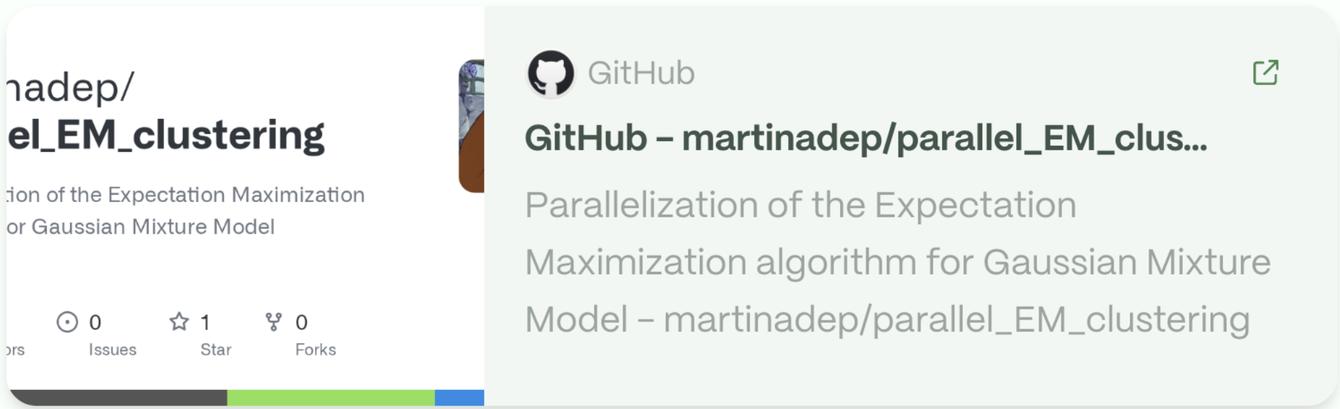
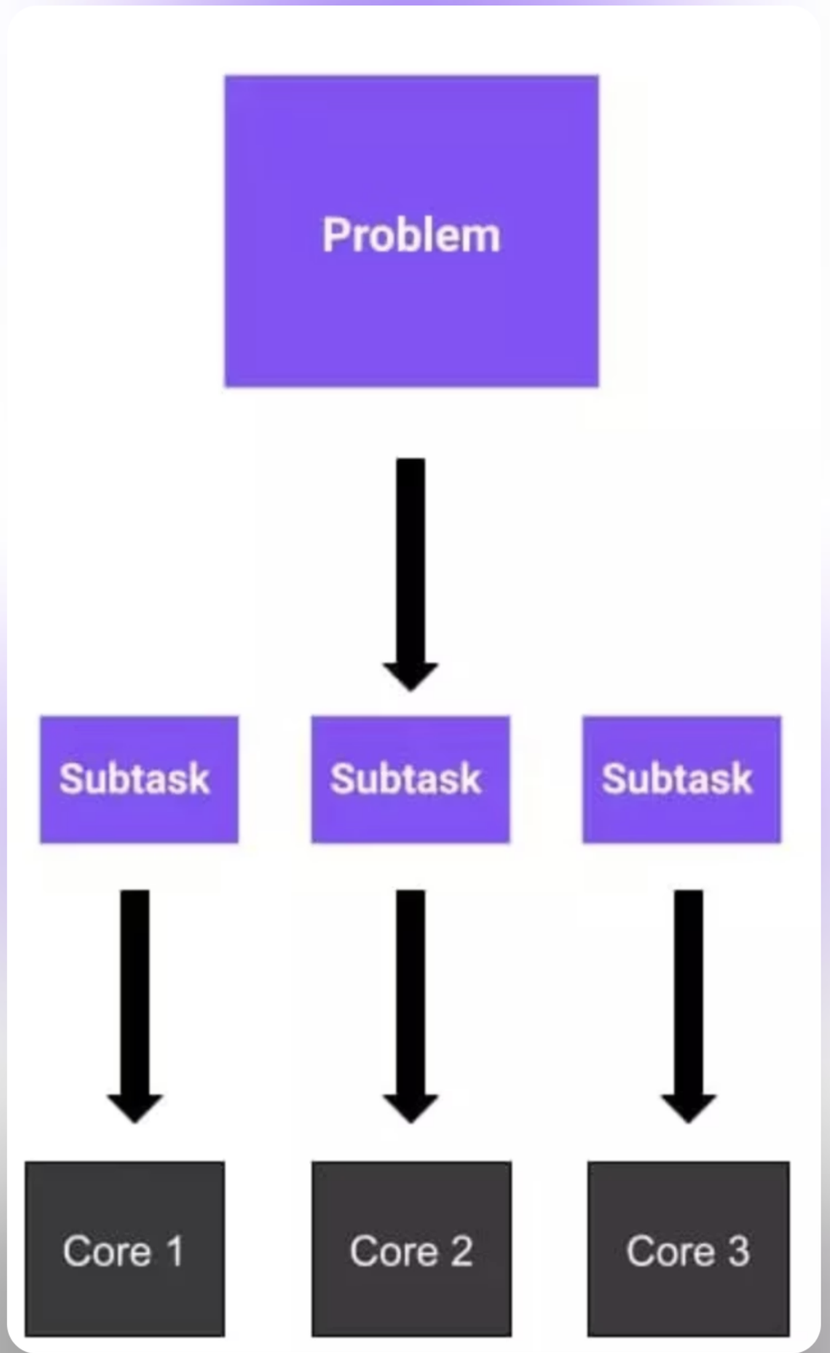
Parallel Expectation-Maximization for GMM

