MPI IMPLEMENTATION OF K-MEANS CLUSTERING

S.Hayati and A.Khornegah

March 31, 2025

Abstract

This project develops a faster parallel version of the K-Means Clustering algorithm, a widely used machine learning method, implemented with MPI in C++. The work begins with a basic serial version, modified to run in parallel for improved speed and efficiency compared to the original. An initial study analyzes the serial code to identify sections suitable for division across multiple processors. Scalability is tested by varying dataset sizes (from 20,000 to 80,000 points) and the number of processors. Tests are conducted on Google Cloud Platform virtual machines, utilizing a cluster of connected computers to enhance computational power.

1 WHAT IS K-MEANS CLUSTERING?

K-means is an unsupervised learning method for clustering data points. The algorithm iteratively divides data points into K clusters by minimizing the variance in each cluster. [1]

The main objective of k-means clustering is to partition your data into a specific number (k) of groups, where data points within each group are similar and dissimilar to points in other groups. It achieves this by minimizing the distance between data points and their assigned cluster's center, called the centroid. [2]

2 DATASETS GENERATION

To evaluate the algorithm, synthetic datasets were generated in Python using NumPy. For each dataset, three cluster centers were fixed at coordinates [1, 1], [3000, 3000], and [5000, 1], with a standard deviation of 400. For instance, in the 20,000-point dataset, 2D random points were created around these centers: 6,000 points around the first, 6,000 around the second, and 8,000 around the third, totaling 20,000 points.

Additional datasets ranging from 20,000 to 80,000 points were produced by proportionally increasing the number of points. Each dataset was saved as a CSV file and visualized using Matplotlib, as shown in (F1).

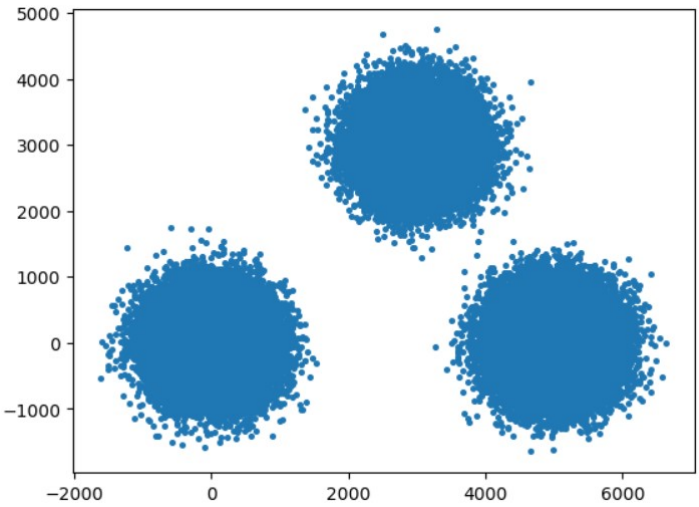

Figure 1: Generated dataset with three distinct clusters for K-Means clustering

3 SERIAL IMPLEMENTATION

This section describes the serial implementation of the K-Means Clustering algorithm, developed from scratch in C++ by utilizing Python code for the K-Means algorithm [3] as a reference and subsequently translating it into C++ and matplotlib-cpp library is used to visualize the clusters and centroids.

The focus lies on outlining the steps of the algorithm and providing an a priori estimate of computational costs to identify opportunities for parallelization. Additionally, the implementation includes functionality to visualize the clusters and centroids. The provided code performs K-Means clustering on a set of two-dimensional points.

3.1 Parameters

The serial version relies on the following parameters

Parameter	Description
dataFile	Path to the input data file
nSamples	Number of data points
K	Number of clusters
nFeatures	Number of features per sample (e.g., 2 for 2D points)
maxIterations	Maximum number of iterations

Table 1: Description of the parameters used in K-Means clustering.

