Implementation of a Parallelized K-Medoids Algorithm Suitable for Gene Clustering

1st Vasundhara Acharya
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
acharv2@rpi.edu

2nd Harshaa Narayan
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
hiremh@rpi.edu

3rd Dhruva Narayan
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
narayd@rpi.edu

4th Runbin Chen
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
chenr13@rpi.edu

5th Christopher D. Carothers
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, USA
chrisc@cs.rpi.edu

***Abstract*—Clustering is a critical step in the processing of gene data. In this study, we accomplish gene clustering using the k-medoids approach. Initialization and iterative medoids refinement are the two key phases of the k-Medoids algorithm. To begin with, k-Medoids are chosen at random from the dataset, and then they are iteratively refined to reduce the separation between the points in each cluster. After a predetermined number of iterations, the Silhouette score is used to assess how well the clustering performed. The AML/ALL and Rice datasets are used in the experimentation. The experiments were conducted on the AiMOS supercomputer. To evaluate the clustering quality, we use different distance metrics, including Euclidean and Manhattan. However, computing pairwise distances between all data points to find the closest medoids can be time-consuming for large datasets. To address this issue, we parallelize the k-medoids algorithm using Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA). Finally, we compare the performance of the parallelized version with the serial version of the codes. Heatmaps were used to display the cluster assignments that were derived from the experiments. The outcomes demonstrated that the Manhattan distance metric produced the best outcomes for the Rice dataset, whereas the Euclidean distance metric was the best metric for the AML/ALL dataset. Additionally, we evaluated the performance of MPI file reading with 8 MPI ranks versus serial file reading, and we discovered that the MPI file reading operation was 44.925667 times faster. By removing sampling from the serial code, we were also able to compare the parallel and serial versions of the code fairly. The code with 8 MPI ranks was 25.6174923547 faster than the serial code with Manhattan distance metric and MPI code with 8 MPI ranks is 45.8304664498 faster than the serial code with Euclidean distance metric in terms of clock cycles. In our experimentation on computing the Silhouette score of the AML/ALL dataset, we observed that the MPI code was 7.46985909 times faster than the serial code and 21.0581375 times faster than the serial code while using the Rice dataset. The CUDA implementation was 6.04379562 faster than serial code in clustering considering the time and 6.04757231 faster than serial code in clustering considering the clock cycles when experimented with AML/ALL dataset.**

***Index Terms*—Clustering, Gene,K-Medoids, Parallel, MPI, CUDA,**

Identify applicable funding agency here. If none, delete this.

I. Introduction

Research in genomes, proteomics, functional genomics, and biomedicine has advanced significantly during the last ten years. We are now better equipped to comprehend the intricate mechanics that underpin the operation of biological systems, such as the human body. It is now possible to analyze an organism's entire genetic makeup, including that of humans. Important details regarding the composition and operation of genes, their evolution, and their connection to disease have been discovered through the identification and comparative analysis of the genomes of humans and other species. This massive data can be analyzed using clustering algorithms. In our work, we implement the K-Medoids algorithm to achieve clustering. Clustering divides things into groups, or clusters so that objects inside the clusters are similar to one another but different from those in other clusters. The monitoring of thousands of genes' levels of expression is made possible by DNA microarray technology. In the data mining process, clustering algorithms are crucial for examining the natural structure and spotting trends in the data. The expression levels of genes, which are DNA segments that code for particular proteins, are key factors in determining the cellular phenotype. Gene expression data, which simultaneously monitors the expression levels of thousands of genes, has grown in significance in biomedical research as a result of its ability to shed light on disease causes and advance the fields of drug discovery and personalized treatment. However, because gene expression data is multidimensional and complicated, it can be challenging to infer relevant information from the raw data alone. Genes with comparable expression patterns can be grouped together using clustering algorithms, which can also be used to spot patterns that can shed light on the biological processes at work. Clustering of genes has various applications. Co-expressed genes may be involved in the same biological process or pathway, which can be determined

by using clustering algorithms, which can group genes with comparable expression patterns. This data can be utilized to anticipate the effects of perturbations and estimate the function of unidentified genes. Using clustering techniques, diseases can be categorized based on the expression patterns of the affected tissues. This can assist in identifying illness subgroups that might have various underlying causes and call for various therapies. Clustering algorithms can be used to pinpoint the genes or biochemical processes that a drug's effects would influence. Potential therapeutic targets or disease-diagnosing biomarkers can be found using this data. Clustering techniques can be used to find various biomarkers. K-Medoids and K-means are the two popular clustering algorithms used in machine learning and data analysis. K-means is a centroid-based clustering algorithm that partitions a dataset into K clusters, where K is a user-specified parameter. K-Medoids is a partition-based clustering algorithm that is similar to K-means but uses medoids instead of centroids. K-Medoids is more robust to outliers [1] and is also computationally expensive. In K-Medoids, we need to compute the pairwise distances between all the data points to find the closest medoids for each cluster, which can be time-consuming for large datasets [2]. When utilizing K-Medoids, it is important to experiment with various distance metrics because the selection of a distance metric has a significant impact on the clustering results quality. Although it is frequently employed, the Euclidean distance metric may not necessarily be the best option for all datasets. Other distance metrics like Manhattan, Minkowski, or Mahalanobis may produce better clustering outcomes in specific circumstances [3]. In the proposed work, we experiment with Euclidean and Manhattan distance metric.

The CPU's advantage is its effective low-latency architecture, which enables it to manage a limited number of threads and processes at once. Modern GPUs, in comparison, have a high throughput-oriented design and are made up of thousands of cores that can run thousands of threads concurrently. Utilizing all available computational resources is achievable by integrating the MPI and CUDA programming models, which leads to much better performance and scalability [4]. We aim to parallelize our proposed algorithm with the help of MPI and CUDA.

The rest of the paper is organized as follows: Section 2 gives related work in clustering. Section 3 explains the proposed methodology . Section 4 describes the parallel implementations of the K-Medoids algorithm and the evaluation metric . Section 5 explains the the results obtained. Section 6 concludes the paper.

II. Related Works

In the past, there have been many attempts to work with gene expression data to draw insights about various biological markers from the dataset. One of the most popular machine-learning tasks that are applied to data of this type is clustering. In [5] the authors have proposed a novel approach to clustering microarray gene expression data by combining pre-existing clustering techniques that have been used for this task in the past with SVMs to obtain a better-performing fuzzy clustering. One of these pre-existing models was a genetic algorithm-based model developed by Wu et al, as shown in [6]. In this paper, a fusion model combining a genetic algorithm approach and the iterative approach to clustering was used with great success to cluster two types of gene expression data. As an alternative to the standard k-means clustering algorithms that are very popular in this domain, Huang et al. in [7] have developed a spectral clustering-based approach to cluster gene expression data. They believe that this helps overcome inaccurate clusterings that may occur due to the inherent heterogeneity in such datasets. We also see a rise in the popularity of the k-medoid variant of clustering gene expression data in papers such as [8] and [9]. The authors have used an innovative method of the k-medoid algorithm where they use a data matrix to identify initial medoids that will give improved clustering results. They have also shown that their approach takes less time to execute overall as their approach eliminates the need to iterate over the dataset multiple times [8]. A modified version of the K-Medoids algorithm was used, which proved to work very well on many different datasets, including gene-expression data [9].

A paradigm that is being explored widely nowadays is parallel computing. With the exponential rise in the amount of data we are able to collect and process, there is a need to avoid using sequential iterations and adopt more parallel approaches. One paper that makes use of this paradigm is [10], where the authors have been able to get close to a 7% speedup in the clustering of gene expression data by using multiprocessing techniques. [11] is another paper that shows us the great benefits of using a parallelized version of the k-means algorithm. Another useful paper is [12] which compares the performance of two commonly used clustering algorithms, k-means and K-Medoids using MPI. Although the final results show that k-means produces a lower final error, the K-Medoids algorithm is also extremely effective.

Finally, innovations in parallel clustering algorithms have made their way into the domain of gene expressions, as shown in papers [13] and [14]. The authors have come up with a 2 step approach to obtain biclusters in tumor gene expression data. They use a parallel k-means algorithm in the first part of their process, followed by a bicluster algorithm. Using this approach, Ardaneswari et al. [13] were able to identify 5 biclusters in their dataset. The authors of [14] have made use of Principal component analysis in the initial pre-processing of their gene expression data to come up with more robust data-structures to perform parallel k-means. They have utilized the SNOW package in their experimentation to parallelize their algorithms.

III. Methodology

A. Proposed K-medoids algorithm

The algorithm used in the proposed work is shown in 1. It has two main steps:

- Initialization and iterative refinement of medoids: In this stage, K initial medoids are chosen at random from the

data points, and then they are iteratively refined to find the medoids that minimize the sum of the distances between the points in each cluster. This process is repeated a predetermined number of times.
- Cluster assignment evaluation using the Silhouette score: The data points are classified to clusters depending on their distance from the closest medoid after collecting the complete set of medoids. The quality of the clustering is then measured using the Silhouette score, which is produced for the final cluster assignments. For various K values, the algorithm repeats this phase in order to determine the ideal number of clusters.

B. Dataset Description

This paper employs two distinct datasets. The first dataset is a gene expression dataset sourced from a proof-of-concept study by Golub et al. [15] in 1999. The study demonstrated the use of gene expression monitoring via DNA microarray to classify new cases of cancer and thus provide a general approach to identify and categorize tumors. The dataset was used to classify patients with acute myeloid leukemia (AML) and acute lymphoblastic leukemia (ALL).

The second dataset is a rice gene expression matrix [16] derived from RNA-seq datasets on a large scale. This dataset contains a gene expression matrix utilized to construct rice gene co-expression networks. The matrix comprises expression values (log2(cpm+1)) for 33,846 genes obtained from 8,456 publicly accessible high-quality rice RNA-seq runs downloaded from the NCBI SRA database as of June 2021. But for this project due to memory constraints we have employed only 10000 genes.

TABLE I
DATASET DESCRIPTION

Dataset	Number of Records	Number of Features	Ref
AML/ALL Dataset	7129	34	[15]
Rice Gene Dataset	10000	35	[16]

Prior to clustering, the dataset underwent several preprocessing steps. First, the data was scaled using a standard scaler to ensure that all features were on the same scale and to prevent any features from dominating the clustering process due to differences in magnitude. In addition to scaling, any columns in the dataset that were deemed unnecessary for the clustering process were eliminated to reduce the dimensionality of the data and improve performance.The dataset was initially in CSV format, but was converted to binary files before being fed into the MPI codes. This was likely done to improve the efficiency of reading and writing the data, as binary files tend to be smaller and can be read more quickly than CSV files [17].

C. Silhouette score: Evaluation Metric

A clustering algorithm's efficacy is measured using a statistic called the silhouette coefficient, also called the silhouette score. The Silhouette Score, which has a value between -1 and 1, indicates how well-separated the clusters are. The Silhouette Score is calculated as follows:

- Calculate the average distance a_i between each point i and every other point in the cluster C_i.
- Calculate the average distance (b_i) between each point (i) in cluster C_i and every other point (i) in the closest neighboring cluster.
- The Silhouette Score is calculated as (b_i - a_i) / max(a_i, b_i) for each point i. The average of the silhouette scores for all points is used as the clustering algorithm's overall silhouette score.