Martina De Piccoli & Jago Revrenna

University of Trento, High Performance Computing for Data Science





Parallelizing EM for GMM

This project explores **parallelization of the Expectation-Maximization (EM) algorithm** for Multivariate Gaussian Mixture Models (GMM) using two approaches:

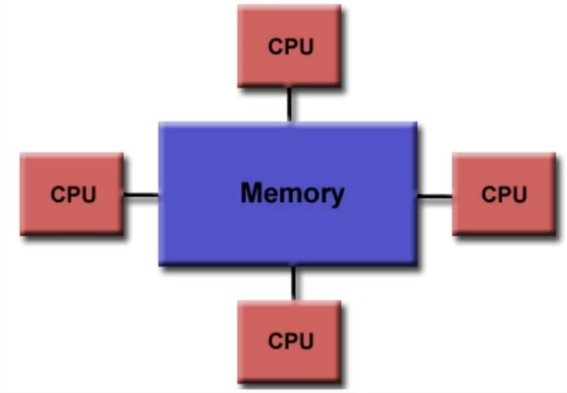
Message Passing Interface (MPI)

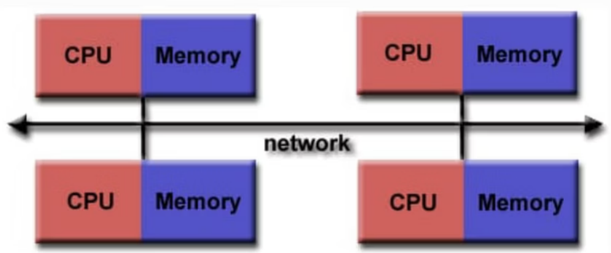
Distributed memory technique for full-MPI parallelization.

OpenMP

Shared-memory technique for full-OpenMP parallelization.

The study aims to compare efficiency, scalability, and computational performance.





Introduction: The Need for Parallelization

Clustering groups data points based on patterns.

When data distribution shape is known but parameters are not, EM helps learn these parameters.

