Which Metrics for Vertex-Cut Partitioning?

Hlib Mykhailenko, Giovanni Neglia, Fabrice Huet

Which Metrics for Vertex-Cut Partitioning?

Hlib Mykhailenko*[1], Giovanni Neglia†[1], and Fabrice Huet‡[2]

[1]Université Côte d'Azur, Inria
[2]Université Côte d'Azur, CNRS, I3S

November 23, 2016

Abstract

In this paper we focus on vertex-cut graph partitioning and we investigate how it is possible to evaluate the quality of a partition before running the computation. To this purpose we scrutinize a set of metrics proposed in literature. We carry experiments with the widely-used framework for graph processing Apache GraphX and we perform an accurate statistical analysis. Our preliminary experimental results show that communication metrics like vertex-cut and communication cost are effective predictors on most of the cases.

1 Introduction

Many distributed architectures have been proposed to process large graphs, like for example GraphX [16], Pregel [13], PowerGraph [8], and GraphLab [12]. Before starting the actual computation, the graph is partitioned among the executors. Each executor then processes only a subset of the graph, but it needs to periodically share intermediate-computation results with the other executors. Partitioning affects then the total execution time in two different ways. First, the executor computation time is related to the size of the subset it needs to process. Second, the communication time depends on how much overlap there is among the different subsets. As a trivial example, if the graph has different components and each executor gets assigned one of them, executors can in many cases work almost independently. While partitioning can significantly affect the total execution time, finding the best partition for a given graph, cluster setting and application task can be itself a hard computation problem. In practice, existing partitioners rely on heuristics to find good enough trade-offs between

*hlib.mykhailenko@inria.fr
†giovanni.neglia@inria.fr
‡fabrice.huet@unice.fr

balance (subsets should have roughly the same size), and communication (subsets should be as disjoint as possible).

A classic way to distribute the graph is the **edge-cut** approach where each vertex is assigned to a different partition. An edge is *cut*, if its vertices belong to two different partitions. In most applications, the executor storing one of the two vertices, say it the source, will need to be updated when the destination value is modified. Edge-cut partioners try then to minimize the number of edges cut. More recently, **vertex-cut** partitioning has been advocated as a better approach to process graphs with a power-law degree distribution (very common in real-world datasets) [8]. In this case, edges are mapped to partitions and vertices are cut if their edges happen to be assigned to different partitions. The effect can be qualitatively explained with the presence in a power-law graph of hubs, i.e. nodes with degree much larger than the average. In an edge-cut partition, attributing a hub to a given partition easily leads to i) computation unbalance, if its neighbors are also assigned to the same partition, or ii) to a large number of edges cut and then strong communication requirements. A vertex-cut partitioner may instead achieve a better trade-off, by cutting only a limited number of hubs. Analytical support to these findings is in [6]. For this reason many new graph computation frameworks, like GraphX [16] and PowerGraph [8], rely on vertex-cut partitioners.

In this paper we focus on vertex-cut partitioning and we investigate how it is possible to evaluate the quality of a partition before running the computation. Answering this question would be useful both for the data analysts who need to choose the partitioner appropriate for their graphs and for developers who aim at proposing more efficient partitioners. In this paper we make a step towards providing an answer by analysing a set of metrics proposed in literature to evaluate the intrinsic quality of a partition. To this purpose we carry experiments with the widely-used framework for graph processing Apache GraphX and perform an accurate statistical analysis.

The paper is organized as follows. Apache GraphX is presented in Section 2 together with the metrics for graph partitioning which we consider in our study. In Section 3 we discuss the graph partitioning algorithms implemented in GraphX. In Section 4 we discuss our statistical methodology to study the dependency between the execution time of different graph-processing algorithms and partitioning metrics. We discuss the experimental setup and illustrate the results in Section 5. Finally, we conclude in Section 6.

The source code is available here [3].

2 Background

We start this section by presenting Apache GraphX. Then, we discuss the metrics used to evaluate the quality of graph partitions.

2.1 Apache GraphX

Apache Spark [1] is a large-scale distributed framework for data processing. It relies on the concept of Resilient Distributed Dataset [17] (RDD) which is an immutable distributed collection mainly stored in RAM memory. The basic API provides methods for mapping and reducing RDDs.

