Cut to Fit: Tailoring the Partitioning to the Computation

Iacovos G. Kolokasis
FORTH & University of Crete, Greece
kolokasis@ics.forth.gr

Polyvios Pratikakis
FORTH & University of Crete, Greece
polyvios@ics.forth.gr

ABSTRACT

Graph analytics applications are very often built using off-the-shelf analytics frameworks, which are profiled and optimized for the general case and have to perform for all kind of graphs. As performance is affected by the selection of the partition strategy, analytics frameworks often offer a selection of partitioning algorithms. In this paper we evaluate the impact of partitioning strategies on the performance of graph computations. We evaluate eight graph partitioning algorithms on a diverse set of graph datasets, using four standard graph algorithms by measuring a set of five partitioning metrics.

We analyze the performance of each partitioning strategy with respect to (i) the properties of the graph dataset, (ii) each analytics computation and (iii) the number of partitions. We confirm that there is no optimal partitioner across all experiments and moreover, find no metric always correlated with performance, that could be targeted by novel partitioners. Finally, we find that partitioning time may become a significant part of total time, and investing a lot of time to approximate perfect partitioning may not be worth it. We propose that a "good enough" strategy may be to have a very fast (and locally computable) heuristic to select among the best performing partitioners for any given problem instance. We demonstrate this by proposing PARSEL, a very simple partitioner selector. PARSEL selects among the two best-performing partitioners very fast; we demonstrate that even such a simple heuristic can outperform either partitioning strategy alone.

KEYWORDS

GraphX; Partitioning; Evaluation

ACM Reference Format:
Iacovos G. Kolokasis and Polyvios Pratikakis. 2019. Cut to Fit: Tailoring the Partitioning to the Computation. In *2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA) (GRADES-NDA'19), June 30-July 5 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3327964.3328498

1 INTRODUCTION

Graph analytics computations are a significant part of a growing number of big data applications. Many different distributed analytics frameworks target graph analytics computations [1, 5, 12, 18, 27, 29], offering high-level programming models for the development of complex graph algorithms. Partitioning and placement are very important in distributed graph computations [7, 21, 24], as graph processing algorithms very often have irregular computational dependencies, which correspond to the graph structure. Sub-optimal partitioning and placement of the graph may cause load-imbalance, incur high communication costs, or delay the convergence of an iterative computation. Thus, many graph analytics frameworks provide a way for the user to control partitioning, aiming to allow for algorithm-specific optimizations.

Most frameworks use vertex cut partitioners, as these tend to produce more balanced partitions [2, 16], replicating vertices as required. Since communication cost tends to be somewhat proportional to the number of replicated vertices, vertex cut algorithms aim to find the partitioning that minimizes such vertices, implicitly assuming communication cost is the most important factor of performance variability.

However, communication cost is neither perfectly correlated with the number of replicated vertices, nor with the performance of all analytics computations on the partitioned graph. Other metrics have been found to affect performance, including the vertices of the largest partition, the ratio of replicated vertices to total vertices, the ratio of edges in the largest partition to total edges, and more [14]. In general, it is not straightforward which of these metrics can predict the performance of any given distributed graph computation, and thus should be targeted by a partitioning algorithm. As a result, there are several graph partitioning strategies proposed in the literature. Many such partitioning algorithms optimize one or more metrics, assuming that these are highly correlated with the performance of subsequent distributed computations on the partitioned graph. This raises two problems: First, there is no single optimal partitioner for all problems [9], *i.e.*, no simple way to predict the performance of the computation for the produced partitioning, for each partitioner algorithm. Secondly, using complex partitioners results into increased partitioning time, which may negate any benefits from obtaining a near optimal partitioning of the graph.

This work aims to study these two problems. We use a wide range of partition metrics to quantify differences between partitioner algorithms on a diverse range of graph datasets. We study how these metrics are correlated with computation performance over a wide range of graph algorithms. We confirm that there is no "one size fits all" partitioning metric which is always correlated with computation performance: there is no single optimal partitioning of a graph that optimizes all

GRADES-NDA'19, June 30-July 5 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6789-9/19/06…$15.00
https://doi.org/10.1145/3327964.3328498

computations on that graph for all datasets. We also find a trade-off between the cost of discovering the optimal partitioning of a given graph for a given computation, and the speedup gained over other sub-optimal partitionings: it may not be worth paying the cost of discovering the optimal partitioning for a computation, if *e.g.*, it is only a part of a larger analytics workflow. Finally, we observe that the fastest dataset for a graph algorithm can be the slowest for another; for example, a small, flat, non power-law graph of road networks that is trivial to analyze in all but one stages of an analytics workflow, can still dominate the total running time.

We test this trade-off between the cost of optimal partitioning and the resulting speedup, by proposing PARSEL, a dynamic partitioner selector for GraphX [27] that defaults to one of the two best-performing partitioners based on a simple heuristic. We selected GraphX/Spark as these have currently the most active communities in terms of repository commits on github.com and technical discussions on stackoverflow.com. Note, however that Spark materializes shuffle operations to disk, which can be slow and incurs a high serialization cost [1]; this may in part explain why we found task granularity to be very significant. To reduce the impact of this artifact caused by Spark's design, we used high performance SSD devices for Spark's `/tmp` filesystem in all experiments. We found PARSEL to perform similarly to state-of-the-art partitioning algorithms that calculate near-optimal partitioning, especially when taking into account the partitioning time.

Overall, the contributions of this paper are:

- We systematically evaluate a set of partitioning algorithms using a wide set of metrics on a diverse set of graphs, over a set of four graph algorithms, showing that there is no single best partitioner for all algorithms or all datasets.
- We quantify the correlation of each partitioning metric with execution time for various graph computations, showing there is no single partitioning metric that can consistently predict best performance for all graph computations or all datasets.
- We propose PARSEL, a heuristic way to select a partition strategy, which performs equally or better to any single partitioner over all datasets and all algorithms, and is more resistant to outliers.

We found that even a simple dynamic partitioner selector that reduces partitioning time by quickly selecting among simple, fast partitioners can outperform complex computations that achieve near optimal partitioning, especially when part of an analytics workflow with multiple computation steps.

The remainder of the paper is organized as follows. Section 2 presents the impact of partitioning algorithms on total computation time, showing no single best partitioner; Section 3 presents a set of metrics we use to characterize each partitioning and understand why the optimal partitioning depends on the computation and dataset; Section 4 proposes a very simple partitioner selector that demonstrates the advantages of dynamic adaptation; Section 5 discusses related literature; and Section 6 concludes.