1

E-step

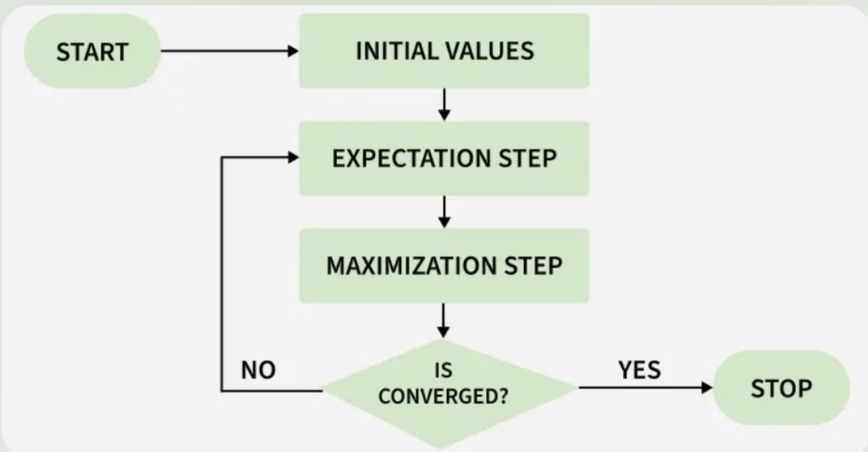
Calculate probability of data points belonging to specific Gaussians based on current parameters.

2

M-step

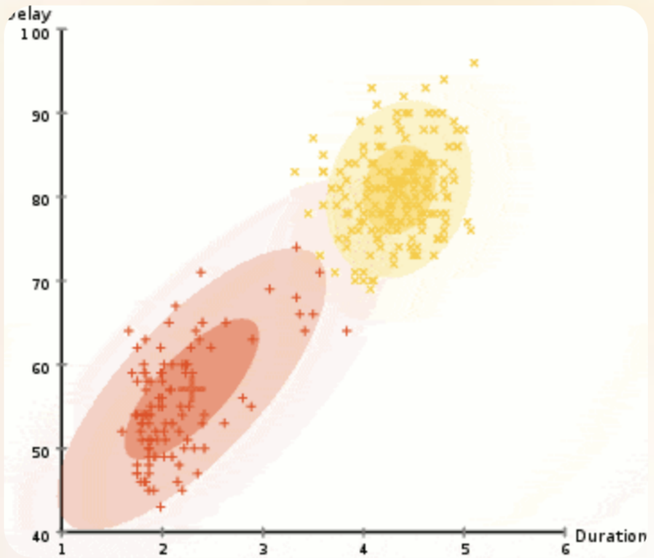
Update Gaussian means, covariances, and mixing weights to maximize data likelihood.

EM is computationally intensive for large datasets, making **HPC crucial for reducing execution time**.



State of the Art

Master-slave architectures were used, but suffered from communication overhead, while modern solutions favor **SPMD (Single Program, Multiple Data)** with collective communications.



Expectation-Gathering-Maximization

Adds a gathering step to combine statistics from parallel nodes.

Asynchronous EM

Allows asynchronous model updates in the M-phase, with threads independently updating models.

GPU/Hybrid Systems

Exploits GPU parallel units for impressive speedup, especially with increasing dimensionality.

EM Clustering for GMM: Mathematical Foundation

GMM assumes data points are generated from a mixture of Gaussian distributions with unknown parameters.

The probability of a d-dimension observation is

$$p(x_n|\theta) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x_n|\mu_k, \Sigma_k)$$

The EM algorithm aims to **find the Maximum Likelihood Estimate (MLE)** of these parameters

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

–> intractable, EM start by random guesses and iterates to eventually find the optimal solution.

E-step and M-step

E-step: Responsibilities

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

Computes the posterior probability that point x_n belongs to cluster k.

M-step: Parameter Update

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_{nk} x_n$$

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

$$\pi_k^{new} = \frac{N_k}{N}$$

Updates parameters by maximizing the expected log-likelihood using responsibilities.

These steps repeat until convergence is reached.

Sequential EM Implementation

EM convergence and quality are sensitive to initial conditions.

A `init_gmm` function was developed for robust initialization

- **Means (μ_k)**

  Initialized using `K-means++` seeding heuristic for widespread initial clusters.

- **Covariance Matrices (Σ_k)**

  All clusters initialized with global covariance to prevent narrow or overly spread Gaussians.

- **Mixing Weights (π_k)**

  Uniformly initialized as $\pi_k = 1/K$ due to no prior knowledge.

- **Handling of multivariate Gaussian PDF**

  Using LUP decomposition instead of the canonical co-factor methods for matrix inversion.

