DynamicDFEP: A Distributed Edge Partitioning Approach for Large Dynamic Graphs

Chayma Sakouhi
Jendouba University, Tunisia
sakouhichayma@gmail.com

Sabeur Aridhi
Aalto University, Finland
sabeur.aridhi@aalto.fi

Alessio Guerrieri
University of Trento, Italy
a.guerrieri@unitn.it

Salma Sassi
Jendouba University, Tunisia
sassisalma@yahoo.fr

Alberto Montresor
University of Trento, Italy
alberto.montresor@unitn.it

ABSTRACT

Distributed graph processing has become a very popular research topic recently, particularly in domains such as the analysis of social networks, web graphs and spatial networks. In this context, graph partitioning is an important task. Several partitioning algorithms have been proposed, such as DFEP, JABEJA and POWERGRAPH, but they are limited to static graphs only. In fact, they do not consider dynamic graphs in which vertices and edges are added and/or removed. In this paper, we propose a graph partitioning method for large dynamic graphs. We present an implementation of the proposed approach on top of the AKKA framework, and we experimentally show that our approach is efficient in the case of large dynamic graphs.

CCS Concepts

•Theory of computation → Dynamic graph algorithms; Distributed algorithms;

1. INTRODUCTION

In recent years, we have observed an enormous growth in the size of real-world graphs from multiple domains (e.g., social networks and web graphs). Consequently, large-scale distributed/parallel frameworks such as PREGEL [8], GRAPHLAB [7] and GIRAPH [3] have emerged. Each framework introduces a new programming abstraction that allows users to describe their graph algorithms and a corresponding runtime engine that efficiently executes these algorithms on multicore and distributed systems. The common pattern in the programming abstractions of all these frameworks is that they need to to partition the graph over multiple machines in order to allow scalability. The partitioning approaches can be divided into two categories: (1) vertex-based and (2) edge-based. In *vertex partitioning*, each partition is defined by the subgraph induced by a subset of the vertex set of the original graph. The edges that have its incident vertices in different partitions are called *cut edges* and may be considered as ”communication channels” that the nodes will use to coordinate the different partitions [8]. In *edge partitioning*, partitions are defined by graphs induced by subsets of the edge set, where each edge is inside exactly one partition; this time, communication happens through *frontier vertices*, i.e. those that are present in more than one partition [11].

In our work, we are considering the issues of scale and dynamism in the case of edge partitioning approaches. We present DYNAMICDFEP, an edge partitioning method for large dynamic graphs. DYNAMICDFEP can be used to partition large graphs and also to update the partitioning when new edges and vertices are added or removed.

More specifically, our contributions are: (i) we introduce DYNAMICDFEP and its update strategies; (ii) we present an implementation of DYNAMICDFEP on top of AKKA framework, a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications; (iii) we experimentally evaluate the performance of the proposed approach.

The rest of the paper is organized as follows. In Section 3, we present the problem formulation. We present some related works in the following section. Section 4 presents the DYNAMICDFEP framework. Finally, we describe our experimental evaluation in Section 5.

2. RELATED WORK

Many heuristic algorithms and greedy approaches have been designed to meet the challenges of the graph partitioning problem. While most algorithms perform vertex partitioning, a few solve edge partitioning, the topic of this paper.

METIS [5] is a vertex partitioning algorithm based on the Multilevel Graph Partitioning concept [4]. The input graph of METIS goes through three phases: i) the graph is coarsened into a sequence of smaller and smaller graphs, by collapsing pairs of vertices that form a maximal matching; 2) the coarsened graph is partitioned using the edge-cut bisection algorithm [9] 3) the partitions are propagated back in the sequence of graphs and refined to adapt to the original graph. This approach has been adapted for parallel and distributed computing, but will find partitions of worse quality.

