Parallel Expectation-Maximization clustering for GMM

Martina De Piccoli (mat. 267338) & Jago Revrenna (mat. 267325)

University of Trento, High Performance Computing for Data Science — 2025/2026

github.com/martinadep/parallel_EM_clustering

martina.depiccoli@studenti.unitn.it, jago.revrenna@studenti.unitn.it

ABSTRACT

This project focuses on the parallelization of the Expectation-Maximization (EM) algorithm for Multivariate Gaussian Mixture Models (GMM), investigating two different approaches. The first explores the Message Passing Interface, implementing a full-MPI parallelization; whereas the second explores the shared-memory technique through a full-OpenMP parallelization. The aim of this study is to provide insights into the effectiveness of different HPC techniques comparing their efficiency, scalability, and overall impact on computational performance. Code and results are available in the GitHub repository(6).

1 INTRODUCTION

In recent years, the volume of data being generated, transferred, and managed has grown exponentially, introducing the need of adapting and optimizing data analysis techniques for large-scale datasets. In this context, clustering procedures are fundamental tools for data understanding, as they are able to group homogeneous data points based on underlying patterns or similarities, without the explicit need to define boundaries. Nevertheless, in some specific cases, the *shape* of the data distribution is known, but we lack information about the *parameters* that describes it. Here comes in help the Expectation-Maximization algorithm, a statistical procedure that is able to learn these parameters directly by observing the provided samples. In this project, data are generated from a multivariate Gaussian Mixture Model, therefore EM clustering assumes data to be combination of multiple Gaussian distributions.

Expectation Maximization for GMM is composed of two primary steps:

(i) The **Expectation** (E) step: Given the current parameters for each Gaussian component, calculate the probability that a given data point was generated by a specific Gaussian. The sum over data points of these probabilities represents the ”responsibilities” of each Gaussian component.

(ii) The **Maximization** (M) step: Update the means, covariances, and mixing weights of each Gaussian component using the responsibilities as weights. This maximizes the likelihood of the data given the current cluster assignments.

However, despite being intuitive, EM procedure consists of intensive computations, especially when considering large or high dimensional datasets. Therefore, the aim of this project is leveraging High-Performance Computing (HPC) techniques to distribute the workload across multiple processing nodes, allowing for significant execution time reductions.

This report explores two different parallel implementations of a the EM algorithm for GMM, one using Message Passing Interface, and one exploiting shared-memory using OpenMP. Overall, the focus is on partitioning large datasets to minimize communication overhead, managing the global update of parameters during the M-step using specific directives and, finally, on evaluating how the system performs as the number of processors and the dataset size increase.

[H]

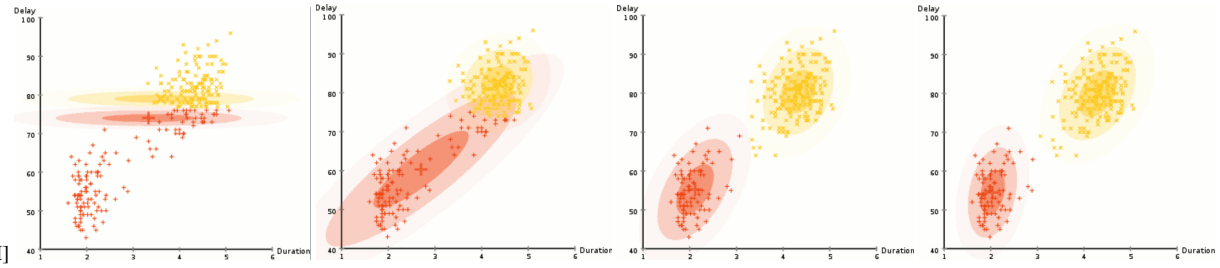


Figure 1. EM clustering for dimensional dataset. The estimations are adjusted throughout the iterations, eventually fitting the data (18).

1.1 EM Clustering for GMM

The Gaussian Mixture Model (GMM) is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. Mathematically, the probability of a d-dimensional observation x_n is defined as a weighted sum of K multivariate Gaussian densities $\mathcal{N}(x_n|\mu_k, \Sigma_k)$:

$$p(x_n|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k), \qquad \mathcal{N}(x|\mu, \Sigma) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right) \tag{1}$$

where π_k represents the mixing coefficient for the k-th component, such that $\sum_{k=1}^{K} \pi_k = 1$.
The Expectation-Maximization (EM) algorithm aims to find the Maximum Likelihood Estimate (MLE) of the parameters $\theta = \{\pi_k, \mu_k, \Sigma_k\}$, therefore it aims to maximize:

$$\ln L(\theta) = \sum_{n=1}^{N} \ln(\sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)) \tag{2}$$

However, the maximization of Equation 2 is analytically untractable due to the summation inside the logarithm that doesn't reduce to a *"closed-form"*. To overcome this problem, the EM algorithm starts by random guesses of the parameters and proceeds iteratively:

- **E-step**: Computes the *responsibilities* γ_{nk}, which represent the posterior probability that point x_n belongs to cluster k:

$$\gamma_{nk} = \frac{\pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x_n|\mu_j, \Sigma_j)} \tag{3}$$

- **M-step**: Updates the parameters by maximizing the expected log-likelihood using the responsibilities calculated in the E-step:

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} x_n, \quad \Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk}(x_n - \mu_k)(x_n - \mu_k)^T, \quad \pi_k^{new} = \frac{N_k}{N} \tag{4}$$

where $N_k = \sum_{n=1}^{N} \gamma_{nk}$ is the effective number of points assigned to cluster k.

2 STATE-OF-THE-ART

The parallelization of the Expectation-Maximization algorithm for Gaussian Mixture Models has been extensively studied in literature, especially by exploiting parallel techniques that could overcome the computational bottlenecks of the algorithm for large scale datasets. However, despite being computationally intensive, EM is highly parallelizable by nature, since most of its iterations are independent.

Historically, distributed memory implementations utilized a Master-Slave architecture (10; 11). This model involves a central process that manages data and job scheduling, while slave processes perform the intensive calculations such as the multivariate Gaussian PDF. Although this model represent a robust approach by balancing workloads, it can suffer from communication overhead when managing many processes.

