DFEP: Distributed Funding-based Edge Partitioning

Alessio Guerrieri[1] and Alberto Montresor[1]

DISI, University of Trento, via Sommarive 9, Trento (Italy)

Abstract. As graphs become bigger, the need to efficiently partition them becomes more pressing. Most graph partitioning algorithms subdivide the *vertex set* into partitions of similar size, trying to keep the number of cut edges as small as possible. An alternative approach divides the *edge set*, with the goal of obtaining more balanced partitions in presence of high-degree nodes, such as hubs in real world networks, that can be split between distinct partitions. We introduce DFEP, a distributed edge partitioning algorithm based on the metaphor of currency distribution. Each partition starts from a random edge and expands independently by spending currency to buy neighboring edges. After each iteration, smaller partitions receive an higher amount of currency to help them recover lost ground and reach a similar size to the other partitions. Simulation experiments show that DFEP is efficient and obtains consistently balanced partitions. Implementations on both Hadoop and Spark show the scalability of our approach.

1 Introduction

One of the latest trend in computer science is the emergence of the "big data" phenomena that concerns the retrieval, management and analysis of datasets of extremely large dimensions, coming from wildly different settings.

Although the collected data is often structured, several interesting datasets are unstructured and can be modeled as graphs. An obvious example is the World Wide Web, but there are many other examples such as social network topologies, biological systems or even road networks. While graph problems have been studied since before the birth of computer science, the sheer size of these datasets makes even classic graph problems extremely difficult. Even solving the shortest path problem needs too many iterations to complete when the graph is too big to fit into memory. The big Internet players (such as Google, Yahoo and Facebook) have invested large amount of money in the development of novel distributed frameworks for the analysis of very large graphs and are working on novel solutions of many interesting classic problems in this new context [8,2].

The most common approach to cope with this huge amount of data using multiple processes or machines is to divide the graph into non-overlapping subsets, called *partitions*. Edges between vertices that have been assigned to distinct partitions, called *cut edges* in the literature, act as communication channels between the partitions themselves.

When such partitions are assigned to a set of independent computing nodes (being them actual machines or virtual executors like processes and threads, or even mappers and reducers in the MapReduce model), their size matters: the largest of them must fit in the memory of a single computing entity. A common solution to the problem of optimizing the usage of memory in such cases is to compute partitions that have similar sizes. Dividing the vertex set in equal sized partitions can still lead to an unbalanced subdivision, though: having the same amount of vertices does not imply having the same size, given the unknown distribution of their degrees and the potential high assortativity of some graphs.

In this paper we study a different approach: edges are partitioned into disjoint subsets, while vertices are associated to edges and thus may belong to multiple partitions at the same time. The advantage of such approach is that it makes possible to obtain well-balanced partitions, because the adjacency lists of high-degree nodes may be subdivided among multiple computing nodes. A good load balancing enables the use of a smaller number of computing units.

This type of partitioning can then be used by edge-centric programming models to speed up computation. For example, the Gather-Apply-Scatter model introduced by GraphLab[4] is executed independently on each edge in both the Gather and Scatter phase, and thus needs an efficient edge partitioning. Their system uses Powergraph[4], a one-pass greedy edge partitioning algorithm that can scale to huge graphs and is described in Section 5.

The main contribution of this paper is DFEP, a distributed graph partitioning algorithm that divides the edge set in partitions of similar size. The paper thoroughly evaluates DFEP, using both simulations and then implementation on top of both Hadoop and Spark, using the Amazon EC2 cloud. The experiments show that DFEP is efficient, scalable and obtains consistently good partitions.

2 Edge partitioning

The task of subdividing a graph into partitions of similar size, or *partitioning*, is a classical problem in graph processing, and has many clear applications in both distributed and parallel graph algorithms. Most solutions, from Lin's and Kernighan's algorithm [6] in the 70's to more recent approaches [10], try to solve *vertex* partitioning. This approach, however, may lead to unbalanced partitions, because even if they end up having the same amount of vertices, an unbalanced distribution of edges may cause some subgraphs to be much larger than others. Approaching the problem from an edge perspective, thus, may bring us to interesting and practical results.

