Analysis of Graph Partitioning Schemes on Distributed Clustering

(Project report for DS256 Scalable Systems for Data Science, Jan 2023)

Paritosh Tiwari
Ph.D. Student
Robert Bosch Center for Cyber-Physical Systems
Indian Institute of Science
Bangalore, India
paritosht@iisc.ac.in

Ruchi Bhoot
M.Tech (Research)
Department of Computational and Data Sciences
Indian Institute of Science
Bangalore, India
ruchibhoot@iisc.ac.in

***Abstract*—Through this project, we aim to validate the benefit of preserving partitions on distributed clustering in the domain of graph partitioning. We also propose to extensively study the effects of different partitioning schemes, analyze our experiments using suitable metrics and extend the existing frameworks.**

***Index Terms*—Graph Partitioning, Distributed Systems, Clustering**

I. Problem Description

Graph partitioning in the domain of distributed systems involves dividing a large graph into smaller subgraphs that can be processed by multiple computing nodes in a distributed system. The goal is to minimize the communication overhead between nodes while ensuring that each node has a roughly equal workload. This is important because it allows the distributed system to process the graph efficiently and in a parallel manner, which can greatly speed up computations. There are many algorithms and techniques for graph partitioning in distributed systems, including methods based on graph structure, vertex attributes, and communication patterns.

Edge-based partitioning schemes, as opposed to the common vertex-based partitioning, are particularly effective in cases where the edges of the graph are much more numerous and have a higher communication cost than the vertices. They generally try to minimize the number of edges between subgraphs. One common approach to edge partitioning is to use a hashing function to map each edge to a partition. The hashing function takes as input the source and destination nodes of the edge and assigns the edge to a partition based on the hash value. This approach ensures that edges with the same source and destination nodes are assigned to the same partition, which can reduce the communication cost between nodes. Edge partitioning is a powerful technique that can be used in various applications, such as large-scale social network analysis, web crawling, and machine learning.

The choice of a partitioning algorithm depends on the specific requirements of the distributed system and the characteristics of the graph being partitioned. We aim to explore the downstream task of running various distributed clustering algorithms and analyze the effects of different partitioning schemes on them. Clustering algorithms are used to group similar nodes or edges in a graph together, and they can be computationally expensive and memory-intensive. In a distributed system, these algorithms need to be partitioned across multiple computing nodes for efficient processing.

The quality of clustering results can be affected by how the graph is partitioned. If highly connected clusters are separated into different partitions, then the quality may be lower than if the clusters were kept together in the same partition. The scalability of the clustering algorithm can also be affected by graph partitioning. If done effectively, it allows the algorithm to run efficiently on a larger number of compute nodes and scale to larger graphs and higher throughput. And as previously mentioned, partitioning the graph can also reduce the amount of communication between computing nodes. If each node only needs to communicate with nodes in its own partition while running the clustering algorithm, then communication can be limited to a smaller subset of the graph, reducing the overall communication overhead. Therefore, our analysis will take into account different partitioning schemes and their impact on distributed clustering algorithms.

Our contributions are the following:

- Modified Apache Giraph from a vertex-centric to an edge-centric graph processor.
- Custom partitioner logic to load graph vertices into partitions/workers.
- Ran triangle count algorithm for community detection on a subgraph of ORKUT for different partitioning schemes.

II. Motivation for Novelty/Scalability

The quality of graph partitions produced by partitioning algorithms is evaluated either by *partition metrics* or *execution metrics*.

Partition metrics enable us to evaluate the quality of partitions produced prior to actual graph processing. Some examples are, replication factor, edge/vertex balance and edge/vertex standard deviation. They are suitable for general comparison and are less costly to evaluate and measure.

Execution metrics can only be used after processing the graph. They are tightly coupled to specific applications and execution environments Examples of execution metrics include partitioning time, processing time, number of rounds performed by the partitioner, and network communication overhead.