In this work, the score was calculated for the clustering performed with different distance metrics. The scores obtained for the AML/ALL Dataset are plotted in the table II that is tabulated in the results section.

IV. PARALLEL IMPLEMENTATION OF K-MEDOIDS ALGORITHM AND SILHOUETTE SCORE:MPI

We have used MPI for three main functions in our work Te first is the finding the closest medoids, next is computing the medoids and updating them and finally the computation of the Silhouette score.

In the findclosestmedoids function, MPI is used for two reasons:

- Broadcasting the medoids to all processes Reducing the local minimum distances to find the global minimum distance MPI_Bcast is called to broadcast the medoids array to all processes. The broadcast is performed by process 0 and all processes receive the same data.
- To reduce the local minimum distances to find the global minimum distance.

The parameters to MPI_Bcast are medoids: pointer to the medoids array, K * Samples: number of elements in the array, MPI_DOUBLE: data type of the array elements, 0: root process that broadcasts the data and lastly MPI_COMM_WORLD: communicator used for the broadcast operation. MPI_Allreduce is called to find the global minimum distance among all the local minimum distances calculated by each process. The parameters to MPI_Allreduce are: local_min_dist: pointer to the local minimum distance calculated by the process, min_dist: pointer to the global minimum distance, which will be updated by the reduction operation, 1: count of the elements being reduced, in this case only one element ,MPI_DOUBLE: data type of the element being reduced ,MPI_MIN: reduction operation, which in this case is taking the minimum value and lastly MPI_COMM_WORLD: communicator used for the reduction operation.

In the computeMedoids function, MPI is used for splitting the K clusters across the processes. The parameters used for splitting the clusters are: medoids_per_proc: the number of medoids assigned to each process, remainder: the number of remaining medoids after dividing the medoids equally among processes start_idx: the starting index of the medoids for the current process, end_idx: the ending index of the medoids for the current process These parameters are used to assign

Input: Data points $X = x_1, x_2, ..., x_n$, Number of clusters K, Distance metric $d(\cdot, \cdot)$
Output: Cluster assignments $C = C_1, C_2, ..., C_K$, Silhouette score s
Initialize medoids $M = m_1, m_2, ..., m_K$ randomly from X ; **for** $i = 1$ *to* 10 **do**
    Assign each data point x_i to the nearest medoid m_j based on $d(\cdot, \cdot)$
    Update each medoid m_j to be the data point with the lowest total distance to all other points in the cluster C_j ;
end
Compute cluster assignments C based on the final medoids M and distances $d(\cdot, \cdot)$;
Compute the silhouette score s for the cluster assignments C and distance metric $d(\cdot, \cdot)$; **for** $K = 2$ *to* K_{max} **do**
    Run the K-medoids algorithm with K clusters on X using $d(\cdot, \cdot)$;
    Compute the silhouette score s for the cluster assignments C and distance metric $d(\cdot, \cdot)$;
end

Algorithm 1: K-medoids clustering algorithm with Silhouette score. Euclidean distance metric and Manhattan distance metric are utilized.

each process a subset of medoids to compute. The medoids are then computed by each process independently, and the resulting medoids are used in the findclosestmedoids function to find the closest medoids to each data point. These two functions are run for 10 iterations as it was thought to be enough for the dataset we used. Finally to obtain and print the medoids, MPI_Gather() is used in the main function after calling computeMedoids() to gather the computed medoids from all processes into the root process.

In the computation of the Silhouette score, the pairwise distances between the data points are computed in parallel in the code above using MPI to distribute the data over multiple processors. The dataset is divided into equal-sized chunks, and each chunk is given to a different algorithm to accomplish this. The pairwise distances are then calculated for each process's own data set. Each process uses MPI_Send to transmit its results to the master process (rank 0) after computing the pairwise distances. The master process then uses MPI_Recv to compile all of the output from the other processes and then combines them to produce the whole distance matrix. The Silhouette Coefficient is calculated by first determining which cluster each data point belongs to based on the k-Medoids result file once the entire distance matrix has been acquired.

A. Parallel Implementation of K-Medoids algorithm:CUDA

In the proposed work, we have employed CUDA for two functions. The first function is the find_medoids and the second function is the compute_medoids. In both functions, each thread corresponds to one data point at a time and we distribute the workload evenly among all threads. In the find_medoids function, the thread compares the point corresponding to the thread with each of the k medoids and assigns the index of the medoid with the minimum distance to the global array of labels. In the compute medoids function, we are computing the new medoids for each cluster. Here, we are dividing the clusters across the GPUs and MPI ranks. For each cluster index, for each point in that cluster index, we are first finding the average distance of the point with every other point in the cluster index. After this stage we get an array which has the average distance of each point to every other point in the cluster. Next, we feed this array into another cuda kernel function that uses cuda reduction to find the index of the point that has the minimum average distance amongst all the points in the cluster. This will be the new medoid for this cluster. I

B. Experimental Setup

On the AiMOS supercomputer, we ran scaling tests utilizing the Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA) programming models. We were able to examine the scalability of our algorithms and models in a high-performance computing environment which is an eight petaflop IBM POWER9 system loaded with artificial intelligence applications [18]. The timing experiment was performed using the clockcycle file provided.

V. Experimental Results

A. Silhouette scores:AML/ALL dataset

For the AML/ALL dataset, the Euclidean distance metric resulted in the best Silhouette scores. The number of clusters that resulted in the best score was k=3. The score obtained was **0.154721**. The plot of clusters versus the score is shown in the figure 1

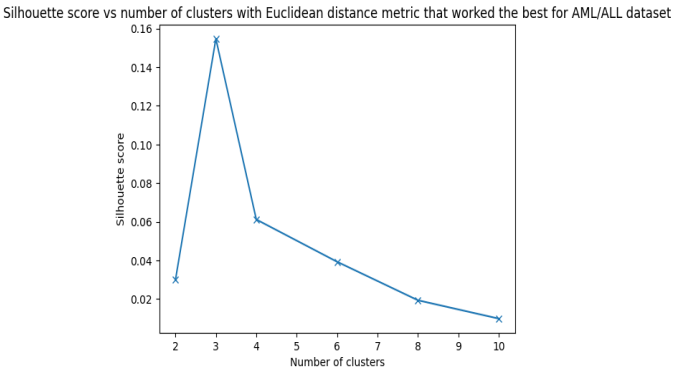

Fig. 1. Silhouette score obtained for AML/ALL dataset by employing the best distance metric

B. Silhouette scores:Rice dataset

For the Rice dataset, the Manhattan distance metric resulted in the best Silhouette scores. The number of clusters that resulted in the best score was k=2. The score obtained was

0.675864. The plot of clusters versus the score is shown in the figure 2

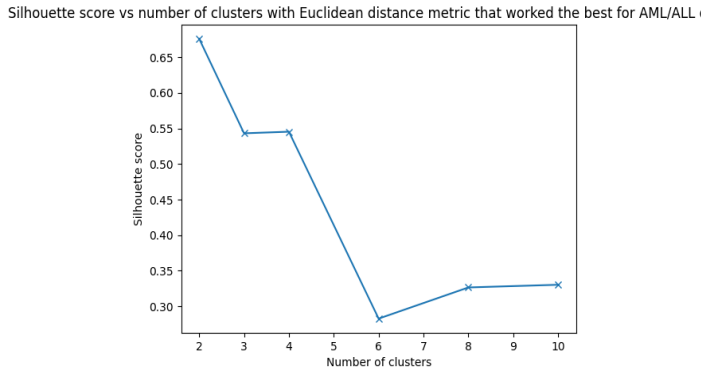

Fig. 2. Silhouette score obtained for Rice dataset by employing the best distance metric

C. Performance of Serial Codes: AML/ALL dataset

The performance of the serial version of the code is tabulated in Table II. The experiments were conducted across different cluster size and distance metrics.

Due to the large size of the data, pairwise distance computation can be a computationally expensive operation. To complete the runs in time, we used the approach of sampling. Sampling is the process of choosing a portion of the data and using that portion rather than the complete dataset. As a result, pairwise distance computation will less processing power. The performance plots are depicted in the figure 3,4, 5 and 6.

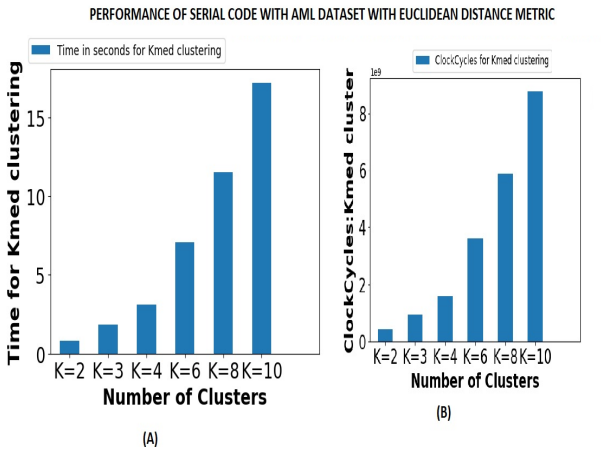

Fig. 3. Serial code: Clustering with Euclidean distance metric

D. Performance of Serial Codes: Rice dataset

The performance of the serial version of the code with the Rice dataset is tabulated in the table III.

In this table, the results obtained by using both the distance metric across different cluster size is shown. The performance plots are depicted in the figure 7,8, 9 and 10.

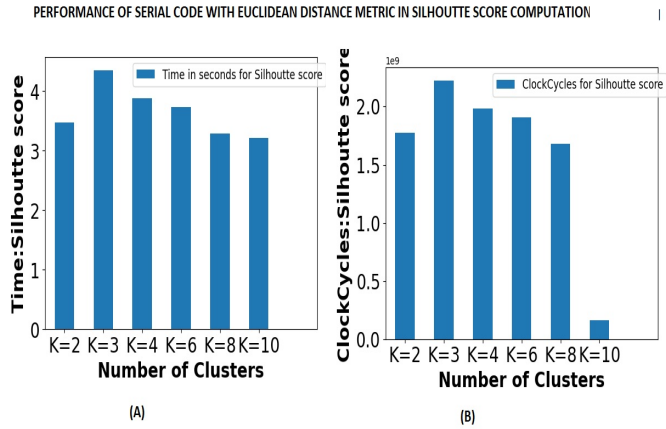

Fig. 4. Performance of Serial code: Silhouette coefficient with Euclidean distance metric

