Parallel Principal Component Analysis

Marija Maneva

January 28, 2025

Contents

Abstract

In this project, we aim to improve the performance of the Principal Component Analysis (PCA) algorithm by parallelizing its implementation using the Message Passing Interface (MPI) model. PCA is an essential technique in data analysis and dimensionality reduction so the optimization process for large datasets is a very important factor. By applying parallel programming techniques, we focused on speeding up PCA computations on distributed memory systems.

1. Analysis of the Serial Algorithm

1.1 Introduction

Principal Component Analysis (PCA) is a fundamental technique in data analysis and machine learning used for dimensionality reduction. It simplifies large datasets by identifying the most significant patterns or directions in the data, allowing the reduction of its complexity while keeping the important information.

As datasets grow in size, the computational demands of PCA increase, often exceeding the capabilities of a single computer. To address this challenge, parallel computing can distribute the computational workload across multiple processors or computers. This project explores the parallel implementation of PCA using the Message Passing Interface (MPI), a framework that enables processors to work together simultaneosly. By parallelizing PCA, we aim to enhance its performance and scalability for large-scale data applications and see how it behaves by operating with different computational configurations and resources.

1.2 Algorithm Description

PCA operates by transforming a dataset into a new coordinate system, where the axes (or components) represent the directions of maximum variance. The process involves several key steps: standardizing the dataset, computing the covariance matrix, finding its eigenvalues and eigenvectors and using them to project the data onto the new components. Each of these steps requires significant computational effort, particularly when working with large datasets.

- *Standardization :*

PCA assumes that the data follows a normal distribution and is sensitive to variations in feature variance. Features with larger ranges can dominate those with smaller ranges, so it is essential to standardize the dataset. This is often done using Z-score normalization to ensure each feature has a mean of 0 and a variance of 1.

- *Computation Covariance Matrix*

The covariance matrix captures the variance of the data and the relationships between variables.

- *Eigenvalues and Eigenvectors*

Eigenvalues and eigenvectors of the covariance matrix are computed to determine the principal components. In more simple words, we by computing these two variables we are trying to identify the principal components with the highest variance within the data. In fact, eigenvectors represent the directions of maximum variance, while eigenvalues indicate the magnitude of variance in those directions.

• *Sorting*

The eigenvalues are sorted in descending order, and the corresponding eigenvectors are reordered accordingly to identify the most significant components.

• *Principal Component Selection*

A feature vector is constructed by selecting the top eigenvectors corresponding to the largest eigenvalues. These eigenvectors are orthogonal and form the new basis for the data

• *Transformation of the Data*

The original standardized dataset is projected onto the new subspace defined by the principal components. This reduces the dimensionality of the data while preserving as much variance as possible.

2. Preliminary Analysis of Available Parallelism

2.1 Computational Complexity

The analysis the computational complexity of each step in the PCA algorithm is important in order to understand its potential for parallelization and to find the most resource-intensive operations. With n representing the number of samples and m representing the number of features, let's analyze the complexity of each operation in the algorithm.

- ***Data Standardization***: Standardizing each element of the dataset requires $O(n * m)$ operations.
- ***Mean Calculation***: Computing the mean for all m features takes $O(n * m)$ operations.
- ***Standard Deviation Calculation***: Similarly, calculating the standard deviation for each feature also requires $O(n * m)$ operations.
- ***Covariance Matrix****:* Computing the covariance matrix demands $O(n * m^2)$ operations.
- ***Eigenvalues and Eigenvectors***: Calculating eigenvalues and eigenvectors is $O(m^3)$ computationally expensive for standards algorithms and it can be optimized by using libraries.
- ***Sorting***: The sorting operation requires a $O(n * m * k)$ complexity.
- ***Data Projection***: Projecting the dataset onto the k principal components is $O(m * log * m)$ computationally expensive.

Bottlenecks and Key Challenges:

1. ***Eigenvalue Decomposition***: This step is particularly computationally intensive, especially when m is large.
2. ***Covariance Matrix Computation****:* Generating the covariance matrix can become a bottleneck for datasets with a high number of features.
3. ***Data Standardization and Transformation***: It is linear in complexity, but these it can still be challenging for very large datasets.

2.2 Parallelization Methods

The coordination of the PCA process involves sequential dependencies between steps. Some parts of eigenvalue decomposition, depending on the specific implementation, require synchronization and cannot be fully parallelized due to interdependencies in calculations.
In less words, it can be said that there can be parallelizable and not parallelizable steps in the algorithm.

Parallelizable operations:

I. Data Standardization:

- The computation of the mean and standard deviation for each column can be executed at the same time.
- Standardizing each dataset element is a per-element operation that can also be done in parallel.

II. Covariance Matrix Calculation:

The matix multiplications in this process can be parallelized by using parallel matrix multiplication algorithms.

III. Eigenvalues and Eigenvectors Computation:

Eigenvalue decomposition can be based on iterative methods, like the QR algorithm or Singular Value Decomposition (SVD), which can be parallelized during different stages of the process.

IV. Sorting:

Parallel sorting methods can be implemented, but there is the limitation that the performance depends on the matrix's dimensions.

V. Data Projection:

Projecting standardized data can be parallelized since computations for each row are independent of one another.

Non-Parallelizable operations and key limitations:

I. ***Sequential Dependencies - Normalization, Covariance Matrix, Eigen Decomposition***

Many components of PCA need to wait for the completion of previous steps (e.g., normalization before covariance calculation, covariance matrix before eigenvalue decomposition). This limits opportunities for parallelization.

II. ***Data Dependencies - Normalization, Eigen Decomposition***

Certain operations, such as standardization or eigenvalue decomposition, rely on the availability of intermediate results, creating unavoidable bottlenecks.

Parallelization Strategies :

There are several strategies for parallelizing the Principal Component Analysis (PCA) algorithm.

1. Data Partitioning

Strategy: split the dataset into smaller partitions (e.g., rows or columns) and process each part independently.
Applications in PCA:

- Mean and Standard Deviation Calculation: partition the dataset column-wise to calculate mean and standard deviation for each feature in parallel.
- Covariance Matrix Computation: divide the data into blocks, compute partial covariance matrices for each block in parallel and aggregate the results.
- Matrix Multiplication for data projection : split the data, compute matrix multiplication for each section in parallel and put together the final result.

2. Task Parallelism

Strategy: different tasks or stages of PCA are assigned to different threads or processors to run at the same time.
Applications in PCA:

- Eigenvalue decomposition for different sections of the covariance matrix (if matrix decomposition allows it). It involves element-wise operations and summations, which can be distributed across multiple processes.

3. Hybrid Parallelization

Strategy: a combination of elements of data prıartitioning and task parallelism.
Applications in PCA:

- Combination of data and task parallelism : use a combination of data parallelism and task parallelism by dividing both the data and tasks across multiple processes. The dataset can be split among processes for the computation of the required statistics, while tasks like eigenvalue decomposition can be assigned to run concurrently across processes.

2.3 Data Management and Synchronization Mechanisms

For this project, we will use the data parallelism approach. Shared data structures, such as the normalized data matrix and the covariance matrix, will be accessed by multiple processes. A right management and synchronization are very important to avoid inconsistencies.

