Accelerating Large-Scale Prioritized Graph Computations by Hotness Balanced Partition

Shufeng Gong, Yanfeng Zhang, and Ge Yu, *Senior Member, IEEE*

Abstract—Prioritized computation is shown promising performance for a large class of graph algorithms. It prioritizes the execution of some vertices that play important roles in determining convergence. For large-scale distributed graph processing, graph partitioning is an important preprocessing step that aims to balance workload and to reduce communication costs between workers. However, existing graph partitioning methods are designed for round-robin synchronous distributed frameworks. They balance workload without distinction of vertex importance and fail to consider the characteristics of priority-based scheduling, which may limit the benefit of prioritized graph computation. In this article, to accelerate prioritized iterative graph computations, we propose Hotness Balanced Partition (HBP). In prioritized graph computation, high priority vertices are likely to be executed more frequently and are likely to pass more messages, which result in hot vertices. Based on this observation, we partition graph by distributing vertices with distinction according to their hotness rather than blindly distributing vertices with equal weights, aiming to evenly distribute the hot vertices among workers. We further provide two HBP algorithms: a streaming-based algorithm for efficient one-pass processing and a distributed algorithm for distributed processing. Our results show that our proposed partitioning methods outperform the state-of-the-art partitioning methods, Fennel, HotGraph, and SNE.

Index Terms—Hotness balanced partition, graph partitioning, distributed computing

1 Introduction

To handle massive graphs, distributed graph processing systems [1], [2], [3], [4] partition the graph data into multiple graph partitions and process them on a cluster of workers, with each worker working on a graph partition. During the distributed graph computation process, 1) heavy communication cost between workers due to a large number of edge/vertex cuts and 2) idle workers due to unbalanced workload may exist, which degrade the performance of distributed computing. In order to reduce the communication cost and idle workers, many research efforts [5], [6], [7], [8], [9], [10], [11] have been put on finding smart graph partitioning methods as a preprocessing step for large-scale distributed graph processing, aiming at minimizing connections between graph partitions and making the workload evenly distributed among partitions.

Prior graph partitioning algorithms all use the assumption that underlying distributed graph mining frameworks adopt a synchronous parallel processing model. In synchronous parallel model, there is a global synchronous barrier after each iteration (super step). On each worker, the assigned vertices are processed in a round-robin manner. In round-robin scheduling, the vertices are processed in circular order. All the vertex computations are invoked in each round without discrimination. Round-robin is widely used since it is simple, easy to implement, and starvation-free. In each iteration, each vertex is only processed once, and each edge only delivers one message. Therefore, the number of vertices (for vertex-centric frameworks) or edges (for edge-centric frameworks) of a graph partition indicates the workload of a worker, and the number of cutting edges (or vertex replicas in vertex-cut systems) indicates the communication cost between workers. If the workload of workers is imbalanced, the light loaded worker will wait for the heavy loaded worker. Furthermore, a large number of cutting edges (or vertex replicas) will result in significant network overhead. For this reason, existing graph partitioning algorithms aim to i) balance the vertices (or edges) of each partition and ii) minimize the number of cutting edges (or vertex replicas).

Recently, a number of research works pay attention to asynchronous parallel processing, such as GraphLab [2], GraphUC [12], Giraph++ [13], GRAPE+ [14] and Maiter [15]. In these asynchronous distributed frameworks, the global synchronous barriers are removed. Thus there is no waiting time between workers, and the vertices/edges can be processed at any time. In other words, there is no such constraint that each vertex/edge can only be processed once in each iteration, and some vertices/edges can be processed more times than the others. Recent studies [2], [16] show that some of the vertices do play important decisive roles in determining the final converged outcome. In asynchronous computation, the importance of vertices is not consistent. In priority processing, only a subset of vertices (i.e., the important vertices) are selected to be processed in each round, so that these important vertices are processed more frequently than others. Compared with round-robin processing, priority processing tends to filter the inefficient and invalid computations. As shown in Fig. 1, the prioritized graph processing

• *The authors are with the School of Computer Science and Engineering, Northeastern University, Shenyang 110819, China. E-mail: shidashufeng@163.com, {zhangyf, yuge}@mail.neu.edu.cn.*

Manuscript received 27 Feb. 2020; revised 8 Oct. 2020; accepted 9 Oct. 2020. Date of publication 21 Oct. 2020; date of current version 10 Nov. 2020. (Corresponding author: Ge Yu.) Recommended for acceptance by A. Sussman. Digital Object Identifier no. 10.1109/TPDS.2020.3032709

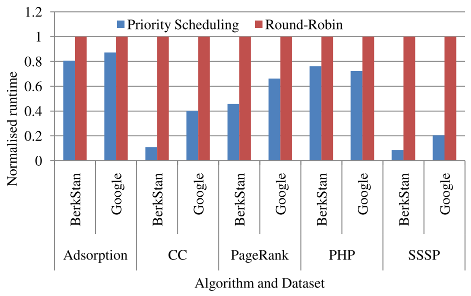
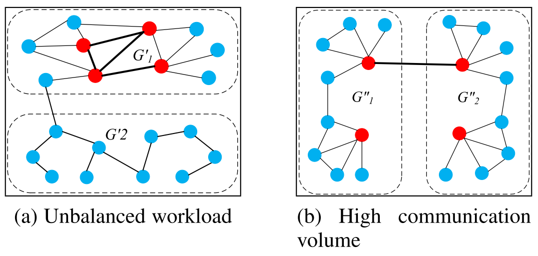


Fig. 1. Priority scheduling versus round-robin scheduling for asynchronous parallel processing.



(a) Unbalanced workload (b) High communication volume

Fig. 3. Example partitions. High-priority vertices are with red color. The width of solid line implies the amount of communication.

(with priority scheduling) is 1.2-10x faster than the round-robin scheduling for various graph algorithms.

Example 1. Prioritized PageRank [16] is a representative graph algorithm with priority scheduling. In each iteration, rather than updating the complete set of node ranking scores, prioritized PageRank only selects a small subset of vertices (e.g., top 1 percent important vertices) for computation, so that the computation is more effective and tends to converge faster. However, Without consideration of the characteristics of priority scheduling, the traditional graph partitioning methods may limit the performance of prioritized PageRank computation. As shown in Fig. 2a, Fennel [9] results in longer runtime than naive Hash partition, even if Fennel has less communication. This is because that Fennel tries to balance the number of vertices without considering the execution priority of vertices. As shown in Fig. 3a, all the high-priority vertices (with red color) that require more computation resources are assigned to one partition, so that the actual workload is not really balanced though the number of vertices is balanced, which results in stragglers and slows down the convergence process. In addition, as shown in Fig. 2b, although Fennel provides smaller number of edge cuts than Hash, Fennel still results in more communication. Fennel aims to partition the graph to minimize the number of edge cuts. However, as shown in Fig. 3b, though the number of edge cuts is minimized, the cuts of the edges that connect high-priority vertices will lead to heavy communication traffic between workers and longer runtime.

Motivation. Although many efforts have been devoted to graph partitioning, most of them assume the round-robin synchronous graph processing (abbr. round-robin processing). The prior works are far from ideal for prioritized asynchronous processing (abbr. prioritized processing). To design graph partitioning method for prioritized processing, we need to pay attention to the following two key differences between prioritized processing and round-robin processing.

First, the workload balance does not necessarily imply high efficiency. With the elimination of synchronous barriers, there is no waiting time between different machines and no idle workers. All the workers are busy processing all the time. However, some vertices play important roles in helping the convergence of iterative computation [15], [16]. These important vertices are worth to be processed with more updates than the others. The graph partitioning methods should take this property into account for computation effectiveness.

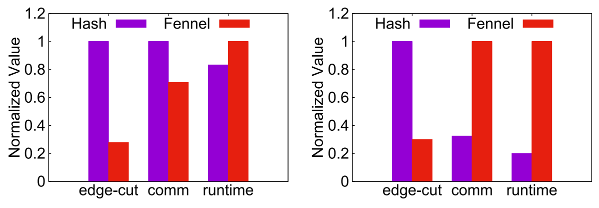
Second, a small number of edge cuts or vertex replicas do not necessarily imply less communication cost. Due to priority scheduling, the number of updates on each vertex is not consistent, and the number of messages passed through each edge is not consistent. Even with few edge cuts/vertex replicas, it may still result in a large amount of communication cost since the number of messages delivered along each edge is not identical. If an edge of a high priority vertex is cut, there still can be a large number of messages delivered along this edge.

In summary, prioritized graph computation leads to the distinction of vertex hotness. The states of hot vertices are updated more frequently, and more messages are propagated from these hot vertices. The existing graph partitioning methods designed for round-robin processing are not suitable for prioritized processing. Thus, a new partitioning method designed for prioritized graph processing is desired.

Contribution. In this paper, we propose the idea of *Hotness Balanced Partition (HBP)*, which partitions graph according to vertex's hotness. We first estimate each vertex's hotness and then propose three objectives of graph partitioning for prioritized processing. To efficiently solve the multi-objective optimization problem, we propose a heuristic streaming-based algorithm, *SPb-HBP*, to partition the graph, which only requires one pass of the graph data. We further propose a distributed version of SPb-HBP, *DSPb-HBP*, to partition massive graph data.

We perform experiments by comparing hash partition and two state-of-the-art graph partitioning schemes, Fennel [9] and HotGraph [17]. Our results show that our SPb-HBP is much more effective for prioritized iterative graph processing. Especially, SPb-HBP can reduce 40-50 percent runtime of that by hash partition, 5-75 percent runtime of that by Fennel, and 22-31 percent runtime of that by HotGraph.

The rest of this paper is organized as follows. In Section 2, we introduce a typical distributed prioritized iterative computation model, DAIC. We formalize hotness balanced



(a) 8 partitions (b) 12 partitions

Fig. 2. Hash partition versus Fennel partition for prioritized PageRank on LiveJournal. Three metrics are shown, the number of cut edges, the communication cost, and the total runtime.

