ParViz: Visualizing Graph Partitioners

Hadi Sinaee

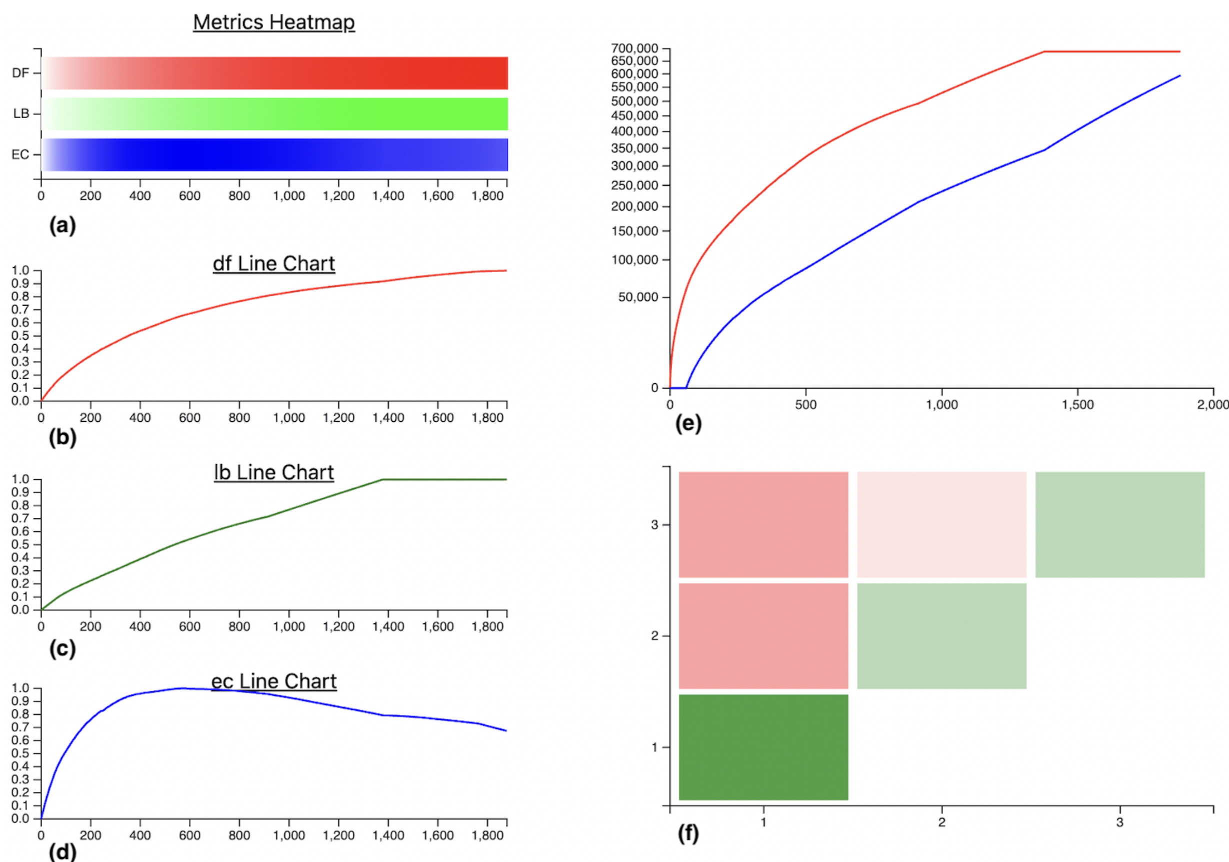

Fig. 1. Hadi Sinaee

Abstract— Graph analysis is an important research area due to the prevalence of graphs for modelling relationships. Real-world graphs are big in terms of the number of nodes and edges, and this causes many problems for fast graph analysis. One approach for dealing with this complexity is to divide a big graph into smaller graphs. The algorithms that perform this division are called graph partitioners. While there is ongoing research for designing new partitioners, little effort has been made to visualize their underlying algorithms to provide better understandability for future algorithm designers. ParViz is a visualization tool that helps algorithm designers to understand the underlying behaviour of partitioners through visualizing their partitioning metrics and final results.