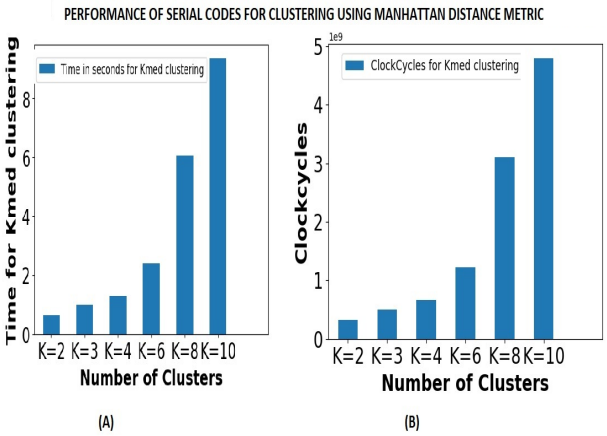

Fig. 5. Performance of Serial code: Clustering with Manhattan distance metric

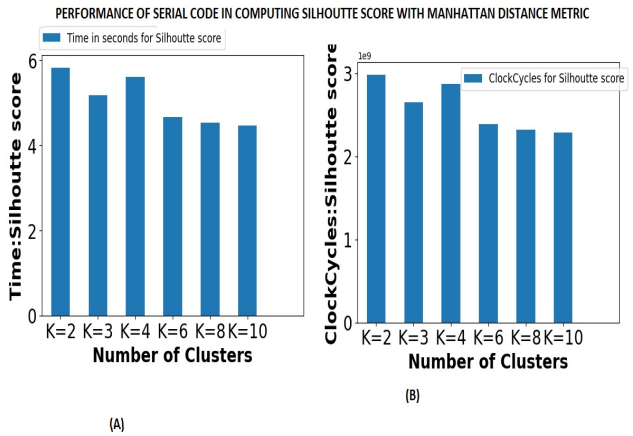

Fig. 6. Performance of Serial code: Silhouette coefficient with Manhattan distance metric

E. Performance of K-Medoids Clustering using MPI

1) AML Dataset: The Silhouette score computed on the dataset demonstrated that the best distance metric that was

TABLE II
PERFORMANCE OF SERIAL CODE ACROSS VARIOUS DISTANCE METRICS AND CLUSTERS:AML/ALL DATASET

K	Distance Metric	SilScore	Time :clusters	Cycles:clusters	Time: SilScore	Cycles
2	Manhattan	0.032087	6.27E-01	320844092	5.83	2984193407
3	Manhattan	0.030603	9.90E-01	506705473	5.18	2652492149
4	Manhattan	0.039463	1.30	665430659	5.61	2871969128
6	Manhattan	0.013255	2.38	1219364339	4.66	2386169904
8	Manhattan	0.003616	6.06	3100494716	4.54	2324564603
10	Manhattan	0.010874	9.37	4799504192	4.47	2286103285
2	Euclidean	0.029956	8.28E-01	423729691	3.47	1776021093
3	**Euclidean**	**0.154721**	**1.84**	**942565677**	**4.35**	**2225862866**
4	Euclidean	0.061187	3.11	1593269805	3.88	1985094496
6	Euclidean	0.039295	7.07	3619882406	3.73	1910370677
8	Euclidean	0.019372	1.15E+01	5887046886	3.28	1680149766
10	Euclidean	0.009878	1.72E+01	8781836652	3.21	164541993

TABLE III
PERFORMANCE OF SERIAL CODE ACROSS VARIOUS DISTANCE METRICS AND CLUSTERS:RICE DATASET

K	Distance Metric	SilScore	Time :clusters	Cycles:clusters	Time:SilScores	Cycles
2	**Manhattan**	**0.675864**	**7.80**	**3994559856**	**1.13E+01**	**5808582475**
3	Manhattan	0.543068	7.81	3998738453	1.02E+01	5236499172
4	Manhattan	0.545298	7.81	3999879547	1.02E+01	5240704751
6	Manhattan	0.282759	7.82	4005176431	9.10E+00	4658362194
8	Manhattan	0.326556	7.82	4003746381	9.04E+00	4629724302
10	Manhattan	0.330335	7.84	4013373734	9.41E+00	4819855665
2	Euclidean	0.466717	1.16	591404151	6.95E+00	3560189609
3	Euclidean	0.361662	1.72	878883557	6.21E+00	3178010931
4	Euclidean	0.332326	2.38	1220088368	6.16E+00	3154655753
6	Euclidean	0.246667	3.36	1722732028	6.08E+00	3112153174
8	Euclidean	0.23206	4.47	2287562894	5.71E+00	2921833206
10	Euclidean	0.218921	5.91	3028204695	5.80E+00	2967372504

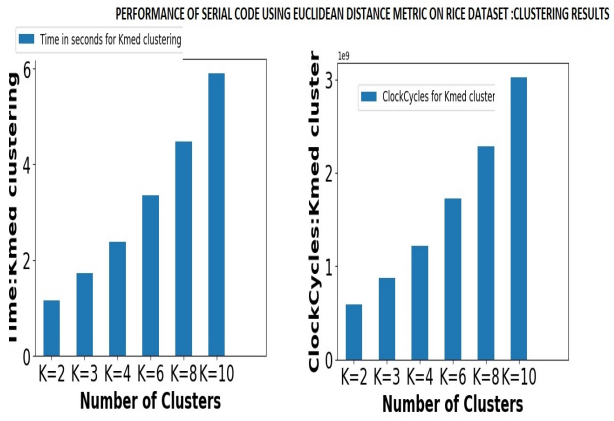

Fig. 7. Serial code: Clustering with Euclidean distance metric

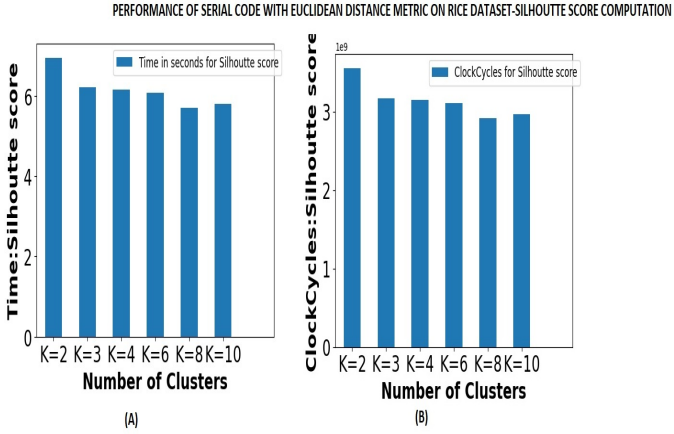

Fig. 8. Performance of Serial code: Silhouette coefficient with Euclidean distance metric

suitable for the AML/ALL dataset was the Euclidean distance metric. Hence, the MPI experiments are conducted only using this distance metric. To show the power of MPI, the sampling approach followed in the serial code was eliminated.

The performance of K-Medoids clustering across various MPI ranks is tabulated in the Table IV. **Strong scaling is done by keeping the clusters (K) same and increasing the MPI Ranks)**. To do the "weak scaling" study we increase the value of K with increasing MPI Ranks. The plots are depicted in the figure 11. Also, we demonstrated weak scaling by choosing another larger Rice dataset that demonstrated the growth of the problem with the MPI ranks.

2) Rice Gene Dataset: The Silhouette score computed on the dataset demonstrated that the best distance metric that was suitable for the Rice dataset was the Manhattan distance metric. Hence, the MPI experiments are conducted only using this distance metric. To show the power of MPI, the sampling approach followed in the serial code was eliminated.

The performance of K-Medoids clustering across various

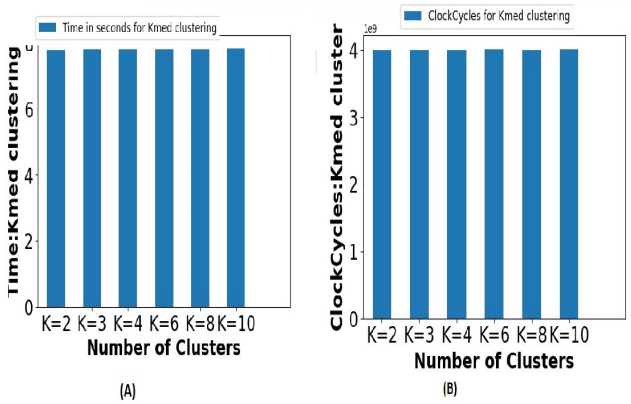

Fig. 9. Performance of Serial code: Clustering with Manhattan distance metric

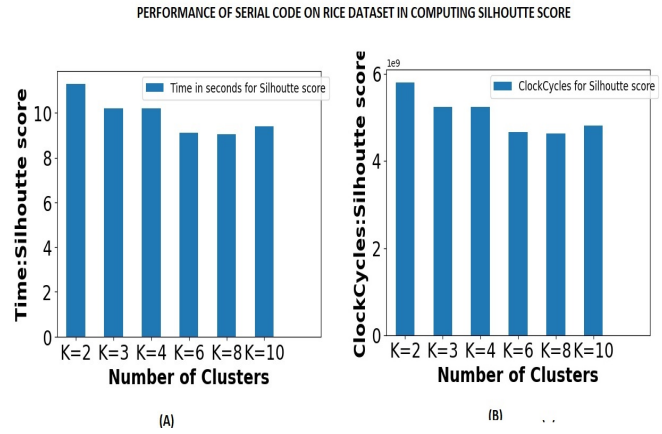

Fig. 10. Performance of Serial code: Silhouette coefficient with Manhattan distance metric

MPI ranks is tabulated in the Table V. **Strong scaling is done by keeping the clusters (K) same and increasing the MPI Ranks)**. To do the "weak scaling" study we increase the value of K with increasing MPI Ranks. The plots are depicted in the figure 12.

3) Computation of Speedup: To show a fair comparison between the serial and MPI codes, an experiment was conducted where sampling was eliminated from the serial code. The results of this experiment are shown in table VI and VII. The results are separately plotted with Euclidean distance metric and Manhattan metric. We also reduced the number of genes to 7000 (from 7129) and samples to 10 (from 34). The main reason behind the reduction was to complete the serial code run within given time. As the dimensions of samples increased, the time also increases. Hence, it was crucial to perform this step. The plot of time and clock cycles using different distance metric is shown in the figure 13 and 14. The speedup is computed using the equation 1 and 2. From the table, it can be seen that 8 MPI ranks resulted in the best results with Manhattan distance metrics and the Euclidean distance metric.

