Journal of Data Intelligence, Vol. 2, No. 2 (2021) 000–000
© Rinton Press

B-GRAP: BALANCED GRAPH PARTITIONING ALGORITHM FOR LARGE GRAPHS FOR JOURNAL OF DATA INTELLIGENCE

ADNAN EL MOUSSAWI
adnan.el-moussawi@lisn.fr

NACERA BENNACER SEGHOUANI
nacera.seghouani@lisn.fr

FRANCESCA BUGIOTTI
francesca.bugiotti@lisn.fr

Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, Orsay, 91400, France

Received (received date)
Revised (revised date)

The definition of effective strategies for graph partitioning is a major challenge in distributed environments since an effective graph partitioning allows to considerably improve the performance of large graph data analytics computations.

In this paper, we propose a multi-objective and scalable Balanced GRAph Partitioning (B-GRAP) algorithm, based on Label Propagation (LP) approach, to produce balanced graph partitions. B-GRAP defines a new efficient initialization procedure and different objective functions to deal with either vertex or edge balance constraints while considering edge direction in graphs. B-GRAP is implemented of top of the open source distributed graph processing system Giraph.

The experiments are performed on various graphs with different structures and sizes (going up to 50.6M vertices and 1.9B edges) while varying the number of partitions. We evaluate B-GRAP using several quality measures and the computation time. The results show that B-GRAP (i) provides a good balance while reducing the cuts between the different computed partitions (ii) reduces the global computation time, compared to LP-based algorithms.

Keywords: graph partitioning, label propagation, vertex balance, edge balance, vertex-centric parallel computing, Giraph.
Communicated by: to be filled by the Editorial

1. Introduction

In recent years, large-scale graph analytics and mining have been widely used in various domains such as communication network, urban transportation, biological data, and social networks. In this context the efficient processing of large graphs becomes a new challenging task. Many research works focused on graph-based parallel computation algorithms in distributed systems [1, 2, 3]. The distribution of the workloads on several machines helps to reduce the overhead computation time. However, this distribution requires multiple exchanges of messages between the machines with a typically high cost.

Graph Partitioning (GP) algorithms have taken a lot of attention in recent decade as a key prerequisite for an efficient processing and many works focused on this problem [4, 5, 6, 7]. An efficient partitioning algorithm allows to minimize the total computation cost while a good balanced load makes better leverage of the entire system. The GP problem aims to divide the graph into a given number of partitions while minimizing the number of their inter-connecting edges (called *cuts*) and balancing their sizes w.r.t. the number of vertices or the number of edges. An *edge-balanced* GP divides the edges of the graph into nearly equal sized partitions. In contrast, a *vertex-balanced* GP divides the vertices of the graph into equisized partitions. Each objective has its own advantage. For a graph analysis task where the vertices exchange messages frequently, balancing the number of edges has more advantage. On the contrary, for tasks that need a few communications between vertices, the *vertex-balanced* GP is more beneficial.

Different GP partitioning approaches have been studied in the literature. Multilevel GP approach, called METIS [8] defined a well *vertex-balanced* partitioning algorithm which showed high quality in terms of *cuts*. But, it requires high resource usage and computation time, therefore it does not scale with large graphs. Streaming GP methods [9, 10] used an online graph partitioning, by considering *vertex-balance* or *edge-balance* constraints which reduces the overhead computation comparing to multilevel approaches. However, the results of such methods are of less quality and depend on the order of vertex or edge processing. Moreover the partitioning doesn't deal with the graph's changes. Recent works have taken advantage of the lightweight mechanism of Label Propagation (LP) approach to improve the partitioning process [11, 12, 13]. In [12, 14], the authors used LP to coarsen the graph in a multilevel partitioning approach while balancing the vertices. In [11] LP algorithm is extended to compute the entire partitioning basing on Giraph [1] programming model, while considering only *edge-balance* constraint. However, these approaches didn't pay attention to the initialization phase of LP which, as we will show, has a direct impact on the convergence and the performance of the partitioning.

This research work extends the article presented in [15]. In this paper we propose a new multi-objective and scalable *B*alanced *GRA*ph *P*artitioning algorithm (B-GRAP), based on LP, to produce balanced graph partitions. B-GRAP provides the following main contributions:

- An efficient partitioning initialization that helps to improve the propagation of labels and to reduce the computational overhead comparing to similar approaches.
- Two New scalable and parallel partitioning algorithms B-GRAP$_{\mathrm{VB}}$ and B-GRAP$_{\mathrm{EB}}$ that respectively address the vertex balance and the edge balance problems on both directed and undirected graphs.
- A full implementation of B-GRAP developed on top Giraph, a distributed open source graph processing system Giraph [1]. This allows us to take advantage from its parallel processing capabilities in order to effectively parallelize B-GRAP.
- The evaluation of B-GRAP, using different criteria (quality measures and execution time) on large heterogeneously structured real-worlds and synthetic graphs (going up to 50.6M vertices and 1.9B edges), shows good performance while scaling with the number of partitions and size of a graph.

This paper is structured as follows. In Section 2, we present a survey of some approaches related to graph partitioning problem. In Section 3, we present LP approach and B-GRAP main notations. In Section 4, we define B-GRAP, its initialization and the propagation objective functions for vertex or edge balance, then the measures we use to evaluate the quality of the partitioning are presented in Section 5. In Section 6, we highlight highlight the implementation of B-GRAP on top of Giraph. In Section 7, we detail the experimental study we conducted in order to evaluate B-GRAP. Finally, in Section 8, we draw our conclusions and future perspectives.

2. Related Work

During the last decade, research communities working on graph datasets have given a lot of interest to the definition of new strategies for large graph parallel computing and analytics in a distributed environment. This context opened up new challenges to define efficient graph partitioning algorithms [4, 7, 12, 16, 17]. One of the main challenges consists in defining methods that allow to balance the workload among the nodes of a distributed computing environment and to reduce, at the same time, the communication load over the network. In this section we discuss works on graph partitioning approaches classifying them in different categories taking into account the main characteristics of the algorithms they propose.

Spectral partitioning approaches Spectral approaches are based on the principles defined in [18] and rely on properties of the entire graph to compute the partition [19, 20]. The algorithms derive the partitions from the vector computation of the adjacency matrix of the graph. They have been widely applied in image segmentation context [21] where they show great performances and results thanks to the characteristics of input data [22].

Machine learning approaches During the last years graph data and graph data analysis have been widely considered together with machine learning techniques [23]. In the context of graph partitioning the literature focuses both on the selection of the best already-defined strategy for a specific input graph [24], and on the development of new partitioning approaches [25]. In this context we can cite GAP [26] framework that applied a deep learning approach to graph partitioning or Revolver [27] that applies reinforcement learning and label propagation to adaptively partition a graph.

Multilevel partitioning approaches Another common strategy in large graph partitioning is to use multilevel approaches [4]. The idea behind these works is to generate a first partition on the basis of a reduced view of the graph in which a vertex represents many vertices of the original graph. For example a triangle of three vertices can be reduced to one. The algorithms then expands the graph taking into account the whole initial graph. This family of approaches alternates three main phases: (i) coarsen the graph by collapsing adjacent vertices satisfying some matching criteria, (ii) partition the coarsened graph using any partitioning algorithm, (iii) the un-coarsening or refinement, which means generalizing the partition from last phase by mapping back the results to the original graph. METIS [8] is one of the multilevel graph partitioning algorithm family. This algorithm uses the Kernighan-Lin (KL) algorithm [28]. Starting with a random partition of two blocks (a random bisection), is known for its ability to produce partitioning with high quality w.r.t. the number of cuts, but with the disadvantage of the high computation time to obtain several intermediate results. Another widely well known multilevel graph partitionner is Scotch [29] which deals with the graph changes and does not require to start the partitioning from scratch, in contrast to METIS. The parallel version of both algorithms, ParMETIS [30] and Pt-Scotch [31], show good cuts quality but their performance scales poorly with respect to the number of processors as shown in [12].

