GraBi: Communication-Efficient and Workload-Balanced Partitioning for Bipartite Graphs

Feng Sheng
KLISS, WNLO, HUST
Wuhan, Hubei, China
shengfeng@hust.edu.cn

Qiang Cao*
KLISS, WNLO, HUST
Wuhan, Hubei, China
caoqiang@hust.edu.cn

Hong Jiang
University of Texas at Arlington
Arlington, TX 76019, USA
hong.jiang@uta.edu

Jie Yao*
Sch. of Computer Sci. & Tech., HUST
Wuhan, Hubei, China
jackyao@hust.edu.cn

ABSTRACT

Machine Learning and Data Mining (MLDM) applications, such as recommendation and topic modeling, generally represent their input data in bipartite graphs with two disjoint vertex-subsets connected only by edges between them. Despite the prevalence of bipartite graphs, existing graph partitioning frameworks have rarely sufficiently exploited their unique structures, especially the highly lopsided subset sizes and extremely skewed vertex degrees. As a result of poor partitioning quality, problems, particularly of high communication cost and severe workload imbalance, arise during subsequent computation over these bipartite graphs in distributed environments such as datacenters or HPC systems, significantly hampering the performance of MLDM applications.

In this paper, we approach these problems by communication-efficient and workload-balanced partitioning of bipartite graphs, which fully exploits the vertex vectorization in MLDM algorithms and inherent asymmetry in bipartite graphs. To this end, we present GraBi, a two-stage partitioning framework that partitions a bipartite graph first vertically and then horizontally. The first partitioning stage divides each vectored vertex into multiple vertex-chunks such that the bipartite graph is *vertically* partitioned into multiple layers, to strike an appropriate tradeoff between inter- and intra-vertex communication. In the second partitioning stage, for each layer, the vertex-chunks in the larger vertex-subset are first assigned to nodes, to minimize vertex replicas. To be specific, these vertex-chunks are *horizontally* decomposed into one or more sub-chunks with an upper-bounded number of edges, and then the sub-chunks are evenly assigned over nodes of a distributed system using a set of hash functions, to achieve workload balance among all computing nodes. GraBi is a lightweight partitioning framework for bipartite graphs, and can be generalizable to most MLDM applications. Our evaluation, driven by real-world bipartite graphs processed in an 8-node cluster, shows that GraBi significantly improves the partitioning quality for bipartite graphs. Particularly, it decreases the computation time of MLDM algorithms by up to 5.41x, 4.32x, 1.89x over three state-of-the-art partitioning frameworks Hybrid-cut, Bi-cut, and 3D-partitioner respectively.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**.

KEYWORDS

Distributed graph processing, graph partitioning, bipartite graphs

ACM Reference Format:
Feng Sheng, Qiang Cao, Hong Jiang, and Jie Yao. 2020. GraBi: Communication-Efficient and Workload-Balanced Partitioning for Bipartite Graphs. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404453

1 INTRODUCTION

Large-scale graph processing has become increasingly important for a broad range of applications, such as recommendation [19] and topic modeling [20]. In essence, these problems can be encoded as vertex-centric programs following the "think-like-a-vertex" philosophy [18], where vertices update their respective values concurrently and communicate with one another through edges. Although a number of remarkable projects [14, 25] have been developed to accommodate graph processing on a single node for cost-effectiveness, their performance and scalability are severely challenged by the rapid growth of graph datasets. As an alternative, many distributed graph processing systems [4, 9, 10] have recently emerged to perform graph computation on a cluster of nodes.

For distributed processing of large-scale graphs, graph partitioning is a mandatory preprocessing step before computation. It distributes vertices and edges in the graph onto all nodes, and creates replicas of vertices and edges to generate a local consistent subgraph on each node. During subsequent graph computation, the replicas synchronize with their masters in every update iteration, inevitably leading to often costly cross-node communication. Besides

*Qiang Cao and Jie Yao are the joint corresponding authors. Key Laboratory of Information Storage System (KLISS), Ministry of Education. Wuhan National Laboratory for Optoelectronics (WNLO), Huazhong University of Science and Technology (HUST).

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404453

the replicas, graph partitioning also determines the number of vertices and edges assigned to each node, thus impacting the balance (or lack thereof) of workload among nodes. Therefore, graph partitioning plays a vital role in distributed graph processing, directly affecting the communication cost and workload distribution.

Bipartite graphs, as an important class of graphs, have been widely used in Machine Learning and Data Mining (MLDM) applications. In a bipartite graph, the vertices are separated into two disjoint subsets, and every edge connects exactly one vertex each from the two subsets. Many complex relationship networks in real life can be abstracted as bipartite graphs, such as the user-item relationships in a recommendation system. MLDM algorithms are then executed on these bipartite graphs to update vertex values iteratively, and finally provide the computational results to users.

Despite the prevalence of bipartite graphs, existing partitioning frameworks in distributed graph processing systems are oblivious to some of their unique graph structures. For example, many frameworks [9, 10] do not distinguish the vertices from different subsets, and apply a uniform partitioning strategy to all vertices, thus incurring high communication cost and workload imbalance during graph computation. Although some works have considered designing tailored partitioning frameworks for bipartite graphs, they are usually inefficient in three key aspects. First, they [7] require prior knowledge of the entire graph or a time-consuming multi-round repartitioning stage (i.e., not lightweight), making them unscalable to large-scale graphs; Second, they [8] propose partitioning strategies suitable only for a specific type of MLDM algorithms (i.e., not generalizable), rendering their applicability severely limited; And third, they (e.g., Hybrid-cut [4], Bi-cut [5], and 3D-partitioner [23]) exploit only parts of the structures of bipartite graphs, failing to fully exploit useful features of the graphs.

In this paper, we attempt to propose a *lightweight* and *generalizable* partitioning framework for bipartite graphs, which comprehensively exploits the structures of bipartite graphs to improve overall performance. For this purpose, we investigate and analyze many real-world bipartite graphs and popular MLDM algorithms, explicitly revealing three important observations that can be leveraged to improve the partitioning quality of bipartite graphs: 1) The value of each vertex in MLDM algorithms is generally a vector of multiple data elements; 2) The sizes of the two vertex-subsets in a bipartite graph can be significantly lopsided; 3) The degrees of vertices within each subset are often highly skewed. Motivated by the three observations, we present GraBi, a communication-efficient and workload-balanced partitioning framework for bipartite graphs, which assigns graph data over nodes in a structure-aware and fine-grained way. To be specific, GraBi is composed of two sequential processing stages, partitioning a bipartite graph first vertically and then horizontally, briefly as follows:

Vertex-vector Chunking: Given that the value of each vertex in MLDM algorithms is usually a multi-element vector, the elements in the vector can be grouped and then assigned to different nodes. Therefore, this stage *vertically* divides, or chunks, each vertex vector into multiple subvectors, referred to as vertex-chunks, each of which comprises a portion of the elements. Accordingly, the whole bipartite graph is vertically partitioned into multiple graph layers, and each layer consisting of a certain vertex-chunk each from all vertices is allocated to a fraction of nodes in the cluster. As a result, every node is assigned with fine-grained vertex-chunks, and the interrelated vertex-chunks of more vertices can be localized on a single node, potentially decreasing the communication cost between different vertices, or inter-vertex communication. However, this stage also introduces a new kind of cross-node communication between different vertex-chunks belonging to the same vertex, or intra-vertex communication. To this end, we propose an empirical method for determining the number of layers, to strike an appropriate tradeoff between the inter- and intra-vertex communication.