3.2 Algorithm Steps

The serial K-Means algorithm processes a dataset through a sequence of steps to cluster data points based on their proximity to centroids. Given a dataset and the parameters above, the goal is to assign each point to one of K clusters and compute the final centroids. The steps are as follows (F2):

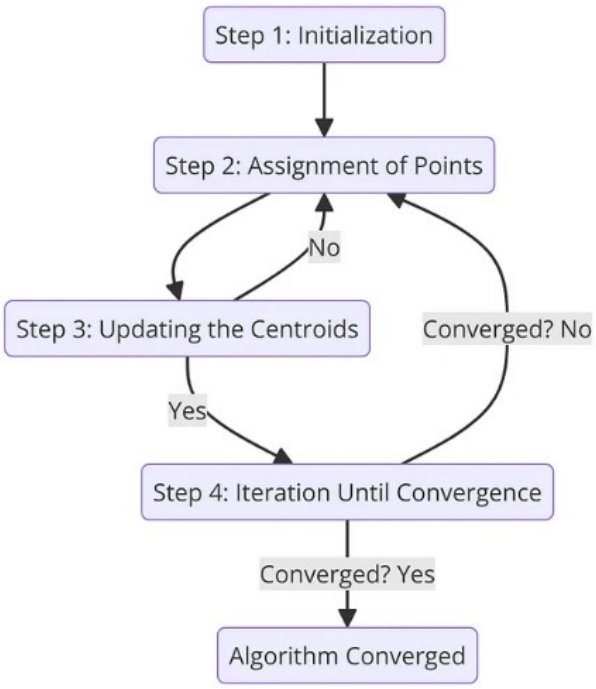

Figure 2: The sequence of steps in the K-Means clustering algorithm [4]

- **Initialization**: The algorithm begins by randomly selecting K initial centroids from the dataset or generating them randomly within the data range. This step establishes the starting point for clustering.

- **Assignment**: For each data point, the Euclidean distance to all K centroids is computed, and the point is assigned to the nearest centroid. This step groups the points into K clusters.
- **Update**: Each centroid is recalculated as the mean of all points assigned to its cluster.
- **Iteration**: The assignment and update steps repeat until centroids stabilize (no significant change) or the maximum number of iterations is reached.
- **Output**: The final centroids and cluster assignments are returned.

3.3 Application Example

The algorithm was tested on synthetic datasets generated in Python using NumPy, as described earlier. For instance, a dataset of 40,000 points was used with centers at [1, 1], [3000, 3000], and [5000, 1], a standard deviation of 400, and $K = 3$. The data was saved as `data_20K.csv` and loaded into the C++ implementation. The serial code was executed on the 40,000-point dataset on ubuntu as follows (F3) :

```
./serial_SA data_40K.csv
```

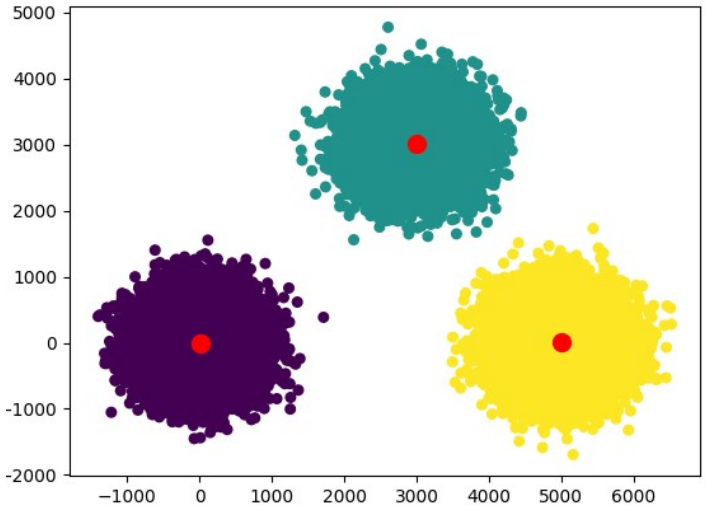

Figure 3: The executed serial code on the 40,000-point dataset

4 OBSERVATION FOR PARALLELIZATION

We installed the profiling tool `perf` on Ubuntu by setting up the required packages and adjusting the `perf_event_paranoid` setting to enable performance monitoring. Using `perf stat`, we analyzed the performance of the `serial_SA` program with a 40K dataset.

So MPI was used:

4.1 Data Distribution and Broadcasting:

Parallel Version:To distribute the workload and reduce communication overhead efficiently. (F4)

- **Function**: Constructor of `ParallelCluster`
- **MPI Functions Used**: `MPI_Bcast`
- **What was parallelized:**
  - Only rank 0 reads the CSV dataset and initializes the `Coordinate` objects.

– All `Coordinate` data is broadcast to other processes so that each process has a copy of the full dataset.

