Towards Clustering Graph Representation of Geographic Dataset for Distributed Systems

Subhadeep Bhattacharya† Fahim Chowdhury† Rupak Roy†
Florida State University†
{bhattach,fchowdhu,rroy}@cs.fsu.edu

I. Introduction

Graph representation of huge dataset used for solving real world problems is a popular format to feed computable data to the datamining applications. Oftentimes, these data structures can be complicated with respect to its connectivity among the elements of the representation. This dependency among the datapoints or nodes of the graph not only makes it a complex task to extract the insights from the data through datamining, but also renders the application workflow as a hindrance to the parallelization for efficiently leveraging high performance computing (HPC) infrastructures. The goal of the project is to perform research on applying clustering techniques on large graph representation of huge real world dataset in order to maximize data parallelism in the datamining models for utilizing HPC resources while ensuring reasonable accuracy of the system.

In particular, the research intends to initialize satisfying the general motivation of applying graph clustering to improve parallelism by focusing into a special and narrow sector as a proof of concept. This sector deals with feeding large geographic map data to a system for preprocessing and formatting the data in graph representation where a coordinate, i.e., tuple of longitude and lattitude of a specific geographic point, is treated as a node and the connecting road between two coordinates is presented as the edge. This large graph presenting the map can be used by the system to solve different problems in the transportation system. For example, this data can be used for finding the shortest path between two points of the graph.

Using legacy algorithms like Dijkstra's algorithm or even comparatively more modern techniques like A* search will pose immense time complexity if run on sequential systems. Performance maximization via using distributed HPC systems can only be possible if divide and conquer strategy can be applied on the data representation by clustering the graph into smaller chunks and run the algorithm upon the lower volume data in parallel, and finally combining all the small results to deduce the final outcome.

Taking all these research requirements into consideration, this project proposes the following contributions.

- A thorough survey on the state-of-the-art graph clustering techniques in general.
- Application of different graph clustering on huge geographic data as a special case handling.
- Execution of a shortest path algorithm as a proof of concept for evaluating the efficacy of different clustering techniques.
- Verification of performance and accuracy of the application for different clustering on HPC systems via thorough experiments.
- Comparison among different clustering techniques and recommendation for the specific data workload in concern, i.e. geographic map data.

II. Literature Survey

Clustering the graph representation for geographical dataset continues to gain huge attention as it shows its potential for solving many real world problems. However, time taken by the graph algorithms is still an important challenge for real-time analysis which can be minimized by using the HPC infrastructure. In [1] Bogle et. al. presents a new distributed-memory, BFS based parallel algorithm to identify and remove disconnected components or hinge vertices for graph-algorithms while significantly improving the convergence of iterative solvers. This work is extremely related in our context as the objective is to develop an efficient and accurate parallel algorithm for geographical dataset using its graphical representation.

Clustering the graphs in each node plays an important role in solving this scenario and state of the art

The authors are listed in alphabetic order of the last name. All the authors have equal contribution.

researches need to be explored in this regard. Efficient semi-supervised learning can be one of the choice while doing the graph clustering. Dhillon et. al. [2] shows semi-supervised learning can improve the graph clustering results and states an objective function for weighted kernel k-means by using Hidden Markov Random Fields, certain class of constraint penalty function and squared Euclidean Distance. Kulis et. al. [3] proposes SS-KERNELKMEANS to optimize a semi-supervised clustering objective for graph-base inputs by proposing a more generalized formulation and also exhibits an equivalence between the kernel k-means objective function and a special case of the HMRF-based semi-supervised clustering objective.

Detecting densely connected groups in a large graph is also a primary requirement in our approach. Zhou et. al. [4] proposes SA-Cluster, a novel graph clustering algorithm based on both structural and attribute similarities using a unified distance measure. This method considers the heterogeneous vertex properties apart from the overall topological structure for clustering and shows its effectiveness both theoritically and experimentally.

Even the possibilities of using deep learning in graph clustering is explored by Tian et. al. [5] and the work illustrates that it has strong theoritical foundation. The autoencoder based approach used in this paper also outperforms conventional spectral clustering with much lower computational complexity. Bruna et. al. [6] investigates a generic family of graph neural networks and shows data-driven approaches to clustering can be used with real dataset. Even it shows that the data-driven approach can consume less computational steps while performing better than other rigid parametric models.