Given a graph $G = (V, E)$ and a parameter K, an *edge partitioning* of G subdivides all edges into a collection $E_1, \ldots, E_K$ of non-overlapping edge partitions:

$$E = \cup_{i=1}^{K} E_i \qquad \forall i, j : i \neq j \Rightarrow E_i \cap E_j = \emptyset$$

The i-th partition is associated with a vertex set V_i, composed of the end points of its edges:

$$V_i = \{u : (u, v) \in E_i \vee (v, u) \in E_i\}$$

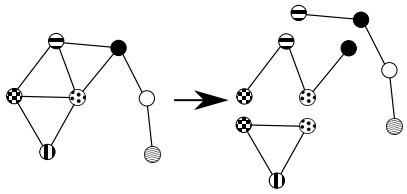


Fig. 1: Edge partitioning example: each edge appears in only one partition, while frontier vertices may appear in more than one partition

The edges of each partition, together with the associated vertices, form the subgraph $G_i = (V_i, E_i)$ of G, as illustrated in Figure 1.

The *size* of a partition is proportional to the amount of edges and vertices $|E_i| + |V_i|$ belonging to it. Given that each edge $(u, v) \in E_i$ contributes with at most two vertices, $|V_i| = O(|E_i|)$ and the amount of memory needed to store a partition is strictly proportional to the number of its edges. This fact can be exploited to distribute fairly the load among machines. Vertices may be replicated among several partitions, in which case are called *frontier vertices*. We denote with $F_i \subseteq V_i$ the set of vertices that are frontier in the i-th partition.

3 Distributed Funding-based Edge Partitioning

The properties that a "good" partitioning must possess are the following:

- **Balance**: partition sizes should be as close as possible to the average size $|E|/K$, where K is the number of partitions, to have a similar computational load in each partition. Our main goal is to minimize the size of the largest partition.
- **Communication efficiency**: given that the amount of communication that crosses the border of a partition depends on the number of its frontier vertices, the total sum $\sum_{i=1}^{K} |F_i|$ must be reduced as much as possible.
- **Connectedness**: the subgraphs induced by the partitions should be as connected as possible. This is not a strict requirement and later in this section we illustrate a variant of our algorithm that relax it.

Balance is the main goal; it would be simple to just split the edges in K sets of size $\approx |E|/K$, but this could have severe implications on communication efficiency and connectedness. The approach proposed here is thus heuristic in nature and provides an approximate solution to the above requirements.

Since the purpose is to compute the edge partitioning as a preprocessing step to help the analysis of very large graphs, we need the edge partitioning algorithm to be distributed as well. As with most distributed algorithms, we are mostly interested in minimizing the amount of communication steps needed to complete the partitioning.

Ideally, a simple solution could work as follows: to compute K partitions, K edges are chosen at random and each partition grows around those edges.

Then, all partitions take control of the edges that are *neighbors* (i.e., they share one vertex) of those already in control and are not taken by other partitions. All partitions will incrementally get larger and larger until all edges have been taken. Unfortunately, this simple approach does not work well in practice, since the starting position may greatly influence the size of the partitions. A partition that starts from the center of the graph will have more space to expand than a partition that starts from the border and/or very close to another partition.