Serial version: Entire data is read by one process, no sharing or distribution needed.

```
// Allocate data points for all processes
dataPoints.resize(dataSize);

// Root process populates dataPoints
if (processId == 0) {
    for (int idx = 0; idx < dataSize; idx++) {
        dataPoints[idx] = new Coordinate(inputData[idx]->at(0), inputData[idx]->at(1));
    }
}

// Broadcast all data points to all processes to ensure consistency
for (int i = 0; i < dataSize; i++) {
    Coordinate temp;
    if (processId == 0) temp = *dataPoints[i];
    MPI_Bcast(&temp, sizeof(Coordinate), MPI_BYTE, 0, MPI_COMM_WORLD);
    if (processId != 0) dataPoints[i] = new Coordinate(temp.getCoordX(), temp.getCoordY());
```

Figure 4: Data Distribution via `MPI_Bcast`

4.2 Initial Centroid Selection and Distribution:

Parallel Version:These were performed by the root process and shared using `MPI_Bcast`.(F5)

- **Function**:Constructor of `ParallelCluster`
- **MPI Functions Used**: `MPI_Bcast`
- **What was parallelized:**
  - Only the root process randomly selects the `k` initial centroids.
  - Then broadcasts the initial centroids to all other processes so each can begin computation.

Serial version: Initial centroids are selected and used by one process, without sharing.

```
// Initialize cluster centers only on root and broadcast once
srand(time(NULL) + processId);
if (processId == 0) {
    vector<int> selectedIndices;
    int count = 0;
    while (count < clusterQty) {
        int randIdx = rand() % dataSize;
        if (find(selectedIndices.begin(), selectedIndices.end(), randIdx) == selectedIndices.end()) {
            selectedIndices.push_back(randIdx);
            clusterCenters.push_back(Coordinate(dataPoints[randIdx]->getCoordX(),
                                                dataPoints[randIdx]->getCoordY()));
            count++;
        }
    }
}
clusterCenters.resize(clusterQty);
MPI_Bcast(clusterCenters.data(), clusterQty * sizeof(Coordinate), MPI_BYTE, 0, MPI_COMM_WORLD);
```

Figure 5: Initial Centroid Selection and Distribution via `MPI_Bcast`

4.3 Workload Partitioning, Distance Calculation and Cluster Assignment:

Parallel Version:Distance calculations were divided among processes, and cluster assignments were synchronized using `MPI_Allgatherv`.

- **Function**:Constructor of `findNearestCenters()`
- **MPI Functions Used**: `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Allgatherv`
- **What was parallelized:**

- The Euclidean distance between each of its local data points and all cluster centroids is computed. (F6)
- Each data point is assigned to the nearest cluster based on these distances.(F7)
- After local assignments are complete, the results from all processes are gathered and synchronized using `MPI_Allgatherv`, ensuring that every process has the complete cluster assignment list.(F8)

Serial version: A single loop computes the distance between all data points and centroids, assigns the closest cluster, and stores the result in one process—no need for communication or data distribution.

```
// Calculate local segment for this process
int segmentSize = globalDataSize / totalProcesses;
int remainder = globalDataSize % totalProcesses;
int beginIdx = processId * segmentSize + min(processId, remainder);
int endIdx = beginIdx + segmentSize + (processId < remainder ? 1 : 0);
if (endIdx > globalDataSize) endIdx = globalDataSize;
```

Figure 6: Workload Partitioning

```
vector<AlignedIndex> localNearest(globalDataSize);
vector<float> distances;

// Compute nearest centers only for local segment
for (int i = beginIdx; i < endIdx; i++) {
    distances.clear();
    for (int j = 0; j < clusterCenters.size(); j++) {
        distances.push_back(computeDistance(dataPoints[i], clusterCenters[j]));
    }
    auto minDist = min_element(distances.begin(), distances.end());
    localNearest[i].clusterNum = distance(distances.begin(), minDist);
```

Figure 7: Distance Calculation + Assignment

```
// Gather all assignments efficiently
vector<int> recvCounts(totalProcesses);
vector<int> displs(totalProcesses);
int offset = 0;
for (int i = 0; i < totalProcesses; i++) {
    recvCounts[i] = segmentSize + (i < remainder ? 1 : 0);
    displs[i] = offset;
    offset += recvCounts[i];
}

MPI_Allgatherv(localNearest.data() + beginIdx, endIdx - beginIdx, MPI_BYTE,
               localNearest.data(), recvCounts.data(), displs.data(), MPI_BYTE,
               MPI_COMM_WORLD);

return localNearest;
```

Figure 8: Result Synchronization

4.4 Parallel Centroid Update with Reduction:

Parallel Version:For centroid updates, local sums and counts were reduced across all processes using `MPI_Allreduce`.(F9)

- **Function**: `updateCenters`
- **MPI Functions Used**: `MPI_Allreduce`
- **What was parallelized:**
  - Each process calculates local sums and counts for centroid updates.
  - All local results are reduced to a global sum using `MPI_Allreduce` to compute the new centroids.

