PartViz: Visualizing Graph Partitioners

Hadi Sinaee
University Of British Columbia
sinaee@cs.ubc.ca

***Abstract*—Once upon time ...**

I. Introduction

Graphs are considered as one of the fundamental mathematical models for describing relationships. Social media platforms, such as Facebook, Twitter, or Pinterest, are good examples of these relationships where people are entities in a graph and their relationship modeled after a specific definition, such as friendship or follower-followee. Social medias are not the only fields that benefit from graphs; bioinformatics, astrology, or machine learning are a few examples of domains that use graphs.

During the past decades, the importance of using graphs has become the interest of researchers, and various graph analysis algorithms have been introduced to answer different questions about graphs; to name a few of them, there are algorithms detecting communities in a graph, counting the number of triangles, computing the PageRank. Also, researchers and companies have built many specialized graph systems, such as GraphChi, Pregel, or GraphOne, which are capable of storing graph data and/or running graph algorithms. All of these systems try to show high performance when it comes to running graph algorithms.

To achieve high performance, besides optimizing the algorithms or data structures, graph systems rely on parallel computation models in which an algorithm run in parallel or multiple queries can be executed simultaneously. A common approach is to partition a graph into sub-graphs, where algorithms can run in parallel on each of them. Graph partitioners are the algorithms that create a set of sub-graphs from a graph. There are different partitioning algorithms each of which has its strategy to create those sub-graphs. Vertex Partitioners assign each vertex and its connected edges to a partition, whereas, Edge Partitioners assign each edge with its connected vertices to a partition. Also, there are Hybrid Partitioners that assign edges or vertices to partitions.

For algorithm designers to devise new partitioners, they need to know each partitioning algorithm and its corresponding partitioning quality. Understanding of a partitioning algorithm means that one should know the strategy of an algorithm for partitioning; how the final partitioning looks like. Partitioning quality, on the other hand, are a set of defined metrics, which are calculated after a partitioner is done. These metrics show how good a partitioning is and what trade-offs we paid for partitioning. Having these two in hand, algorithm designers are able to 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 metrics of a partitioner.

Suppose we want to see the nodes assignment of a partitioner: each node and its edges belong to which partition. Since there is no standard method for providing the result of a partitioning, an algorithm designer has to write custom scripts to achieve his goal. For example, a common method for graph representation is the METIS format, which the number of nodes and edges along with edge information are provided in a file. The partitioner could produce the resulting sub-graphs in the METIS format, and the algorithm designer can use them as inputs to his scripts. Also, the initial goal of seeing the nodes assignment could be hard when the number of edges or nodes are significant. This problem exacerbates when our goal is comparing two different nodes assignments by two different partitioners.

Quality metrics of graph partitioners are the common approach for choosing a partitioner. Therefore, understanding the behavior of algorithms through the lenses of these metrics becomes crucial. However, these metrics are calculated at the end of partitioning, and the changes of these metrics during the partitioning process are missed. For algorithm developers, it is useful and helpful to see these intermediate changes since it helps them to 1- understand the behavior 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 handwritten logs in a partitioner's code base, which could be time consuming.

ParViz is a viz tool to address mentioned challenges for understanding graph partitioners. At first step, it provides a tool for visualizing the quality metrics of a graph partitioner as it evolves. In its second step, it provides a way to visualize the nodes assignment of an algorithm. Having these features, algorithm designers can understand behaviour of each partitioner and compare them with each other.

II. Related Work

III. Domain and Task Abstraction

This section goes deeper into terminologies of the graph partitioners and provides background information for the rest of this paper. Then, it finishes this section with the task abstraction. It describes dataset type and data types along with a description of the project's goal.

A. 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 1 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 1, the degree of vertex number 3 is four since there are four edges from vertex 3 to other vertices.

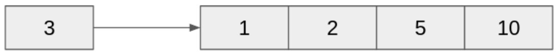

Fig. 1. Adjacency list for the vertex 3. Vertex 3 has edges to vertices 1, 2, 5, and 10.

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

B. 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 :

$$P_i = (V_i, E_i) \ , \ V_i \subset V \ and \ E_i \subset E \qquad (1)$$

In other words, in a k-partitioning method, 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 and 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.