Index Terms—Graphs, Graph Partitioners, Line Chart, Heatmap, Scalability

1 INTRODUCTION

Graphs are considered as one of the fundamental mathematical models for describing relationships. Social media platforms, such as Facebook or Twitter, are good examples of this type of modelling. Usually, in these graphs, people are vertices, and edges are relationships between them, which are defined based on a specific definition, such as friendship or follower-followee. Social media is not the only field that benefits from graphs; bioinformatics, astrology, or machine learning are a few examples of domains that use graphs.

During the past decades, graphs have become the interest of many researchers in different areas. However, the questions that are asked share the same type. For example, detecting communities in a graph, counting the number of triangles and computing the PageRank are among the most common frequently asked questions. Consequently, various graph analysis algorithms have been implemented to answer different questions about graphs. Also, researchers and companies have built many specialized systems for graphs, such as GraphChi [1] or Pregel [?], capable of storing graph data efficiently and running graph algorithms.

Besides optimizing the algorithms or data structures, parallel computation models are used to achieve high performance since they can run an algorithm in parallel or multiple queries simultaneously. Partitioning a graph into a set of sub-graphs is considered a common approach for parallel computation. Graph partitioners are algorithms that create a

• *Hadi Sinaee is with Systopia Group at the University of British Columbia. E-mail: sinaee@cs.ubc.ca*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

set of sub-graphs from a graph, which we call partition.

There are three types of partitioners: Vertex Partitioners, Edge Partitioners, and Hybrid Partitioners. Vertex Partitioners assign each vertex and its connected edges to a partition. On the other hand, Edge Partitioners assign each edge and its connected vertices to a partition. Also, there are Hybrid Partitioners that assign edges or vertices to partitions.

To devise new partitioners, algorithm designers need to know each partitioning algorithm and its corresponding partitioning quality. Understanding a partitioning algorithm means that one should know the strategy of an algorithm for partitioning; what the final partitioning looks like. Partitioning quality is a set of defined metrics, which are calculated after a partitioner is done. These metrics show how good partitioning is and what trade-offs we paid for that. Having these two in hand, algorithm designers can devise new algorithms or improve current ones. However, it is not as easy as it seems since there is no standard way of showing the partitioning result or analyzing the metrics of a partitioner.

Suppose we want to see the nodes' assignments of partitioning: which node and its edges belong to which partition. Since there is no standard method for showing the result of partitioning, an algorithm designer has to write custom scripts to achieve his goal. For example, a common approach for representing graphs is the METIS format, in which the number of nodes and edges along with edge information is provided in a file. The partitioner could produce the resulting partitions in the METIS format, and the algorithm designer can use them as inputs to his scripts. Furthermore, the initial goal of seeing the nodes' assignments could be more difficult when the number of edges or nodes is significant. This problem exacerbates when our goal is to compare two different nodes' assignments by two different partitioners.

Partitioning quality can be measured via a set of metrics and is used as a standard approach for choosing a partitioner for a task. Therefore, understanding the behaviour of algorithms through the lenses of these metrics becomes crucial. However, these metrics are calculated at the end of partitioning. Consequently, the changes to these metrics during the partitioning process are missed. For algorithm developers, it is useful and helpful to see intermediate changes since it helps them to 1- understand the behaviour of each partitioner as it partitions the graph and 2- provide a way to compare two algorithms at a more detailed granularity. Unfortunately, the only way of accessing intermediate changes is via analyzing ad hoc logs in a partitioner's codebase, which could be time-consuming.

ParViz is a viz tool to address mentioned challenges for understanding graph partitioners. At the first step, ParViz implements a tool for visualizing the quality metrics of a graph partitioner as they evolve. Having this feature, algorithm designers can understand the behaviour of each partitioner and compare them with each other.

2 RELATED WORK

3 DOMAIN

This section proceeds into the terminologies of the graph partitioners and provides background information for the rest of this paper. Next, in the task abstraction, we describe dataset type and data types, along with a description of the project's goal.