Stream partitioning During the last years stream graph partitioning has been proposed in order to reduce the complexity [10] of multilevel approaches, since they take into account the entire input graph during the whole computation. These algorithms assign edges and vertices to various partitions by running a single pass through the whole graph. The goal, of the most part of these algorithms, is to guarantee the edge balance [10, 32] and to find a partitioning that reduces the usage of the resources and the computation overhead. These methods are faster than multilevel algorithms but they build partitioning with lower quality, in term of cuts, due to the sensitivity to the stream order. Moreover, it's generally difficult to parallelize streaming algorithms.

Hybrid partitioning approaches Some works have tried to use mixed strategies in order to take advantage of multiple of the described approaches. The authors of [33] proposed a new algorithm that uses both multilevel and LP. The coarsening phase is done by using the LP by combining the vertices having the same label iteratively; the partition phase is done by applying METIS [30] or KL [28]. Their results showed that for large graphs the use of LP is more efficient than matching criteria in the coarsening phase. However, their algorithm has to restart partitioning from scratch if any change to the graph occurs.

The authors of [12] proposed ParHIP, a distributed memory parallel partitioning algorithm, that takes advantage of both multilevel and LP approaches. The authors adapted and parallelized LP technique for both coarsening and refinement step, using the Message Passing Interface (MPI), while considering the vertex-balance constraint. Their experimental results demonstrated that ParHIP is more scalable and achieves higher quality than existing state of the art methods like ParMETIS and PT-Scotch.

Label propagation partitioning approaches Other works have used the label propagation approach (LP) [34] to partition large graphs. LP was mainly used for community detection in social networks [35, 36]. Making use of LP for the graph partitioning problem was motivated by the lightweight mechanism that uses the network structure to guide its progress. LP partitioning methods generate less intermediary results than multilevel approaches, which need to store many intermediate results such as the coarser graph, and run with a lower complexity. Furthermore, LP method is semantic-aware, given the existence of local closely connected substructures, a label tends to propagate within such structures. Finally, in [11] the authors defined a distributed partitioning algorithm called Spinner that considers only edge balance. Spinner is based on LP approach and runs on the top of Giraph API [1]. Compared to the previous work, Spinner supports the parallelism and can adapt an existing partitioning to consider graph updates by adding or removing vertices and edges and changing the number of partitions. The algorithm divides N vertices across K partitions, while trying to keep similar the number of edges in each partition. In the same context other approaches have been proposed to take advantage from distributed computation and Map-Reduce programming paradigms. In [37] the authors embedded the nodes onto a line, and then processed them in a distributed manner guided by a linear embedding order. Their focus was on balanced-partitioning and on minimizing the total cut size.

In many applications contexts graphs show a structure that changes continuously over the time. Some research works have focused their interest in how to deal with such characteristic and introduced dynamic partitioning techniques. In [38] the authors proposed a dynamic graph partitioning algorithm that simultaneously incorporates replication of the data. The algorithm focuses in maintaining quality partitioning as the graph structure changes over time taking into consideration only local parts of the graph when reassigning vertices to partitions. Exploring the opportunities offered by the techniques developed in streaming-data analysis [39] proposed a graph partitioning in streaming setting and [40] introduced a stratified graph partitioning technique where approximately the same graph is routinely streamed and analyzed. Thanks to this last approach it is possible to balance simultaneously each set of node attribute in the strata.

In this paper we present a new algorithm for balanced graph partitioning based on LP approach and using Giraph programming model. Compared to the literature, our algorithm B-GRAP deals with edge-based or vertex-based balanced partitioning while decreasing the number of cuts and computation time. Moreover, B-GRAP defines an initialization heuristic which allows to improve the propagation of labels across the graph and to accelerate the convergence of the algorithm on large graphs.

3. Preliminaries

The label propagation algorithm was defined in the context of community detection in social networks [35, 36]. This approach re-used in graph partitioning research context thanks to its lightweight and intuitive mechanism. Comparing with multilevel approaches LP methods generate less intermediary results and run with a lower complexity. Given the number of desired partitions of the graph, the naive LP algorithm simply works as follows: (i) At first, each vertex is assigned to a partition randomly; (ii) Then, the label of each vertex is propagated and updated iteratively to its neighborhood, where each vertex takes the most frequent label among its neighborhood as its own label. The process ends when labels no longer change. In the following we describe formally the LP algorithm.

Problem Formulation and Notations Given a number of partitions K, a directed graph $G = \langle V, E, \omega \rangle$, where V is a set of vertices and E a set of weighted edges with $\omega : E \to \mathbb{R}^+$. Let $L = \{l\}_{l=1}^{K}$ be a set of partition labels defined by a labeling function $\phi : V \to L$ such that $\phi(v) = l$ means that v belongs to the partition with label l. The naïve LP algorithm proceeds as follows. Initially, a unique label l_v is assigned to each vertex v. Then, the label of each $v \in V$ is propagated and updated iteratively to its neighborhood $N(v) = \{u \in V | (v, u) \vee (u, v) \in E\}$ and is updated until a given convergence criteria is reached. The label updating is done by taking into account the most frequent label among $N(v)$ labels. More formally, let $\mathcal{F}_{\mathrm{LP}}(v, l)$

be the frequency of a label l in the neighborhood of v, defined by:

$$\mathcal{F}_{\mathrm{LP}}(v,l) = \sum_{u \in N(v)} \omega(v,u)\delta\big(\phi(u), l\big) \tag{1}$$

where $\phi(u)$ gives the current label of u and δ is the Kronecker delta function, which is equal 1 if $\phi(u) = l$, and 0 otherwise. The label of vertex v is replaced by the label that maximizes the frequency function:

$$l_v = \underset{l}{\operatorname{argmax}}\ \mathcal{F}_{\mathrm{LP}}(v,l) \tag{2}$$

If many maximal labels exist and do not include the current label of v, one of them is randomly chosen. LP algorithm stops if $\sum_{v \in V} \sum_{l \in L} \mathcal{F}_{\mathrm{LP}}(v,l)$ converges according to a given threshold ϵ.

We note that naïve LP algorithm does not take into account the directions of edges. To consider directed graphs, virtual edges are added such that: $\forall (v,u) \in E, \omega(v,u) = 2$ and if $(u,v) \notin E, (u,v)$ is added with $\omega(u,v) = 1$ which we call *virtual edge*. Note that the Giraph data model is a distributed directed graph, where every vertex is aware of its outgoing edges only, but not of the incoming ones. Adding virtual edges in this case, allows to a vertex to discover its entire neighborhood $N(v)$, while the weight allows to consider the direction as well as to distinguish these virtual edges added.

4. B-GRAP algorithm

Our goal is to define a K-balanced and LP-based partitioning algorithm that decreases the total cuts while considering the vertex balance or the edge balance constraints in directed graphs.