C. Node Assignment

A Vertex Partitioner assigns each vertex and all of its connected edges to a partition. As described in Section B, the result of vertex partitioning causes duplicated edges since it might be the case that an edge having two vertices belonging to two different partition. The final partitioning result for a k-partitioning method is a set of K sub-graphs, $\{P_1, P_2, ..., P_k\}$, of the form in equation 1. Also, as we described in Section A, each $P_i = (V_i, E_i)$ is stored in the adjacency list format illustrated in Figure 1.

If one is interested in finding the nodes assignment of a k-partitioning, one has to iterate all K sub-graphs and deal with each duplicated edge. As the number of partitions and the size of a graph could be large, extracting the node assignment could be challenging.

D. Partitioning Metrics

Algorithm designers have introduced 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 quality of a partitioner that are calculated differently based on the type of a 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. *Load Balancing* measure the number of vertices per 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 think of these three metrics in terms of a vector:

$$metrics\ vector = \begin{bmatrix} DF \\ LB \\ EC \end{bmatrix} (2)$$

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 a partitioning, as the partitioner has decided about a node's partition, we can calculate equation 2. Therefore, we can have a series of these metrics vectors:

$$metrics\ vector_i = \begin{bmatrix} DF_i \\ LB_i \\ EC_i \end{bmatrix} \ , \ i = 1, 2, ..., T \qquad (3)$$

T is the total number of iterations of a vertex partitioning. For vertex partitioners, it is the number of vertices of a graph.

E. Task Abstraction

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

1) Task 1: How was a metric vector evolving during a partitioning process? One of the challenges for understanding a graph partitioner is to see how the metric vector has been evolved during the partitioning. If a programmer wants to understand it, it has to go through the whole log and look for a finding changes.

2) Task 2: How does the final nodes assignment look like? The result of partitioning is usually accessible after the partitioner is done. The final result usually is available as part of internal data structure of the partitioner or exported as a file. Therefore, putting all information about the nodes assignment of an algorithm is hard.

F. Data

There are two different dataset for this project. The first one deals with *Task 1* in which we are looking for a way to see how a metrics vector evolve during the partitioning. The second one deals with *Task 2* in which we are looking for the final nodes assignment.

1) Task 1: For this task, we have a tabular dataset where each row corresponds to an iteration of our partitioner. The first two columns show the vertex number and its assigned partition. The remaining three columns show our metrics vector after the node assignment.

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 project, we are going to focus on graphs with less than 100M vertices.

2) Task 2: For this task, we need the sub-graphs information. As we mentioned in Section III.A, the adjacency list is the way we store our graph. Therefore, the nearest file format to that representation is METIS format. For instance, Figure 2 shows a sample unweighted undirected graph. The annotations in Figure 2 start with L, which is the line number, and is not part of the METIS format(it was shown for clarity purposes). The first line, $L0$, starts with two numbers: the number of vertices and the number of edges. The rest of the lines, $L1$ to $L7$, each of them represents an adjacency list for its corresponding vertex. For example, $L3$ shows the adjacency list for vertex 3 ($\{(v_3, v_5), (v_3, v_4), (v_3, v_2), (v_3, v_1)\}$ is the edges of vertex 3).

All of our sub-graphs are stored in a METIS format and shows relationships between entities. Therefore, our dataset type is Network. The only available data type in this dataset is key-value, since it shows to identify an entity, i.e. a vertices and all its connected edges. It is not numerical since ordering doesn't have any meaning for the vertices.

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

```
L0: 7 11
L1: 5  3  2
L2: 1  3  4
L3: 5  4  2  1
L4: 2  3  6  7
L5: 1  3  6
L6: 5  4  7
L8: 6  4
```

Fig. 2. Sample METIS format for an unweighted undirected graph with 7 vertices and 11 edges. For example, $L3$ shows the adjacency list for vertex 3. Lines annotated with a label L, which is not part of the format(it is shown for the clarity purposes).

to 10. However, in most scenarios we set K between 2 and $256 (2 <= K <= 256)$.

Each key in our dataset is an integer, representing a unique entity, i.e. a vertex. Our values are a list of integers that show the corresponding vertex numbers for an edge. While our key and values are integers, it doesn't mean that there an ordering for them. The range of our keys is bounded by the number of vertices in our graph, i.e. $|V|$. Each member of our adjacency list is a vertex number and consequently is an integer. However, each adjacency list is has a length corresponds to the degree of its vertex. This length depends on the maximum degree in a graph.