Serial version: Single process calculates total sums for each cluster and updates centroids.

```
// Reduce sums and counts globally across all processes
vector<Coordinate> globalCenters(totalClusters, Coordinate(0, 0)); // Initialize to avoid garbage
vector<int> globalCounts(totalClusters, 0);
MPI_Allreduce(localCenters.data(), globalCenters.data(), totalClusters * 2, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
MPI_Allreduce(localCounts.data(), globalCounts.data(), totalClusters, MPI_INT, MPI_SUM, MPI_COMM_WORLD);

// Update centers on all processes
for (int i = 0; i < totalClusters; i++) {
    if (globalCounts[i] > 0) {
        clusterCenters[i].setCoordX(globalCenters[i].getCoordX() / globalCounts[i]);
        clusterCenters[i].setCoordY(globalCenters[i].getCoordY() / globalCounts[i]);
    } else {
        // If no points assigned to a cluster, keep the old center to avoid NaN
        clusterCenters[i] = clusterCenters[i];
    }
```

Figure 9: Centroid Update

5 AMDAHL'S LAW

To guess how much faster it can get, Amdahl's Law is used:

$$Speedup(N) = \frac{1}{S + \frac{P}{N}}$$

- N: Number of CPUs.
- S: Part of the code that has to stay on one CPU.
- P: Part that can split across CPUs (and $P + S = 1$).

In this project, approximately 75% of the code was estimated to be parallelized.

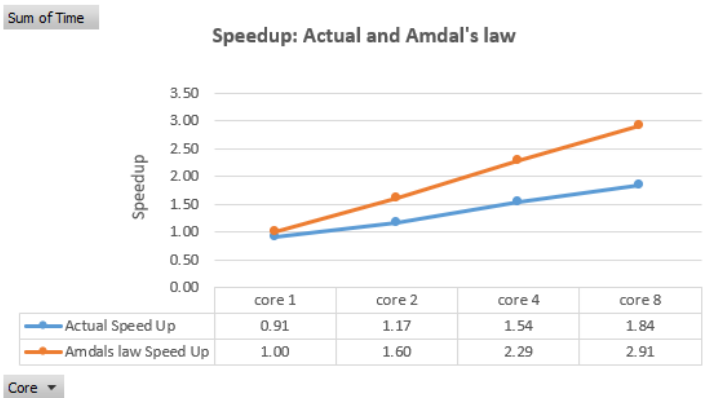

Figure 10: Amdahl's Law and the actual speedup

The comparison between the estimated speedup from Amdahl's Law and the actual speedup (as shown in the chart) reveals a gap between theoretical and practical performance. This highlights the influence of factors such as communication overhead, memory bandwidth limitations, and hardware inefficiencies.(F10)

6 PERFORMANCE ANALYSIS

This section evaluates the performance of the solution, focusing on execution time. Initial testing and debugging were conducted on local machines. Once the code passed various tests, it was deployed to Google Cloud Platform (GCP) clusters for performance evaluation.

To measure the performance of the MPI-based solution, two types of clusters were set up on GCP:

- **Fat Cluster:**
  - Configuration: Two nodes, each with 4 vCPUs.
  - Location: Intra-regional, both in the same region (us-central1-a and us-central1-b)

- **Light Cluster:**
  - Configuration: Four nodes, each with 2 vCPUs.
  - Location: Intra-regional, all in the same region (us-central1-a)

To make testing easier, a bash script was created to check the MPI-based K-Means program. The script uses a pre-generated dataset with at least 20,000 samples, 2 features, and 3 clusters. Larger datasets up to 80,000 samples were also tested, but starting with a single CPU for initial runs ensured manageable execution times before scaling to larger clusters. The user can choose to run tests on either the fat cluster (two nodes, 4 vCPUs each, in us-central1) or the light cluster (four nodes, 2 vCPUs each, in us-central1-a), varying the number of samples and CPU cores used. Other parameters, like the number of features (2) and clusters (3), affect execution time but were kept fixed to simplify the performance analysis. Execution times are displayed in the terminal and manually recorded into a comprehensive CSV file, capturing results for both serial and parallel runs on fat and light clusters.