The `em_algorithm` function iteratively calls `e_step` and `m_step` until numerical convergence, checked by `log_likelihood` within an `EPSILON` threshold.

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

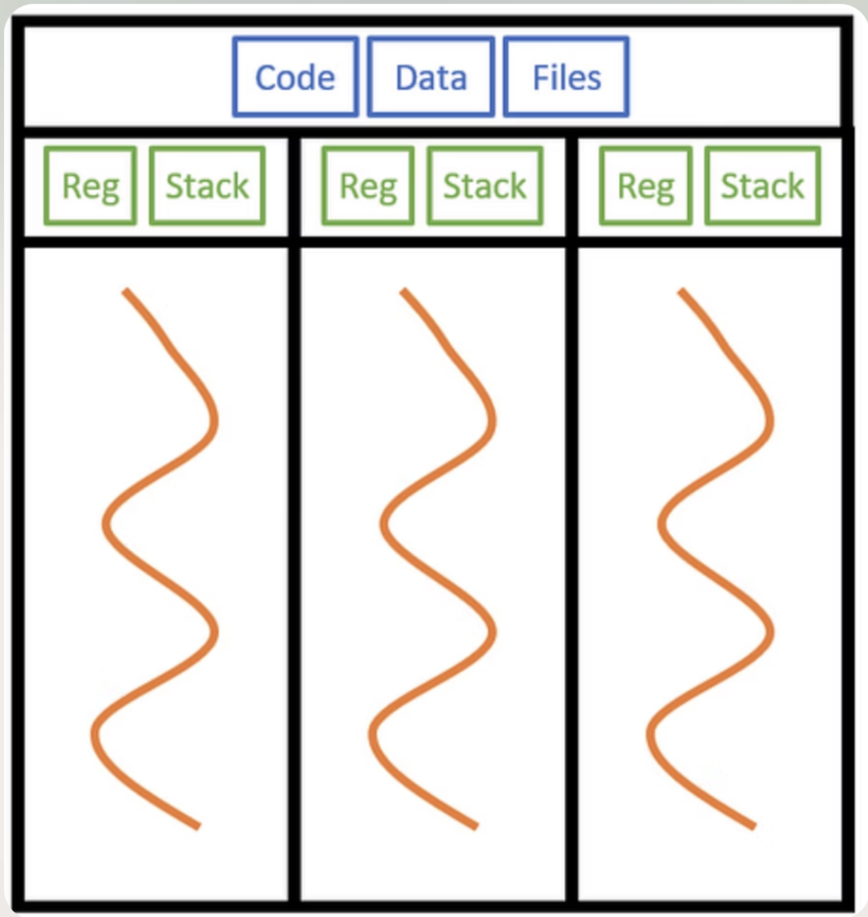
Data Dependencies & Parallel Design

Initialization	**No dependencies** between clusters (parameters are initialized independently)
Main Loop	**Sequential dependency** between iterations (each iteration depends on the parameters calculated in the previous state)
E-Step: Responsibility	**No dependencies** between data points, allowing massive parallelization over `N`.
E-Step: Accumulation	**Reduction dependency**, since responsibilities of all points `N` must be accumulated into shared variables.
M-Step: Parameters	**Reduction dependency** for (updates to) weighted means and covariances, calculated across `N` points.
Convergence Check	**Reduction dependency** for global log-likelihood, **sequential dependency** on log-likelihood of previous iteration.

Parallel design focused on **data decomposition**, partitioning `N` data points across processing units for balanced workloads.