TABLE IV
AML DATASET SCALING RESULTS: VARIOUS MPI RANKS FOR EACH CLUSTER SIZE

Clusters	MPI RANKS	Time in seconds	Clockcycles
K=2	2	1.324391e+02	67808832037
	4	7.497441e+01	38386898590
	8	1.318035e+02	67483398290
	16	1.158008e+02	59290034854
	32	1.379250e+02	70617579705
	64	1.340464e+02	68631779165
	128	1.337826e+02	68496695555
	256	1.299019e+02	66509748498
K=3	2	1.692315e+01	8664652365
	4	1.688877e+01	8647048695
	8	1.682384e+01	8613804292
	16	1.677571e+01	8589165800
	32	1.764187e+01	9032636310
	64	1.758246e+01	9002221176
	128	1.719860e+01	8805681849
	256	1.661813e+01	8508483043
K=4	2	1.325085e+02	67844350213
	4	1.320744e+02	67622083731
	8	1.317417e+02	67451724814
	16	1.314518e+02	67303324034
	32	1.355009e+02	69376463890
	64	1.367251e+02	70003244026
	128	1.343060e+02	68764653618
	256	1.299189e+02	66518487265
K=6	2	1.324486e+02	67813703990
	4	1.320351e+02	67601992667
	8	1.317630e+02	67462643414
	16	1.314642e+02	67309660647
	32	1.298120e+02	66463744474
	64	1.354925e+02	69372182448
	128	1.338126e+02	68512039521
	256	1.33E+02	68110642113
K=8	2	1.323653e+02	67771058227
	4	1.320443e+02	67606696272
	8	1.322112e+02	6769214937
	16	1.315528e+02	67355024977
	32	1.354968e+02	69374381831
	64	1.347422e+02	68987982619
	128	1.326959e+02	67940299624
	256	1.329501e+02	68070459595
K=10	2	1.324145e+02	67796248386
	4	1.351092e+02	69175914763
	8	1.350388e+02	69139879836
	16	1.344430e+02	68834828286
	32	1.411929e+02	72290780274
	64	1.374962e+02	70398057163
	128	1.366881e+02	69984316393
	256	1.298093e+02	66462344036

$$\text{Speedup} = \text{Time (serial code)}/\text{Time with the best MPI Rank} \quad (1)$$

$$\text{Speedup} = \text{Clockcycle (serial code)}/\text{Clockcycle with the best Rank} \quad (2)$$

Using the above equation for the time, we see that the MPI code with 8 MPI ranks is **25.6574923547** faster than the serial code with Manhattan distance metric. Using the above equation for the clockcycles, we see that the MPI code with 8

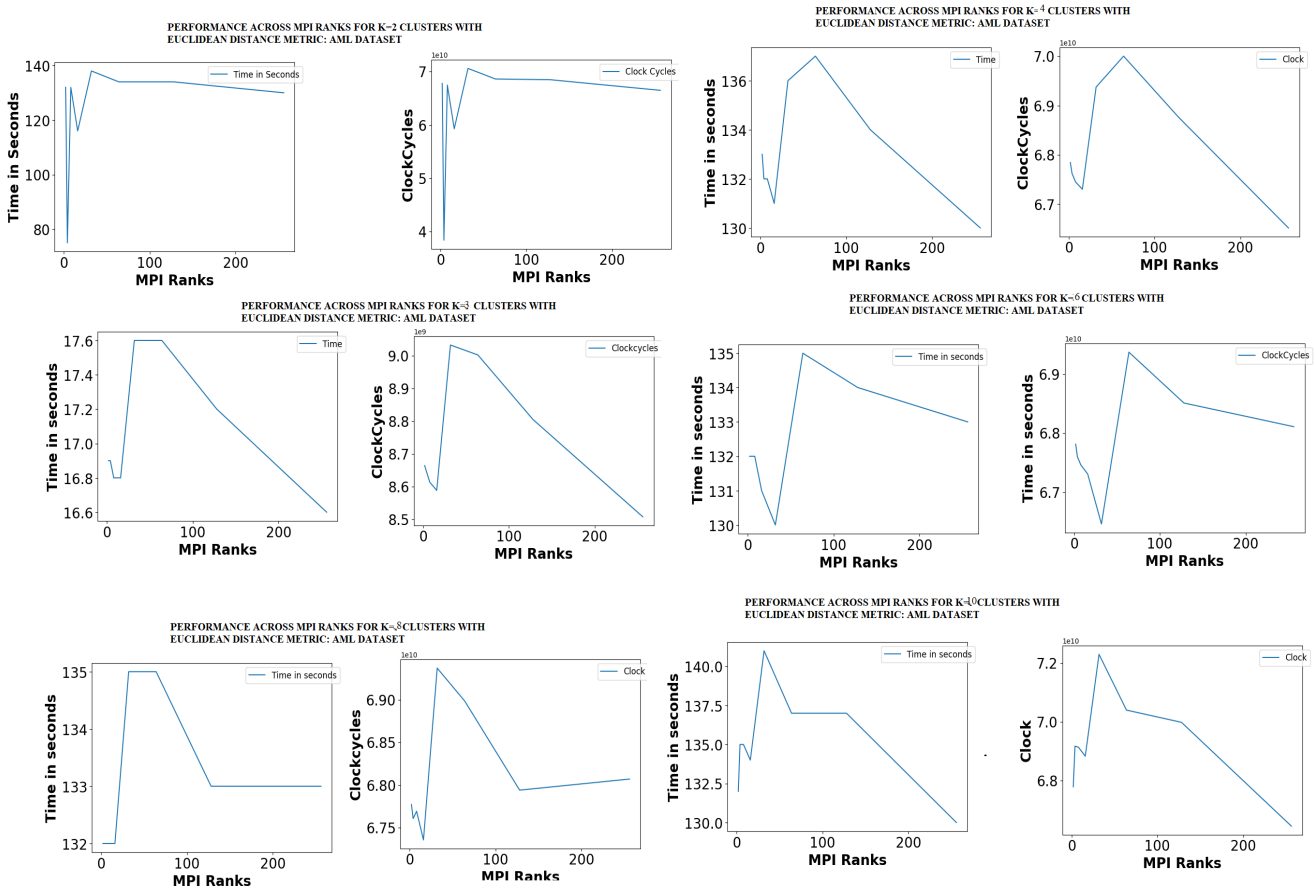

Fig. 11. Time and Clock Cycles obtained across different MPI Ranks and Cluster size

MPI ranks is **25.61** faster than the serial code with Manhattan distance metric. Using the above equation for the time, we see that the MPI code with 8 MPI ranks is **45.8333333333** times faster than the serial code with Euclidean distance metric. Using the above equation for the clockcycles, we see that the MPI code with 8 MPI ranks is **45.8304664498** faster than the serial code with Euclidean distance metric. The 8 MPI ranks performed the best might be because the dataset size was insufficient to reap the benefits of using 16 MPI ranks or higher. Another reason for the slower performance could be that when MPI ranks increased, the communication overhead between the processes did as well.

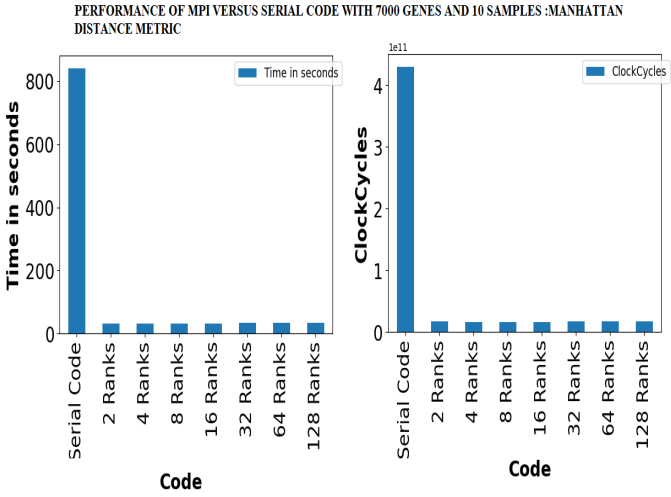

Fig. 13. Performance comparison between serial and MPI codes with Manhattan distance metric

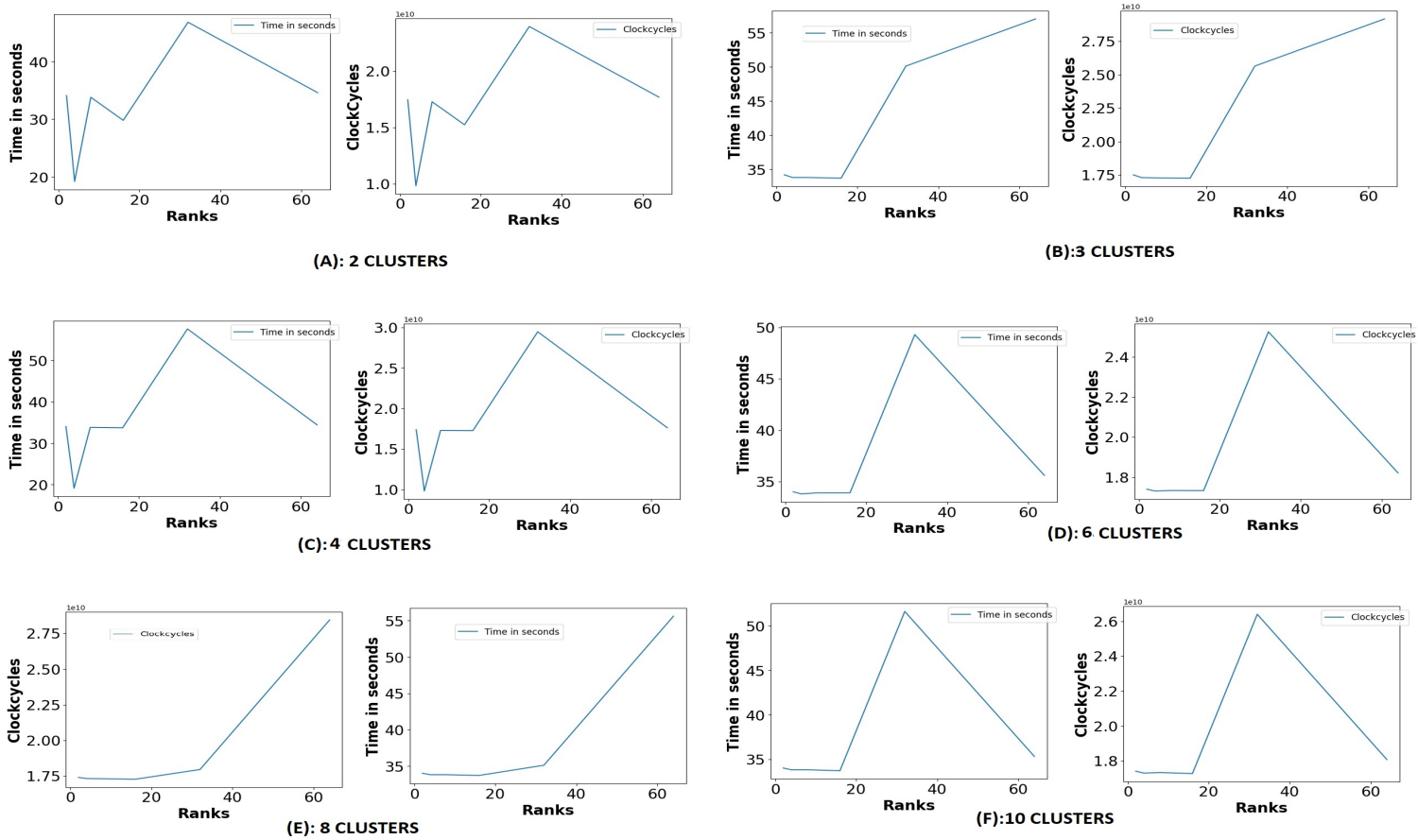

Fig. 12. Rice Dataset: Time and Clock Cycles obtained across different MPI Ranks and Cluster size

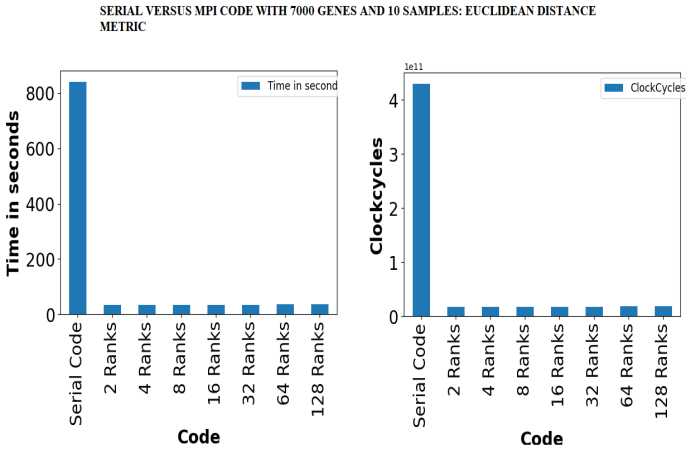

Fig. 14. Performance comparison between serial and MPI codes with Euclidean distance metric

F. Performance of MPI I/O with different MPI Ranks

In the proposed work, to demonstrate the MPI I/O (Input/Output), we employed multiple MPI processes to read and write data to and from files.