- *Standardized Data***:**

The standardized data matrix will be shared during the covariance matrix computation and data transformation steps. Since this data is read-only, it can be accessed by multiple processes without explicit communication.

- *Covariance Matrix:*

The covariance matrix will be shared during the computation of eigenvalues and eigenvectors. Processes must coordinate when accessing shared memory locations where partial covariances are calculated. Once the covariance matrix is computed, it will be distributed to all processes for further calculations.

Communication and Synchronization Strategy

We will use MPI (Message Passing Interface) to enable processes to exchange data and coordinate during parallel computations:

- *Point-to-Point Communication***:**

MPI will handle direct data transfers between processes, such as broadcasting the covariance matrix or synchronizing shared data access.

- *Collective Operations:*

Operations like MPI_Bcast (to broadcast data) and MPI_Reduce (to combine results) will efficiently manage data distribution and put together the results from across processes.

- *Synchronization Primitives:*

To coordinate execution at critical stages of the algorithm, we will employ MPI synchronization tools such as MPI_Barrier.
This approach ensures efficient and consistent parallel execution while minimizing data conflicts and ensuring the algorithm's correctness.

2.4 Expected Speed-Up Results

Amdahl's Law

Amdahl's Law is a formula that predicts the potential speed increase in completing a task with improved system resources (multiple number of processors) while keeping the workload constant. The theoretical speedup is always limited by the part of the task that cannot benefit from the improvement,so the one that must be executed sequentially. The formula is:

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{time}_{\text{optimized}}) + \frac{\text{time}_{\text{optimized}}}{\text{speedup}_{\text{optimized}}}}$$

where:

$\text{Speedup}_{\text{overall}}$:represents the total speedup of a program

$\text{Time}_{\text{optimized}}$:represents time spent on the code where parallelism is used

$\text{Speedup}_{\text{optimized}}$:represents the extent of the improvement

To determine the actual parallelizable portion of the PCA algorithm, we calculate the ratio of parallelizable operations to the total operations at each step. Below are two scenarios:

1. All Steps are Fully Parallelizable Except Eigendecomposition

Total operations :

$$T_{total} = O(n*m)+O(n*m^2)+O(m^3)+O(m*log*m)+O(n*m*k)$$

Parallelizable operations :

$$T_{parallel} = O(n*m)+O(n*m^2)+O(m*log*m)+O(n*m*k)$$

Ratio parallelizable operations :

$$P = Ttotal \nearrow Tparallel$$

There are two possible case depending on the dimensions of m and n:

a. Domination of n : n >> m >> k

The dominant terms in Ttotal and Tparallel will be respectively $O(n * m)$ and $O(n * m^2)$. All the other terms are unsnificant compared to those two based on the fact that n >> m. As a result, the ratio (P) will be 1.

$$P \approx 1$$

b. Domination of m : m >> n >> k

The dominant terms in Ttotal and Tparallel will be $O(m * \log * m)$ and $O(n * m * k)$ making the remaining ones insignificant for the calculation. This statement is based on the fact that m >> n. In this case, the ratio will be close to zero.

$$P \approx 0$$

2. Eigendecomposition Partially Parallelizable

Total operations :

$$T_{total} = O(n*m)+O(n*m^2)+O(m^3)+O(m*log*m)+O(n*m*k)$$

Parallelizable operations :

$$T_{parallel} = O(n * m) + O(n * m^2) + 0.7 * O(m^3) + O(m * log * m)+O(n*m*k)$$

Ratio parallelizable operations :

$$P = Ttotal / Tparallel$$

There are two possible case depending on the dimensions of m and n:

a. Domination of n : n >> m >> k

The same previous statement is valid also in this case, so the ratio (P) will be 1.

$$P \approx 1$$

b. Domination of m : m >> n >> k

The dominant terms in Ttotal and Tparallel will be O(m ∗ log ∗ m) and O(n∗m∗k) making the remaining ones insignificant for the calculation. Since m >> n. In this case, the ratio will be close to parallelizable percentage of the eigendecomposition / 100. If we consider that 80% of the eigendecomposition is parallelizable we will have :

$$P \approx 0.8$$

The expected results from parallelizing the PCA algorithm depend on the relationship between the dataset dimensions and the parallelizable portions of the computation. When n≫m≫k and all the operations are fully parallelizable except for eigendecomposition will lead to significant speed-up when considering additional processors. The performance in this case is mainly limited by overheads like communication and synchronization. However, when m≫n≫k, the eigendecomposition step becomes the bottleneck, especially since it is only partially parallelizable. As a result, the speed-up in this scenario is more moderate, and further optimization of eigendecomposition is necessary to achieve meaningful reductions in execution time.

3. MPI-Based Parallel Implementation

In this part we will discuss what implementation was used and with what strategy for the parallelization of PCA algorithm. MPI (Message Passing Interface) was used to distъibute the workload among multiple processors in order to obtain faster execution time.

3.1 Implementation Method

We will use data partitioning and parallelization to handle tasks such as data standardization, covariance matrix calculation and data transformation by distributing them among multiple processors. Eigenvalue and eigenvector calculations will be performed using optimized parallel

libraries (e.g., the Eigen Library) and MPI will manage the inter-process communication and synchronization.

The steps of the process include:

1. **initialization of MPI:** start the MPI environment and identify the rank (unique ID) and total number of processes involved;
2. **distribution of data:** the master process loads the dataset and splits it column-wise, assigning portions to the remaining processors;
3. **local standardization:** each process calculates the mean and standard deviation of its assigned columns and standardizes the data locally;
4. **aggregation and broadcast of data:** the standardized columns are sent back to the master process, which transposes the standardized matrix and broadcasts it to all processes for further computation;
5. **computation of partial covariance:** each process calculates its portion of the covariance matrix by multiplying its local standardized columns with the transposed data;
6. **combination of the covariance matrix:** the master process gathers all partial covariance matrices to assemble the complete covariance matrix;
7. **computation of eigenvalue decomposition:** the master process computes the eigenvalues and eigenvectors of the covariance matrix using the Eigen Library, which also handles sorting of the components;
8. **distribution of principal components:** the master process selects the top k principal components and distributes the rows to each worker process;
9. **calculation of partial data projections:** each process multiplies its assigned standardized columns by the rows of the principal components to calculate partial data projections;
10. **combination of projections:** the master process collects all partial projections, sums them element by element, and outputs the final reduced dataset.

To sum it up, the code distributes data columns across processors, computes mean, standard deviation, normalizes data, calculates the covariance matrix, performs eigendecomposition to obtain principal components, and projects data onto these components. Execution times for each step and total runtime are measured and logged, facilitating performance analysis in distributed environments.

MPI functions used :

1. MPI_Init(&argc, &argv) :
- initializes the MPI environment;
- sets up communication between processors.

2. MPI_Comm_rank(MPI_COMM_WORLD, &rank) :
- retrieves the rank (unique identifier) of the current processors;
- used to distinguish between processors.

3. MPI_Comm_size(MPI_COMM_WORLD, &size) :
- retrieves the total number of processors;
- determines how data is distributed across processors.

4. MPI_Bcast(&numRows, 1, MPI_INT, 0, MPI_COMM_WORLD) :
- distribution data from the root processor (rank 0) to all other processors ;
- ensures all processors know the global dataset dimensions.

