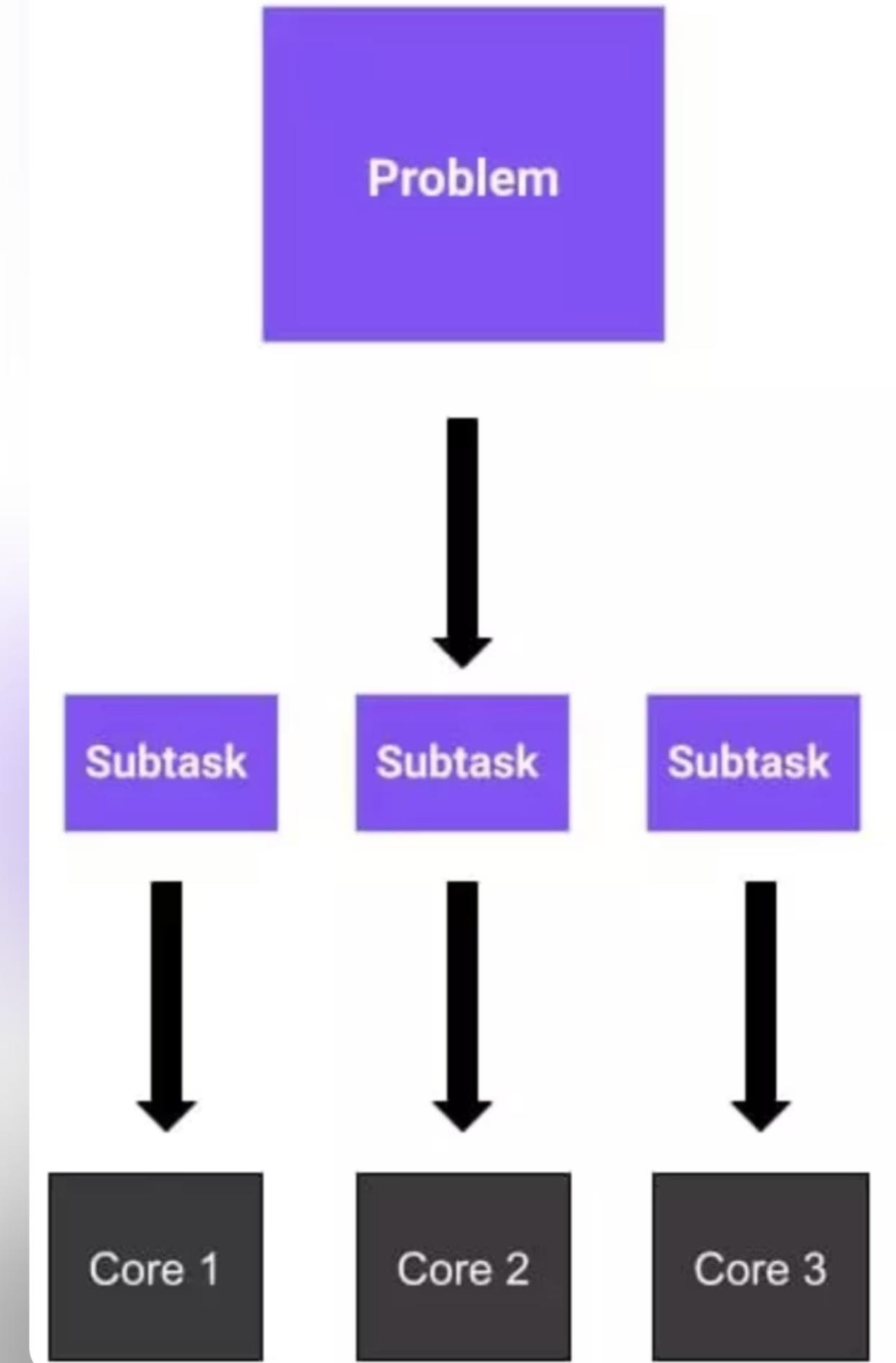
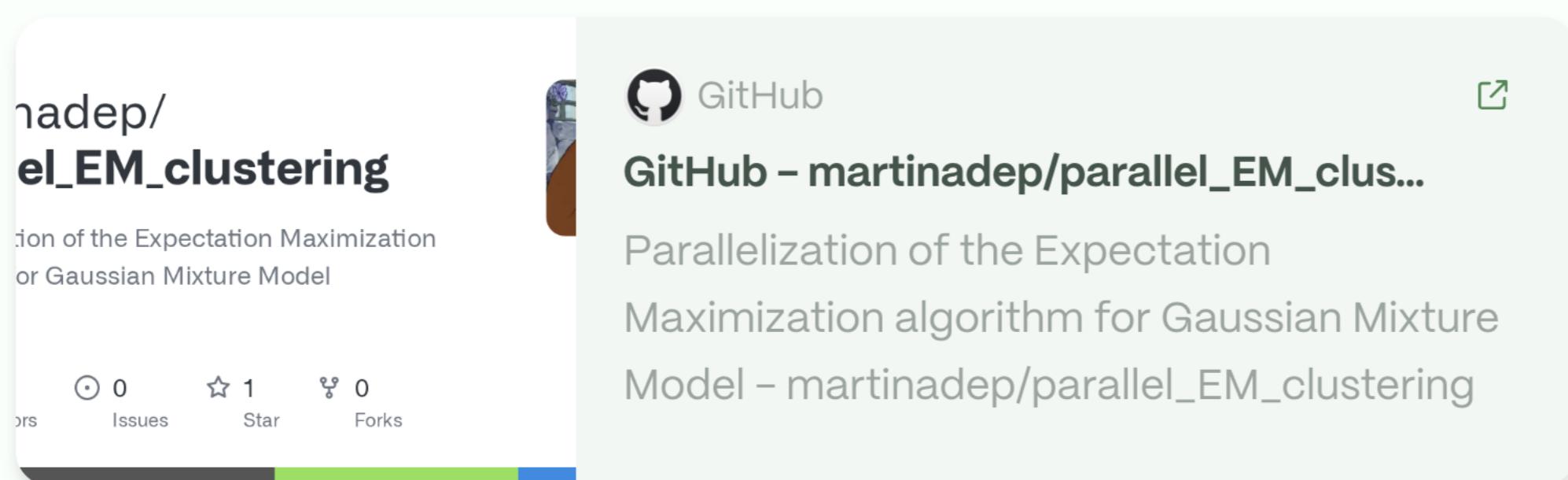


Parallel Expectation-Maximization for GMM

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:

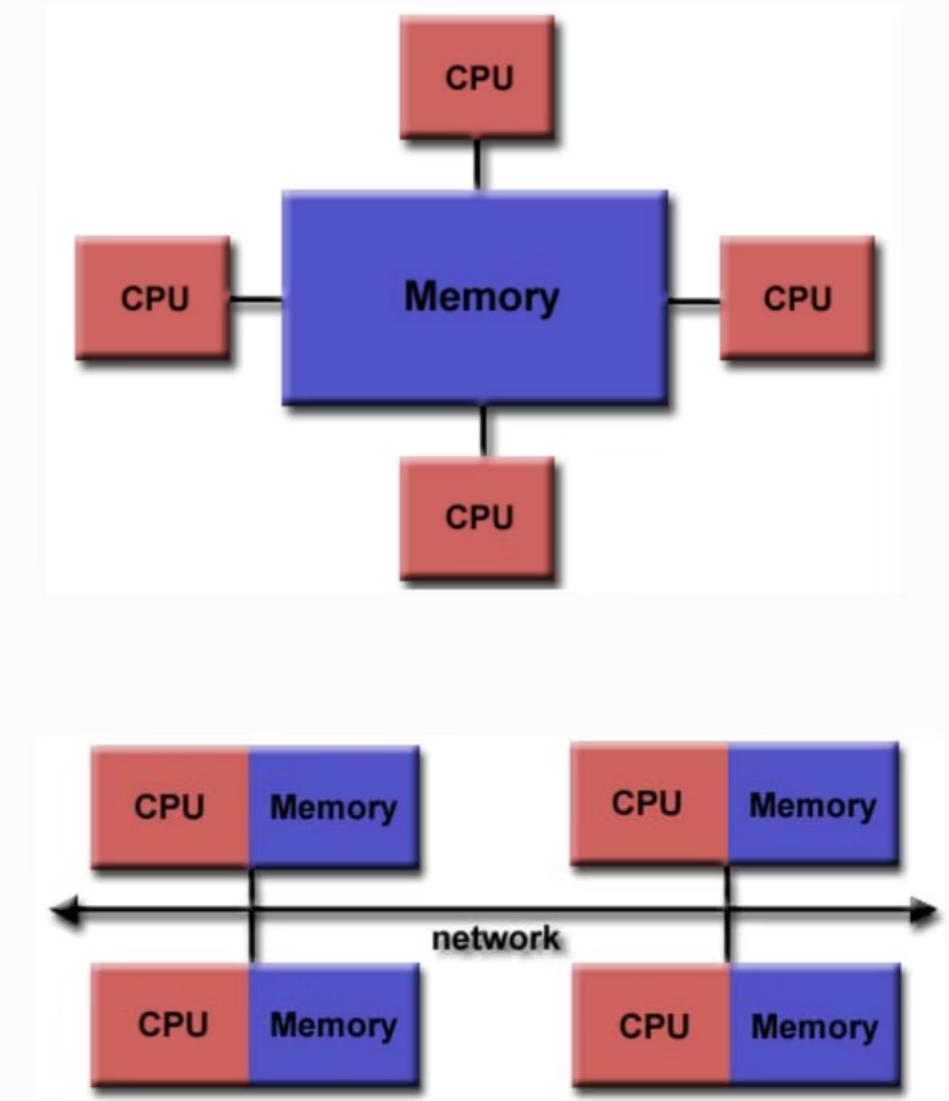
Message Passing Interface (MPI)

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.



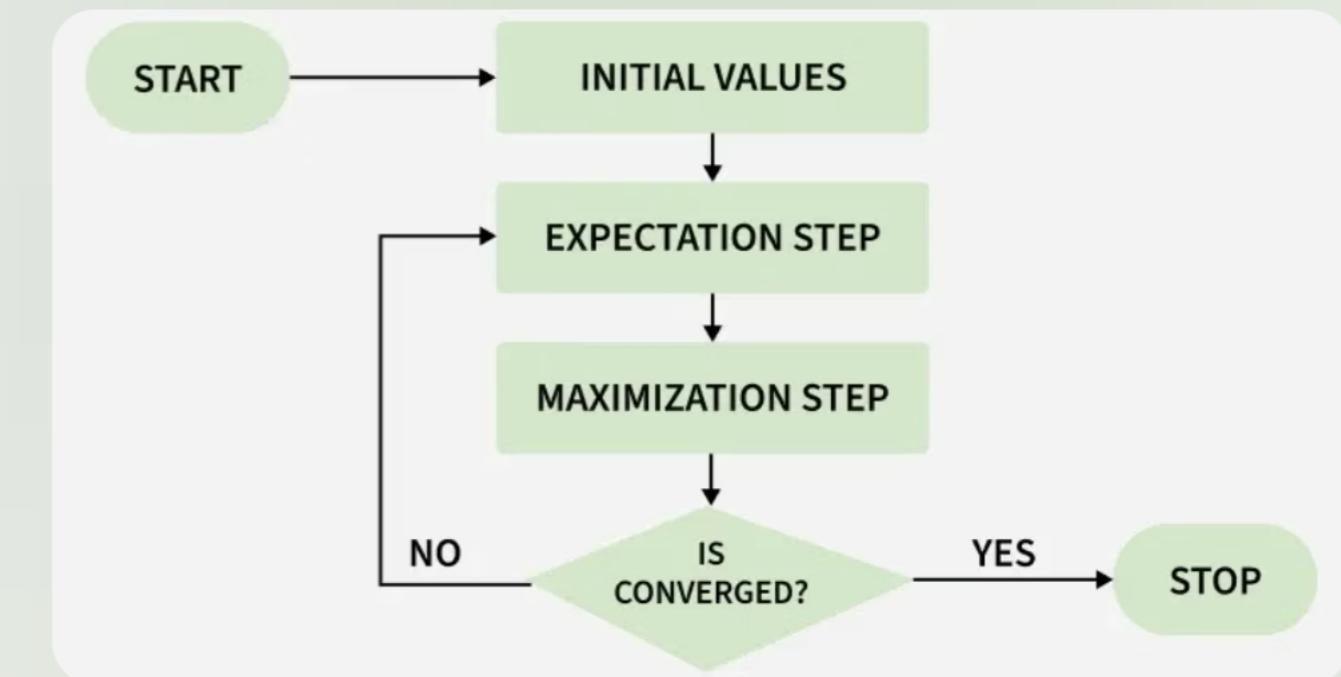
E-step

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

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

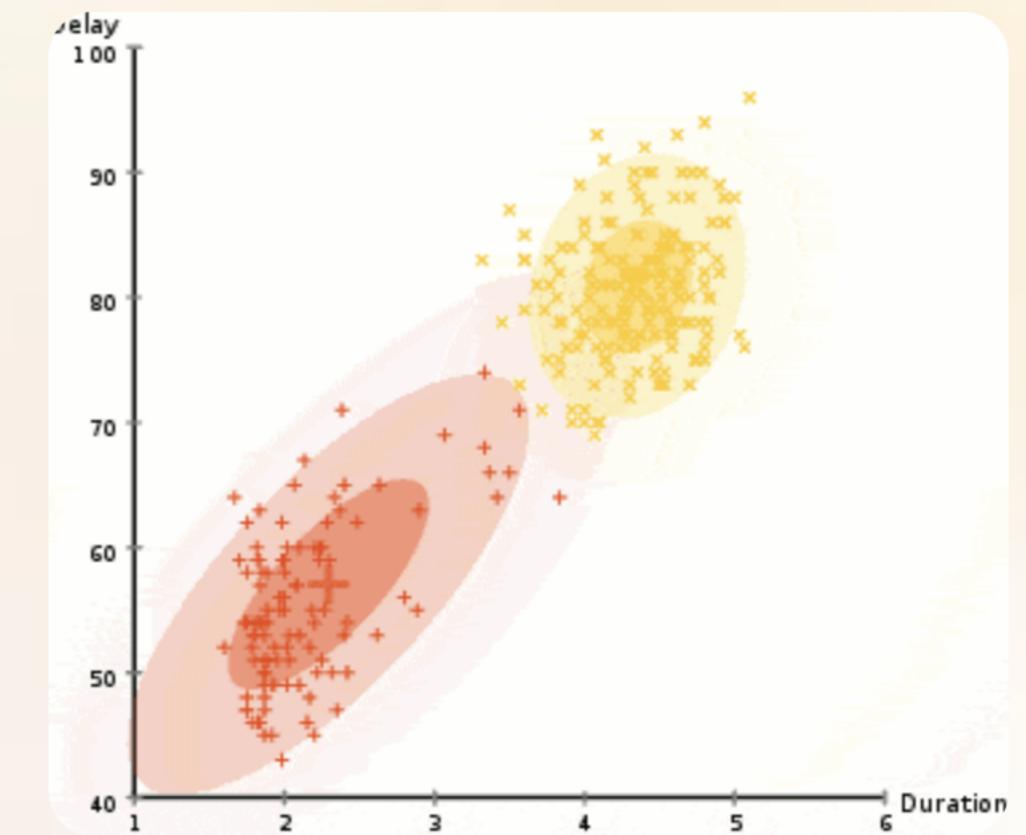
Adds a gathering step to combine statistics from parallel nodes.