$d(v)$	degree of vertex v
$E(v)$	edges incident on vertex v
$V(e)$	vertices incident on edge e
$M_i[v]$	units of partition i in vertex v
$M_i[e]$	units of partition i in edge e
E_i	edges bought by partition i
$owner[e]$	the partition that owns edge e

Table 1: Notation

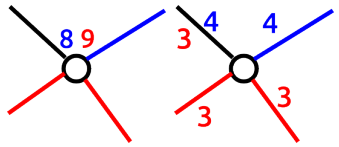


Fig. 2: Step 1

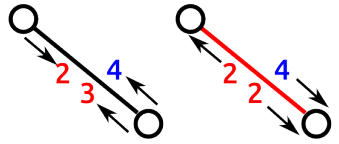


Fig. 3: Step 2

Algorithm 1: DFEP Init
Executed by the coordinator

```
foreach edge e ∈ E do
    owner[e] = ⊥
for i = 1 to K do
    v ← random(V)
    M_i[v] = |E|/K
```

Algorithm 2: DFEP Step 1
Executed at each vertex v

```
for i = 1 to K do
    if M_i[v] > 0 then
        eligible = ∅
        foreach e ∈ E(v) do
            if owner[e] = ⊥ or
            owner[e] = i then
                eligible = eligible ∪ {e}
        foreach e ∈ eligible do
            M_i[e] =
            M_i[e] + (M_i[v]/|eligible|)
        M_i[v] = 0
```

Algorithm 3: DFEP Step 2
Executed at each edge e

```
best = argmax_p(M_p(e))
if owner[e] = ⊥ and M_best(e) ≥ 1 then
    owner[e] = best
    M_best[e] = M_best[e] − 1
for i = 1 to K do
    if owner[e] = i then
        foreach v ∈ N(e) do
            M_i[v] = M_i[v] + M_i[e]/2
    else
        S = vertices that funded
        partition i in e
        foreach v ∈ S do
            M_i[v] = M_i[v] + M_i[e]/|S|
    M_i[e] = 0
```

Algorithm 4: DFEP Step 3
Executed by the coordinator

```
AVG = ∑_{i∈[1...K]}(|E_i|)/K
for i = 1 to K do
    funding = min(10, AVG/E_i)
    foreach v ∈ V do
        if M_i(v) > 0 then
            M_i(v) = M_i(v) + funding
```

To overcome this limitation, we introduce DFEP (Distributed Funding-based Edge Partitioning), an algorithm based on concept of "buying" the edges through an amount of *funding* assigned to partitions. Initially, each partition is assigned the same amount of funding and an initial, randomly-selected vertex. The algorithm is then organized in a sequence of *rounds*. During each round, the parti-

tions try to acquire the edges that are neighbors to those already taken, while a coordinator monitors the sizes of each partition and sends additional units of funding to the smaller ones, to help them overcome their slow start.

Table 1 contains the notation used in the pseudocode of the algorithm. For each vertex and edge we keep track of the amount of units that each partition has committed to that vertex or edge. Algorithm 1 presents the code executed at the initialization step: each partition chooses a vertex at random and assigns all the initial units to it. The edges are initialized as unassigned. Each round of the algorithm is then divided in three steps. In the first step (Algorithm 2), each vertex propagates the units of funding to the outgoing edges. For each partition, the vertex can move its funding only on edges that are free or owned by that partition, dividing the available units of funding equally among all these eligible edges. During the second step (Algorithm 3), each free edge is bought by the partition which has the most units committed in that edge and the units of funding of the losing partitions are sent back in equal parts to the vertices that contributed to that funding. The winning partition loses an unit of funding to pay for the edge and the remaining funding is divided in two equal parts and sent to the vertices composing the edge. In the third step (Algorithm 4), each partition receives an amount of funding inversely proportional to the number of edges it has already bought. This funding is distributed between all the vertices in which the partition has already committed a positive amount of funding. Two examples are illustrated in Figures 2-3. The red and blue color represents partitions, while black edges are still free.

DFEP creates partitions that are connected subgraphs of the original graph, since currency cannot traverse an edge that has not been bought by that partition. It can be implemented in a distributed framework: both Step 1 and Step 2 are completely decentralized; Step 3, while centralized, needs an amount of computation that is only linear in the number of partitions.