Accordingly, modern solutions shifted toward a SPMD (Single Program, Multiple Data) approach, leveraging primitives for collective communications that enable more efficient data aggregation across nodes. In 2013, Chen et al. introduced the *Expectation-Gathering-Maximization* framework, which adds a gathering step to combine sufficient statistics from independent parallel nodes (4). Similarly, an asynchronous EM clustering model was proposed by Plant et. al (13). Their idea was allowing asynchronous model updates in the M-phase. Each thread holds its own local data and current parameter estimates, until, depending on its own workload and speed, and availability of the network, the process independently decides when to perform a model update.

Other contemporary solutions focused on new architectures like GPUs, using CUDA (8; 9), or hybrid CPU-GPU systems (17). These architectures requires a specific code restructure to exploit GPU's parallel units, but enables impressive speedup compared to the CPU-only versions, particularly as the dimensionality increases.

This project concentrates on the SPMD parallel approach, by dividing independent computations among threads and then gathering and sharing the results when needed. A first, more structured, implementation consists on Message Passing Interface and the use of explicit sending/receiving messages (14). The second implementation exploits OpenMP shared-memory framework, which enables efficient parallelizations through compiler directives, with minimal changes to the original serial code (14).

3 SEQUENTIAL IMPLEMENTATION

Before delving into the details of the C implementation, it is noteworthy to discuss the parameters initialization. It is widely known that the convergence rate and the quality of the final Expectation-Maximization model are highly sensitive to initial conditions. Consequently, instead of purely random initialization, a `init_gmm` function has been developed, which works as follows:

- **Means (μ_k):** Each centroid is initialized by leveraging the *K-means++* seeding heuristic (2). The first centroid is chosen randomly from the dataset and each subsequent centroid is selected by finding the data point x_n that maximizes the distance to its nearest existing centroid. This approach ensures that the initial clusters are spread across the feature space, enabling each Gaussian to explore a different region and allowing for more data coverage.
- **Covariance Matrices (Σ_k):** All clusters are initialized with the *global covariance* of the entire dataset. The function first computes the global mean $\bar{x}$ and then the total covariance $\Sigma_{global} = \frac{1}{N}\sum(x_n - \bar{x})(x_n - \bar{x})^T$. This prevents Gaussians to be too "narrow", possibly avoiding points, or too "spread", possibly interfering and overlapping with each other in the early stages.
- **Mixing Weights (π_k):** Assuming no prior knowledge, the weights are initialized uniformly as $\pi_k = 1/K$.

3.1 EM implementation

The sequential implementation follows the algorithm described in Section 1.1, and is lead by the `em_algorithm` function, which calls `e_step` and `m_step` until numerical convergence is reached. The `log_likelihood` is calculated after each iteration, and whether the update stays within a certain `EPSILON` threshold, the algorithm has converged.

A critical component of this implementation is the handling of the multivariate Gaussian probability density function, since it requires to calculate the inverse of the covariance matrix Σ^{-1} to compute the exponent term. Using the canonical co-factor methods for matrix inversion would soon degrade performance due to its high computational cost for large matrices. This problem was addressed using LUP decomposition, which provides a more stable and fast method for solving the inverse (3). Furthermore a regularization term of 10^{-6} is added to the diagonal elements of Σ during the M-step, in order to prevent singular covariance matrices which would lead clusters to collapse. The following pseudo-code 1 summarizes the sequential execution flow implemented in this project.

Algorithm 1 Sequential EM for GMM

Initialize: π_k, μ_k, Σ_k using `init_gmm`
$L_{prev} \leftarrow -\infty$
for $iter = 0$ **to** MAX_ITER **do**
  {E-Step}
  for all point $n \in \{1 \dots N\}$ **do**
    $norm \leftarrow 0$
    for all cluster $k \in \{1 \dots K\}$ **do**
      $resp[n][k] \leftarrow \pi_k \cdot$ multiv_gaussian_pdf$(x_n, gmm[k])$
      $norm \leftarrow norm + resp[n][k]$
    end for
    Normalize $resp[n]$ by $norm$
    Accumulate $gmm[k].class\_resp$ for M-step
  end for
  {M-Step}
  for all cluster $k \in \{1 \dots K\}$ **do**
    $\pi_k \leftarrow gmm[k].class\_resp/N$
    $\mu_k \leftarrow$ weighted average of x_n using $resp[n][k]$
    $\Sigma_k \leftarrow$ weighted covariance of $(x_n - \mu_k)$
  end for
  $L_{curr} \leftarrow$ log_likelihood(X, θ)
  if $|L_{curr} - L_{prev}| < \epsilon$ **then**
    break
  end if
  $L_{prev} \leftarrow L_{curr}$
end for

As illustrated in the pseudo-code (Algorithm 1), the Gaussian Mixture Model is represented by an array named `gmm`. This represents the Gaussian Mixture Model and consists of K Gaussian components. Although employing a structure may introduce performance overhead and requires custom management for OpenMP `reduction` clauses, this design choice was prioritized to enhance code readability and maintainability.

```
typedef struct {
    double *mean;       // Mean vector
    double **cov;       // Covariance matrix
    double weight;      // Mixture weight (pi_k)
    double class_resp;  // Class responsibility
} Gaussian;
```

Listing 1: Gaussian Structure Definition

4 DATA DEPENDENCIES

The dependencies for the EM algorithm for GMM (Algorithm 1.1) are summarized in the following table. This helps identifying opportunities for parallelization and detecting in advance potential bottlenecks related to shared data or synchronization requirements.