III. Methodology

This section discusses the characteristics and source of geographic map data, various clustering techniques and algorithms to find the shortest path between two data points.

A. Geographic Dataset

The data for this project will be collected from OpenStreetMap [7], which is an open source platform for maintaining map data of the entire earth. There are several sources of getting the data, e.g., extracting directly from the website for a small region or the entire planet, or continent, country or state based data from Geofabrik Download Server [8], etc. Data normally comes in the human-readable XML formatted .osm files. A basic format of the data presenting a datapoint by $node$ and a connected path by way XML DOM element.

The $node$ and way information can be used to construct a weighted adjacency matrix denoting the entire map, where weight of an edge is the distance between two nodes.

B. Clustering Techniques

One of the most vital goals of this project is to perform research on different clustering techniques when applied on different graph datasets. We have conducted a thorough study starting from very basic clustering like K-means and hierarchical clustering to advanced techniques discussed in Sec. II. [1], [4], [5] Specifically, some assumptions regarding transportation network can be leveraged to set as constraints in the clustering techniques in order to simplify the transportation network graph processing and also to acquire faster convergence and ensure higher parallelism with the help of clustering.

1) Shortest Path Parallel Algorithm: Using the above heuristics, the algorithm for finding a shortest path between two vertices after clustering can be devised as mentioned in Algorithm 1.

C. Algorithm Description

Our proposed algorithm can be divided into 4 steps.

- Clustering graph
- Gateway Detection
- Intra-cluster shortest path detection
- Inter-cluster shortest path detection

1) Clustering Graph: As the transport network will not change rapidly over time, we are doing the clustering in a pre-processing step. This pre-processing phase will not affect the run time of our proposed algorithm. In this phase, we are extracting the data from the web and feeding the necessary information to our data structure. Then we are passing the data through various clustering algorithms e.g. Kmeans clustering, Agglomerative clustering. For our proposed algorithm, we are clustering entire dataset into small segments. In Fig. 1, the whole graph has been partitioned into the 3 small clusters making it suitable for running our proposed distributed algorithm.

2) Gateway Detection: After clustering the dataset, there are still some gateway edges, which connect the clusters. These gateway edges are necessary for computing the overall shortest path. For detecting these edges, we select the nodes who have at least one of their edges connected to any of the nodes in other cluster. We call this nodes gateway nodes. In this step, alongside with

Algorithm 1 Find Shortest Path with Clustering

Input: graph, src, dest, cluster_list
Output: path

1: Create subgraphs using clustering technique
2: Construct $graph_c$ representing the gateway nodes as nodes and the weighted connection between them as edges. Both inter and intra cluster gateway connections are taken into consideration.
3: **if** $(compute\_node == master)$ **then**
4: **if** $(src.cluster\_id == dest.cluster\_id)$ **then**
5: Send both src and dest information to $cluster\_list[src.cluster\_id]$
6: **else**
7: Send src to $cluster\_list[src.cluster\_id]$
8: Send $dest$ to $cluster\_list[dest.cluster\_id]$
9: Apply $FindShortestPath$ in $graph_c$
10: Merge the $shortest\_paths$ after getting required information from slaves
11: **return** $path$ found by the $shortest\_path$ search algorithm
12: **end if**
13: **else if** $(compute\_node == slave)$ **then**
14: $cluster = cluster\_list[compute\_node]$
15: **if** $(src.cluster\_id == dest.cluster\_id)$ **then**
16: $FindShortestPath$ with $(cluster, src, dest)$
17: **else**
18: **if** $(src \in cluster)$ **then**
19: Apply $FindShortestPath$ with $(cluster, src, gateways)$
20: **else if** $(dest \in cluster)$ **then**
21: Apply $FindShortestPath$ with $(cluster, gateways, dest)$
22: **else**
23: Apply $FindShortestPath$ with $(cluster, gateways, gateways)$
24: **end if**
25: **end if**
26: Send required information to master
27: **end if**

detecting gateway edges we also detect the subgraphs of inter-cluster gateways.

In Fig. 2, the gateway nodes have been marked with yellow and the gateway edges have been colored green. These gateway edges connects the three clusters externally.