In our implementation, the amount of initial funding is equal to what would be needed to buy an amount of edges equal to the optimal sized partition. A smaller quantity would not decrease the precision of the algorithm, but it would slow it down during the first rounds. The cap on the units of funding to be given to a small partition during each round (10 units in our implementation) avoids the over-funding of a small partition during the first rounds.

In a distributed setting the algorithm will follow the Bulk Synchronous Processing model: each machine receives a subset of the graph, executes Step 1 on each of its vertices independently, sends money to the correct edges (that may be on other machines), wait for the other machines to finish Step 1, and executes Step 2. Step 3 must be executed by a coordinator, but the amount of computation is minimal since the current sizes of the partitions can be computed via aggregated counting by the machines. Once the coordinator has computed the amount of funding for each partition, it can send this information to the machines that will apply it independently before Step 1 of the successive iteration. If the coordinator finds that all edges have been assigned, it will terminate the algorithm.

3.1 Variant: DFEPC

If the diameter is very large, there is the possibility that a poor starting vertex is chosen at the beginning of the round. A partition may be cut off from the rest of the graph, thus creating unbalanced partitions. A possible solution for this problem involves adding an additional dynamic, at the cost of losing the connectedness property.

A partition is called *poor* at round i if its size is less than $\frac{\mu}{p}$, with μ being the average size of partitions at round i and p being an additional parameter; otherwise, it is called *rich*. A poor partition can commit units on already bought edges that are owned by rich partitions and try to buy them. This addition to the algorithm allows small partitions to catch up to the bigger ones even if they have no free neighboring edges and results in more balanced partitions in graphs with larger diameter.

4 Results

We evaluated our algorithms with both simulations (experiments repeated 100 times) and actual implementations (experiments repeated 20 times. The metrics considered to evaluate DFEP in our simulation engine are the following:

- **Rounds**: the number of rounds executed by DFEP to complete the partitioning. This is a good measure of the amount of synchronization needed and can be a good indicator of the eventual running time in a real world scenario.
- **Balance**: Each partition should be as close as possible to the same size. To obtain a measure of the balance between the partitions we first normalize the sizes, so that a partition of size 1 represents a partition with exactly $|E|/K$ edges. We then measure the standard deviation of the normalized sizes.
- **Communication costs**: Each partition will have to send a message for each of its frontier vertices, to share their state with the other partitions. We thus use the frontier nodes to estimate the communication costs: $M = \sum_{i=1}^{K} F_i$.

Since the simulation engine is not able to cope with larger datasets, we used different datasets for the experiments in the simulation engine and the real world experiments. For both types of datasets we list the size of the graphs, the diameter D, the clustering coefficient CC and the clustering coefficient RCC of a random graph with the same size.

The first four datasets in Table 2 have been used in the simulation engine. ASTROPH is a collaboration network in the astrophysics field, while EMAIL-ENRON is an email communication network from Enron. Both datasets are small-world, as shown by the small diameter. The USROADS dataset is a road networking the US, and thus is a good example of a large diameter network. Finally, WORDNET is a synonym network, with small diameter and very high clustering coefficient.

The three larger graphs are used in our implementation of DFEP on the Amazon EC2 cloud. DBLP is the co-authorship network from the DBLP archive,

#	Name	$\|V\|$	$\|E\|$	D	CC	RCC
1	ASTROPH	17903	196972	14	1.34×10^{-1}	1.23×10^{-3}
2	EMAIL-ENRON	33696	180811	13	3.01×10^{-2}	3.19×10^{-4}
3	USROADS	126146	161950	617	1.45×10^{-2}	2.03×10^{-5}
4	WORDNET	75606	231622	14	7.12×10^{-2}	8.10×10^{-5}
5	DBLP	317080	1049866	21	1.28×10^{-1}	2.09×10^{-5}
6	YOUTUBE	1134890	2987624	20	2.08×10^{-3}	4.64×10^{-6}
7	AMAZON	400727	2349869	18	5.99×10^{-2}	2.93×10^{-5}

Table 2: Datasets used in the simulation engine (1-4) and EC2 (5-7)

YOUTUBE is the friendship graph between the users of the service while AMAZON is a co-purchasing network of the products sold by the website.