Apache GraphX [16] is a widely-used Spark library for graph processing. GraphX programs are executed as Spark jobs. The framework relies on a vertex-cut partitioning approach where a vertex can be split among several partitions. There exist many different algorithms for choosing where to map a given edge. We discuss in Section 3 six such algorithms implemented in GraphX.

By default, GraphX considers all input graphs as directed ones, but then an algorithm can work on its input as if it were undirected. This is for example the case of the `Connected Components` algorithm built-in in GraphX. Other algorithms, like GraphX implementation of `PageRank` [14], instead assume the input is directed. In this case, if one wants to process an undirected graph, he/she can modify directly the code or pre-processing the input and replace each undirected edge by two edges with opposite directions.

GraphX is executed on a cluster with one master machine and N_m machines. A graph is stored in GrahpX as two RDDs: a vertex RDD and an edge RDD. The vertex RDD stores the ids of the vertices and their values. The edge RDD stores source ids of the edges, destination ids of the edges, and the values assigned to the edges. Each of these RDDs is split in N subsets each assigned to a different executor[1]. The vertex RDD is always partitioned by a hash function based on vertex ids, while the edge RDD is partitioned using a user-specified partitioner. Note that the number of partitions N can be different from the number N_m of machines (for example a machine can be assigned multiple executors). GraphX distributes the N partitions among the machines in a round robin fashion.

2.2 Metrics for partitioning

We introduce here some additional notation to simplify the definitions.

Let us denote the input directed graph as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of vertices and $\mathcal{E}$ is a set of edges. Vertex-cut partitioning splits the set of edges into N disjoint subsets, $E_1, E_2, \ldots, E_N$. With some abuse of terminology, we also say that a vertex belongs to the (edge) partition E_i, if it is the source or the destination of an edge in E_i. Using this convention, an edge can only belong to one partition, but a vertex is at least in one partition and at most in N partitions. Let $V(E_i)$ denote the set of vertices that belong to partition E_i. The vertices that appear in more than one partition are called frontier vertices. Each frontier vertex has then been cut at least once. $F(E_i)$ denotes the set of frontier vertices that are inside partition E_i. $\bar{F}(E_i)$ is the set of vertices in partition E_i that were not cut, and this set equals to $\bar{F}(E_i) = V(E_i) - F(E_i)$.

[1]It is also possible to assign multiple partitions to the same executor, but we do not consider this possibility.

There exist several metrics to evaluate the quality of a graph partition. In this paper we want to understand how well they fulfill this goal, i.e. to which extent they can be used to predict the final performance. In our study we considered six metrics which were used by the authors of graph partitioning algorithms presented in [15] and [9]. The first three of them aim to quantify how homogeneous are the sizes of the subsets in the partition and then how *balanced* are the computation tasks across all the executors. The last three instead target the communication cost, by considering how many vertices are shared across the different subsets.

`Balance` (denoted as BAL) is the ratio between the maximum number of edges in one partition and the average number of edges in all partitions, $\text{BAL} = \frac{\max_{i=1,\dots N}(|E_i|)}{|E|/N}$.

`NSD` is the normalized standard deviation of partition size, $\text{NSD} = \sqrt{\sum_{i=1}^{N} \left(\frac{|E_i|}{|E|/N} - 1\right)^2 \frac{1}{N}}$.

`Largest partition` (denoted as LP) shows the number of vertices in the largest partition of a graph, $\text{LP} = \max_{i=1,\dots N} |V(E_i)|$.

`Vertex-cut` (denoted as VC) shows the number of vertices that were cut during graph partitioning, $\text{VC} = |\mathcal{V}| - \sum_{i=1}^{N} |\bar{F}(E_i)|$.

`Replication factor` (denoted as RF) is the ratio between the total number of vertices in a partitioned graph and the original graph, $\text{RF} = \sum_{i=1}^{N} |V(E_i)| \frac{1}{|\mathcal{V}|}$.

`Communication cost` (denoted as CC) is the total number of frontier vertices among all the partitions, $\text{CC} = \sum_{i=1}^{N} |F(E_i)|$.

It should be noticed that RF, CC, and VC metrics are linear combinations of each others: $\text{RF} = \sum_{i=1}^{N} |V(E_i)| \frac{1}{|\mathcal{V}|} = \sum_{i=1}^{N} (|\bar{F}(E_i)| + |F(E_i)|) \frac{1}{|\mathcal{V}|} = 1 - \frac{\text{VC}}{|\mathcal{V}|} + \frac{\text{CC}}{|\mathcal{V}|}$. For this reason it is sufficient to consider only 2 of them. We selected VC and CC. [2]

GraphX does not have a built-in function which provides values of these metrics, this is why we have implemented new GraphX functions to compute them [3].