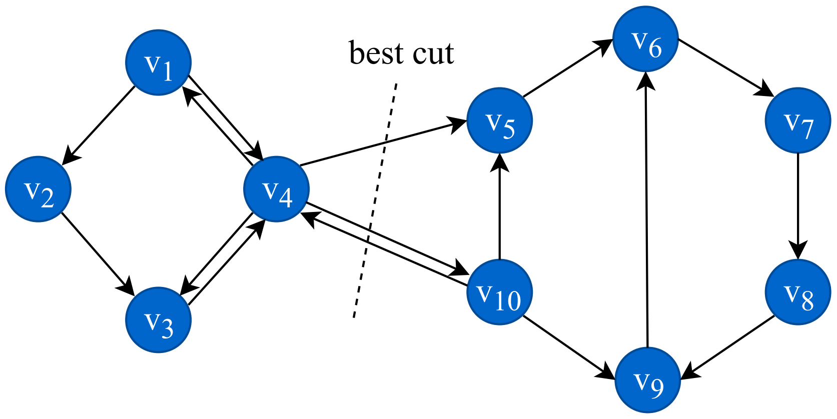

Fig. 1. Vertex-balanced and edge-balanced 2-partitioning graph example.

To illustrate our objectives we consider the example provided in Figure 1. This example presents a small graph of 10 vertices and 16 directed edges. We would divide the graph into two balanced partitions, using either a vertex-balanced partitioning or the edge-balanced partitioning. First, we note that the best 2-partitioning that minimizes the edge cuts is

$P_1 = \{v_1, v_2, v_3, v_4\}$ and $P_2 = \{v_5, v_6, v_7, v_8, v_9, v_{10}\}$, where the cuts $(v_4, v_5), (v_4, v_{10}), (v_{10}, v_4)$. To achieve a vertex-balanced partitioning, a vertex from P_2 should be moved to P_1, while caring about the cuts. In this case, moving v_{10} to P_1 is the most advantageous because it introduces less edge cuts. For the edge-balanced partitioning, no change is needed. Indeed, both partitions holds $|E|/2$ *directed edges* and the number of edge cuts is minimized. Note that if the directness of edges is ignored, this partitioning remains unbalanced. In fact, in an undirected graph, each edge is considered to be bidirectional, as a result the number of edges in P_1 is 12 and 16 in P_2.

In the following, we present in detail B-GRAP algorithm. First, we present the initialization strategy, then we define the update functions $\mathcal{F}$ to build vertex (or edge) balanced partitions, and finally we present the measures used for the evaluation of partitioning quality.

4.1. *B-GRAP initialization*

To improve the performance of propagation approach in our algorithm, we define an initialization strategy, called B-GRAP$_{init}$, which considers only *hub* vertices having a high outgoing degree $d^+(.)$. The intuition behind this choice is that the higher $d^+(v)$, the more $\phi(v)$ will be propagated and considered as frequent label. This differs from LP approach that considers all the vertices. As a result, the candidates to be considered as frequent labels (i.e. the labels with high probability to be the most frequent) are propagated faster at the first *partial propagation* iteration. The initialization we defined should guarantee a faster label propagation and smaller number of exchanges between vertices, given the fact that the nodes having the higher probability to be selected to propagate their label have been already initialized.

B-GRAP is described in Algorithm 1. Let τ be a given minimum out degree threshold to consider that a vertex v as a *hub* vertex. The algorithm proceeds as follows. First, we initialize the set of labels L (Line 1). Then, each $v \in V$, such $d^+(v) > \tau$ is assigned a random label $\in L$ and those labels are propagated to neighbors (Line 2). Then, the label of these neighbors are updated and propagated iteratively using an update function (Lines 4-7). The vertices are then checked and those not reached by the update/propagation step are initialized randomly, to ensure that all vertices are assigned a label (Line 9). The algorithm repeats the update/propagate step until convergence (Line 10).

4.2. *B-GRAP objective functions*

In the basic LP partitioning, the label update is done without caring about the size of the partitions. Consequently, this can lead to an unbalanced partitioning. Moreover the update function of LP Eq. (1) has a trivial optimal solution that consists of assigning all vertices to a single label, i.e. to a single partition. A standard resolution approach to deal with such a problem is to integrate the balance constraints to the update function via a penalty function. The LP update function becomes:

$$\mathcal{F} = \mathcal{F}_{\text{LP}} + \lambda \mathcal{P} \tag{3}$$

where $\mathcal{P}$ represents penalty terms and λ is a weight parameter. In B-GRAP algorithm, we define two update functions $\mathcal{F}_{\text{VB}}$ and $\mathcal{F}_{\text{EB}}$ which respectively deal with vertex and edge balance constraints.

Algorithm 1 B-GRAP

```
input  $G = \langle V, E, w \rangle$, $K$, $\tau$, $\epsilon$
output  a partitioned graph $G = \langle V, E, w, L \rangle$
1: Initialize the set of labels $L = \{l\}_{l=1}^{K}$
2: for $\big(v \in V, d^{+}(v) > \tau\ \big)$ initialize $\phi(v)$ randomly from $L$ and propagate to $N(v)$
3: repeat
4:     {Search frequent labels}
5:     for $(v \in V, l \in L)$ get the set of frequent labels w.r.t an update function
6:     {Update and propagate}
7:     Update and propagate $\phi(v)$ to $N(v)$
8:     {Check unassigned vertices}
9:     for $\big(v \in V,\ \phi(v) = \emptyset\big)$ initialize $\phi(v)$ randomly from $L$ and propagate to $N(v)$
10: until $\Delta\big(\mathcal{F}_{\mathrm{LP}}(G, L)\big) \leq \epsilon$
11: return $G = \langle V, E, w, L = \{\phi(v)\}_{v \in V} \rangle$
```

4.2.1. *Vertex balance*

Given a directed graph $G = \langle V, E, \omega \rangle$, a vertex-balanced partitioning divides the vertices into disjoint partitions of nearly equal size, while minimizing the number of edge cuts between partitions. Let $size(V, l)$ be the number of vertices having l as label, $size(V, l) = |\{v \in V \mid \phi(v) = l\}|$.

In a perfect balanced partitioning, the size of each partition should be equal to $|V|/K$. In other words, the distribution of vertices in the partitions should be close to a uniform distribution $\mathbf{U} = \langle 1/K, \ldots, 1/K \rangle$, where $1/K$ is called the balance factor. To handle the balance between the partitions, we define vertex-balance $\mathcal{P}_{\mathrm{VB}}$ penalty function that penalizes $\mathcal{F}$ when trying to assign a vertex to a partition violating the balance constraints as follows:

$$\mathcal{P}_{\mathrm{VB}}(l) = \frac{1}{K} - \frac{size(V, l)}{|V|} \tag{4}$$

This function measures the difference between the balance factor $1/K$ and the ratio of vertices assigned to l label. The larger the ratio of vertices with label l is, the higher the penalty to update the vertex label with l is.

At this stage, the number of edge cuts between the partitions is not considered. Thus, a vertex could move to a partition that increases the edge cuts. Given a vertex v and label l, we define a second penalty function as follows:

$$\mathcal{P}_{\mathrm{EC}}(v, l) = \frac{|cut(v, l)|}{d^{+}(v)} \tag{5}$$

where $cut(v, l) = \{(v, u) \in E \mid \phi(u) = l\}$ is the set of edges outgoing from v to vertices in a partition with label l. This function measures the ratio of cuts which penalizes a vertex v to move to a partition with l label if the number of its outgoing edges to this partition is low (normalized to the out degree of v). Thus, when a vertex has more connections to a partition than to the others, the penalty gives more advantage to move to this partition and vice versa. By considering the penalty functions defined in Eq. (4) and Eq. (5), the vertex balance update function is defined in the following equation:

$$\mathcal{F}_{\text{VB}}(v,l) = n\mathcal{F}_{\text{LP}} + \lambda\Big(\kappa\mathcal{P}_{\text{EC}}(v,l) + (1-\kappa)\mathcal{P}_{\text{VB}}(v,l)\Big) \tag{6}$$

where n is a normalization constant equal to $\frac{1}{\sum_{u\in N(v)}\omega(v,u)}$. The balance factor $\frac{1}{K}$ could be omitted as it is constant, in this case $\mathcal{P}_{\text{VB}}$ variate $\in [0\ldots 1]$. The parameter κ is a weight ranging between 0 and 1 which gives more or less importance to balance penalty against the edge cuts penalty. We set κ to 0.5 by default.