Vertex-chunk Assignment: By exploiting the lopsided sizes between the two vertex-subsets of a bipartite graph, this stage assigns the vertex-chunks in the larger subset first within each layer. As a result, vertex replicas are created only for the smaller subset and thus the communication cost can be greatly reduced. Furthermore, to eliminate workload imbalance incurred by the skewed vertex degrees within each vertex-subset, this stage *horizontally* decomposes every high-degree vertex-chunk into multiple sub-chunks with an upper-bounded edge-count, and evenly assigns them over nodes using a set of hash functions. Thus, edges of each high-degree vertex-chunk are distributed over multiple nodes for concurrent updates, and edges of each low-degree vertex-chunk are localized on a single node for accessing locality, with negligible overhead.

Specifically, the contributions of this paper are as follows:

(1) We investigate real-world bipartite graphs and MLDM algorithms, and explicitly reveal three key observations.
(2) We present GraBi, a lightweight and generalizable partitioning framework that partitions a bipartite graph vertically and then horizontally, fully exploiting these observations.
(3) We evaluate GraBi with three MLDM algorithms and five real-world bipartite graphs in an 8-node cluster. The experiments show that GraBi reduces the computation time by up to 5.41x over Hybrid-cut, 4.32x over Bi-cut, and 1.89x over 3D-partitioner respectively.

The rest of this paper is organized as follows. Section 2 overviews the background and motivates the proposal of the two-stage partitioning framework GraBi, which is elaborated in Section 3. Section 4 reports and analyzes the experimental results. We discuss related works in Section 5 and conclude this paper in Section 6.

2 BACKGROUND AND MOTIVATION

In this section, we first briefly introduce the background of graph partitioning, bipartite graphs, and Machine Learning and Data Mining (MLDM) algorithms. Then, we present the three important observations that motivate our work.

2.1 Graph Partitioning

Distributed graph processing systems heavily rely on graph partitioning to evenly distribute vertices and edges over all computing nodes, in order to achieve low communication cost and workload balance among the nodes during subsequent graph computation. Recently, many graph partitioning strategies have been proposed, which can be roughly classified into two categories: edge-cut and vertex-cut. Edge-cut tends to equally distribute vertices among nodes. As an example shown in Figure 1(a), there are totally 4 vertices and 3 nodes, and each vertex is assigned to a node along

Graph	Abbr.	$\|\mathbb{U}\|$	$\|\mathbb{V}\|$	$\|E\|$	$\|\mathbb{U}/\mathbb{V}\|$	$\lambda_{Hybrid-cut}$	λ_{Bi-cut}	$\lambda_{3D-partitioner}$	λ_{GraBi}
DBLP	DBLP	4,000K	1,426K	8.6M	2.81	2.74	3.08	1.38	1.45
Netflix	NF	480K	18K	100.5M	27.02	3.37	2.14	1.16	1.20
LiveJournal	LJ	7,489K	3,201K	112.3M	2.34	2.64	3.47	1.30	1.52
Yahoo	YH	1,001K	625K	256.8M	1.60	3.34	4.43	1.53	1.56
Orkut	OK	8,731K	2,783K	327.0M	3.14	3.20	3.26	1.44	1.51

Table 1: A collection of bipartite graphs. λ_x represents the replication factor achieved by the partitioning framework x.

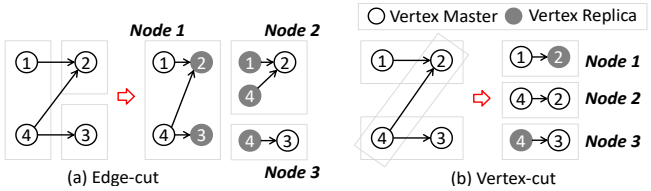


Figure 1: Edge-cut versus vertex-cut.

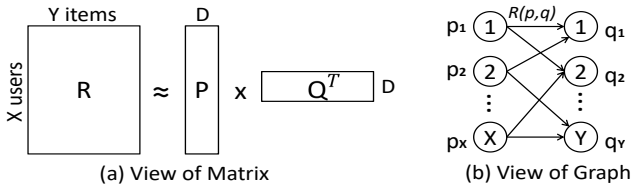


Figure 2: Collaborative Filtering problem.

with all its adjacent edges. Specially, an edge is cut if its two endpoints are distributed onto different nodes. As a result, edge-cut creates replicas of every cut edge and remote neighboring vertex, to generate a local subgraph on each node.

In contrast, vertex-cut attempts to evenly distribute edges among nodes. As an example shown in Figure 1(b), each edge is assigned to a node along with its two endpoints. Particularly, a vertex is cut if its adjacent edges are distributed onto different nodes, leading to some replicas of the vertex. Afterwards, one of the replicas is selected as the *vertex master* according to a certain algorithm [9]. Note that, since vertex-cut assigns each edge along with both its two endpoints, there are no cut edges and edge replicas. During graph computation, vertex values are synchronized between vertex masters and their replicas on different nodes, so that the total communication cost is mainly decided by the number of replicas. Therefore, the average number of replicas per vertex, formally termed as *replication factor* (λ), is also an important metric for graph partitioning. In this work, we adopt the widely-used vertex-cut strategy, because computational workload of many graph applications largely depends on the number of edges [14], which is the target that vertex-cut tries to balance.

2.2 Bipartite Graphs and MLDM algorithms

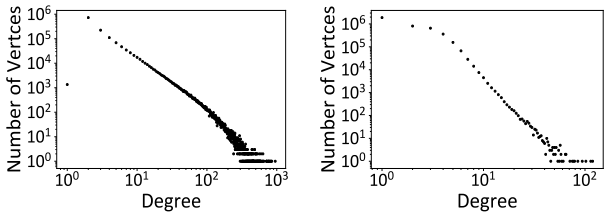
A bipartite graph G is formally represented as $G(\mathbb{U}, \mathbb{V}, E)$, where $\mathbb{U}$ and $\mathbb{V}$ are two disjoint subsets of vertices, and every edge in E connects a vertex from $\mathbb{U}$ to one from $\mathbb{V}$. Many real-world relationship networks can be modeled as bipartite graphs, upon which MLDM algorithms are executed to extract valuable insights for users. Take the Collaborative Filtering problem for example, it predicts the missing ratings from users to items, based on existing ratings. In a matrix representation of the problem, the input is a sparse rating matrix R, and the goal is to find two dense matrices P and Q, such that $R \approx P \times Q^T$. As shown in Figure 2(a), X and Y are the numbers of users and items respectively, and D is the size of the feature vector for each user or item.

As illustrated in Figure 2(b) , when the relationship network is abstracted as a bipartite graph, users and items constitute the two disjoint subsets of vertices separately, and the weight on each edge $R(p, q)$ represents an existing rating given from a user p to an item q. Based on this bipartite graph, MLDM algorithms (e.g., Alternating Least Squares [24]) generally associate each vertex with a vector of D elements, and then perform element-wise operations on these vectors, to estimate the weight of every absent edge between these two vertex-subsets.