Dataset	Vertices	Edges	Type	Size
web-wikipedia-link-fr	4.9M	113.1M	Power-Law	1.6G
soc-twitter-2010	21.2M	265.0M	Power-Law	4.4G
road-road-usa	23.9M	28.8M	Low-Degree	469.7M
soc-sinaweibo	58.6M	261.3M	Long-Tailed	3.8G
socfb-uci-uni	58.7M	92.2M	Long-Tailed	1.5G

Table 1: Characterization of datasets.

2 PARTITIONER EVALUATION

This section shows the evaluation of a set of partitioning algorithms using five metrics on five graphs having diverse properties, over a set of four graph workloads. We partition each graph with all partitioners and compute five metrics that characterize the resulting partitioning.

2.1 Datasets

We use a set of five graphs to compare the partitioner algorithms. All graphs were obtained from the Network Repository, a large comprehensive collection of network graph data [17]. Namely, *web-wikipedia-link-fr* is a directed network of hyperlinks between the articles of Wikipedia in French, *soc-twitter-2010* include friend and follower relations of the follower network from Twitter. *road-road-usa* is the road network of the USA, *soc-sinaweibo* is a connected part of the Sina Weibo online social network, and *socfb-uci-uni* is a social friendship network extracted from Facebook, consisting of nodes and edges representing people and friendship ties, respectively.

Table 1 shows some representative characteristics of the datasets, as reported in the corresponding publications, or, when missing, as we measured them using GraphX. The first column shows the name of each dataset, ordered by the number of vertices. The second and third columns show the size of each graph, its number of vertices and edges, respectively. The fourth column shows the type of the datasets, *i.e.*, Low-Degree, Power-Law, Long-Tailed. The last column shows the size of the each graph on disk.

2.2 Partition Strategies

We use both simple and complex partitioners for two reasons. First, simple partitioners have lower ingress (partitioning) time, even if they may not produce an optimal result. Second, complex partitioners have high ingress time which they aim to amortize by producing a better partitioning that will lower the computation execution time. We use four simple partitioners provided by GraphX, two simple partitioners we developed, and two complex partitioners proposed in related work.

GraphX uses vertex cut partitioning; it first distributes graph edges into Spark-RDD partitions[2] and then builds a graph partition representation, local to each RDD partition, containing local and replicated vertices as well as metadata describing all necessary communication to implement a BSP computation. GraphX includes four ways to initially partition the edge list that represents each graph, which result into four different vertex cut strategies. We developed and use two additional partitioning strategies aiming to explore the design space and optimize for different metrics. Also, we use two

[1] We would like to thank Reviewer 2 for pointing out this observation.

[2] GraphX uses Spark-RDDs to represent graphs and maps operations in the Bulk-Synchronous Parallel (BSP) programming model into low level RDD operations.

complex partitioners introduced by Mykhailenko *et al.* [13], namely Hybrid Cut and Hybrid Cut Plus[3]. Specifically, we use the following GraphX partitioners:

Random Vertex Cut (RVC) assigns edges to partitions by hashing together the source and destination vertex IDs, resulting in a random vertex cut that collocates all same-direction edges between two vertices.

Edge Partition 1D (1D) assigns edges to partitions by hashing the source vertex ID. This causes all edges with the same source vertex to be collocated in the same partition.

Edge Partition 2D (2D) arranges all partitions into a square matrix and picks the column on the basis of the source vertex's hash and the row on the basis of the destination vertex's hash. This strategy guarantees a $2 \times \sqrt{N}$ upper bound on vertex replication where N is the number of partitions. Moreover, this strategy works best if the number of partitions is a perfect square; if not, the algorithm uses the next largest number and potentially creates unbalanced partitioning.

Canonical Random Vertex Cut (CRVC) assigns edges to partitions by hashing the source and destination vertex IDs in a canonical direction, resulting in a random vertex cut that collocates all edges between two vertices, regardless of direction. For example (u, v) and (v, u) hash to the same partition in Canonical Random Vertex Cut but not necessarily under RVC.

Hybrid Cut (HC) selects some vertices as "hubs" based on their number of incoming edges, and then places edges into partitions using a Destination Cut strategy when the destination is a hub, or a Source Cut strategy when it is not. We use the GraphX implementation provided by the authors [13].

Hybrid Cut Plus (HCP) distributes edges using the Edge Partition 2D strategy when source and destination vertices are both hubs or both not hubs; if only one of them is a hub, the algorithm places the edge near the non-hub vertex.

To further explore the design space, we designed and implemented two additional partitioners that change some of the assumptions in GraphX partitioners:

Source Cut (SC) assigns edges to partition by simple modulo of the source vertex IDs with the total number of partitions. This is similar to the 1D partitioner, but also assuming that vertex IDs may capture a metric of locality. Furthermore, GraphX internally construct vertices partitions by simple modulo of the each vertexId with the total partitions number. We expected this partitioner to result into less balanced partitions, as the hashing function in 1D achieves a more uniform distribution, but in cases where vertex ID similarity captures locality, we expected this partitioner to take advantage of it.

Destination Cut (DC) assigns edges to partitions by simple modulo on only the destination vertex IDs with the total number of partitions. This is similar to SC except it uses the vertex ID of the edge destination to assign edges to partitions. As in SC, we expect any correlation between vertex IDs and locality to be captured at the expense of load-balancing.

2.3 Partition Metrics

For each of these partitioners we measure a set of metrics that capture the properties of the partitioning, and help understand how a partitioner works on each different dataset. We use a set of standard metrics that have been used to predict performance [14]. Note that even if GraphX uses vertex cut partitioning, it does not store solely edges on each partition, but instead reconstructs the vertices per edge partition, and finally creates a data structure that includes the partition's vertex list. Specifically, for every produced partitioning we compute the following metrics:

Balance aims to capture how balanced partition sizes are, and is equal to the ratio of the number of edges in the biggest partition, over the average number of edges per partition.

Normalized Edge Partition Standard Deviation (NSTDEV) is the normalized standard deviation of the number of edges per partition. Similarly to Balance, it constitutes a measure of imbalance in the edge partitions. Note, however, that imbalance in edge partitions does not necessarily mean unbalanced usage of memory, as the final partitions also hold the vertices of all included edges.