4.2.2. *Edge balance*

An edge-balanced partitioning divides the graph into disjoint partitions holding nearly equal number of edges, while minimizing the number of edge cuts between partition. Let $size(E,l)$ be the number of outgoing edges from a partition with label l, $size(E,l) = \sum_{v\in V,\phi(v)=l}|d^+(v)|$. Similarly to the vertex-balance partitioning, we define the following edge-balance penalty function:

$$\mathcal{P}_{\text{EB}}(l) = \frac{1}{K} - \frac{size(E,l)}{|E|} \tag{7}$$

This function discourages a vertex to move to a partition with l label, when the ratio of edges in the partition l is closer or larger than the balance factor. Comparing to vertex balance, edge balance maximizes the edge locality in each partition, which contributes to minimizing the edge cuts. Thus, there is no need to add additional penalty to the update function as defined in Eq. (6). The edge-balance update function is formulated as follows:

$$\mathcal{F}_{\text{EB}}(v,l) = n\mathcal{F}_{\text{LP}} + \lambda\mathcal{P}_{\text{EB}}(l) \tag{8}$$

We note that Spinner algorithm [11] (see Section 2) uses the normalized unbalance as penalty function. Comparing to Eq. (7), the edge-size of a partition is normalized by the size of a perfect balanced partition, i.e. $\frac{|E|}{K}$. Moreover, their penalty function that measures the edge balance for each partition, considers both virtual and real edges. The function we defined in Eq. (8) considers only real edges.

5. Partitioning Evaluation Measures

To evaluate the quality of the partitioning produced by B-GRAP, we use two standard measures: the ratio of edge cuts **EC** and the Jensen Shannon divergence (**JSD**) [41].

The edge cuts ratio is the ratio of edges connecting each two vertices in two different partitions w.r.t the total number of edges.

$$\textbf{EC} = \frac{\sum_{v\in V}\sum_{l=1}^{K}|cut(v,l)|}{|E|} \tag{9}$$

The Jensen Shannon divergence (JSD) is the symmetric version of the Kullback–Leibler divergence known as a standard measure to compute the divergence between two distributions. This is a symmetric measure varying in the interval $[0\ldots 1]$, where a value close to 0 indicates that the distributions are similar. Let $\mathbf{P} = \langle p_1, \ldots, p_K \rangle$ and $\mathbf{Q} = \langle q_1, \ldots, q_K \rangle$ two distributions with the same size. The **JSD** divergence is computed as follows:

$$\mathbf{JSD}(\mathbf{P}||\mathbf{Q}) = \frac{1}{2}\Big(D_{\mathrm{KL}}(\mathbf{P}\|M) + D_{\mathrm{KL}}(\mathbf{Q}\|M)\Big)$$
$$\text{with } D_{\mathrm{KL}}(\mathbf{P}\|\mathbf{Q}) = \sum_{l=1}^{K} p_l \log(\frac{p_l}{q_l}) \quad \text{and } M = \frac{1}{2}\big(\mathbf{P} + \mathbf{Q}\big) \qquad (10)$$

In our case, $\mathbf{P}$ represents the distribution of vertices (or edges) on the partitions, where p_l is the ratio of vertices (or edges) in the partition with label l, and $\mathbf{Q}$ equals to the uniform distribution $\mathbf{U}$. The **JSD** considers the balance of the whole partitioning, comparing to other measures such the maximum normalized unbalance metric (**MNU**) [11]. This last is used to measure unbalance and represents the percentage-wise difference of only the largest partition from a perfectly balanced partition.

$$\mathbf{MNU}_{\mathrm{VB}} = \frac{\max(|V_l|)}{|V|/K},$$
$$\mathbf{MNU}_{\mathrm{EB}} = \frac{\max(|E_l|)}{|E|/K}, \quad \text{with } l \in L. \qquad (11)$$

Finally, it is important to notice that for **EC**, **JSD**, and **MNU** we consider the directed edges in the original input graph. The virtual edges added for neighborhood discovery (see Section 3) are not taken into account.

6. Giraph parallel processing system

Processing large graphs introduces specific challenges that are not tackled by traditional tools for data analytics. Giraph [1] is a framework designed to easily parallelize and execute iterative algorithms on massive graph datasets, using Apache Hadoop's MapReduce. Giraph was inspired by Pregel parallel computation model, proposed by Google a previously published article [2]. Pregel defines a graph parallel processing system with a vertex-centric paradigm. In this paradigm, a computation function is executed in parallel by each vertex during each iteration (called *superstep*). This paradigm helps to achieve a high degree of parallelism, because the computation function can run in parallel on the graph vertices, and to deal with the scalability problem. However, it comes at the trade-off of being more restrictive during the implementation.

Giraph works in synchronized supersteps, but the execution inside each superstep is asynchronous. Each vertex can be active or inactive in a superstep: (i) if it is active, the vertex will be able to access its own value, access its edges, send messages to it's neighbors and/or vote to halt the computation, (ii) if it is inactive, the node will not execute. On the first superstep all nodes are active and the computation function can be executed once on each vertex. In following supersteps only vertices that receive a message, from previous superstep, will become active (even if previously they voted to halt) and can run the same or another computation function. A Giraph job therefore ends when all vertices have voted to halt and no further messages are being sent. The execution of all active nodes is parallelized and therefore executed in an asynchronous manner. Since in order to halt the computation we

need to make sure that no messages are being sent, Giraph waits until the last vertex finishes it's execution before deciding if another superstep is required or the job must be ended.
In order to facilitate global calculation, Giraph provides the *Aggregators*, in which we can store global statistics. We can think of them as global variables, to which vertices can provide some values. These values get aggregated by Giraph at the end of each superstep then the results become available to all vertices in the following superstep. Note that, there is no fixed ordering in which the aggregations are done, as the vertex execution is asynchronous. Thus, commutative and associative operations are only supported by the *Aggregators*. This mechanism makes Giraph, overall, a synchronous platform.

B-GRAP implementation Every algorithm implemented on Giraph system needs to deeply think about how to define the algorithm using a *Think-LiKe-A-Vertex* way. In our case, each step of B-GRAP, described in Algorithm 1, is implemented as a computation function to be run on the vertices. We use two lists of Aggregators to store the number of edge and the number of vertices in each partition. The vertex-centric paradigm of our algorithm can be described as follows:

(i) Initialization superstep: each hub vertex choose randomly a label l ($\in [1, K]$) as the identifier of its partition. Then it provides the value 1 to the Aggregator corresponding to the number of vertices in this partition, the value $d^+(v)$ for the Aggregator corresponding to the number of edges. At the end of the superstep, all the values get aggregated (summed) to be used in the next superstep to compute the balance.
(ii) Frequent label computation superstep: each vertex computes the score for each label based on the set of labels of incoming neighbors and on one of the objective functions defined in Section 4.2. If a new partition has a higher score (or depending on the heuristics used) than the current vertex's partition, the vertex decides to try to update its partition during the following superstep. Otherwise, it does nothing.
(iii) Partition's update and propagation superstep: concerned vertices try to update their partition according to the ratio of vertices that decided to move to a partition i and the remaining capacity of partition i. Vertices which succeed the update, communicate new values to the Aggregators of i and propagate their new partitions to their neighbors.
(iv) Supersteps (ii) and (iii) keep on being alternated until convergence is reached. The convergence condition is checked on the master program, before running a new frequent label computation superstep. Once the convergence reached, the evaluations measures can be computed on the basis of the final states of the partitions stored in the lists of Aggregators.