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

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

-> 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

```
1: Initialize:  $\pi_k, \mu_k, \Sigma_k$  using init_gmm
2:  $L_{prev} \leftarrow -\infty$ 
3: for  $iter = 0$  to  $MAX\_ITER$  do
4:   {E-Step}
5:   for all point  $n \in \{1 \dots N\}$  do
6:      $norm \leftarrow 0$ 
7:     for all cluster  $k \in \{1 \dots K\}$  do
8:        $resp[n][k] \leftarrow \pi_k \cdot \text{multiv_gaussian_pdf}(x_n, gmm[k])$ 
9:        $norm \leftarrow norm + resp[n][k]$ 
10:    end for
11:    Normalize  $resp[n]$  by  $norm$ 
12:    Accumulate  $gmm[k].class\_resp$  for M-step
13:  end for
14:  {M-Step}
15:  for all cluster  $k \in \{1 \dots K\}$  do
16:     $\pi_k \leftarrow gmm[k].class\_resp/N$ 
17:     $\mu_k \leftarrow \text{weighted average of } x_n \text{ using } resp[n][k]$ 
18:     $\Sigma_k \leftarrow \text{weighted covariance of } (x_n - \mu_k)$ 
19:  end for
20:   $L_{curr} \leftarrow \text{log\_likelihood}(X, \theta)$ 
21:  if  $|L_{curr} - L_{prev}| < \epsilon$  then
22:    break
23:  end if
24:   $L_{prev} \leftarrow L_{curr}$ 
25: end for
```

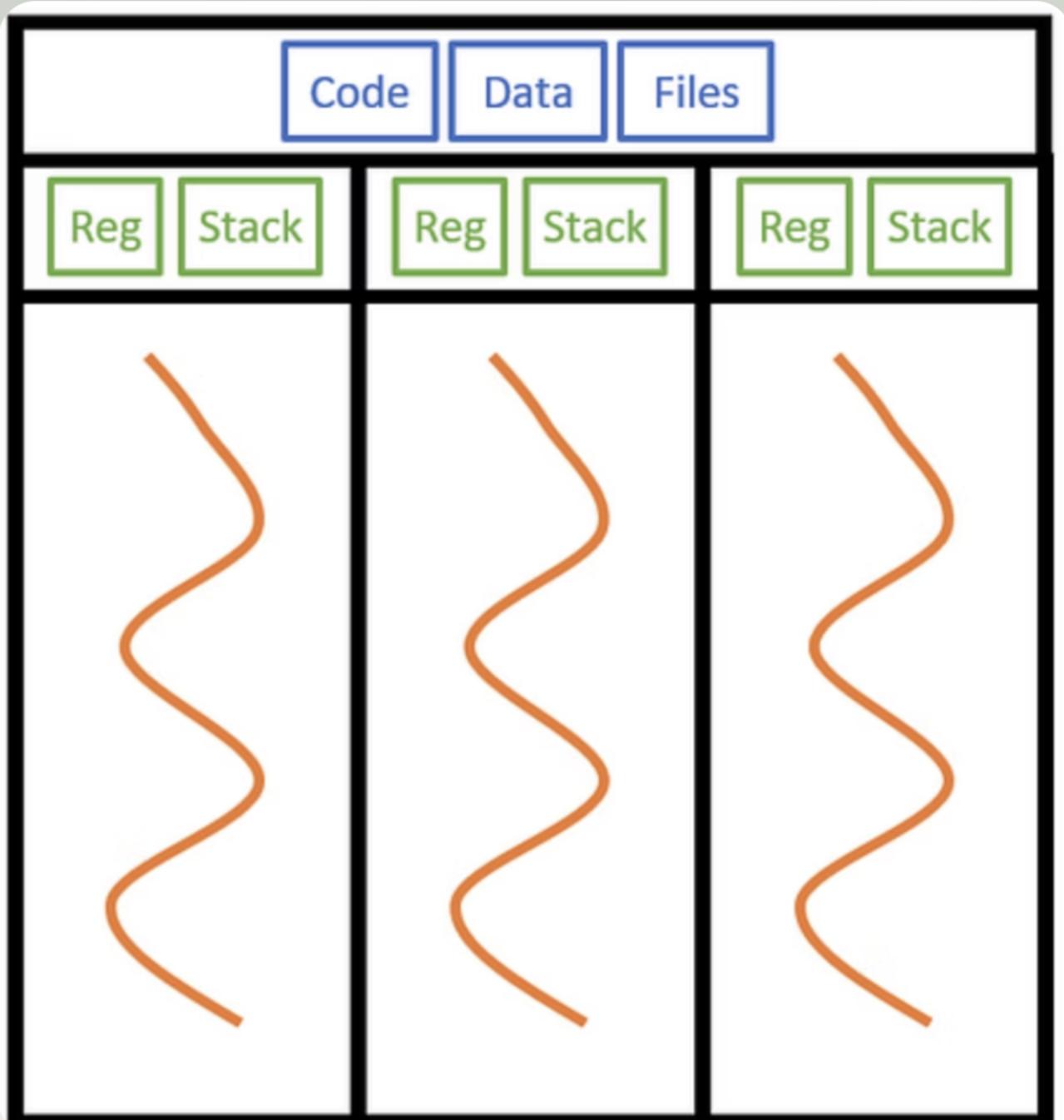
Data Dependencies & Parallel Design

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.