Stage	**Notes on Data Dependencies**
Initialization [Line 1]	Parameters (π_k, μ_k, Σ_k) are initialized independently hence there are **no dependencies** between clusters.
Main Loop [Line 3]	Each iteration depends on the parameters calculated in the previous state ($\theta^{(t-1)}$), resulting in a **sequential dependency** that prevents parallelizing the iterations themselves.
E-Step: Responsibility [Lines 5-11]	The calculation of γ_{nk} for each observation is independent of all other observations. There are **no dependencies** between data points, thus allowing for massive parallelization over N. Additionally, this is also highly convenient considering that N is typically much larger than the number of clusters K or the dimensionality D of each point
E-Step: Accumulation [Line 12]	A **reduction dependency** exists because the responsibilities of all points N must be accumulated into shared variables ($gmm[k].class\_resp$) for the M-step, therefore this requires additional synchronization to avoid inconsistency.
M-Step: Parameters [Lines 15-18]	Updating cluster parameters depends on the responsibilities from the E-step. A **reduction dependency** occurs during the calculation of weighted means and covariances across N points.
Convergence Check [Lines 20-22]	Calculating the global log-likelihood requires a **reduction** across all points. The stop condition has a **sequential dependency** on the log-likelihood of the previous iteration.

Table 1: Data dependency analysis for the GMM EM algorithm.

While other complementary functions such as file I/O and matrix operations could benefit from parallelization, the optimization efforts in this project were focused specifically on the EM clustering algorithm, as it represents the primary objective of this study.

5 PARALLEL DESIGN

This section analyzes the two parallel implementation developed from the sequential EM clustering for GMM presented in Section 3. Design choices are discussed for both OpenMP and MPI approaches.

In line with standard HPC practices, we chose to flatten the $N \times D$ matrix into a contiguous data structure. This solution, which incorporates data dependencies considerations in Table 1, was implemented to leverage spatial and temporal cache locality, maximizing memory access efficiency during parallel execution.

5.1 Parallel Implementations

In designing the parallel version, two main strategies were initially evaluated. First it was considered to distribute the K Gaussian clusters among processors. However, this would have lead to high load imbalance when K is small or if some clusters converge faster than others. Thus, also considering Table 1, our design focused on data decomposition, where the N data points were partitioned across processing units, ensuring more balanced workloads and overall better performance.

5.1.1 OpenMP (Shared-Memory)

The `e_step` and `m_step` functions were parallelized using OpenMP by distributing the computational workload across the N data points. In the `e_step`, a `#pragma omp parallel for` directive enables the concurrent calculation of responsibilities, while the reduction of the each responsibility is handled via the `reduction(<op>:<var>)` clause. In the `m_step`, parallelization is nested within the cluster loop, distributing the accumulation of means and covariance matrices over the number of data points. Unlike MPI, OpenMP operates on a shared-memory architecture, all threads maintain direct access to the global data structures, which eliminates the need for explicit data transfers with send and receive calls.

However, this shared-memory model requires careful management of concurrent updates to avoid race conditions or performance degradation typically associated with `atomic` or `critical` sections. To address this we exploited the `reduction` clause which allows each thread to accumulate partial sums into private copies of the arrays, and later automatically aggregates these partial results at the end of the parallel region. This approach minimizes synchronization overhead and avoids high contention on the global memory bus, though its scalability remains limited at low thread counts because of the contention of cache lines and memory bandwidth limits.

5.1.2 MPI (Distributed-Memory)

As already mentioned above, the dataset is physically partitioned among the available processes, with each rank responsible for computations on a distinct subset of size N/P, while the problem dimensions are broadcasted to all the processes using `MPI_Bcast`.

- **Data distribution:** The 1D array containing data is first distributed using `MPI_Scatter`, and each process reconstructs a local array of pointers to maintain compatibility with the algorithm's logic.

```
// distribute chunks to all processes
MPI_Scatter(flat_dataset, local_N * dim, MPI_DOUBLE, local_flat_data,
            local_N * dim, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Listing 2: Data scattering (main.c)

- **Parallel execution:** The Expectation step (E-step) is executed locally without communication, as responsibilities depend only on local data points and the current global parameters.
Instead, the Maximization step (M-step) requires global aggregation of sufficient statistics (sum of weights, means, and covariances). To achieve this efficiently, each process computes partial local sums, which are then aggregated across all nodes using `MPI_Allreduce` with the `MPI_SUM` operator. This ensures that all processes simultaneously obtain the updated global parameters for the next iteration without a dedicated broadcast step.

```
// global reduction: sum partial results from all processes
MPI_Allreduce(local_sum_resp, global_sum_resp, num_clusters,
                MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

MPI_Allreduce(local_sum_means, global_sum_means, num_clusters * dim,
                MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Listing 3: Global reduction of statistics (em_algorithm.c)

Finally, `MPI_Gather` is employed at the end of the execution to collect the final cluster assignments (labels) from all workers back to the Master node for output generation.

6 EXPERIMENTS

In HPC, the *scaling* analysis measures the ability of a system to maintain or improve performance as the number of processors or workloads increases (16). Two main types of scalability are commonly considered:

- **Strong scaling**: the number of processors is increased while keeping the problem size fixed, resulting in a reduced workload per processor (15). Ideally, the execution time should halve when doubling the number of processors.
- **Weak scaling**: both the number of processors and the problem size are increased, maintaining a constant workload per processor (15). In this case, the execution time should ideally remain constant as the system scales.

In this work, **strong scalability** is mainly considered since the complexity of the algorithm depends on many variables, hence, defining a uniform "workload unit" for weak scaling is inherently complex. In fact, EM algorithm's complexity is affected not only by the dataset size (N), but also by the number of clusters (K), the dimensionality of the features (D), and the number of iterations (I) required for convergence, with an approximate time complexity of $O(NKD^3I)$. Since I is highly dependent on the data and can vary significantly as the problem size scales, strong scaling provides a more reliable measure of how parallelization reduces the execution time.

6.1 System Description

Following benchmarks were conducted on a Linux system with an x86-64 architecture supporting 64-bit operations and configured with 96 physical CPUs distributed across 4 sockets of 24 cores each. The processor used is the Intel(R) Xeon(R) Gold 6252N CPU, with a hierarchical cache design as following: L1d and L1i of 32KB, L2 of 1024KB, and a shared L3 of 36MB cache per socket. The system is NUMA-based with 4 NUMA nodes. The OpenMP implementation was compiled using `gcc 9.1.0`, whereas the MPI implementation was compiled using `OpenMPI 4.0.4` and `gcc 9.1.0`.