2.3 Observations and Opportunities

Although bipartite graphs have become increasingly popular, existing graph partitioning frameworks tend to generate suboptimal partitioning that incurs high communication cost and workload imbalance during graph computation, because they fail to fully exploit some potentially beneficial features of bipartite graphs. As examples shown in Table 1, the average replication factors achieved by the two widely-used graph partitioning frameworks Hybrid-cut [4] and Bi-cut [5] are 3.06 and 3.28 respectively, which are relatively high for an 8-node cluster, considering that the replication factor is bounded by the number of participating nodes. Although the novel 3D-partitioner [23] attains a lower average replication factor, it suffers from serious workload imbalance during computation. To better understand this problem, we investigate many real-world bipartite graphs and MLDM algorithms, and obtain the following three key observations:

Observation 1: *The vertex value in many MLDM algorithms is a divisible vector of multiple elements.* For traditional graph algorithms (e.g., PageRank), each vertex is associated with an indivisible value, such as an integer. As a result, the value of a vertex in its entirety must be assigned to a single node, referred to as the *horizontal partitioning* that distributes vertices over all nodes. On the contrary, the value of each vertex in MLDM algorithms is generally a vector consisting of multiple data elements. For example, the authors in [23] associate each vertex with a vector of up to 128 elements, and the users of PowerGraph [9] can configure each vertex value



(a) Author Degree Distribution (b) Publication Degree Distribution

Figure 3: Both two vertex-subsets in the graph DBLP exhibit power-law degree distribution.

as a vector of thousands of elements. This introduces a new *vertical partitioning* that divides each vertex vector into multiple subvectors, each of which comprises a portion of the elements. Afterwards, these subvectors, rather than an entire vector, are distributed over multiple nodes via the horizontal partitioning, possibly reducing total communication cost.

Observation 2: *The sizes of two vertex-subsets in a bipartite graph can be highly lopsided.* It is not surprising that, the number of vertices in a subset is much larger than that in the other one. As an example, for Netflix shown in Table 1, the number of users is about 27x that of movies. As another concrete example, there are 2,780 words in the English Wikipedia dataset [2], but the number of articles is 273,959, leading to a skew ratio of up to 98.5x. Such imbalanced numbers of vertices heighten the necessity of processing these two vertex-subsets with different priorities.

Observation 3: *Even within a vertex-subset, the vertices usually exhibit power-law degree distribution [9].* This means that within each subset, a very small number of vertices connect much more edges than most other vertices. It is common in daily life that, the bestsellers are much more widely purchased and commented by consumers than other less appealing goods. Take DBLP in Table 1 as an example, the vertices in both its two subsets exhibit power-law degree distribution, as shown in Figure 3. It is well-known that the skewed degree distribution hurts the performance of graph computation, by introducing stragglers into concurrent vertex updates [22]. Therefore, it is essential to distinguish the vertices of different degrees, and assign them with differential strategies.

In summary, the three state-of-the-art partitioning frameworks, Hybrid-cut, Bi-cut, and 3D-partitioner, implicitly exploit one of the above unique features of bipartite graphs respectively. To comprehensively exploit these features, we are motivated to design a new partitioning framework, GraBi, to improve the performance of MLDM algorithms on bipartite graphs, which will be elaborated in the next section.

3 TWO-STAGE PARTITIONING

This section presents GraBi, a communication-efficient and workload-balanced partitioning framework that partitions a bipartite graph first vertically and then horizontally. We summarize the overall partitioning workflow of GraBi in Algorithm 1, and elaborate on these two stages in the following subsections.

Algorithm 1: Two-stage Partitioning Framework

Input: A bipartite graph G, Number of elements D, Number of nodes N, A set of hash functions F, Two thresholds R_{total} and R_{per_vertex}.
Output: A global table of adopted hash functions T, Mapping of vertices to nodes M.

```
/** STAGE 1: Vertex-vector Chunking **/
Calculate the number of layers L ← GCD(D, N)
Divide G into L layers;
/** STAGE 2: Vertex-chunk Assignment **/
foreach layer j ∈ [0, L − 1] do
    Get the larger subset U_j;
    Get the smaller subset V_j;
    foreach vertex-chunk u ∈ U_j do
        while u.edges > 0 do
            Get the assignment of current hash function
              node_k ← f_current(u.id, N/L);
            if node_k.totalEdges > R_total then
                Replace current function with next one
                  f_current ← f_next ∈ F;
                continue;
            end
            Assign the first R_per_vertex edges of u to node_k;
            Record < u.id, f_current.id > in the local table
              on node_k;
            Remove the assigned edges from u
              u.edges ← (u.edges − R_per_vertex);
            Replace current function with next one
              f_current ← f_next ∈ F;
        end
        Record the id and master location of u in M;
    end
    foreach vertex-chunk v ∈ V_j do
        Create replicas of v according to its edges;
        Choose one replica as the master;
        Record the id and master location of v in M;
    end
    Aggregate all local tables into a global table T;
    Compress the global table T ← compress(T);
    Send each item in T to correspoding sub-chunk;
end
```

3.1 (Vertical) Vertex-vector Chunking

As proposed in *Observation 1*, MLDM algorithms generally associate each vertex in a bipartite graph with a multi-element vector, as the sample graph shown in Figure 4(a) wherein the three vectored vertices are represented as cylinders. Figure 4(b) illustrates how this sample graph is partitioned over 3 computing nodes of a cluster by traditional horizontal partitioning. Specifically, as the partitioning strategy adopts vertex-cut that evenly assigns all edges over nodes, a mod-based hash function is applied to each edge for determining its location (e.g., by hashing the sum of vertex IDs of its two endpoints). As a result, each node is assigned with one edge and entire vectors

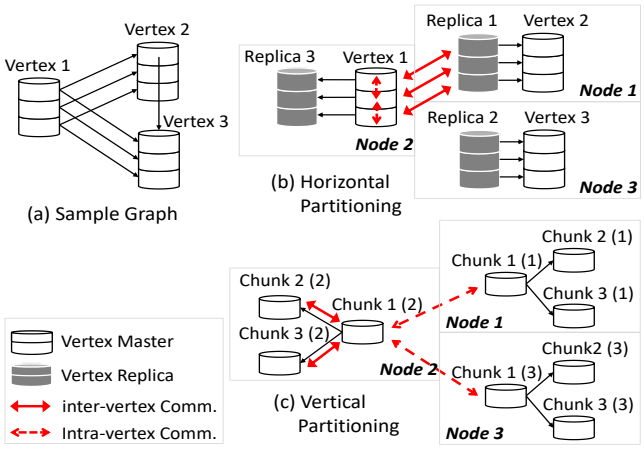


Figure 4: Horizontal versus vertical partitioning.

of the two endpoints. During subsequent computation, each vertex master synchronizes with its replicas on different nodes, referred to as *inter-vertex communication*.

Traditional horizontal partitioning implicitly assumes that the vertex value is indivisible, which is not true for MLDM algorithms where each vertex is associated with a vector of multiple elements. To explore the potential benefit brought by the vertex vectorization in MLDM algorithms, we present the first stage of GraBi, Vertex-vector Chunking (line 2-3 in Algorithm 1) that divides each vertex vector into multiple subvectors, which we refer to as *vertex-chunks*. Accordingly, the bipartite graph is partitioned into multiple layers, and each layer consists of a certain vertex-chunk each from all vertices. Figure 4(c) shows how the sample graph is partitioned by the new vertical partitioning, in which *Chunk i (j)* denotes the j-th chunk of Vertex i. To be specific, the sample graph is divided into 3 layers that are each assigned to a different node. For example, Node 1 holds the first chunks of all vertices, Node 2 holds the second chunks of all vertices, and so on. By doing so, updating elements in a vertex vector can be concurrently executed on all the nodes that the vertex-chunks of the vertex are assigned to. More importantly, due to the element-wise operations performed by MLDM algorithms, a given j-th chunk only communicates with the j-th chunks of other vertices. Since each node holds the entire layer comprising all interrelated chunks, the inter-vertex communication, marked by the solid arrows in Figure 4(c), is localized on a single node. However, the vertical partitioning is dominated by another kind of synchronization, *intra-vertex communication*, which happens between any pair of adjacent chunks in a vectored vertex, as indicated by the dashed arrows in Figure 4(c).