3.1 Graphs

Graphs are commonly used as a mathematical tool for modeling relationships: vertices show entities and edges show relationships between vertices. A graph G is defined as $G = (V,E)$, where V is the set of vertices in the graph, $V = \{v_1, v_2, ..., v_N\}$, and E is the set of edges connecting two vertices in that graph, $E = \{(v_i, v_j) | v_i, v_j \in V \text{ where } i, j = 1, ..., N\}$. Size of G is determined by its number of vertices and edges, which are denoted as N and M respectively ($|V| = N, |E| = M$). If for every (v_i, v_j), there is a corresponding edge (v_j, v_i), we say the graph is *undirected*; otherwise, it is *directed*. Also, there are *weighted graphs*, where each edge, e_{ij}, can have an associated weight, w_{ij}. In this project, we are focusing on *undirected* and *unweighted* graphs.

There are various methods for storing graphs. Adjacency list is a data structure that stores a graph in the form of a vertex and all its connected edges. Figure 2 shows the adjacency list for the vertex 3. As it shows, there are edges between 3 and 1,2,5, and 10. For storing a graph in the form of adjacency list, we store all vertices and their edges in this form.

In an undirected graph, the degree of a vertex is the number of edges it has. In Figure 2, the degree of vertex number 3 is four since there are four edges from vertex 3 to other vertices.

Fig. 2. Adjacency list for the vertex 3. Vertex 3 has edges to vertices 1, 2, 5, and 10.

We use adjacency list for storing graph data for the rest of this project. Also, when we are mentioning about graph data, we mean the adjacency list unless otherwise stated.

3.2 Graph Partitioning

Partitioning of a graph means to create multiple sub-graphs from the original one. The set $\{P_1, P_2, ..., P_k\}$ is a *k-partition* of a graph G if each P_i is a sub-graph of G :

$$P_i = (V_i, E_i) \ , \ V_i \subset V \text{ and } E_i \subset E \tag{1}$$

In other words, in a k-partitioning algorithm, we create K number of sub-graphs from the original graph G. Each created sub-graph has to have a set of vertices and edges that are subset of the original one.

There are different approaches for partitioning a graph: Vertex Partitioning, Edge Partitioning and Hybrid Partitioning. In Vertex Partitioning, a partitioner assigns each $v_i \in V$ and all its connect edges to a partition. Therefore, for an edge (v_i, v_j), if both of v_i and v_j were assigned to two different partitions, we would have a duplicated edge, i.e. (v_i, v_j), in their corresponding partitions. However, in Edge Partitioning, a partitioner assigns each edge $(v_i, v_j) \in E$ and its vertices, i.e. v_i and v_j, to a partition. Similarly, if two edges shared the same vertex, we would have duplicated vertices in different partitions. Finally, in hybrid partitioning, we assign both edges and vertices to different partitions, and, consequently, we would have both duplicated edges and vertices in each partition.

ParViz focuses on the Vertex Partitioning approaches since the number of vertices in graphs, which we use in this course project, are relatively manageable compared to the large number of edges that might exist.

3.3 Partitioning Metrics

Algorithm designers have introduced a variety of metrics for determining the quality of a partitioner. Due to the diversity of partitioners themselves, it is not possible to have a set of metrics that are suitable for all of them. Therefore, there are general metrics for determining the quality of a partitioner that are calculated differently based on the type of partitioner, e.g. Vertex Partitioning or Edge Partitioning. Among them, in this project, we are focusing on *Duplication Factor (DF)*, *Load Balancing (LB)* and *Edge-Cut (EC)*.

Duplication Factor measures the ratio of duplicated edges with respect to the original graph: sum of the total number of edges in each partition to the total number of edges in a graph. *Load Balancing* measure the load on the most loaded partition. *Edge-Cut* measure the number of edges between two partitions where the vertices of those edges do not belong to the same partition.

We can formulate these three metrics in terms of a vector of the following form:

$$MetricsVector = \begin{bmatrix} DF \\ LB \\ EB \end{bmatrix} \tag{2}$$

All of these metrics are usually measured at the end of the partitioning. If one wanted to compare the quality of a partitioner with another one, one could do that by comparing their corresponding metrics. However, it is possible to measure all of these metrics as the partitioner makes progress. At the end of each iteration of partitioning, as the partitioner has decided about a node's partition, we can calculate equation 2. Therefore, we can have an array of these metrics vectors:

$$MetricsVector_i = \begin{bmatrix} DF_i \\ LB_i \\ EC_i \end{bmatrix},\ i = 1, 2, ..., T \tag{3}$$

T is the total number of iterations of a vertex partitioning. For vertex partitioners, it is the number of vertices of a graph since vertex partitioners assign each node to a partition in each iteration of the algorithm. Therefore, the total number of iterations equals to the total number of vertices.