TABLE 1
Notations

G, G_i	the graph and its ith partition
V, V_i (resp. E, E_i)	the set of vertices (resp. edges) in G and G_i
$IN(v)$ (resp. $OUT(v)$)	the incoming (resp. outgoing) neighbors of v
γ_v, Γ_G	the value of v and its set in G
$m_{u \to v}$	the message from u to v
Δ_v	the change of γ_v during iterations
h_v	the hotness value of v
$com_{u,v}$	the communication cost between u and v
$\mathcal{H}_j$	the set of vertices whose h_v are in jth hotness bin
$\mathcal{H}_{ji}$	the set of vertices in $V_i \cap \mathcal{H}_j$
P, P_i	the probability distribution of hotness histogram in G and G_i

partition problem in Section 3. In Section 4, we present the streaming-based partitioning algorithm SPb-HBP. In Section 5, we further propose a distributed partitioning method DSPb-HBP. Experimental results are shown in Section 6. The related work is presented in Section 7. We conclude our paper in Section 8.

2 Preliminaries

A number of existing frameworks have been proposed to support asynchronous parallel execution. Among these frameworks, Maiter [15] and its variants [18], [19], [20] employ the *asynchronous Delta-based Accumulative Iterative Computation (asynchronous DAIC)* model to guarantee correctness, which is a fully asynchronous model since neither distributed lock nor global synchronous barrier exists. Furthermore, DAIC supports priority scheduling. In this paper, we will analyze prioritized graph computation based on the asynchronous DAIC model. The notations of this paper are summarized in Table 1.

2.1 Asynchronous DAIC and Prioritized Execution

Let $G(V, E)$ denote a graph, where $(u, v) \in E$ is an edge from vertex u to v, $IN(v) = \{u \,|\, (u, v) \in E\}$ is the incoming neighbors set of vertex v, $OUT(v) = \{w \,|\, (v, w) \in E\}$ is the outgoing neighbors set of vertex v, $|V| = n$ is the number of vertices, and $|E| = m$ is the number of edge. Let $\Gamma_G = \{\gamma_v \,|\, v \in V\}$ denote the vertex states set of all vertices V, e.g., γ_v is the ranking score value of v in the PageRank algorithm. The traditional synchronous iterative graph algorithms can be expressed as an iterative update process.

$$\Gamma_G^k = f(\Gamma_G^{k-1}), \tag{1}$$

where f is the update function of graph mining algorithm, and k is the iteration times.

While in asynchronous DAIC model [15], the vertex-based update can be executed on any vertex at any time. The update of vertex v at time point t can be formalized as follows.

$$\begin{aligned} receive:\ & \Delta_v^t = \mathcal{A}(\Delta_v^{t-1}, m_{u\to v}(\Delta_u^{t-1})), u \in IN(v) \\ update:\ & \gamma_v^t = \mathcal{A}(\gamma_v^{t-1}, \Delta_v^t) \\ send:\ & m_{u\to w}(\Delta_v^t), w \in OUT(v), \end{aligned} \tag{2}$$

where Δ_v^t denotes the "change" from γ_v^{t-1} to γ_v^t, $\mathcal{A}$ is an aggregation function that accumulates the received messages from neighbors and is used to update γ_v, and $m_{u\to v}(\Delta_u^t)$ is the message from u to v that is generated based on Δ_u^t.

By using DAIC model, the computation of any vertex can be performed at any time point. In other words, the vertex updates can be scheduled in any order. The scheduling order is crucial to computation effectiveness. As shown in Fig. 1, the priority scheduling that selects important vertices for frequent processing helps algorithms converge faster than the round-robin scheduling for various graph algorithms. However, it is difficult to find the optimal scheduling order. Recent studies [16], [18] found that some of the vertices play an important decisive role in determining the final converged outcome. In [15], the authors pick the vertex that can maximize the "change" of graph state as the scheduling candidate, i.e., $v = argmax_v|\mathcal{A}(\gamma_v^{t-1}, \Delta_v^t) - \gamma_v^{t-1}|$. Since γ_v is monotonically increasing/decreasing, a great "change" implies a big move to the fixed point that makes the current state closer to the final state. Therefore, the execution priority of each vertex is set as the amount of change. Performing computations on the high priority vertices will accelerate the convergence, which is referred to as *prioritized execution*. In this paper, we focus on optimizing the graph partitioning for prioritized execution.

2.2 Prioritized DAIC Graph Algorithms

Not all graph algorithms can be converted into DAIC form. The sufficient conditions for equivalent conversion have been well studied in [15], [21]. There is a large class of graph mining algorithms satisfying these conditions that can be processed in DAIC framework. Next, we take PageRank [22], Adsorption [23], and Penalized Hitting Probability [24] algorithms as examples to briefly present how traditional iterative algorithms are converted into their DAIC forms.

PageRank. PageRank [22] is a popular algorithm initially proposed for ranking web pages. In DAIC version of PageRank [15], γ_v is the accumulated rank value of vertex v, $\mathcal{A}$ is the sum aggregation (i.e., "+"), and $m_{u\to v}(\Delta_u^t) = d \cdot \frac{\Delta_u^t}{|OUT(u)|}$ where d is a damping factor (e.g., 0.85). Then the update of DAIC PageRank can be rewritten as follows.

$$\begin{aligned} receive:\ & \Delta_v^t = \Delta_v^{t-1} + d \cdot \frac{\Delta_u^{t-1}}{|OUT(u)|}, u \in IN(v) \\ update:\ & \gamma_v^t = \gamma_v^{t-1} + \Delta_v^t \\ send:\ & d \cdot \frac{\Delta_v^t}{|OUT(v)|} \quad \text{to vertex } w, w \in OUT(v). \end{aligned}$$

High execution priority is given to the vertex that has the largest Δ_v since it can maximize the "change" of graph state.

Adsorption. Adsorption [23] is a graph label propagation algorithm that provides personalized recommendations for contents. In DAIC version of Adsorption, γ_v is the accumulated score of vertex v, $\mathcal{A}$ is the sum aggregation (i.e., "+"), and $m_{u\to v}(\Delta_u^t) = p_v^{cont} \cdot \omega_{u,v} \cdot \Delta_u^t$ where p_v^{cont} is a constant associated with each v and $\omega_{u,v}$ is the weight of edge (u, v). Then the update of DAIC Adsorption can be rewritten as follows.

$$\begin{aligned} receive:\ & \Delta_v^t = \Delta_v^{t-1} + p_v^{cont} \cdot \omega_{u,v} \cdot \Delta_u^{t-1}, u \in IN(v) \\ update:\ & \gamma_v^t = \gamma_v^{t-1} + \Delta_v^t \\ send:\ & p_w^{cont} \cdot \omega_{v,w} \cdot \Delta_v^t \quad \text{to vertex } w, w \in OUT(v). \end{aligned}$$

Similar to PageRank, high execution priority is given to the vertex that has the largest Δ_v.

Penalized Hitting Probability (PHP). PHP [24] is used to measure the proximity (similarity) between a given source vertex s and any other vertex v. In DAIC version of PHP [20], γ_v is the accumulated score of vertex v, $\mathcal{A}$ is the sum aggregation (i.e., "+"), and $m_{u\to v}(\Delta_u^t) = d \cdot \omega_{u,v} \cdot \Delta_u^t$ where d is a damping factor and $\omega_{u,v}$ is the weight of edge (u, v). The update of DAIC PHP can be rewritten as follows.

$$\begin{aligned} receive:\ & \Delta_v^t = \begin{cases} \Delta_v^{t-1} + d \cdot w_{u,v} \cdot \Delta_u^{t-1}, & (v \neq s) \\ 0, & (v = s) \end{cases} \\ update:\ & \gamma_v^t = \gamma_v^{t-1} + \Delta_v^t \\ send:\ & d \cdot w_{v,w} \cdot \Delta_v^{t-1} \quad \text{to vertex } w, w \in OUT(v). \end{aligned}$$

Then high execution priority is given to the vertex that has the largest Δ_v.

Besides PageRank, Adsorption, and PHP, there exist many other DAIC graph algorithms that can be performed with prioritized execution, such as SSSP, Connected Components, SimRank, Katz Metric and so on. Please refer to [15] for more details.

From the implementation's point of view, maintaining a priority queue is expensive due to the frequent update and sort operations. The existing prioritized graph computation systems [15], [16] employ an approximate approach. They achieve the approximate prioritization in a round-by-round manner. In each round, a subset of vertices with the highest priority values (e.g., top $n\%$) are selected for computation, where a sampling-based approach [16] is used to avoid the expensive global sort. The size of the prioritized subset balances the trade-off between the gain from accurate priority scheduling and the cost of frequent extractions for the prioritized subset.

3 Problem Formulation

As we have discussed in Section 1, the traditional partitioning methods fail to meet the requirements of priority scheduling frameworks. In priority scheduling frameworks, some vertices are given higher execution priority, so they become *hot vertices* in priority scheduling execution systems. The frequency of vertex updates is called vertex's *hotness*.

Given a graph with hotness $G = (V, E, H)$ and a partition number k where each vertex is associated with a hotness value h_v and $H = \{h_v, v \in V\}$ is the hotness values of all vertices, an edge-cut graph partitioning aims to find a partition scheme $\mathcal{G} = \{G_1, G_2, \ldots, G_k\}$, where $G_i = (V_i, E_i, H_i)$ is a partition of G. V_i is the set of vertices in G_i such that $V = \bigcup_{i=1}^{k} V_i$ and $V_i \bigcap V_j = \varnothing$, and E_i is the set of edges whose source vertices are in V_i, i.e., $E_i = \{(u,v) \mid u \in V_i\}$. Then, each partition is assigned to a worker for parallel processing. In this section, we first introduce how to estimate the hotness and then analyze the goals in hotness balanced partition based on the edge-cut partitioning model.

3.1 Hotness Estimation

The precondition of hotness balance partition is that we have obtained the hotness of vertices and the communication cost between two vertices. But these are unknown before the computation starts. Hence, we first present how to estimate vertex hotness and communication costs between vertices.

According to the priority scheduling introduced in Section 2.1, the priority value of vertices is determined by their Δ_v value, and Δ_v is collected from its in-neighbors $IN(v)$. Thus, if vertex v has a strong ability to collect Δ_v, vertex v is likely to be with a higher execution priority and is likely to be hot. We estimate the hotness of vertex v as follows.

$$h_v = \sum_{u \in IN(v)} \frac{w_{u,v}}{\sum_{w \in OUT(u)} w_{u,w}}, \tag{3}$$

where $w_{u,v}$ is the weight of edge (u, v). If there is no weight on edges, we assume $w_{u,v} = 1$.