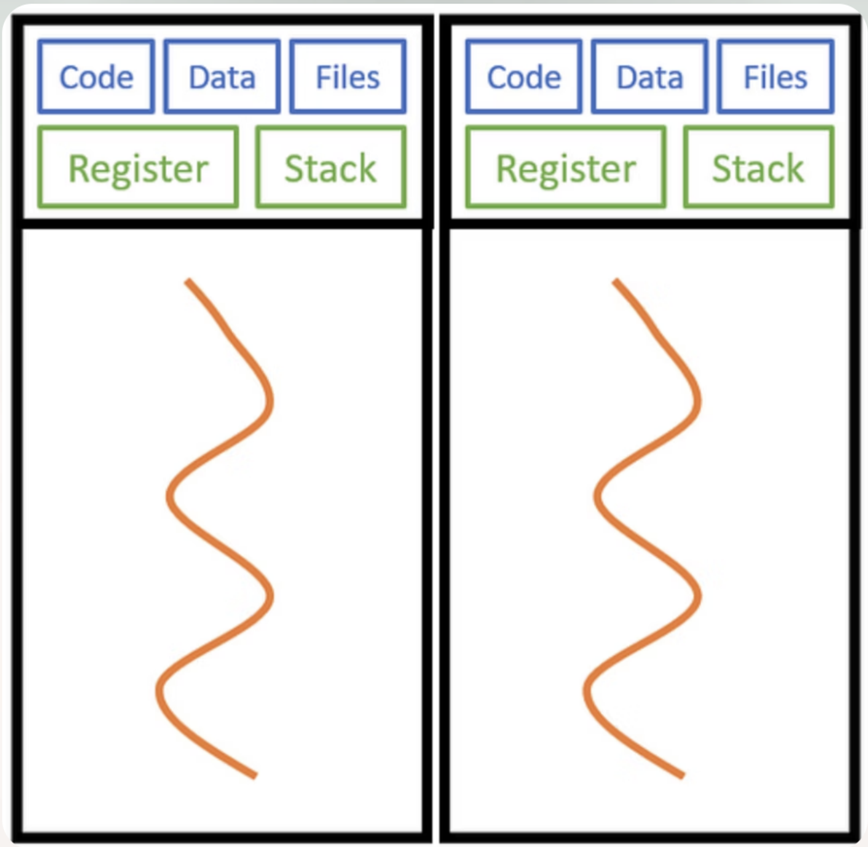
Parallel implementation: OpenMP (Shared-Memory)

1. `#pragma omp parallel for` is used for concurrent responsibility calculation in E-step
2. `reduction` clause handles accumulation, minimizing synchronization overhead.
3. in M-step, a `reduction` clause manages the accumulation of means and covariance matrices



Parallel implementation: MPI (distributed memory)

1. Dataset partitioned among processes using `MPI_Scatter`, while problem dimensions (`N`, `D`, `K`, initials GMM parameters) are broadcasted to all processes using `MPI_Bcast`.
2. E-step runs locally (responsibilities depend only on local data points and the current global parameters).
3. M-step uses `MPI_Allreduce` for global aggregation of statistics (with `MPI_SUM` operator), ensuring all processes get updated parameters simultaneously.
4. `MPI_Gather` collects final labels in the master process.



System

Benchmarks were conducted on a Linux system with an x86-64 architecture, configured as follows:

- 96 physical CPUs, distributed across 4 sockets (24 cores each)
- Intel(R) Xeon(R) Gold 6252N CPU
- 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 following compilers and libraries were used:

- OpenMP implementation was compiled using `gcc 9.1.0`
- MPI implementation was compiled using `OpenMPI 4.0.4` and `gcc 9.1.0`

Experimental Scenarios

Datasets were generated in R, **strong scalability** was chosen since it's difficult to define an uniform workload unit. Complexity is $O(NKD^3I)$.

1 **Scaling on N - dataset size**

K = 5, D = 6, while N = 200k, 300k, 500k.

Large datasets.

2 **Scaling on K - model complexity**

N = 50k, D = 6, while K = 5, 10, 15.

More PDF evaluations per point (more communication).