3.4 Data

There are two different datasets for this project. The first one is a log file for a partitioner name Fennel. The input graph of this partitioning is the Yahoo! Messenger graph, which has about 2 million nodes and 4 million edges. The graph is undirected. Nodes in this graph are the users of the messenger and edges show whether one of the users is a contact of the other one.

Our dataset type is Table with Items and Attributes, where each row corresponds to a step/iteration of our partitioner. The first two columns show the vertex number and its assigned partition, and the three leading columns show our metrics vector after the node assignment was done.

The number of rows in this dataset is equal to the number of vertices in our graph, i.e. $|V|$. Therefore, the dimensionality of our dataset is $|V| \times 5$. In this dataset, the number of rows is two million.

The number of partitions, K, is a parameter that a user specifies at the beginning of the partitioning. It ranges from 2 to the number of available computation resources. For example, if the partitioner will be deployed in a distributed environment, where we have ten workers, then K will be set to 10. However, in most scenarios we set K between 2 and 256 ($2 <= K <= 256$). In this dataset, the number of partitions is $K = 3$.

Since the number of iterations or steps for this partitioner was high, we had to sample our datapoints with a fixed frequency. The frequency of sampling in this project was 1000, which means for every 1000 steps, we only select the last 1000th step.

The second dataset is also of type Table with Items and Attributes, where each row corresponds to the number of edge cuts between two partitions. Each row has two partition numbers and the number of edge cuts for them.

4 Task Abstraction

In this section, we go over the task abstraction in ParViz. There are three tasks that this project is designed to do.

4.1 Task 1

While the approaches and heuristics of partitioners are different, they try to optimize the partitioning metrics(Section 3.3) in their underlying algorithms. Therefore, being able to see these evolutions not only can help understand a partitioner's behaviour but also let us create a new one better.

4.2 Task 2

The second task tries to give an overview of how partition sizes grow as the partitioner tries to fill each of them. Since not all partitioners partition a graph into similar sub-graphs, the sizes of the final partitions are different. As to better understand the behaviour of a partitioner, it is helpful to see how each partition grows.

What: Data	Table; DF, LB, EC quantitative attributes
What: Derived	Partitioner Steps: ordered key attribute Normalized Values of DF, LB, EC
How: Encode	Express DF, LB, EC; using a heat map with different hues and changing saturation
How: Reduce	Sampled at every 1000 point
Why: Task	Overview of changes in metrics
Scale	Items: 2K

Table 1. What-Why-How analysis for Task 1 in Figure 1

4.3 Task 3

In a vertex partitioner, edges are being duplicated because, in each iteration of partitioning, two vertices of the same edge might be assigned to two different partitions. Edges that spans two partitions incur communication cost since if a vertex needs the value of the other vertex, it has to ask a remote partition to receive it. Therefore, in the third task, we are interested in the number of edge cuts between two partitions.

5 Solution

In ParViz, we used the D3js library as the visualization tool and the NodeJs framework as the backend technology. A user provides two input files in the CSV format, each corresponding to the dataset discussed in section 3.4. Figure 1 is the result of running ParViz on the dataset described in section data. In this section, we walk through these six idioms of Figure 1 as an example for the mentioned tasks in section 4.

5.1 Metrics Heatmap

The metrics heatmap, Figure 1 part (a), shows the overview of changes in metrics during the partitioning(Task 1). The x-axis shows the sampled step of the partitioner, and the y-axis values show the normalized values of metrics. The purpose of this idiom is to give an overview of changes in metrics. Each metric is encoded with a different hue and its normalized values are encoded with different saturation. 1. The lower these values are the better they would be. Table 1 shows the What-Why-How analysis this idiom.