All the networks have been taken from the SNAP graph library [7] and cleaned for our use, by making directed edges undirected and removing disconnected components.

4.1 Simulations

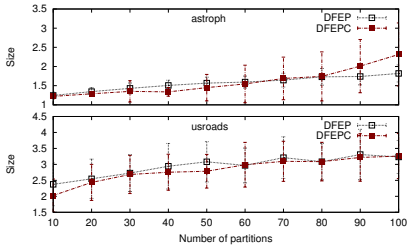
Figure 4 shows the performance of the two versions of DFEP against the parameter K, in the ASTROPH and USROADS datasets. As expected, the larger the number of partitions, the larger is the variance between the sizes of those partitions and the amount of messages that will have to be sent across the network. The rounds needed to converge to a solution go down with the number of partitions, since it will take less time for the partitions to cover the entire graph.

The diameter of a graph is a strong indicator of how our proposed approach will behave. To test DFEP on graphs with similar characteristics but different diameter we followed a specific protocol: starting from the USROADS dataset (a graph with a very large diameter) we remapped random edges, thus decreasing the diameter. The remapping has been performed in such a way to keep the number of triangles as close as possible to the original graph, to avoid introducing bias in the experiment by radically changing the clustering coefficient.

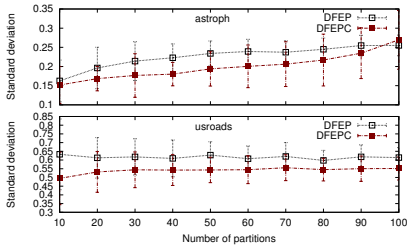
Figure 5 shows that changing the diameter leads to completely different behaviors. The size of the largest partitions and the standard deviation of partitions size rise steeply with the growth of the diameter, since in a graph with higher diameter the starting vertices chosen by our algorithm affect more deeply the quality of the partitioning. As expected, the number of rounds needed by DFEP to compute the partitioning also rise linearly with the diameter. Since the partitions will be more interconnected, the amount of messages sent across the network will decrease steeply with a larger diameter. Our variant of DFEP is able to cope well also in case of graphs with large diameter.

Finally, we compare the two version of DFEP against JaBeJa [9] and PowerGraph [4]. Since JaBeJa is a vertex-partitioning algorithm, its output has been converted into an edge-partitioning.

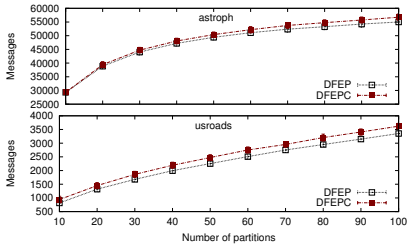
PowerGraph processes the graph one edge at a time, assigning it to the best partition according to which partitions already contain the nodes of the current edge. The sequential version of the algorithm needs at each step complete