Replication Factor (RF) is the ratio between the total number of vertices of each partition, including replicated vertices, and the total number of vertices of the original graph.

Cut Vertices is the number of vertices that exist in more than one partition, irrespective of how many copies of each cut vertex there are. In essence, these are the unique vertices copied across partitions into the number of copies that forms the Communication Cost metric.

Communication Cost (CommCost) aims to approximate the communication cost incurred by the partitioning for a Bulk-Synchronous Parallel computation that stores a fixed-sized state on all vertices. It is equal to the total number of copies of replicated vertices that exist in more than one partition, as this is the number of messages that need to be exchanged on every superstep to agree on the state stored in these vertices.

2.4 Evaluation

We ran our experiments on a cluster of 5 servers with 32-core Intel(R) Xeon(R) E5-2630 CPUs and 256GB of main memory each, configured as 1 Spark Driver and 4 Spark Workers containing 6 Executors each. Following Spark guidelines on how to use node resources, each Executor used 29GB of memory and 5 cores, resulting into 120 total cores. Nodes connect with a 40Gb network. Each PageRank (PR) execution iterates 100 times, whereas Connected Components (CC) runs to completion. For this evaluation we use 240 partitions per RDD, *i.e.*, twice the number of cores. We restart Spark between runs to always start with a new JVM.

Figures 1(a) through (e) show the measurement of each partitioning metric on all datasets, namely (a) shows Balance, (b) NSTDEV, (c) RF, (d) Cut Vertices, and (e) CommCost. The x-axis lists partitioners[4] and the y-axis shows the resulting metric measured after partitioning with the given partitioner.

[3] https://github.com/larryxiao/spark/

[4] Note that the x-axis is not continuous, line gradients are only drawn for better visibility compared to points or bars.

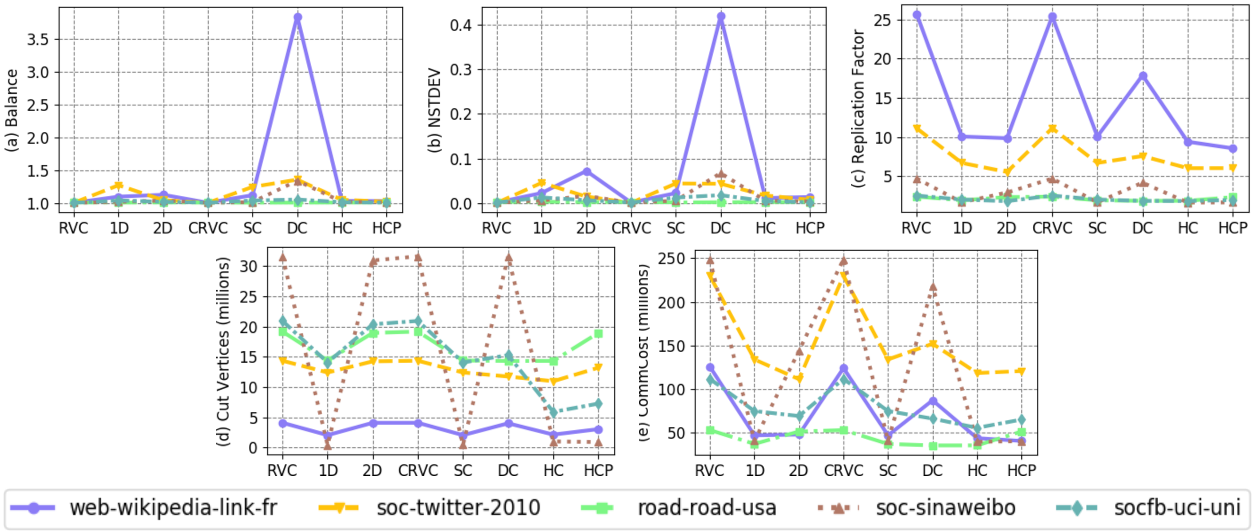


Figure 1: Partition metrics describing each partitioner for diverse graph datasets.

Since for Cut and CommCost the absolute measured values would not be comparable across datasets, the figures shows these metrics normalized over the maximum value observed for each dataset.

Figures (a) Balance and (b) NSTDEV show that all but one partitions produced by all partitioners are quite balanced, having a balance score under 1.5 and Standard Deviation under 0.1. In one case, for web-wikipedia-link-fr, DC produced unbalanced partitions. Figure (c) shows that RVC and CRVC have the highest RF across all datasets; DC follows, having a lower RF in grid graphs, possibly due to node IDs being correlated with locality; 1D, 2D and SC behave similarly with respect to the resulting RF; HC and HCP produce the lowest RF, although not for all datasets and not by a large factor. Overall, power-law graphs result into a higher RF for all partitioners. Figure (d) shows that RVC and CRVC have the highest number of Cut Vertices across all datasets; HC has the lowest number for most of the datasets. A low number of cut vertices usually means a low replication factor, except for datasets with cut vertices that are replicated multiple times; this occurs for HC and 2D on the soc-twitter-2010 dataset. Figure(e) shows that RVC and CRVC have the highest CommCost overall; HC has the lowest CommCost for most datasets; DC has low CommCost for the grid dataset but a high CommCost for power law graphs.

3 PARTITION METRICS AS PERFORMANCE PREDICTORS

We try to investigate which partition metric is actually correlated with the performance of each partitioner on different graph algorithms. A correlated metric might be a useful tool for the users to select a partitioner based on their application and dataset properties.

3.1 Graph Analytics Workloads

To measure the effect of the differences in partitioning presented above, we ran the following graph algorithms in GraphX. We chose the following workloads because: (i) they have different characteristics — for example, PageRank and Connected Components are iterative algorithms that involve iterative computations overall vertices, while Triangle Count operates only on a part of the graph, (ii) they are very popular algorithms in graph analytics.

PageRank (PR) computes the importance of websites within the web graph. A vertex is ranked highly when it has many incoming edges. PageRank is a way to measure the importance of each vertex based on the shape of the graph around it.