Generally speaking, partitioning a bipartite graph inevitably introduces two types of communication, namely, the inter- and intra-vertex communication. Intuitively, trading off between these two types of communication can be realized by adjusting the number of layers, denoted as L. Particularly, if L equals 1, it is a totally horizontal partitioning, in which the inter-vertex communication dominates. If L equals the number of nodes N, it is a totally vertical partitioning, in which the intra-vertex communication dominates. Therefore, the value of L should be chosen between 1 and N. Moreover, L also influences the workload distribution, by deciding the number of elements in each vertex-chunk and the number of nodes for each layer. Despite the importance of L, state-of-the-art 3D-partitioner [23] that also adopts the vertical partitioning lacks a systematic method to decide the value of L before computation. To address this issue, we present an empirical method to calculate an appropriate L value, which can guarantee an adequate level of performance for most MLDM applications.

Assuming that there are N nodes in the cluster and D elements in each vertex vector, a balanced vertical partitioning is one in which all vertex-chunks in a layer contain an equal number of elements and are evenly distributed over nodes. To this end, L is set as the greatest common divisor (GCD) of D and N, such that each vertex-chunk consists of D/L elements, and each layer is assigned to N/L nodes, wherein the layer is horizontally partitioned among the N/L nodes as elaborated in the next subsection. However, if the GCD equals 1, we prefer to balance the number of nodes each layer is assigned to, and thus set L as the maximum number that is divisible to N. As a result, the j-th chunk of a vertex contains the elements in the range $[firstElement(j), firstElement(j+1))$, where $j = 0, ..., L-1$. Particularly, $firstElement(j)$ is the sequence number of the first element in the j-th chunk, which is an integer between 0 and $D-1$, and is calculated as follows:

$$firstElement(j) = j \times (\lfloor D/L \rfloor) + \min(j, D\%L) \quad (1)$$

Furthermore, it may be the case that the vectors contain different numbers of elements. Fortunately, some methods have been proposed to uniformize these vectors, for example, extending the smaller vectors with zeros when adding these vectors.

We evaluate this method of determining the L value in Section 4.6. To summarize, the Vertex-vector Chunking is simple element-grouping for every vectored vertex.

3.2 (Horizontal) Vertex-chunk Assignment

The first stage of GraBi vertically divides a bipartite graph into multiple layers, with each layer being allocated to N/L nodes. We next present the second stage, *Vertex-chunk Assignment*, to assign vertex-chunks to nodes within each layer.

Subset-aware Prioritization: Traditional partitioning strategies are oblivious to the two vertex-subsets in a bipartite graph, and thus assign vertex-chunks from different subsets without differentiation. Figure 5(b) shows an example of the traditional partitioning strategy, wherein the location of each edge is decided by applying a mod-based hash function to the sum of vertex IDs of its two endpoints, without distinguishing the endpoints from different subsets. As it shows, totally 15 replicas are created, potentially incurring high network and memory pressure during subsequent graph computation.

In a bipartite graph, every edge connects a pair of vertices each from a different subset. This implies that distributing only vertices from the same subset over nodes would not generate any replicas, because there are no edges connecting them. Therefore, if vertices in a subset are first assigned, then the created replicas must come from the other subset. Moreover, as mentioned in *Observation 2*, the sizes of two vertex-subsets in a bipartite graph can be highly lopsided. Based on this analysis, we present *subset-aware prioritization* (line

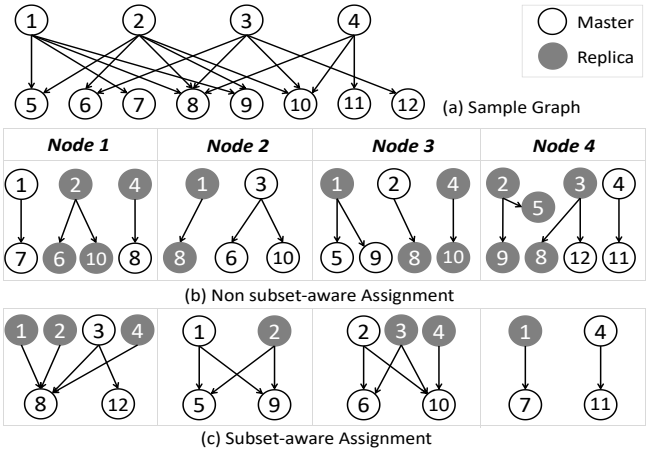


Figure 5: Non subset-aware versus subset-aware assignment.

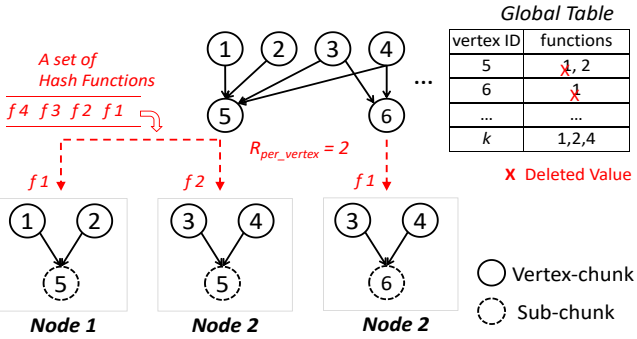


Figure 6: Sub-chunk assignment with a set of functions.

6-7 in Algorithms 1), which always assigns vertex-chunks from the larger subset first within each layer. As a result, the replicas are created only for vertex-chunks in the smaller subset, setting a much lower upper limit to the number of replicas. An example of this subset-aware partitioning strategy is shown in Figure 5(c), wherein the larger subset comprising the vertex-chunks ⑤ through ⑫ is granted a higher priority. Accordingly, each edge is assigned to a node by applying the hash function only to the vertex ID of endpoint in the larger subset. As a result, the number of replicas is decreased from 15 to 7, significantly alleviating the network and memory pressure. More importantly, since all edges are assigned according to endpoints in the larger subset, accessing locality is preserved for a majority of vertex-chunks.

Nevertheless, *subset-aware prioritization* only gives a preliminary assigning order to the two vertex-subsets. The specific mapping of vertex-chunks in a subset to nodes must be further determined. Additionally, as mentioned in *Observation 3*, even within a subset, the vertices often exhibit power-law degree distribution. In other words, a few high-degree vertices connect much more edges than most other vertices. For example, in Figure 5(c), the vertex-chunk ⑧ connects 4 edges, more than that of the other vertex-chunks in the same subset (1-3 edges). As a result, this high-degree vertex-chunk may consume more time than all other vertex-chunks in each update iteration, incurring workload imbalance during graph computation.

Early solutions [9, 10] rely on vertex-cut that splits edges of each high-degree vertex over multiple nodes, to distribute its high workload. However, it also inevitably splits edges of low-degree vertices, incurring many unnecessary replicas and high communication cost. Therefore, Hybrid-cut [4] is proposed to distinguish the vertices of different degrees. To be specific, it applies vertex-cut to high-degree vertices and edge-cut to low-degree vertices, to achieve workload balance and low communication cost simultaneously. Nevertheless, Hybrid-cut has to pre-count the degree of every vertex to decide whether it is high-degree or low-degree, causing large preprocessing overhead that may not be amortized by subsequent computation [13]. To address these issues, we present two sequential steps, *bounded chunk-cutting* and *sub-chunk assignment*, to assign vertex-chunks in the larger subset to nodes.