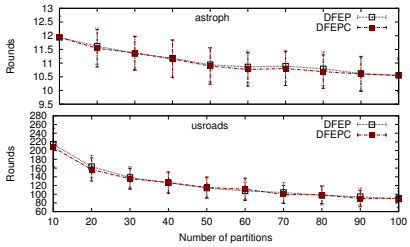
(a) Size of the largest partition



(b) Std. Dev. of partition sizes



(c) Communication cost



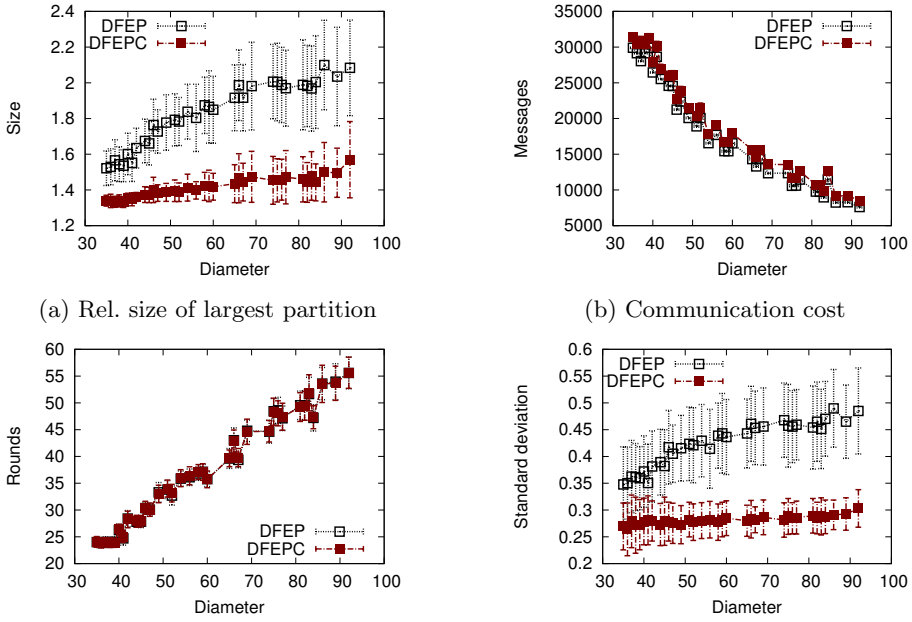
(d) Rounds to converge

Fig. 4: Behavior of DFEP and DFEPC with varying values of K

knowledge of the choices of the previous iteration. The authors also illustrate a version called "Oblivious PowerGraph" in which each process behaves independently on a subset of the edges. The quality of the partitioning thus depends on the number of independent processes used. In our comparison, we used both the centralized version (labeled "PowerGraph") and the oblivious version (labeled "Oblivious PowerGraph"). In the oblivious version, we tested the algorithm by simulating two distinct processes.

Both PowerGraph versions create remarkably balanced partitions and are extremely fast, since they work in a single pass over the graph. On the downside, their partitions are less connected than DFEP and thus incur in more communication costs.

Figure 6 shows the experimental results over 100 samples, on the four different datasets. A pattern can be discerned: the algorithms have wildly different behaviors in the small world dataset than in the road network. In the small world datasets our approaches results in more balanced partitions, while needing less rounds to converge than JaBeJa. In the USROADS dataset JaBeJa creates more balanced partitions, but with a communication costs that is roughly ten times higher. This result shows the importance of creating partitions that are as much connected as possible. Powegraph instead gets balanced, but not very connected partitions in all cases. With the oblivious version of the algorithm the quality degrades, since the approach will obtain a partitioning of worse quality the higher the number of the processes that participate in the computation.



(a) Rel. size of largest partition

(b) Communication cost

(c) Rounds needed to converge

(d) Std. Dev. of partition sizes

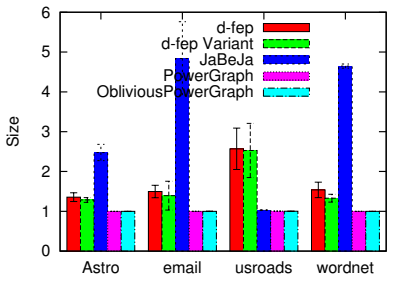
Fig. 5: Behavior of DFEP and DFEPC with varying diameter ($K = 20$)

Since JaBeJa uses simulated annealing to improve the candidate solution, the number of round needed is mostly independent from the structure of the graph. As shown in Figure 5 the number of rounds DFEP needs depend mostly from the graph diameter. Both versions of PowerGraph work in a single pass over the edge set, and therefore is a better choice if the amount of computation needed after the partitioning step is not large enough to warrant a more precise partitioning.