Among the MPI I/O functions we used MPI_File_open() to open a file for MPI I/O operations ,MPI_File_close() to close the file, MPI_File_read() to read the contents of the bin file and MPI_File_write() to write data to a file using MPI I/O. The time and clockcycles taken to read the binary file of AML dataset using MPI functions is shown in the table VIII. **The serial code took 2.488882e-01 seconds and 127430766 cycles. From the table, it is evident that 8 MPI ranks resulted in the best results. Hence we compute the speedup with respect to 8 MPI Ranks.** The performance plots are shown in the figure 15. The eq.1 and 2 are used here. **The MPI File read operation was 44.9256679 times faster in terms of time in seconds and 44.9250687 times faster in terms of clock cycles.**. For this experiment, we utilized only the AML/ALL dataset. However, this experiment can also be extended to read the slightly larger Rice dataset too. Also, the scaling was done only until 8 MPI ranks as for the larger MPI ranks the file size was not evenly divisible. However, we could not achieve significant speedup in MPI file writing when compared to serial writing. We believe the main reason is that MPI file write involves communication between processes, which can lead to additional overhead and synchronization costs.

TABLE V
RICE DATASET SCALING RESULTS: VARIOUS MPI RANKS FOR EACH CLUSTER SIZE

K	MPI ranks	Time in seconds	Clockcycles
2	2	3.41E+01	17458020400
2	4	1.92E+01	9832803349
2	8	3.38E+01	17283453767
2	16	2.98E+01	15243144156
2	32	4.68E+01	23953652277
2	64	3.46E+01	17692492229
3	2	3.42E+01	17517529348
3	4	3.38E+01	17311680548
3	8	3.38E+01	17286268834
3	16	3.37E+01	17266653652
3	32	5.01E+01	25653192117
3	64	5.70E+01	29174217925
4	2	3.40E+01	17390569502
4	4	1.91E+01	9801451280
4	8	3.38E+01	17288299390
4	16	3.37E+01	17271225154
4	32	5.76E+01	29484668222
4	64	3.44E+01	17617959060
6	2	3.40E+01	17404212529
6	4	3.38E+01	17311802056
6	8	3.39E+01	17338433734
6	16	3.39E+01	17334701306
6	32	4.93E+01	25253044490
6	64	3.56E+01	18215198144
8	2	3.40E+01	17394103406
8	4	3.38E+01	17315482450
8	8	3.38E+01	17303072623
8	16	3.37E+01	17267324710
8	32	3.51E+01	17949288405
8	64	5.56E+01	28447604654
10	2	3.40E+01	17408142929
10	4	3.38E+01	17301149377
10	8	3.38E+01	17330129359
10	16	3.37E+01	17274866050
10	32	5.16E+01	26399287304
10	64	3.53E+01	18076301818

TABLE VI
PERFORMANCE OF MPI VERSUS SERIAL WITH MANHATTAN DISTANCE METRIC

Code	Time in seconds	ClockCycles
Serial Code	8.39E+02	4.29E+11
2 Ranks	3.30E+01	16888799507
4 Ranks	3.27E+01	16762452765
8 Ranks	**3.27E+01**	**16750045575**
16 Ranks	3.28E+01	16788092844
32 Ranks	3.44E+01	17621845317
64 Ranks	3.38E+01	17313782588
128 Ranks	3.35E+01	17137160399

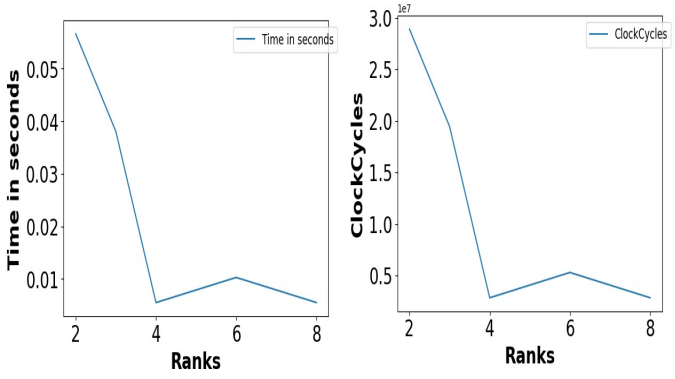

Fig. 15. File Read using MPI :Time and Clockcycles across MPI Ranks

TABLE VII
PERFORMANCE OF MPI VERSUS SERIAL WITH EUCLIDEAN DISTANCE METRIC

Code	Time in seconds	ClockCycles
Serial Code	1.54E+03	7.87361E+11
2 Ranks	3.38E+01	17313744669
4 Ranks	3.36E+01	17224154721
8 Ranks	**3.36E+01**	**17179860058**
16 Ranks	3.38E+01	17288302844
32 Ranks	3.49E+01	17865506448
64 Ranks	3.60E+01	18456796249
128 Ranks	3.59E+01	18403830615

TABLE VIII
MPI FILE READ PERFORMANCE ACROSS DIFFERENT RANKS

MPI Ranks	Time in seconds	Clock cycles
2	5.65E-02	28919187
3	3.80E-02	19473013
4	**5.53E-03**	**2833675**
6	1.03E-02	5291682
8	5.54E-03	2836518

In this project, the output was written into three files: Firstly the clustered gene data, secondly the computed medoids and lastly the cluster assignments. As the number of MPI ranks increases, the loop will be executed in parallel by each rank as each rank executes the same code. The data will be written in sections by each rank, resulting in interleaved writes to the file. The overall size of the data and the number of rankings will determine how much data each rank writes.

TABLE IX
PERFORMANCE OF MPI FILE WRITE ACROSS MPI RANKS

Code	Compute Nodes	Time in seconds	Clock cycles
2	1	7.33E-02	37540362
4	1	7.75E-02	39662027
8	**1**	**4.70E-02**	**24088954**
16	1	9.22E-02	47201932
32	1	1.13E-01	57627425
64	2	5.93E-02	30355279
128	4	8.34E-02	42700338
256	8	6.91E-02	35359676

Each rank will write Genes/4 elements to the file, for instance, if the data has 4 ranks (or the file is run with 4 MPI ranks). The experiment was conducted across 2,4,8,16,32,64, 128, and 256 MPI ranks. The clock cycles and time were computed to evaluate its performance. The result of scaling is shown in Table 16. Figure 16 shows the plot of the time and clock cycles. The best MPI rank that resulted in the fastest write was 8 MPI Ranks.

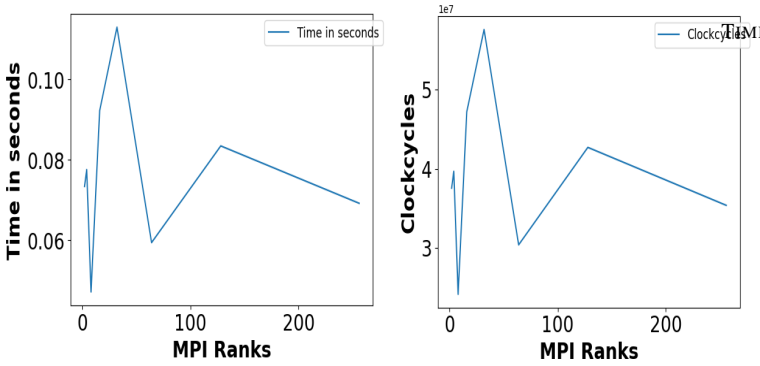

Fig. 16. File Write using MPI :Time and Clockcycles across MPI Ranks

G. Performance of Silhouette score using MPI:AML/ALL dataset

The table X shows the time and clock cycles needed to compute Silhouette score with different MPI ranks and cluster size. From the results, we see that across all cluster sizes, the best results are obtained with 128 MPI ranks. The plot of time and clock cycles across various ranks and cluster sizes is shown in the figure 17.

Computation of the speedup: The speedup is computed using eq. 1 and 2. **Considering the time, the MPI code was 7.4742268 times faster than serial code in computing the Silhouette score with k=3 and 128 MPI ranks. The k =3 was chosen because it led to the highest score. Considering the clockcycles, the MPI code was 7.46985909 times faster than the serial code**

H. Performance of Silhouette score using MPI: Rice dataset

The table XI shows the time and clock cycles needed to compute Silhouette score with different MPI ranks and cluster size by using the Rice dataset. From the results, we see that across all cluster sizes, the best results are obtained with 128 MPI ranks except for K=2. The plot of time and clock cycles across various ranks and cluster sizes is shown in the figure 18.

Computation of the speedup: The speedup is computed using eq. 1 and 2. **Considering the time, the MPI code was 20.9647495 times faster than serial code in computing the Silhouette score with k=2 and 64 MPI ranks. The k =2 was chosen because it led to the highest score. Considering the clock cycles, the MPI code was 21.0581375 times faster than the serial code**

I. Performance:CUDA

Due to the unavailability of the compute nodes on the AiMOS, we ran the CUDA implementations with one GPU. The scaling was done by changing the cluster size (K) and also by increasing the block size.

The experiment was conducted only on the AML/ALL dataset. The performance of the code is tabulated in the table XII. The clock cycles and time were recorded for each of the experimental runs. The plots are represented in the figure 19.

TABLE X
TIME TAKEN TO COMPUTE SILHOUETTE SCORE WITH DIFFERENT MPI RANKS AND CLUSTER SIZE

Clusters	MPI RANKS	Time	Clockcycles
K=2	2	1.26E+01	6453887258
	4	6.354568	3253538667
	8	3.327717	1703790922
	32	1.16E+00	593869655
	64	6.64E-01	340084418
	128	6.06E-01	310233743
	256	9.01E-01	461198101
K=3	2	1.26E+01	6450782676
	4	6.343563	3247904417
	8	3.266749	1672575615
	32	1.046158	535632788
	64	7.29E-01	373489200
	128	**5.82E-01**	**297979231**
	256	9.89E-01	506359456
K=4	2	1.12E+01	5725273091
	4	5.640849	2888114546
	8	2.912415	1491156499
	32	1.066681	546140541
	64	5.77E-01	295201028
	128	6.16E-01	315520356
	256	6.04E-01	309103855
K=6	2	1.07E+01	5500059995
	4	5.422237	2776185386
	8	2.784624	1425727614
	32	9.47E-01	484708087
	64	6.17E-01	315968505
	128	5.62E-01	287689416
	256	1.11E+00	569385580
K=8	2	9.518088	4873261158
	4	4.838664	2477396113
	8	2.55195	1306598391
	32	1.342403	687310199
	64	6.08E-01	311456183
	128	5.84E-01	298824101
	256	9.24E-01	473074428
K=10	2	9.291799	4757401089
	4	4.734511	2424069502
	8	2.431409	1244881663
	32	9.80E-01	501785994
	64	6.27E-01	320837853
	128	5.12E-01	262355794
	256	9.92E-01	507927642

The best results were obtained with K=2 and block size of 16. The time obtained was 1.37E-01 seconds and 70066081 clock cycles. The speedup was computed using the equation 1 and 2. **The CUDA implementation was 6.04379562 faster than serial code in clustering considering the time and 6.04757231 faster than serial code in clustering considering the clock cycles**. We believe the reason behind this is the ideal balance between shared memory use and thread utilization was supplied by blocks with a size of 16. With 16 threads per block, it was possible to efficiently access and update the required data by each thread while also making sure that every thread in each block was used and there were no idle threads.

