Advanced Computer Architecture

K-means clustering (Lloyd - Elkan - Hamerly - Yinyang)

August 2025

University of Pavia - Faculty of Engineering

INSTRUCTORS:
Prof. Marco Ferretti, Prof. Luigi Santangelo

ISSUED BY
Fatemeh Mohammadi (550691), Fereshteh Mohammadi (549898)

Table of Contents

Introduction

This project investigates the implementation of K-Means clustering, including four major algorithmic variants: Lloyd, Elkan, Hamerly, and Yinyang. Our focus is on developing an **MPI-based parallel version** of these algorithms and evaluating their performance in distributed environments. To this end, we deployed and tested the implementations on **Google Cloud Platform (GCP) clusters**, enabling experiments with different numbers of zones and cores. The goal of this study is to analyze the **scalability, efficiency, and runtime behavior** of each variant under parallel execution, and to compare the benefits and limitations of algorithmic optimizations versus parallelism in handling large datasets.

All source code and raw experimental results are available at:
https://github.com

K-means Algorithm

The K-Means algorithm is one of the most widely used clustering methods in unsupervised machine learning. Its goal is to partition a dataset into *K* clusters by minimizing the distance between data points and their assigned centroids. Several algorithmic variants have been proposed to improve efficiency, especially when dealing with large datasets

In this project, we focus on four major variants: **Lloyd (Classic), Elkan, Hamerly, and Yinyang**.

Lloyd (Classic)

The Lloyd algorithm is the standard and most commonly used implementation of K-Means. It follows an iterative process: (1) assign each point to the nearest centroid, and (2) recompute centroids as the mean of all assigned points. This process repeats until convergence. While straightforward, Lloyd's algorithm can be computationally expensive since it requires calculating distances between every point and every centroid in each iteration.

Elkan

Elkan's variant introduces optimizations based on the triangle inequality to reduce the number of distance computations. By maintaining upper and lower bounds for distances between points and centroids, it avoids redundant calculations when certain conditions are met. This leads to significant performance improvements, especially for high-dimensional data, though it requires additional memory to store the bounds.

Hamerly

Hamerly's algorithm simplifies Elkan's approach by keeping only one upper bound and one lower bound for each point. This reduces the memory requirements compared to Elkan while still eliminating many unnecessary distance computations. Although the pruning is less aggressive than Elkan's method, Hamerly's algorithm often achieves a favorable balance between speed and memory efficiency.

Yinyang

The Yinyang K-Means algorithm takes optimization further by grouping centroids and using multiple bounds per group to prune distance calculations more effectively. Instead of comparing each point to all centroids, only relevant groups are considered, which greatly reduces computational effort. Yinyang typically achieves the fastest performance among the variants, particularly when the number of clusters (K) is large. However, its implementation is more complex compared to the other approaches.

Serial Algorithm Performance and A Pre-Study of Parallelism

To evaluate the potential for parallelism, we measured separately the time spent in file input/output (I/O) and in the clustering computation. For this purpose, five independent serial runs were executed on a **133 MB dataset containing about 2 million rows**.

- **File read times (s):** 1.78062, 1.77465, 1.83601, 1.84473, 1.78048
- **Function (compute) times (s):** 2.41808, 2.46293, 2.50315, 2.69347, 2.38389

Averages:

- File I/O ≈ **1.80330 s**
- Compute ≈ **2.49230 s**
- Total ≈ **4.29560 s**

Function	Time (%)	Cumulative sec	Self Sec	Calls
Kmeans Compute	58.02%	4.296	2.492	1
Read File	41.98%	4.296	1.803	1

Thus, the portion we aim to parallelize is approximately:

$$p = 1 - \frac{I/O}{Total} = 1 - \frac{1.80330}{4.29560} \approx 0.580$$

According to Amdahl's Law:

$$Speedup(n) = \frac{n}{n + p\,(1 - n)}$$

where:

- n = number of cores (or MPI processes)
- p = parallelizable fraction of the program (≈ 0.580)