During the computation, when a vertex is updated, it will send a message to its outgoing neighbors, so that the number of messages passed through an edge is proportional to its source vertex's update times. In other words, the edge communication cost can be estimated as its source vertex's hotness. Thus, the communication cost of edge (u, v) is estimated as follows.

$$com_{u,v} = h_u. \tag{4}$$

Note that, the estimation methods for vertex hotness and edge hotness are heuristic and can be customized according to algorithms' characteristics. It is also possible to perform dynamic graph partitioning or dynamic workload balancing during computation since we can obtain the exact vertex hotness during computation. But our preliminary results show unsatisfactory performance because dynamic load balancing could bring significant migration cost as the hotness values are not stable. In this paper, we focus on static hotness estimation and leave the dynamic graph partitioning as future work.

3.2 Partition Goals

In asynchronous frameworks with priority scheduling, the hotter the vertices are, the more computation resources they need. We should assign computation resources according to vertex hotness. Suppose a homogeneous cluster where workers have similar computation power, in order to globally improve work effectiveness, we should balance the hotness among partitions, so that the same amount of computation resources are assigned to each partition. Therefore, *our partitioning scheme aims to assign the same amount of vertex hotness to workers, which is the first goal*.

There is an assumption in the above analysis, which is that a global priority scheduler is used. However in practice, a global scheduler is very expensive in large scale clusters. We use a local scheduler that runs on each worker to simulate the global scheduler, i.e., the priority scheduling works in parallel and on a per-worker basis, so that the expensive global coordination overhead is avoided. As discussed in Section 2.2, under priority scheduling scheme, a subset of vertices (more than one vertices) could be scheduled in each

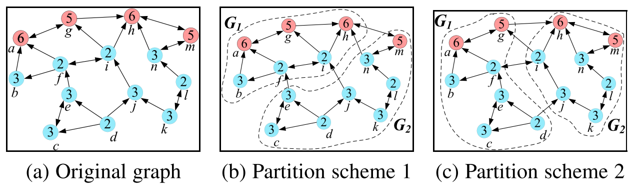

(a) Original graph (b) Partition scheme 1 (c) Partition scheme 2

Fig. 4. Hotness based partitioning examples. Red vertices are with higher hotness values, while blue vertices are with lower hotness values.

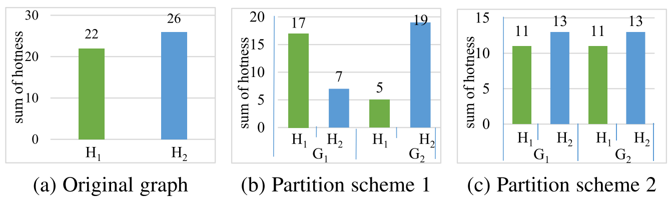

(a) Original graph (b) Partition scheme 1 (c) Partition scheme 2

Fig. 5. Histogram distribution of hotness values corresponding to Fig. 4.

round on a worker. For example in Fig. 4, let us suppose the hottest 2 vertices are selected for computation in each round. As shown in Fig. 4c, since only one hot vertex is assigned to worker 2 (with others being cool vertices), there could always be at least one cool vertex scheduled. With limited computation resources, this could make the computation on worker 2 less effective.

Given a hotness values set $\mathcal{H}$, we have its cumulative distribution function $F(x) = P(X \leq x)$. Under priority scheduling, the subset of vertices with high hotness values, $\mathcal{H}'$, are likely to be selected for prioritized execution. Let us assume its proportion to the whole set as $n\% = \frac{|\mathcal{H}'|}{|\mathcal{H}|}$. By distributed computing, we divide $\mathcal{H}$ into k disjoint partitions, $\mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_k$, where $\mathcal{H}_1 \cup \mathcal{H}_2 \cup \ldots \cup \mathcal{H}_k = \mathcal{H}$ and $\mathcal{H}_1 \cap \mathcal{H}_2 \cap \ldots \cap \mathcal{H}_k = \emptyset$. We select the top $n\%$ elements with the highest priority from each partition and obtain k candidate subsets $\mathcal{H}'_1, \mathcal{H}'_2, \ldots, \mathcal{H}'_k$ respectively. The following theorem shows that, if the local hotness distribution on each worker is consistent with the global hotness distribution, the local priority scheduling can achieve the same effect as the global priority scheduling.

Theorem 1. *If the local cumulative hotness distribution $F_j(x)$ on each worker j is consistent with the global cumulative hotness distribution $F(x)$, i.e., $\forall j, y$, $F_j(y) = F(y)$, we have $\mathcal{H}' = \mathcal{H}'_1 \cup \mathcal{H}'_2 \cup \ldots \cup \mathcal{H}'_k$.*

Proof. Because of the definition of $\mathcal{H}'$ and $\mathcal{H}'_j$, we have $\forall j, \frac{|\mathcal{H}'_j|}{|\mathcal{H}_j|} = \frac{|\mathcal{H}'|}{|\mathcal{H}|} = n\%$. Suppose the largest element in set $\mathcal{H} - \mathcal{H}'$ is m, i.e., $max\{\mathcal{H} - \mathcal{H}'\} \leq m \leq min\{\mathcal{H}'\}$, we have $F_j(m) = F(m) = P(X \leq m) = \frac{\mathcal{H}-\mathcal{H}'}{\mathcal{H}} = 1 - n\%$. That is, for any partition j, we have $1 - F_j(m) = \frac{|\mathcal{H}'_j|}{|\mathcal{H}_j|}$, i.e., $max\{\mathcal{H}_j - \mathcal{H}'_j | 1 \leq j \leq k\} \leq m \leq min\{\mathcal{H}'_j | 1 \leq j \leq k\}$. In other words, for each $\mathcal{H}_j$, all the elements that are larger than m must be in $\mathcal{H}'_j$, and $\mathcal{H}'_j$ does not contain any element that is smaller than m. So we have $\mathcal{H}' = \mathcal{H}'_1 \cup \mathcal{H}'_2 \cup \cdots \cup \mathcal{H}'_k$. □

We use hotness histogram to describe the hotness distribution. The vertex hotness values are divided into z continuous non-overlap intervals. Each interval corresponds to a bin $\mathcal{H}_j$ holding the vertices whose hotness values are in the jth interval, then we have $\max\{h_v \,|\, v \in \mathcal{H}_{j-1}\} < \min\{h_v \,|\, v \in \mathcal{H}_j\}$ and $\max\{h_v \,|\, v \in \mathcal{H}_j\} < \max\{h_v \,|\, v \in \mathcal{H}_{j+1}\}$. The height of the histogram bar is the sum of hotness values of vertices in each bin, i.e., $\sum_{v \in \mathcal{H}_j} h_v$. Then the probability distribution of hotness histogram in original G is defined as

$$P(\mathcal{H}_j) = \frac{\sum_{v \in \mathcal{H}_j} h_v}{\sum_{v \in V} h_v}. \tag{5}$$

Similarly, the probability distribution of hotness histogram in partition G_i is defined as

$$P_i(\mathcal{H}_j) = \frac{\sum_{v \in \mathcal{H}_{ji}} h_v}{\sum_{v \in V_i} h_v}, \tag{6}$$

where $\mathcal{H}_{ji} = \mathcal{H}_j \cap V_i$.

Jensen Shannon (JS) distance [25] is a metric that measures the similarity between two probability distributions. We use *Hotness Jensen-Shannon distance (HJS)* to quantify the variance between hotness distributions of partition G_i and graph G as follows.

$$\begin{aligned} HJS(P||P_i) = &\frac{1}{2}\left[\sum_{j=1}^{z} P(\mathcal{H}_j) log\left(\frac{P(\mathcal{H}_j)}{\frac{P(\mathcal{H}_j)+P_i(\mathcal{H}_j)}{2}}\right)\right. \\ &\left. + \sum_{j=1}^{z} P_i(\mathcal{H}_j) log\left(\frac{P_i(\mathcal{H}_j)}{\frac{P_i(\mathcal{H}_j)+P(\mathcal{H}_j)}{2}}\right)\right]. \end{aligned} \tag{7}$$

If $P_i(.) = P(.)$, i.e., the hotness distribution of G_i is consistent with the global hotness distribution, we have $log(\frac{P(\mathcal{H}_j)}{\frac{P(\mathcal{H}_j)+P_i(\mathcal{H}_j)}{2}}) = log(\frac{P_i(\mathcal{H}_j)}{\frac{P_i(\mathcal{H}_j)+P(\mathcal{H}_j)}{2}}) = 0$ and $HJS(P||P_i) = 0$. The smaller the HJS distance is, the more consistent the hotness distributions are.

Example 2. We divided the hotness values of vertices in G in Fig. 4a into two intervals, i.e., $z = 2$, and $\mathcal{H}_1 = \{v, 8 > h_v \geq 5\}$, $\mathcal{H}_2 = \{v, 5 > h_v \geq 1\}$. Fig. 5shows the corresponding hotness histogram distribution of G, and partition G_1 and G_2. For the partition scheme in Fig. 4b, $HJS(P||P_1) \approx 0.296$ and $HJS(P||P_2) \approx 0.296$. For the partition scheme in Fig. 4c, $HJS(P||P_1) = 0$, $HJS(P||P_2) = 0$.

It is noticeable that the number of bins z affects density estimation. A larger number of bins (wider bins) gives greater precision to the density estimation but may increase noise due to sampling randomness. There is no "best" number of bins. Our results in Section 6.5 show that the performance becomes stable when the number of bins is larger than 4. Therefore, with a fixed z, *our second goal is to minimize the HJS distance between each partition and the original graph.*

As known in distributed prioritized iterative computation, network communication has a great impact on the performance of distributed computing. Heavy network traffic may lead to message blocking when sending the important messages that help accelerate prioritized computation. Therefore, *our third goal is to minimize the communication cost between partitions.*

Based on the above discussion, our hotness balanced partition tailored for distributed prioritized graph computation is defined as follows.