J. Heatmaps

Heatmaps offer a visual depiction of trends in the data that may not be immediately obvious from the raw numerical values alone, making them a useful tool for visualizing

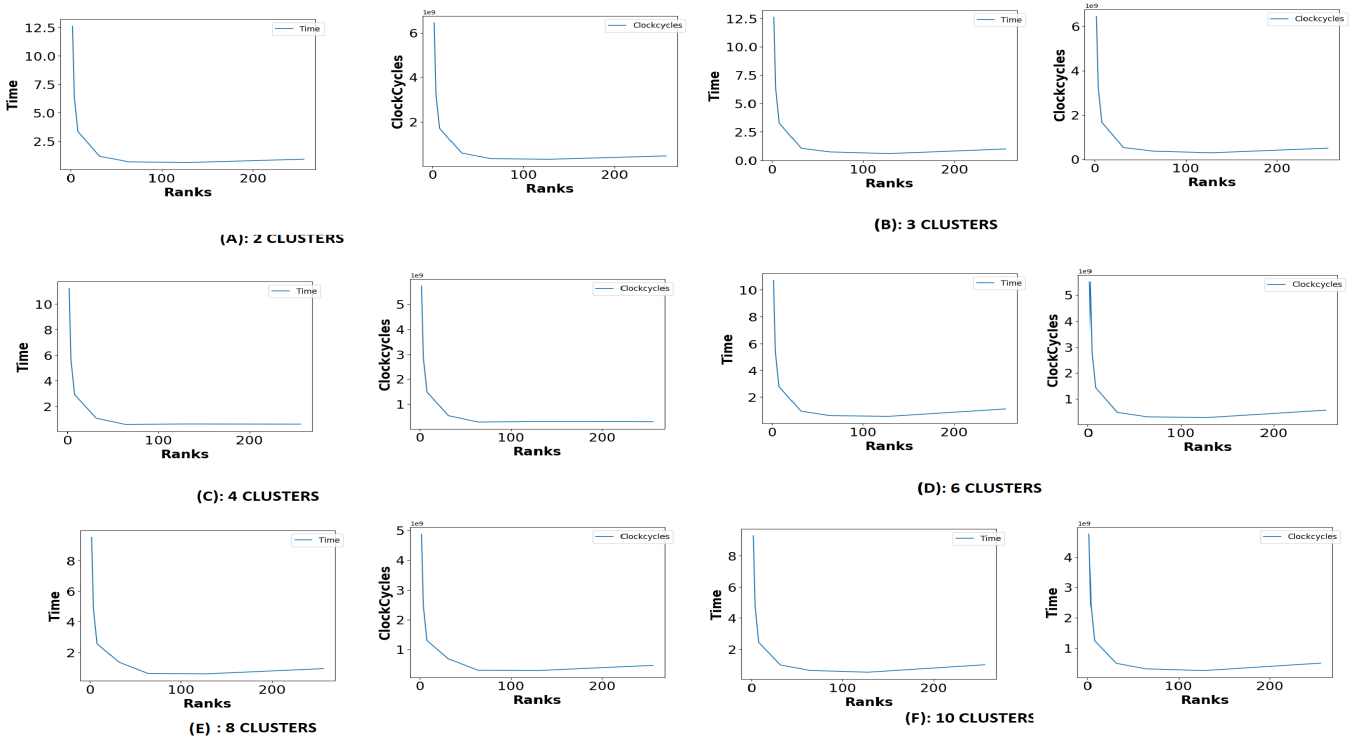

Fig. 17. Time and Clock Cycles obtained across different MPI Ranks and Cluster size for computation of Silhouette scores:AML Dataset

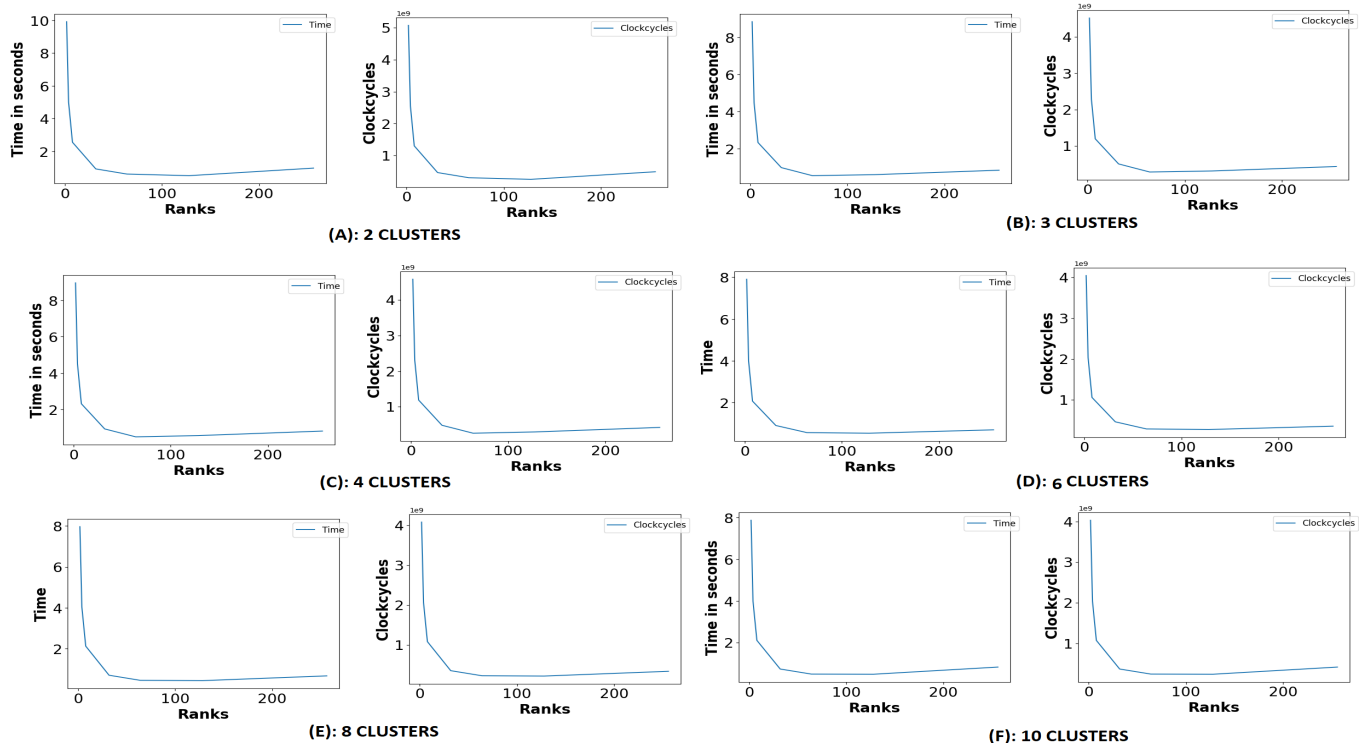

Fig. 18. Time and Clock Cycles obtained across different MPI Ranks and Cluster size for computation of Silhouette scores:Rice Dataset

correlations between gene accession numbers and cluster assignments. Heatmaps can illustrate trends and patterns in the data that can be challenging to see otherwise by utilizing color coding to reflect expression levels.

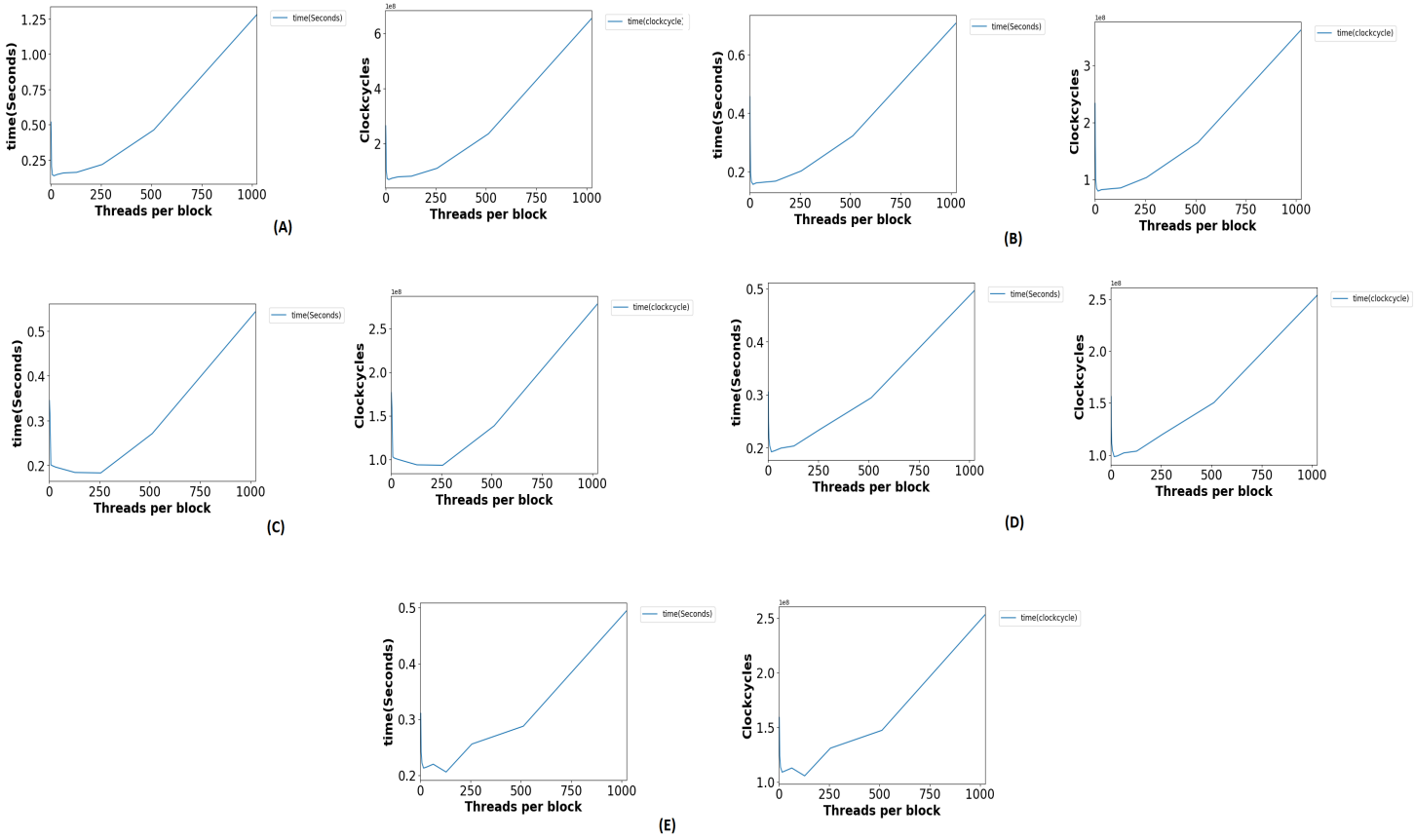

Fig. 19. (A). Performance with K=2. (B). Performance with K=4. (C). Performance with K=6. (D) Performance with K=8. (E). Performance with K=10

This can help in the discovery of gene networks and pathways and make it simple to find groups of genes that are co-regulated or have similar expression characteristics. After obtaining the cluster assignments from our experiments, we plotted the gene accession numbers and the cluster assignments in the form of heatmaps. For the AML/ALL dataset,the gene accession numbers were provided. But for the RiceHap dataset, we considered the ”Genes” column as the gene accession number. The heatmap plots are shown in the figure 20, 21, 22,23.