For n=14 and p=0.58:

$$Speedup(n) = \frac{14}{14 + 0.58\,(1 - 14)} = \frac{14}{14 - 7.54} \approx 2.17$$

It means the **parallel version (with 14 cores)** can run up to **2.17 times faster than the serial version in theory**, according to Amdahl's Law:

$$\frac{2.49}{2.17} \approx 1.15\ s$$

MPI Implementation

Our parallelization targets the **assignment/update loop** of k-means while keeping file I/O centralized on rank 0. The dataset is a semicolon-separated text file; the set of feature columns (DIM) is chosen at runtime.

Work partitioning

Let N be the number of valid rows after cleaning and P the number of MPI ranks. Rank 0 computes a block (almost) even partition:

$$\text{count}_r = \left\lfloor \frac{N}{P} \right\rfloor + \begin{cases} 1, & r < N \bmod P \\ 0, & \text{otherwise} \end{cases}, \qquad \text{displ}_r = \sum_{j<r} \text{count}_j$$

We keep the points in a **row-major, flattened** layout of length N×DIM. Therefore the buffer sizes used by `MPI_Scatterv` / `MPI_Gatherv` are `count_r * DIM` and `displ_r * DIM`.

Data distribution and setup

1. **Configuration broadcast:** Rank 0 parses CLI flags, computes `DIM`, and broadcasts (K,limit,DIM) and the selected column indices with:

```
MPI_Bcast(&K, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&limit, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&DIM, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
MPI_Bcast(cols.data(), DIM, MPI_INT, 0, MPI_COMM_WORLD);
```

2. **Centralized I/O:** Rank 0 opens `household_power_consumption.txt`, tokenizes each line by "**;**", skips rows with missing values **(?)**, converts only the chosen columns to **`double`**, and pushes them into **`allP`** (size N×DIM). We time this as **I/O time**.

3. **Row count broadcast:** `MPI_Bcast(&N, 1, MPI_LONG_LONG_INT, 0, COMM)` so all ranks know the global problem size.

```
long long N64 = N;
MPI_Bcast(&N64, 1, MPI_LONG_LONG_INT, 0, MPI_COMM_WORLD);
```

4. **Data scatter.** Each rank receives its contiguous block with:

```
MPI_Scatterv(rank == 0 ? allP.data() : nullptr,
             sc_counts.data(),
             sc_displs.data(),
             MPI_DOUBLE,
             P.data(),
             nloc * DIM,
             MPI_DOUBLE,
             0,
             MPI_COMM_WORLD);
```

5. **Initial centroids.** Rank 0 samples K distinct points from **`allP`** (simple shuffle) and broadcasts **C** (shape K×DIM) via **`MPI_Bcast`**.

Iterative k-means (parallel region)

We iterate until convergence or `max_iter`. Each iteration has the following phases:

1. **Precompute on rank 0:**
    a. Center displacements

$$\Delta_c = \left| C_c - C_c^{prev} \right|_2$$

    b. For **Hamerly**, the center-separation bounds

$$S_c = \frac{1}{2} min_{j \neq c} \left| C_c - C_j \right|$$

c. For **Elkan** and **Yinyang**, the full pairwise center distance matrix

$$D_{cc'}$$

d. For **Yinyang**, a centroid grouping g(c) into G groups (small k-means on centroids).

e.

These small vectors/matrices are broadcast with **`MPI_Bcast`** so all ranks can prune distance checks.

2. **Local assignment on each rank:**
   Depending on the chosen variant (Lloyd/Elkan/Hamerly/Yinyang), each process assigns its **`count_r`** points using the appropriate bounding logic and accumulates:

   - $sum\,[c,\ d] = \sum local\ points\ in\ cluster\ c\ of\ coordinate\ d$
   - $cnt[c] = \#\ local\ points\ in\ cluster\ c$

3. **Global reduction of partials:**
   `MPI_Allreduce` on **`sum`** and **`cnt`** to obtain global **`gsum`** and **`gcnt`**.