IV. Solution

ParViz provides two different scenarios, which are associated with *Task 1* and *Task 2*, for users to interact with it, Figure 3. These two different scenarios are separated with two different user journeys chosen based on the first action of the user, i.e. a user at the beginning need to select one of *Task 1* or *Task 2*.

An step-by-step journey for *Task 1*, after the end user chose it, is as follows. First a user needs to select the path to his dataset file. Then, he click on a button to start the visualization. Since it might be a lengthy process, there is a loading window. After ParViz is done with the visualization, the result will be shown. There are T rows, the number of iterations for that partitioner, and three columns, *DF*, *LB** and *EC**. *LB** and *EC** are normalized values of their corresponding *LB* and *EC*. All normalized values are shown using the saturation channel of a single hue. We are using the same hue for all of these three attributes to avoid information overload.

The user can interact with the idiom by hovering on it. As he hovers on different rows, the corresponding metrics vector is shown to him. The values will be shown on the right hand side of the idiom. There is an scroller in the idiom to navigate through the rest of of data set. Since the number of rows could be high, we adopt a lazy loading approach for loading the next batch of data.

For *Task 2*, the first three steps are the same as *Task 1*. The final idiom shows a graph where the vertices are partition numbers and the edges are edges that cross two partitions. The thickness of these edges corresponds to the numbers crossing

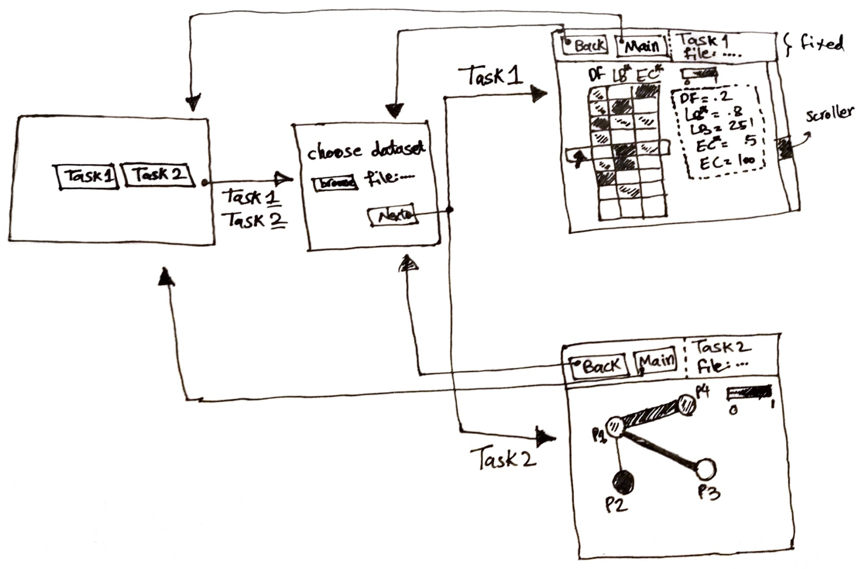

Fig. 3. An sketch of ParVis. Depending on the task, user will be navigated to a different window. For Task 1, if a user hovers over a row, the corresponding values are shown that row.

edges. Also, each node color encoded with the duplication factor, *DF* (Section III.D). Since *DF* is a value between 0 and 1, we use saturation channel for a single hue.

V. Milestones

In this section, we provide an estimate of how we are going to implement the ParVis, Table V. We try to adapt an iterative approach for designing and implementing the system, for each task phases. Since we use *D3.Js* as our tool and there is no familiarity in our team, we need to learn the framework first. Then, we build our sketch skeleton, 3. This skeleton is not functional; however, the navigation works. Then, we need to prepare each dataset for each task. Finally, in the last two phases, we work on our tasks, i.e. Task 1 and Task 2. We start with Task 1 since it is our main goal in this project. However, there is an estimation for Task 2, which we try to aim at whenever we are done with the first one.

TABLE I
Phases of ParVis and its estimations

Phase	**Estimation(hours)**
Learning D3.Js	8
Building the skeleton of ParVis	8
Extracting Dataset	8
Working on Task 1	40
Working on Task 2	16
Total	**80**