i|$, $\bar{d}$ is the average degree of the graph. With this weighted balance indicator, during the partition, the goal is to make $\mathcal{W}_i$'s be equal. In particular, $c = 0$ corresponds to the edge balance indicator, which makes the number of edges of each subgraph equal, and $c = 1$ corresponds to the vertex balance indicator, which makes the number of vertices of each subgraph equal. For the weighting factor, we use equal weights of $|V_i|$ and $|E_i|$ based on our empirical study, i.e., we set $c = \frac{1}{2}$ by default.

Based on the new indicator $\mathcal{W}_i$, we follow the stream-based partition work flow, and compute the score of vertex $v$ for subgraph $G_i$, i.e., $S(v, G_i)$, as follows:

$$S(v, G_i) = |V_i \cap N(v)| - \alpha\gamma\mathcal{W}_i^{\gamma-1}, \tag{2}$$

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 large number of $|V_i \cap N(v)|$ denotes less edge cuts between subgraphs. $\alpha$ and $\gamma$ are constants used for adjusting the weights of the edge-cut number and the balanced degree.

By using the weighted policy in partitioning, we can make the skewness of the distributions of the number of vertices and edges of the partitioned subgraphs be reduced, and in particular, the number of vertices could be inversely proportional to the number of edges. The rationale is that as $\mathcal{W}_i$'s are equal, then the subgraph 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 still use the same setting as that in Figure 6, and the results are presented in Figure 8. We can see that neither $|V_i|$ nor $|E_i|$ is balanced among subgraphs, while the skewness is decreased compared with the results in Figure 6, and besides, the two distributions of $|V_i|$'s and $|E_i|$'s are inversely proportional to each other. This implies that we could realize the desired balance for

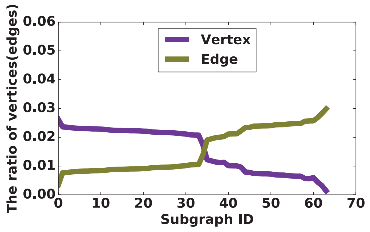


**Figure 8: The ratios of $|V_i|$'s and $|E_i|$'s with the weighted policy (subgraphs are reordered).**

both vertices and edges through appropriate combinations of these small subgraphs.

## 3.3 Combing Phase Design

Suppose that we want to partition a graph into $N$ subgraphs and aim to achieve a two-dimensional balanced partition for both vertices and edges. We can first partition the graph into $2 \times N$ small subgraphs based on the weighted balance indicator defined in Equation (1), then sort these small subgraphs by the number of vertices in each subgraph, i.e., $|V_i|$. According to the inversely proportional nature, the subgraph with a smaller $|V_i|$ generally has a larger $|E_i|$, and vice versa. Therefore, at each time, we can combine the subgraph with the least number of vertices (also with the most number of edges) and the subgraph with the most number of vertices (also with the least number of edges) into a larger subgraph, and keep doing this combination for the remaining subgraphs. Finally, we can get $N$ larger subgraphs, and these combined $N$ subgraphs have more balanced vertices and edges.

**Multi-layer combination.** We note that only one shot of combination is usually not enough to achieve a good balance as desired. Luckily, the two distributions of $|V_i|$ and $|E_i|$ of the combined subgraphs are still inversely proportional, and this guides us to design a *m*ulti-layer combination strategy, which keeps doing the combination in multiple rounds until the balanced condition is satisfied. Specifically, as illustrated in Figure 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 subgraphs and go into the next layer to do another round of combination. For example, in the second layer, suppose that we need to partition the remaining graph into $N_r$ subgraphs, i.e., $N - N_r$ subgraphs combined in the first layer already get balanced. As shown in Figure 9, we first partition the remaining graph into $4 \times N_r$ small subgraphs (here $N_r = N - 1$), and then do the combination in two rounds. In the first round, $4 \times N_r$ small subgraphs are combined to generate $2 \times N_r$ subgraphs, then in the second round, these $2 \times N_r$ subgraphs are further combined to generate the final $N_r$ subgraphs. After that, we check the balanced degree of $|V_i|$ and $|E_i|$ of each combined subgraph once again and repeat the above process until all combined subgraphs are balanced for both vertices and edges. Generally, we can get the desired balanced partition for both vertices and edges after two or three rounds of combinations based on our experiments.