Bounded Chunk-cutting: The rationale behind this step is to constrain the number of edges that a vertex-chunk can allocate to each node. For this purpose, we propose two key parameters. The first parameter is the total number of edges that can be received by each node, denoted as R_{total}, and is calculated as follows:

$$R_{total} = (1 + \frac{1}{N/L}) \times (\frac{|E|}{N/L}) \tag{2}$$

Given that many global properties of the input graph (e.g., $|E|$, $|\mathbb{U}|$ and $|\mathbb{V}|$) are usually provided in advance, R_{total} is calculated before computation. Intuitively, R_{total} ensures that all edges in a layer are evenly distributed over the N/L nodes it is assigned to.

The second parameter is the maximum number of edges that each node can receive from a given vertex-chunk, which is denoted as R_{per_vertex} and calculated as follows:

$$R_{per\_vertex} = \alpha \times (\frac{|E|}{|\mathbb{U}|}) \tag{3}$$

Wherein, α is an amplification factor that is greater than 1, and can be determined empirically according to the skewness of power-law degree distribution. We evaluate the impact of α in Section 4.6. The second term in the above expression, $|E/\mathbb{U}|$, represents the average degree of vertices in the larger subset $\mathbb{U}$, given that all edges are assigned according to the endpoints in $\mathbb{U}$. Therefore, a vertex-chunk will be decomposed into multiple sub-chunks, if its degree is greater than R_{per_vertex}.

Sub-chunk Assignment: The inefficiency of Hybrid-cut essentially stems from employing only a single hash function, which tends to gather all edges of a vertex onto a single node. If the partitioning strategy has a set of hash functions, it can distribute edges of a high-degree vertex using multiple hash functions, and gather edges of a low-degree vertex using one hash function. For this end, this step uses a set of hash functions to assign the generated sub-chunks to nodes. An example is shown in Figure 6, as R_{per_vertex} equals 2, the high-degree vertex-chunk ⑤ is cut into 2 sub-chunks, each of which connects 2 edges. In contrast, the low-degree vertex-chunk ⑥ has only one sub-chunk. The first sub-chunks of ⑤ and ⑥ are respectively assigned to Node 1 and Node 2, by applying the first hash function $f1$ to their vertex IDs. Since the vertex-chunk ⑤ still has a sub-chunk unassigned, the next hash function $f2$ is used to assign the remaining sub-chunk to Node 2. As a result, the

two sub-chunks of ⑤ are assigned over different nodes. In essence, all hash functions will be sequentially used, until there are no unassigned sub-chunks. Moreover, for a certain vertex-chunk, some hash functions will be skipped, if the total number of edges on a node has already reached R_{total} (line 11-14). As a result, the hash functions adopted by this vertex-chunk are nonconsecutive.

Intuitively, the number of hash functions should equal N/L, such that the edges of an extremely high-degree vertex-chunk can be split onto all nodes in the layer. Moreover, to distribute edges of a vertex-chunk, any two hash functions in a set must map a certain vertex ID onto different nodes. This can be realized by applying the same modulus operator on the vertex ID and then adding an incremental offset. As a result, the set of hash functions works in a round-robin fashion. That is, two adjacent hash functions in the set map a vertex ID onto two nodes of contiguous physical IDs.

To locate all sub-chunks of a given vertex-chunk, the hash functions adopted by the vertex-chunk should be identifiable. To this end, each node maintains a local table to store its received sub-chunks and corresponding hash functions in the form of key-value pairs, where the key is a vertex ID and the value is the sequence number of the adopted hash function (line 16). After graph partitioning, all local tables are aggregated into a global table that summarizes all hash functions adopted by each vertex-chunk (line 27).

The global table is generally of moderate size, because only a tiny fraction of vertex-chunks in a power-law graph are high-degree and thus adopt multiple hash functions. We further decrease the table size by classifying the items in the table. First, for the item with a value of 1 (e.g., the vertex-chunk ⑥ in Figure 6), it is removed from the table, because it only adopts the first hash function that is the default one. Second, for the item with a value of consecutive numbers (e.g., the vertex-chunk ⑤), its value is reduced to the maximum sequence number of adopted hash functions, which can be easily recognized. Last, for the item with a value of nonconsecutive numbers (e.g., the vertex-chunk ⓚ), its value is stored integrally. In this way, the size of the global table is greatly decreased. We have evaluated the effect of this compression technique, and find that it can save the table size by an average of 90.4%, for those five graphs listed in Table 1. After the compression, the sub-chunk assigned by the first hash function receives a copy of corresponding item from the global table, such that it can use the received hash functions to locate all other sub-chunks for communication. Finally, the master and replicas of every vertex-chunk in the smaller subset are created, according to its adjacent edges.

In summary, the *bounded chunk-cutting* and the *sub-chunk assignment* work together to evenly distribute edges of each high-degree vertex-chunk over multiple nodes, and gather edges of each low-degree vertex-chunk onto the same node, avoiding the overhead of pre-counting degrees in Hybrid-cut.

3.3 Summary of GraBi

GraBi partitions a bipartite graph first vertically and then horizontally, for different goals. To be concrete, the *Vertex-vector Chunking* first vertically partitions the entire graph into multiple layers, to strike an appropriate tradeoff between inter- and intra-vertex communication. In the *Vertex-chunk Assignment*, the larger vertex-subset within each layer is first assigned with a higher priority, to limit the number of replicas. More specifically, the *bounded chunk-cutting* decomposes each vertex-chunk into one or more sub-chunks with an upper-bounded edge-count, and then the *sub-chunk assignment* allocates sub-chunks over nodes using a set of hash functions, to achieve workload balance with negligible overhead. By this way, the bipartite graph is partitioned in a structure-aware and fine-grained way.

In addition, GraBi takes each vertex as input, requiring no prior knowledge or repartitioning stage. To be specific, it divides each vectored vertex into multiple vertex-chunks, and then assigns corresponding sub-chunks of the vertex-chunks over nodes using a series of hash functions. Therefore, GraBi is lightweight. Finally, GraBi is not confined to specific algorithms, and thus is generalizable to most MLDM applications.

4 EVALUATION

In this section, we conduct a comprehensive evaluation on GraBi, against three existing partitioning frameworks. First, we evaluate the overall performance of GraBi, and break down the performance into graph partitioning phase and computation phase respectively. Next, we assess the scalability of GraBi by increasing the graph size. Finally, we study the impact of two important parameters in GraBi, the number of layers L and the amplification factor α.

4.1 Experimental Setup

GraBi is implemented as a separate partitioning framework in PowerLyra [4], a distributed in-memory graph processing system. The experiments are conducted on an in-house 8-node cluster. Each node has one Intel Xeon E5-2650 processor (8 cores) and 16GB DRAM, all nodes are connected via 1Gb Ethernet. We use a collection of real-world bipartite graphs collected from the Konect Network Dataset [1]. Table 1 shows the basic global properties of these graphs. We select three representative MLDM algorithms, Alternating Least Squares (ALS) [24], Stochastic Gradient Descent (SGD) [21], and Non-negative Matrix Factorization (NMF) [12], as the application drivers working on these graphs to assess the partitioning quality. The number of elements in each vector D is set as 20 for all the three algorithms. Finally, we set the default value of L and α in GraBi as 4 and 2 respectively, and compare GraBi against three state-of-the-art partitioning frameworks:

Hybrid-cut [4] is a partitioning framework integrated in PowerLyra. It combines edge-cut and vertex-cut to assign the vertices following power-law degree distribution. Hybrid-cut has two versions, Random Hybrid-cut and Heuristic Hybrid-cut (i.e., Ginger). As GraBi adopts hash functions to assign vertices and edges, we choose Random Hybrid-cut as the counterpart for fair comparison. Besides, we have tested many thresholds used by Hybrid-cut to identify high-degree vertices, and use the best-performing one for each graph.