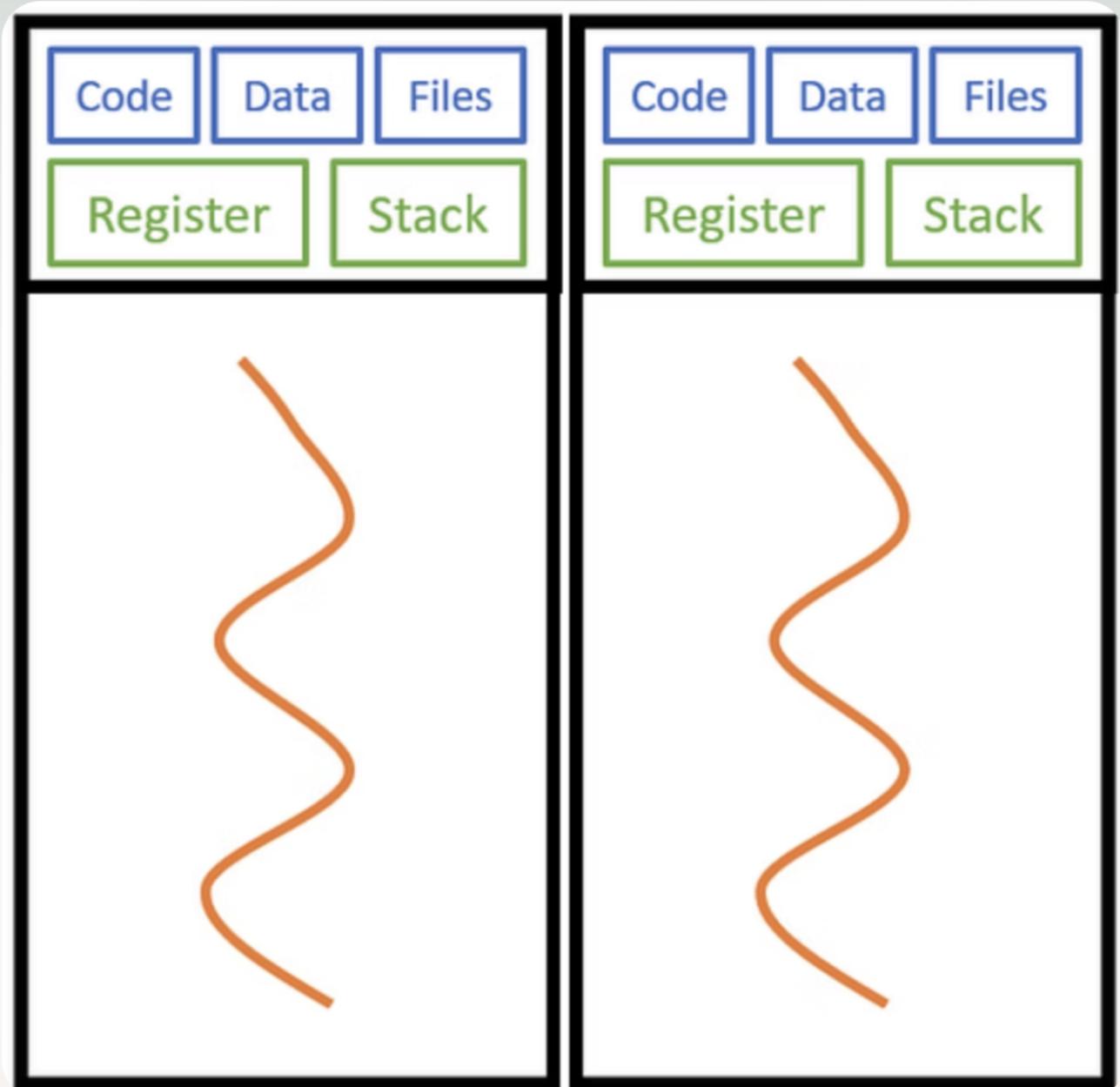
Parallel implementation: OpenMP (Shared-Memory)

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.