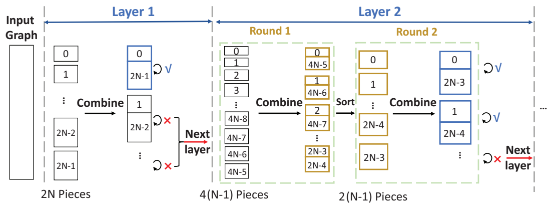


**Figure 9: Multi-layer combination.**

**Connectivity of the combined subgraphs.** With the combination-based graph partition scheme, as the final partitioned subgraph is combined with multiple small-size subgraphs, one may worry about whether the small subgraphs which belong to the final combined one are still well connected, i.e., whether they still have enough number of edge connections between these small subgraphs or not. We would like to point out that even we partition the graph into many small subgraphs, there are still a lot of edge connections between any two of these small subgraphs. To demonstrate this, we take Friendster, which has 3.6 billion edges in total, as an example, and partition it into 64 small subgraphs, and we find that there are at least 50,000 edge connections between any two subgraphs, and in most of the cases, the above number is 500,000. This result implies that it is safe to guarantee that the two-phase partition in BPart will not make the combined subgraphs disconnected.

## 4 EVALUATION

To demonstrate the effectiveness and efficiency of BPart, we compare with four commonly used graph partition algorithms, Chunk-V, Chunk-E, Fennel, and Hash. In the experiments, we take both KnightKing [57], which is the state-of-the art distributed graph system for running random walk algorithms, and Gemini [58], which also supports other graph algorithms, as the code bases, and integrate all partition algorithms into the systems for comparison. We first show the balanced degree in both dimensions by comparing with Chunk-V, Chunk-E and Fennel, then we compare the balance of the computing loads and the total running time for various graph algorithms. After that, we show the comparison with hash based partition, including the number of edge cuts and the running time for different graph algorithms.

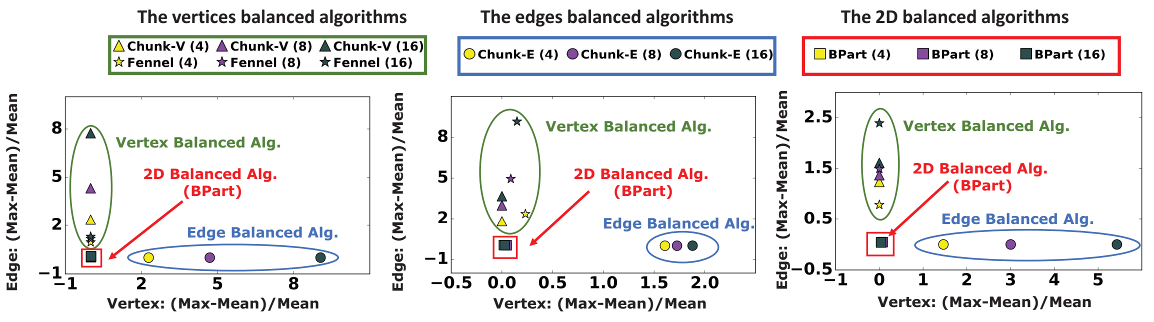
## 4.1 Experiment Setup

**Testbed.** Our testbed consists of a cluster of eight machines, which are connected with 56 Gbps Ethernet. Each machine is equipped with two 24-core Intel Xeon CPU E5-2650 v4 processors and 64GB DRAM, and runs CentOS. In the experiments, we may vary the number of machines being really used so as to study the impact of the cluster scale.

**Datasets**. We consider three graph datasets with different scales. Table 1 shows the statistics of the graph datasets. All these graphs are real-world social networks, and they are also widely used to evaluate many graph systems [8, 17, 55, 57].