Definition 1. Hotness Balanced Partition. *Given a graph $G = (V, E, H)$ and a partition number k, hotness balanced partition aims to find a partitioning scheme $\mathcal{G} = \{G_1, \ldots, G_k\}$ such that*

1) *the sum of hotness values of each partition is balanced, i.e.,* $\min \sum_{i=1}^{k} | \sum_{v\in V_i} h_v - \frac{\sum_{v\in V} h_v}{k}|$.
2) *the variance of hotness distributions between each partition and original graph is minimized, i.e.,* $\min \sum_{i=1}^{k} HJS(P||P_i)$.
3) *the communication is minimized, i.e.,* $\min \sum_{i=1}^{k} Com_i$, *where* $Com_i = \sum_{u\in V_i, v\notin V_i}(com_{u,v} + com_{v,u})$ *and* $com_{u,v}$ *is the amount of messages sent from u to v.*

4 SPb-HBP

As an NP-hard problem, it is difficult to find the optimal partitioning scheme that meets all the three requirements at the same time. As a multi-objective optimization problem, it is also very expensive to use some local search-based approaches to find a local optimal solution. In this section, we propose an efficient streaming-based heuristic algorithm, Streaming-based Per-Bin Hotness Balanced Partition (*SPb-HBP*).

4.1 Per-Bin Hotness Balanced Partition

According to Definition 1, there are three goals we want to achieve in HBP. We observe that if each bin is partitioned with balanced hotness, we can achieve both of the first and second goals at the same time. For example in Fig. 5c, the vertices in the first bin are approximately evenly partitioned with regard to hotness, and the vertices in the second bin are also approximately evenly partitioned. Then the sums of hotness values in G_1 and G_2 are identical. $HJS(P_1||P)$ and $HJS(P_2||P)$ are both very small. The following theorem states this fact.

Theorem 2. *For a given graph $G(V,E,H)$ with z hotness interval bins $\{\mathcal{H}_1,\ldots,\mathcal{H}_z\}$, if we partition each bin $\mathcal{H}$ into k hotness-balanced bin partitions $\{\mathcal{H}_{j1},\ldots,\mathcal{H}_{jk}\}$ and merge the bin partitions that have the same partition id into a graph partition $V_i = \bigcup_{j=1}^{z}\mathcal{H}_{ji}$, the hotness of each graph partition is balanced*

$$\sum_{v\in V_i} h_v = \frac{\sum_{v\in V} h_v}{k}, \tag{8}$$

and the HJS distance between G and G_i is minimized

$$HJS(P||P_i) = 0. \tag{9}$$

Proof. Since $\{\mathcal{H}_{j1},\ldots,\mathcal{H}_{jk}\}$ is the hotness balanced partition of $\mathcal{H}_j$ and $V_i = \bigcup_{j=1}^{z}\mathcal{H}_{ji}$, then we have

$$\sum_{v\in V_i} h_v = \sum_{j=1}^{z}\sum_{v\in\mathcal{H}_{ji}} h_v = \sum_{j=1}^{z}\frac{\sum_{v\in\mathcal{H}_j} h_v}{k} = \frac{\sum_{v\in V} h_v}{k}. \tag{10}$$

The probability distribution of hotness histogram in partition G_i is

$$P_i(\mathcal{H}_j) = \frac{\sum_{v\in\mathcal{H}_{ji}} h_v}{\sum_{v\in V_i} h_v} = \frac{\frac{1}{k}\sum_{v\in\mathcal{H}_j} h_v}{\frac{1}{k}\sum_{v\in V} h_v} = P(\mathcal{H}_j). \tag{11}$$

By applying $P_i(\mathcal{H}_j) = P(\mathcal{H}_j)$ in Equation (7), we have $HJS(P||P_i) = 0$. □

Based on the above discussion, the first two HBP goals in Definition 1 can be reduced to one

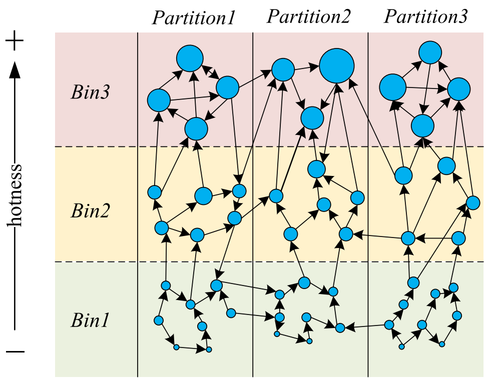


Fig. 6. Pb-HBP graph partitioning.

$$\min \sum_{i=1}^{k}\left| \sum_{v\in\mathcal{H}_{ji}} h_v - \frac{\sum_{v\in\mathcal{H}_j} h_v}{k}\right|. \tag{12}$$

Finally, our partition goals become i) *balancing the hotness of bin partitions and ii) minimizing the communication cost*. We call this variant of HBP as Per-Bin Hotness Balanced Partition (Pb-HBP). The idea of Pb-HBP is also depicted in Fig. 6. The vertices in various hotness bins (with various levels of hotness) are partitioned separately, at the same time the communication cost between partitions is minimized.

4.2 Streaming-Based Heuristic Algorithm

Based on the idea of Pb-HBP, we derive a heuristic streaming algorithm by greedily assigning vertices to clusters. The greedy assignment is performed as follows. Given that the current vertex partitions are $\{V_1, V_2,\ldots,V_k\}$, a newly scanned vertex is assigned to a partition such that the goals of Pb-HBP are most likely to be satisfied. Suppose our cost function to be *minimized* is f, where $f(V_i\cup\{v\})$ is the cost when assigning v to V_i. Vertex v is assigned to partition i such that

$$f(V_i\cup\{v\}) \le f(V_j\cup\{v\}), \forall 1\le j\le k. \tag{13}$$

The greedy vertex assignment requires only one pass of the graph data, which is quite efficient for large graph. However, the key is to design a cost function that describes the goals of Pb-HBP.

There are two goals of Pb-HBP, which describe the per-bin hotness balancing requirement as shown in Equation (12) and the communication minimization requirement. We unify them in a single cost function and greedily choose the best partition i that results in the least cost.

$$\begin{aligned} i = \mathop{\arg\min}_{i:\left\{\boldsymbol{h}_{ji}\le\tau\frac{\sum_{v\in\mathcal{H}_j} h_v}{k}\right\}} & \alpha\cdot\left((\boldsymbol{h}_{ji}+h_v)^{\beta} - \boldsymbol{h}_{ji}^{\beta}\right) \\ & + (1-\alpha)\cdot\left(\sum_{u\notin V_i} com_{u,v} + \sum_{w\notin V_i} com_{v,w}\right), \end{aligned} \tag{14}$$

where $0\le\alpha\le 1$, $\tau > 1$, $\beta > 1$, and $\boldsymbol{h}_{ji} = \sum_{v\in\mathcal{H}_{ji}} h_v$ denotes the current sum of hotness values of partition G_i in the jth bin.

In this cost function, the first part describes the imbalance cost, while the second part describes the communication cost. Parameter α controls the weight of the workload imbalance cost and the communication cost. The imbalance cost is minimized when $H_{ji} = \frac{\sum_{i=1}^{k} H_{ji}}{k}$ for each i. Parameter τ is a small constant ($\tau > 1$) that defines the tolerance to hotness imbalance, so that the sum of hotness values in each partition is prevent from extreme large, i.e., $\sum_{v \in V_i} h_v \leq \tau \cdot \frac{\sum_{v \in V} h_v}{k}$. Parameter β controls how much preference to assign vertex to low hotness partitions since adding vertex to high hotness partitions may increase the risk of generating over-hot partitions, where larger β ($\beta > 1$) results in higher cost when assigning vertex to higher hotness partitions. The communication cost $\sum_{u \notin V_i} com_{u,v} + \sum_{w \notin V_i} com_{v,w}$ is depicted by the increased communication of cut edges when assigning v to the ith partition. Based on the cost function depicted in Equation (14), we sequentially read each vertex v and assign it to the partition i that results in the minimum cost.

Algorithm 1 summarizes the whole process of Streaming-based Per-Bin Hotness Balanced Partition. We first estimate the hotness values H of all vertices (Line 1) and initialize z bins $\{\mathcal{H}_1, \ldots, \mathcal{H}_z\}$ based on the hotness histogram of H (Line 2). Line 3 initializes vertex partitions, and Line 4 initializes the sum of hotness values in each bin partition. We then perform the vertex assignment operation. We measure the increased cost when assigning a vertex to each partition (Line 13). If a bin partition is over-hot, it will not be assigned with new nodes (Line 10 - 12). The partition that results in the minimum increased cost will be selected for assigning the vertex (Line 15 - 16), and the bin partition's hotness is correspondingly increased (Line 17).

Algorithm 1. Streamed Per-Bin Hotness Balanced Partition

```
Input: Input graph $G(V, E)$, number of partitions $k$, number of
       intervals $z$;
Output: Graph partitions $\{V_1, \ldots, V_k\}$;
 1: Estimate hotness values $H$ of all vertices;
 2: Init $\mathcal{H}_j$ based on histogram of $H$, $1 \leq j \leq z$;
 3: Init $V_i = \emptyset$, $1 \leq i \leq k$;
 4: Init $\boldsymbol{h}_{ji} = 0$, $1 \leq i \leq k$, $1 \leq j \leq z$;
 5: Assign($V$, $\{V_1, \ldots, V_k\}$);
 6: procedure Assign$V$, $\{V_1, \ldots, V_k\}$
 7:   for each $v$ in $V$ do
 8:     Find $\mathcal{H}_j$ where $v$ resides;
 9:     for each $V_i$ do
10:       if $\boldsymbol{h}_{ji} > \tau \cdot \frac{\sum_{v \in \mathcal{H}_j} h_v}{k}$ then
11:         $c_i = +\infty$;
12:       end if
13:       $c_i = \alpha \cdot \left((\boldsymbol{h}_{ji} + h_v)^{\beta} - \boldsymbol{h}_{ji}^{\beta}\right) + (1 - \alpha) \cdot \Big(\sum_{u \notin V_i} com_{u,v}$
          $+ \sum_{w \notin V_i} com_{v,w}\Big)$;
14:     end for
15:     $i = \arg\min_i c_i$;
16:     $V_i = V_i \cup v$;
17:     $\boldsymbol{h}_{ji} = \boldsymbol{h}_{ji} + h_v$;
18:   end for
19: end procedure
```

In this paper, we focus on edge-cut partitioning for demonstration, but it can be easily extended to vertex-cut partitioning [1], [7]. In vertex-cut partitioning, the edge data are evenly distributed to workers without copies, and the vertices that span workers with vertex replicas. Since each edge is stored exactly once, changes to edge data do not need communication. However, changes to vertex state must be copied to all the machines it spans, thus the storage and network overhead depend on the number of vertex replicas. In vertex-cut hotness balanced partition, the edge-centric processing model is adopted, so that the hotness is measured on a per-edge basis, which is the communication cost as defined in Equation (4). On the other hand, the communication cost in vertex-cut scenario becomes the vertex hotness as defined in Equation (3). Then the partition goals are redefined as follows. 1) balancing hotness values of edges; 2) making the edge hotness distributions consistent; 3) minimizing the number of vertex cuts. To achieve these goals, we can employ a streaming approach similar to [7], where the input is each edge instead of each vertex, and evaluate the cost score for each partition by considering the three optimization objectives. The partition that leads to the least score is picked for assigning this edge.