3 GraphX partitioners

Six partitioning algorithms are considered in this work and described in this section. The first four partitioners given below are the GraphX built-in ones. The fifth (`HybridCut`) was proposed in [7] and implemented for GraphX by Larry Xiao [2] together with its variant `HybridCutPlus`.

[2]We decided to sacrifice RF because including RF leads easily to negative coefficients in the linear regression model.

`RandomVertexCut` (denoted as RVC) partitioner randomly assigns edges to partitions to achieve good balance (with high probability). In particular it computes the hash value for each pair (source vertex id, destination vertex id). The hash space is partitioned in N sets with the same size. As a result, in a multigraph all edges with the same source and destination vertices are assigned to the same partition. On the contrary, two edges among the same nodes but with opposite directions belong in general to different partitions.

`CanonicalRandomVertexCut` (denoted as CRVC) algorithm is similar to the previous partitioner but in this case the two ids are ordered before calculating the hash value. Hence, all edges between two vertices are allocated to the same partition, regardless of their direction.

`EdgePartition1D` (denoted as EP1D) targets a small replication factor by keeping all outgoing edges from one vertex in a single partition.[3] More precisely, the algorithm assigns each edge based on the hash value of its source vertex id.

`EdgePartition2D` (denoted as EP2D) guarantees a bound on the replication factor metric and it can provide also good balance if the number of partitions is appropriately chosen. The algorithm splits a set of vertices in $M = \lceil\sqrt{N}\rceil$ subsets $S_1, S_2, \ldots, S_M$ using a hash function. Then it is possible to partition the adjacency matrix in M^2 blocks, where the block (i, j) contains all edges with a source in S_i and a destination in S_j. Finally, the M^2 blocks are assigned to partitions in a round robin fashion. A vertex can then appear in at most $2M-1 \leq 2(\sqrt{N}+1)$ partitions. If N is chosen to be a square number, partitions will be on average well balanced.

`HybridCut` (denoted as HC) tries to reduce the number of frontier vertices without sacrificing the balance. The basic idea is that hubs will be cut in any case to avoid unbalance, while the edges of the nodes with small degree can be kept in the same partition. In particular, the algorithm considers the destination vertex of each edge. If this vertex is a hub, i.e. if its in-degree is larger than a given threshold, then the edge is assigned to a partition based on the hash value of its source id. Otherwise, the assignment is based on the destination vertex id value.

`HybridCutPlus` (denoted as HCP) is a hybrid partitioner which combines HC and EP2D. It computes the *inDegree* value (number of incoming edges) of the source and the destination vertices. If both values are either greater or less than a given threshold, then the EP2D partitioner is used for this edge. If the *inDegree* value of the source vertex is less than the threshold and the *inDegree* value of the destination vertex is greater than the threshold, then the hash value of the source is used to assign the edge to a random partition. If the *inDegree* value of the source vertex is greater than the threshold and the *inDegree* value of the destination vertex is less than the threshold the hash value of the destination is used to assign the edge to a random partition. The HCP partitioner targets balance and communication efficiency properties.

[3]Still the vertex can appear in multiple partitions because of its incoming edges.

4 Statistical analysis methodology

We want to study if the partition metrics considered in literature and described above in Sec. 2.2 are good predictors for the final execution time. To this purpose, we would like ideally to design experiments where we could randomly and independently select the 5 different metrics, generate an input graph with these quintuple of characteristics, evaluate the execution time of the application of interest and finally use some statistical tool like regression models to identify the contribution of each metric to the execution time. Unfortunately, it is not possible to select first the partition metrics and then produce a graph with such characteristics. We can only use a partitioner on a given graph to produce a partition and then evaluate its metrics. Their values will be far from independent and in particular the structure of their correlation can be a function of the specific partitioner used.

For this reason, in order to obtain a variety of different configurations, we use all the 6 partitioners on the same graph. For each graph this leads to 6 different quintuples. A statistical model based on 6 points in a 5-dimension space would very likely overfit the data. Considering other graphs allow us to obtain other samples. The drawback is that if we use real datasets, they are going to differ for the number of nodes and of edges. Now, these two metrics are very likely to have an effect much more important on the execution time than the partition metrics, which are the focus of our study.[4] The contribution of the partition metrics would be dwarfed by the size change.