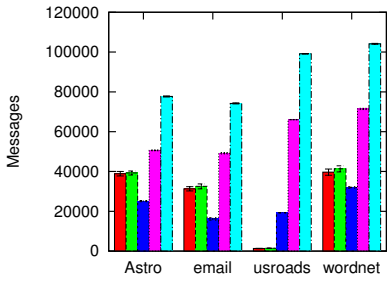
4.2 Experiments in EC2

DFEP has been implemented in both Apache Hadoop in the MapReduce model and in Spark/Graphx, and have been tested over the Amazon EC2 cloud. All the experiments have been repeated 20 times on *m1.medium* machines.

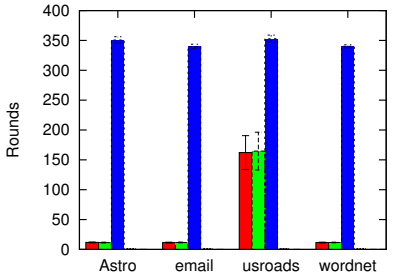
It was not possible to implement DFEP in Hadoop using a single Map-Reduce round for each iteration while keeping exactly the same structure as in the pseudocode. Each instance of the Map function is executed on a single vertex, which will output messages to its neighbor and a copy of itself. Each instance of the Reduce function will receive a vertex and all the funding sent by the neighbors on common edges. The part of the algorithm that should be executed on each edge is instead executed by both its neighboring vertices, with special care to make sure that both executions will get the same results to avoid inconsistencies in the graph. This choice, which sounds counterintuitive, allows us to use



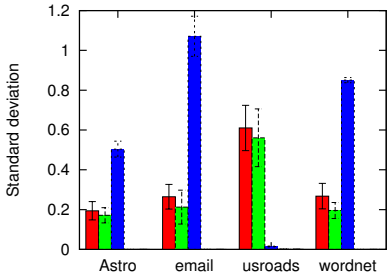
(a) Rel. size of largest partition



(b) Communication cost

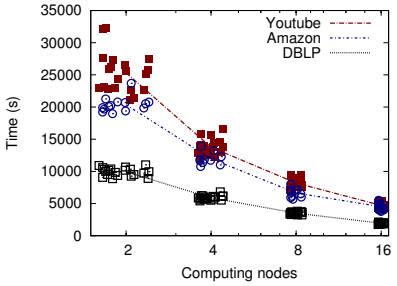


(c) Rounds needed to converge

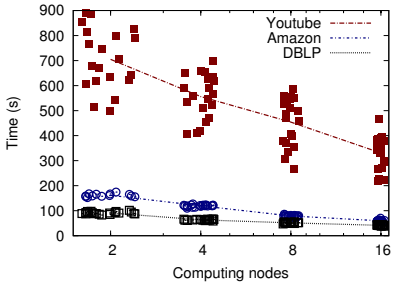


(d) Std. Dev. of partition sizes

Fig. 6: Comparison between DFEP, DFEPC, JaBeJa and PowerGraph ($K = 20$)



(a) Hadoop



(b) Spark

Fig. 7: Speedup of real implementation of DFEP in the amazon cloud

a single Map-Reduce round for each iteration of the algorithm, thus decreasing the communication and sorting costs inherent in the MapReduce model.

Figure 7a presents the scalability results, when run with the datasets in Table 2, with $K = 20$. The algorithm scales with the number of computing nodes, with a speedup larger than 5 with 16 nodes instead of 2.

Our Spark/Graphx implementation of DFEP is still unstable, and thus, while faster, it is not able to reach the scalability of the Hadoop implementation. Figure 7b shows a speedup of just 2 with 16 nodes instead of 2 nodes, with a very large variance.

5 Related work

The literature on graph partitioning is huge, but given that edge partitioning has not been studied in equal depth, we will mostly focus on the different approaches developed to solve vertex graph partitioning. The edge partitioning problem can be reduced to the vertex partitioning problem by using the line graph of the original graph, but the massive increase in size makes this approach infeasible.