Based on the shown heatmap in Figure 1, the EC metric reaches its maximum value early in the partitioning process, and then its value decreases. The other two metrics show that their values go to their maximum value early on and do not change after that.

5.2 Metrics Line Charts

The metrics heatmap might not be helpful given that the colours are saturated early. Therefore, three completing line charts separately express the changes in these metrics; part (b), (c) and (d) in Figure 1. The x-axis shows the sampled step of the partitioner, and the y-axis values show the normalized values of metrics. The goal of this idiom is to complete the shortcomings of the metrics heatmap. Each metric is encoded with the same hue as its corresponding one in the heatmap. Table 2 shows the What-Why-How analysis for only DF metrics. The remaining LB and EC follow the same approach.

As it is shown, the EC metric reaches its maximum value and then it started to decrease. EC shows the sum of the number of edges that cross two partitions. If the EC value is high, it means that a node might need to ask a remote partition for data in process of running an algorithm. Asking remote nodes for data is resource-consuming and incurs communication costs. Therefore, the lower it is the better it would be.

The interpretation of this behaviour is that this graph partitioner, Fennel, is trying to decrease the number of edge cuts as soon as its maximum value is approaching. This observation was not obvious in previous studies by the author.

5.3 Partition Sizes

For Task 2, we again use the line chart to see changes in partition sizes at the end of each step of our partitioner; part (e) Figure 1. The

What: Data	Table; DF quantitative attributes
What: Derived	Partitioner Step: Ordered key attributed Normalized DF
How: Encode	Express DF horizontally; the x-axis is the steps the y-axis shows the DF value
How: Reduce	Sampled every 1000 steps
Why: Task	Overview of changes in DF
Scale	Items: ∼2K

Table 2. What-Why-How analysis for Task 1 in Figure 1

What: Data	Table; NodeId key attribute PartNum categorical key attribute
What: Derived	Matrix of the size: 2K * 3 Each cell is shows the number nodes at a specific step in a specific partition
How: Encode	Facet; Superimpose the size of partitions for at each step
How: Reduce	Sampled every 1000 steps
Why: Task	Overview of changes in partitions size
Scale	Items: ∼2K * 3

Table 3. What-Why-How analysis of the Task 2 in Figure 1

What: Data	Table; Part1, Part2 categorical key attribute ec quantitative attribute nedge_part quantitative attribute nedge quantitative attribute
What: Derived	Matrix of the size: 3 * 3 Each cell shows the normalized EC between two corresponding partitions
How: Encode	Express the normalized ec value using a heat map
Why: Task	Overview of ec in partitions size
Scale	Items: 3 * 3

Table 4. What-Why-How analysis of for Task 3 in Figure 1

Table 5. Work Breakdown: the estimated and actual time spent on building ParViz. Every 8 hours corresponds to a full day work

Phase	Estimation (hours)	Actual (hours)
Learning D3.Js	8	8
Building the skeleton of ParVis	8	3
Extracting Dataset	8	18
Task 1	40	40
Task 2	16	16
Task 3	8	12
Write-Up	8	16
Total	**96**	**113**

x-axis shows the sampled step of our partitioner and the corresponding y-axis shows the number of nodes in each partition. For each partition, we used different line charts with different hues. All line charts are superimposed. The Table 3 shows the What-Why-How analysis of this idiom.

As the chart shows, the red line chart, which corresponds to partition 1, is being filled first by the partitioner, while the other two are empty. This shows that the preference of this partitioner is to fill the first partition into a number and then try to fill the other ones. However, the maximum partition size of partition 1 had been reached when the partitioner switched from partition 1 to another one. This observation was not seen by the author before that and is considered as helpful insight for further study of the partitioner behaviour. The problem with the chart is that since Partition 2 and Partition 3 have about the same size, their corresponding charts hide each other.

5.4 EC Matrix Heatmap

Part (f) of Figure 1 is designed to answer the question for Task 3. The heatmap shows the partition edge cut number. The x-axis and y-axis show the partition number and the corresponding cell shows the normalized value of the EC metric for that cell. This heatmap uses two different hues: one for when the partition number is different, and the other one is the diagonal value where we have the same partition number. Also, for each hue, we use saturation to show its size.