In order to overcome this difficulty, we generated 10 input graphs with the same size simply randomly permutating the ids of the vertices. Even if the different graphs are homeomorphic, partitions are calculated using the ids, and then each partitioner produces different subsets with different values for the metrics of interest. This approach allowed us to have 60 different quintuples for each graph: the ten permutations of the original graph multiplied by the six partitioners. In this way the risk of an overfitting model is significantly reduced.

To identify the most important metrics, we then used a linear regression model (denoted as LRM) with the 5 partition metrics as input variables (or predictors) and the execution time as output variable (or response). We want to find the most important predictors. Obviously the more predictors we have in a LRM the smaller is the residual error, but which ones are really important? For this purpose we used the *best subset selection* method [11, Chapter 6]. In particular, given the small number of predictors (5), we were able to consider all the possible 5! linear models for each original graph. Models with the same number of predictors can be easily compared through their R^2 value. In this way 5 models were selected, the best one was finally identified as the one with the smallest Akaike information criterion [5].[5] Once the best model was selected, we ordered its predictors by considering those that lead to the largest increase

[4]Remember that our goal is choosing the best partitioner for a given input graph of interest, whose size in terms of number of nodes and edges is out of our control.

[5]We looked at other indices (Bayesian Information Criteria, sample-size corrected AIC, R^2 adjusted) and in most cases the results did not differ.

of the R^2 when added as predictors.

5 Experiments

All our experiments were performed on a cluster of nodes with dual-Xeon E5-2680 v2 @2.80GHz with 192GB RAM and 10 cores (20 threads). We used Spark version 1.4.0 and Spark standalone cluster as a resource manager. We used two different cluster configurations. In the first configuration, master and executor share the same machine. In the second case, one machine is dedicated to the master and one to each of ten executors. We configured 4 Spark properties: *spark.executor.memory* equals to 4 GB (if there are 10 executors) or 40 GB (if there is 1 executor), *spark.driver.memory* equals to 10 GB, *spark.cores.max* equals to 10, *spark.local.dir* points to local directory.

Two different processing algorithms were evaluated: `Connected Components` and `PageRank` with 10 iterations. `PageRank` exhibits the same computation and communication pattern at each stage (the PageRank of each vertex value is updated according to the same formula and then propagated to the neighbors), while `Connected Components` implements a label propagation mechanism, where as time goes on, less updates are required.

As datasets, we used two undirected graphs: the *com-Youtube* graph (1,134,890 vertices/2,987,624 edges) and *com-Orkut* graph (3,072,441 vertices/117,185,083 edges) from SNAP project [4].

For each experiment we, first, randomly selected a partitioner, a graph processing algorithm, and a permutation of an input graph. Second, we applied the selected partitioner to the input graph. Finally, we executed the selected graph processing algorithm. To overcome execution time variability (due to a shared network, shared distributed file system, operational system layer, etc.), every combination has been tested at least 30 times, obtaining 95% confidence intervals for the execution time whose relative amplitude is always less than 10%.

As we mentioned above, for each graph we obtained 60 different quintuples for the partition metrics. Table 1 shows their correlation matrix for *com-Youtube* graph. The matrix clearly identifies the two groups of metrics: those relative to partition balance (BAL, LP NSTDEV) and those related to communication (VC, CC). The high level of collinearity also explains why in the LRM some coefficients are negative [11, Chapter 3].

We computed and selected linear regression models for *com-Youtube*, *com-Youtube doubled*, and *com-Orkut* graph, as it was discussed in Section 4. Table 2 and Table 3 summarize our experimental results respectively for `PageRank` and `Connected Components`. For each input graph and number of machines, they show the best LRM with the factors listed in decreasing order of importance. We note that, although a single node has enough resources to perform all the computation, the results in Table 2 and Table 3 show that the average execution time decreases as the number of machines increases. A similar result was observed in [10] where it was due to the higher memory contention in case of a

Table 1: Correlation matrix for partition metrics *com-Youtube*

	BAL	LP	NSD	VC	CC
BAL	1	**0.9332**	**0.9445**	0.2070	-0.2448
LP		1	**0.9799**	0.0993	-0.3177
NSD			1	0.1275	-0.2667
VC				1	**0.8737**
CC					1

single machine. It is possible that the effect has the same cause here.