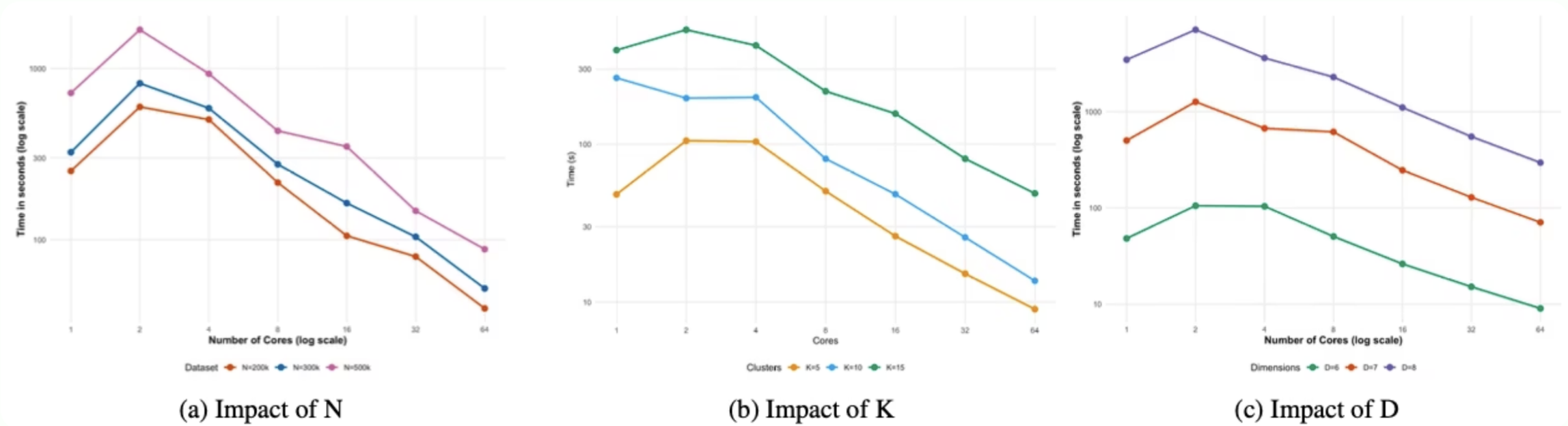
3 **Scaling on D - dimensionality**

N=50k, K=5, while D = 6, 7, 8.

$O(D^3)$ matrix operations

Experiments results (OMP)

Time



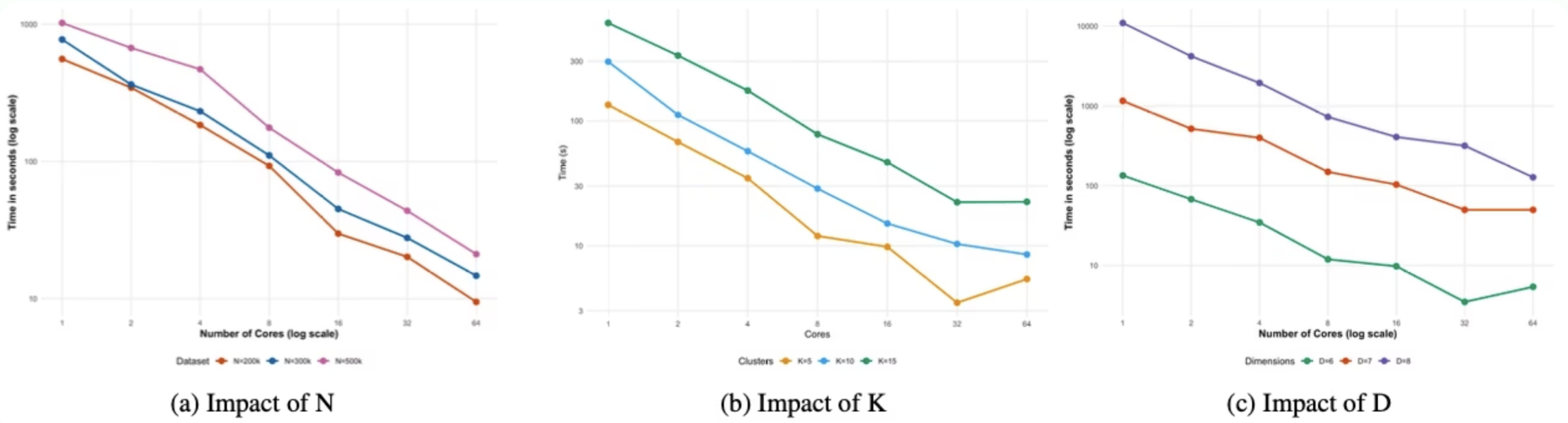
(a) Impact of N (b) Impact of K (c) Impact of D

Effective parallel scaling only becomes evident, and approximately linear, at 8 threads.