6.2 Dataset Description

For the experiments several different GMM has been utilized as further explained below. Each dataset was generated with R (code available in the GitHub repository (6)).

6.3 Experiments Description

To evaluate the scalability and robustness of the two parallel implementations, three specific testing scenarios were designed. Each scenario isolates a specific variable (N, K, or D). All tests were executed as **strong scaling** benchmarks, varying the number of cores $P \in \{1, 2, 4, 8, 16, 32, 64\}$ while keeping the problem size fixed.

(i) **Scaling on N**: this scenario assesses the system's ability to handle large volumes of data, stressing memory management and the initial scatter operations.

- **Setup:** Fixed $K = 5$, $D = 6$.
- **Datasets:** $N \in \{200\,000, 300\,000, 500\,000\}$.
- **Objective:** to verify if increasing the dataset size results in a proportional (and manageable) increase in computational time and to determine the break-even point where the parallel speedup outweighs the communication overhead.

(ii) **Scaling on K**: this scenario increases the complexity of the statistical model. Increasing K linearly increases the number of probability density function (PDF) evaluations per point, but also increases the size of the messages exchanged during the reduction steps (weights, means, and covariances).

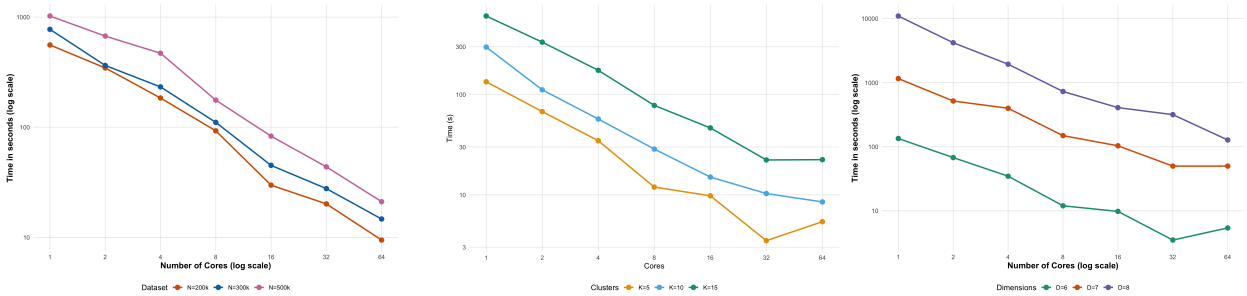
- **Setup:** Fixed $N = 50\,000$, $D = 6$.
- **Datasets:** $K \in \{5, 10, 15\}$.
- **Objective:** to evaluate the impact of global network communication (`MPI_Allreduce`) overhead, therefore assessing the difference between shared-memory and distributed-memory in managing bigger K. A higher K increases the computation that each process has to substain.

(iii) **Scaling on D**: this scenario represents the most computationally intensive tests. For instance, matrix operations (inversion and determinant calculation) have a computational complexity of $O(D^3)$ when using Gauss-Jordan elimination technique and LU decomposition technique(5)(7). Even a small increase in dimensions drastically increases the CPU workload.

- **Setup:** Fixed $N = 50\,000$, $K = 5$.
- **Datasets:** $D \in \{6, 7, 8\}$.
- **Objective:** to stress the CPU's floating-point units. Since this workload relies heavily on local matrix algebra with relatively little communication (small N, K), we expect near-linear speedup as the overhead is negligible compared to the calculation time.

6.4 Experiments Results (MPI)

Below are reported the results for the three cases experiments for MPI implementation. Figure 2(a) shows the impact of varying the dataset size, Figure 2(b) shows the impact of the varying number of clusters for fixed dataset size and dimensions, and Figure 2(c) shows the impact of the dimensionality of the datasets for constant N and K.



(a) Impact of N (b) Impact of K (c) Impact of D

Figure 2: Performance analysis: Dimensionality, Dataset Size, and Complexity.

- **Impact of Dataset Size** (Scenario i - Figure 2(a))

The execution time scaled linearly with N. Processing $500k$ points took approx. $1023s$ on 1 core, while $200k$ points took $558s$. A notable anomaly was observed at 16 cores for the $200k$ and $300k$ datasets, where the execution time dropped sharply (e.g., from $92s$ at 8 cores to $29s$ at 16 cores for $N = 200k$). This sudden jump suggests a transition where the local data chunk size fits into the CPU cache, drastically reducing memory access latency.

- **Impact of Model Complexity** (Scenario ii - Figure 2(b))

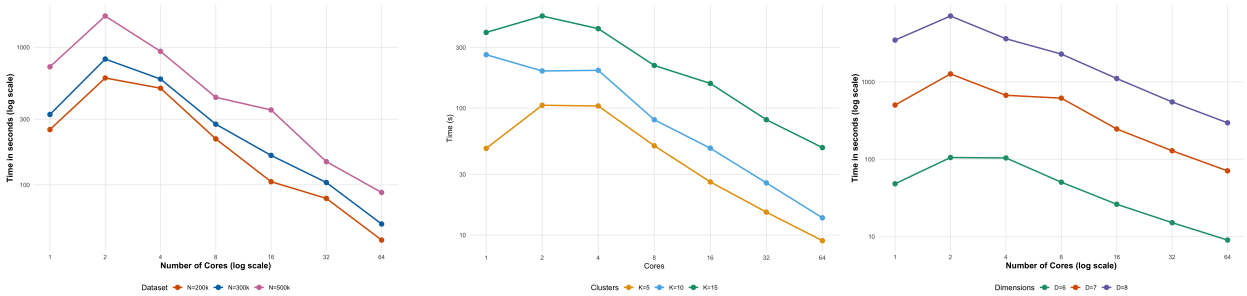
Scaling K from 5 to 15 resulted in a proportional increase in computational load. The single-core time rose from $134s$ ($K = 5$) to $608s$ ($K = 15$). While the message size for `MPI_Allreduce` triples with K, the network overhead remained negligible compared to the increased computational density of the E-step, therefore performance gains were observed across all test cases.