JABEJA [10] is a natively decentralized vertex partitioning algorithm. This approach will start by assigning a random partition to each vertex. They will then try to swap their membership with other vertices, trying to optimize a cost function based on the local edge-cut. This process is re-

IDEAS '16, July 11-13, 2016, Montreal, QC, Canada
© 2016 ACM. ISBN 978-1-4503-4118-9/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2938503.2938506

peated in parallel until there are no changes, using simulated annealing to avoid getting stuck in local optima. An inherent advantage of this approach is that the partitions cannot change their size across the computation and therefore their balance can be guaranteed.

POWERGRAPH [1] presents a greedy approach that partitions a stream of edges. For each arriving edge, the algorithm will study two properties for each of the two connecting vertices: the number of edges of that vertex that have still to the processed and the set of partitions that already contain an edge of that vertex. The algorithm will assign the edge to the suitable partition using the following rules:

- if the two sets of partitions intersect, then the edge should be assigned to the smallest partition in the intersection.
- if the two sets of partitions are not empty and do not intersect, then the edge should be assigned to the smallest partition from the vertex with the most edges to be processed.
- if only one of the two sets of partitions is not-empty, then choose the smallest partition from that set.
- if both sets are empty, then assign the edge to the least loaded partition.

This algorithm can produce balanced partitions, but is susceptible to the specific order of arrival of the edges and the quality of its partitions decrease in a distributed setting.

3. PROBLEM FORMULATION

Let $G = (V, E)$ be a undirected graph with $n = |V|$ vertices and $m = |E|$ edges. $d_G(u)$ denotes the degree of the vertex u in G, $N(e)$ the set of two vertices that are connected by the edge e.

The graph edge-partitioning problem can be defined as follows. Let $G = (V, E)$ be an input graph and let k be the number of desired partitions. An *edge partitioning* of G a set of subgraphs $\{G_1 = (V_1, E_1), G_2 = (V_2, E_2), \ldots, G_k = (V_k, E_k)\}$, such that

- $E = \bigcup_{i=1}^{k} E_i$
- $\forall i, j : i \neq j \Rightarrow E_i \cap E_j = \emptyset$
- $\forall i : V_i = \{u | e \in E_i \wedge u \in N(e)\}$

Note that the partitions are made of edges, not vertices, and therefore each edge is part of exactly one partition. All vertices that connect edges of different partitions are called *frontier vertices*. Each frontier vertex will have two or more *replicas*, one for each partition that contains at least an edge of that vertex. A good partitioning provides balanced partitions, therefore we would like the size $|E_i|$ of each partition to be as close as possible to $|E|/k$. As a secondary requirement, we want to keep the number of frontier vertices as small as possible.

In this work, we consider the issues of scale and dynamism for graph partitioning and we aim to update the partitioning when new edges and vertices are added or removed.

3.1 Metrics

In order to evaluate all proposed approaches, we use the following metrics:

- **Runtime (T)**: the running time of the actual implementation of the proposed approach.
- **Number of rounds (R)**: the number of iterations needed to complete the execution of the algorithm.
- **Standard deviation of the partitions' size (SD)**: this metric is used to measure the balance of the graph partitioning. The actual value is defined by:

$$SD = \sqrt{\frac{\sum_{i=1}^{k} \left(\frac{|E_i|}{E/k} - 1\right)^2}{k}}$$

  where $|E|$ is the size of the graph, $|E_i|$ is the size of i^{th} partition, and k is the number of partitions.
- **Frontier vertices (FN)**: The fraction of frontier vertices in the graph
- **Replicas (RN)**: the number of replicas in the partitions. The average number of replicas per vertex is computed as follows:

$$RN = \sum_{i=1}^{k} \frac{|C_{v_j}|}{|V|}$$

  where $|V|$ is the total number of vertices and $|C_{v_j}|$ is the number of partitions of vertex v_j

4. DYNAMIC DFEP

The main idea of our contribution is to find a suitable approach to process changes in a dynamic graph easily and efficiently. Assuming that the graph has already been partitioned, re-running a graph partitioning algorithm from scratch, just because a few vertices or edges have been added, is extremely inefficient. However, new vertices and nodes should still be processed and assigned to the right partitions.