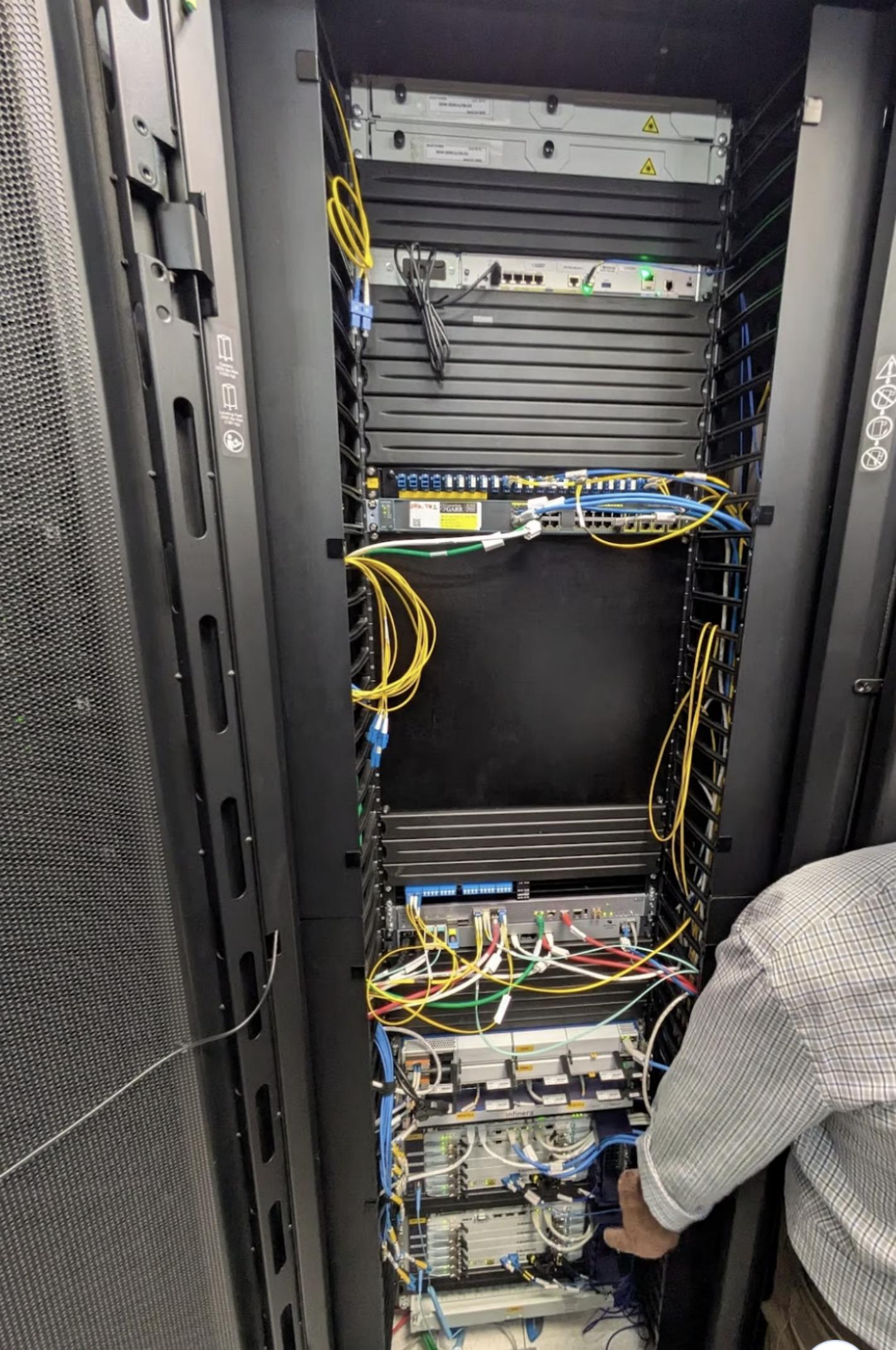
System

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`



, "x2", "x3", "x4", "x5", "x6"
565830542493, 4.03105630077893, 5.61364922200066, -6.510671142007
013526285398, 9.74426797524299, 4.96048423618217, -8.643706912340
336878919896, 9.43123120693424, 2.5015762257427, -10.054907938192
022753105646, 9.81550788397365, 0.422829544148017, -6.96970718995
6852762109064, 13.5833670610212, 13.8086590410441, 1.321286073621
773849802289, 7.89753802672502, 0.169682892805282, -11.1807065700
1700684983187, 12.6899626014923, 16.4784872558005, 0.281540161420
1500285680302, 10.6819082613141, -11.4115308926082, -10.192751481
752712526139, 5.54750201028713, 0.646961968546941, -4.00505665247
0619069484429, 12.9326141476519, 12.0454212415446, 0.854613116296
993835909789, 7.74303546356378, 5.68345361104377, -6.324446036510
8608108590697, 8.8510701500942, -15.2181567301509, -8.01044787967
394615068866, 8.14263820171191, 4.29638858510205, -9.736022393676
889903929024, 7.25577804671097, 1.79836965215191, 0.8971858798358
649469751, 12.180778917585, 2.04723636438465, 5.93142867616359, 14
06354786913, 8.75970768104134, 1.21297191043338, -8.9063785416717
472404077811, -14.1309320756099, 16.5709126528163, -8.86753111945
646599233324, 7.43660359575809, 2.49375184833421, -8.928482966109
.838862898642, 10.2804947229057, 3.08951401550205, -7.509777321349
962057632036, -13.2708359640483, 15.948702287042, -11.17158040676
1633195619737, 10.9644467892179, -10.7112560574878, -11.487173750
766074037528, 7.7932670760715, 10.4796226398148, -1.0148519798560
9876418618776, 11.216992782337, 13.8832127649583, 1.8722372392523
3364127077622, 9.5067741111909, -12.0159244535928, -11.1287168053
0114072652608, 13.2799957867566, 14.912289708193, 0.4811422270845
770849356188, 2.482479365646, 3.66582890677639, -2.61064807205153
163699232154, 8.29113597247296, 6.37876907850742, 6.9935064170807
1661053709317, 8.82365693591928, -10.0460397805222, -7.4864739884
7470030044659, 10.3513534973201, 4.65520688089913, 7.081260163659
681482746364, -12.6144638656774, 16.693480077777, -12.56929816523
645765060877, 7.5026241219785, 4.94909753570465, -6.5896203577350
576815963154, -11.3045854360402, 16.2260112969341, -12.6238286013
001832161621, -12.5280985329182, 15.2200476582092, -12.8847435183
76840632912, -7.96027359242799, -1.44948740110614, 10.76330479922
569717576274, 9.69915059984765, -0.296347899426021, -10.709471501
3238939262539, 5.50484451206369, 7.55615034842064, -1.87239567672

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

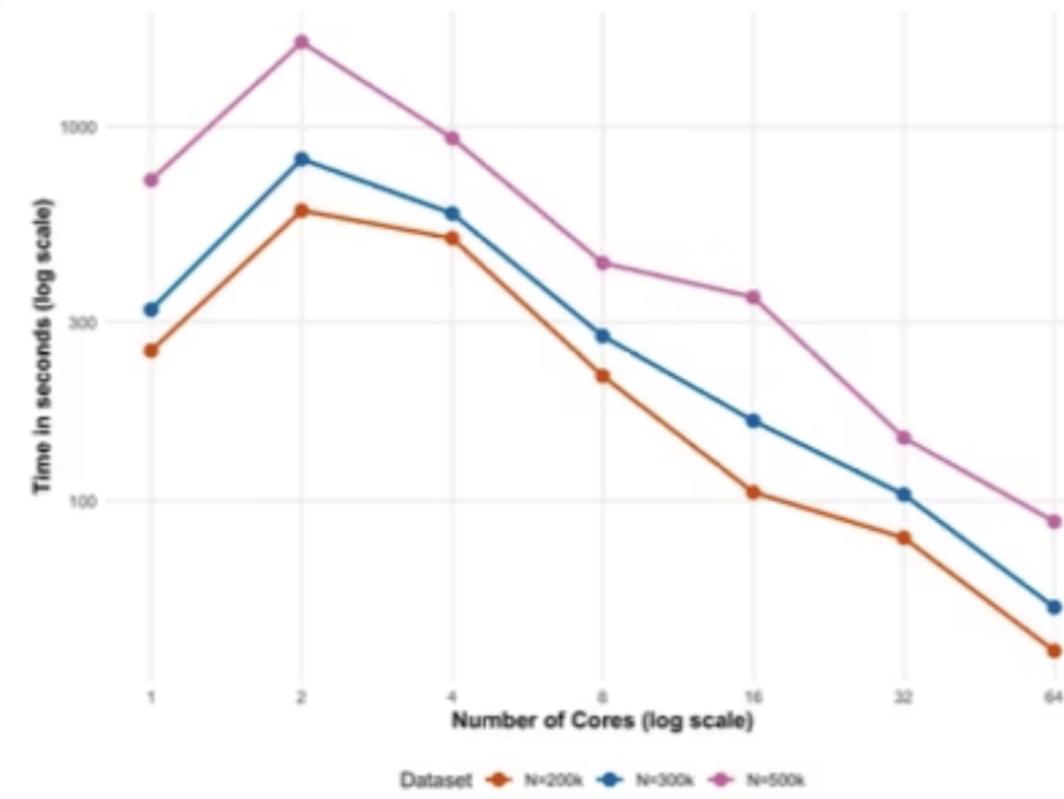
Scaling on D - dimensionality

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

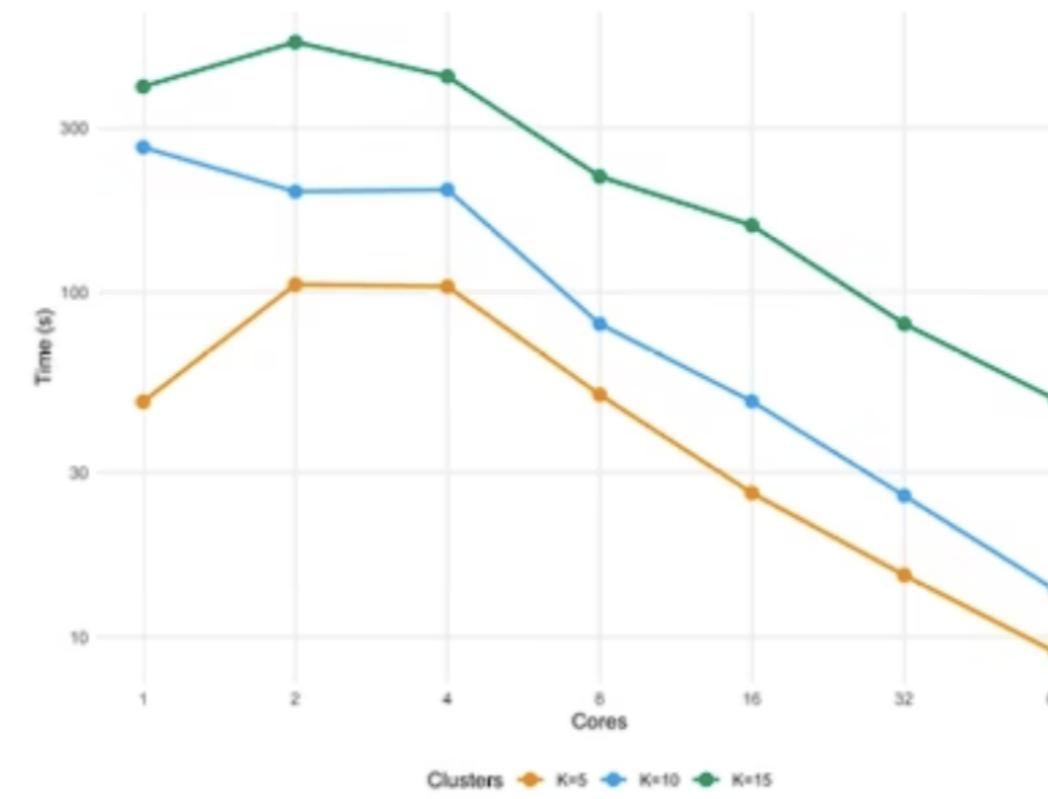
$O(D^3)$ matrix operations

Experiments results (OMP)