Our motivation for this project is to conduct a thorough analysis of graph partitioning strategies best suited for distributed clustering algorithms. This study has not yet been done to the best of our knowledge. We will rely heavily on execution metrics but will consider all metrics for our study. We might have to invent a new metric, tightly coupled with the task of distributed clustering, if we find no existing metrics suitable enough for our analysis. We eventually aim to find the best partitioner for distibuted clustering and to study the role of application based partitioners over general partitioners.

III. Gaps/Related Work Review

Existing literature mostly focus on theoretical metrics and neglect the impact of graph partitioning on workload execution performance. In contrast, we will study the performance of SGP (Streaming Graph Parittioners) algorithms on distributed clustering algorithms to better understand the design space and its implications on workload performance.

Stanton and Kliot [1] formulate the streaming model for graph partitioning in the context of graph loading and propose various streaming heuristics for graph partitioning. These algorithms exhibit common characteristics: (i) they maintain a synopsis in memory rather than the entire graph, which enables them to scale to very large graphs with limited resources, (ii) they perform single-pass over the graph stream and make partitioning decisions on the fly to significantly reduce the partitioning time, (iii) they can provide significant improvements over random partitioning, even comparable to traditional offline heuristics for certain scenarios.

Guo et. al. present a performance study of edge-cut SGP algorithms that are available in Oracle PGX.D [2]. Performance evaluation and the analysis are tightly coupled with Oracle PGX.D graph analytics system and it is limited to the offline graph analytic workloads.

Verma et. al. study vertex-cut SGP algorithms that are present in three scale-out graph processing systems, namely PowerGraph, PowerLyra and GraphX [3]. Authors study the vertex-cut model and focus on the replication factor as the indicator of workload performance.

Abbas et. al. conduct a performance study of existing SGP algorithms on Apache Flink, a general purpose data processing framework [4]. Even though the study covers a wide range of SGP algorithms and workloads, it fails to capture the effect of workload and graph characteristics on the workload performance.

There are two advanced partitioning algorithms that operate on the edges of streaming data: HDRF (High Degree Replicated First) [5] and DBH (Degree Based Hashing). DBH uses a hash table of size $O(V)$ to maintain the degree of the vertices of incoming edges incrementally. When a new edge arrives, DBH chooses the vertex with the lower degree and employs a hash function on its ID to assign it to a partition. On the other hand, HDRF prioritizes the replication of high-degree vertices from incoming edges, as they can aid in distributing the load and decreasing the overall replication factor. The algorithm also considers the current load of a partition when evaluating its score to minimize imbalance. HDRF involves communicating with all partitions for each incoming edge, which results in a substantial communication overhead.

To the best of our knowledge, ours will be the first study to consider online SGP algorithms tightly coupled with a distributed clustering workload. In addition, we will also try to draw a conclusion for best choices of partitioners, given a workload.

IV. Technical Approach

Our approach broadly consists of three sequential steps: *graph partitions*, *distributed clustering* and *evaluation*.

A. Graph Partitions

There are two mainstream graph partitioning strategies, *edge-cut* and *vertex-cut* partitioning, both of which are used to optimize objectives of load-balancing and min-cut (for either edges or vertices), so that the overall performance of distributed graph systems can be improved. The vertex cut partitioning strategy evenly assigns graph edges to distributed machines in order to minimize the number of times that vertices are replicated. Theoretically and empirically, vertex-cut partitioning is proved to be significantly more effective than its counterpart for web graph processing because most real graphs follow power law distributions.

Edge-cut graph partitioning partitions a graph in such a way that the number of edges crossing the partitions is minimized. Minimizing the number of edges crossing the partitions can reduce communication costs and improve the performance of the parallel algorithm. Edge-cut graph partitioning algorithms typically use heuristics to partition the graph.

We used the following partitioning schemes:

- Degree Based Hashing (DBH)
- High Degree Replicated First (HDRF)
- BTH

B. Giraph Modification

Apache Giraph is an iterative graph processing framework for distributed system which implements Google's Pregel model. It provides an abstraction to run graph algorithms using a vertex-centric model. It uses Hash partitioner by default and HDFS to store large graph datasets.