In this section, we first discuss DFEP, the distributed partition algorithm that makes the initial partitioning. We then propose three different heuristics to insert new edges into the already partitioned graph, while keeping partitions balanced as much as possible. Furthermore, we propose a method to handle the deletion operations under the assumption of an already load-balanced partitioning.

4.1 DFEP

DFEP [2] is a previously published distributed partitioning algorithm based on diffusion. While a more complete description of the algorithm is available in the original paper, we can summarize it as follows:

1. We randomly choose a single node for each of the k desired partitions, and give it an initial amount of "funding" associated to that partition.
2. Each node will use its funding to the neighbors to try to "buy" additional edges. The partition will therefore buy the edges that are closer to the randomly chosen nodes and start getting bigger.

3. Since the initial amount of funding is insufficient for the partitions to cover the entire graph, additional funding is assigned to the partitions, in a manner inversely proportional to their size. A small partition (which may have been started far from the center of the graph) will receive more funding and therefore be more likely to grow than a larger partition.
4. Steps 2-3 are repeated until all edges have been bought by a partition.

Since partitions can only diffuse funding through edges that they themselves own, they will buy connected subgraphs of the original graph. The balance of their sizes is caused by the funding mechanism.

4.2 Update strategies

The main objective of this work is to find a suitable solution to process easily and efficiently dynamic graphs. In the next sections, we consider three alternative methods for dealing with the insertion of new edges and one method to deal with deletions.

4.2.1 *Complete Partitioning method (*COM-INS*)*

The simplest method to react to changes to the graph uses a complete re-partitioning for the dynamic graph. This model destroys the old graph partitioning and all further information associated to the assignment and restarts from the scratch by running DFEP.

4.2.2 *Partial Partitioning method (*PART-INS*)*

We suggest that inserting the new edge and change the graph structure without changing the partitioning of the graph can be useful to decrease the computation time. The partial partitioning is designed not to reformulate the partitioning of the graph for every incoming vertex or edge, but just adding vertices and edges to the previous partitions. The program initializes the subgraph and then executes a sequence of iterations to partition the sub-datasets.

This approach applies DFEP starting from the current partitioned graph. Since it starts with funding already distributed in the graph, it will need a very small number of iterations before all the new edges and vertices have been partitioned.

4.2.3 *Unit-Based Insertion (*UB-INS*)*

We note that after running the initial partitioning, our algorithm keeps track of the amount of units that each partition has committed to each vertex and each edge. The amount of units of partition i in vertex v (respectively in edge e) is denoted by $M_i[v]$ (respectively $M_i[e]$). Each edge e maintains $owner(e)$, which denote its partition. The UB-INS method exploits the existing funding in the outgoing vertices of each new edge in order to assign it to the best partition. The set of outgoing vertices of edge e is denoted by $N(e)$. The algorithm used to partition the new edges is described in Algorithm 1.

First, UB-INS computes, for each new edge, the amount of funding committed by each partition. Then, the *maxFunding*() function is applied on each new edge in order to choose the partition that committed the maximum amount of funding to buy the edge. Using this approach, there is no need to re-compute the DFEP partitioning since new edges can be assigned to a partition very quickly.

Algorithm 1: Unit-based Insertion

```
foreach $v \in N(e)$ do
    for $i \leftarrow 1$ to $k$ do
        $M_i[e] \leftarrow M_i[e] + M_i[v]$
$best \leftarrow maxFunding(e)$
$owner(e) \leftarrow best$
```

4.2.4 *Balanced-graph deletion (*BAL-DEL*)*

Deletion of edges or vertices with adversarial choice is a very difficult task, therefore in this section we concentrate our efforts on the case in which removal is random.

- When we remove one node from the graph, it is mandatory to remove all edges directly connected to that node.
- When we delete an edge from the graph, it is mandatory to clean it from the list of affiliated edges for each outgoing vertices. In consequence, the two vertices lose their direct connection.
- After the deletion, the graph may become disconnected; its size decreases.
- Another disadvantage in this case is that the deletion may produce an unbalance in graph partitions.