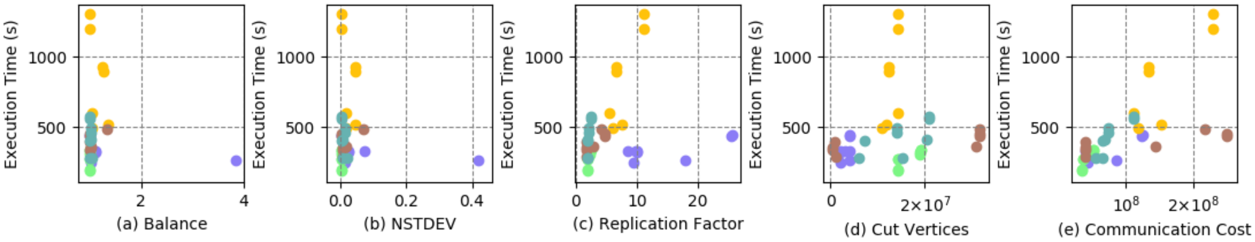
Connected Components (CC) computes the number of connected components of the graph. The connected components algorithm labels each connected component of the graph with ID of its lowest-numbered vertex.

Triangle Count (TC) computes the number of triangles passing through each vertex and sums to find the number for the whole graph. The algorithm is relatively straightforward and makes the computation in three steps. It first computes the set of neighbors for each vertex and broadcasts it on all edges; then intersects received sets at each vertex to find common neighbors; sends the corresponding count to each neighbor; finally, all counts are summed at each vertex and divided by three since each triangle was counted three times.

Single Source Shortest Path (SSSP) computes the shortest paths from a given vertex to all others and returns a graph where each vertex attribute is a map containing the shortest-path distance from the given source.

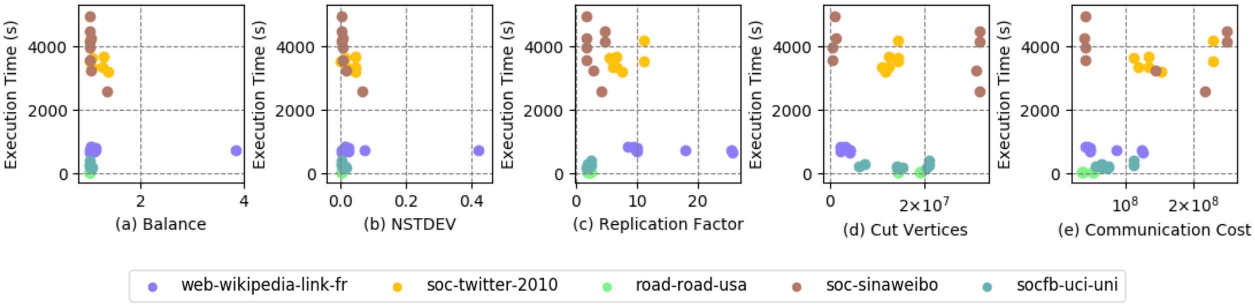
3.2 Evaluation

We measured the correlation between the execution time — not including partitioning time— and each partitioning metric, for each algorithm. Overall, we found that no single metric can be used as a reliable predictor of execution time across all benchmarks and algorithms. Indicatively, Figures 2 and 3 show correlation between the partitioning metrics and execution time for the PR and TR algorithms. We omit the plots for CC



(a) Balance (b) NSTDEV (c) Replication Factor (d) Cut Vertices (e) Communication Cost

Figure 2: Correlation between performance and partition metrics of each partitioner for PR



(a) Balance (b) NSTDEV (c) Replication Factor (d) Cut Vertices (e) Communication Cost

Figure 3: Correlation between performance and partition metrics of each partitioner for TC

and SSSP for economy of space, as they show similarly large variability.

Specifically, Figures 2(a) and (b), and Figures 3(a) and (b), show partition balance to not correlate with running time; in contrast, on the two largest datasets the correlation is negative. This may be an artifact of GraphX's load-balancing scheduler being able to hide the impact. Figure 2(c) and Figure 3(c) show that RF correlates with execution time for PR but not for TC. CC and SSSP (not shown) resemble PR, showing small positive correlations per dataset, except for the road-road-usa and socfb-uci-uni datasets for which RF has no correlation with running time. Figures 2(d) and (e) and Figures 3(d) and (e) show that Cut Vertices and Communication Cost are better predictors of execution time for PR (and for SSSP –not shown) than the other metrics, but not for TC. For CC (not shown), Cut Vertices and Communication Cost are similarly correlated with execution time with a nearly-zero, yet positive, coefficient, for all datasets except road-road-usa.

Based on these results, we conclude that there is no perfect partitioning metric that can predict performance of each partitioner for each computation algorithm. We found that RF, Cut Vertices, and CommCost can be used as predictors each in some cases, but not with the same confidence and not for all datasets or algorithms.

4 PARTITIONER SELECTOR (PARSEL)

The experiments of the previous section confirm the practitioner's view that there is no single best partitioner for all datasets or algorithms. This has resulted in multiple partitioning algorithms supported by each graph analytics framework, enabling the fine-tuning of computation on a case-by-case basis, as there are no clear guidelines for even selecting the optimal partitioner.

In addition, however, we refine this view by finding that most common partitioning metrics optimized by partitioning algorithms, are also not reliable predictors of performance. We interpret this to mean that selecting the optimal partitioning algorithm depends not only on the computation that is to take place, but also on dynamic properties of the data, such as the density and size of the graph, granularity of partitioning, etc. To test this hypothesis, we implemented a very simple dynamic partitioner that selects between partitioning algorithms based on a dynamic property, namely the granularity of partitioning. We train this simple, threshold-based partitioner selector (PARSEL) on the experiments presented in the previous section, and test it on a different collection of datasets and configurations. We found PARSEL to be less sensitive to the diversity between datasets and computations, indicating that dynamic partitioner selection may be tailored to outperform static selection by experts.