3) Intra-cluster shortest path detection: For detecting the shortest path in the clusters we run single source shortest path algorithms e.g. dijkstra's algorithm, indi-

Fig. 1. Graph clustering

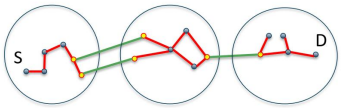


Fig. 2. Detection of gateways

vidually in the different clusters. Based on the given source and destination we classify the clusters as source cluster, destination cluster and intermediate clusters. In the source cluster, we find all the shortest paths from source to gateway nodes. In case of intermediate clusters, we determine the shortest paths between all possible gateways. In the destination cluster, shortest path is detected from all gateway points to the destination nodes. In Fig. 3, in the source cluster, marked as S, we compute

Fig. 3. Detection of intra-cluster shortest paths

the shortest path from source node to the two gateway nodes colored yellow. In the intermediate cluster, shortest paths are detected between all possible gateways. in the destination cluster, marked as D, we compute the shortest paths from gateway nodes to destination node.

4) Inter-cluster shortest path detection: In this step, shortest paths calculated between inter-cluster edges. After creating the subgraphs of gateway nodes and edges, single source shortest path algorithm is applied on each gateway subgraph. the goal is to find the shortest inter-cluster paths.

Fig. 4. Detection of inter-cluster shortest paths

In Fig. 4, the inter-cluster shortest paths have been colored as blue.

IV. Implementation Details

This section focuses on an elaborate description of the implementation details of distributed graph traversal algorithms. Also, it describes certain assumptions and challenges faced during the implementation phase.

A. Assumptions

Before diving deep into the implementation details, it is necessary to describe certain assumptions we have considered to make the graph traversal algorithm distributed in nature. First of all, the clustering methodology is incorporated to divide the data into partitions with the nearest nodes. Each node in our dataset consists of the real-world location taken from the geographical dataset mentioned in the previous section. We have considered the data-parallel approach with master-slave architecture for designing our distributed algorithm. Each slave is assigned with a cluster of nodes along with the graph consists of connectivity between them. Once a source and destination point is supplied, the master node will find out the cluster to which the source and destination points belong to. In some instances, the source and destination can belong to the same cluster, which is our best-case scenario. In this case, instead of traversing the entire graph, the shortest path algorithm is run in that particular cluster with full computation power, which will lead to less time than traversing the entire graph. However, if the source and destination do not belong to the same cluster, the master node detects the clusters with source and destination node and sends them to the corresponding nodes for computation.

Secondly, clustering is done during the data processing phase, which is before starting the actual graph traversal. We assumed that the number of clusters is equal to the number of slave nodes. Hence, we can easily assign each cluster to a slave for shortest path computation.

Finally, both partitioned based and hierarchical clustering is used for generating the distributed data partitions. K-means is used as a representative of partitioned based clustering whereas Agglomerative clustering is used for the hierarchical one. It is expected that an excellent clustering method will detect a neighborhood with the smallest distances between each pair of locations.

B. Implementation Details

Our entire methodology can be divided into two steps. The first part is to divide the entire graph network into clusters of neighbors, and the second part is to parallelize the shortest path algorithm.

Python is used as the default language for our implementation, and mainly the functionality provided by the NetworkX package is used for different graph-related algorithmic implementations. The data import pipeline is implemented using APIs provided by NetworkX and OSMNX. First, we converted the Open Street Map data into a graphical representation using OSMNx, and then it is supplied to the NetworkX to generate a NetworkX graph. After the graph is generated, we run clustering techniques over that graph to divide the data and stored it in their respective nodes as a pickle object. Each graph object will consist of some nodes which provide a gateway to other nodes. We select the nodes in each cluster that have an edge connecting to the other cluster and defined them as gateway nodes. After the gateway nodes are detected, we also store that information into a pickle for further processing.

Although we can divide the data into different clusters, we need to find the path among clusters. This path will primarily consist of all the gateways. We first find out which gateways inside a node are connected to each other. If there is a path between two nodes, we join then with a single edge to represent the existence of an intra-cluster gateway path. So, we finally have all the gateway nodes and their connection information, which we used to construct a gateway to gateway graph and store that information in the master node for further processing.