After deleting vertices and edges, we measure the balance of partitions after the deletion. If the partitions are quite close and still balanced, no action is required, otherwise, we need to repeat the partitioning step using our complete partitioning method.

4.3 System overview

Figure 1 presents the global architecture of our implementation. Many copies of the user program are being executed on a cluster of machines. One of these copies acts as the master. The master decides how many partitions the graph will have and execute the update function in case of addition or deletion of vertices and edges. Furthermore, it is responsible for coordinating the activity of the workers.

Once the system is set up, DFEP will be executed and it will perform a partitioning of the edge set [2]. Once DFEP has completed its execution DYNAMICDFEP will wait for updates.

The incremental computation starts when the master performs an update in the graph structure. In order to update the graph, the master split the data into k partitions, distributes the data to the workers according to the hash method. Each worker receives its portion and starts to update the graph. When the worker terminates send a message to the master, in order to measure if the partitions are still balanced or if the re-partition is required.

5. EXPERIMENTS

We have performed an extensive set of experiments to evaluate the effectiveness and efficiency of our approach on a number of different real and synthetic datasets.

5.1 Experimental environment

We have implemented DYNAMICDFEP and the proposed update strategies on top of the AKKA framework, a toolkit

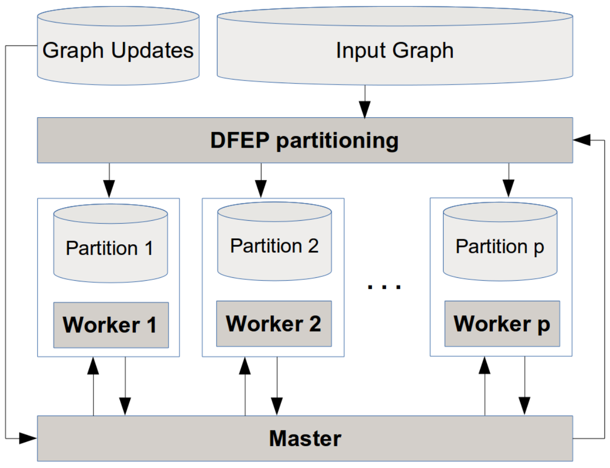


Figure 1: Basic architecture of DynamicDFEP

and runtime for building highly concurrent, distributed, resilient message-driven applications. In order to evaluate the performance of DynamicDFEP, we used a cluster of 17 `m3.medium` instances on Amazon EC2. Each `m3.medium` instance contained 1 virtual 64-bit CPU, 3.75 GB of main memory, and 8 GB of local instance storage.

5.2 Experimental protocol

In order to evaluate the performance and the scalability of DynamicDFEP, we used the same dataset for both the partitioning and the update process. In order to simulate dynamism in each dataset, we use only 90% of the graph in the partitioning step and we insert the remaining 10% in the update step. Each experiment is repeated five times and the numeric results in the following sections consists of the average over all runs.

5.2.1 Experimental data

We used five real-world datasets in our experimental study (see Table 1). The used datasets are made available by the Stanford Large Network Dataset collection [6].

Table 1: Datasets used on the Amazon EC2 cloud

Dataset	♯ Nodes	♯ Edges	⌀	Avg. CC
Facebook-ego	4,039	88,234	8	0.6055
Email-Enron	36,692	183,831	11	0.4970
Amazon	33,4863	925,872	44	0.3967
Youtube	1,134,890	2,987,624	20	0.0808

5.2.2 Experiments

In this section, we empirically study the results of our update strategies.

Accuracy and speedup.

Figure 2 presents the experimental results of DynamicDFEP in terms of execution time.

As shown in Figure 2, the update time is inversely proportional to the number of partitions. We note that the UB-Ins update strategy provide better results than other methods in terms of running time. We also note that the Com-Ins update strategy is faster than Part-Ins method (see Figure 2). This can be explained by the fact that Com-Ins needs more rounds than the Part-Ins update strategy.