In both versions, the partitioning problem is not only NP-complete, but even difficult to approximate [1]. Most work in this field are thus heuristics algorithms with no guaranteed approximation rate. Kernighan and Lin developed the most well-known heuristic algorithm for binary graph partitioning in 1970 [6]. At initialization time, each vertex in the network is randomly assigned to one of two partitions and the algorithm tries to optimize the vertex cut by exchanging vertices between the partitions. This approach has been later extended to run efficiently on multiprocessors by parallelizing the computation of the scoring function used to choose which vertices should be exchanged [3].

METIS [5] is a more recent and highly successful project that uses a multi-level partitioning approach to obtain very high quality partitions. The graph is coarsened into a smaller graph, which is then partitioned and the solution is then refined to adapt to the original graph. An effort to create a parallelizable version of the program has lead to P-METIS, a version built for multicore machines. The quality of the partitions obtained with this approach does not seem to be of the same quality than the centralized version, as expected.

The presence of additional constraints has driven the research field towards more specialized algorithms. For example, in the streaming scenario it is infeasible to use the classical partitioning algorithm, since the data is continuously arriving. A greedy algorithm that assign each incoming vertex to a partition has been proposed [10] and computes partitions of only slightly less quality than most centralized algorithms.

The two algorithms selected for our comparison are JaBeJa [9] and Powergraph [4]. JaBeJa is a completely decentralized partitioning algorithm based on local and global exchanges. Each vertex in the graph is initially mapped to a random partition. At each iteration, it will try to exchange its mapping with one of its neighbor or with one of the random vertices obtained via a peer selection algorithm, if the exchange decreases the vertex cut size. An additional layer of simulated annealing decrease the likelihood of returning to a local minima. JaBeJa is similar in approach to Kernighan and Lin's algorithm, but moves the choices from the partition level to the vertex level, greatly increasing the possibility for parallelization.

Powergraph instead uses a greedy approach, processing and assigning each edge before moving to the next. It keeps in memory the current sizes of each partition and, for each vertex, the set of partitions that contain at least one edge of that vertex. If both endpoints of the current edge are already inside one common partition, the edge will be added to that partition. If they have no partition in common, the node with the most edges still to assign will choose one of its partitions. If only one node is already in a partition, the edge will be

assigned to that partition. Otherwise, if both nodes are free, the edge will be assigned to the smallest partition. This heuristic can be run independently on N subsets of the edge set to parallelize the workload, at the cost of lower quality partitions.

6 Conclusions

This paper presented DFEP, an heuristic distributed edge partitioning algorithm based on a simple funding model. Our experimental results, obtained through simulation and through an actual deployment on an Amazon EC2 cluster, show that DFEP scales well and is able to obtain balanced partitions.

As future work, we are working on an efficient Spark implementation of DFEP, to allow us to partition larger graphs and analyze the scalability of our approach. We will study how does the algorithm behaves in presence of dynamism (such as addition and deletion of edges) and how to use external information about nodes and edges to obtain a better partitioning.

References

1. K. Andreev and H. Räcke. Balanced graph partitioning. In *Proc. of the* 16^{th} *annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 120–124. ACM, 2004.
2. A. Bialecki, M. Cafarella, D. Cutting, and O. O'Malley. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 2005.
3. J. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *Int. Journal of Parallel Programming*, 16(6):427–449, 1987.
4. 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 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.
5. G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minnesota, 1995.
6. B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System technical journal*, 1970.
7. J. Leskovec. Stanford large network dataset collection. *URL http://snap. stanford. edu/data/index. html*, 2011.
8. G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. of the 2010 international conference on Management of data*, pages 135–146. ACM, 2010.
9. F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *In Proc. of 7th Int. Conf. on Self-Adaptive and Self-Organizing Systems (SASO'13)*, pages 51–60. IEEE, 2013.
10. C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. *Microsoft Research*, 2012.