In the second phase, we tried to parallelize the shortest path algorithm using data-parallel approach. After the

designated cluster is located with the information provided by source and destination, we divided our tasks into mainly 4 kinds of subtasks as follows:

- Determine the shortest path from source to the gateway nodes of the respective cluster
- Determine the shortest path from destination to destination for the respective cluster.
- Find out the gateway to gateway shortest paths inside each cluster other than source and destination
- find the shortest path between inter-cluster gateways

The master node determines the respective cluster for source and destination, and it sends the source and destination node to their respective clusters. The source cluster tries to find out the shortest paths from the source node to the gateways. Also, the destination node tries to find the shortest path from the gateway to the destination in the destination cluster in parallel. Other nodes also try to find the shortest gateway to gateway paths in their cluster. The master node loads the graph of gateways from the pickle and tries to find out the inter-cluster shortest path using gateways. After all these four parallel processes completed, the respected results are sent to the master node. Finally, the master node uses the provided information and tries to find out the shortest path between source and destination by merging the paths provided in previous steps. MPI is used for running the tasks in parallel and also for the exchange of necessary pieces of information between master and slaves. Our implementation also has the provision to designate one node to run as both master and slave in parallel.

C. Challenges

We have faced several challenges during our implementation phase, both in terms of accuracy and latency of our algorithm. The primary challenge was to prepare an inter-gateway graph creation, which will give us necessary information about the existence of inter-cluster shortest paths. We tried to keep the graph as simple as possible by only keeping the gateway nodes in this graph and replacing the entire path between them with a single edge if there is any path between two gateway nodes. However, the most complicated part of our implementation was to merge the information provided by each cluster and use the gateway graph information to determine the shortest path from source to cluster. In the initial step, we tried to call the single source shortest path algorithm multiple times to find out the shortest path from source gateway to the destination gateway, which increases the latency of our algorithm significantly. However, we solve that issue by using a multi-source shortest path for each destination, and it improves the overall performance of our implementation. Further, we face some challenges to join the source and gateway paths as we can get multiple shortest paths for each of the gateway nodes. We keep them in sorted order by distance and use that notion while doing the merging at master.

So overall, our algorithm tries to find out the shortest path between the source and the gateway nodes, the shortest path between inter-gateway nodes, and also the shortest path between gateway nodes and destination. Finally, it merges the paths with the shortest distances, which give a complete shortest path from source to destination.

V. Experimental Results

In this section, we discuss the experimental setup of the implementation evaluation. We demonstrate how the experiments were performed step-by-step by solving problems that we faced during implementation. We state the latency and accuracy of the implemented parallel shortest path algorithm based on Dijkstra's single pair shortest path algorithm for different clustering techniques and configurations. For each of the experiments, we run 10 times by randomly picking source and destination nodes for the shortest path. Each time we run for all types of clustering configurations for fair comparison of accuracy and latency.

A. Experimental Setup

1) Testbed: We run the experiments on *Innovation*, an HPC cluster maintained by the Computer Architecture and Systems Research Laboratory (CASTL) [9]. In this cluster, each node has 10 dual-socket Intel Xeon(R) CPU E5-2650 cores, 64 GB memory, and a 1 TB Seagate ST91000640NS SATA disk. The communication network is build on FDR InfiniBand interconnect with Mellanox ConnectX-5 NIC.

2) Configurations: In the experiments, we have used MPICH-3.1 as the communication substrate for running in the distributed environment. We follow a data parallel approach where each clustered graph is assigned to each client in the system. We choose the total number of processes equal to the number of clients, so that each client is assigned exactly one process.

B. Evaluation

In this section, we discuss the latency breakdown on different phases of the implementation process followed

by accuracy measurement. We use MPI.Wtime for logging the time of each part of the total execution time. For each of the clustering techniques, we compare the output, i.e., final shortest path, with the path determined from the experiments on clustering with k=1 where the whole graph data is fed to the system. We demonstrate these two metrics by increasing the number of clusters from 1 to 8 in geometric order for both K-means and Agglomerative clustering. We dumped pickle files for each cluster when we preprocessed the whole graph representation of the map of Tallahassee offline using `prepare_cluster.py` file. As shown in Fig. 5, in the intial stage implementation, we run the experiments for both K-means and Agglomerative clustering. We analyze the latency breakdown to pinpoint the exact cause of the performance overhead and found that the main bottleneck was in the user space instead of being in the kernel space for communication due to the distributed architecture. So, we improved the algorithm and ran all the experiments again for all clustering configurations as demonstrated in Fig. 6