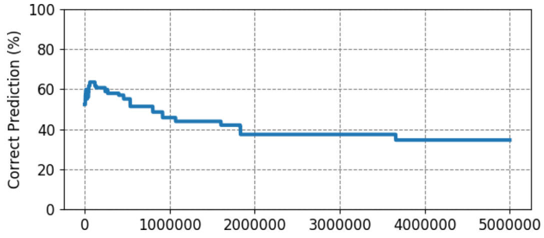


Figure 4: PARSEL threshold selection in relation to the correct predictability rate

We used a diverse collection of datasets of various sizes and properties to test PARSEL. To avoid bias and over-fitting, we use a new set of graphs to train PARSEL. We run all graph algorithms on the new datasets for each partitioner, using three different granularity configurations: (i) 128 partitions scheduled on 128 cores, (ii) 256 partitions scheduled on 128 cores and (iii) 512 partitions scheduled on 128 cores. We selected the number of partitions to be equal, two times greater, and four

Dataset	Vertices	Edges	Size on Disk	Type
RoadNet-PA [10]	1.0M	3.0M	83.7M	Low-Degree
YouTube [28]	1.1M	2.9M	74.0M	Power-Law
RoadNet-TX [10]	1.3M	3.8M	56.5M	Low-Degree
Pocek [23]	1.6M	30.6M	404.0M	Long-Tailed
RoadNet-CA [10]	1.9M	5.5M	83.7M	Low-Degree
Orkut [28]	3.0M	117.1M	3.3G	Long-Tailed
socLiveJournal [3]	4.8M	68.9M	1.0G	Long-Tailed
follow-jul	17.1M	136.7M	2.7G	Power-Law
follow-dec	26.3M	204.9M	4.1G	Power-Law
web-uk-2005 [17]	39.4M	936.3M	16.0G	Long-Tailed

Table 2: Datasets of various sizes and properties to train the PARSEL

times greater than the total number of cores, to explore the overhead/load-balancing trade-off, while using all resources.

We ran all experiments on the same cluster of 5 32-core Intel(R) Xeon(R) E5-2630 CPUs with 256GB of main memory. However, to reduce the effect of the system configuration on the training, we configured Spark differently: we use 1 Spark Driver and 4 Spark Workers, containing 1 Executor each. Each Executor used 225GB of memory and 32 cores, resulting into 128 total cores. Nodes were connected using a 40Gb network. We use the average time of 5 runs, and we restart Spark every time to start with a new JVM.

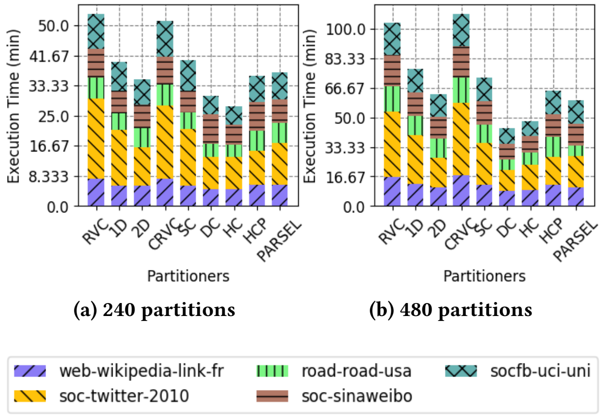
Table 2 shows the datasets we used for the training part. The first column shows the dataset name, the second column shows the number of vertices, the third column shows the number of edges and the last column shows the size of the dataset on disk, in Gigabytes. The *follow-jul* and *follow-dec* datasets are parts of the twitter.com follow graph that we crawled using the twitter API starting July 2016, and up to July 2017 and December 2017 respectively and that we have anonymized by hashing user IDs [15]. The first dataset is a subset of the second, and both include friend and follower relations of any users that have published tweets in the Greek language during the corresponding time period. We use the same GraphX algorithms, namely PR, CC, TC and SSSP.

We repeated all experiments of the previous section using the new configurations and datasets, and we found that the two partitioners that rank first for most experiments are 2D and DC. Based on this observation we configure PARSEL to select between 2D and DC, based on a selection threshold. As selection must not depend on global properties of the graph that would require partitioning before selecting, we opted to use the expected partition size as a decision metric to select between 2D and DC. This is straightforward to compute locally, without processing the graph or scheduling work to executors, as it is simply the ratio of two known parameters: (i) the total number of edges and (ii) the total number of partitions.

Formally, average partition size $P(G, n)$ for a graph $G = (V_G, E_G)$ and n partitions is

$$P(G, n) = \frac{|E_G|}{n}$$

To use this single metric to decide between the two best performing partitioners, DC and 2D, we express $T_{\text{PARSEL}}(f, G, n, P_t)$ the running time PARSEL would take to run algorithm f on dataset G partitioned into n partitions, using threshold P_t as:



(a) 240 partitions **(b) 480 partitions**

Figure 5: PR performance on all datasets for each partitioner.

$$T_{\text{PARSEL}}(f, G, n, P_t) = \begin{cases} T_{\text{DC}}(f, G, n) & \text{if} P(G, n) < P_t \\ T_{\text{2D}}(f, G, n) & \text{if} P(G, n) >= P_t \end{cases}$$

To train the threshold value P_{PARSEL} for PARSEL, we ran all algorithms $\mathcal{F} = \{\text{PR}, \text{CC}, \text{TC}, \text{SSSP}\}$ for the training datasets $\mathcal{D}$ and configurations C, and select the optimal threshold P_{PARSEL} that minimizes the total running time:

$$P_{\text{PARSEL}} = \underset{P_t}{\arg\min} \left(\sum_{f \in \mathcal{F}} \sum_{G \in \mathcal{D}} \sum_{n \in C} T_{\text{PARSEL}}(f, G, n, P_t) \right)$$

Figure 4 shows the percentage of experiments where PARSEL indeed selects the best of the two partitioners, for each threshold value P_t; formally, the percentage of all f, G, n configurations where:

$$T_{\text{PARSEL}}(f, G, n, P_t) = \min\left(T_{\text{DC}}\left(f, G, n\right), T_{\text{2D}}\left(f, G, n\right)\right)$$

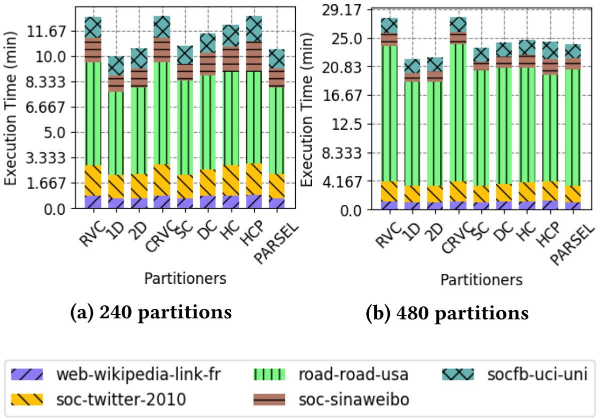
Specifically, we found the optimal threshold value for the training datasets and configurations to be 119,000, which selects the best of the two partitioners for 63.55% of the experiments.

4.1 Evaluation

For the evaluation of the selected threshold T_{PARSEL}, we ran all experiments using the cluster described in Section 2.4.