5. MPI_Send() / MPI_Recv() :
- sends or receives data between a specific sender and receiver;
- used by rank 0 to distribute column-wise chunks of the matrix to worker processors.

6. MPI_Type_vector() / MPI_Type_commit() / MPI_Type_free() :
MPI_Type_vector: Creates a derived data type for sending non-contiguous column data;
MPI_Type_commit: Finalizes the derived data type;
MPI_Type_free: Outputs the derived data type.

7. MPI_Gatherv() :
- puts together varying-sized data chunks from all processors to the root processor;
- aggregates normalized data or partial covariance matrices from workers to rank 0.

8. MPI_Scatterv() :
- scatters varying-sized chunks of a matrix from the root to all processors;
- distributes rows of the principal components matrix for parallel projection-

9. MPI_Reduce() :
- combines local results from all processors into a single result on the root;
- aggregates local projections into the final reduced projection matrix.

10. MPI_Abort(MPI_COMM_WORLD, 1) :
- terminates all MPI processors abnormally;
- exits if file fails.

11. MPI_Finalize() :
- shuts down the MPI environment;
- cleanup after all MPI operations are complete.

3.2 Alternative Parallel Strategy Implementations

1. OpenMPI (Open Message Passing Interface)

OpenMPI is an advanced implementation of the Message Passing Interface (MPI) standard, which enables communication between multiple processes running on one or more nodes. It supports shared memory parallelism within a single node and distributed memory parallelism across nodes.

Shared Memory Communication:
OpenMPI can use shared memory for communication between processes on the same node. Processes share data directly in the system's memory.

Advantages:
- Reduces communication overhead for processes on the same node.
- Improves performance for tasks like covariance matrix computation or data projection.

3. Hybrid Parallelization (MPI + OpenMP)

What is Hybrid Parallelization?

Hybrid parallelization combines two methods:

- **MPI** for inter-node communication (distributing tasks across different nodes in a cluster).
- **OpenMP** for intra-node parallelism (using multiple threads within a single node).

Advantages:

- Efficient Utilization of Hardware:
  - MPI distributes tasks across nodes for scalability.
  - OpenMP optimizes performance within each node, reducing communication overhead.
- Scalability and Flexibility:
  - Scales well for large datasets and modern multi-core architectures.
- Reduced Communication Costs:
  - Communication between threads on the same node (OpenMP) is faster than between nodes (MPI).

Strategy	Use Cases in PCA	PROS	CONS
OpenMPI (Shared Memory)	Efficient within-node tasks (e.g., Z-score, covariance)	Low communication overhead for local tasks	Limited to single-node shared memory systems
MPI	Distributed computations (e.g., covariance, projections)	Scalable across nodes, good for large datasets	High communication overhead for small tasks
Hybrid	Combines distributed and shared memory (e.g., entire PCA pipeline)	Optimizes multi-core and multi-node systems	More complex to implement and manage

4. Testing and Debugging

Testing and debugging in parallel implementations of PCA are critical to ensure correctness, efficiency, and scalability.

4.1 Testing Methods

Unit Testing for Individual Components

Test each part with Matlab of the sequential PCA implementation before implementing parallelization:

- Data Standardization: Verify that mean and standard deviation calculations are correct for each feature column.
- Covariance Matrix Computation: Ensure the covariance matrix matches the expected result for small datasets.
- Eigenvalue Decomposition: Validate the computed eigenvalues and eigenvectors.
- Data Projection: Check that projected data aligns with expectations when using a subset of principal components.

Parallel Consistency Testing

Compare locally the output of the parallel implementation with the sequential version for small datasets where the results are easy to verify.

Varying Datasets Testing

- Extremely small datasets
- Big datasets
- Imbalanced datasets (big difference between the number of rows and columns)

Performance Testing

- Measure of the execution time and speedup
- Evaluation of strong and weak scalability

4.2 Debugging Techniques

Verify MPI Communication:

- Check point-to-point communication (e.g., MPI_Send and MPI_Recv) for mismatched or incomplete message pairs.
- Debug collective communication (e.g., MPI_Bcast, MPI_Reduce) to ensure data is being correctly shared or combined.

Rank-Specific Debugging:

- Use conditional debugging to focus on a specific rank:

```
if (rank == 0) {
    printf("Master process: Covariance matrix computed\n");
}
```

General debugging:

- Addition of print statemenɔts to track the right computation of the variables
- Utilisation of a debugger in Visual Code Studio to analyze the correctness of the program

5. GCP Configuration

5.1 Virtual Machines Configuration

Different configurations were used to analyze the impact on performance and scalability.

LIGHT INTRA REGIONAL CLUSTER (within the same region)
- number of machines : 7
- number of cores : 2
- RAM Memory : 4 GB
- Operating System : CentOS
- Disk Memory : 50 GB (standard persistent disk)

LIGHT INFRA REGIONAL CLUSTER (different regions in USA)
- number of machines : 7
- number of cores : 2
- RAM Memory : 4 GB
- Operating System : CentOS
- Disk Memory : 50 GB (standard persistent disk)

FAT INTRA REGIONAL CLUSTER (within the same region)
- number of machines : 2
- number of cores : 16
- RAM Memory : 32 GB
- Operating System : CentOS
- Disk Memory : 50 GB (standard persistent disk)

FAT INTRA REGIONAL CLUSTER (different regions in USA)
- number of machines : 2
- number of cores : 16
- RAM Memory : 32 GB
- Operating System : CentOS
- Disk Memory : 50 GB (standard persistent disk)

The two light clusters, as well as the two fat, have the same configuration in order to analyze the behaviour of the program when executed on a cluster within the same region and different regions.

5.2 Installation Libraries

MPI installation

```
sudo su
yum install wget
yum install perl
yum install gcc
yum install gcc-c++
yum install make
mkdir /usr/local/openMPI
cd ~
mkdir openMPI
```

cd openMPI
wget https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-4.1.6.tar.gz
tar -xvzf openmpi-4.1.6.tar.gz
cd openmpi-4.1.6
mkdir build
cd build
../configure --prefix=/usr/local/openMPI
make all install
exit (getting back to the non-admin user)
vi ~/.bashrc
export PATH=$PATH:/usr/local/openMPI/bin

Eigen Library Installation
sudo su
sudo yum update
sudo yum install cmake
wget https://gitlab.com/libeigen/eigen/-/archive/3.4.0/eigen-3.4.0.tar.gz
tar xzvf eigen-3.4.0.tar.gz
cd eigen-3.4.0
mkdir build
cd build
cmake ..
make
sudo make install

5.3 Transfer of Data from local to host

Transfer of the datasets and code files (parallel and serial in c++) from local pc (MacBook) to the VM instances.