7. Experiments

We achieve different experiments on different graph data sets in order to evaluate the quality of edge and vertex-balanced partitioning using **EC**, **JSD**, and **MNU** measures defined previously. We compare our approach to Spinner [11] because it has shown better results comparing to some existing algorithms. Moreover, since both B-GRAP and Spinner are developed using Apache Giraph environment, we can also provide an evaluation in the same system conditions.

In the following, we first describe the data sets and the experiment settings. Then, we present in detail the results of B-GRAP$_{\text{VB}}$ and B-GRAP$_{\text{EB}}$, compared against Spinner, and achieved on nine graphs.

7.1. *Data sets description and experiment settings:*

All the experiments are done on a Hadoop cluster of 8 machines, with 64GB RAM and 8 compute cores. B-GRAP algorithm is implemented in Java using Apache Giraph environment [1]. Giraph is an open source implementation of distributed programming framework Pregel [2], designed for Google cluster architecture, with several performance improvement like multi-threading and memory usage optimization. It's built on Hadoop infrastructure to make distributed graph processing and can work with many data storage system supporting graph data (Neo4j, DEX, RDBMS, etc.). In Giraph, the graph is randomly partitioned on several workers (machines)after a complete in-memory load. As in Pergel, Giraph uses a vertex-centric approach to deal with large scale graph processing. In their approach, the computation of the user defined function is done locally, i.e. on each vertex, and in parallel. A vertex contains information about itself and its outgoing edges, it can change its state and the state of these edges by exchanging messages with other vertices at the same iteration, called *super-step*.

In our experiments, we use nine graph data sets of different degree distributions and different sizes in terms of edge and vertex number as summarized in Table 1. Wikitalk (W), Pockec (P), Flixster (F), LiveJournal (L), and Orkut (O) are social online networks graphs. BerkeleyStanf (B) is the berkely.edu and stanford.edu web graph, SK-2005 (S) is hyperlinks on '.sk' web. DelaunaySC (D) and Graph500 (G) are synthetic graphs. Notice that only (G) is an undirected graph.

Table 1. Data sets description

Graph	WikiTalk	BerkeleyStanf	Flixster	DelaunaySC	Pokec	LiveJournal	Orkut	Graph500	SK-2005
	(W)	(B)	(F)	(D)	(P)	(L)	(O)	(G)	(S)
Directed	yes	yes	yes	yes	yes	yes	yes	no	yes
$\lvert V \rvert$	2.4M	0.7M	2.5M	8.4M	16M	4.8M	2.7M	4.6M	50.6M
$\lvert E \rvert$	5M	7.6M	7.9M	25.2M	30.1M	69M	117.2M	258.5M	1.9B
Source	[42]	[43]	[44, 45]	[16, 44]	[46]	[47]	[44]	[44]	[12]

Experimental setting: We evaluate our algorithm over all the graphs presented in Table 1, by varying the number of partitions K from 2 to 32. More precisely, we execute 10 runs of B-GRAP$_{\text{VB}}$ and B-GRAP$_{\text{EB}}$ for each graph and each value of K to ensure the significance of the results. For all experiments, we compute the average variation of the following measures with respect to the number of partitions K and over the runs:

- The maximum normalized unbalance of vertices (**MNU**$_{\text{VB}}$) and of edges (**MNU**$_{\text{EB}}$).
- The divergence between the distribution of vertices (respectively of edges) and the uniform distribution **JSD**$_{\text{VB}}$ (respectively **JSD**$_{\text{EB}}$).
- The edge-cuts ratio (**EC**).
- The computation time saving ratio (**ΔTime**) of B-GRAP w.r.t Spinner. This ratio is computed using the total CPU time in seconds spent to execute the algorithm, from the

initialization until the convergence, and it is computed with:

$$\mathbf{\Delta Time} = \frac{\mathbf{\Delta Time}_{\text{Spinner}} - \mathbf{\Delta Time}_{\text{B-GRAP}}}{\mathbf{\Delta Time}_{\text{Spinner}}}$$

.

Note that $\mathbf{\Delta Time} > 0$ means a better performance of our algorithm and a value close to 0 means similar performances with Spinner.

For all experiments, we set $\epsilon = 10^{-3}$ as a threshold stop value and we set τ average out degree $\bar{d}^{+} = \frac{|E|}{|V|}$. The penalty term weight parameter λ in the update function $\mathcal{F}$ is set to 1. This gives an equal importance to the penalty term $\mathcal{P}$ and to $\mathcal{F}_{\text{LP}}$ according to the update functions defined in Section 4.2.

7.2. *Initialization impact*

In this section, we evaluate the impact of the initialization step of B-GRAP$_{init}$ on the LP approach.This experiment shows that B-GRAP$_{init}$ has a positive impact on the convergence time of a LP based partitioning. More precisely, we show how B-GRAP$_{init}$ integrated in Spinner partitioning algorithm helps to reduce the number of iterations required until the convergence, while scaling the number of edges, vertices, and partitions. Besides, our initialization does not impact the quality of cuts neither the edge balance objective of Spinner algorithm. It is important to note that Spinner algorithm considers the balance of both original and virtual edges.

Table 2. The gain ratio percentage of Spinner combined with B-GRAP$_{init}$ w.r.t. the original Spinner algorithm according to the number of iterations (**ΔIter**), to the number of cuts (**ΔEC**), and to the computation time (**ΔTime**). A value > 0 means a better results for B-GRAP$_{init}$ and vice-versa.

	WikiTalk (W)			BerkeleyStanf (B)			Flixster (F)		
K	**ΔIter** %	**ΔTime**	**ΔEC** %	**ΔIter** %	**ΔTime**	**ΔEC** %	**ΔIter** %	**ΔTime**	**ΔEC** %
2	**43**	35.1	22.2	**27**	19.1	21.7	**34**	24.6	-39.3
4	**41**	32.0	31.1	**19**	8.4	11.1	6	4.1	-2.4
8	**46**	34.9	28.8	**30**	22.4	13.7	**23**	16.8	-6.3
16	**52**	43.3	26.1	**40**	30.1	8.7	**33**	25.1	-2.6
32	**64**	54.1	22.7	**38**	30.3	6.1	**55**	47.8	-2.9
	DelaunaySC (D)			Pokec (P)			LiveJournal (L)		
K	**ΔIter** %	**ΔTime**	**ΔEC** %	**ΔIter** %	**ΔTime**	**ΔEC** %	**ΔIter** %	**ΔTime**	**ΔEC** %
2	**33**	25.2	10.6	-10	-6.8	-3.0	**14**	11.6	-12.4
4	**13**	10.0	-10.8	6	4.9	0.4	**12**	6.5	1.2
8	3	-0.4	-1.5	4	2.5	-2.2	8	3.2	1.2
16	4	1.5	0.6	1	-1.2	-0.2	**10**	6.3	-1.1
32	5	4.0	-0.7	**10**	5.7	-1.4	**11**	7.9	-0.1
	Orkut (O)			Graph500 (G)			SK-2005 (S)		
K	**ΔIter** %	**ΔTime**	**ΔEC** %	**ΔIter** %	**ΔTime**	**ΔEC** %	**ΔIter** %	**ΔTime**	**ΔEC** %
2	**16**	10.2	3.5	9	-0.1	-2.2	8	6.5	-43.4
4	-1	-2.8	3.9	-10	-20.7	1.0	5	-4.1	-23.7
8	1	-3.5	0.7	**49**	32.6	-0.6	8	-8.2	-33.7
16	-2	-6.3	0.7	**42**	28.7	-0.3	**14**	-11.7	-14.1
32	-10	-12.4	1.6	**46**	35.3	-0.2	**20**	4.9	-16.8