We used two partitioning configurations: (i) 240 partitions scheduled on 120 cores and (ii) 480 partitions scheduled on 120 cores. Figures 5, 6, 7 and 8 show the total time including of partition and computation time for running PR, CC, TC and SSSP workloads respectively, on all datasets, for all partition strategies, for configurations (i) and (ii). Each figure shows the sum of partitioning and computation time for each partitioner on each dataset in seconds.

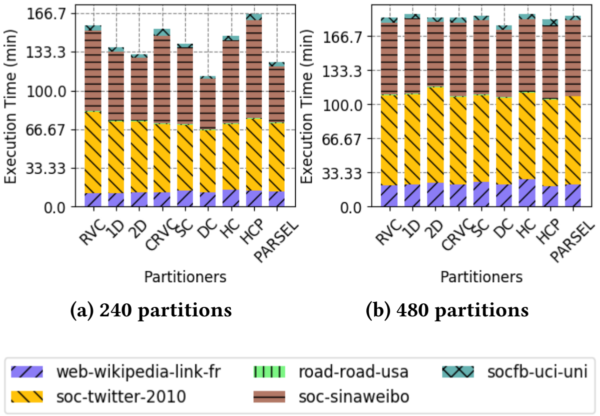
Figure 5(a), shows that PARSEL achieves fourth place among partitioners for the coarse-grain configuration for PR. Specifically, PARSEL achieving 28% less total execution time than CRVC and 18% more total execution time than DC. Figure 5(b) shows the performance of each partitioner for configuration (ii). PARSEL achieves the third place overall, achieving 45% less



(a) 240 partitions

(b) 480 partitions

Figure 6: CC performance on all datasets for each partitioner.



(a) 240 partitions

(b) 480 partitions

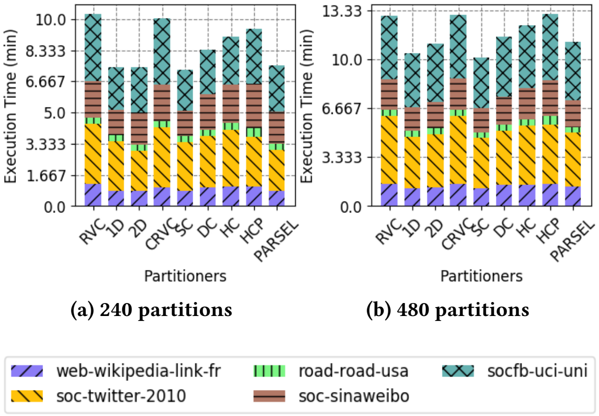
Figure 7: TR performance on all datasets for each partitioner.

total execution time than CRVC and 36% more total execution time than DC.

Figure 6(a), shows that PARSEL achieves second place among partitioners for the coarse-grain configuration for CC. Specifically, PARSEL achieving 17% less total execution time than HCP and 4.5% more total execution time than 1D. Figure 6(b) shows the performance of each partitioner for configuration (ii). PARSEL achieves the fourth place overall, achieving 14.1% less total execution time than CRVC and 8.9% more total execution time than 1D.

Figure 7(a), shows that PARSEL achieves second place among partitioners for the coarse-grain configuration for TC. Specifically, PARSEL achieving 24.8% less total execution time than HCP and 9.5% more total execution time than DC. Figure 7(b) shows the performance of each partitioner for configuration (ii). PARSEL achieves the third place overall, achieving 1% less total execution time than HC and 5% more total execution time than DC.

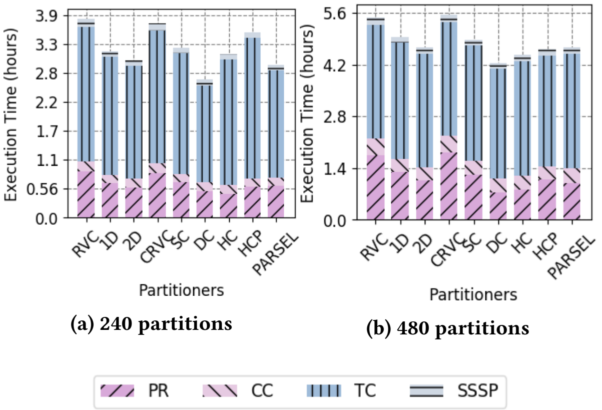
Figure 8(a), shows that PARSEL achieves fourth place among partitioners for the coarse-grain configuration. Specifically, PARSEL achieving 26.4% less total execution time than RVC



(a) 240 partitions

(b) 480 partitions

Figure 8: SSSP performance on all datasets for each partitioner.



(a) 240 partitions

(b) 480 partitions

Figure 9: Total computation time of an analytics workflow with multiple steps.

and 3.4% more total execution time than SC. Figure 8(b) shows the performance of each partitioner for configuration (ii). PARSEL achieves the third place overall, achieving 9% less total execution time than HCP and 14.5% more total execution time than SC.

4.2 Partitioners Over Analytics Workflow

In analytics workloads, users commonly perform more than one computation on each graph dataset. We examine the effect of the partitioning algorithm on complex computations in an analytics workflow with multiple steps, by assuming a workflow consisting of all algorithms presented above. Figures 9(a) and 9(b) show the performance of each partitioner on a complex analytics workflow computation. Note that this is not merely the sum of execution times of the experiments presented in the previous section, as subsequent analytics computations will not re-partition the graph; so, partitioning cost is better amortized over all stages. The y-axis represents the computation time for a workflow that partitions once with each partitioner and runs all algorithms.

In configuration (i), PARSEL outperforms by 29% the worst partitioner (RVC) and under-performs by 10% the best partitioner (DC). Also, in comparison with HC and HCP, PARSEL performs 7% and 20% better, respectively. In addition with configuration (ii), PARSEL outperforms by 19% the worst partitioner (CRVC) and under-performs by 9% the optimal partitioner (DC). Compared with HC and HCP, PARSEL under-performs by 5% and 2%, respectively. Overall, we found that dynamic partitioner selection can better tolerate different computations, datasets and resource configurations compared to complex partitioning strategies. We also found, however, that complex partitioning strategies can outperform simpler partitioners for workflows that do not require re-partitioning, as the cost of partitioning is amortized over all computation steps. In support of this, the next section presents a breakdown of the partitioning cost per partitioner.