Algorithm 2. Local Compression

```
Input: Initial graph partition $G_i = (V_i, E_i)$, compression ratio $r$,
number of intervals $z$;
Output: Super-vertices set $\overline{V_i}$ and super-edges set $\overline{E_i}$;
 1: $\mathcal{H}_{ji}$ =Init $z$ hotness bins, $1 \leq j \leq z$;
 2: $v$-$\overline{v}$-$map$=Init vertex-to-supervertex map;
 3: for each $\mathcal{H}_{ji}$ do
 4:   $\{\overline{v}\}$ =Randomly pick $(r \cdot |\mathcal{H}_{ji}|)$ super-vertices
 5:   for each $v$ in $\mathcal{H}_{ji}$ do
 6:     for each super-vertex $\overline{v}$ do
 7:       $com_{v,\overline{v}} = 0$;
 8:       for each vertex $u$ in $\overline{v}$ do
 9:     $com_{v,\overline{v}} = com_{v,\overline{v}} + com_{v,u} + com_{u,v} + \sum_{w \in \{OUT(v) \cap OUT(u) \cap \mathcal{H}_j\}}$
        $(com_{u,w})$;
10:       end for
11:     end for
12:     Select $\overline{v}$ that results in maximal $com_{v,\overline{v}}$;
13:     $\overline{v} = \overline{v} \cup v$; $h_{\overline{v}} = h_{\overline{v}} + h_v$;
14:     Update $v$-$\overline{v}$-$map$;
15:   end for
16: end for
17: $\overline{V_i} = \{\overline{v}\}$;
18: Broadcast $v$-$\overline{v}$-$map$ to other workers;
19: Receive all $u$-$\overline{u}$-$map$ from other workers;
20: for each vertex $\overline{v} \in \overline{V_i}$ do
21:   for each vertex $v$ in $\overline{v}$ do
22:     for each $u \in OUT(v)$ do
23:       Lookup $\overline{u}$ in $u$-$\overline{u}$-$map$ where $u \in \overline{u}$;
24:       $com_{\overline{v},\overline{u}} = com_{\overline{v},\overline{u}} + com_{v,u}$;
25:     end for
26:   end for
27: end for
28: $\overline{E_i} = \{com_{\overline{v},\overline{u}} \,|\, \overline{v} \in \overline{V_i}\}$;
29: Send $\overline{V_i}$ and $\overline{E_i}$ to the global worker;
```

5 Distributed SPb-HBP

The proposed single-pass SPb-HBP algorithm sequentially processes vertices and their associated edges. This is efficient enough in most cases. However, when graph data is extremely large, the graph data may be stored distributively or collected distributively, i.e., graph data are initially

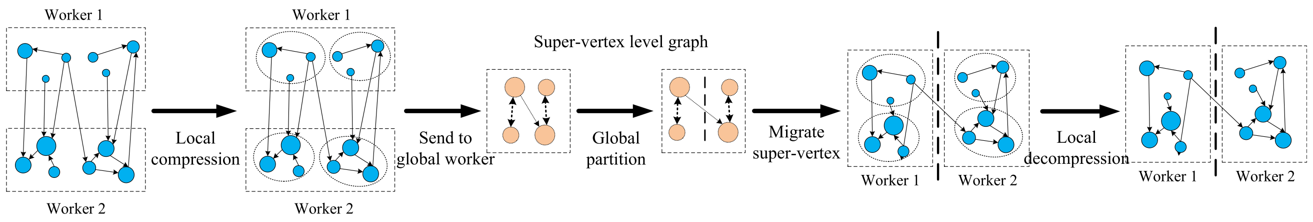


Fig. 7. Overview of distributed SPb-HBP graph partitioning.

stored on multiple workers. In order to use SPb-HBP to partition the graph, a distributed partitioning algorithm is desired.

Intuitively, a distributed graph partitioning algorithm requires a large amount of shuffle communication to exchange vertex data and edge data. To have a knowledge of neighborhood information, we may need several rounds of data shuffling and synchronizations, which may result in a large amount of network traffic and degrade the performance.

Our idea is described as follows. Each distributed worker compresses its assigned graph data into a compact and informative graph summary (with combined vertices and combined edges) that uncovers the underlying topology characteristics and the vertex/edge hotness of the original graph. These local graph summaries are then sent to a global worker, where SPb-HBP is performed on these local summaries to obtain a global partition result. Based on the global result on graph summaries, the combined vertices may be migrated from one worker to another worker, where they are decompressed to their original vertices. Since only the graph summaries are shuffled between workers, the resulted communication cost can be greatly saved.

We show the distributed graph partitioning process in Fig. 7, which contains three phases, including local compression, global partition, and local decompression. We present the details of the three phases in the following.

Local Compression. Local compression is performed on each worker in parallel. Based on the initial assigned partial graph data $G_i(V_i, E_i)$, we will compress the original graph data $G_i(V_i, E_i)$ into a graph summary $\overline{G_i}(\overline{V_i}, \overline{E_i})$ where $\overline{V_i}$ contains $r \cdot |V_i|$ super-vertices (r is a compression ratio parameter) and $\overline{E_i}$ represents the super-edges between super-vertices. We describe the local compression process in Algorithm 2.

To combine vertices into super-vertex, a few principles are considered. 1) The vertices that have higher communication cost should be combined in a super-vertex, in order to avoid cross-partition communication. 2) The vertices that have the same neighbor should be combined in a super-vertex, in order to reduce the number of super-edges. 3) Only the vertices that are in the same hotness bin can be combined, in order to keep the hotness bin information for the next global partition phase. Based on these principles, Line (9) of Algorithm 2 evaluates how much communication cost can be saved by assigning vertex v to super-vertex $\overline{v}$. Then we select the $\overline{v}$ that results in the maximal communication cost as vertex v's host super-vertex, and we can finally obtain the super-vertices set $\overline{V_i}$.

In order to measure the communication cost between super-vertices (i.e., super-edges), we need to have knowledge of the vertex-to-supervertex mapping information from other workers. Therefore, each worker broadcasts the computed the vertex-to-supervertex mapping information to other workers (Line 18), based on which each parallel worker computes the communication cost between its super-vertices and other super-vertices generated by other workers (Line 20-27). Finally, each worker sends the super-vertices set $\overline{V_i}$ and super-edges set with communication cost $\overline{E_i}$ to a global worker for global partition (Line 29).

Global Partition. A global worker is dedicated to performing global partition based on the collected local compressed subgraphs $\overline{G_i}(\overline{V_i}, \overline{E_i})$ from distributed workers. Since the original large graph has been significantly compressed, we can use the sequential one-pass SPb-HBP algorithm (Algorithm 1) to efficiently partition the compressed graph. It is noticeable that after local compression, the hotness values of super-vertices could be larger than that of their contained vertices, and the hotness levels of these super-vertices might be increased. Since we only combine vertices in the same hotness bin during local compression, we can keep these super-vertices in their original hotness bins during global SPb-HBP partition, in order to make the hotness distributions consistent among partitions.

Local Decompression. After global partition, the super-vertices with their associated original vertices and edges data are shuffled among workers to have a hotness balanced partition with minimum communication cost. These super-vertices are then decompressed on each worker in parallel. The vertices and edges are also relabelled with new ids based on the vertex-to-supervertex map and the supervertex-to-partition map. The distributed SPb-HBP graph partitioning is finished at this time. From Fig. 7 we can see that a more hotness balanced and communication-minimized partition is achieved.

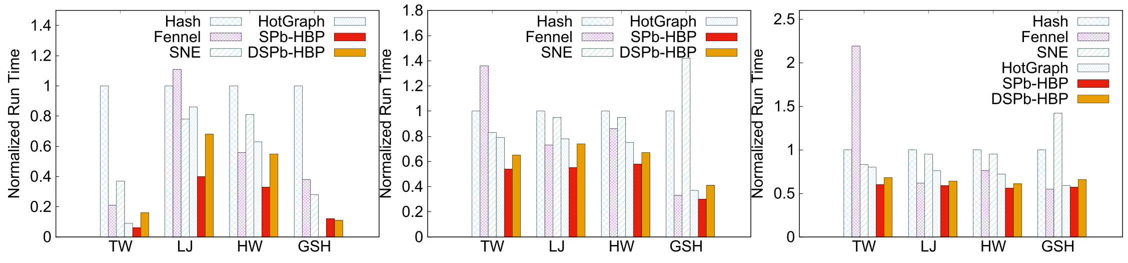
6 Experimental Evaluation

In this section, we present the experimental evaluation of our hotness balance partition.

6.1 Preparation

We first partition input graphs as a preprocessing step and then distributively run prioritized graph algorithms on Maiter [15]. Maiter is an implementation of the asynchronous DAIC framework and supports priority scheduling. We record the runtime of graph computation and the volume of network traffic to illustrate the effectiveness of different graph partitioning approaches.

Competitors. We compare our proposed SPb-HBP and its distributed version DSPb-HBP with the state-of-the-art stream partitioning methods Fennel [9], HotGraph [17], SNE [10] and Hash partition. Fennel is a vertex balanced



(a) Adsorption (b) PageRank (c) PHP

Fig. 8. Runtime comparison with different graph partitioning methods.

partition algorithm. It uses an objective function that combines vertex balance and number of cut edges to determine which partition a vertex should be assigned. HotGraph is designed for asynchronous parallel graph processing in shared-memory multi-core systems. It extracts the backbone structure of the graph and assigns it to a partition, so that most of vertex state propagations are occurred with this partition, which results in the communication overhead between partitions. SNE is a stream-based variant of the NE algorithm. NE [10] partitions edges iteratively for k rounds, where an edge partition is generated that greedily maximizes edge locality. The traditional iterative graph partitioning methods, such as METIS [26], usually require multiple passes of the graph data and multiple iterations for refining the partition result, which result in extremely high preprocessing cost (the graph partitioning time may be longer than the runtime of graph algorithms), so we do not compare with them for fairness.