We mention that the high efficiency of UB-Ins can be explained by that face that it only needs one round to complete which is convenient for very big graphs. For example, using 16 partitions, Youtube dataset took 3.32 seconds while Part-Ins and Com-Ins need respectively 807.61 seconds and 381.7 seconds to update the partitioning.

Balance .

In order to measure the quality of update, we need to measure the balance of partitions after insertion of new data. First of all, a good partitioning of the graph provides, as an output, partitions with equal size. For our implementation, we used the output of dfep, as an input, to insert a set of data. We thus measure the impact of the update strategy on the balance of partitions.

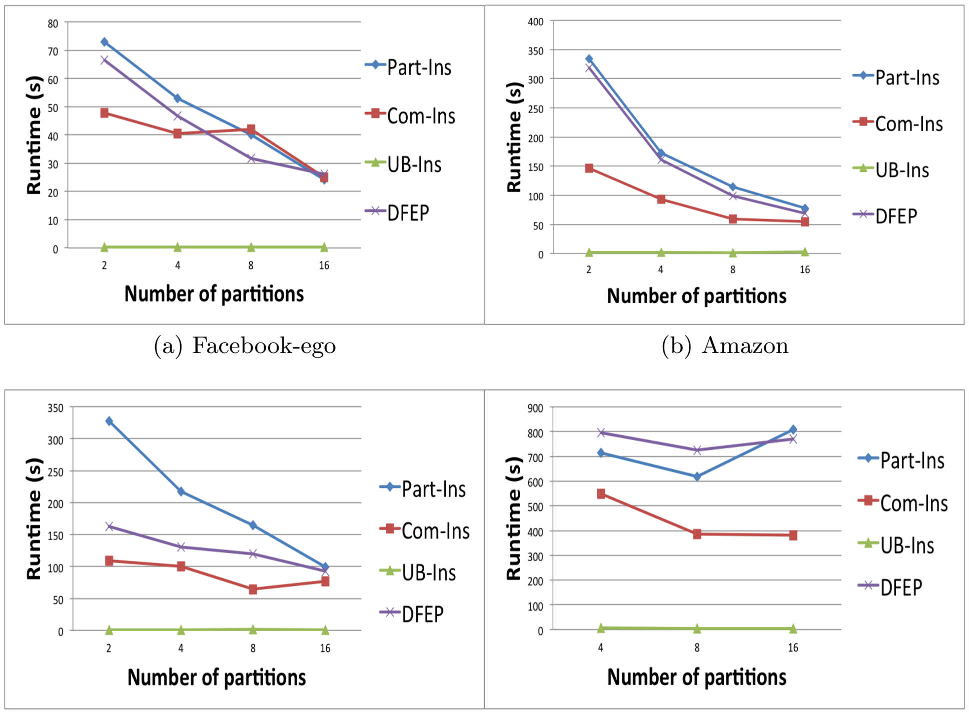
Figure 3 shows the impact of the update strategy on the balance of the graph partitioning. We note that the UB-Ins has a large deviation between the partitions compared to Part-Ins and Com-Ins methods. This is due to the programming model of UB-Ins where new data is bought by the partition that provides more funding. We also note that the Part-Ins method has a less magnitude than the Com-Ins method. This explains that Part-Ins method generates more balanced partitions. In addition, the Part-Ins method has not a big influence on the results given by the previous step of partitioning. For example, on Amazon dataset and using 2 partitions, the standard deviation is 184130.08 whereas after the insertion its value become 192107.28.