1) Latency Evaluation: We measure the latency for both K-means and Agglomerative clustering in the initial implementation and the improved imeplementation. In case of detecting the shortest path using the data clustered through K-means clustering, we observe that most of the latency overhead occur in the user space. As shown in Fig. 5(a) for K-means, with increasing number of clusters, the time measured for the user space application execution increased drastically from 2.53 seconds for 1 cluster to 57.06 seconds for 8 clusters, while the kernel space latency does not increase that much, i.e., from 1.82 seconds for 1 cluster to 10.08 seconds for 8 clusters. Even though latency improvement was expected due to incorporating additional HPC resources, the high deependency among each clusters for the geographic map data made difficult for the application and kernel to resolve the dependency among parallel units of data. On the other hand, the data clustered using Agglomerative clustering technique behaved much better than that clustered using K-means. For instance, as shown in Fig. 5(d), the user space latency increased less steeply, i.e., from 2.93 seconds for 1 cluster to 11.049 seconds for 8 clusters. This is about 90% improvement for 4 clusters and about 80% improvement for 8 clusters. The kernel space latency also improved a lot for Agglomerative clustering. It was only 1.924 seconds for 1 cluster, stayed almost the same for 2 and 4 clusters, and increased slightly to 3.1 seconds for 8 clusters.

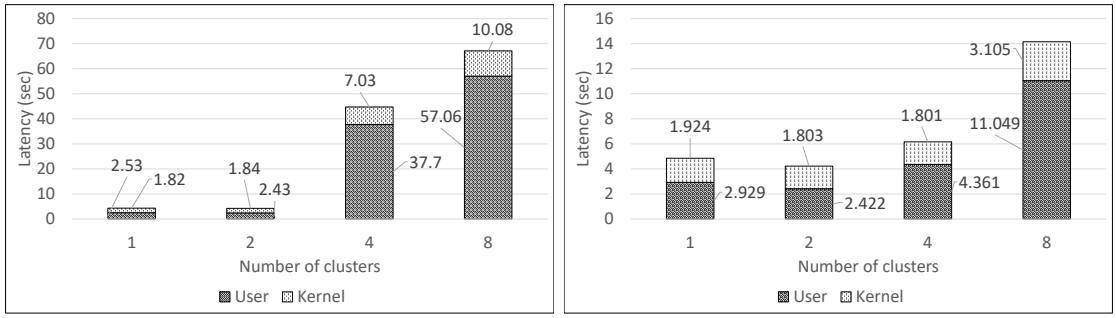
Although we observed improvement by using data clustered by Agglomerative clustering technique, the latency measures were not only higher than what we anticipated while planning the project, but also it was taking a lot of time in the user space. This means there are problems in the algorithm implementation. After some profiling, we pinpointed the code snippet that was creating performance bottleneck, i.e., the all source gateways to all destination gateways shortest path determination. After this improvement, we ignored the kernel space latency and only focused on the user space latency. We further break the latency down to data loading time from preprocessed pickle files and actual execution time of the distributed algorithm. As shown in Fig. 6(a), for K-means, the latency increased from 0.12 seconds for 1 cluster, increased to 5.24 seconds for 4 clusters, and decreased to 2.28 seconds for 8 clusters. We observe about 56% improvement from cluster 4 to 8. On the other hand, for Agglomerative clustering, as stated in Fig. 6(b), the latency for the algorithm execution increased from 0.17 seconds for the whole graph to 3.40 seconds for 2 clusters. Later, it had about 65% decrement to 1.17 seconds for 4 clusters and increased a little to 1.64 seconds for 8 clusters. For choosing Agglomerative clustering over K-means clustering, we can achieve about 78% improvement in latency in this improved case. For data loading time, it decreased slightly from 0.42 seconds for whole graph to 0.13 seconds for 8 clusters for K-means, and 0.41 seconds for whole graph and 0.14 seconds for 8 clusters for Agglomerative clustering. This is obvious, because with increased cluster numbers, we assign more HPC resources and each computing client have to read less amount of data.

2) Accuracy Measurement: In the accuracy measurement, we compare the paths reported by the algorithm using the data from each clustering configuration for different cluster numbers with the same path found from running the algorithm in 1 cluster or the whole graph. We know that the whole graph contains all the connectivity and the path detected from whole graph is the ideal case scenario. For calculating the output accuracy, we followed the equation below.