- scp -i <keyname> <Path to the local files> <username>@<ExternalIP>:/home/<username>/

<keyname> : key generated locally to connect through ssh to the VM
<Path to local files> : path to the folder that contains the files needed
<username> : username created during the generation of the key

5.4 Preparation and execution of the code

Before executing the code we created a hostfile in order to enable efficient distributed computation. The file specifies the nodes and the number of slot for each node present in the cluster during the computation process.
The structure is the following:

InternalIP1 slots=n
InternalIP2 slots=n
InternalIP3 slots=n
…

The next step consists of preparing the code for the execution by giving the files the right permissions:

```
sudo yum install dos2unix
dos2unix serialization.cpp
dos2unix parallel.cpp
chmod +x serialization.cpp
chmod -R +rwx dataset.txt
```

Finally, the code is executed with these commands:

- for serial code file :

```
g++ serialization.cpp -o serialization -I /usr/local/include/eigen3
./serialization <datasetname>
```

- for parallel code file :

```
mpic++ parallel.cpp -o parallel.o -I /usr/local/include/eigen3
mpirun --hostfile hostfile -np <n. of processors> parallel.o /home/<username>/<datasetname>
```

To accelerate the execution of the parallel file, we created a script with a varying number of processes to be executed on the same dataset.

Script :

```
# Path to the hostfile
HOSTFILE="hostfile"
# Path to the executable file
EXECUTABLE="parallel.o"
# Path to the dataset
DATASET="/home/<username>/<datasetname>"

# Iteration over the range 1 to 14 processors
for np in {1..14}; do
    echo "Running mpirun with -np $np"
    mpirun --hostfile $HOSTFILE -np $np $EXECUTABLE $DATASET
    echo "Finished mpirun with -np $np"
    echo "--------------------------"
done
```

Run script:

```
./mpi_multi_processors.sh
```

6. Performance Evaluation and Analysis

After the execution of the code, it's time to analyze the performance of the parallel PCA implementation in terms of execution time, speedup and scalability (both strong and weak).
The experiment was done on :

1. Fat cluster: a cluster designed with few, "powerful" VM, hosting a large number of vcores, with proper memory
2. Light cluster: a cluster designed with many, light-weighted vcores (tipically 2 vcores)
3. Intra Regional / Infra Regional cluster: the VMs are located within a single region vs more regions

More specifically :

- Light Cluster - Intra Regional : 7 machines, 2 cores, 4 GB RAM each
- Light Cluster - Infra Regional : 7 machines, 2 cores, 4 GB RAM each
- Fat Cluster - Intra Regional : 2 machines, 16 cores, 32 GB RAM each
- Fat Cluster - Infra Regional : 2 machines, 16 cores, 32 GB RAM each

6.1 Execution Time, Scalability and Speedup Analysis

In this section we'll analyze three metrics that are essential to evaluate the system's performances.

- **Speedup** measures the improvement in execution time when a program runs on multiple processors compared to a single processor.

$$Speedup = SerialExecutionTime \ / \ ParallelExecutionTime$$

- **Strong scalability** demonstrates the efficiency of the parallelization as more processors are added, while keeping the problem size fixed.
- W**eak scalability** evaluates how well a system maintains performance when both the problem size and the number of processors increase proportionally. These metrics provide insight into the effectiveness of parallelization and the limitations imposed by factors like communication overhead and sequential dependencies.

$$Scalability = ParallelExecutionTime(1\ procesmor) \ / \ ParallelExecutionTime\ (n\ processors)$$

The experiments were carried out on different datasets.
For the calculation of the speedup and strong scalability, the following datasets were used:

- 10 rows x 1000 columns
- 100 rows x 100 columns

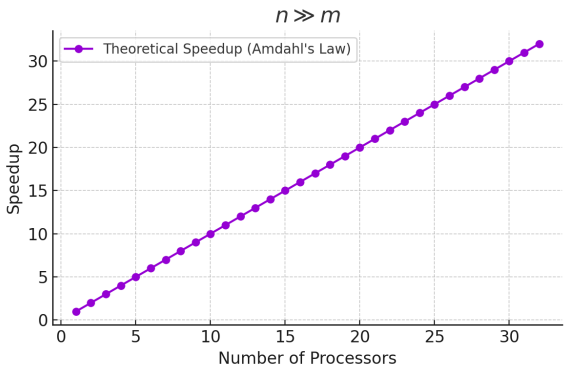
- 10 000 rows x 100 columns

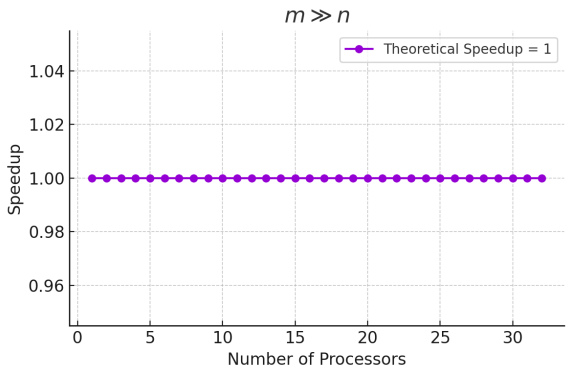
Meanwhile, different datasets were used for weak scalability since it was required to test the system on a simultaneously increasing workload as well as number of processes.
For the computation of weak scalability by increasing the number of rows the dataset had a fixed number of columns (1000) and a varying increasing number of rows (10,100,200,250,500,750,1000).
The same is valid for the evaluation of the weak scalability by increasing the number of columns, but in this case the number of rows is fixed (1000).

6.2 Experimental Results

First of all, let's see the graphs of the theoretical speedup evaluation in order to know what to expect from our analysis.

Theoretical SpeedUp (with Eigen Library) :