VI. Conclusion and Future Work

In this paper, the distance-based K-medoids algorithm has been utilized for gene clustering on two different datasets. The algorithm selects initial medoids randomly and updates them using an existing distance matrix. The number of clusters is determined based on the Silhouette score. The K-medoids algorithm is advantageous in that it is less sensitive to outliers. The experimental results demonstrated that utilizing MPI to compute pairwise distances and medoids significantly increased the algorithm’s speed. MPI file read was also faster than serial file writing, but MPI file write did not yield a substantial speedup compared to serial write. We believe it is due to the fact that MPI file write involves communication between processes, which can lead to additional overhead and synchronization costs. Future scope of this work includes implementing Silhoutte score using CUDA and experimenting with larger datasets to evaluate the effectiveness of the proposed algorithm.

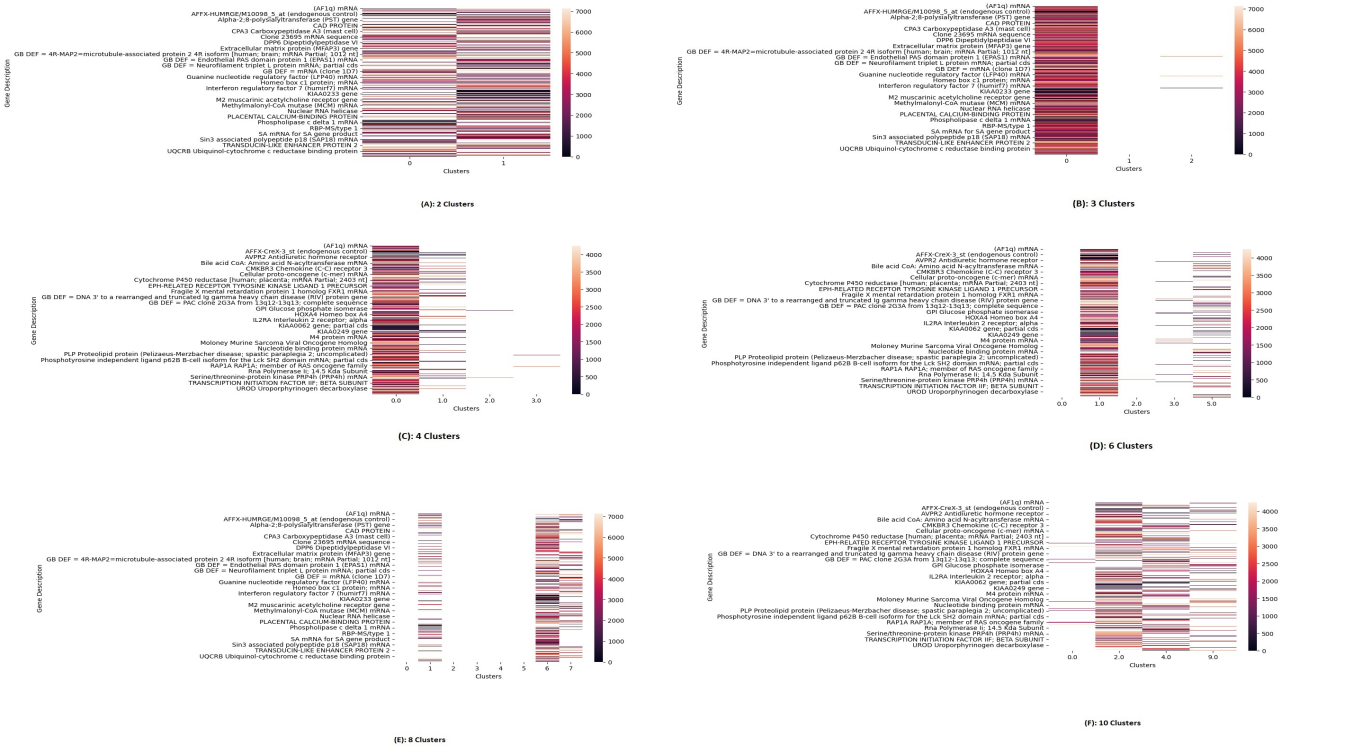

Fig. 20. AML/ALL Dataset:Heatmaps of cluster assignments versus the gene accession numbers obtained by employing Euclidean distance metric

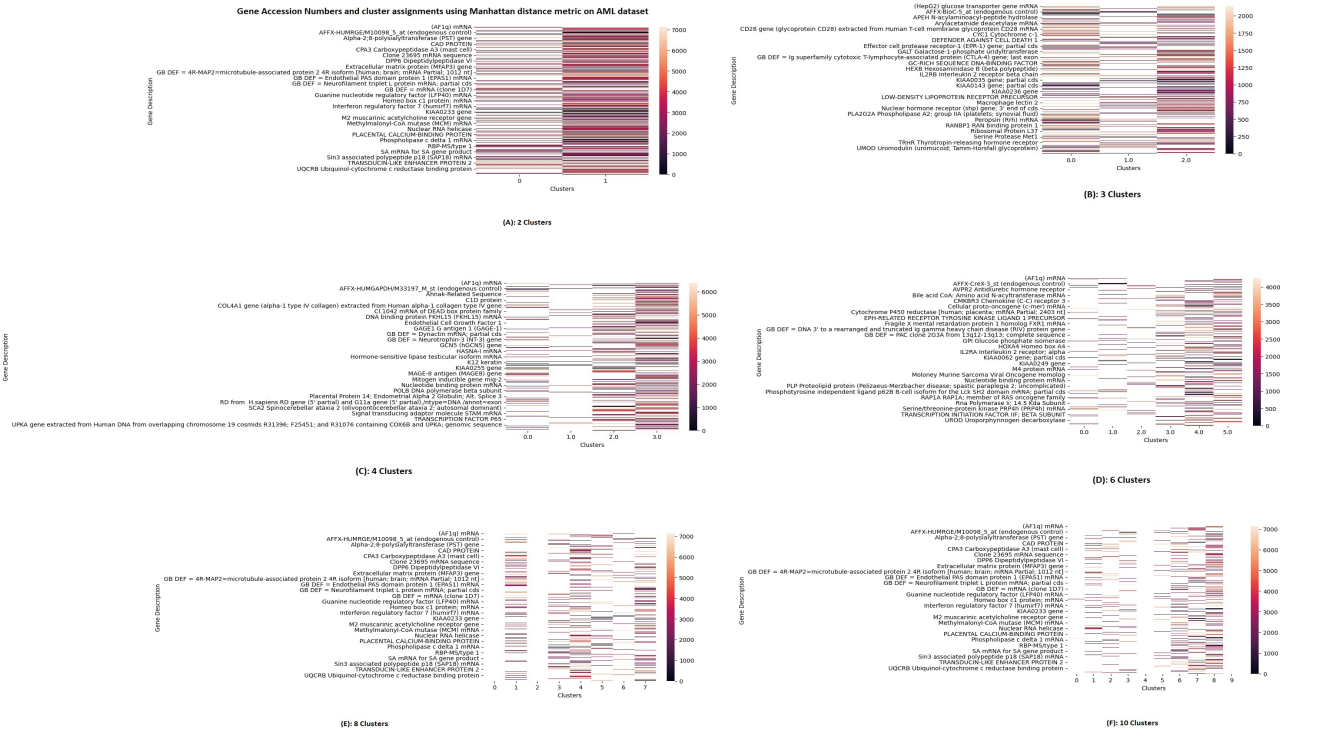

Fig. 21. AML/ALL Dataset:Heatmaps of cluster assignments versus the gene accession numbers obtained by employing Manhattan distance metric

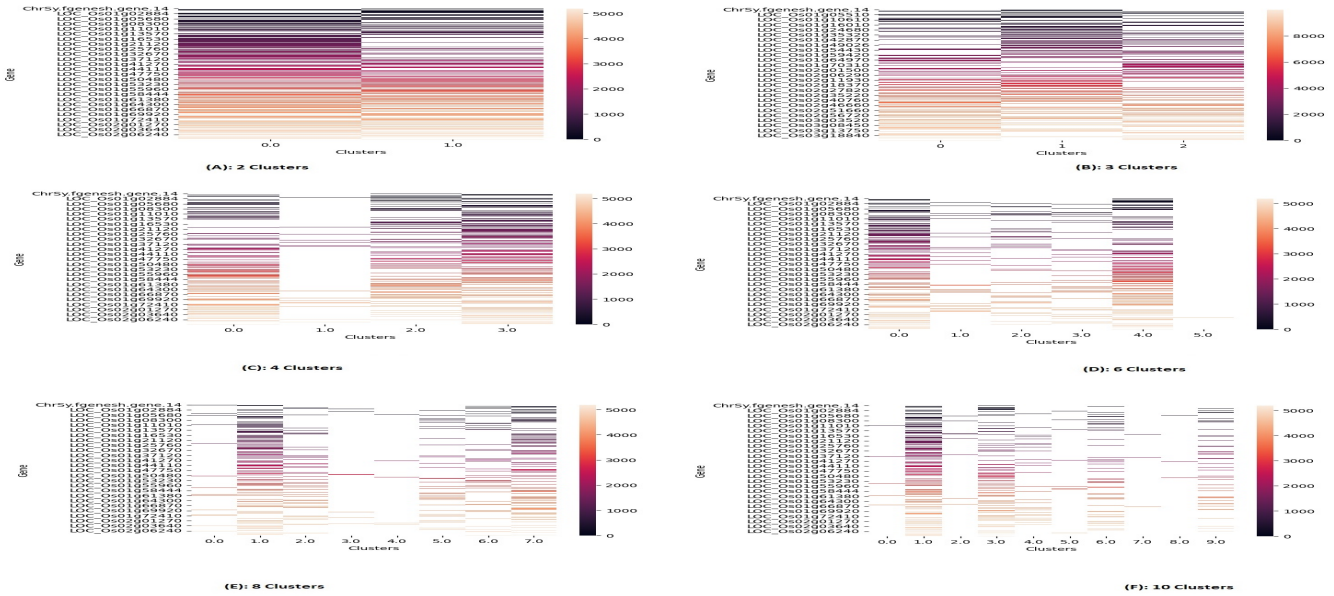

Fig. 22. Rice dataset:Heatmaps of cluster assignments versus the gene accession numbers obtained by employing Euclidean distance metric