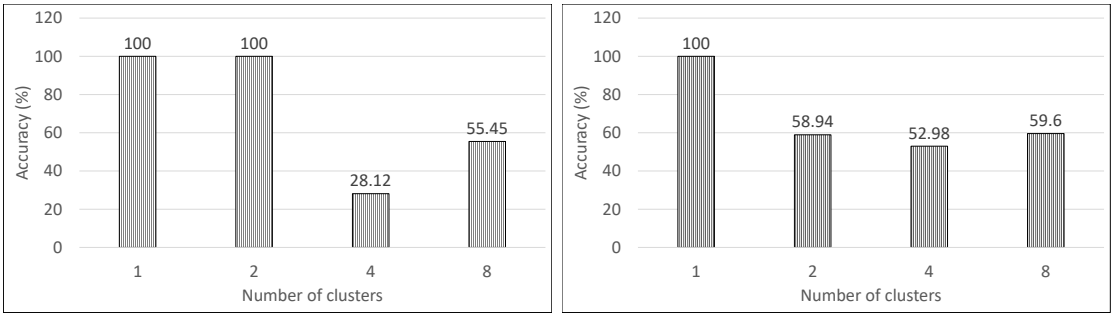
$$accuracy = n_c/n_i$$

where, n_c = number of common nodes in ideal and given path, and n_i = number of nodes in the ideal path.

As shown in Fig. 5(c), for K-means clustering, the accuracy is 100% for 2 clusters, but it drastically drops to 28.12% for 4 clusters and got a little better to 55.45% for 8 clusters. On the contrary, for Agglomerative clustering, as depicted in Fig. 5(d), we observed better accuracy

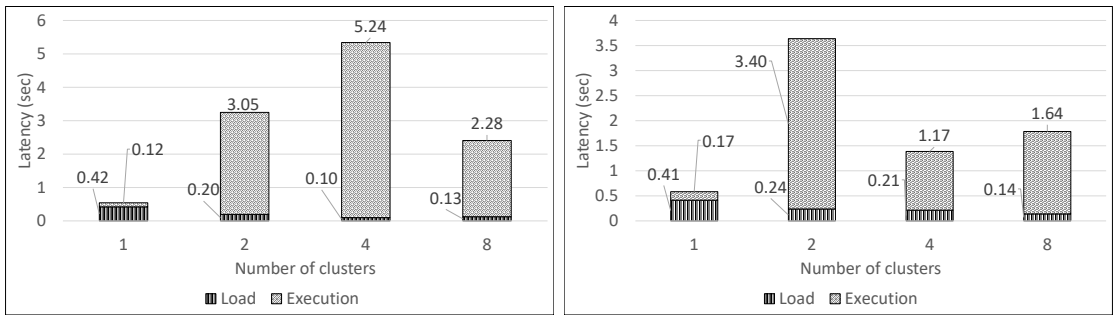


(a) Latency (K-means) (b) Latency (Agglomerative)

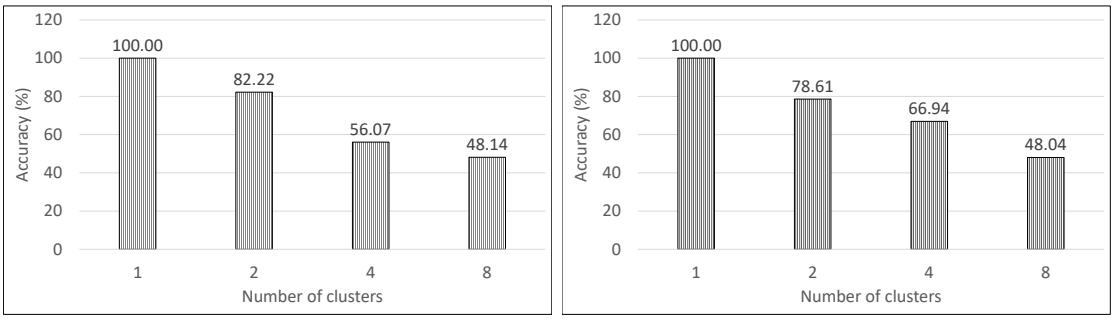


(c) Accuracy (K-means) (d) Accuracy (Agglomerative)

Fig. 5. Initial latency breakdown and accuracy for distributed shortest path detection algorithm on data clustered using K-means and Agglomerative clustering



(a) Latency (K-means) (b) Latency (Agglomerative)



(c) Accuracy (K-means) (d) Accuracy (Agglomerative)

Fig. 6. Improved latency breakdown and accuracy for distributed shortest path detection algorithm on data clustered using K-means and Agglomerative clustering

trend than that for K-means clustered data, though it is quite less than what is expected in general. For number of clusters more than 1, the accuracy stays in the range to 52.98% to 59.6%.