**Metrics of balanced degree.** We use the following two metrics to characterize the balanced degree of subgraphs:

- **Bias**: We define bias as the difference between the maximum value and the mean value, normalized by the mean value,



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

**Figure 10: Balanced degree measured with the bias metric for both the number of vertices and the number of edges. Note that the numbers in the parentheses denote the number of partitioned subgraphs.**

| Graphs | # of Vertices | # of Edges | Avg Degree |
|---|---|---|---|
| LiveJournal [36] | 7.5M | 225M | 29.99 |
| Twitter [51] | 41.39M | 1.48B | 35.72 |
| Friendster [15] | 65.60M | 3.6B | 54.87 |

**Table 1: Statistics of the graph datasets.**

mathematically, for a set of $n$ values $\{x_i | i = 0, 1, ..., n-1\}$, the bias is defined as

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

where $\max(x_i)$ is the maximum value of $x_i$'s, and $\text{mean}(x_i)$ denotes the mean. This bias metric is chosen because the computation time in each iteration is determined by the slowest machine, i.e., the machine with the maximum computing load. All other machines need to wait for the slowest machine to finish its computation before going to the next iteration. Note that $x_i$'s can be either the numbers of vertices or the numbers of edges of the partitioned subgraphs.

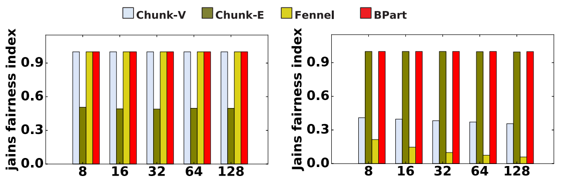
- **Fairness**: We use Jain's fairness index [25], which is defined as follows.

$$\text{Fairness:} \quad F = (\sum_{i=0}^{n-1} |x_i|)^2 / [n \times \sum_{i=0}^{n-1} |x_i|^2]$$

Note that the value of the 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. This metric is also commonly used for fairness measures in many applications [24, 33, 43].

**Graph algorithms.**

We consider seven widely studied algorithms, personalized PageRank (PPR) [14], random walk with jump (RWJ) [23], random walk with domination (RWD) [34], Deepwalk [40] and node2vec [19], PageRank (PR)[39], and Connected Components (CC)[21]. The first five of them are random walk algorithms, and we start $|V|$ walks for all algorithms, by using the same setting as in KnightKing. For each walk, RWJ, RWD, Deepwalk and node2vec are terminated with a fix number of steps. In each step of walk, RWJ jump to a random vertex with probability 0.2, and PPR terminate with probability 0.1. The last two of them are iteration based algorithms, and we run them on Gemini. We run PR for ten iterations and CC until convergence.



(a) Balance of vertices (b) Balance of edges

**Figure 11: Balanced degree (Jain's fairness) when partitioning into different number of subgraphs.**

## 4.2 Balanced Degree of the Partition

We compare the balanced degree of the partition results of Chunk-V, Chunk-E, Fennel and BPart on three real-world graphs. We first show the bias metric for the number of vertices and the number of edges of the partitioned subgraphs, and the results are shown in Figure 10. The x-axis and y-axis denote the bias of the number of vertices and the bias of the number of edges, respectively. Each point in the figure denotes the result of a partition algorithm which partitions the large graph into a certain number of subgraphs (e.g., 4, 8, or 16). We can see that existing partition algorithms achieve balance only in one dimension. Specifically, Chunk-E can realize a balanced edge partition, but the bias of the number of vertices can reach up to 9.06. When the number of subgraphs gets larger, the bias also gets larger. On the contrary, Chunk-V and Fennel can realize a balanced vertex partition, but the numbers of edges among subgraphs are highly imbalanced, and the bias of the number of edges can reach up to 9.15. However, BPart can always realize two-dimensional balanced partition. Besides, when the number of partitioned subgraphs grows, the bias grows fast for Chunk-V, Chunk-E, and Fennel, but for BPart, the bias always keeps to be small, e.g., it is always smaller than 0.1.

We also compare the balanced degree with offline partition algorithms [1, 3, 28, 47]. These algorithms usually need to load the whole graph data into memory to traverse multiple times, so they are often time-consuming. The state-of-the-art algorithm is Mt-KaHIP [3], which uses a multi-level approach. In detail, it first coarsens the original graph into a smaller graph, and this is done by treating multiple vertices as one vertex by using the label propagation method, then it partitions the coarsened graph, and finally, it adopts uncoarsening/local search to recovery from these coarsened subgraphs. We consider three real-word graphs mentioned above, and use Mt-KaHIP to partition each graph into 8 subgraphs. We show the bias metrics for the number of vertices and the bias metrics for the number of edges. The bias metrics of the number of vertices are 0.03 for all the three datasets, and the bias metrics of the number of edges are 2.5853, 2.5622, and 0.7046 for LiveJournal, Twitter, and Friendster, respectively. The results imply that the number of edges is quite imbalanced. However, BPart can always achieve a balanced partition in two dimensions, and the bias can be always smaller than 0.1 in both dimensions.