We successfully extended the framework to adapt to edge-centric algorithms. Giraph currently iterates over vertices and runs a user-defined compute function on each active vertex. We modified it to iterate and run the compute function on each edge instead. Originally, it does not deal with duplicate vertices, but our modification now supports that for edge-centric algorithms.

Partitioner	Vertices/ Edges	Running Time
BTH	v = 413920 e = 2307092	16m 58s
DBH	v = 272367 e = 1530557	13m 31s
HDRF	v = 302179 e = 1705851	14m 24s

TABLE I
RUNNING TIMES OF TRIANGLE COUNT ON ORKUT SUBGRAPH FOR DIFFERENT PARTITIONERS.

C. Partitioner Logic

The partitions created as a result of the partitioning algorithms mentioned previously, store the graph as a list of paired vertices, representing edges. In case of cross partition edges, the same vertex ID is replicated with another vertex in a separate partition file. Giraph however, does not accept replicated vertex IDs. Thus we had to modify the partition edge lists as an adjacency list as follows. Each vertex ID is modified to include the partition/worker index ID it resides in, after a decimal point. For example, vertex 100 in partition 2 will now be vertex 100.2 representing the partition, along with the vertex. This allows us to aggregate all of the edges in a single record.

Thus, instead of edges being represented as a pair of vertices: $(100, 150), (100, 170), (100, 200)$, we can now represent them as lists, along with the partition index info: $[100.0, 0, [[150.0, 0], [170.1, 0], [200.3, 0], [100.6, 1]]]$. The additional $0/1$ after the comma represents a vertex/edge. This was done to include replicated vertices in the same list with replicated vertex being indexed with a 1 after the comma, as shown in the last entry.

This logic allows us to instruct Giraph on how to distribute vertices among workers explicitly. However, the framework took too long to load the input file and send the vertices to the respective partitions/workers as the vertex objects being sent were to large in size.

D. Triangle Count

Triangle count community detection algorithm is a graph clustering algorithm that uses the number of triangles in a graph to identify densely connected communities. A triangle in a graph is a set of three vertices that are mutually connected by edges. The algorithm works as follows: first, the number of triangles that each vertex participates in is computed. Then, vertices are grouped together based on the number of triangles they participate in, with vertices that participate in a high number of triangles being grouped together into communities.

The triangle count community detection algorithm is effective in identifying communities in large graphs, particularly those with a high degree of clustering or transitivity. It has been used in several real-world applications, including social network analysis, bioinformatics, and recommendation systems. One limitation of the triangle count community detection algorithm is that it is sensitive to noise and outliers in the graph, which can result in suboptimal community assignments. Additionally, the algorithm can be computationally expensive for very large graphs, as the computation of triangle counts requires significant computation. Nonetheless, the triangle count community detection algorithm remains a useful and effective graph clustering algorithm for many applications.

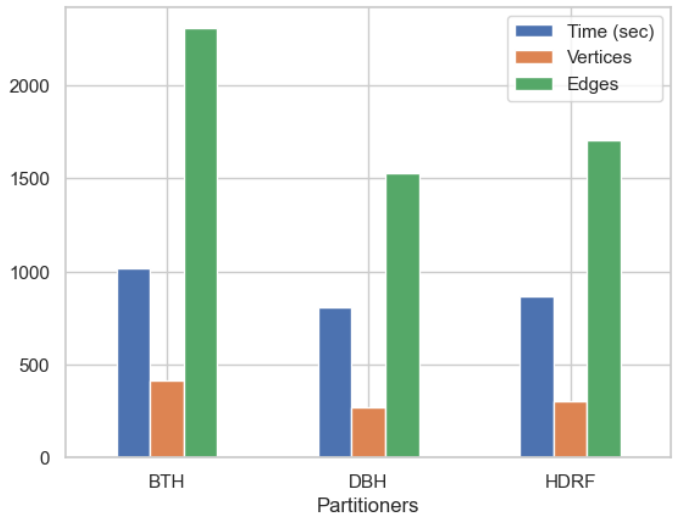

Fig. 1. Running times of triangle count on ORKUT subgraph for different partitioners.