5 RELATED WORK

Apache GraphX implements the Bulk Synchronous Processing (BSP) model [25] for graph processing, first introduced for large-scale graph analytics in Pregel [12] and adopted by many frameworks [1, 5, 8, 11, 19, 29] As graph analytics frameworks aim to be generic and support any algorithm, they are often forced to generalize their design over all graph computations, resulting in sub-optimal performance compared to a hand-crafted implementation of each algorithm, resulting in a huge performance gap from optimal hand-crafted implementations [20].

Several comparisons of these frameworks show that performance greatly depends on the dataset, computation, and partitioning of the graph [4, 6, 26], Most conclude that there is no single best partitioning strategy, with some concluding that usually Canonical Random Vertex Cut is a good choice of a partitioner [6]. Our results show that this conclusion is not always true, and partitioners performance affected from the application properties and the number of partitions.

There are several approaches to graph partitioning in the literature, aiming to optimize the performance of graph processing frameworks. Fennel [24] is a one-pass streaming graph partitioning algorithm achieving significant performance improvement than previous implementations. Stanton *et al.* [21] present a streaming graph partitioner that eliminates communication cost of partitioning by partitioning as the graph is loaded. Karypis *et al.* [7] have proposed hierarchical partitioning similar to clustering and community detection computations, to optimize communication costs.

Different graph partitioners try to minimize the communication cost and achieve locality. These partitioners works for some graph algorithms. Kumar *et al.* [9] investigate a cost model for the Pregel-like design systems in Spark GraphX. Their cost model shows the relation between the graph size, the total number of partitions, the application properties and the cluster configuration parameters. Their cost depends on several variables, that are not well known, making the selection of the best partitioner unclear.

Sun *et al.* [22] investigate how graph partitioning affects the performance of graph processing engines, showing that partitions have noticeably different connectivity compared to the original input graph. They point out that an imbalanced number of vertices per partition has a significant effect on performance, mainly for algorithms which make passes over the vertices and not solely over the edges. As a result, highly-connected vertices become an important issue at a higher number of partitions, as some partitions contain more vertices compared to others.

6 CONCLUSIONS

Distributed graph analytics frameworks efficiency is highly dependent on the partitioning strategies used. We test an array of computations on multiple datasets and confirm the effect on computation time can be very large; we moreover found there is no single optimal partitioner for all problems and no simple way to predict the performance of the computation for the produced partitioning, as no single metric is a good predictor of workload execution time across workloads. We also measure ingress times for multiple partitioners and demonstrate the trade-off between the benefit in computation cost by using complex partitioners, that however incur an increased partitioning time.

Our findings can be interpreted to mean that spending a lot of time to approximate perfect partitioning may not be worth it, depending on the computation steps used in an analytics workflow. We propose PARSEL, a simple partitioner selector based on a single decision that can be taken locally without overhead. PARSEL selects very fast between the two best performing partitioners based on a threshold computed from a set of datasets and configurations. It then can achieve results better than these two static partitioners on a different set of datasets and configurations. This indicates that more complex techniques of partitioner selection, even if imperfect, can yield even better results and performance less sensitive to configurations, workloads and datasets. We plan to explore such models based on more metrics, in future work.

7 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. Also we thankfully acknowledge the support of the EuroEXA Project (Grant Agreement N^o 754337), funded by the European Union Horizon 2020 Research and Innovation Programme.

A INGRESS TIME VS COMPUTATION TIME

This section presents an analysis of ingress times per partitioner and dataset, which explains the differences in performance between simpler (DC) and more complex (HC) partitioners for a single computation at a time, versus workflows of many computations. Here, *Ingress Time* is the time spent by a partitioner after loading the graph dataset, to assign partition IDs to edges and vertices and distribute the data across nodes. *Computation Time* is the time needed for the workload execution after partitioning is finished and all data get distributed to partitions.

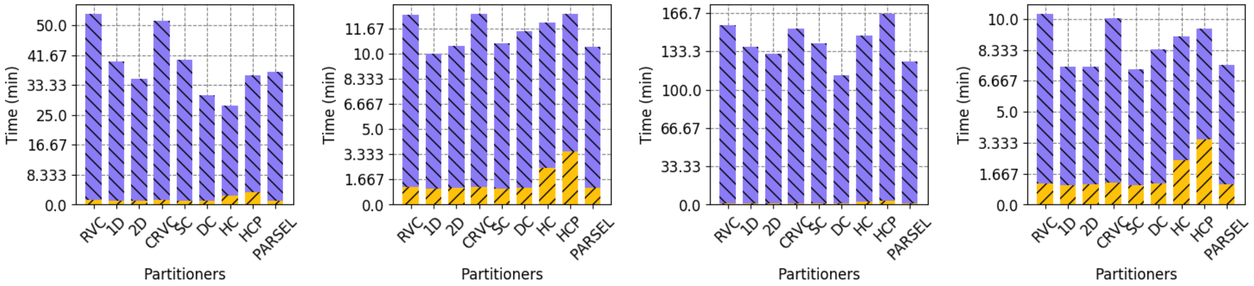


Figure 10: Ingress Time vs Computation time for PR, CC, TC and SSSP workloads across all datasets using 240 total number of partitions

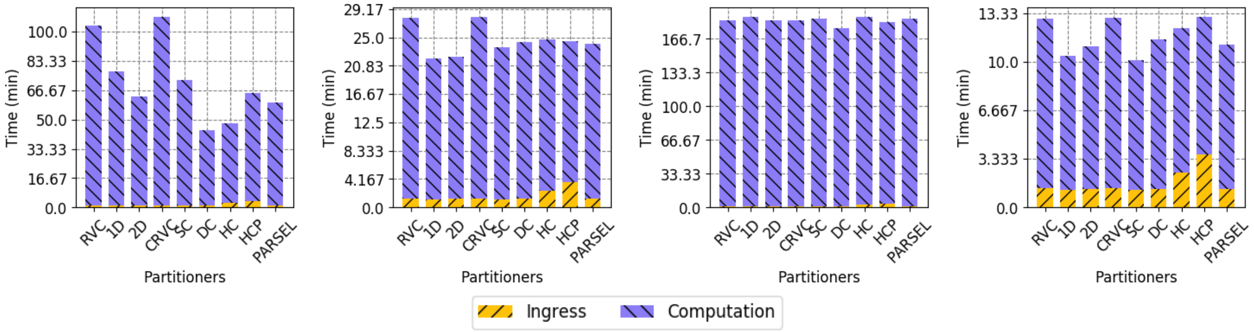


Figure 11: Ingress Time vs Computation time for PR, CC, TC and SSSP workloads across all datasets using 480 total number of partitions

As ingress time is included in the total cost, it should be low. Although it is very often implicitly assumed to be negligible or amortized, we found that depending on the computation, it may not be straightforward to choose between "simple but crude" or "complex but optimized" partitioners. Simple partitioners have lower ingress times, but in most cases —but not all— the total computation time for the application is higher.