Workload and Datasets. We perform evaluation on three representative prioritized graph algorithms, Adsorption, PageRank and Penalized Hitting Probability (PHP). We also select four different real graphs with different types, Twitter (TW) [27], Hollywood (HW) [28], LiveJournal (LJ) [29], and General Shallow Hosts (GSH) [30]. Based on these real graphs, we assign random weights to edges to construct weighted graphs. The detail information of these datasets is shown in Table 2.

Evaluation Metrics. We use the runtime and the communication cost of Adsorption [23], PageRank [22] and PHP [24] caused by different partitioning schemes as the metrics to evaluate the effectiveness of different partitioning methods. The Adsorption, PageRank and PHP are performed on Maiter framework to record the runtime. The communication cost is measured as the total network traffic volume generated during graph algorithms computation.

Experiment Cluster. Our experiments are conducted on a cluster of machines on Alibaba Cloud, which consists of 65 ecs.cs.large nodes with one additional node as Master. Each node runs Ubuntu 14.04 LTS and is equipped Intel Xeon (Skylake) Platinum 8163 CPU (2.5 GHz, 2 cores), 8 GB of memory and 40 GB hard disk. The nodes of the cluster are connected by 1 Gbps network.

TABLE 2
Datasets

Graph	Vertices	Edges	Type
TW	456,626	14,855,819	Social Network
HW	1,139,905	116,050,145	Collaboration Networks
LJ	4,847,571	69,555,947	Social Network
GSH	30,809,122	602,119,716	Web Graphs

Parameters Setup. By default, we set the number of bins $z = 2$ and the compression ratio $r = 0.01$, unless otherwise specified. For fairness, we set $\tau = 1.1$ which is the same as that in Fennel, SPb-HBP and DSPb-HBP. Our HBP's β and Fennel's γ play the same role, thus we set them both as 1.5. Following Fennel's original paper [9], we set $\alpha = \sqrt{k}\frac{\sum_{v\in V} h_v}{\sum_{(u,v)\in E} com_{u,v}^{3/2}}$, which depicts the tradeoff between the load imbalance cost and the communication cost. We set the other parameters of the competitors by referring to their papers.

6.2 Compare With Other Partition Methods

To test the effectiveness of our proposed hotness balanced partition methods, we compare SPb-HBP and DSPb-HBP with other state-of-the-art partitioning methods. We partition the input graph into 4 partitions by using different partitioning methods. We then use 5 workers (1 master and 4 slaves) to perform Adsorption, PageRank and PHP algorithms on these partitions (DSPb-HBP runs distributed partition on 4 nodes).

Fig. 8 shows the normalized runtime comparison on three graphs (TW, HW, LJ and GSH[1]). We use the runtime of Adsorption, PageRank and PHP caused by Hash partition as the baselines and draw the normalized runtime results caused by other partitions. Similarly, we show the normalized communication cost on these graphs in Fig. 9. It is noticeable that the runtime and communication cost are resulted from graph algorithm's computation but not graph partitioning.

We can see that SPb-HBP always results in shorter runtime than other methods no matter for Adsorption, PageRank or PHP computation. SPb-HBP can reduce 40-90 percent runtime of that by hash partition, 5-75 percent runtime of that by Fennel, 22-50 percent runtime of that by HotGraph, and 17-65 percent runtime of that by SNE. The distributed version of SPb-HBP, DSPb-HBP, shows less effectiveness than SPb-HBP since SPb-HBP compresses the original big graph into smaller coarse graph for stream partitioning. In general, DSPb-HBP results in shorter runtime than Fennel, HotGraph, and SNE except for the runtime of Adsorption on TW and PHP on GSH. This is because that Adsorption and PHP result in more communication cost on the TW and GSH datasets under DSPb-HBP. Besides,

1. In this experiment, we extracted 20 percent of the GSH to perform graph algorithms, because the whole GSH requires a too long time to converge.

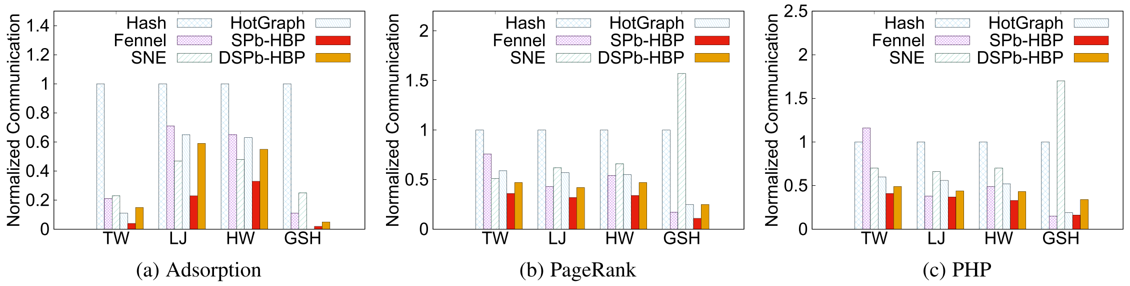


Fig. 9. Communication cost comparison with different graph partitioning methods.

TABLE 3
Partition Time and Runtime (PageRank)

	Hash	Fennel	SNE	HotGraph	SPb-HBP
PT(TW)	0s	2.0s	6.0s	4.8s	3.0s
RT(TW)	66.2s	90.2s	55.4s	52.7s	36.4s
PT(HW)	0s	14.6s	49.2s	32.0s	11.9s
RT(HW)	1327.3s	1142.8s	1266s	998.1s	783.0s
PT(LJ)	0s	14.4s	68.3	26.0s	12.6s
RT(LJ)	823.9s	601.8s	786s	646.6s	456.6s

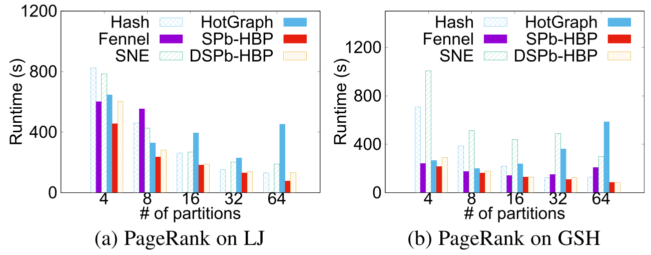


Fig. 10. Runtime comparison when scaling number of partitions.

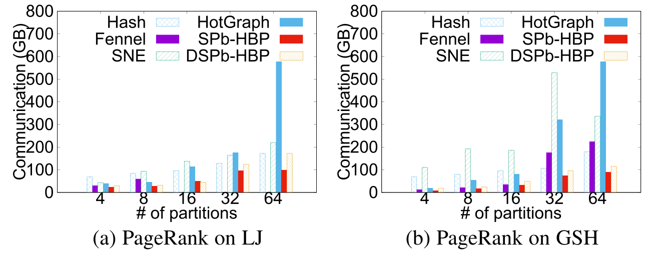


Fig. 11. Communication cost comparison when scaling the number of partitions.

Fennel, HotGraph and SNE even results in longer runtime and more communication than the naive Hash partition on the TW and GSH dataset. Based on our analysis, we found that they cut a large number of heavy traffic edges as it does not consider edge hotness. In distributed priority scheduling systems, the delayed transmission of important messages may impact the efficiency of graph algorithms [17]. What's worse, the GSH's partitions produced by HotGraph, the Adsorption algorithm fail to convergence in a reasonable time period.

We also show the partition time (PT) and PageRank runtime (RT) of these partition methods on different datasets in Table 3. The partition time only contains the time for computing the vertices assignment information without the time for deploying these vertices. Comparing with computation runtime, these partition methods are all efficient and take much shorter partition time. Hash partition does not involve any graph structure related computation, so its partition time can be ignored. SPb-HBP and Fennel are both streaming-based one-pass partition algorithms, so they are faster than HotGraph. As a stream-based partition algorithm, SNE's processing unit is edge, i.e., it will compute the best partition assignment for each edge. Since the number of edges is much larger than that of vertices, SNE requires more computation time than the vertex-based methods. DSPb-HBP is a distributed partition method. It is not fair to compare it with the single machine partition method. We will evaluate the partition time of DSPb-HBP in later experiments.

6.3 Scaling Performance

It is necessary to show the performance when running on large sized cluster with more partitions. The larger the cluster size (i.e., the greater the number of partitions), the larger the amount of communications, due to more edge cuts. We evaluate the effectiveness of these partitioning methods by running on larger sized clusters (i.e., varying the number of partitions). With different partitioning methods, we partition the LJ and GSH graphs into 4, 8, 16, 32 and 64 partitions respectively,[2] and run PageRank algorithm on the clusters with 4, 8, 16, 32, and 64 nodes, respectively. Figs. 10 and 11 show the comparison results of runtime and communication cost, respectively.

In most cases, with the increase of the partition numbers, the runtime of PageRank is decreasing while the communication is increasing. But our SPb-HBP and DSPb-HBP partition methods always result in shorter runtime and less communication cost than their counterparts when varying the number of partitions (varying the cluster size). Again, SPb-HBP and DSPb-HBP exhibit comparable effectiveness. It is notable that with the graph partitions produced by Fennel, the PageRank algorithm fails to convergence in a reasonable time on the LJ dataset when the partition number is 16 or larger. This is because that Fennel may cut the important edges and would delay the important message passing, which will seriously hurt the performance of asynchronous framework Maiter. With HotGraph partition results, the runtime of PageRank first decreases then increases with the increase of the partition numbers. This is because that HotGraph is designed for shared-memory environment and

2. DSPb-HBP runs distributed partition on different sized clusters.

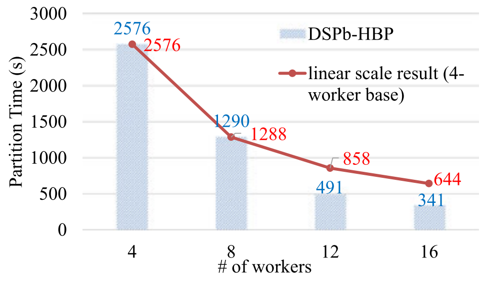


Fig. 12. Graph partitioning time of DSPb-HBP when scaling number of workers (GSH graph).

does not take the cross-worker communication cost into consideration. With the increase of the partition number, the communication cost can increase dramatically and overweigh the benefit of more computation resources.