Results Using the experimental setting defined in the paper, we compute over all runs of Spinner and Spinner combined with B-GRAP$_{init}$, for each graph and for each number of partitions $K \in 2, 4, 8, 16, 32$, the ratio of difference (Δ) between the results of Spinner and Spinner combined with B-GRAP$_{init}$. More precisely, we compute and present in Table 2 the following scores:

- the convergence gain ratio according to the total number of iterations needed until the convergence
$$\mathbf{\Delta Iter} = \frac{\mathbf{Iter}_{\text{Spinner}} - \mathbf{Iter}_{\text{B-GRAP}_{init}}}{\mathbf{Iter}_{\text{Spinner}}}$$
- the edge-cuts gain ratio
$$\mathbf{\Delta EC} = \frac{\mathbf{EC}_{\text{Spinner}} - \mathbf{EC}_{\text{B-GRAP}_{init}}}{\mathbf{EC}_{\text{Spinner}}}$$
- the time saving ratio
$$\mathbf{\Delta Time} = \frac{\mathbf{\Delta Time}_{\text{Spinner}} - \mathbf{\Delta Time}_{\text{B-GRAP}_{init}}}{\mathbf{\Delta Time}_{\text{Spinner}}}$$

A gain ratio > 0 a better performance of B-GRAP$_{init}$ and a value close to 0 means similar performance with Spinner. We emphasize in Table 2 significant values of $\mathbf{\Delta Iter} \geq 10\%$.

The values of $\mathbf{\Delta Iter}$ reported in Table 2 show that our proposed initialization B-GRAP$_{init}$ helps to reduce the number of iterations of the LP algorithm for all graphs. The gain ratio $\mathbf{\Delta Iter}$ is significant on (W), (B), (F), (L), and (G) and slightly high for others. In fact, their is only three notable exceptions on (P) for $K = 2$, (O) for $K = 32$ and (G) for $K = 4$.

Similarly, the time saving ratio is almost $\mathbf{\Delta Time} > 0$ (on 7 graphs over 9), which means that the Spinner algorithm running faster when using our initialization B-GRAP$_{init}$. The exception that we notice on (O) and (S) graphs can be explained by the numbers of messages exchanged between during the whole B-GRAP$_{init}$, which was higher than the number of messages exchanged with Spinner.

The results on the $\mathbf{\Delta EC}$ show that our proposed initialization speeds up the partitioning convergence without decreasing the quality of cuts. The quality of cuts obtained with B-GRAP$_{init}$ is better than Spinner for (W) and (B) graphs, similar for (D), (P), (L), (O), and (G). However, B-GRAP$_{init}$ shows bad quality for (S). We suppose that this result is a consequence of the structure of the dataset that shows *hub vertex* strongly connected. We are analyzing this case and we will study it in more detail in the future extension of our work.

Finally, we note that for the gain ratio of the maximum normalized edge unbalance $\mathbf{MNU}_{eb}$ (respectively the $\mathbf{JSD}_{\text{EB}}$ for edge balance) scores between Spinner and B-GRAP$_{init}$, we had the same behavior as the $\mathbf{\Delta EC}$. This result shows that our initialization does not impact the edge balance of the partitioning. B-GRAP$_{init}$ improves then both the cuts and the convergence speed.

7.3. *B-GRAP Vertex balance:*

The experiments presented in this section consider the vertex balance constraints to partition a graph. The main objective is to evaluate the ability of our algorithm B-GRAP$_{\text{VB}}$ to produce balanced partitions with respect to the number of vertices, while improving the quality of cuts, using the vertex-balance update function defined in Eq. (6).

For this aim, using the experimental protocol described in Section 7.1, we compare the balance and the cuts quality of B-GRAP$_{\mathrm{VB}}$ partitioning with Spinner partitioning.

Results: The results are presented in Figure 2. This figure shows, for each graph and algorithm, the average variation of the $\mathbf{MNU}_{\mathrm{VB}}$, $\mathbf{JSD}_{\mathrm{VB}}$, **EC**, and $\mathbf{\Delta Time}$ according the number of partitions K. The figure horizontally shows a particular measure and vertically a particular set of graphs.

`figure/Spinner-vs-BGRAP_vb-JDI.pdf`

Fig. 2. Variation of the average scores of $\mathbf{MNU}_{\mathrm{VB}}$, $\mathbf{JSD}_{\mathrm{VB}}$, **EC**, and $\mathbf{\Delta Time}$ for the partitioning obtained with B-GRAP$_{\mathrm{VB}}$ and Spinner, w.r.t. K

We analyze first the variation of the unbalance degree $\mathbf{MNU}_{\mathrm{VB}}$ and the total balance of the partitioning $\mathbf{JSD}_{\mathrm{VB}}$. As shown on the Figure 2, B-GRAP$_{\mathrm{VB}}$ produces generally a low unbalance degree for the most part of graphs (seven over nine w.r.t. $\mathbf{MNU}_{\mathrm{VB}}$) while varying the number of partitions K. On the other side, the results of Spinner show a high unbalance degree $\mathbf{MNU}_{\mathrm{VB}}$ (> 1.1) when scaling with K, in particular for $K \geq 4$, except for (D) graph. We notice only two exceptions for B-GRAP$_{\mathrm{VB}}$ on (B) and (W) graphs, when $K \geq 24$. However, the $\mathbf{MNU}_{\mathrm{VB}}$ of B-GRAP$_{\mathrm{VB}}$ is still lower then Spinner for (B) graph.

Similarly, the results of $\mathbf{JSD}_{\mathrm{VB}}$ show that B-GRAP$_{\mathrm{VB}}$ performs generally better than Spinner. The value of $\mathbf{JSD}_{\mathrm{VB}}$ is very close to 0 over all graphs and for all K. This means that B-GRAP$_{\mathrm{VB}}$ produces high balanced partitions. B-GRAP$_{\mathrm{VB}}$ gives better results for 6 graphs over 9 (with 5 significant differences for (B), (D), (P), (O), and (S) and similar results for the others). We note that for the exceptions on (W) and (B) noticed previously for $\mathbf{MNU}_{\mathrm{VB}}$, the $\mathbf{JSD}_{\mathrm{VB}}$ values are very close to 0 which means that the partitioning has a high global balance degree.

We compare the quality of cuts for both algorithms. Figure 2 shows similar quality of **EC**. B-GRAP$_{\mathrm{VB}}$ shows significant better results on *BerkeleyStanf* and *WikiTalk* graphs.

Finally, the $\mathbf{\Delta Time}$ curves show that B-GRAP$_{\mathrm{VB}}$ improves significantly the computation time on all graphs. The time saving percent $\mathbf{\Delta Time}$ is higher than 10% for all the graphs and all K values, except of (O) graph, where the the results are better but less significant.

7.4. *B-GRAP Edge balance:*

Now we compare the performance of our algorithm B-GRAP$_{\mathrm{EB}}$ with Spinner, using the edge-balance update function defined in Eq. (8).