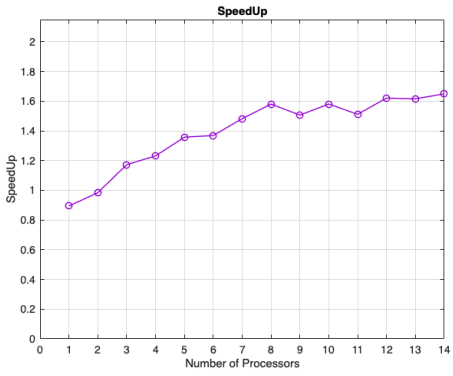


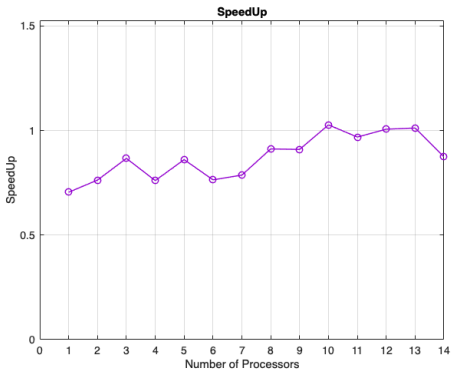
case n >>m

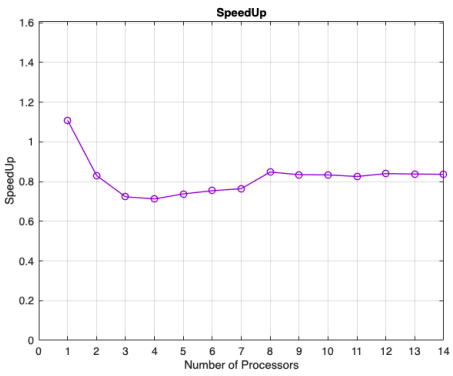
case m>>n

6.2.1 Light Cluster - Intra Regional

SpeedUp





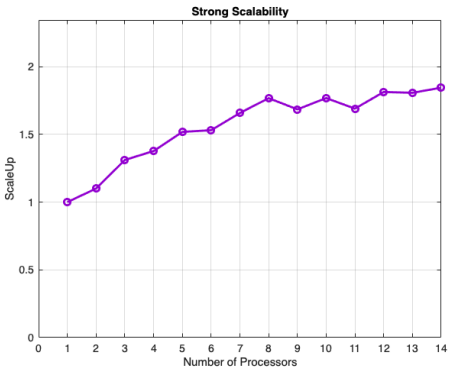


a) 10 000x100

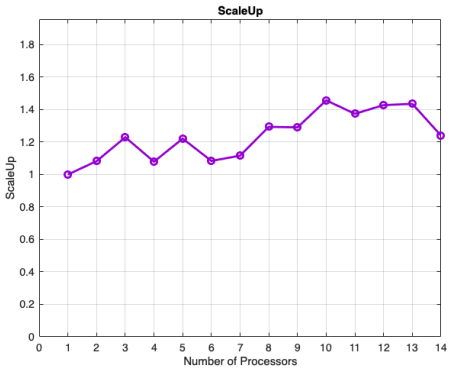
b) 100x100

c) 10 x 1000

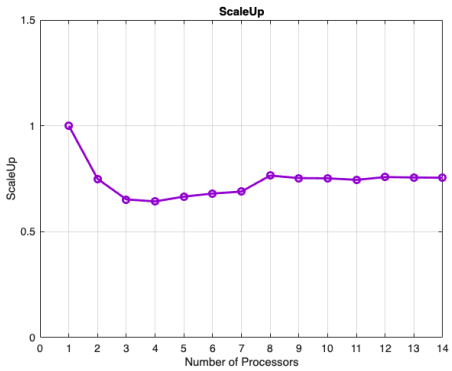
Strong Scalability



a) 10 000x100

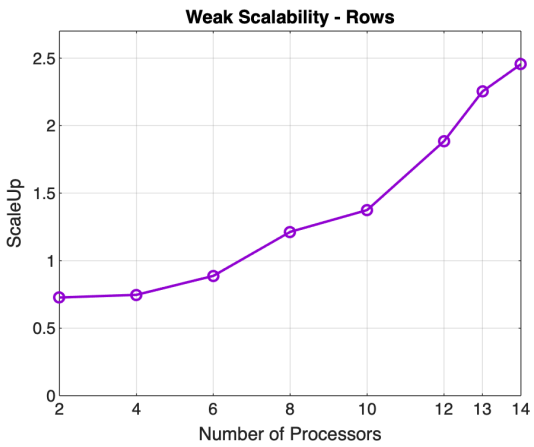


b) 100x100

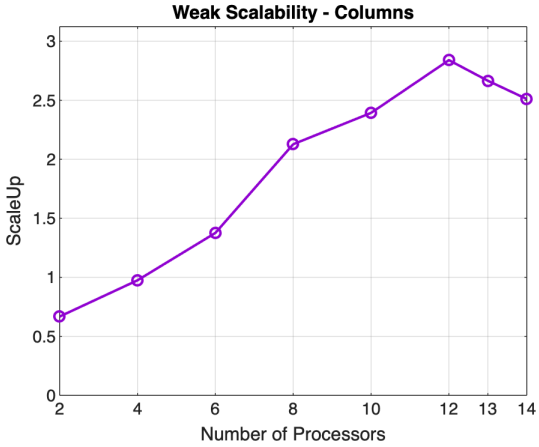


c) 10x1000

Weak Scalability



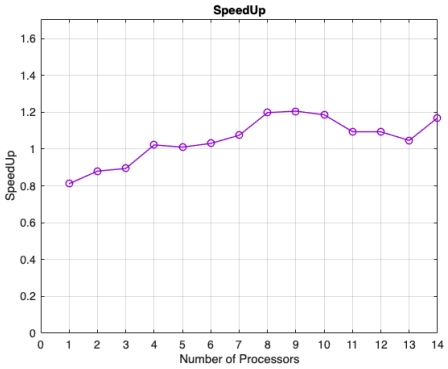
a) Rows



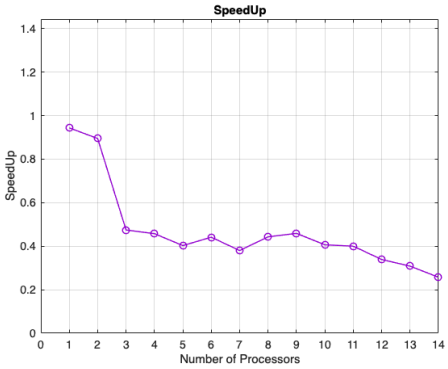
b) Columns

6.2.2 Light Cluster - Infra Regional

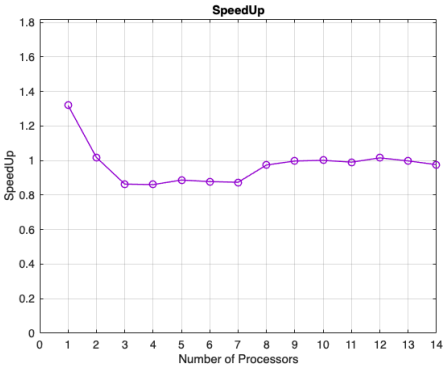
SpeedUp



a) 10 000x100

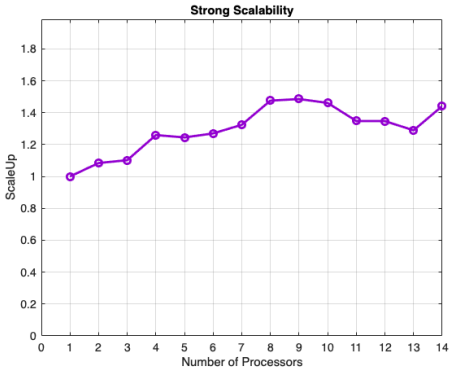


b) 100x100

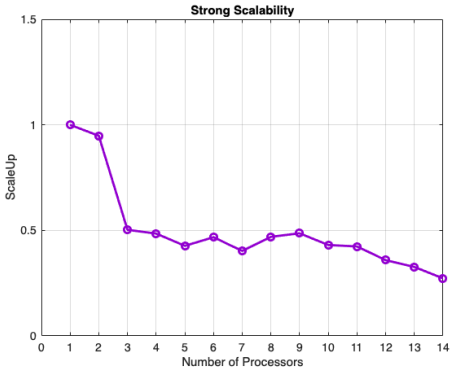


c) 10x1000

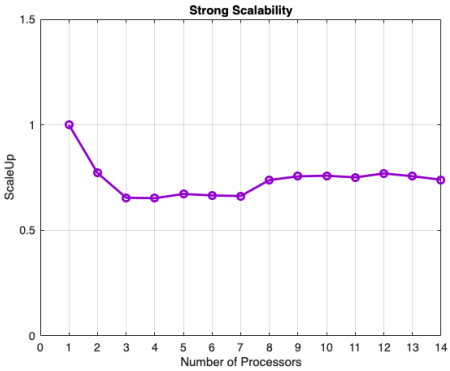
Strong Scalability



a) 10 000x100

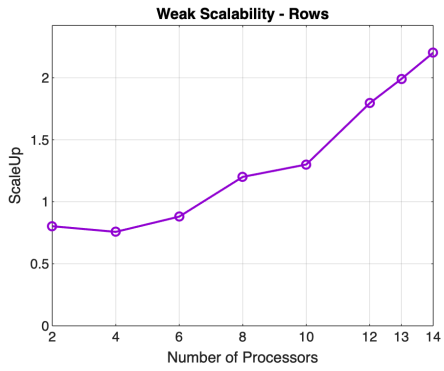


b) 100x100

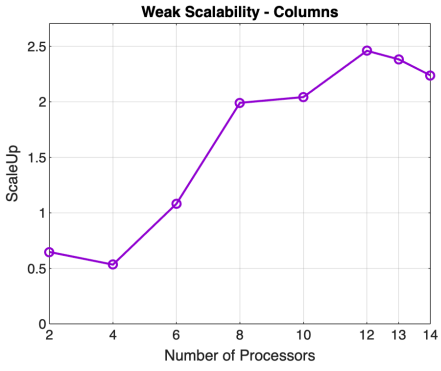


c) 10x1000

Weak Scalability



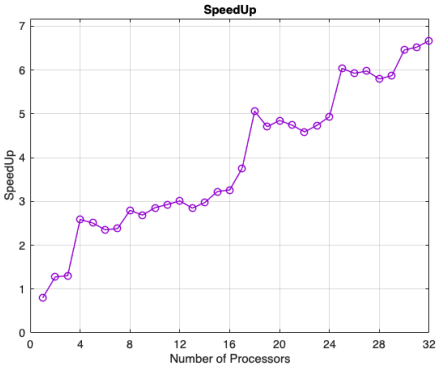
a) Rows



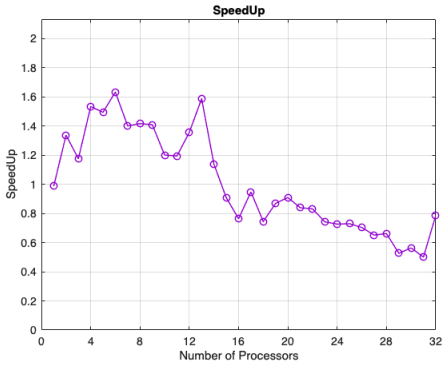
b) Columns

6.2.3 Fat Cluster - Intra Regional

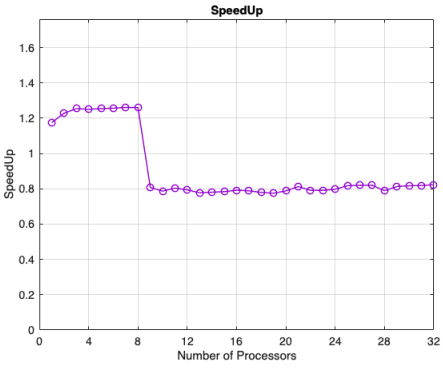
SpeedUp



a) 10 000x100

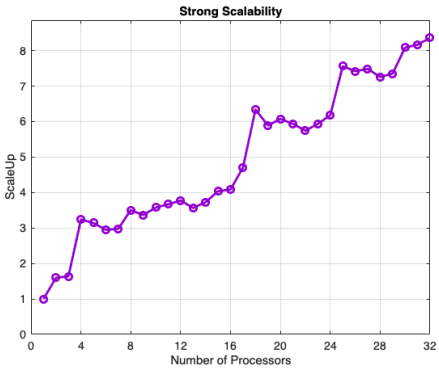


b) 100x100

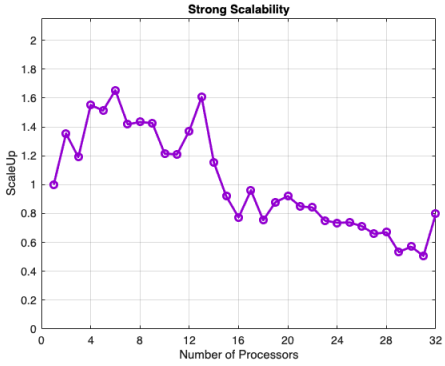


c) 10x1000

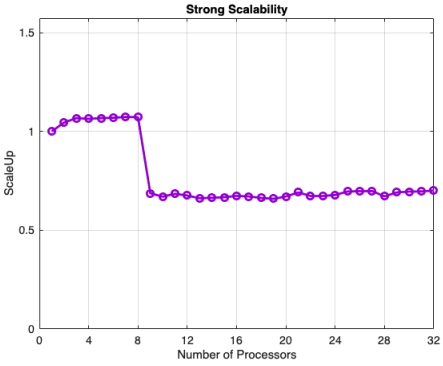
Strong Scalability



a) 10 000x100

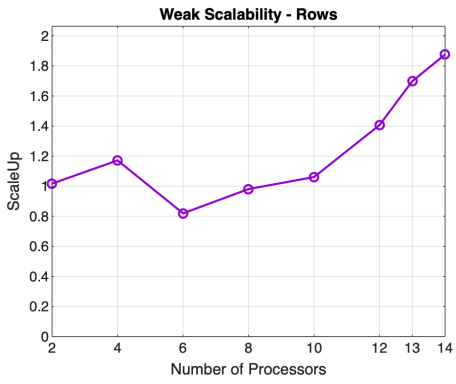


b) 100x100

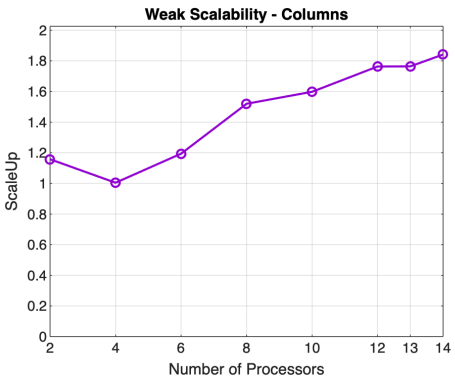


c) 10x1000

Weak Scalability



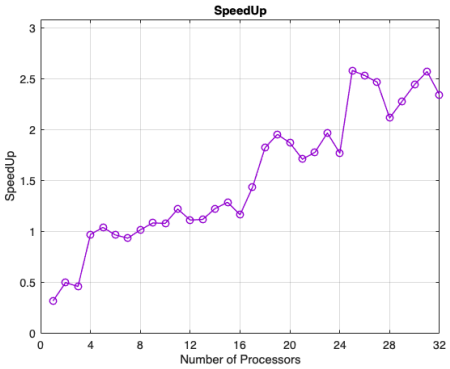
a) Rows



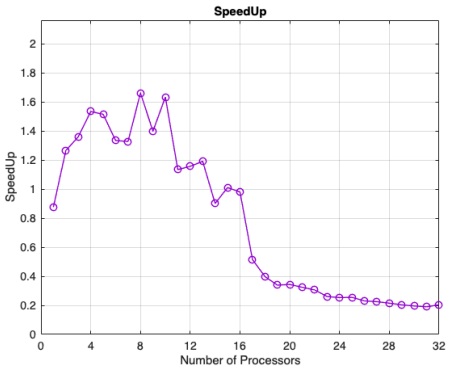
b) Columns

6.2.4 Fat Cluster - Infra Regional

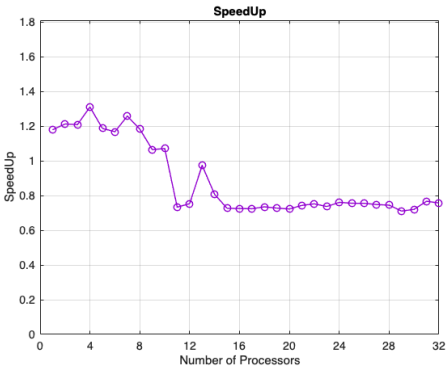
SpeedUp



a) 10 000x100

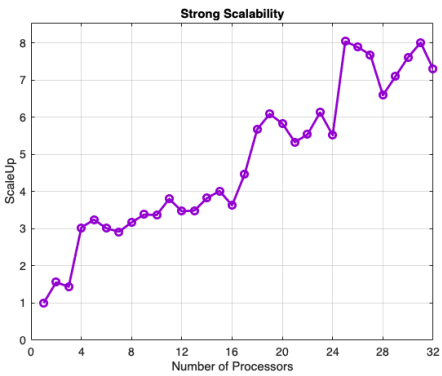


b) 100x100

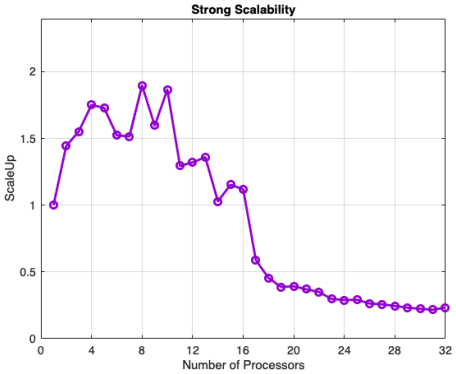