4. **Centroid update on rank 0:**
   - For each cluster with $gcnt[c] > 0$, set

$$C_c = \frac{gsum_c}{gcnt_c}$$

   - Track **max_shift = max_c || C_c _ C_c^{prev} ||**

5. **Convergence broadcast:**
   - Rank 0 broadcasts the new **`C`** and **`max_shift`**.
   - If **max_shift < tol**, terminate.

Result collection

After convergence, each rank holds its **`label`** array. We gather the global labels on rank 0:

```
MPI_Gatherv(label.data(), nloc, MPI_INT,
            rank == 0 ? allLab.data() : nullptr, counts_lbl.data(), displs_lbl.data(), MPI_INT,
            0, MPI_COMM_WORLD);
```

Rank 0 then writes **`clustering_result.txt`** (final centers and per-cluster point lists).

MPI primitives used

- **`MPI_Bcast`**: configuration, initial/final centroids, and per-iteration metadata.
- **`MPI_Scatterv`** / **`MPI_Gatherv`** : distribute and collect variable-sized contiguous blocks.
- **`MPI_Allreduce`**: sum partial centroid statistics across ranks.
- **`MPI_Barrier`**: delimit timing regions.
- **`MPI_Wtime`**: wall-clock timing for I/O and compute.

Timing model (what we report)

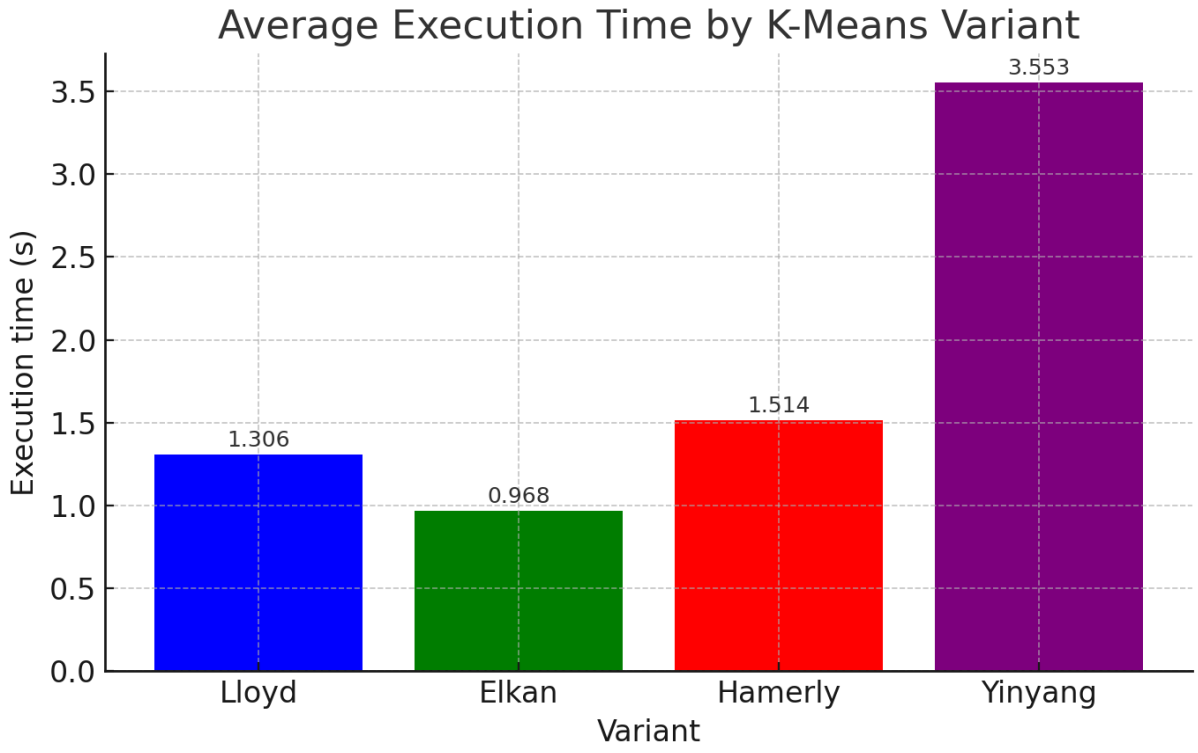
- **I/O time**: wall time on rank 0 for reading, cleaning, and parsing the file.
- **Compute time**: wall time (all ranks) covering the iterative loop: broadcasts of metadata, local assignments, allreduces, centroid updates on rank 0, and centroid broadcasts.
- **Total time**: I/O + Compute.