Results: We present the results of this experiment in Figure 3. For each graph and algorithm we show the average variation of the following measures w.r.t. K: $\mathbf{MNU}_{\mathrm{EB}}$, $\mathbf{JSD}_{\mathrm{EB}}$, **EC**, and $\mathbf{\Delta Time}$.

Figure 3 shows that the partitioning produced by B-GRAP$_{\mathrm{EB}}$ has a low edge unbalance degree for all graphs under analysis. In fact, the average $\mathbf{MNU}_{\mathrm{EB}}$ is generally less than 1.05, except in the case of *WikiTalk* for $K = 28$ and $K = 32$ where the average $\mathbf{MNU}_{\mathrm{EB}}$ is equal to 1.12 and 1.13, respectively. However, if we analyze the results obtained from running Spinner, we see that we obtain an unbalance degree $\mathbf{MNU}_{\mathrm{EB}}$ generally higher than 1.05 and $\mathbf{MNU}_{\mathrm{EB}}$ shows bad values while increasing the number of partitions K. On the contrary, the variation of $\mathbf{MNU}_{\mathrm{EB}}$ for B-GRAP$_{\mathrm{EB}}$ shows that it scales with K with a stable balance quality.

The behaviour of $\mathbf{JSD}_{\mathrm{EB}}$ shows that B-GRAP$_{\mathrm{EB}}$ generally scales up with K while maintaining a good global balance, with few exceptions. Furthermore, B-GRAP$_{\mathrm{EB}}$ obtains better performance than Spinner over five graphs and gives similar $\mathbf{JSD}_{\mathrm{EB}}$ scores for the others.

The quality of cuts is generally close for both algorithms (Figure 3), with only one significant better result on *WikiTalk*.

Finally, $\mathbf{\Delta Time}$ variation shows significant better results for B-GRAP$_{\mathrm{EB}}$ on (W), (B), (F) and (G) graphs. The computation time is slightly better for other graphs. In fact, this is only with few exceptions on (G) for $K \leq 4$ and for (S).

To summarize this experimental part, B-GRAP$_{\mathrm{EB}}$ algorithm computes higher edge balanced partitioning without impacting the quality of cuts and while showing generally faster

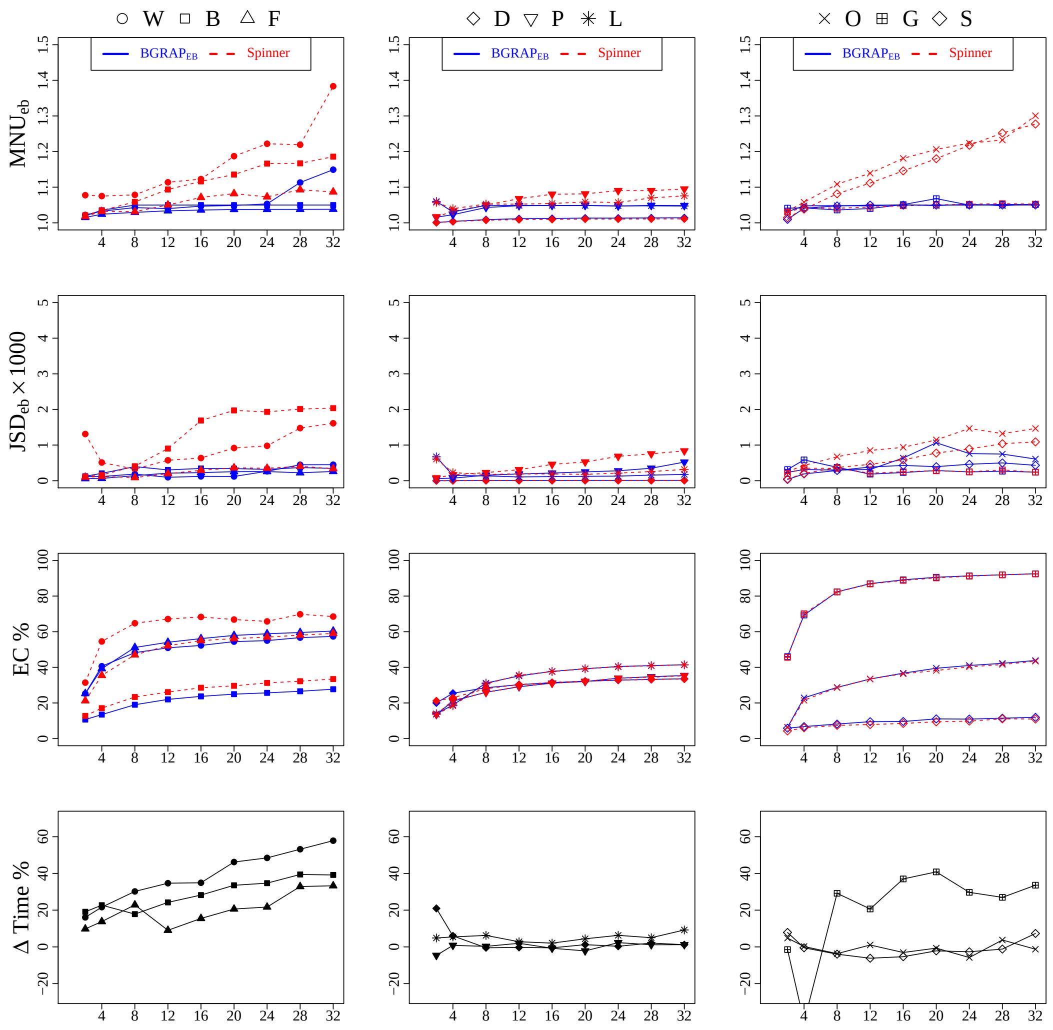

Fig. 3. Variation of the average scores of $\mathbf{MNU}_{\mathrm{EB}}$, $\mathbf{JSD}_{\mathrm{EB}}$, **EC** and $\mathbf{\Delta Time}$ for the partitioning obtained with B-GRAP$_{\mathrm{EB}}$ and Spinner, w.r.t. K

computation time. Moreover, the results on the undirected graph *Graph500* show that the initialization step is efficient for the time execution of the algorithm. Finally, the results given for the $\mathbf{JSD}_{\mathrm{EB}}$ and $\mathbf{MNU}_{\mathrm{EB}}$ show that a better balance can be obtained if we consider the directness of edges for a directed graph.

8. Conclusion and Perspectives

In this paper we proposed two scalable and parallel partitioning algorithms B-GRAP$_{\mathrm{VB}}$ and B-GRAP$_{\mathrm{EB}}$, based on LP, that address the vertex balance and the edge balance problems on both directed and undirected graphs. We defined the initialization strategy of our algorithm that allows to speed up the convergence and two update functions to produce either vertex balanced or edge balanced partitioning.

Our results show good performances of B-GRAP on various graphs and with different scales. We show the positive impact of our proposed initialization conducted on 9 Graphs, where B-GRAP$_{init}$ speeds up the partitioning convergence without decreasing the quality of cut, comparing to another LP based partitioning method. We show that B-GRAP produces high vertex balanced and high edge balanced partitioning with a good cut quality comparing to Spinner algorithm (significant values for 5 graphs and slightly better values for others), on either directed and undirected graphs. Moreover, the computation time of B-GRAP is better than Spinner, with a few exceptions for B-GRAP$_{\mathrm{EB}}$ on two graphs.

As a future work, we plan to study other heuristics to build seeds for initialization based on graph sampling approaches, while exploring the connectivity of vertices. We believe that the relevant graph sample allows us to improve the propagation step in B-GRAP and to optimize more the quality of the partitioning.