The diagonal values of this heatmap show that how many edges have their vertices inside their partition: there is no need to ask a remote partition for a vertex value. The higher the saturation is, the better the situation would be. For example, in Partition 1 concerning Partition 2 or Partition 3, an algorithm may incure less communication cost when it is running since all vertice data would be available there.

In non-diagonal cells, it is the opposite: the higher the saturation is the worst the situation is. However, these values would be helpful in comparison with other cells. For example, the number edge cuts between Partition 2 and Partition 3 is less than Partion 1 and Partition 2. Therefore, if a running algorithm needs only Partition 2 and Partition 3 data, it would be faster than the case if it needed Partition 2 and Partition 1 data. This observation was not obvious to the author before this work.

6 Implementation And Milestones

In this section, we provide an estimate and actual time spent on implementing the ParVis, Table 6. I tried to adopt an iterative approach for designing and implementing ParViz, for each task phase. Since I use *D3.Js* as the visualizing library and there is no familiarity in my team, I needed to learn the framework first.

After learning the basics of D3Js, I built the prototype of the system. However, soon after implementing the prototype, I realized that I need to change the idioms and find better approaches for visualizing the results. The main problems with the first prototype were:

- A Wrong Idiom: The initial idea was to show each step of the partitioning process using a heatmap. There were three columns each representing one of three metrics. The number of rows was the same as the number of nodes in a graph. Each cell used a different hue and saturation to encode the value of a metric in that cell. A user could hover over a cell to see complementary information, such as the node id, its assigned partitioner, the normalized and un-normalized values of metrics. However, after seeing the result I realized that the granularity of this approach is too much and it is not helpful or even useful.

- Lack of Scalability: To ensure my approach works properly, I used a small graph with 7 nodes and 11 edges to ParViz. However, after I increased the size of the graph gradually, I noticed that I need to my approach from using the whole data to the sampled version of data.

I used NodeJs as a web framework to set up the backend for ParViz. I chose NodeJs since both D3Js and NodeJs come from the same technology family: both of them were written in Javascript.

7 Conclusion

ParViz is a tool for visualizing the partitioning process of a vertex partitioner. The main goal of this project is to provide an easy understanding of the behaviour of a vertex partitioner. It gives an overview of the changes of three key partitioning metrics in four different idioms: a heatmap where one can see all the metrics in the same idiom, three different line charts for each metric, a line chart that shows the growth of partitions sizes, and heatmap that shows the corresponding edge-cut number between two partitions.

ParViz could reveal insights that were not obvious at first to the author of this paper showing that the final result was helpful for understanding the behaviour of the testing partitioner named Fennel.

8 Future Work

To the best of my knowledge, ParViz is the first visualization tool that has been built for graph partitioners. The prototype focused on a simple version of the idea to test its feasibility and how much it could be useful. Therefore, there are limitations for this prototype that forms the future work of this project. The following are potential ideas for making ParViz better.

- ParViz is limited to vertex partitioners while there are two other major categories of partitioners: Edge Partitioners and Hybrid Partitioners.
- A sampling of input data is crucial for ParViz since it cannot render log files with millions of steps. Therefore, one potential future work is adding lazy loading or dynamic sampling techniques to deal with graphs with millions of nodes.
- Due to the available screen area, ParViz cannot render idioms for Task 2 and Task 3 with distinguishable line charts or heatmap when the number of partitions is large, e.g more than 256. One way of overcoming this limitation is by filtering partitions based on a certain threshold value.
- ParViz lacks the interactivity that a user might need. The interactivity requirement comes from the fact that a user might be interested in more information that might have been discarded due to sampling.
- Understanding the behaviour of an algorithm is helpful for designing new ones. However, it is also valuable to be able to compare two different algorithms against each other. Jaxtapous idioms could be considered for this purpose, but also it would be challenging because of the increasing number of items in an idiom.
- The colour selection for heatmaps should be improved, especially there has to be a better colour schema for the metrics heatmap. I have tried different color schemas but could not decide on a informative one.
- There should be more legends and extra information on each idiom to help the user to understand each idiom.

References

[1] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, p. 31–46. USENIX Association, USA, 2012.