- **Impact of Dimensionality** (Scenario iii - Figure 2(c))

The dimensionality D proved to be the bottleneck for sequential performance. As shown in the results, iterating with $D = 8$ on a single core took approximately $10,986s$ (over 3 hours), compared to just $134s$ for $D = 6$. This massive increase confirms the cubic complexity $O(D^3)$ of the matrix operations involved. However, the distributed memory implementation effectively mitigated this: on 64 cores, the execution time for $D = 8$ dropped to $127s$, making the problem manageable.

6.5 Experiments Results (OpenMP)

Below are reported the results for the three cases experiments for OpenMP implementation. Figure 3(a) shows the impact of varying the dataset size, Figure 3(b) shows the impact of the varying number of clusters for fixed dataset size and dimensions, and Figure 3(c) shows the impact of the dimensionality of the datasets for constant N and K.



(a) Impact of N (b) Impact of K (c) Impact of D

Figure 3: Performance analysis: Dimensionality, Dataset Size, and Complexity.

(i) **Impact of Dataset Size** (Scenario i - Figure 3(a))
The strong scaling results across different dataset sizes (N=200,000, N=300,000, and N=500,000) demonstrate a consistent performance trend. For every setup, a notable anomaly occurs when moving from a single thread to two threads, where execution times significantly increase instead of reducing as expected. For example, the execution time for 500k points more than doubles, going from 721.74 seconds in sequential, to 1688.87 seconds with two threads. Effective parallel scaling only becomes evident, and approximately linear, at 8 threads, where the execution times finally drops below each single thread baseline. The system achieves its highest efficiency at 64 threads, reducing the computation time to 22.05 s for the 200k points, 38.14 s for 300k points, and 63.91 s for 500k points.

(ii) **Impact of Model Complexity** (Scenario ii - Figure 3(b))
The results observed for different cluster configurations (K=5,10,15) are consistent with the ones seen for different dataset sizes. In all cases, there is an initial counterintuitive drop in execution time, when going from one to four cores. This suggests that for a small number of threads, the effort to manage and synchronize threads is higher than the time saved by dividing the work. This overhead is more visible in the simplest case (K=5), since the total amount of work is overall smaller and the thread management effort becomes more significant. However, again, for 8 or more cores, the parallelization starts to effectively reduce the execution time, with the peak performance, for all configurations, reached at 64 cores.

(iii) **Impact of Dimensionality** (Scenario iii - Figure 3(c))
The plots for scaling over D follows the previous two behaviors. Moreover, as expected from the complexity of the algorithm, higher dimensions (D=8) take orders of magnitude longer than lower ones (D=6), but they also appear to benefit more consistently from higher core counts. This also proves that the dimensionality has the highest impact on the total execution time, with a significant jump from D=6 and D=8.

The sudden drop in performance observed for 2 and 4 threads in all cases was primarily attributed to the system architecture. As detailed in the System Description, the CPU utilized for these benchmarks is equipped with 4 physical sockets of 24 cores each and is NUMA based (Non-Uniform Memory Access). When using a limited number of threads, the operating system might place them on different physical sockets. This placement introduces significant communication overhead because each socket is directly interfaced with its own local RAM bank. Accessing data residing in a remote socket's memory requires traversing the interconnection between the two, leading to an increasing latency compared to the uniform access typical of single-socket systems. This was addressed setting `OMP_PLACES=sockets` and `OMP_PROC_BIND=close`. This way, OpenMP place threads as close as possible considering the sockets.

Nonetheless, these optimizations only provided negligible improvements to OpenMP performance, therefore they were not included in the final results. This suggest that the performance drop observed at 2 and 4 cores is not primarily due to thread placement but is instead attributable to other factors, most notably the synchronization overhead and the memory management costs associated with OpenMP directives, which are discussed in further detail in the following sections.

7 EVALUATION

Evaluating the performance of parallel implementations requires metrics that capture how efficiently additional computational resources are used. Therefore, scalability is expressed in terms of *Speedup* and *Efficiency*. These two metrics quantify the performance improvement achieved by parallelization in relation to the sequential execution.

Specifically the **Speedup** is defined as the ratio between the execution time of the sequential algorithm and that of the parallel algorithm. With p processors:

$$S(p) = \frac{T_1}{T_p}$$

where T_1 is the execution time using a single processor, and T_p is the time using p processors. Ideally, the speedup should coincide with the number of processors used (i.e. with 4 processors we should obtain a speedup value of 4). However, as stated by Amdahl's law (1), as more processes are added, the impact of the non-parallelizable portion becomes dominant, and scalability flattens or even reverses when overhead exceeds parallel gains. Therefore we expect the Speedup to worsen when using a high number of processes.

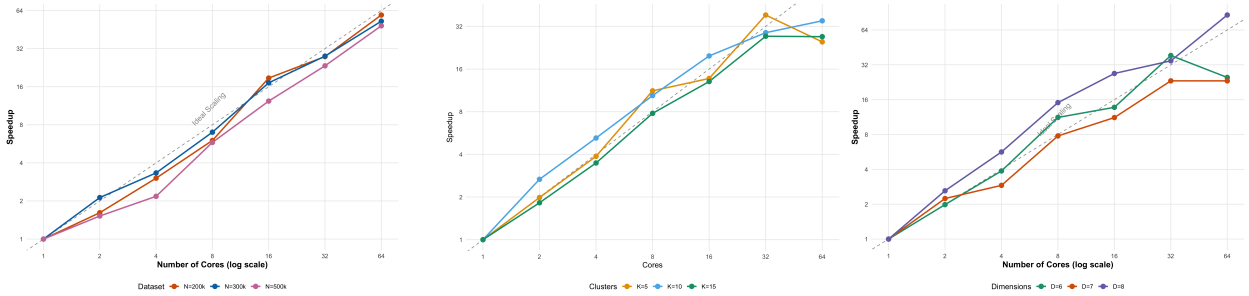
On the other hand, **Efficiency**, measures how effectively the computational resources are utilized, and is defined as the speedup normalized by the number of processors:

$$E(p) = \frac{S(p)}{p} \times 100\%$$

High Efficiency indicates that adding more processors yields a proportional reduction in runtime.