Below we illustrate our main findings about the partition metrics.

5.0.1 The execution time depends on the partition metrics

For `PageRank` the R^2 value is very high (always above 0.99) while the Root Mean Square Error (RMSE) is less than 5% of mean execution time. This leads to conclude that indeed the execution time depends on these metrics. The dependency is less stronger for `Connected Components`. This can be explained with the fact that `PageRank` roughly does the same operations at each stage, while `Connected Components` is a label propagation algorithm where very soon the values of most of the vertices will not be updated. The execution time likely depends on graph properties (e.g. the graph diameter) that are not well captured by these metrics.

Figures 1 and 2 confirm these results by comparing the experimental execution time with the one predicted using the best LRM model we found. Results are grouped by partitioners, for each of them results for the 10 graph permutations are shown. The plot for the *com-Orkut* is similar (we do not show it here due to lack of space).

En passant we observe that HC appears to be the best partitioner confirming the results in [7], it also outperforms its variant HCP.

5.0.2 Communication metrics are the most important

In all the LRMs, the most important metric is always a communication metric (CC or VC), even when a single machine is used and then the network is scarcely used.

This result is also confirmed by comparing the 5 single-predictor LRMs: the LRMs using VC or CC have larger R^2 value. Figure 3 shows how LRM models using only VC or CC are able to produce quite good predictions at least of the relative performance of the different partitioners. On the contrary, the corresponding plot for balance metrics (BAL, LP, NSTDEV) shows an almost horizontal line.

As a side note, we tried to find a setting where balance metrics would have become the most important ones. In particular we considered i) a single machine, ii) very limited memory for each partitioner (70MB), iii) a modified version of `PageRank` where each computation is effectuated 1000 more times. The purpose was to make communication among executors as fast as possible, while increasing the computational burden on each executor. Even in this settings, communication metrics appeared to be the most important ones.

5.0.3 Results are robust to the partitioners considered

One of the purposes of our analysis is to be able to rank a priori different partitioners before carrying on the experiments. We evaluated how much our results are sensitive to the specific set of partitioners used to tune the LRM. To this purpose, we carried on the same analysis using only 5 partitioners (we did not use HC). We then used the new LRM model to predict the execution time for the HC partitioner. Figure 4 shows that predicted execution time for HC is underestimated. However, the order of the partitioners in terms of execution time is the correct one. Then the LRM model correctly estimates that using HC the application would finish faster. One could argue that the LRM has been trained using HCP that is a variant of HC. We then performed the same set of experiments removing both HC and HCP. In this case the predicted executed time is really off, but the LRM still correctly predicts the ranking of the different partitioners.

5.0.4 There is space for better partition metrics

While the results above are encouraging, the following experiment suggests that these partition metrics do not capture all the relevant features.

We considered a version of *com-Youtube* graph where we doubled each edge so that both directions are present in the input graph (and then `PageRank` correctly computes the PageRank of the original undirected graph). We denote the doubled graph as *com-Youtube doubled*.

The execution time of both algorithms on *com-Youtube doubled* is larger, roughly 30% (see Table 4). Nevertheless, the communication partition metrics that we identified as the most important ones may fail to detect the change. Indeed, this is evident if we compare the CRVC partitioner with the other ones. Indeed, as shown in Table 4 for a specific input graph, the CRVC partitioner provides the same values for all the partition metrics, and in particular for VC and CC metrics, both for *com-Youtube doubled* and *com-Youtube*, since it partitions edges based on the vertex (either source or destination) with the smallest id. For the other partitioners instead, the metrics VC and CC are different. The difference is evident if we compute the LRM including or not CRVC. In the first case the R^2 value of the LRM for `PageRank` significantly decreases from 0.99 for *com-Youtube* (see Table 2) to 0.82. In the second case the LRMs are equally good.

We think a more meaningful communication metric could be defined that would permit to identify the difference between CRVC and the other partitioners. We plan to investigate this aspect in the future.