Cluster Results

We tested different cluster configurations on GCPs Computer engine and measured the execution time. Here are the configurations used:

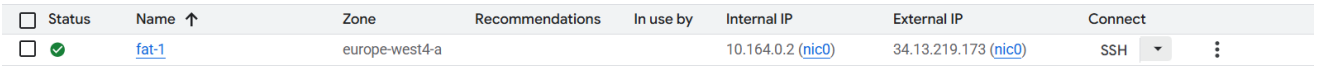
- **Light Multi-Zone Cluster (3 zones, 3 VMs, 4 cores each)**
    - Zones used: us-central1-a, us-central1-b, us-central1-c

State	Name	Area	Suggestions	Used by	Internal IP	External IP	Connect
✓	mpi-1	us-central1-a			10.128.0.2 (nic0)	34.172.178.233 (nic0)	SSH
✓	mpi-2	us-central1-b			10.128.0.3 (nic0)	35.232.77.18 (nic0)	SSH
✓	mpi-3	us-central1-c			10.128.0.5 (nic0)	34.59.180.186 (nic0)	SSH

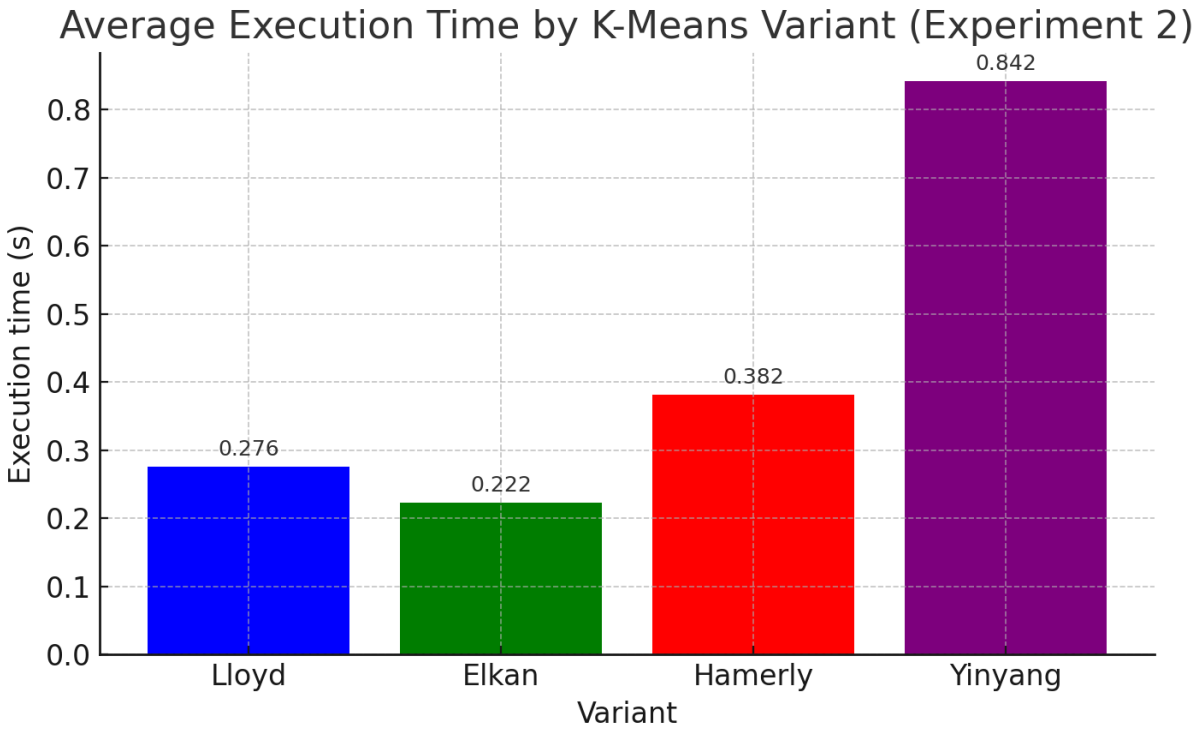