Bi-cut [5] is another built-in partitioning framework of PowerLyra. It partitions a bipartite graph by differentiating the vertices from different subsets. Similar to Hybrid-cut, Bi-cut also has two versions, Random Bi-cut and Heuristic Bi-cut (i.e., Aweto). We choose Random Bi-cut as the counterpart for fair comparison.

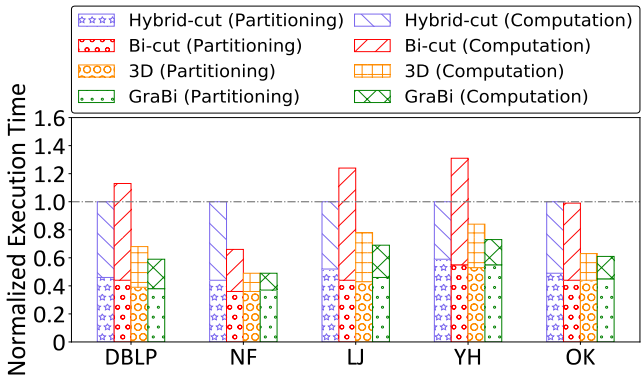
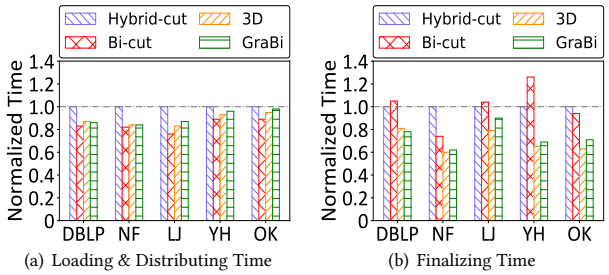


Figure 7: The end-to-end execution time on different graphs, achieved by the four partitioning frameworks. The execution time of each framework includes both the graph partitioning time and computation time, and is normalized to that of Hybrid-cut.



(a) Loading & Distributing Time (b) Finalizing Time

Figure 8: The partitioning time on different graphs, achieved by the four partitioning frameworks. The partitioning time is composed of the loading & distributing time, and the finalizing time. Each type of time is normalized to that of Hybrid-cut.

3D-partitioner [23] is a partitioning framework proposed by a novel distributed graph processing system CUBE. It advocates an appropriate vertical partitioning of graphs. Specifically, we also set the number of layers L in 3D-partitioner as 4, and employ Random Bi-cut as the horizontal partitioning strategy within each layer.

4.2 Overall Performance

We first compare the end-to-end execution time, including both the graph partitioning time and computation time, achieved by the four partitioning frameworks. As shown in Figure 7, GraBi improves the execution time by an average of 1.65x over Hybrid-cut, 1.70x over Bi-cut, and 1.09x over 3D-partitioner respectively. To be specific, GraBi surpasses Hybrid-cut and Bi-cut in both the partitioning and computation phases. Besides, GraBi outperforms 3D-partitioner in the computation phase, but slightly underperforms in the partitioning phase.

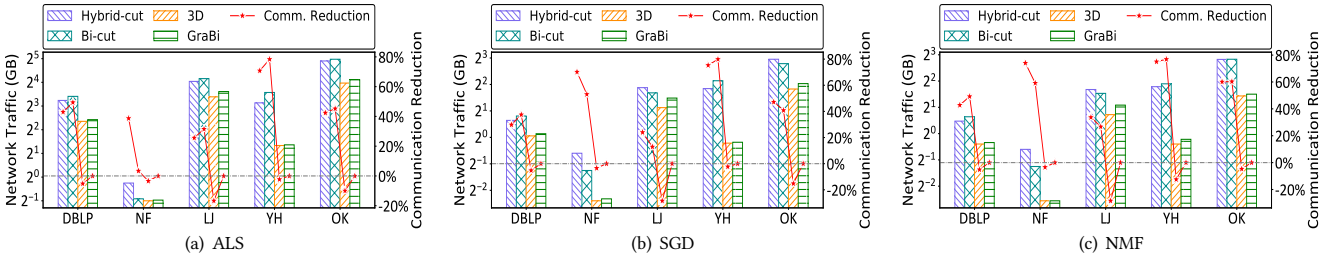
In essence, as 3D-partitioner employs Bi-cut to assign vertex-chunks over nodes within each layer, it exploits the vertex vectorization in MLDM algorithms (i.e., *Observation 1*) and the lopsided subset sizes in bipartite graphs (i.e., *Observation 2*). However, it does not consider the skewed degree distribution inherent in many graphs (i.e., *Observation 3*), and thus suffers from workload imbalance during computation. In contrast, GraBi exploits the three key observations simultaneously, and adopts the *Vertex-chunk Assignment* within each layer to effectively alleviate the workload imbalance. Therefore, GraBi runs faster than 3D-partitioner on power-law graphs in the computation phase, such as LiveJournal and Yahoo. Nevertheless, for graphs where the degree distribution is not very skewed, such as Netflix, the execution time of GraBi is comparable or even longer than that of 3D-partitioner. To better understand the performance advantage of GraBi over each of the other partitioning frameworks, we break down the performance into the graph partitioning phase and computation phase respectively, as presented in the next two subsections.

Last but not the least, a partitioned graph is generally repeatedly used in real-world scenarios, and the partitioning time can be amortized by multiple executions on the same graph. Thus, the performance advantage of GraBi can be much more pronounced if a partitioned graph is repeatedly analyzed by many algorithms.

4.3 Partitioning Phase

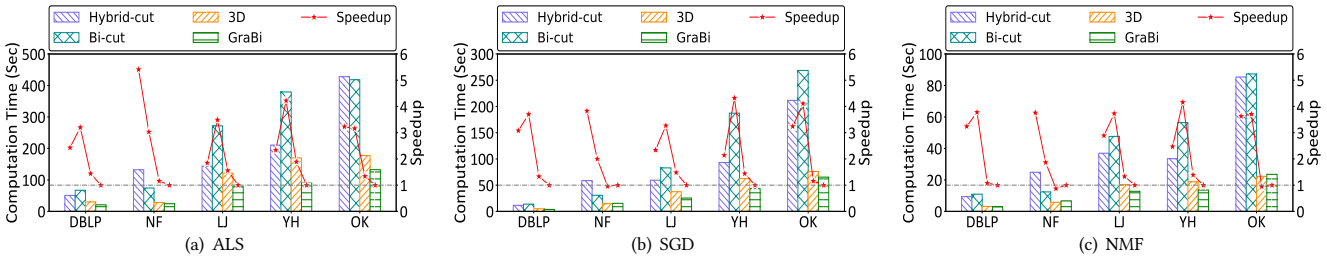
The partitioning phase includes loading a graph, distributing the graph, and finalizing subgraphs on all nodes. We evaluate the performance in the partitioning phase, by measuring the replication factor and partitioning time respectively. Table 1 shows the replication factors of different graphs. Overall, the average replication factors over the five graphs achieved by Hybrid-cut, Bi-cut, 3D-partitioner, and GraBi are 3.06, 3.28, 1.36, and 1.45 respectively. Specifically, Hybrid-cut performs better on bipartite graphs with power-law degree distribution, for example, its replication factor on Yahoo is 1.32x lower than that of Bi-cut. On the other hand, Bi-cut works better on bipartite graphs with lopsided subset sizes, for example, its replication factor on Netflix is 1.57x lower than that of Hybrid-cut. However, both Hybrid-cut and Bi-cut are oblivious to the divisible vertex vector in MLDM algorithms, and thus perform no vertical partitioning. As stated earlier, the replication factor is bounded by the number of participating nodes. Due to the vertical partitioning, 3D-partitioner and GraBi assign each layer to 2 nodes (i.e., $N/L = 2$), setting the upper limit of replication factors as only 2. For example, the replication factors on Netflix achieved by 3D-partitioner and GraBi are 1.16 and 1.20 respectively. Note that, the average replication factor of GraBi is slightly higher than that of 3D-partitioner. This is because GraBi considers power-law degree distribution, and thus creates more replicas than 3D-partitioner to realize workload balance in the computation phase.