We also show the balanced degree of the partitioned subgraphs by using the Jain's fairness index as the balanced degree metric. To also study the impact of partitioning into a large number of subgraphs, we consider the case of partitioning into 8, 16, 32, 64, and 128 subgraphs. In the interest of space, we show only the result under Twitter graph, and we observe similar results for other graphs. As shown in Figure 11, we can see that the fairness index when using our proposed BPart is always very close to one, and this implies that BPart can always achieve balance in both dimensions. Besides, the balanced degree keeps stable when varying the number of partitioned subgraphs, so BPart can be used to realize two-dimensional balanced partition for distributed graph systems running on tens to hundreds of machines.

**Partition Overhead.** Before studying the impact of BPart on system performance, we first show its overhead. Specifically, we count the time cost of partitioning the three real-world graphs into 8 subgraphs, respectively, and show the results in Table 2. We can see that chunk-V, Chunk-E cost much less time to finish the partition process compared with Fennel and BPart. Hash also costs less time than Fennel and BPart, but costs more time than Chunk-V and Chunk-E. The reason is that for each vertex, computing the score required by Fennel and BPart is more time-consuming than generating a random number by hash, while computing hash is more time-consuming than simply adding the number of nodes or neighbors in Chunk-V and Chunk-E. Fortunately, the partition is usually executed in preprocess, and it only needs to execute once for all graph analytic tasks. Therefore, costing hundreds of seconds to partition the graph is still acceptable. Besides, we also observe that our proposed partition algorithms cost relatively more time, compared with the corresponding baselines. This is because our algorithms may need multiple rounds of combination to realize a two-dimensional balanced partition, which brings a higher partition overhead. We believe that this overhead is acceptable, as we can make the computation more balanced, and thus save a lot of application running time and improve the efficiency of distributed graph processing.

|  | 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 | 17.1727 | 89.6942 | 210.3751 |

**Table 2: Time overhead (s) of partition algorithms.**

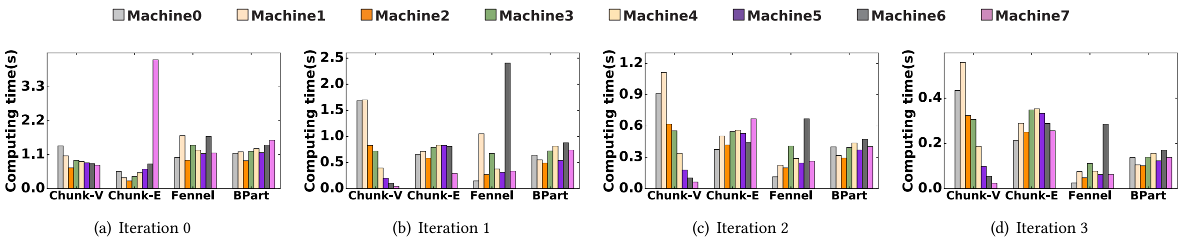
## 4.3 Balance of Computing Loads

We now evaluate the balanced degree of the computing loads among the cluster of machines, so as to demonstrate the impact of balanced graph partition on the performance of distributed graph processing systems. We adopt KnightKing as code base and integrate various partition algorithms into it for experiments. Note that we do not modify the computation process of KnightKing. As KnightKing is optimized for random walks, we start $5|V|$ simple random walks simultaneously and let each walk run four steps, that is, the system runs four iterations in total. Figure 13 shows the ratio of waiting time during the whole execution of these random walks, which is defined as the total waiting time of all machines divided by the total running time of completing all these random walks. We can see that with the partition algorithms, Fennel, Chunk-V and Chunk-E, there is up to 70% of time wasted on waiting for the slowest machine to finish computation due to the imbalanced distribution for either the number of vertices or the number of edges. On average, the ratios of waiting time are 45% and 55% for the cases of having 4 machines and 8 machines, respectively. On the other hand, due to the balanced partition in both two dimensions, BPart costs much less time on waiting, e.g., the ratios are only 10% and 20% for the cases of using 4 machines and 8 machines, respectively.