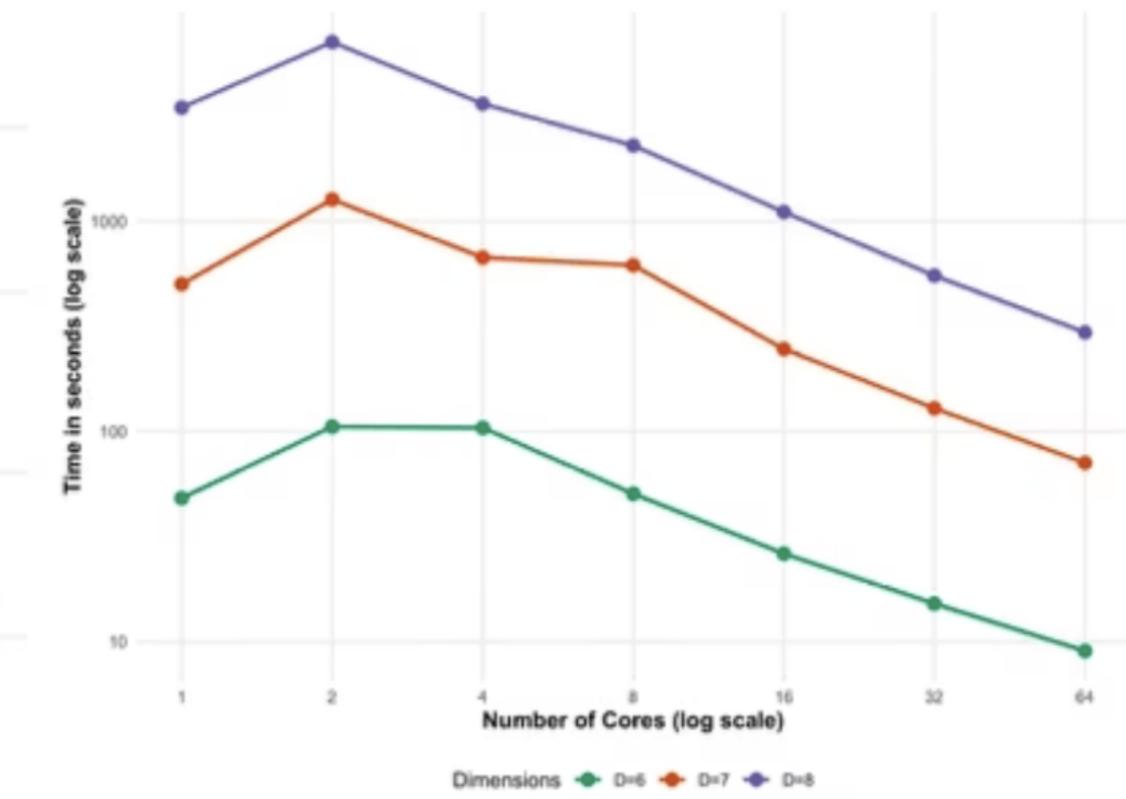
Time



(a) Impact of N



(b) Impact of K



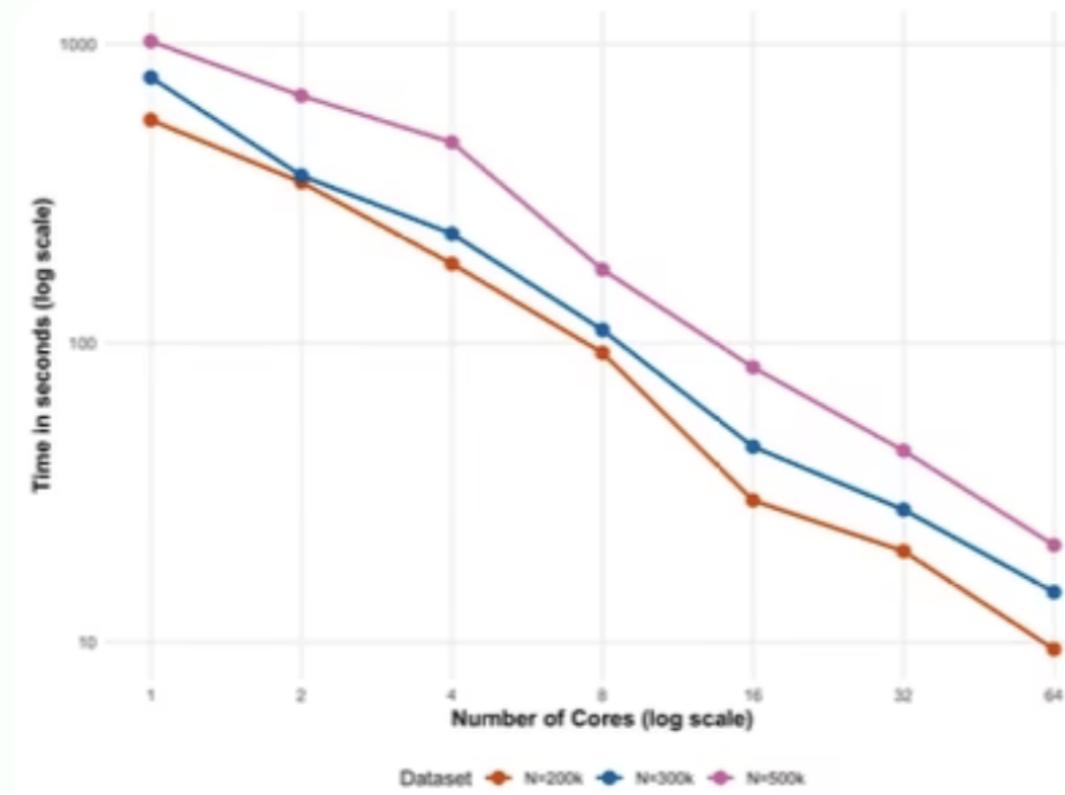
(c) Impact of D

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

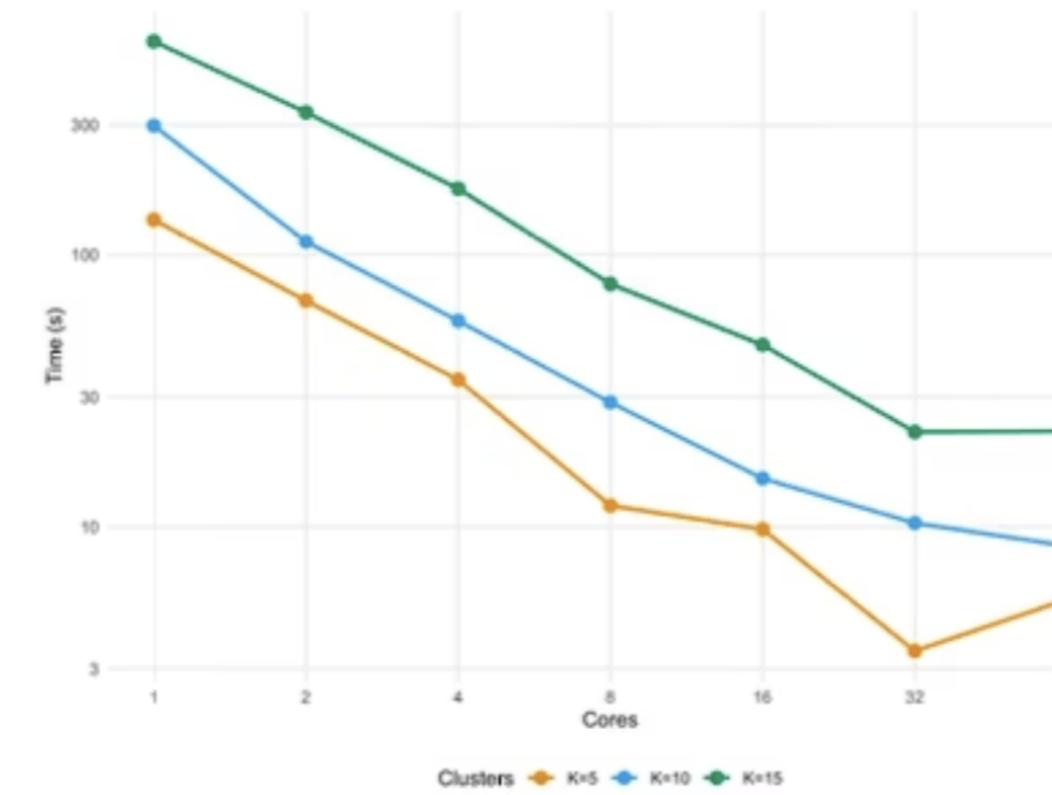