We further break down the partitioning time into two separate parts, namely the loading & distributing time and the finalizing time. As shown in Figure 8(a), Hybrid-cut has the longest loading & distributing time on all the five graphs, due to its high overhead of pre-counting vertex degrees in the graph loading step. Since 3D-partitioner employs Bi-cut as the partitioning strategy within each layer, the loading & distributing time of 3D-partitioner is always longer than that of Bi-cut. Furthermore, the average loading & distributing time of GraBi is slightly longer than that of 3D-partitioner, especially on larger graphs (e.g., Yahoo and Orkut), due to the overhead of switching between multiple hash functions to



(a) ALS (b) SGD (c) NMF

Figure 9: The network traffic of different MLDM algorithms, achieved by the four partitioning frameworks. The communication reduction is attained by GraBi over each of the other partitioning frameworks.



(a) ALS (b) SGD (c) NMF

Figure 10: The computation time of different MLDM algorithms, achieved by the four partitioning frameworks. The speedup is attained by GraBi over each of the other partitioning frameworks.

assign vertices and edges. As shown in Figure 8(b), the finalizing time of each partitioning framework is almost proportional to the corresponding replication factor. For example, on Yahoo, Bi-cut has the highest replication factor of 4.43, and its finalizing time is also the longest among the four partitioning frameworks. Besides, the finalizing time of GraBi is also slightly longer than that of 3D partitioner, due to the overhead of constructing global tables.

4.4 Computation Phase

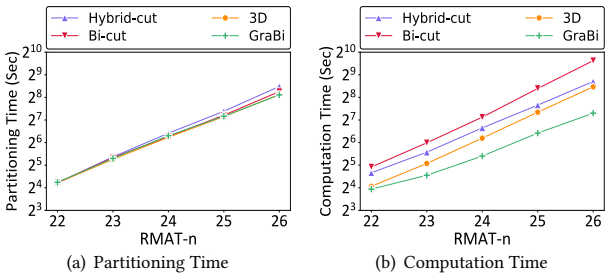
We evaluate the performance in the computation phase, by measuring the network traffic and computation time respectively. Figure 9 shows the network traffic of each MLDM algorithm on different graphs. Overall, ALS is the algorithm with the heaviest network workload, for example, when Hybrid-cut and Bi-cut execute ALS on Orkut, the network traffic reaches up to 29.9GB and 31.4GB respectively. Therefore, the network bandwidth can be a performance bottleneck when executing ALS in these two frameworks. To make the comparison more straightforward, we present the communication reduction achieved by GraBi over each of the other frameworks, which is calculated as $(\frac{Traffic_{Other} - Traffic_{GraBi}}{Traffic_{Other}})$. As Figure 9 shows, GraBi reduces the network traffic in Hybrid-cut and Bi-cut by an average of 45.3% and 48.7% respectively. However, GraBi incurs more network traffic than 3D-partitioner by average 11.5%, because of its slightly higher replication factors.

Figure 10 shows the computation time of each MLDM algorithm on different graphs. Overall, Grabi outstrips Hybrid-cut, Bi-cut, and 3D-partitioner by an average of 3.12x (up to 5.41x), 3.41x (up to 4.32x), and 1.30x (up to 1.89x) respectively. Even though both 3D-partitioner and GraBi achieve shorter computation time because of much lower replication factors and communication cost, 3D-partitioner is oblivious to the skewed degree distribution inherent in many graphs, and thus suffers from workload imbalance in the computation phase. This imbalance becomes much more serious when executing ALS, which is a computation-intensive algorithm. To overcome this problem, GraBi relies on the *bounded chunk-cutting* to decompose each high-degree vertex-chunk into multiple sub-chunks, and deploys some necessary replicas to perform concurrent vertex updates. To be specific, when running ALS on Yahoo where the vertices exhibit power-law degree distribution, GraBi outperforms 3D-partitioner in the maximum speedup by 1.89x, appropriately trading off between minimizing replicas and balancing workload.

4.5 Scalability

We next assess the scalability of GraBi by increasing the graph size. Particularly, since there are no tailored tools for generating bipartite graphs, we use a general graph generator R-MAT [3] to produce a series of power-law graphs in various sizes, where the graph RMAT-n contains about 2^n vertices and 2^{n+4} edges. Afterwards, we randomly classify the vertices in each graph into two disjoint subsets, and delete all the edges connecting vertices from the same subset. By this way, we obtain a collection of synthetic bipartite graphs in different sizes. Finally, we measure the partitioning time and computation time respectively of the four partitioning frameworks, when running ALS on each synthetic bipartite graph. The other two algorithms have similar trends as ALS.

As shown in Figure 11(a), all the four partitioning frameworks exhibit decent scalability in terms of the partitioning time, which increases linearly with the graph size. However, as the graph size increases to a certain extent (e.g., on RMAT-26), Hybrid-cut suffers from the overhead of pre-counting vertex degrees, and Bi-cut has



(a) Partitioning Time (b) Computation Time

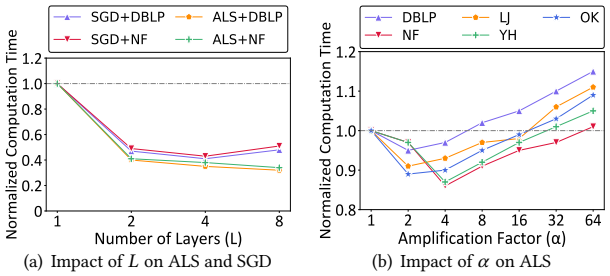
Figure 11: The partitioning time and computation time respectively achieved by the four partitioning frameworks, when running ALS on synthetic graphs in different sizes.

to create much more replicas in the finalizing step, incurring obviously longer partitioning time than 3D-partitioner and GraBi. As shown in Figure 11(b), in terms of the computation time, Hybrid-cut and GraBi exhibit better scalability than the other two frameworks, since they consider the power-law degree distribution inherent in the synthetic graphs. Moreover, given that the vertices in each graph are randomly separated into two disjoint subsets, the two vertex-subsets have similar sizes, and thus Bi-cut consistently underperforms the other frameworks on all synthetic graphs.