6.4 Efficiency of DSPb-HBP

DSPb-HBP is a distributed partition approach, which can be expensive when partitioning massive graphs on a large cluster. We also test the efficiency of DSPb-HBP on the large GSH graph dataset. In this experiment, we partition the graph with different numbers of workers and record the partition time rather than the runtime of graph algorithms. We set the compression ratio r of DSPb-HBP as 0.001.

Fig. 12 shows the partition time of DSPb-HBP when the number of workers varies from 4 to 16. For a given graph, the partition time should be inversely proportional to the number of workers. Then, we use the partition time with 4 workers as the baseline and draw the linear scale result, i.e., $t_k = t_4 \cdot \frac{4}{k}$, where t_k is the partition time with k workers. Our DSPb-HBP shows a better result than the linear scale result. This is because that there are some high-degree vertices in GSH, which need long time to compress. With fewer workers, the problem of load imbalance for these high-degree vertices is more serious. This shows the superiority of DSPb-HBP on a larger scale cluster (i.e., scalability).

6.5 Parameter Studies

There are two important parameters in our approach, the number of bins z in Pb-HBP and the compress ratio r in DSPb-HBP. We study the effects of z and r by partitioning LJ graph into 4 partitions with different z and r value and performing the PageRank. Fig. 13a shows the running time of PageRank when varying the number of bins z from 2 to 64. We can see that the runtime is reduced at the beginning but is almost stable when z is larger than 4.

Fig. 13b shows the running time of PageRank when varying the compression ratio r. With a small compression ratio r, the runtime of PageRank prolongs. This is because that a small compression ratio results in fewer super vertices, which could result in inaccurate partitioning results. However, even when $r = 0.01$, we still can get a better partitioning result than other state-of-the-art methods, as shown in Fig. 8

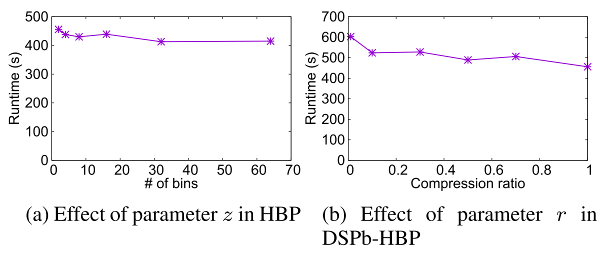


Fig. 13. The effect of parameters.

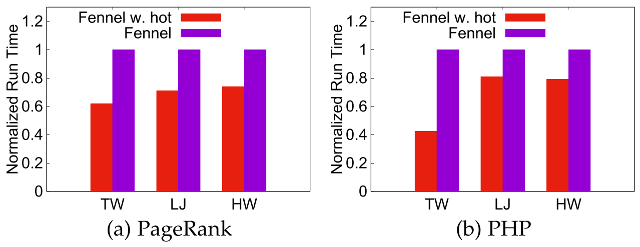


Fig. 14. Fennel w. hot versus Fennel on running time.

6.6 Hotness Balanced Versus Vertex Balanced

In order to directly verify the idea of hotness balanced partition, in this experiment, we extend the Fennel partition to support hotness balance partition. Basically, in the objective function of Fennel, we replace the number of vertices with the sum of vertex hotness values, and replace the number of cut edges with the sum of communication of cut edges. With such an extension, the new Fennel partition has been upgraded to take hotness into account. We then compare Fennel with the new Fennel with hotness consideration (Fennel w. hot) to illustrate how much can be achieved by considering hotness balance. In this experiment, we use the same parameters as used in [9] and partition the graph into 4 partitions and use 4 workers to run graph algorithms.

Fig. 14 shows the runtime of PageRank and PHP on three graphs. Fig. 15 shows the traffic volume in cluster. Fennel partition with hotness consideration reduces the runtime and communication to about 40-80 percent of original Fennel partition.

6.7 Accuracy of Hotness Estimation

A critical work of HBP is to estimate the hotness of vertices precisely. The hotness cannot be known before the computation finishes but we need to pre-know the approximate hotness values in our partition algorithm. We use a lightweight hotness estimation method in Section 3.1 which only considers the first-order neighbors. In this experiment, we evaluate the accuracy of our hotness estimation method.

The real hotness of each vertex can be measured by the number of updates in a stand-alone single thread environment. Because in stand-alone single thread environment, hotter vertices always get updated more frequently with the

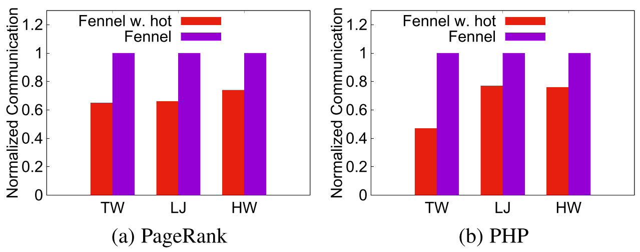


Fig. 15. Fennel w. hot versus Fennel on communication cost.

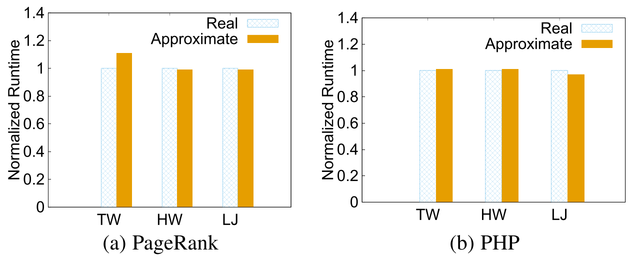


Fig. 16. The impact of real hotness versus approximate hotness.

help of priority scheduling, which can avoid the uncertainty resulted from the parallel execution environment. Therefore, we record the number of updates of each vertex in a stand-alone single thread environment as the "real" hotness of each vertex. We treat the hotness that obtained by our estimation method as approximate hotness. To test the effectiveness of our estimation method, we use the "real" hotness and the approximate hotness to partition the graph.

Fig. 16 shows the normalized runtime PageRank and PHP on three graphs when we using the real hotness and the approximate hotness. We can see that our partition method using the approximate hotness has comparable effectiveness with that using "real" hotness. It only shows less than 15 percent difference in runtime.

7 Related Work

Graph partitioning is an essential yet challenging task for massive graph analysis in distributed computing. The problem of graph partitioning has been thoroughly studied for a few decades. In recent years, streaming-based graph partitioning and vertex-cut graph partitioning are two main directions, which will be discussed in the following.

Streaming-Based Graph Partitioning. As opposed to offline methods, which first load the complete graph in memory and then divide it into partitions, streaming graph partitioning operates online, while ingesting the graph data as a stream [8]. Linear Deterministic Greedy partitioning (LDG) [8] uses a greedy heuristic that assigns a vertex to the partition containing most of its neighbors while respecting certain capacity constraints. FENNEL [9] is another streaming-based partitioning scheme whose heuristic combines locality-centric measures (low edge-cut) with balancing goals. HDRF [7] and IOGP [31] are streaming-based vertex-cut partitioning methods. HDRF is designed for distributed OLAP transactions, while IOGP is designed for distributed OLTP transactions. IOGP takes the connectivity and the vertex degree changes into account when partitioning graphs. Degree Based Hashing (DBH) [32] employs hashing for partitioning and prioritizes cutting those vertices that have the highest degree. Minit [33] is a quasi-streaming graph partitioning method by separating the whole edge stream into a series of batches where the batch size is a constant multiple of the number of partitions. LEOPARD [34] and LogGP [35] both use streaming-based method to dynamically repartition the massive graphs in accordance with the structural changes. AKIN [36] exploits the similarity measure on the degree of vertices to gather structural-related vertices in the same partition as much as possible, which reduces the edge-cut ratio. More related works on streaming-based graph partitioning can be referred to [37], [38].

Edge-Cut and Vertex-Cut. Edge-cuts approach is used by systems such as GraphLab [2], LFGraph [39], PBE [40] and Pregel [3]. For systems that utilize edge-cuts, vertices are assigned to partitions and thus edges can span partitions. In contrast, in vertex-cuts partitioning scheme, edges are assigned to partitions and thus vertices can span partitions. Unlike edges which could be cut across only two partitions, a vertex can be cut across several replicas as its edges may be assigned to several partitions. In PowerGraph [1], the authors first demonstrate that the presence of very high-degree vertices in power-law graphs present unique challenges from a partitioning perspective, and motivate the use of vertex-cuts in such cases. GraphX [41] provides a range of built-in partitioning functions. For efficient lookup of edges by their source and target vertices, the edges within a partition are clustered by source vertex id using a compressed sparse row representation and hash-indexed by their target id. SBV-Cut [42] presents a structural balance vertex-cut graph partitioning algorithm, which searches for vertices where the graph is balanced in terms of distances to the extremities as well as its connectivity to the rest and cuts the graph incrementally along these dominant balance vertices.

Application-Driven Partitioning. The optimal graph partitioning strategy may depends on the application characteristics, the input graphs and the number of workers [43]. PowerLyra [6] takes a hybrid partitioning approach, which applies edge-cut to low-degree vertices and vertex-cut to high-degree vertices, which aims to reduce the replication factor of low-degree vertices, since high-degree vertices inevitably need to be replicated on most of the machines. MDBGP [44] produces balanced partitions according to multiple user-specified weight functions while maintaining edge locality. Fan *et al.* [45] propose an application-driven (APD) hybrid partitioning strategy to learn a cost model for different applications, and develop partitioner that refines an edge-cut or vertex-cut partitioning to fit in with the cost patterns of applications.