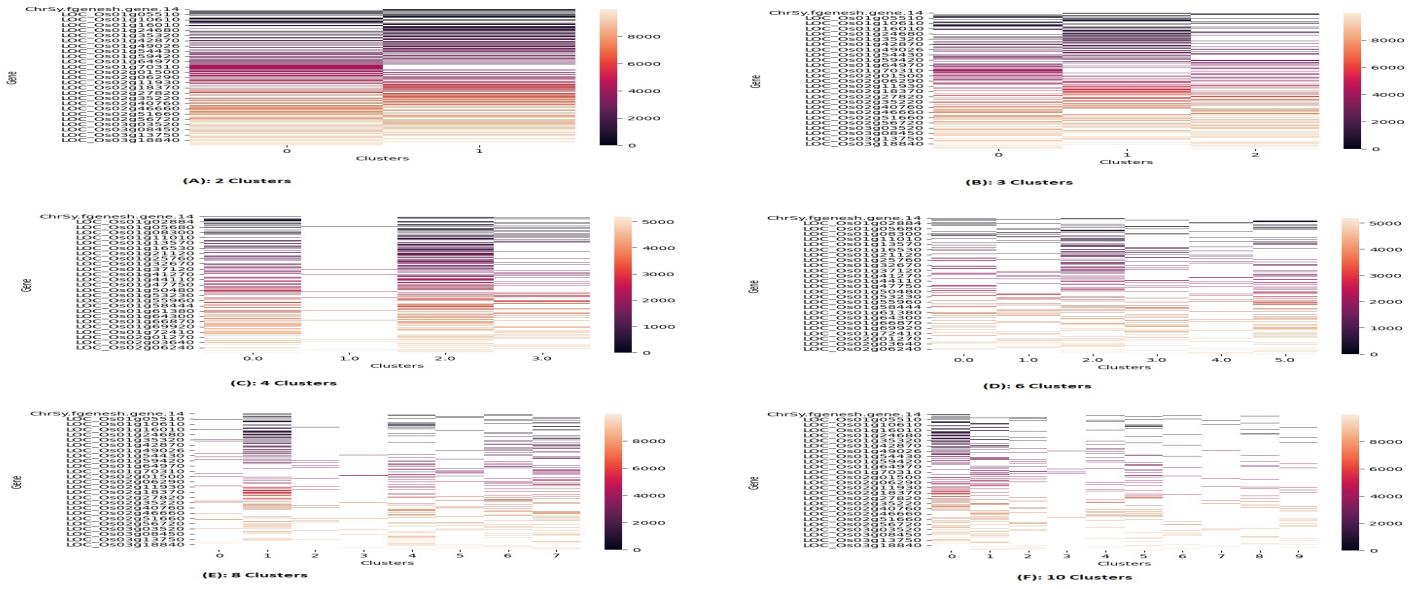

Fig. 23. Rice dataset:Heatmaps of cluster assignments versus the gene accession numbers obtained by employing Manhattan distance metric

TABLE XI
TIME TAKEN TO COMPUTE SILHOUETTE SCORE WITH DIFFERENT MPI RANKS AND CLUSTER SIZE

Clusters	MPI RANKS	Time	Clockcycles
K=2	2	9.91E+00	5071440551
	4	4.997218	2558575474
	8	2.550989	1306106200
	32	9.23E-01	472724434
	64	6.09E-01	311663081
	128	5.10E-01	261044008
	256	9.72E-01	497758953
K=3	2	8.84E+00	4524374573
	4	4.46734	2287278097
	8	2.327268	1191561236
	32	9.74E-01	498833823
	64	**5.39E-01**	**275835528**
	128	5.94E-01	304167172
	256	8.34E-01	427160627
K=4	2	8.957616	4586299625
	4	4.51E+00	2308525341
	8	2.32E+00	1188809462
	32	9.38E-01	480292683
	64	5.02E-01	257195104
	128	5.72E-01	293026856
	256	8.20E-01	419806572
K=6	2	7.900536	4045074679
	4	3.99E+00	2041452332
	8	2.074884	1062340615
	32	9.09E-01	465578879
	64	5.69E-01	291515196
	128	5.41E-01	276898104
	256	7.02E-01	359350408
K=8	2	7.963312	4077215942
	4	4.031678	2064218957
	8	2.12E+00	1083739462
	32	6.97E-01	356813654
	64	4.49E-01	230001492
	128	4.33E-01	221789063
	256	6.66E-01	340934876
K=10	2	7.871711	4030316169
	4	3.98687	2041277612
	8	2.10E+00	1074074276
	32	7.23E-01	370050622
	64	4.80E-01	245982683
	128	4.73E-01	242128531
	256	8.17E-01	418337870

TABLE XII
PERFORMANCE OF THE CUDA WITH DIFFERENT BLOCK SIZE AND DIFFERENT CLUSTER SIZES

K	Threads per block	Clockcycles	time in seconds
2	1	265395639	5.18E-01
	2	223456403	4.36E-01
	4	104145038	2.03E-01
	8	73904502	1.44E-01
	16	70066081	1.37E-01
	32	74865749	1.46E-01
	64	80149676	1.57E-01
	128	82200694	1.61E-01
	256	111007783	2.17E-01
	512	236582353	4.62E-01
	1024	654644557	1.28E+00
4	1	233513983	4.56E-01
	2	161190155	3.15E-01
	4	100891022	1.97E-01
	8	84485916	1.65E-01
	16	79992679	1.56E-01
	32	82387928	1.61E-01
	64	83520470	1.63E-01
	128	85538267	1.67E-01
	256	103673201	2.02E-01
	512	164964109	3.22E-01
	1024	361980297	7.07E-01
6	1	176796137	3.45E-01
	2	169862724	3.32E-01
	4	157998182	3.09E-01
	8	103101269	2.01E-01
	16	101711387	1.99E-01
	32	100446468	1.96E-01
	64	98247500	1.92E-01
	128	94036895	1.84E-01
	256	93540682	1.83E-01
	512	138778132	2.71E-01
	1024	278122477	5.43E-01
8	1	156361799	3.05E-01
	2	127605184	2.49E-01
	4	111136899	2.17E-01
	8	104063987	2.03E-01
	16	98425239	1.92E-01
	32	99087763	1.94E-01
	64	101995498	1.99E-01
	128	103715952	2.03E-01
	256	119856591	2.34E-01
	512	150545740	2.94E-01
	1024	253801164	4.96E-01
10	1	158434204	3.09E-01
	2	159094300	3.11E-01
	4	123892087	2.42E-01
	8	113415570	2.22E-01
	16	108925422	2.13E-01
	32	110179033	2.15E-01
	64	112721605	2.20E-01
	128	105547216	2.06E-01
	256	130866779	2.56E-01
	512	147230718	2.88E-01
	1024	252958443	4.94E-01

REFERENCES

[1] P. Arora, S. Varshney, *et al.*, "Analysis of k-means and k-medoids algorithm for big data," *Procedia Computer Science*, vol. 78, pp. 507–512, 2016.

[2] A. Martino, A. Rizzi, and F. M. F. Mascioli, "Distance matrix pre-caching and distributed computation of internal validation indices in k-medoids clustering," in *2018 international joint conference on neural networks (IJCNN)*, pp. 1–8, IEEE, 2018.

[3] S. Gultom, S. Sriadhi, M. Martiano, and J. Simarmata, "Comparison analysis of k-means and k-medoid with ecluidience distance algorithm, chanberra distance, and chebyshev distance for big data clustering," in *IOP Conference Series: Materials Science and Engineering*, vol. 420, p. 012092, IOP Publishing, 2018.

[4] D. Bikov, M. Pashinska, and N. Stojkovic, "Parallel programming with cuda and mpi," 2020.

[5] A. Mukhopadhyay and U. Maulik, "Towards improving fuzzy clustering using support vector machine: Application to gene expression data," *Pattern Recognition*, vol. 42, no. 11, pp. 2744–2763, 2009.

[6] F.-X. Wu, W.-J. Zhang, and A. J. Kusalik, "A genetic k-means clustering algorithm applied to gene expression data," in *Advances in Artificial Intelligence: 16th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2003, Halifax, Canada, June 11–13, 2003, Proceedings 16*, pp. 520–526, Springer, 2003.

[7] G. T. Huang, K. I. Cunningham, P. V. Benos, and C. S. Chennubhotla, "Spectral clustering strategies for heterogeneous disease expression data," in *Biocomputing 2013*, pp. 212–223, World Scientific, 2013.

[8] M. Arock *et al.*, "Distance-based k-medoids clustering for gene expression data.," *IUP Journal of Information Technology*, vol. 6, no. 2, 2010.

[9] T. Geetha and M. Arock, "Data clustering using modified k-medoids algorithm," *International Journal of Medical Engineering and Informatics*, vol. 4, no. 2, pp. 109–124, 2012.

[10] P. R. Kamble and R. D. Wajgi, "Parallel clustering of gene expression dataset in multicore environment," *International Journal of Innovative Research in Computer and Communication Engineering*, vol. 3, pp. 2300–2305, 2015.

[11] Y. Zhang, Z. Xiong, J. Mao, and L. Ou, "The study of parallel k-means algorithm," in *2006 6th World Congress on Intelligent Control and Automation*, vol. 2, pp. 5868–5871, IEEE, 2006.

[12] F. Nhita, "Comparative study between parallel k-means and parallel k-medoids with message passing interface (mpi)," *International Journal on Information and Communication Technology (IJoICT)*, vol. 2, no. 2, pp. 27–27, 2016.

[13] G. Ardaneswari, A. Bustamam, and T. Siswantining, "Implementation of parallel k-means algorithm for two-phase method biclustering in carcinoma tumor gene expression data," in *AIP Conference Proceedings*, vol. 1825, p. 020004, AIP Publishing LLC, 2017.

[14] B. Deb and S. N. Srirama, "Parallel k-means clustering for gene expression data on snow," *International Journal of Computer Applications*, vol. 71, no. 24, 2013.

[15] H. Kim, G. H. Golub, and H. Park, "Missing value estimation for dna microarray gene expression data: local least squares imputation," *Bioinformatics*, vol. 21, no. 2, pp. 187–198, 2005.

[16] S. Ma, "A rice gene expression matrix derived from large-scaled RNA-seq datasets," 7 2022.

[17] Unknown, "Ompi problem with mpi_file_read." Online forum, unspecified unspecified. Retrieved on April 23, 2023, from https://users.open-mpi.narkive.com/ZnPOW8bA/ompi-problem-with-mpi-file-read-2.

[18] "Aimos - artificial intelligence multiprocessing optimized system." https://cci.rpi.edu/aimos. Accessed: April 20, 2023.