To further illustrate why unbalanced partition brings a lot of waiting time, we also show the distribution of the computation time of each machine in every iteration. The results are shown in Figure 12, and each sub-figure shows the results in one iteration. Here we only show the results for Friendster on 8 machines, and we observe similar results for other datasets and cluster scales. Note that the y-axis denotes the computation time of each machine. From the results, we can observe a highly imbalanced results for the computation time among different machines in almost all iterations, for Fennel, Chunk-V and Chunk-E. That is, it wastes a long time for certain machines to wait for the slowest machine to complete its computation in each iteration. While for BPart, the computation time distribution among machines is much more balanced in all iterations, this benefits from the two-dimensional balanced partition. As a result, BPart can save a lot of waiting time and improve the efficiency of distributed graph processing.

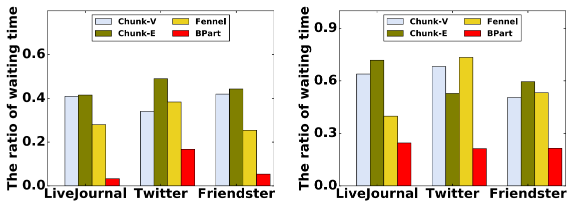
## 4.4 Running Time of Graph Applications

We now evaluate the total running time of various graph applications when using different graph partitioning schemes. Note that we consider seven different graph applications (see §4.1). The results are shown in Figure 14. The x-axis represents different graph applications, and the y-axis represents the normalized computation time. We normalize the time when using Chunk-V as one. The results show that BPart always outperforms other partition algorithms in all situations. Specifically, BPart 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



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

**Figure 12: The computing time on each machine in different iterations: unbalanced partition leads to unbalanced computation.**



(a) Four machines (b) Eight machines

**Figure 13: 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.**

number of edges in each subgraph, BPart balances the computing loads, so it has better performance for distributed graph processing.

### 4.5 Comparison with Hash

Recall that hash based algorithm can achieve balanced partition in two dimensions, but it results in lots of edge cuts. Thus, we also compare with hash-based partition algorithm. We first show the ratio of the number of edge cuts, i.e., the number of edges between different subgraphs divided by the total number of edges of the graph. The results are shown in Table 3. We can see that Hash and Chunk-E have more edge cuts compared with other algorithms, for example, Hash has around 87% edge cuts for all datasets, but Fennel and BPart have around only 35% and 55% edge cuts, respectively. Note that as BPart partitions the graph into smaller pieces in the partition phase, so it has more edge cuts compared with Fennel, while Fennel achieves balance in only one dimension.

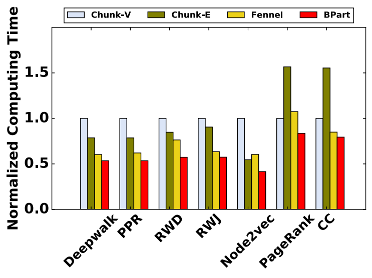
| | LiveJournal | Twitter | Friendster |
|---|---|---|---|
| Chunk-V | 0.5758 | 0.7475 | 0.6592 |
| Chunk-E | 0.9033 | 0.9026 | 0.7645 |
| Fennel | 0.6491 | 0.3338 | 0.3565 |
| Hash | 0.8750 | 0.8749 | 0.8750 |
| BPart | 0.7331 | 0.6226 | 0.5301 |

**Table 3: The ratio of the number of edge cuts (i.e., the edges between partitioned subgraphs) to the total number of edges.**

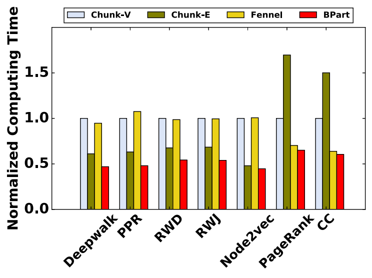
We then show the computation time of the seven graph applications when using Hash and BPart for partition. Other settings are the same as those in §4.4. Figure 15 shows the results, and we normalize the time when using Hash as one. The x-axis denotes different graph applications, and the y-axis denotes the normalized computation time. From the results, we can see that even Hash achieves two-dimensional balanced partition, BPart still outperforms it, e.g., for random walk based algorithms, BPart can decrease 5% to 20% of the total computation time, while for other iteration based algorithms, BPart can reduce the computation time by 20% to 35%. The reduction of the computation time mainly comes from the decrease of the number of edge cuts, because fewer edge cuts incur smaller communication cost during the computation process.