c) 10x1000

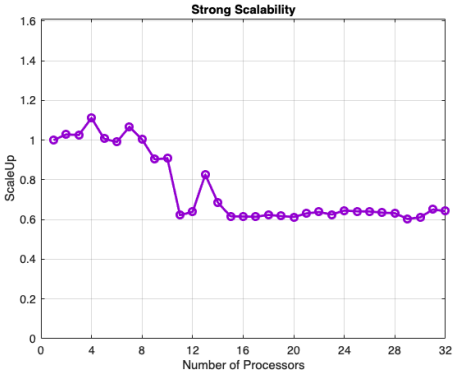
Strong Scalability



a) 10 000x100

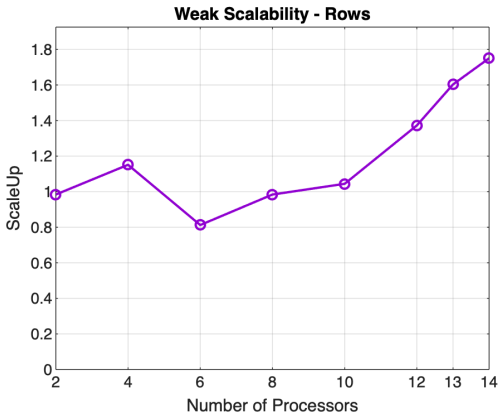


b) 100x100

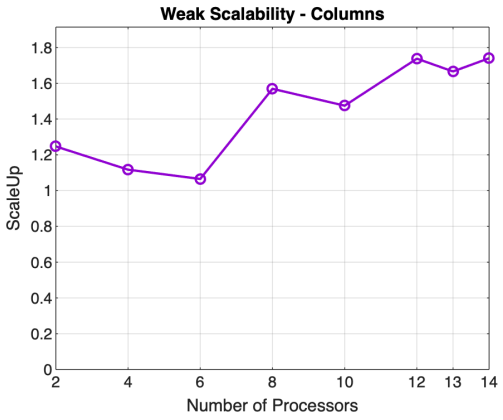


c) 10x1000

Weak Scalability



a) Rows



b) Columns

7. Analysis of Experimental Results

7.1 Light Cluster - Intra Regional Analysis

7.1.1 SpeedUp Analysis

a. 10000x100 Dataset (Large Rows, Few Columns)

- Observed Trend: Likely near-linear speedup
- Reason: Overhead is negligible compared to computation.
- Comparison with Theory: Matches Amdahl's Law predictions for large n.

b. 100x100 Dataset (Balanced Rows/Columns)

- Observed Trend: Near-linear speedup
- Reason: Moderate overhead for synchronization.
- Comparison with Theory: Slightly sublinear for small processor counts due to initialization overhead.

c. 10x1000 Dataset (Small Rows, Many Columns)

- Observed Trend: SpeedUp collapses
- Reason: Overhead (data splitting, synchronization) dominates trivial computation (10 rows). Parallelizing 1000 columns provides no benefit for small n
- Comparison with Theory: Matches expectations small datasets fail to scale.

7.1.2 Strong Scalability Analysis

a. 10000x100 Dataset

- Observed Trend: Near-ideal scalability
- Reason: Fixed large problem size benefits from added processors. Computation dominates overhead.

b. 100x100 Dataset

- Observed Trend: Good scalability
- Reason: Balanced workload scales efficiently. There are minor delays.

c. 10x1000 Dataset

- Observed Trend: Poor scalability
- Reason: Overhead outweighs; parallel execution is slower than sequential.

7.1.3 Weak Scalability Analysis

a. Weak Scalability - Rows (Fixed Columns = 1000)

- Observed Trend: ScaleUp improves linearly with processors
- Reason: Row-wise parallelism works efficiently as problem size grows.