Tried solution -> `OMP PLACES=sockets`, `OMP PROC BIND=close`

Experiments results (MPI)

Time



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

In K and D, at 64 cores there's no improvement (`MPI_Allreduce`)

Evaluation strategies

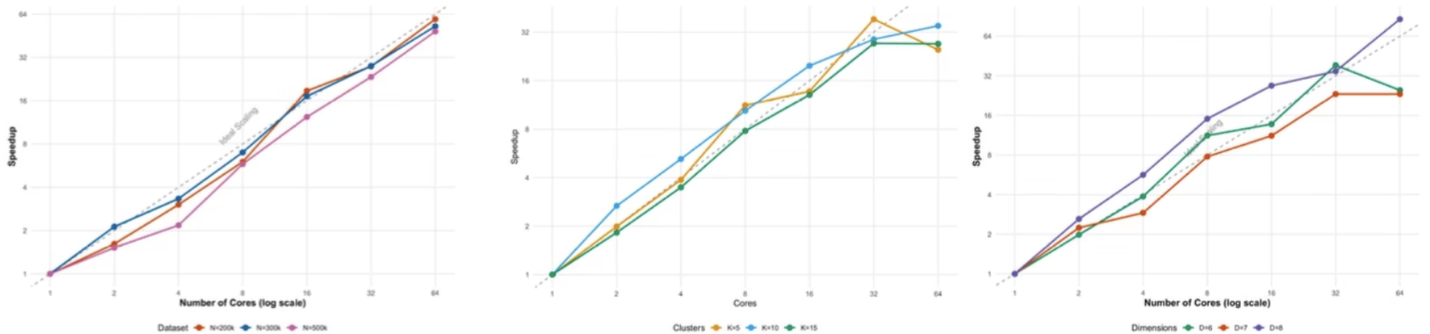
Speedup is the ratio between the execution time of the sequential algorithm and that of the parallel algorithm

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

Efficiency is how effectively the computational resources are utilized

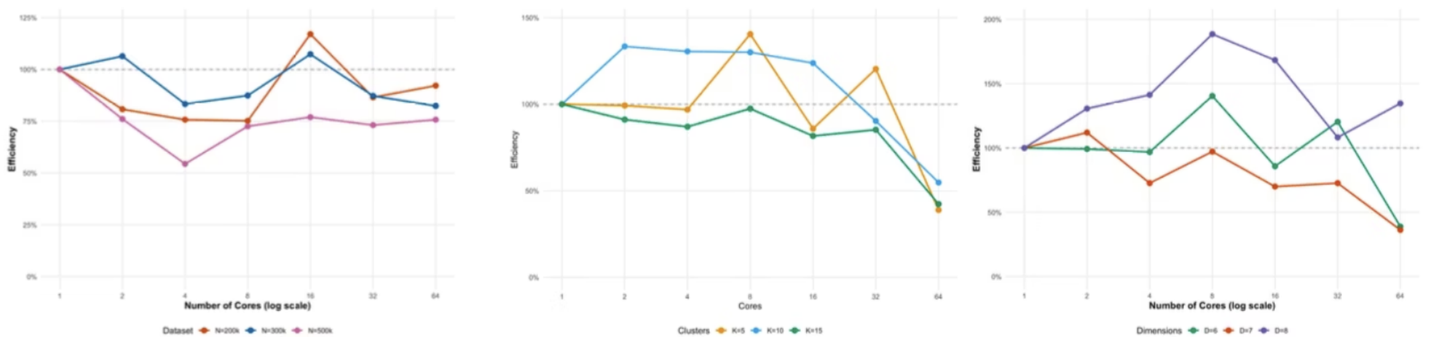
$$E(p) = \frac{S(p)}{p} \times 100\%$$

Scaling analysis (MPI)