## 5 RELATED WORK

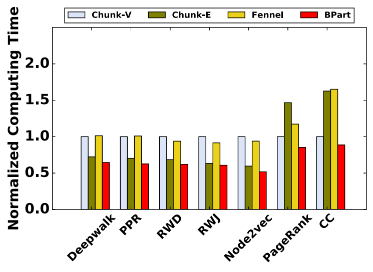
**Graph processing systems.** Many distributed graph processing systems, which leverage a cluster of machines to handle very large graphs, have been proposed in recent years [7, 8, 17, 18, 30, 37, 49]. Pregel [37] first proposed the vertex-centric BSP computation model to support the distributed graph processing on multiple machines, and a lot of succedent works followed this computation model and developed the prototype systems for distributed graph processing. Many design efforts are proposed to improve the system efficiency, such as designing differentiated graph partitioning and computation model according to the scale-free nature of real-world graphs like PowerGraph [17], PowerLyra [8], and Gemini [58], and improving the efficiency of graph mining and graph query [30]. Besides, KnightKing [57] optimized the performance of random walk processing. LiveGraph [59] optimized the graph storage for transactional graph processing situations. GraphScope [11] exposed a unified programming interface with variety graph computing. And Mycelium [44] provided distributed queries with private protection, etc. However, the above works do not pay much attention to the imbalance of computing loads between cluster machines due to imbalanced graph partition. Different from them, BPart targets for realize a balanced computing loads distribution via two-dimensional balanced graph partition, so as to reduce the synchronization overhead and improve the distributed graph processing efficiency. In addition to distributed graph systems, a number of out-of-core single machine graph processing systems are also proposed to handle large graphs [2, 32, 35, 45, 52, 60]. They store the graph data on external storage devices, like SSDs, and iteratively load a subgraph into memory and conduct the computation related to that subgraph.
**Graph partition strategies.** In recent years, various design efforts are made to improve graph partition efficiency and accelerate graph



(a) LiveJournal

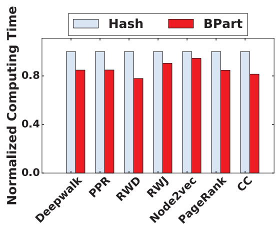


(b) Twitter

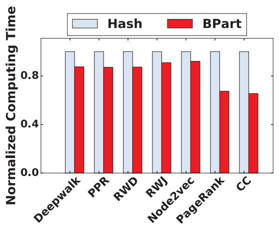


(c) Friendster

**Figure 14: Normalized running time of different graph application algorithms.**



(a) Twitter



(b) Friendster

**Figure 15: The normalized computation time of various graph applications when using Hash and BPart for graph partition.**

processing. These partition strategies can be classified into two categories, vertex-cut partition algorithms and edge-cut partition algorithms. Vertex-cut partition algorithms [8, 17, 20, 41, 56] split the edge set into multiple disjoint partitions, and cut the vertices that have edge connections with more than one subgraph. Generally, each of the subgraphs will store a duplicate of these vertices' information to enable the computing, thus introducing many redundant data. Edge-cut partition algorithms [3, 6, 9, 10, 12, 13, 16, 22, 27–29, 38, 47, 48, 50, 53, 54] 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. Generally, 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, the stream-based partition, which takes the vertices or edges as a stream and sequentially assign the the same number of vertices or edges as a partition, is widely used [35, 60]. However, these algorithms can only realize a balanced partition in one dimension, i.e., either the number of vertices or the number of edges. By randomly assigning each vertex to a subgraph, the hash-based partition can achieve a balanced partition in two dimensions, but it faces lots of edge cuts. Besides, GD [4] uses a gradient descent method to split a graph into two subgraphs, and it can also achieve balanced partition in two dimensions, but it is very time-consuming and only partition a graph into power of two subgraphs. Different from them, BPart targets for partitioning the graph into any number of subgraphs, while aiming for achieving a two-dimensional balanced graph partition with a small number of edge cuts between subgraphs.