For this dataset with **2 million points and k=5**, the results show that **Elkan** is the fastest (~0.97s) thanks to its efficient distance pruning, while **Lloyd** (~1.31s) serves as a solid baseline. **Hamerly** (~1.51s) introduces overhead without enough pruning benefit at low k, making it slower than Lloyd. **Yinyang** (~3.55s) performs worst because its grouping mechanism only becomes advantageous when **k is large**; with small k, the overhead dominates. Overall, for small k values, **Elkan is clearly the best choice**.

- **Fat Cluster (1 zone, 1 VMs, 12 cores)**
    - Zone used: europe-west4-a



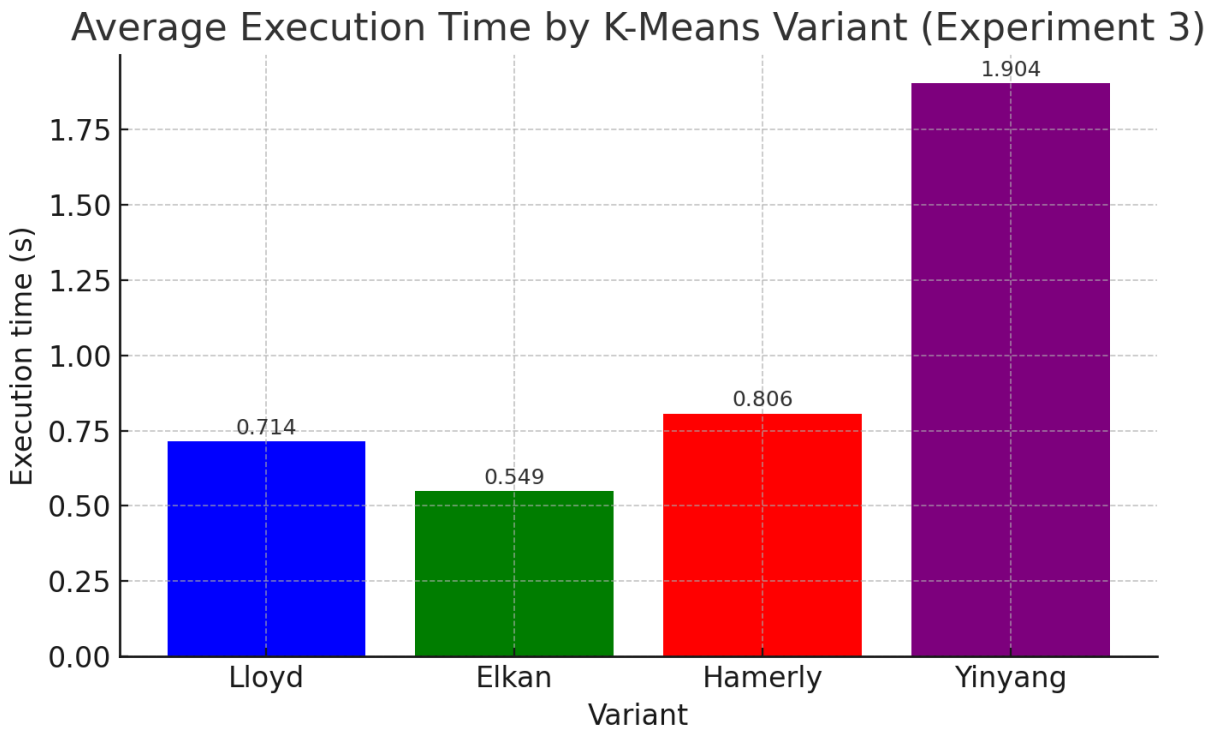
Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	fat-1	europe-west4-a			10.164.0.2 (nic0)	34.13.219.173 (nic0)	SSH