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

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

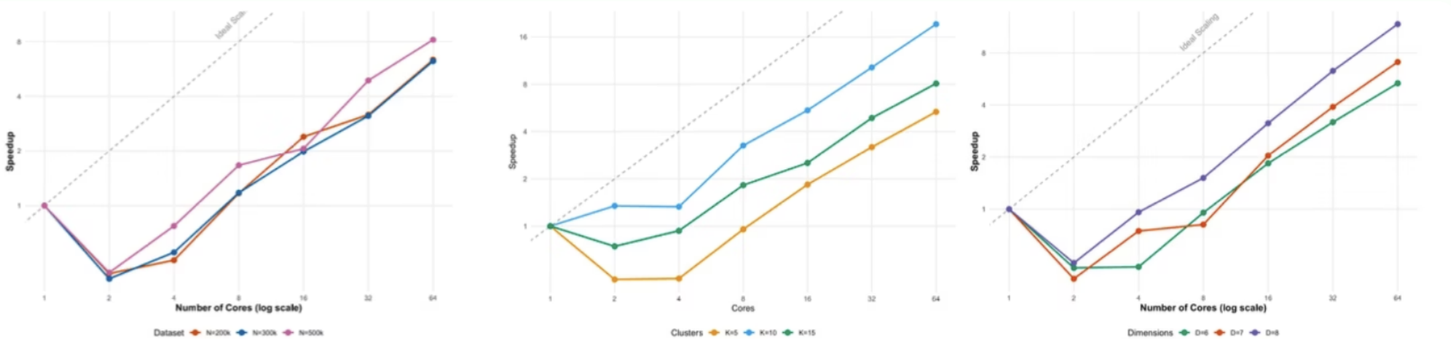


(a) Efficiency (variable N) (b) Efficiency (variable K) (c) Efficiency (variable D)

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

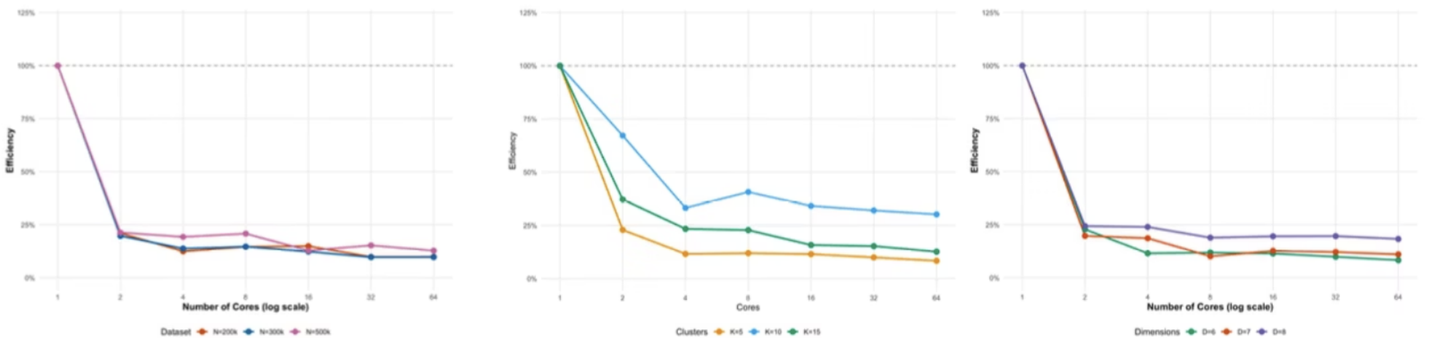
Super-linear speedup driven by cache locality is strong enough to completely mask the MPI Scatter and MPI Gather overheads.

Scaling analysis (OpenMP)



(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.

Important drop, indicator of high constant overhead. Full shared-memory approach is not the optimal solution for EM clustering.

MPI vs. OpenMP: Performance Summary

The experimental results revealed significant performance and scalability differences between the two parallel approaches for the EM algorithm.

MPI (Distributed)

Excellent scalability

Super-linear speedup

OpenMP (Shared)

Synchronization overheads

Lower, though constant, efficiency

Increasing dataset dimensionality (`D`) proved to be the most performance-stressing factor due to complex matrix operations.

Further improvements could be achieved by integrating optimized linear algebra libraries (like `OpenBLAS`).