## 6 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. By integrating BPart into distributed graph systems, it makes the computing loads between machines be well balanced,

## 7 ACKNOWLEDGMENTS

This work is supported in part by the National Key R&D Program of China (2018YFB1800203), NSFC (62172382), Youth Innovation Promotion Association CAS (No. 2019445) and Huawei. Yongkun Li is the corresponding author.

## REFERENCES

[1] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. *International Conference on Very Large Data Bases* 11, 11 (2018), 1590–1603.

[2] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk i/o. In *USENIX Annul Technical Conference.*

[3] Yaroslav Akhremtsev, Peter Sanders, and Christian Schulz. 2020. High-quality shared-memory graph partitioning. *IEEE Transactions on Parallel and Distributed Systems* 31, 11 (2020), 2710–2722.

[4] Dmitrii Avdiukhin, Sergey Pupyrev, and Grigory Yaroslavtsev. 2019. Multi-Dimensional Balanced Graph Partitioning via Projected Gradient Descent. *International Conference on Very Large Data Bases* 12, 8, 906–919.

[5] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11, 3 (2011), 5–9.

[6] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data minin.* 1456–1465.

[7] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-miner: an efficient task-oriented graph mining system. In *European Conference on Computer Systems.* 1–12.

[8] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)* 5, 3 (2019), 1–39.

[9] Cédric Chevalier and François Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34, 6-8 (2008), 318–331.

[10] William E Donath and Alan J Hoffman. 2003. Lower bounds for the partitioning of graphs. In *Selected Papers Of Alan J Hoffman: With Commentary*. World Scientific, 437–442.

[11] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *International Conference on Very Large Data Bases* 14, 12 (2021), 2879–2892.

[12] Wenfei Fan, Muyang Liu, Ping Lu, and Qiang Yin. 2021. Graph Algorithms with Partition Transparency. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[13] Charles M Fiduccia and Robert M Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *19th design automation conference*. IEEE, 175–181.

[14] Dániel Fogaras, Balázs Rácz, Károly Csalogány, and Tamás Sarlós. 2005. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics* 2, 3 (2005), 333–358.

[15] Friendster. 2013. . http://konect.cc/networks/friendster/

[16] Shufeng Gong, Yanfeng Zhang, and Ge Yu. 2020. HBP: Hotness balanced partition for prioritized iterative graph computations. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1942–1945.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX Symposium on Operating Systems Design and Implementations.*

[18] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. Graphx: Graph processing in a distributed dataflow framework. In *USENIX Symposium on Operating Systems Design and Implementations.*

[19] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable feature learning for networks. In *ACM Knowledge Discovery and Data Mining.*

[20] Masatoshi Hanai, Toyotaro Suzumura, Wen Jun Tan, Elvis S. Liu, Georgios Theodoropoulos, and Wentong Cai. 2019. Distributed Edge Partitioning for Trillion-edge Graphs. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2379–2392.

[21] Lifeng He, Yuyan Chao, Kenji Suzuki, and Kesheng Wu. 2009. Fast connected-component labeling. *Pattern recognition* 42, 9 (2009), 1977–1987.

[22] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. 2010. Engineering a scalable high quality graph partitioner. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–12.

[23] Rana Hussein, Dingqi Yang, and Philippe Cudré-Mauroux. 2018. Are meta-paths necessary? Revisiting heterogeneous graph embeddings. In *ACM International Conference on Information and Knowledge Management.* 437–446.

[24] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *ACM Symposium on Operating Systems Principles.*

[25] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* (1984).

[26] Glen Jeh and Jennifer Widom. 2002. Simrank: a measure of structural-context similarity. In *ACM Knowledge Discovery and Data Mining.*

[27] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).

[28] George Karypis and Vipin Kumar. 1999. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review* 41, 2 (1999), 278–300.

[29] Brian W Kernighan and Shen Lin. 1970. An efficient heuristic procedure for partitioning graphs. *The Bell system technical journal* 49, 2 (1970), 291–307.