Tried solution → `OMP_PLACES=sockets, OMP_PROC_BIND=close`

Experiments results (MPI)

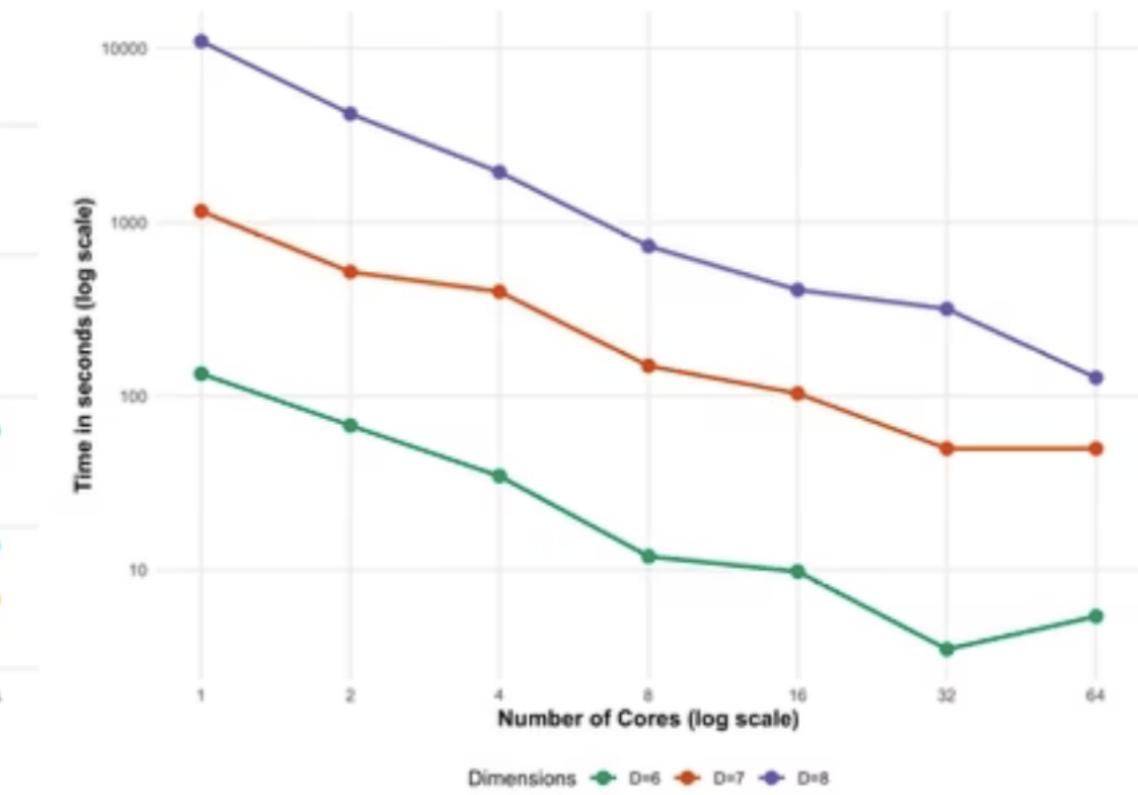
Time



(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)

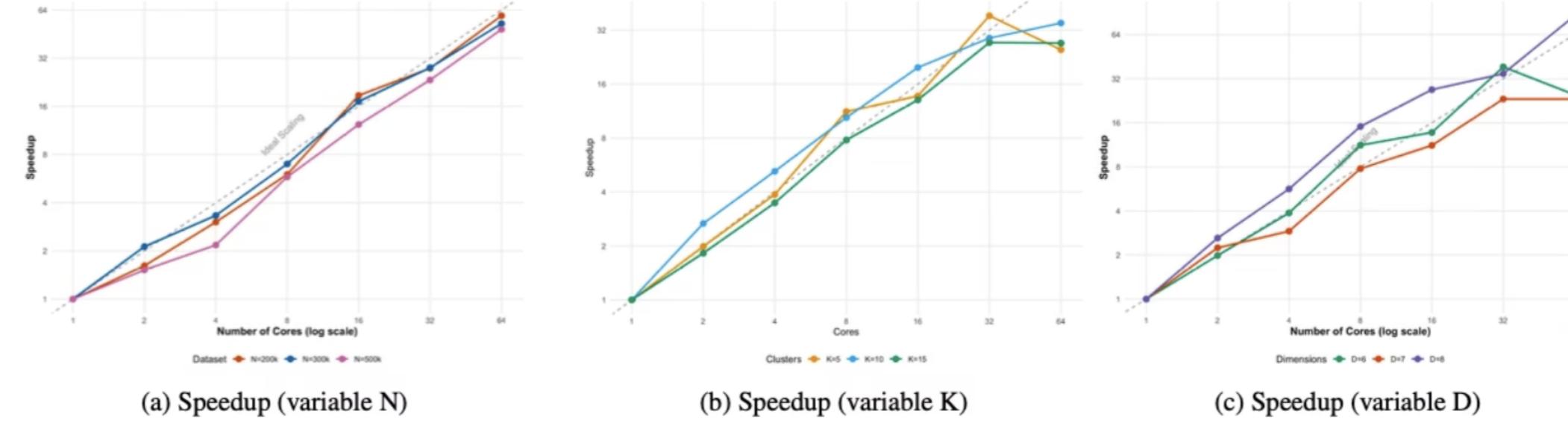


Figure 4: Speedup for different configurations of N, K and D.