4.6 Impact of Parameters

To better understand the aforementioned tradeoff, we examine the impacts of two key parameters in GraBi, the number of layers L and amplification factor α, on the computation phase respectively. The L value influences overall communication cost by trading off between inter- and intra-vertex communication. The α value affects workload distribution by constraining the number of edges that a vertex-chunk can allocate to each node (i.e., R_{per_vertex}). Note that, we omit the partitioning phase, because it is hardly affected by these two parameters.

Figure 12(a) shows the impact of L on the computation time of ALS and SGD respectively, while NMF has a similar trend as SGD. Overall, vertically dividing a graph into multiple layers can reduce the total communication cost, and therefore decreases the computation time. For example, the execution of ALS on DBLP is accelerated by 3.13x when L increases from 1 to 8. In addition, with increasing L, the computation time of SGD first decreases rapidly and then increases, while the computation time of ALS still drops



(a) Impact of L on ALS and SGD (b) Impact of α on ALS

Figure 12: The impact of the number of layers (L) and amplification factor (α) respectively on the computation phase.

but at a lower speed. This finding echoes the communication-cost equations of ALS and SGD, as proposed in [23]. Although ALS and SGD behave best at different values of L, our empirical method of setting L as GCD of N and D (e.g., $L = 4$ in this evaluation) delivers adequate performance for both these two algorithms.

The impact of α on the computation time of ALS is shown in Figure 12(b), and the other two algorithms have similar trends. In general, the best-performing α value for each graph largely depends on the skewness of power-law degree distribution. For example, as the percentage of high-degree vertices in DBLP is lower than that in other graphs, it prefers a lower α value to gather edges for most low-degree vertices. Despite the impact of α, its influence is moderate and stable within a wide value range.

5 RELATED WORKS

Edge-cut versus Vertex-cut: Early partitioning frameworks [15, 16] adopt edge-cut to evenly assign vertices over nodes, but suffer from high communication cost and imbalanced computation on prevalent power-law graphs. Therefore, many partitioning frameworks [9, 10] resort to vertex-cut to equally distribute edges over nodes, but inevitably produce many unnecessary replicas for low-degree vertices. To embrace the best of two worlds, some frameworks develop hybrid approaches that distinguish vertices of different degrees. One of the most impressive is PowerLyra [4] that has been introduced earlier. Besides, Agent-Graph [22] is a vertex-cut variant in the context of communication pattern, but uses edge-cut to partition the residual graph where all high-degree vertices are filtered. Nevertheless, these hybrid approaches usually require high preprocessing overhead or complicated implementation. By comparison, GraBi employs a set of hash functions to uniformly assign vertices, avoiding the costly overhead in distinguishing vertices of different degrees.

Offline versus Streaming: Offline partitioning frameworks usually leverage the knowledge of whole graph, to refine the partitioning quality in multiple rounds. For example, METIS [11] proposes a coarsening method to cut a graph into smaller pieces, and a refining method to reconstruct the original graph from these pieces. However, the cost of refinement often outweighs the benefit of high-quality partitions. In contrast, streaming partitioning frameworks determine the assignment of vertices on-the-fly, by sequentially scanning the whole graph in only one pass. For example, ADWISE [17] is a window-based streaming partitioning framework, which improves the partitioning quality by always choosing the best edge from a set of edges for assignment to a partition. IOGP [6] generates streaming partitions in three sequential stages, by taking both the connectivity and degrees of vertices into consideration. As GraBi is also a streaming but hash-based partitioning framework, the heuristic methods in these works can be incorporated into GraBi to improve partitioning quality.

6 CONCLUSION

This paper presents GraBi, a two-stage partitioning framework that partitions a bipartite graph first vertically and then horizontally, to achieve efficient communication and balanced workload during computation. The first stage vertically divides every vectored vertex in a bipartite graph into multiple vertex-chunks. In the second

stage, vertex-chunks in the larger vertex-subset are decomposed into one or more sub-chunks with bounded edge-count. Then, these sub-chunks are uniformly assigned over nodes using a set of hash functions. Besides, GraBi is a lightweight partitioning framework for bipartite graphs, and generalizes to most MLDM applications.

ACKNOWLEDGMENTS

This work was supported in part by National key research and development program of China under Grant 2018YFA0701805 and Grant 2018YFA0701804, in part by the Creative Research Group Project of NSFC No. 61821003, NSFC No. 61872156, and in part by Alibaba Group through Alibaba Innovative Research (AIR) Program.

REFERENCES

[1] [n.d.]. Konect Network Dataset. http://konect.uni-koblenz.de/.
[2] [n.d.]. Wikimedia Downloads. https://dumps.wikimedia.org/.
[3] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SDM*.
[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: differentiated graph computation and partitioning on skewed graphs. In *Eurosys*.
[5] Rong Chen, Jiaxin Shi, Binyu Zang, and Haibing Guan. 2014. Bipartite-oriented distributed graph partitioning for big learning. In *APSys*.
[6] Dong Dai, Wei Zhang, and Yong Chen. 2017. IOGP: An Incremental Online Graph Partitioning Algorithm for Distributed Graph Databases. In *HPDC*.
[7] Inderjit S. Dhillon. 2001. Co-clustering documents and words using bipartite spectral graph partitioning. In *SIGKDD*.
[8] Bin Gao, Tie-Yan Liu, Xin Zheng, QianSheng Cheng, and Wei-Ying Ma. 2005. Consistent bipartite graph co-partitioning for star-structured high-order heterogeneous data co-clustering. In *SIGKDD*.
[9] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *OSDI*.
[10] 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 *OSDI*.
[11] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Scientific Computing* 20, 1 (1998), 359–392.
[12] Daniel D. Lee and H. Sebastian Seung. 2000. Algorithms for Non-negative Matrix Factorization. In *NIPS*.
[13] Dongsheng Li, Chengfei Zhang, Jinyan Wang, Zhaoning Zhang, and Yiming Zhang. 2017. GraphA: Adaptive Partitioning for Natural Graphs. In *ICDCS*.
[14] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*.
[15] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. 2012. Distributed GraphLab: A Framework for Machine Learning in the Cloud. *PVLDB* 5, 8 (2012), 716–727.
[16] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*.
[17] Christian Mayer, Ruben Mayer, Muhammad Adnan Tariq, Heiko Geppert, Larissa Laich, Lukas Rieger, and Kurt Rothermel. 2018. ADWISE: Adaptive Window-Based Streaming Edge Partitioning for High-Speed Graph Processing. In *ICDCS*.
[18] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2 (2015), 25:1–25:39.
[19] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy J. Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *PVLDB* 9, 13 (2016), 1281–1292.
[20] Alexander J. Smola and Shravan M. Narayanamurthy. 2010. An Architecture for Parallel Topic Models. *PVLDB* 3, 1 (2010), 703–710.
[21] Gábor Takács, István Pilászy, Bottyán Németh, and Domonkos Tikk. 2009. Scalable Collaborative Filtering Approaches for Large Recommender Systems. *J. Mach. Learn. Res.* 10 (2009), 623–656.
[22] Jie Yan, Guangming Tan, and Ninghui Sun. 2015. Study on Partitioning Real-World Directed Graphs of Skewed Degree Distribution. In *ICPP*.
[23] Mingxing Zhang, Yongwei Wu, Kang Chen, Xuehai Qian, Xue Li, and Weimin Zheng. 2016. Exploring the Hidden Dimension in Graph Processing. In *OSDI*.
[24] Yunhong Zhou, Dennis M. Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM*.
[25] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*.