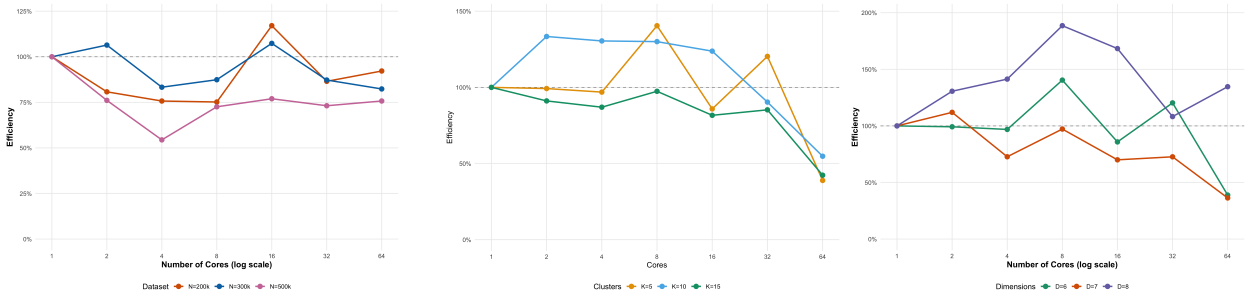
7.1 Scaling Analysis (MPI)

Below are reported both Speedup and Efficiency plots for the different configurations explored by the three scenarios for MPI implementation.



(a) Speedup (variable N) (b) Speedup (variable K) (c) Speedup (variable D)

Figure 4: Speedup for different configurations of N, K and D.



(a) Efficiency (variable N) (b) Efficiency (variable K) (c) Efficiency (variable D)

Figure 5: Efficiency for different configurations of N, K and D.

(i) **Scaling with N**

The implementation demonstrated super-linear scaling behavior for medium-sized datasets. For instance, with $N = 200k$ the efficiency went over 100% at 16 cores. This indicates that the super-linear speedup driven by cache locality is strong enough to completely mask the `MPI_Scatter` and `MPI_Gather` overheads. For the largest dataset ($N = 500k$), scalability was more consistent with theoretical expectations up to 4 cores, then improved significantly at 8 cores, maintaining high efficiency ($\approx 75\%$) even at 64 cores

(ii) **Scaling with K**

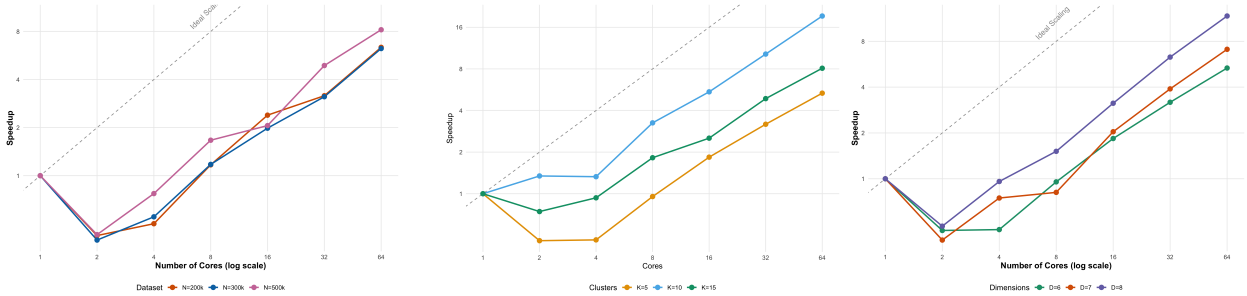
Efficiency frequently exceeded 100% (super-linear scaling) for $K = 5$ and $K = 10$. This anomaly occurs because smaller local partitions fit entirely into the CPU's cache, bypassing RAM latency. However, at 64 cores, efficiency consistently dropped below 50%, confirming that as computation time shrinks, the fixed MPI communication overhead inevitably dominates.

(iii) **Scaling with D**

This scenario exhibited the most impressive speedups. The master process is heavily bottlenecked by RAM latency and $O(D^3)$ math operations when running serially, whereas 64 processes handle the heavy matrix algebra entirely in fast cache, drastically lowering the execution time and resulting in efficiency values well above 100% for $D = 8$.

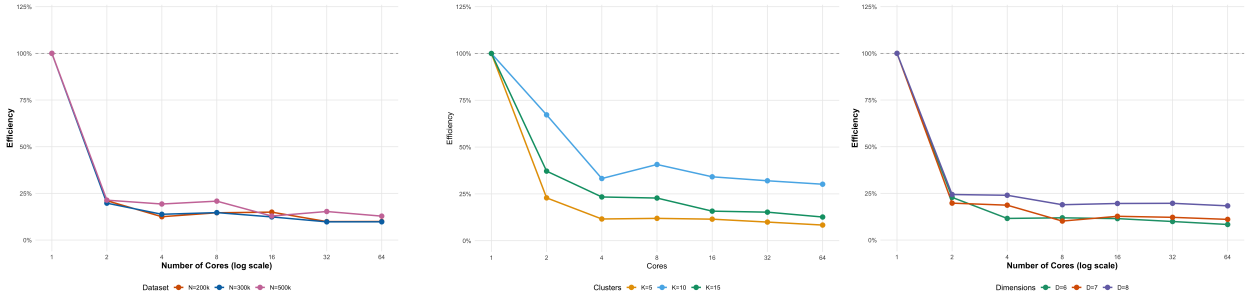
7.2 Scaling Analysis (OpenMP)

Below are reported both Speedup and Efficiency plots for the different configurations explored by the three scenarios for OpenMP implementation.



(a) Speedup (variable N) (b) Speedup (variable K) (c) Speedup (variable D)

Figure 6: Speedup for different configurations of N, K and D.



(a) Efficiency (variable N) (b) Efficiency (variable K) (c) Efficiency (variable D)

Figure 7: Efficiency for different configurations of N, K and D.