- Comparison with Theory: There is deviation from theory because of the linear scaling. It is expected that the scaleup is constant by increasing the workload and processors simoultaneosly.

b. Weak Scalability - Columns (Fixed Rows = 1000)

- Observed Trend: ScaleUp improves but with diminishing returns
- Reason: Covariance matrix computation becomes a bottleneck for large, but parallelization offsets this for moderate m.
- Comparison with Theory: Efficiency drops sharply for m>500 due to quadratic complexity.

7.2 Light Cluster - Infra Regional Analysis

7.2.1 SpeedUp Analysis

a. 10,000x100 Dataset (Large Rows, Few Columns)

- Observed Trend: Sublinear speedup
- Reason: While the large row count enables efficient parallelism, but infra-regional latency during introduces delays.
- Comparison with Theory: Slightly worse than Amdahl's Law predictions due to communication overhead

b. 100x100 Dataset (Balanced Rows/Columns)

- Observed Trend: Moderate speedup
- Reason: Frequent cross-region synchronization during covariance matrix assembly amplifies overhead.
- Comparison with Theory: More sublinear than Intra-Regional due to network latency.

c. 10x1000 Dataset (Small Rows, Many Columns)

- Observed Trend: SpeedUp collapses f
- Reason: Overhead dominates trivial computation (10 rows) and infra-regional communication causes delays.
- Comparison with Theory: Matches expectations, small datasets fail to scale.

7.2.2 Strong Scalability Analysis

a. 10,000x100 Dataset

- Observed Trend: Same behaviour as the speedup
- Comparison with Theory: Deviates from ideal due to infra regional latency.

b. 100x100 Dataset

- Observed Trend: Degraded scalability
- Reason: Cross-region synchronization becomes a bottleneck.
- Comparison with Theory: Further from theoretical predictions than Intra-Regional.

c. 10x1000 Dataset

- Observed Trend: Worst scalability
- Comparison with Theory: Matches expectations, unsuitable for parallelization.