We measured the accuracy again after the code optimization discussed in Sec. V-B1. This latency improve-

ment impacted the accuracy a little, but in general it reported better accuracy trend for different clustering configurations. For instance, as shown in Fig. 6(c) for K-means clustering, the accuracy dropped from 82.22% to 48.14% for 2 and 8 clusters respectively. For Agglomerative clustering, as depicted in Fig. 6(d), it decreased from 78.61% to 48.04% for 2 and 8 clusters respectively. Only for 4 clusters, data clustered by Agglomerative technique reports better accuracy (66.94%) than that (56.07%) found by K-means data. In general, K-means and Agglomerative clustering demonstrate similar accuracy trends.

a) Remarks: The accuracy of the algorithm decreases with increasing number of clusters, because incorporating the clustered data loses the inter-cluster connectivity information of the entire graph while running the algorithm in the clustered data unit assigned to each process per client. Finally, after determining the final path by reducing all the connectivity information via the gateway graph, we end up not finding the optimal solution. Nevertheless, we get a suboptimal solution that represents a path from source node to the destination node.

VI. Conclusion and Future Work

With the easy representability, graph has become a popular data structure for simplifying real world computation intensive problems. These real world problems are usually sequential and difficult to accelerate using modern distributed HPC systems. This work addresses graph parallelization problem, with the use of clustering approach by incorporating a master-slave architecture with the traditional single source shortest path algorithm. Our approach ensures data parallel execution of the algorithm in different partitions of the entire graph with a view to making it suitable for modern HPC systems. We have made a comparative analysis between partitioned-based and hierarchical clustering when applied on graph data structure. Our experiments show that hierarchical agglomerative clustering outperforms kmeans clustering in terms of accuracy and latency for graph parallelization. Due to the high connectivity among the different clusters of the graph data, the final reduction process to generate final output from the sub-outputs from each clusters is a challenging task even for the cutting edge HPC systems. There are still some scopes of improvement which we would like to leave as future work. Extending the existing clustering algorithms to reduce the inter-cluster edges can help us accelerate the proposed algorithm with better accuracy. Moreover, optimizing the gateway detection algorithm further can alleviate the latency degradation caused due to shortest path merging phase. Evaluating our approach for heuristic based algorithms, e.g., A* search, can also be an interesting line of study.

References

[1] I. Bogle, K. Devine, M. Perego, S. Rajamanickam, and G. M. Slota, "A parallel graph algorithm for detecting mesh singularities in distributed memory ice sheet simulations," in *Proceedings of the 48th International Conference on Parallel Processing*. ACM, 2019, p. 1.

[2] I. S. Dhillon, Y. Guan, and B. Kulis, "Kernel k-means: Spectral clustering and normalized cuts," in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '04. New York, NY, USA: ACM, 2004, pp. 551–556. [Online]. Available: http://doi.acm.org/10.1145/1014052.1014118

[3] B. Kulis, S. Basu, I. Dhillon, and R. Mooney, "Semi-supervised graph clustering: a kernel approach," *Machine learning*, vol. 74, no. 1, pp. 1–22, 2009.

[4] Y. Zhou, H. Cheng, and J. X. Yu, "Graph clustering based on structural/attribute similarities," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 718–729, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687627.1687709

[5] F. Tian, B. Gao, Q. Cui, E. Chen, and T.-Y. Liu, "Learning deep representations for graph clustering," in *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, ser. AAAI'14. AAAI Press, 2014, pp. 1293–1299. [Online]. Available: http://dl.acm.org/citation.cfm?id=2893873.2894074

[6] J. Bruna and X. Li, "Community detection with graph neural networks," *stat*, vol. 1050, p. 27, 2017.

[7] "OpenStreetMap," https://www.openstreetmap.org.

[8] "Geofabrik Download Server," https://download.geofabrik.de.

[9] "Computer Architecture and Systems Research Laboratory (CASTL)," https://castl.cs.fsu.edu.