6.1 Fat Clusters

A fat cluster uses a small number of powerful machines to handle heavy computing tasks efficiently. In this project, the fat cluster was set up with two machines, each having 4 vCPUs, located in us-central1 (zones us-central1-a and us-central1-b). This choice was made due to access limits and working on a free-tier GCP account, which restricted the total active vCPUs to 8 at any time.

6.1.1 Strong scalability

Strong scalability measures how well the parallel K-Means algorithm reduces execution time as more CPU cores are added, while keeping the dataset size fixed. The speedup is calculated using the following formula:

$$Speedup(N) = \frac{t(1)}{t(N)} \tag{1}$$

where $t(1)$ is the time to run the task with one core, and $t(N)$ is the time with N cores.

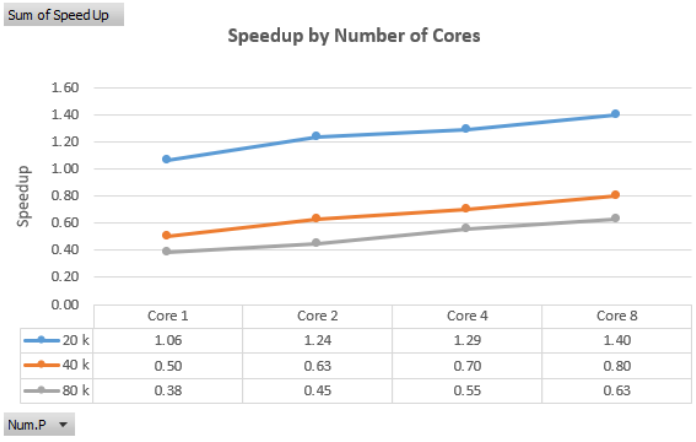

Figure 11: Strong scalability for the fat intra-regional cluster

The fat cluster, with two nodes (4 vCPUs each, totaling 8 vCPUs), was tested with datasets of 20,000, 40,000, and 80,000 points, varying the number of cores from 1 to 8. Figure 4: Strong scalability for the fat intra-regional cluster shows the results. The speedup increases as more cores are used, but not perfectly linearly. (F11)

For the 20,000-point dataset, speedup rises steadily, reaching around 1.4 with 8 cores, showing the best performance. For 40,000 points, it plateaus near 0.8, and for 80,000 points, it stays around 0.5.

This non-linear trend is due to MPI communication overhead (e.g., `MPI_Allgatherv` and `MPI_Allreduce`), which grows with more cores and impacts smaller datasets more, as each core has less work to do, making data transfer time more significant. Larger datasets show lower speedup because the fixed communication cost becomes a bigger fraction of the total time as the workload per core decreases.

6.1.2 Weak scalability

Efficiency shows how well the system scales when we increase both the number of processors and the problem size proportionally. It helps us understand how much useful work each processor is doing.

$$Efficiency(N) = \frac{t(1)}{t(N)} \tag{2}$$

Where:

- N is the number of processing units (e.g., cores or nodes).
- $t(1)$ is the time required to complete a single unit of work using one processing unit.
- $t(N)$ is the time required to complete N units of work using N processing units.

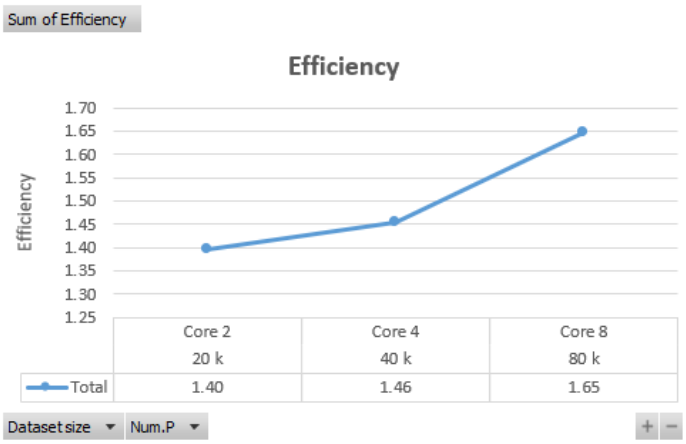

Figure 12: Weak scalability for the fat intra-regional cluster

The efficiency graph shows how well the system performs when both the dataset size and the number of processors grow together. As we can see, efficiency increases from 1.40 (with 2 cores and 20K data) to 1.65 (with 8 cores and 80K data). This means the system scales well: when we add more data and processors, the system still works efficiently without much slowdown. The results suggest good weak scalability and effective use of parallel resources.(F12)