Table 2: Results for `PageRank` algorithm

Graph name	Cluster configuration	Mean time (ms)		Metrics (ordered by importance)					RMSE	R^2
com-Youtube	single machine	12,343	metrics	VC	CC	BAL	LP	NSD	111.83	0.996
			coefficients	1.5507	0.3156	0.42743	-0.49504	0.33548		
	1 master + 10 executors	9,068	metrics	VC	CC	BAL	LP	NSD	125.47	0.993
			coefficients	1.2816	0.27261	0.50989	-0.64857	0.49891		
com-Orkut	single machine	122,929	metrics	CC	LP	VC			1199.4	0.998
			coefficients	2.312	5.039	12.942				
	1 master + 10 executors	94,738	metrics	CC	LP	NSD	VC		2222.8	0.989
			coefficients	1.3285	8.3243	-11.486	12.881			

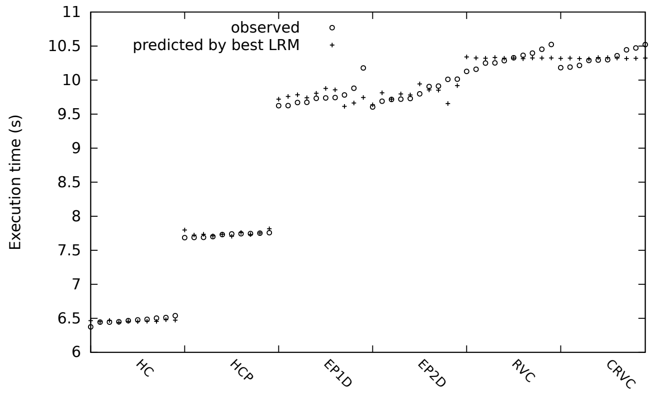

Figure 1: Execution time for `PageRank` algorithm on *com-Youtube* graph: experimental results vs best LRM predictions

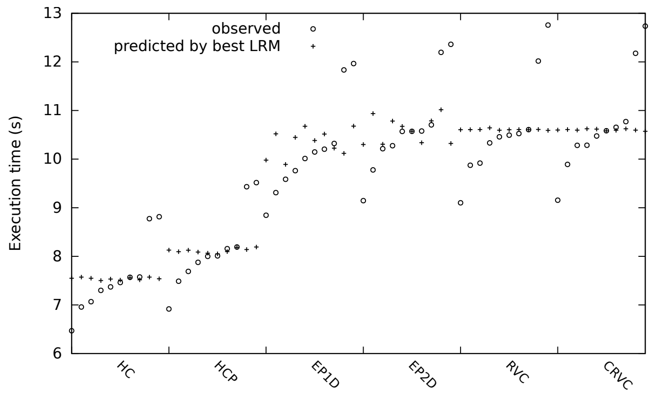

Figure 2: Execution time for `Connected Components` algorithm on *com-Youtube* graph : experimental results vs best LRM predictions

6 Conclusion

We used linear regression models with partitioning metrics as predictors and the average execution time for different graph processing algorithms as the observed

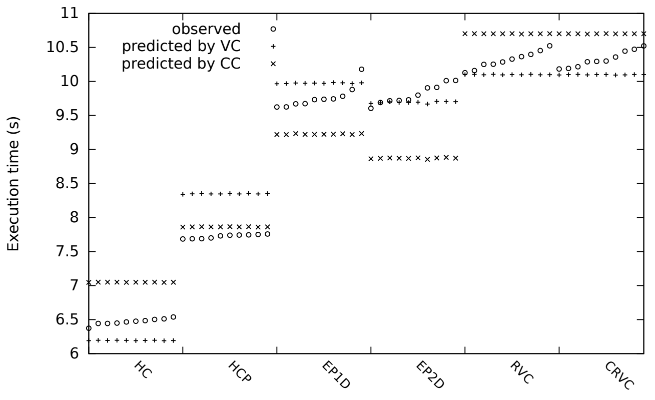

Figure 3: Execution time for `PageRank` algorithm on *com-Youtube* graph: experimental results vs LRM predictions using communication metrics

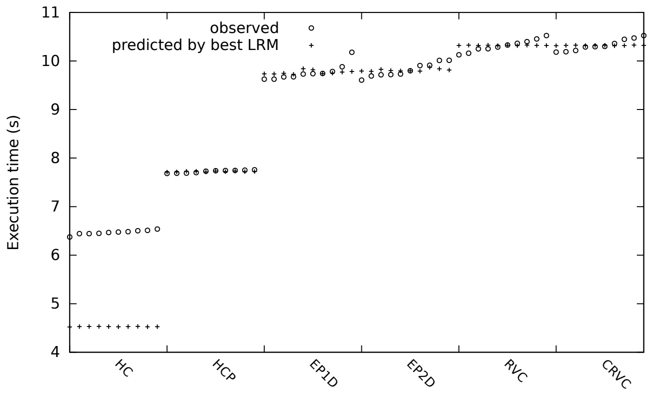

Figure 4: Prediction for HC partitioner (using *com-Youtube* graph, and `PageRank` algorithm)