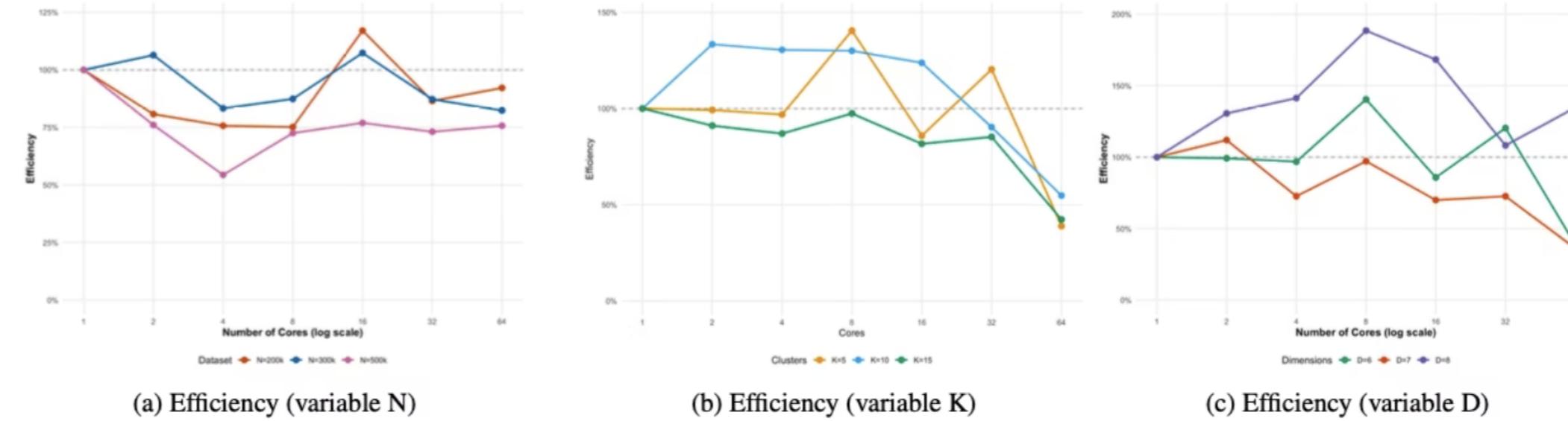


Figure 5: Efficiency for different configurations of N, K and D.

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

Scaling analysis (OpenMP)

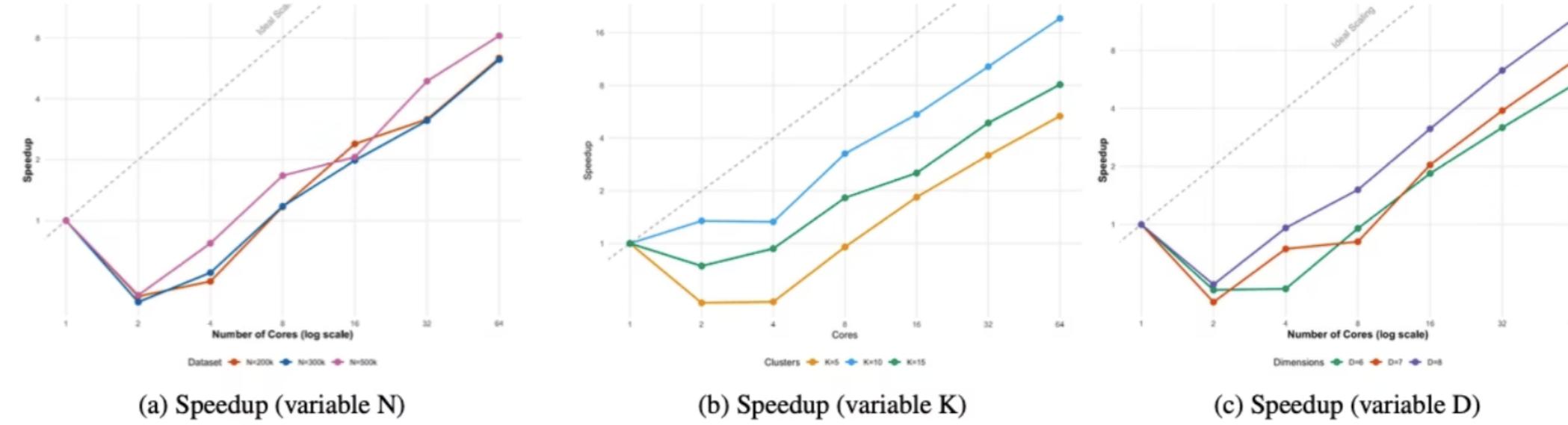


Figure 6: Speedup for different configurations of N, K and D.

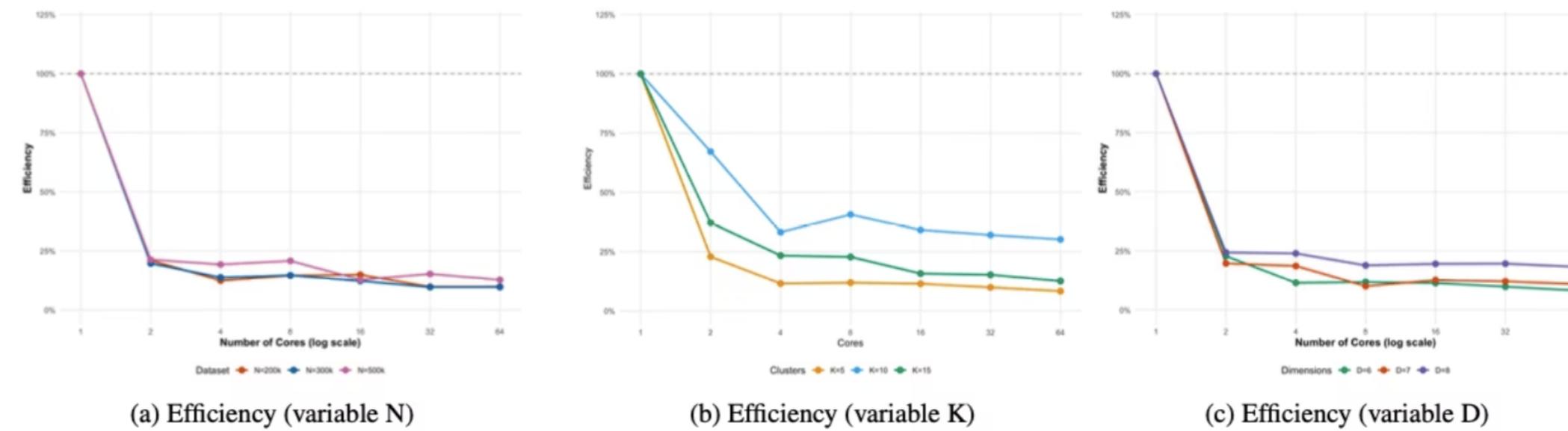


Figure 7: Efficiency for different configurations of N, K and D.

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`).