Table 2: Experimental results on Facebook-ego dataset

Update strategy	k	Partitioning			Update		
		R	FN	FR	R	FN	RN
Part-Ins	2	164.8	0.18	1.18	2	0.20	1.2
	4	125.4	0.29	1.36	3.2	0.31	1.42
	8	101	0.35	1.54	2	0.36	1.67
	16	58.2	0.46	1.89	2	0.48	2.16
Com-Ins	2	152.2	0.19	1.19	84.2	0.29	1.29
	4	136.5	0.27	1.35	75	0.59	1.71
	8	81.8	0.32	1.48	78.4	0.66	2.01
	16	78.6	0.45	1.93	56.4	0.73	2.75
UB-Ins	2	165	0.14	1.14	1	0.15	1.14
	4	117.2	0.23	1.29	1	0.23	1.29
	8	84.4	0.31	1.47	1	0.31	1.47
	16	69.8	0.47	1.95	1	0.47	1.94

Table 3: Experimental results on Amazon dataset

Update strategy	k	Partitioning			Update		
		R	FN	FR	R	FN	RN
Part-Ins	2	70.2	0.08	1.08	2	0.09	1.09
	4	52	0.12	1.15	2	0.14	1.18
	8	47.2	0.17	1.22	2	0.2	1.26
	16	40.8	0.2	1.28	2	0.22	1.32
Com-Ins	2	70	0.05	1.05	26.6	0.16	1.16
	4	54.2	0.13	1.15	25.2	0.3	1.38
	8	40.6	0.17	1.21	25.4	0.43	1.55
	16	39.8	0.2	1.28	27	0.53	1.76
UB-Ins	2	64.6	0.09	1.09	1	0.09	1.08
	4	45	0.13	1.15	1	0.13	1.14
	8	43.8	0.17	1.22	1	0.16	1.21
	16	39.2	0.19	1.27	1	0.19	1.26



(a) Facebook-ego (b) Amazon

(c) Email-Enron (d) Youtube

Figure 2: Speedup of our update strategies

Table 4: Experimental results on Email-Enron dataset

Update strategy	k	Partitioning			Update		
		R	FN	FR	R	FN	RN
Part-Ins	2	83.8	0.10	1.10	2	0.1	2 1.12
	4	82	0.15	1.23	2	0.2	1.30
	8	91.2	0.18	1.35	2	0.2	1.42
	16	60	0.20	1.46	2	0.23	1.53
Com-Ins	2	83.8	0.12	1.12	36.6	0.18	1.18
	4	69.4	0.16	1.23	38	0.22	1.31
	8	67.4	0.18	1.34	30.4	0.28	1.52
	16	65	0.20	1.46	40.2	0.33	1.75
UB-Ins	2	63.4	0.09	1.09	1	0.09	1.09
	4	64.4	0.15	1.2	1	0.15	1.22
	8	75.4	0.18	1.34	1	0.18	1.34
	16	65.2	0.2	1.47	1	0.19	1.46

Table 5: Experimental results on Youtube dataset

Update strategy	k	Partitioning			Update		
		R	FN	FR	R	FN	RN
Part-Ins	4	31.4	0.09	1.12	2	0.12	1.15
	8	42.6	0.11	1.19	2	0.14	1.23
	16	33	0.12	1.27	2	0.16	1.32
Com-Ins	4	40.4	0.09	1.12	21	0.17	1.23
	8	32.8	0.11	1.20	17	0.19	1.34
	16	34.5	0.12	1.26	16.7	0.22	1.48
UB-Ins	4	39.6	0.09	1.12	1	0.08	1.06
	8	40.8	0.10	1.19	1	0.10	1.13
	16	32.33	0.12	1.26	1	0.11	1.19

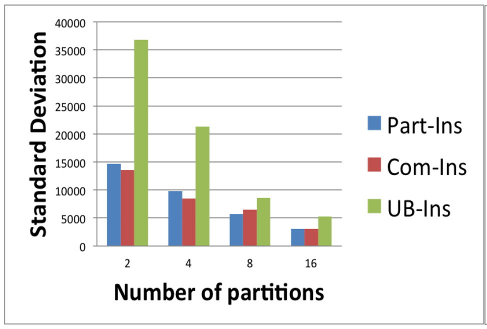
Replicas.

As show in Tables 2, 4, 3 and 5, the number of replicas after the update is greater than before the update, in almost all the datasets. We note that the UB-Ins method gives a tolerant distribution of vertices after the update, while Part-Ins and Com-Ins methods give a severe repartition. This can be explained by the diffusion of units on the vertices coming from different partitions to buy eligible free edges during the update process. However, a vertex can be processed by more than one partition and its eligible edges can be bought by different partitions. As a result, UB-Ins method performs much better than Part-Ins and Com-Ins.