As we see, the pattern is the same as in the previous experiment: **Elkan remains the fastest, Lloyd is baseline, Hamerly adds overhead without benefit at low k, and Yinyang is the slowest due to grouping overhead.**

- **Light Multi-Zone Cluster (2 zones, 4 VMs, 2 cores each)**
    - Zone used: europe-west1-b, europe-west1-d

Status	Name	Zone	Recommendations	In use by	Internal IP	External IP	Connect
✓	light-1	europe-west1-b			10.132.0.10 (nic0)	34.79.4.72 (nic0)	SSH
✓	light-3	europe-west1-d			10.132.0.12 (nic0)	146.148.124.146 (nic0)	SSH
✓	light-4	europe-west1-b			10.132.0.13 (nic0)	34.140.231.200 (nic0)	SSH
✓	light-6	europe-west1-d			10.132.0.15 (nic0)	35.205.108.65 (nic0)	SSH



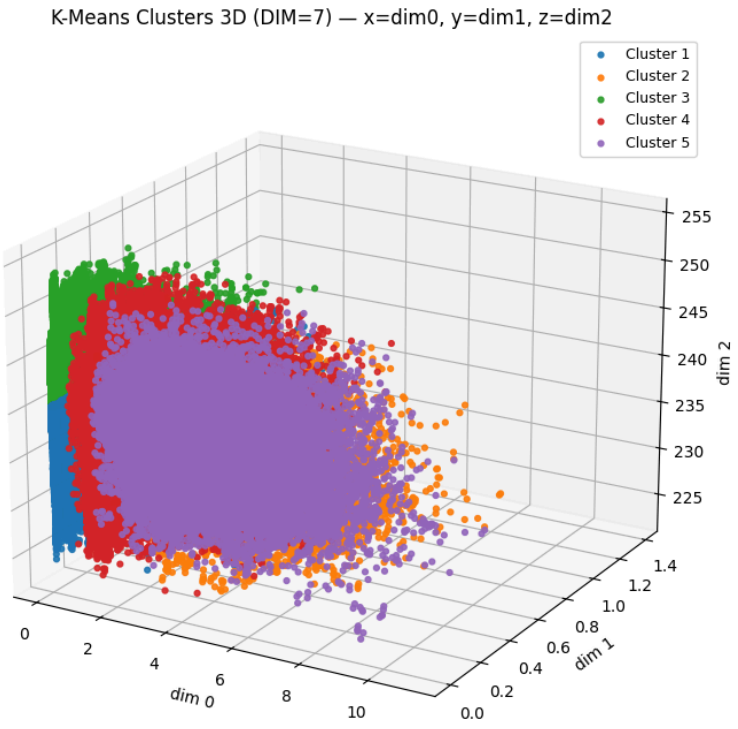
Across all three experiments, the trend is consistent: **Elkan** is always the fastest due to its efficient pruning, **Lloyd** serves as a reasonable baseline, **Hamerly** is slower at low k because the pruning overhead outweighs the benefits, and **Yinyang** is consistently the slowest since its grouping mechanism only pays off with very large k.

Overall, for datasets with **millions of rows but small k (like k=5)**, **Elkan is the best choice**.