Distributed Graph Partitioning. To distributively partition large-scale graphs, several distributed graph partitioning approaches are proposed. Ja-be-Ja [46] swaps the vertices between partitions based on the number of vertices' neighbours, such that the locality of partition is improved meanwhile the number of vertices between partitions is balanced. Sheep [47] transforms the input graph into a strictly smaller elimination tree via a distributed map-reduce operation. Spinner [48] and XtraPuLP [49] partition the graph based on label propagation. XtraPuLP uses weighted label propagation to achieve the balancing objective. In general, these distributed graph partitioning methods use an iterative algorithm to swap the vertices between partitions or construct a distributed structure to help find a better partition. In addition, it is also feasible to rely on streaming-based partitioning and use multi-stream strategy to partition graph in parallel, e.g., PowerGraph [1].

Prioritized Asynchronous Graph Processing. All the above graph partitioning algorithms are designed for distributed synchronous frameworks. They assume that the graph processing system uses a round-robin scheme to process the assigned vertices in parallel. Recently, asynchronous parallel model has attracted more and more researchers for accelerating large-scale graph processing since it can avoid the costly synchronization overhead in a large-scale cluster environment. Several asynchronous graph processing systems have emerged, such as GraphLab [2], Maiter [15] and GRAPE+ [14].

HotGraph [17] is recently proposed for asynchronous parallel graph processing system. However, HotGraph is significantly different from our proposed method in the following two aspects. First, it still assumes round-robin scheduling or FBS [17] scheduling, but not priority scheduling. Second, HotGraph is proposed for shared-memory environment but not for shared-nothing distributed environment. It divides the graph into a single hot partition and multiple cold partitions, which is unbalanced partitioning and not suitable for distributed computation.

8 Conclusion

The traditional k-balanced graph partitioning fails to work in prioritized asynchronous iterative frameworks. In this paper, we propose a novel graph partitioning idea, Hotness Balanced Partition, tailored to prioritized scheduling frameworks. We propose the single-pass Pb-HBP algorithm to efficiently partition graphs and propose a distributed version for partitioning large-scale graphs. Our results show that our proposed graph partitioning scheme can greatly improve the performance of prioritized graph computations and at the same time is quite efficient to partition large-scale graphs.

Acknowledgments

This work was supported in part by National Key R&D Program of China (under Grant 2018YFB1003404), National Natural Science Foundation of China (under Grants 62072082, 61672141, and U1811261) and Fundamental Research Funds for the Central Universities (under Grant N181605017 and N181604016), and Key R&D Program of Liaoning Province 2020JH 2/10100037. A preliminary version of this article has appeared in ICDE 2020 [50].

References

[1] 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.

[2] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: A framework for machine learning and data mining in the cloud,” *Proc. VLDB Endowment*, vol. 5, pp. 716–727, 2012.

[3] G. Malewicz *et al.*, “Pregel: A system for large-scale graph processing,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.

[4] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “iMapReduce: A distributed computing framework for iterative computation,” *J. Grid Comput.*, vol. 10, pp. 47–68, 2012.

[5] 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.

[6] R. Chen, J. Shi, Y. Chen, and H. Chen, “PowerLyra: Differentiated graph computation and partitioning on skewed graphs,” in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–15.

[7] 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.

[8] I. Stanton and G. Kliot, “Streaming graph partitioning for large distributed graphs,” in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2013, pp. 1222–1230.

[9] 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.

[10] C. Zhang, F. Wei, Q. Liu, Z. G. Tang, and Z. Li, “Graph edge partitioning via neighborhood heuristic,” in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2017, pp. 605–614.

[11] H. Reittu, I. Norros, T. Räty, M. Bolla, and F. Bazsó, “Regular decomposition of large graphs: Foundation of a sampling approach to stochastic block model fitting,” *Data Sci. Eng.*, vol. 4, pp. 44–60, 2019.

[12] M. Han and K. Daudjee, “Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems,” *Proc. VLDB Endowment*, vol. 8, pp. 950–961, 2015.

[13] A. Balmin, A. Balmin, S. A. Corsten, S. Tatikonda, and J. Mcpherson, “From ”think like a vertex” to ”think like a graph”,” *Proc. VLDB Endowment*, vol. 7, pp. 193–204, 2013.

[14] W. Fan *et al.*, “Adaptive asynchronous parallelization of graph algorithms,” in *Proc. Int. Conf. Manage. Data*, 2018, pp. 1141–1156.

[15] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 8, pp. 2091–2100, Aug. 2014.

[16] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Priter: A distributed framework for prioritized iterative computations,” in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 1–13.

[17] Y. Zhang, X. Liao, H. Jin, L. Gu, G. Tan, and B. B. Zhou, “HotGraph: Efficient asynchronous processing for real-world graphs,” *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 799–809, May 2017.

[18] J. Yin and L. Gao, “Scalable distributed belief propagation with prioritized block updates,” in *Proc. 23rd ACM Int. Conf. Inf. Knowl. Manage.*, 2014, pp. 1209–1218.

[19] J. Yin and L. Gao, “Asynchronous distributed incremental computation on evolving graphs,” in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, 2016, pp. 722–738.

[20] Z. Wang, L. Gao, Y. Gu, Y. Bao, and G. Yu, “A fault-tolerant framework for asynchronous iterative computations in cloud environments,” in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 71–83.

[21] Q. Wang *et al.*, “Automating incremental and asynchronous evaluation for recursive aggregate data processing,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2439C–2454.

[22] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the web,” Tech. Rep. 1999–66.

[23] S. Baluja *et al.*, “Video suggestion and discovery for YouTube: Taking random walks through the view graph,” in *Proc. 17th Int. Conf. World Wide Web*, 2008, pp. 895–904.

[24] Z. Guan, J. Wu, Q. Zhang, A. Singh, and X. Yan, “Assessing and ranking structural correlations in graphs,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 937–948.

[25] C. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.

[26] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998.

[27] J. Leskovec and A. Krevl, “SNAP datasets: Stanford large network dataset collection,” Jun. 2014. [Online]. Available: http://snap.stanford.edu/data

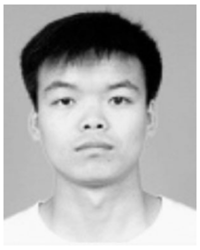
[28] P. Boldi, M. Rosa, M. Santini, and S. Vigna, “Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks,” in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 587–596.

[29] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, “Group formation in large social networks: Membership, growth, and evolution,” in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2006, pp. 44–54.

[30] P. Boldi, A. Marino, M. Santini, and S. Vigna, “BUbiNG: Massive crawling for the masses,” in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 227–228.

[31] D. Dai, W. Zhang, and Y. Chen, “IOGP: An incremental online graph partitioning algorithm for distributed graph databases,” in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 219–230.

[32] 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.
[33] Q.-S. Hua, Y. Li, D. Yu, and H. Jin, “Quasi-streaming graph partitioning: A game theoretical approach,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 7, pp. 1643–1656, Jul. 2019.
[34] J. Huang and D. J. Abadi, “Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs,” *Proc. VLDB Endowment*, vol. 9, pp. 540–551, 2016.
[35] N. Xu, L. Chen, and B. Cui, “LogGP: A log-based dynamic graph partitioning method,” *Proc. VLDB Endowment*, vol. 7, pp. 1917–1928, 2014.
[36] W. Zhang, Y. Chen, and D. Dai, “AKIN: A streaming graph partitioning algorithm for distributed graph storage systems,” in *Proc. 18th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2018, pp. 183–192.
[37] Z. Abbas, V. Kalavri, P. Carbone, and V. Vlassov, “Streaming graph partitioning: An experimental study,” *Proc. VLDB Endowment*, vol. 11, pp. 1590–1603, 2018.
[38] A. Pacaci and M. T. Özsu, “Experimental analysis of streaming algorithms for graph partitioning,” in *Proc. Int. Conf. Manage. Data*, 2019, pp. 1375–1392.
[39] I. Hoque and I. Gupta, “LFGraph: Simple and fast distributed graph analytics,” in *Proc. 1st ACM SIGOPS Conf. Timely Results Operating Syst.*, 2013, Art. no. 9.
[40] W. Guo, Y. Li, M. Sha, B. He, X. Xiao, and K.-L. Tan, “GPU-accelerated subgraph enumeration on partitioned graphs,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1067–1082.
[41] 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.
[42] M. Kim and K. S. Candan, “SBV-Cut: Vertex-cut based graph partitioning using structural balance vertices,” *Data Knowl. Eng.*, vol. 72, pp. 285–303, 2012.
[43] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, “A study of partitioning policies for graph analytics on large-scale distributed platforms,” *Proc. VLDB Endowment*, vol. 12, pp. 321–334, 2018.
[44] D. Avdiukhin, S. Pupyrev, and G. Yaroslavtsev, “Multi-dimensional balanced graph partitioning via projected gradient descent,” in *Proc. IEEE Symp. Very Large-Scale Data Anal. Vis. Endowment*, 2019, vol. 12, pp. 906–919.
[45] W. Fan *et al.*, “Application driven graph partitioning,” in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 1765–1779.
[46] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi, “JA-BE-JA: A distributed algorithm for balanced graph partitioning,” in *Proc. IEEE 7th Int. Conf. Self-Adaptive Self-Organizing Syst.*, 2013, pp. 51–60.
[47] D. Margo and M. Seltzer, “A scalable distributed graph partitioner,” *Proc. VLDB Endowment*, vol. 8, pp. 1478–1489, 2015.
[48] 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.
[49] G. M. Slota, S. Rajamanickam, K. Devine, and K. Madduri, “Partitioning trillion-edge graphs in minutes,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 646–655.
[50] 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.



Shufeng Gong received the MS degree in computer science from Northeastern University, China, in 2016. He is currently working toward the PhD degree in computer science at Northeastern University, China. His research interests include cloud computing, distributed graph processing, and data mining.

Yanfeng Zhang received the PhD degree in computer science from Northeastern University, China, in 2012. He is currently a professor with Northeastern University, China. His research consists of distributed systems and big data processing. He has published many papers in the above areas. His paper in SoCC 2011 was honored with “Paper of Distinction”.

Ge Yu (Senior Member, IEEE) received the PhD degree in computer science from Kyushu University, Japan, in 1996. He is currently a professor with Northeastern University, China. His current research interests include distributed and parallel systems, cloud computing and big data management, blockchain techniques and systems. He has published more than 200 papers in refereed journals and conferences. He is the ACM member, the IEEE senior member, and the CCF fellow.