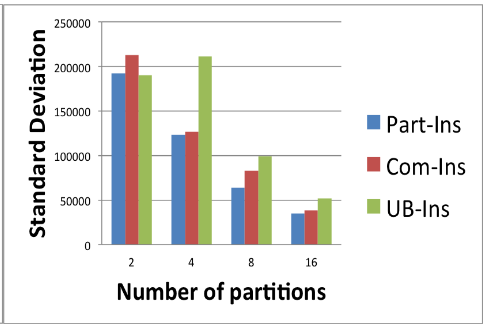
To sum up, we compared the performance and the scalability of our proposal update strategies. In one hand, we concluded that the UB-Ins update strategy is better than other strategies in terms of runtime performance and number of rounds. In the other hand, Part-Ins and Com-Ins strategies show a hight level performance according to the balanced distribution of data over partitions after the update process.

6. CONCLUSIONS

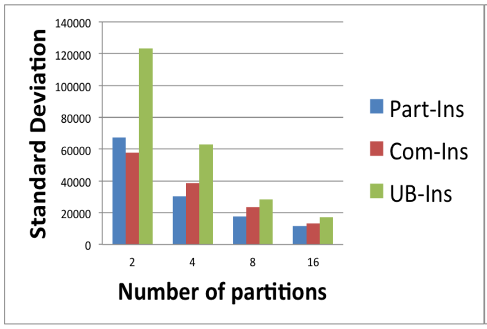
In our work, we are interested to the partitioning of large-scale, dynamic graphs. In our work, dynamism corresponds to the insertion and/or removal of one or more nodes and/or edges. We propose DynamicDFEP, a partitioning method for dynamic graphs. We also propose and a set of update strategies that update the partitioning of a large graph when new edges and nodes are added and/or removed. We implemented the proposed approaches on top of akka, a toolkit and runtime for building highly concurrent, distributed, and resilient message-driven applications. By running experiments on real-world datasets, we have shown that the proposed update strategies are efficient in the case of very large graphs.



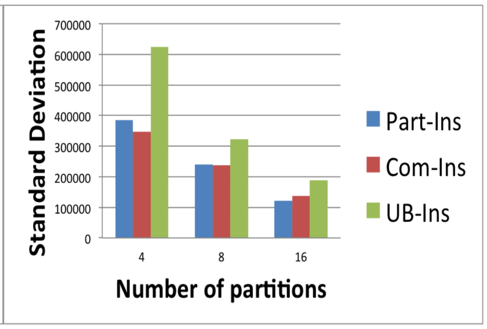
(a) Facebook-ego



(b) Amazon



(c) Email-Enron



(d) Youtube

Figure 3: Impact of the update strategy on the balance of the partitioning

Acknowledgements

This research was partially supported by EIT Digital Project "Sensemaking Service: Entity Linking for Big Linked Data" - Activity Num. 16197 - 2016.

7. REFERENCES

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[2] A. Guerrieri and A. Montresor. DFEP: distributed funding-based edge partitioning. In *Proceedings of the 21st International Conference on Parallel and Distributed Computing*, Europar'15, pages 346–358, 2015.

[3] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in Pregel-like graph processing systems. *Proceedings VLDB Endowment*, 8(9):950–961, May 2015.

[4] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[5] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[6] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, 2014.

[7] Y. Low, D. Bickson, J. Gonzalez, and et al. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proceedings of VLDB Endowment*, 5(8):716–727, Apr. 2012.

[8] 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 *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[9] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, SFCS '91, pages 538–547. IEEE Computer Society, 1991.

[10] F. Rahimian, A. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-Be-Ja: A distributed algorithm for balanced graph partitioning. In *Proceedings of 7th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO'13, pages 51–60. IEEE, 2013.

[11] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.