Here are some screenshots of the shell output and clustering result visualization:

```
fatemeh_mohammadi01@light-1:~$ cd ~/kmeans_clustering/MPI
fatemeh_mohammadi01@light-1:~/kmeans_clustering/MPI$ mpirun -np 8 --hostfile ~/hosts ./kmeans_mpi
K=5 | DIM=7 | cols: 2 3 4 5 6 7 8 | limit=-1
Choose k-means variant:
 [1] Lloyd
 [2] Elkan
 [3] Hamerly
 [4] Yinyang
Enter choice: 1
--------------------------------------------------------------------------
Converged after 42 iteration(s)
Variant: Lloyd
MPI ranks: 8
Rows used (N): 2049280 | Features (DIM): 7 | K: 5
I/O time (s): 9.62857
Compute time (s): 0.698127
Total time (s): 10.3267
Wrote "/home/fatemeh_mohammadi01/kmeans_clustering/MPI/clustering_result.txt"
fatemeh_mohammadi01@light-1:~/kmeans_clustering/MPI$
```

```
 [1] Lloyd
 [2] Elkan
 [3] Hamerly
 [4] Yinyang
Enter choice: 2
--------------------------------------------------------------------------
Converged after 42 iteration(s)
Variant: Elkan
MPI ranks: 8
Rows used (N): 2049280 | Features (DIM): 7 | K: 5
I/O time (s): 9.75097
Compute time (s): 0.537641
Total time (s): 10.2886
Wrote "/home/fatemeh_mohammadi01/kmeans_clustering/MPI/clustering_result.txt"
fatemeh_mohammadi01@light-1:~/kmeans_clustering/MPI$ mpirun -np 8 --hostfile ~/hosts ./kmeans_mpi
Choose k-means variant:
 [1] Lloyd
 [2] Elkan
 [3] Hamerly
 [4] Yinyang
Enter choice: K=5 | DIM=7 | cols: 2 3 4 5 6 7 8 | limit=-1
2
--------------------------------------------------------------------------
Converged after 42 iteration(s)
Variant: Elkan
MPI ranks: 8
Rows used (N): 2049280 | Features (DIM): 7 | K: 5
I/O time (s): 9.73024
Compute time (s): 0.543386
Total time (s): 10.2736
Wrote "/home/fatemeh_mohammadi01/kmeans_clustering/MPI/clustering_result.txt"
fatemeh_mohammadi01@light-1:~/kmeans_clustering/MPI$
```



Conclusion

We implemented a parallel version of the K-Means clustering algorithm supporting multiple variants (Lloyd, Elkan, Hamerly, Yinyang) using MPI. The program processes the *household_power_consumption* dataset, where the input points are read from file and distributed across processes in a row-major flattened layout.

Through testing on a multi-core system, our implementation showed that MPI parallelization significantly reduces computation time compared to the serial baseline. The results demonstrate strong scalability in the compute-intensive phase of K-Means, while I/O remains a bottleneck due to its sequential execution on the master process.

Our comparative study revealed that:

- The MPI-based parallelization achieves notable speedup, with performance trends consistent with Amdahl's Law.
- Different K-Means variants exhibit varying computational overheads: Lloyd remains the most stable baseline, while Elkan and Hamerly provide moderate pruning benefits, and Yinyang shows higher costs on medium-scale data.
- Overall scalability is limited by file I/O and communication overhead, but the compute phase itself is highly parallelizable.

Individual Contributions

While the entire project was developed collaboratively, each member took particular responsibility for different aspects:

Fatemeh Mohammadi (550691): Focused on conducting the experiments on Google Cloud Platform (GCP) clusters, produced 3D performance visualizations using Python to illustrate clustering results.

Fereshteh Mohammadi (549898): Implemented the MPI-based parallelization of the K-Means algorithm, developed and tested the different algorithmic variants (Lloyd, Elkan, Hamerly, Yinyang).

Both students contributed jointly to the preparation, structuring, and writing of the final report.

References

- https://cdn.aaai.org/ICML/2003/ICML03-022.pdf
- https://www.ccs.neu.edu/home/radivojac/classes/2024fallcs6140/hamerly_sdm_2010.pdf
- https://github.com/phrshteh/kmeans_clustering