values. The obtained models confirmed that there is actual dependency between these quantities. More importantly, the most important metrics are CC and VC, the both are an indicator of the amount of communication among the executors. On the contrary, the metrics that quantify load unbalance across the executors are less important. This conclusion holds whether communication is inter-machine or intra-machine, i.e. both if the network is used or not. Our results are robust to the original set of partitioners used to train the model, and the model can correctly rank other partitioners. At the same time, there is probably still space to define more useful metrics to evaluate partions.

Table 3: Results for `Connected Components` algorithm

Graph name	Cluster configuration	Mean time (ms)		Metrics (ordered by importance)			RMSE	R^2
com-Youtube	single machine	12,429	metrics	VC	LP	NSD	1119.2	0.499
			coefficients	1.7884	-1.3177	1.3861		
	1 master + 10 executors	9,635	metrics	VC	LP	NSD	922.27	0.681
			coefficients	2.1131	-1.7528	2.1043		
com-Orkut	single machine	86,253	metrics	CC	LP	NSD	2628.2	0.959
			coefficients	1.2039	9.771	-8.2634		
	1 master + 10 executors	71,952	metrics	CC	LP	NSD	2235,6	0.973
			coefficients	1.0854	15.484	-17.96		

Table 4: Metrics and execution time of `PageRank` for HC and CRVC

Graph name	HC			CRVC		
	Average time (ms)	VC	CC	Average time (ms)	VC	CC
com-Youtube	6468	0.304	1.091	10327	0.452	1.831
com-Youtube doubled	8121	0.702	2.204	13404	0.452	1.831

7 Acknowledgements

This work was partly funded by the French Government (National Research Agency, ANR) through the Investments for the Future Program reference #ANR-11-LABX-0031- 01.

References

[1] Apache Spark. `http://spark.apache.org`

[2] HybridCut and HybridCutPlus partitioners. `https://github.com/larryxiao/spark/`

[3] Source code. `https://github.com/ Mykhailenko/scala-graphx-tw-g5k`

[4] Stanford Large Network Dataset Collection. `https://snap.stanford.edu/data/`

[5] Akaike, H.: Akaikes information criterion. In: International Encyclopedia of Statistical Science, pp. 25–25. Springer (2011)

[6] Bourse, F., Lelarge, M., Vojnovic, M.: Balanced graph edge partition. In: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 1456–1465. ACM (2014)

[7] Chen, R., Shi, J., Chen, Y., Chen, H.: PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In: Proceedings of the Tenth European Conference on Computer Systems. EuroSys '15, ACM, New York, NY, USA (2015)

[8] Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., Guestrin, C.: Powergraph: Distributed graph-parallel computation on natural graphs. In: Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). pp. 17–30 (2012)

[9] Guerrieri, A., Montresor, A.: Dfep: Distributed funding-based edge partitioning. In: European Conference on Parallel Processing. pp. 346–358. Springer (2015)

[10] Hood, R., Jin, H., Mehrotra, P., Chang, J., Djomehri, J., Gavali, S., Jespersen, D., Taylor, K., Biswas, R.: Performance impact of resource contention in multicore systems. In: Parallel & Distributed Processing (IPDPS), IEEE International Symposium on (2010)

[11] James, G., Witten, D., Hastie, T., Tibshirani, R.: An introduction to statistical learning, vol. 6. Springer (2013)

[12] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning and data mining in the cloud. Proceedings of the VLDB Endowment 5(8), 716–727 (2012)

[13] Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010)

[14] Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: bringing order to the web. Technical report, Stanford InfoLab, Stanford, CA (1999)

[15] Rahimian, F., Payberah, A.H., Girdzijauskas, S., Haridi, S.: Distributed Vertex-Cut Partitioning. In: 4th International Conference on Distributed Applications and Interoperable Systems (DAIS). vol. LNCS-8460, pp. 186–200. Berlin, Germany (2014)

[16] Xin, R.S., Gonzalez, J.E., Franklin, M.J., Stoica, I.: Graphx: A resilient distributed graph system on spark. In: First International Workshop on Graph Data Management Experiences and Systems. p. 2. ACM (2013)

[17] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (2012)