Figures 10 and 11 show the total ingress time and computation time in PR, CC, TC, and SSSP workloads across all datasets. The X-axis represents the partition strategies and the Y-axis represents time breakdown in seconds. We found that RVC, 1D, 2D, SC, and DC have lower ingress time —up to 50%— compared to HC and HCP. For the PR workload, however, despite HC having a high ingress time, it manages to produce partitionings resulting in a much lower computation time. This is due to the locality achieved for the specific algorithm.

These results show a trade-off between ingress time and computation time for the selection of a partitioner. In complex analytics workflows consisting of multiple steps and many workloads, using complex partitioners will improve the computation time, but only when re-partitioning is avoided. This, however, may not be straightforward as depending on the computation, a different partitioning may be optimal, as shown in the previous sections. Overall, and depending on the diversity of computation steps in an analytics workflow, this means that re-partitioning the graph using fast and crude partitioners selected dynamically at each step may outperform an optimized partitioner that accelerates one single computation step a lot, but may not be a good fit for subsequent computations.

REFERENCES

[1] Apache giraph. http://giraph.apache.org/.

[2] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124. IEEE Computer Society, Apr. 2006. http://dl.acm.org/citation.cfm?id=1898953.1899055.

[3] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'06, pages 44–54. ACM, Aug. 2006. http://doi.acm.org/10.1145/1150402.1150412.

[4] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, Aug. 2015. http://dx.doi.org/10.14778/2824032.2824077.

[5] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings on the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30. USENIX Association, Oct. 2012. http://dl.acm.org/citation.cfm?id=2387880.2387883.

[6] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 7(12):1047–1058, Aug. 2014. http://dx.doi.org/10.14778/2732977.2732980.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, Dec. 1998. http://dx.doi.org/10.1137/S1064827595287997.

[8] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182. ACM, Apr. 2013. http://doi.acm.org/10.1145/2465351.2465369.

[9] R. Kumar, A. Abelló, and T. Calders. Cost model for pregel on graphx. In *Advances in Databases and Information Systems ADBIS'17*, volume 10509 of *Lecture Notes in Computer Science*, pages 153–166. Springer, Cham, Aug. 2017. https://doi.org/10.1007/978-3-319-66917-5_11.

[10] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large

well-defined clusters. *Internet Mathematics*, 6:29–123, Jan. 2009. https://doi.org/10.1080/15427951.2009.10129177.

[11] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8):716–727, Apr. 2012. https://doi.org/10.14778/2212351.2212354.

[12] 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 ACM SIGMOD International Conference on Management of Data*, SIGMOD'10, pages 135–146. ACM, June 2010. http://doi.acm.org/10.1145/1807167.1807184.

[13] H. Mykhailenko, F. Huet, and G. Neglia. Comparison of edge partitioners for graph processing. In *International Conference on Computational Science and Computational Intelligence*, CSCI'16, pages 441–446. IEEE, Dec. 2016.

[14] H. Mykhailenko, G. Neglia, and F. Huet. Which metrics for vertex-cut partitioning? In *11th International Conference for Internet Technology and Secured Transactions*, ICITST'16, pages 74–79. IEEE, Dec. 2016.

[15] P. Pratikakis. twawler: A lightweight twitter crawler. *CoRR*, abs/1804.07748, Aug. 2018. http://arxiv.org/abs/1804.07748.

[16] F. Rahimian, A. H. Payberah, S. Girdzijauskas, and S. Haridi. Distributed vertex-cut partitioning. In *Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, volume 8460, pages 186–200. Springer-Verlag, June 2014. https://doi.org/10.1007/978-3-662-43352-2_15.

[17] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pages 4292–4293. AAAI Press, Jan. 2015. http://dl.acm.org/citation.cfm?id=2888116.2888372.

[18] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Symposium on Operating Systems Principles*, SOSP'15, pages 410–424. ACM, Oct. 2015. http://doi.acm.org/10.1145/2815400.2815408.

[19] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12. ACM, July 2013. http://doi.acm.org/10.1145/2484838.2484843.

[20] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Management of Data*, SIGMOD'14, pages 979–990. ACM, June 2014. http://doi.acm.org/10.1145/2588555.2610518.

[21] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'12, pages 1222–1230. ACM, Aug. 2012. http://doi.acm.org/10.1145/2339530.2339722.

[22] J. Sun, H. Vandierendonck, and D. S. Nikolopoulos. Graphgrind: Addressing load imbalance of graph partitioning. In *Proceedings of the International Conference on Supercomputing*, ICS'17, pages 16:1–16:10. ACM, June 2017. http://doi.acm.org/10.1145/3079079.3079097.

[23] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International Scientific Conference and International Workshop Present Day Trends of Innovations 2012*, volume 1, May 2012.

[24] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM'14, pages 333–342. ACM, Feb. 2014. http://doi.acm.org/10.1145/2556195.2556213.

[25] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990. http://doi.acm.org/10.1145/79173.79181.

[26] S. Verma, L. M. Leslie, Y. Shin, and I. Gupta. An experimental comparison of partitioning strategies in distributed graph processing. *Proceedings of the VLDB Endowment*, 10(5):493–504, Jan. 2017. https://doi.org/10.14778/3055540.3055543.

[27] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *Graph Data Management Experiences and Systems*, GRADES'13, pages 2:1–2:6. ACM, June 2013. http://doi.acm.org/10.1145/2484425.2484427.

[28] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, Jan. 2015. http://dx.doi.org/10.1007/s10115-013-0693-z.

[29] L. Yucheng, G. Joseph, K. Aapo, B. Danny, G. Carlos, and M. Joseph. GraphLab: A new framework for parallel machine learning. In *Conference on Uncertainty in Artificial Intelligence*, UAI'10, pages 340–349. AUAI Press, July 2010. https://dslpitt.org/uai/displayArticles.jsp?mmnu=1&smnu=1&proceeding_id=26.