(i) **Scaling with N**

The analysis of scaling with respect to the dataset size N demonstrates a consistent trend where the speedup initially drops below the sequential implementation when moving to 2 cores, which is an indicator of high constant overhead. As the number of cores increases, the bigger dataset N=500k, consistently, though slightly, outperforms the smaller datasets, reaching a speedup of roughly 8× at 64 cores. However, the efficiency for all N values drops sharply and, although approximately constant, remains below 25% using multiple threads, suggesting that the full shared-memory approach is not the optimal solution for EM clustering.

(ii) **Scaling with K**

The scaling behavior with respect to the number of clusters K is what benefited the most from parallel execution. As shown in Figure 6(b), a significant jump occurs when moving from 1 to 2 cores, according to Figure 3(b). However, for K=10, the implementation achieves the highest speedup, reaching approximately 20× on 64 cores, which is confirmed by the efficiency plots, and suggests to be a trade-off between data complexity and overhead due to thread management.

(iii) **Scaling with D**

The scaling with respect to D shows again that only after 8 cores the speedup crosses the sequential threshold and begins to climb. While the scaling becomes linear at high core counts, it remains well below the Ideal Scaling line, achieving a maximum speedup of approximately 15x on 64 cores for D=8. This is also reflected in the Efficiency chart, which shows a dramatic drop from 100% (at 1 core) to below 25% as soon as parallelization is introduced. Interestingly, the efficiency remains relatively stable after this drop, suggesting that the overhead scales proportionally with the workload, with, slightly, better results for higher dimensionality (D=8).

8 RESULTS

The Experiments and Evaluation results highlighted a substantial performance and scalability difference between the two parallel architectures that were developed in this project. The MPI (distributed-memory) approach showed excellent scalability and, overall, super-linear speedup. This phenomena is particularly noticeable for increasing dimensionality (D) and number of clusters (K), where the efficiency exceeded the 100% thanks to the better data locality in the cache of the single cores, which helped mitigating RAM latency and the cubic complexity of the matrix operations.

On the contrary, the OpenMP (shared-memory) implementation demonstrated several limitations, especially for low thread count. Precisely, an anomaly occurred in every experiment performed when going from the sequential baseline to the parallel run with 2-4 threads, where execution times even doubled. This was explained by the high synchronization overhead required to manage the reduction clauses which were necessary to avoid concurrent updates of shared variables. Specifically, the reductions create copies of shared variables that allows each thread to work on its own local variable before merging the results. This additional allocation and final synchronization, introduce overhead that, for low number of threads, outweighs the benefits of the parallel work. Finally, although OpenMP reached its higher efficiency at 64 threads, the overall efficiency remains often below 25% for scalability over N, suggesting that the shared-memory model is less convenient than the distributed-memory for EM clustering algorithm for GMM.

Additionally, from the three experiments performed varying N, K and D, what stressed the most the performance was increasing the dimensionality of the dataset D. This could be explained by the nature of the EM algorithm which has a high complexity over D, due to the computationally demanding matrix operations, such as matrix inverse calculation. This could be addressed using specific libraries for linear algebra (e.g. OpenBLAS (12)), which could possibly improve the results.

9 CONCLUSIONS

This project analyzed in details the performance of the Expectation-Maximization (EM) algorithm for Gaussian Mixture Models (GMM) benchmarking two different parallel techniques. The first approach employed Open Multi-Processing (OpenMP) framework, while the second leveraged Message Passing Interface (MPI), allowing for assessing the different gains in using shared or distributed memory architectures.

From the results obtained, MPI implementation clearly stands as most indicate parallelization for EM algorithm for GMM, especially with increasing dataset size. Its workload distribution and local cache management, have demonstrated excellent scalability, which reached super-linear speedup levels. This demonstrates that for high-complexity algorithm, the distributed-memory approach is superior in mitigating RAM bottlenecks of many cores demanding access at the same time.

However, OpenMP results highlighted the criticality of using a shared-memory approach for algorithms that require frequent synchronizations and have strong computational dependencies. The high overhead encountered for a low number of threads suggests that the synchronization cost and memory contention between cores significantly penalize the performance. Thus, OpenMP solution benefits could be seen when using a high number of cores or large-scale datasets.

In conclusion, the Expectation-Maximization algorithm for Gaussian Mixture Models gained optimal performance from the distributed-memory parallelization through MPI, proving to be the most effective choice for handling large-scale datasets with algorithms that requires frequent updates, overcoming the structural limitations of shared-memory architectures.

REFERENCES

[1] Gene Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, pages 483–485, 1967.

[2] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, 2007.

[3] James R. Bunch and John E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[4] Wei-Chen Chen, George Ostrouchov, David Pugmire, Prabhat, and Michael Wehner. A parallel em algorithm for model-based clustering applied to the exploration of large spatio-temporal data. *Technometrics*, 55(4):513–523, 2013.

[5] Peter Danziger. Complexity. Lecture Notes, Toronto Metropolitan University, n.d. Department of Mathematics.

[6] Martina De Piccoli and Jago Revrenna. Parallel expectation-maximization clustering for gmm. GitHub repository, 2026.

[7] Robert L. Jack. Gaussian elimination / LU decomposition. Lecture Notes, University of Cambridge, n.d. Department of Applied Mathematics and Theoretical Physics (DAMTP).

[8] Naveen Kumar et al. A parallel architecture for the expectation maximization algorithm on iti/o processors. In *2009 IEEE International Conference on Multimedia and Expo*, pages 1118–1121. IEEE, 2009.

[9] Jing Li et al. Gpu-based parallel expectation maximization algorithm for gaussian mixture model. *Journal of Parallel and Distributed Computing*, 70(7):700–711, 2010.

[10] P.D. McNicholas, T.B. Murphy, A.F. McDaid, and D. Frost. Serial and parallel implementations of model-based clustering via parsimonious gaussian mixture models. *Computational Statistics Data Analysis*, 54(3):711–723, 2010. Second Special Issue on Statistical Algorithms and Software.