We would also study the impact of the partitioning on algorithms of graph analytics with respect to the balance strategy, such as Shortest Path Computation, PageRank, and Community Detection.

References

1. A. Giraph, *Giraph : Large-scale graph processing in Hadoop.* 2012.
2. G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A system for large-scale graph processing,” in *Proc. of the 2010 ACM SIGMOD Inter. Conf. on Management of Data*, pp. 135–146, 2010.
3. R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “GraphX: a resilient distributed graph system on spark,” in *First International Workshop on Graph Data Management Experiences and Systems*, pp. 1–6, ACM Press, 2013.
4. A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, “Recent advances in graph partitioning,” in *Lect. Notes Comput. Sci.*, vol. 9220, pp. 117–158, 2016.
5. H. Das and S. Kumar, “A parallel TSP-based algorithm for balanced graph partitioning,” in *2017 46th Inter. Conf. on Parallel Processing (ICPP)*, pp. 563–570, IEEE, 2017.
6. Y. Li, C. Constantin, and C. du Mouza, “A Block-Based Edge Partitioning for Random Walks Algorithms over Large Social Graphs,” in *Proc. 17th Int. Conf. Web Inf. Syst. Eng. (WISE)- Vol. 10042*, pp. 275–289, Springer-Verlag New York, Inc., 2016.
7. D. Nguyen, *Graph Partitioning.* ISTE, 2011.
8. G. Karypis and V. Kumar, “Multilevel graph partitioning schemes,” in *Proc. of the 24th Inter. Conf. on Parallel Processing (ICPP) 1955*, vol. 3, pp. 113–122, 1995.
9. A. Henzinger, A. Noe, and C. Schulz, “ILP-based local search for graph partitioning,” 2018.

10. C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Proc. of the 7th ACM Inter. Conf. on Web search and data mining*, pp. 333–342, 2014.
11. C. Martella, D. Logothetis, A. Loukas, and G. Siganos, "Spinner: Scalable graph partitioning in the cloud," in *Proc. - Int. Conf. Data Engineering*, 2017.
12. H. Meyerhenke, P. Sanders, and C. Schulz, "Parallel graph partitioning for complex networks," vol. 28, pp. 2625–2638, 2017.
13. J. Ugander and L. Backstrom, "Balanced label propagation for partitioning massive graphs," in *Proc. of the 6th ACM Inter. Conf. on Web Search and Data Mining*, WSDM '13, pp. 507–516, ACM, 2013.
14. P. Sanders and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, vol. 7933 of *LNCS*, pp. 164–175, Springer, 2013.
15. A. EL Moussawi, N. B. Seghouani, and F. Bugiotti, "A graph partitioning algorithm for edge or vertex balance," in *Database and Expert Systems Applications* (S. Hartmann, J. Küng, G. Kotsis, A. M. Tjoa, and I. Khalil, eds.), (Cham), pp. 23–37, Springer International Publishing, 2020.
16. D. Bader, H. Meyerhenke, and D. Wagner, *Graph Partitioning and Graph Clustering*, vol. 588 of *Contemporary Mathematics.* 2013.
17. S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," vol. 51, pp. 1–53, 2018.
18. K. M. Hall, "An r-dimensional quadratic placement algorithm," *Management Science*, vol. 17, no. 3, pp. 219–229, 1970.
19. M. E. J. Newman, "Spectral methods for network community detection and graph partitioning," *CoRR*, vol. abs/1307.7729, 2013.
20. M. A. Riolo and M. E. J. Newman, "First-principles multiway spectral partitioning of graphs," *J. Complex Networks*, vol. 2, no. 2, pp. 121–140, 2014.
21. L. Grady and E. L. Schwartz, "Isoperimetric graph partitioning for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 3, pp. 469–475, 2006.
22. G. Cheung, E. Magli, Y. Tanaka, and M. K. Ng, "Graph spectral image processing," *Proc. IEEE*, vol. 106, no. 5, pp. 907–930, 2018.
23. Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2020.
24. J. Shen and F. Huet, "Predict the best graph partitioning strategy by using machine learning technology," in *Proceedings of the 2018 VII International Conference on Network, Communication and Computing*, (New York, NY, USA), p. 27–33, Association for Computing Machinery, 2018.
25. W. Y. H. Adoni, T. Nahhal, M. Krichen, B. Aghezzaf, and A. Elbyed, "A survey of current challenges in partitioning and processing of graph-structured data in parallel and distributed systems," *Distributed Parallel Databases*, vol. 38, no. 2, pp. 495–530, 2020.
26. A. Nazi, W. Hang, A. Goldie, S. Ravi, and A. Mirhoseini, "GAP: generalizable approximate graph partitioning framework," *CoRR*, vol. abs/1903.00614, 2019.
27. M. Hasanzadeh Mofrad, R. Melhem, and M. Hammoud, "Revolver: Vertex-centric graph partitioning using reinforcement learning," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 818–821, 2018.
28. B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," vol. 49, no. 2, pp. 291–307, 1970.
29. F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. of the Inter. Conf. and Exhibition on High-Performance Computing and Networking*, HPCN Europe 1996, pp. 493–498, 1996.
30. G. Karypis and V. Kumar, "A parallel algorithm for multilevel graph partitioning and sparse matrix ordering," vol. 48, no. 1, pp. 71–95, 1998.
31. C. Chevalier and F. Pellegrini, "PT-Scotch: A tool for efficient parallel graph ordering," Tech. Rep. 6-8, 2008.

32. J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. of the 10th USENIX Conf. on Operating Systems Design and Implementation*, pp. 17–30, 2012.
33. L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *2014 IEEE 30th Inter. Conf. on Data Engineering*, pp. 568–579, IEEE, 2014.
34. U. N. Raghavan, R. Albert, and S. Kumara, "Near linear time algorithm to detect community structures in large-scale networks.," p. 036106, 2007.
35. T. Chakraborty, A. Dalmia, A. Mukherjee, and N. Ganguly, "Metrics for community analysis: A survey," vol. 50, no. 4, pp. 1–37, 2016.
36. S. Gregory, "Finding overlapping communities in networks by label propagation," vol. 12, no. 10, p. 103018, 2010.
37. K. Aydin, M. Bateni, and V. S. Mirrokni, "Distributed balanced partitioning via linear embedding," *CoRR*, vol. abs/1512.02727, 2015.
38. J. Huang and D. Abadi, "LEOPARD: lightweight edge-oriented partitioning and replication for dynamic graphs," *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 540–551, 2016.
39. C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "Fennel: Streaming graph partitioning for massive scale graphs," WSDM '14, Association for Computing Machinery, 2014.
40. J. Nishimura and J. Ugander, "Restreaming graph partitioning: Simple versatile algorithms for advanced balancing," KDD '13, (New York, NY, USA), p. 1106–1114, Association for Computing Machinery, 2013.
41. J. Lin, "Divergence measures based on the shannon entropy," *IEEE Transactions on Information Theory*, vol. 37, pp. 145–151, Jan 1991.
42. J. Leskovec, D. Huttenlocher, and J. Kleinberg, "Signed networks in social media," in *Proc. of the 28th Inter. Conf. on Human factors in computing systems*, p. 1361, 2010.
43. J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," vol. 6, no. 1, pp. 29–123, 2009.
44. R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. of the 29 AAAI*, 2015.
45. R. Zafarani and H. Liu, "Users joining multiple sites: Distributions and patterns," 2014.
46. J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," 2014.
47. L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks," in *Proc. of the 12th ACM SIGKDD Inter. Conf. on Knowledge discovery and data mining*, p. 44, ACM Press, 2006.