We ran the triangle count algorithm on a subgraph of the ORKUT graph dataset, for different partitioners. Running time was chosen as the metric to evaluate performance. The comparitive results can be seen in Figure 1. Actual numeric values are given in Table I.

V. IMPLEMENTATION

This section gives hardware specifications and the degree of completion of the project. Code repository: https://github.com/paritosh-101/DS_256_Project2023

A. Hardware Environment

We ran our experiments on a commodity cluster with 10 compute nodes, each with an Intel Xeon Gold 6208U CPU with 16 cores (32 HT)@2.90 GHz, 128 GB RAM and 1 Gbps Network Interface, running CentOS 7, Java 8, Apache Hadoop v3.1.1, Spark v3.3.1 and Apache Giraph v1.3.0. Giraph is configured to use one partition per worker and we use 8 workers.

B. Completion

We analysed different partitioners on a subgraph of the ORKUT graph data as shown in Figure 1. We were unable to use other graph datasets such as LIVJ and Brain, as was proposed.

We managed to run the Triangle Count algorithm on the graph data mentioned previously. We were unable to run other clustering algorithms like K-means and DBSCAN on the data.

We managed to evaluate our runs based on the time it took to run said clustering algorithm. We were unable to perform an evaluation based on other metrics such as replication factor and communication calls between worker nodes.

VI. Conclusion

As seen from the results, the DBH partitioning strategy gives the lowest running time for the triangle count clustering algorithm. This result can be compared with the running time for HDRF due to the similar number of vertices and edges as a result of the partitioning scheme. However, the results might need to be normalised due to the large number of edges as a result of the BTH partitioning scheme.

All of the aforementioned limitations to the completion of our project can be attributed to the non-resolution of bugs in using the modified version of Giraph, unexpected running times of Giraph procedures and poor communication and time management between team memebers.

Furthermore, we will need to optimise Girpah to handle our partitioning logic in order to reduce the runtime of sending vertices to appropriate partitions/workers. This will allow us to run the objectives in our proposal thoroughly as originally planned.

References

[1] Isabelle Stanton and Gabriel Kliot. 2012. Streaming Graph Partitioning for Large Distributed Graphs. In Proc. 18th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining. 1222–1230.

[2] Yong Guo, Sungpack Hong, Hassan Chafi, Alexandru Iosup, and Dick Epema. 2017. Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. J. Parallel and Distrib. Comput. 108 (2017), 106 – 121.

[3] Shiv Verma, Luke M. Leslie, Yosub Shin, and Indranil Gupta. 2017. An Experimental Comparison of Partitioning Strategies in Distributed Graph Processing. Proc. VLDB Endowment 10, 5 (2017), 493–504.

[4] Zainab Abbas, Vasiliki Kalavri, Paris Carbone, and Vladimir Vlassov. 2018. Streaming graph partitioning: an experimental study. Proc. VLDB Endowment 11, 11 (2018), 1590–1603.

[5] Fabio Petroni, Leonardo Querzoni, Khuzaima Daudjee, Shahin Kamali, and Giorgio Iacoboni. 2015. HDRF: Stream-Based Partitioning for Power-Law Graphs. In Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15). Association for Computing Machinery, New York, NY, USA, 243–252.

[6] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In ICPP, pages 113–122, 1995.

[7] D. Kong, X. Xie and Z. Zhang, "Clustering-based Partitioning for Large Web Graphs," 2022 IEEE 38th International Conference on Data Engineering (ICDE), Kuala Lumpur, Malaysia, 2022, pp. 593-606.

[8] Anil Pacaci and M. Tamer Özsu. 2019. Experimental Analysis of Streaming Algorithms for Graph Partitioning. In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19). Association for Computing Machinery, New York, NY, USA, 1375–1392.

[9] M. Hai, S. Zhang, L. Zhu and Y. Wang, "A Survey of Distributed Clustering Algorithms," 2012 International Conference on Industrial Control and Electronics Engineering, Xi'an, China, 2012, pp. 1142-1145.