6.2 Light Clusters

6.2.1 Strong scalability

The chart shows that speedup on light clusters does not significantly improve with more cores for fixed dataset sizes. As the number of cores increases, performance gains are limited, especially for larger datasets, due to communication overhead and limited processing power. This highlights scalability challenges in light cluster environments under strong scaling conditions.(F13)

6.2.2 Weak scalability

The efficiency chart for weak scalability on light clusters shows a positive trend. As the dataset size and number of cores increase proportionally—from 20K on 2 cores to 80K on 8 cores—efficiency also increases, reaching up to 0.72. This suggests that the parallel system handles larger workloads more effectively when resources scale accordingly. It indicates good weak scalability behavior, especially considering the limited computational power of light clusters.(F14)

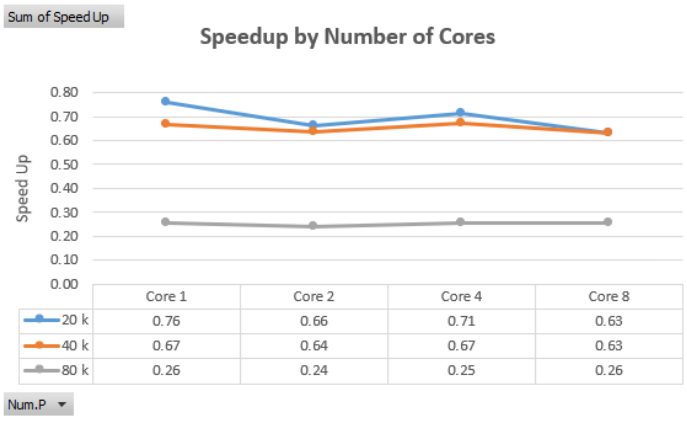

Figure 13: Strong scalability for the fat intra-regional cluster

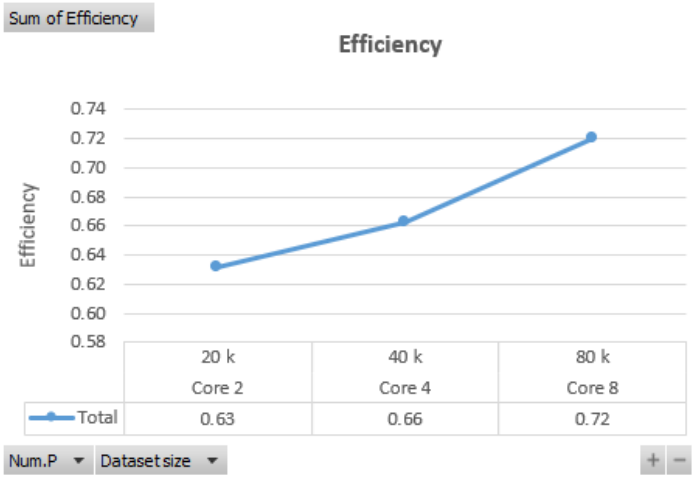

Figure 14: Weak scalability for the fat intra-regional cluster

7 CONTRIBUTIONS

Name	Responsibilities
Sepideh Hayati	- Parallel Implementation - GCP Clusters Configuration + Testing - Performance + Scalability Analysis - Report Writing - PowerPoint
Alireza Khornegah	- Serial Implementations - Datasets Generation - Available Parallelism + Amdahl's Law - GCP Clusters Configuration + Testing - Report Writing

Table 2: Team Members and Their Responsibilities

References

[1] W3School, *Machine Learning - K-means*. Available at: https://www.w3schools.com/python/python_ml_k-means.asp

[2] Pulkit Sharma, *K-Means Clustering Algorithm*. Available at: https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/

[3] Rashida048, *Machine Learning With Python - K-Means Clustering*, GitHub Repository, 2020. Available at: https://github.com/rashida048/Machine-Learning-With-Python/blob/master/k_mean_clustering_final.ipynb

[4] Cristian Leo, *The Math and Code Behind K-Means Clustering*. Available at: https://medium.com/data-science/the-math-and-code-behind-k-means-clustering-795582423666#81af