[11] Robert D Nowak. Distributed em algorithms for density estimation and clustering in sensor networks. *IEEE Transactions on Signal Processing*, 51(8):2245–2253, 2003.

[12] OpenBLAS Development Team. *OpenBLAS: An optimized BLAS library based on GotoBLAS2*, 2024. Available at `https://www.openblas.net/`.

[13] Claudia Plant and Christian Böhm. Parallel em-clustering: Fast convergence by asynchronous model updates. In *2010 IEEE International Conference on Data Mining Workshops*, pages 178–185, 2010.

[14] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 1st edition, 2004.

[15] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, 1st edition, 2004.

[16] Kumar V.P. and Gupta A. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22(3):345–360, 1995.

[17] Jishang Wei, Hongfeng Yu, Jacqueline H Chen, and Kwan-Liu Ma. Parallel clustering for visualizing large scientific line data. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 47–55. IEEE, 2011.

[18] Wikipedia contributors. Expectation–maximization algorithm — Wikipedia, the free encyclopedia, 2024. [Online; accessed 08-January-2026].

Table 2. Strong Scaling by N ($K = 5$, $D = 6$)

Cores	$N = 200000$			$N = 300000$			$N = 500000$		
	Time (s)	Speedup	Eff. (%)	Time (s)	Speedup	Eff. (%)	Time (s)	Speedup	Eff. (%)
MPI									
1	558.22	1.00	100.0	774.33	1.00	100.0	1023.03	1.00	100.0
2	345.55	1.62	80.8	363.86	2.13	106.4	672.41	1.52	76.1
4	184.31	3.03	75.7	232.38	3.33	83.3	470.07	2.18	54.4
8	92.81	6.01	75.2	110.74	6.99	87.4	176.26	5.80	72.5
16	29.81	18.73	117.0	45.08	17.18	107.4	83.07	12.31	77.0
32	20.16	27.69	86.5	27.74	27.92	87.2	43.72	23.40	73.1
64	9.46	59.01	92.2	14.69	52.71	82.4	21.11	48.46	75.7
OpenMP									
1	252.29	1.00	100.0	324.66	1.00	100.0	721.74	1.00	100.0
2	598.26	0.42	21.1	821.56	0.40	19.8	1688.87	0.43	21.4
4	504.30	0.50	12.5	587.55	0.55	13.8	934.25	0.77	19.3
8	215.72	1.17	14.6	275.96	1.18	14.7	432.78	1.67	20.9
16	105.42	2.39	15.0	163.68	1.98	12.4	350.48	2.06	12.9
32	79.65	3.17	9.9	103.97	3.12	9.8	147.46	4.90	15.3
64	39.65	6.36	9.9	51.90	6.26	9.8	88.00	8.20	12.8

Table 3. Strong Scaling by K ($N = 50,000$, $D = 6$)

Cores	$K = 5$			$K = 10$			$K = 15$		
	Time (s)	Speedup	Eff. (%)	Time (s)	Speedup	Eff. (%)	Time (s)	Speedup	Eff. (%)
MPI									
1	134.36	1.00	100.0	297.91	1.00	100.0	608.12	1.00	100.0
2	67.70	1.98	99.2	111.66	2.67	133.4	333.75	1.82	91.1
4	34.67	3.88	96.9	57.08	5.22	130.5	174.79	3.48	87.0
8	11.96	11.24	140.4	28.64	10.40	130.0	78.03	7.79	97.4
16	9.78	13.74	85.9	15.04	19.81	123.8	46.55	13.07	81.7
32	3.49	38.51	120.3	10.31	28.91	90.3	22.29	27.28	85.3
64	5.40	24.89	38.9	8.50	35.06	54.8	22.44	27.10	42.3
OpenMP									
1	48.16	1.00	100.0	263.56	1.00	100.0	394.88	1.00	100.0
2	105.30	0.46	22.9	196.06	1.34	67.2	531.44	0.74	37.2
4	103.91	0.46	11.6	198.47	1.33	33.2	422.71	0.93	23.4
8	50.48	0.95	11.9	80.95	3.26	40.7	216.76	1.82	22.8
16	26.19	1.84	11.5	48.28	5.46	34.1	156.49	2.52	15.8
32	15.14	3.18	9.9	25.74	10.24	32.0	80.99	4.88	15.2
64	9.02	5.34	8.3	13.65	19.31	30.2	48.82	8.09	12.6

Table 4. Strong Scaling by D ($N = 50,000$, $K = 5$)

Cores	$D = 6$ Time (s)	$D = 6$ Speedup	$D = 6$ Eff. (%)	$D = 7$ Time (s)	$D = 7$ Speedup	$D = 7$ Eff. (%)	$D = 8$ Time (s)	$D = 8$ Speedup	$D = 8$ Eff. (%)
MPI									
1	134.36	1.00	100.0	1158.80	1.00	100.0	10986.99	1.00	100.0
2	67.70	1.98	99.2	517.42	2.24	112.0	4205.30	2.61	130.6
4	34.67	3.88	96.9	398.33	2.91	72.7	1943.17	5.66	141.4
8	11.96	11.24	140.4	149.00	7.78	97.2	728.33	15.08	188.6
16	9.78	13.74	85.9	103.44	11.20	70.0	407.92	26.93	168.3
32	3.49	38.51	120.3	49.82	23.26	72.7	317.25	34.64	108.3
64	5.40	24.89	38.9	49.82	23.26	36.3	127.50	86.17	134.6
OpenMP									
1	48.16	1.00	100.0	501.48	1.00	100.0	3466.55	1.00	100.0
2	105.30	0.46	22.9	1267.73	0.40	19.8	7103.88	0.49	24.4
4	103.91	0.46	11.6	671.09	0.75	18.7	3612.46	0.96	24.0
8	50.48	0.95	11.9	616.03	0.81	10.2	2289.10	1.51	18.9
16	26.19	1.84	11.5	246.15	2.04	12.7	1105.78	3.14	19.6
32	15.14	3.18	9.9	128.71	3.90	12.2	550.23	6.30	19.7
64	9.02	5.34	8.3	70.74	7.09	11.1	295.45	11.73	18.3