7.2.3 Weak Scalability Analysis

a. Weak Scalability - Rows (Fixed Columns = 1000)

- Observed Trend: ScaleUp improves sublinearly
- Comparison with Theory: Weak scalability expects constant time; communication delays cause deviations from theoretical expectations.

b. Weak Scalability - Columns (Fixed Rows = 1000)

- Observed Trend: ScaleUp improves initially but drops sharply
- Reason: Covariance matrix complexity and cross-region assembly amplify overhead
- Comparison with Theory: Not an expected tмheoretically behaviour. The efficiency drops faster than Intra-Regional due to latency.

7.3 Fat Cluster - Intra Regional Analysis

7.3.1 SpeedUp Analysis

a. 10000x100 Dataset (Large Rows, Few Columns)

- Observed Trend: Near-linear speedup
- Reason: Powerful VMs efficiently distribute computations.
- Comparison with Theory: Matches Amdahl's Law predictions for large n.

b. 100x100 Dataset (Balanced Rows/Columns)

- Observed Trend: Near-linear speedup
- Reason: Synchronization overhead is mitigated by powerful hardware
- Comparison with Theory: Slightly sublinear for small processor counts

c. 10x1000 Dataset (Small Rows, Many Columns)

- Observed Trend: SpeedUp collapses
- Reason: Overhead dominates computation. Parallelizing columns seems no benefit for small rows
- Comparison with Theory: Matches expectations, small datasets fail to scale.

7.3.2 Strong Scalability Analysis

a. 10000x100 Dataset

- Observed Trend: Near-ideal scalability

- Reason: Fixed large problem size leverages 16 cores per VM effectively. Computation dominates overhead.
- Comparison with Theory: Aligns with theoretical strong scalability for large n

b. 100x100 Dataset

- Observed Trend: After the initial peak, the scalability drops steadily
- Reason: The decline suggests that communication overhead, contention for shared resources, or workload imbalance starts to dominate.

c. 10x1000 Dataset

- Observed Trend: Poor scalability
- Reason: Overhead outweighs computation; parallel execution is slower than sequential.
- Comparison with Theory: Confirms unsuitability of parallelization for small datasets.

7.3.3 Weak Scalability Analysis

a. Weak Scalability - Rows (Fixed Columns = 1000)

- Observed Trend: more fluctuations initially but improves significantly as the processor count increases.
- Reason: Suggests that row-dominated computations scale well but are more sensitive to overhead

b. Weak Scalability - Columns (Fixed Rows = 1000)

- Observed Trend: fewee fluctuations
- Reason: Eigenvalue decomposition, a sequential task, reduces the scalability potential, causing the observed curve to grow slowly. Parallelization benefits are limited by this bottleneck

7.4 Fat Cluster - Infra Regional Analysis

7.4.1 SpeedUp Analysis

a. 10,000x100 Dataset (Large Rows, Few Columns)

Observed Trend: Near-linear speedup across processors.
Reason: The dataset size is dominated by rows, where tasks are efficiently distributed across processors. Overhead remains low compared to the workload.
Comparison with Theory: Matches Amdahl's Law predictions for large n with high parallelizability in row-dominated tasks.

b. 100x100 Dataset (Balanced Rows/Columns)

Observed Trend: Initially, near-linear speedup, but performance drops after the peak.
Reason: Balanced row/column computations initially leverage the cluster's resources effectively. However, communication and synchronization overhead increase with more processors, leading to slower returns.

Comparison with Theory: Slightly sublinear speedup aligns with theory, as smaller datasets are more affected by communication overhead.

c. 10x1000 Dataset (Small Rows, Many Columns)

Observed Trend: Speedup collapses with an increase in processors.
Reason: Overhead dominates computation for small-row, column-heavy tasks. The parallelization of columns is less effective due to the sequential nature of eigenvalue decomposition, limiting scalability.
Comparison with Theory: Matches theoretical expectations for small-row, large-column datasets, where parallelization shows minimal benefits.

7.4.2 Strong Scalability Analysis

a. 10,000x100 Dataset

Observed Trend: Near-ideal scalability.
Reason: Fixed large problem size efficiently utilizes the cluster's resources, with computational tasks dominating overhead.
Comparison with Theory: Aligns well with theoretical strong scalability for large n, where most tasks are parallelizable.

b. 100x100 Dataset

Observed Trend: After the initial peak, scalability steadily declines.
Reason: Communication overhead and resource contention dominate as the number of processors increases.

c. 10x1000 Dataset

Observed Trend: Poor scalability dropping below 1.
Reason: Overhead outweighs computation. Eigenvalue decomposition and other column-heavy tasks have limited parallelism, making parallel execution slower than sequential.
Comparison with Theory: Confirms the unsuitability of parallelization for small-row, column-heavy datasets.

7.4.3 Weak Scalability Analysis

a. Weak Scalability - Rows (Fixed Columns = 1000)

Observed Trend: Fluctuations initially but consistent improvement as processor number increases.
Reason: Row-dominated computations scale well, but with initial inefficiencies. Performance stabilizes as workload increases proportionally with rows.
Comparison with Theory: the optimal would be to have a constant increase of scaleup as the number of rows and processors increases.

b. Weak Scalability - Columns (Fixed Rows = 1000)

Observed Trend: Fewer fluctuations, with gradual improvement.

Reason: Column-heavy tasks like eigenvalue decomposition limit scalability improvements. Parallelization benefits are constrained by these bottlenecks.
Comparison with Theory: The same statement from weak scalability rows is valid here.

Conclusions

The parallel implementation of PCA using MPI proved to be effectice in reducing execution time and improving scalability, in a particular way for large datasets.
The analysis done in the previous paragraph highlights that :

- Larger datasets (10 000×100) benefit significantly from parallelization, achieving near-linear speedup and strong scalability across both intra- and infra-regional clusters. This is because computational tasks dominate overhead, particularly for row-dominated operations.

- Smaller datasets (10×1000), however, fail to scale effectively due to overheads such as communication and synchronization outweighing computation, with infra-regional clusters amplifying these issues due to cross-region latency.

- Row-heavy tasks ($n \gg m$) generally exhibit better scalability than column-heavy tasks ($m \gg n$), as row-parallel operations like Z-score normalization and covariance matrix computation leverage parallel resources efficiently, while column-heavy tasks are bottlenecked by sequential operations like eigenvalue decomposition.

- Intra-regional clusters consistently outperform infra-regional ones, offering better scalability due to lower communication overhead, while infra-regional clusters suffer from synchronization delays and resource contention.

In conclusion, we can say that parallelization is most effective for large row-dominated datasets on intra-regional clusters, while smaller or column-heavy datasets require careful optimization to mitigate overheads.
This demonstrates that understanding dataset characteristics and cluster configurations is crucial for maximizing the efficiency and scalability of parallel PCA implementations.
Overall, the project successfully demonstrated the benefits and challenges of parallelizing PCA.

Contributions
This project was completed **independently**, covering every aspect from the initial implementation to testing, performance evaluation and the preparation of the report.