[30] Arijit Khan, Gustavo Segovia, and Donald Kossmann. 2018. On smart query routing: for distributed graph querying with decoupled storage. In *USENIX Annul Technical Conference.*

[31] Aapo Kyrola. 2013. Drunkardmob: billions of random walks on just a pc. In *ACM conference on recommender systems.*

[32] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale graph computation on just a PC. In *USENIX Symposium on Operating Systems Design and Implementations.*

[33] Kai Li and Paul Hudak. 1989. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* 7, 4 (1989), 321–359.

[34] Rong-Hua Li, Jeffrey Xu Yu, Xin Huang, and Hong Cheng. 2014. Random-walk domination in large graphs. In *IEEE International Conference on Data Engineering*. IEEE.

[35] Hang Liu and H Howie Huang. 2017. Graphene: Fine-grained {IO} management for graph computing. In *Conference on File and Storage Technologies.*

[36] LiveJournal. 2006. . http://konect.cc/networks/livejournal-groupmemberships/

[37] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *ACM Conference on Management of Data.*

[38] Claudio Martella, Dionysios Logothetis, Andreas Loukas, and Georgos Siganos. 2017. Spinner: Scalable graph partitioning in the cloud. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. Ieee, 1083–1094.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank citation ranking: Bringing order to the web.* Technical Report. Stanford InfoLab.

[40] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. Deepwalk: Online learning of social representations. In *ACM Knowledge Discovery and Data Mining.*

[41] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. Hdrf: Stream-based partitioning for power-law graphs. In *ACM International Conference on Information and Knowledge Management.*

[42] Natasa Pržulj, Derek G Corneil, and Igor Jurisica. 2004. Modeling interactome: scale-free or geometric? *Bioinformatics* 20, 18 (2004), 3508–3515.

[43] Injong Rhee, Ajit Warrier, Mahesh Aia, Jeongki Min, and Mihail L Sichitiu. 2008. Z-MAC: a hybrid MAC for wireless sensor networks. *IEEE/ACM Transactions on Networking* 16, 3 (2008), 511–524.

[44] Edo Roth, Karan Newatia, Yiping Ma, Ke Zhong, Sebastian Angel, and Andreas Haeberlen. 2021. Mycelium: Large-Scale Distributed Graph Queries with Differential Privacy. In *ACM Symposium on Operating Systems Principles.*

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

[46] Semih Salihoglu and Jennifer Widom. 2013. Gps: A graph processing system. In *Proceedings of the 25th international conference on scientific and statistical database management.* 1–12.

[47] Peter Sanders and Christian Schulz. 2011. Engineering multilevel graph partitioning algorithms. In *European Symposium on Algorithms*. Springer, 469–480.

[48] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *ACM Knowledge Discovery and Data Mining.*

[49] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. 2013. From "think like a vertex" to "think like a graph". *International Conference on Very Large Data Bases* 7, 3 (2013), 193–204.

[50] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *ACM International Conference on Web Search and Data Mining.*

[51] Twitter. 2010. . https://law.di.unimi.it/webdata/twitter-2010/

[52] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the edges you need: A generic I/O optimization for disk-based graph processing. In *USENIX Annul Technical Conference.*

[53] Chris Walshaw and Mark Cross. 2007. JOSTLE: parallel multilevel graph-partitioning software–an overview. *Mesh partitioning techniques and domain decomposition techniques* 10 (2007), 27–58.

[54] Lu Wang, Yanghua Xiao, Bin Shao, and Haixun Wang. 2014. How to partition a billion-node graph. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 568–579.

[55] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. Graph-walker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *USENIX Annul Technical Conference.*

[56] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed Power-law Graph Computing: Theoretical and Empirical Analysis.. In *Annual Conference on Neural Information Processing Systems*, Vol. 27. 1673–1681.

[57] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. 2019. Knightking: a fast distributed graph random walk engine. In *ACM Symposium on Operating Systems Principles.*

[58] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A computation-centric distributed graph processing system. In *USENIX Symposium on Operating Systems Design and Implementations.*

[59] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2019. Livegraph: A transactional graph storage system with purely sequential adjacency list scans. *arXiv preprint arXiv:1910.05773* (2019).

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annul Technical Conference.*