**Mini-Project: MPI-Based Distributed Matrix Multiplication   
Project Title: Distributed Matrix**  Multiplication using MPI

**Objective:** To implement and evaluate the performance of matrix multiplication across multiple nodes using MPI.

Tasks:

1. **Environment Setup:** Set up an MPI development environment.
2. **Matrix Multiplication:** Implement a standard matrix multiplication algorithm.
3. **Distributed Implementation**: Modify the algorithm for distributed computation using MPI, focusing on data partitioning and inter-process communication.
4. **Performance Metrics**: Develop a system to measure execution time and scalability.
5. **Scalability Testing**: Test the algorithm on different numbers of nodes/processes.
6. **Benchmarking:** Benchmark against a serial implementation to evaluate performance gains.

Deliverables: MPI-based distributed matrix multiplication code, performance metrics, benchmarking report, and detailed documentation

**GitRepo link: https://github.com/rems8505/mpi\_mm.git**

### 1. Environment Setup

A reproducible and lightweight environment was created using:

* Miniconda for Python dependency management
* WSL (Ubuntu) to support native OpenMPI via Linux packages
* OpenMPI and mpi4py for MPI support in Python

#### Installation Steps:

wsl --install # Windows Subsystem for Linux

wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86\_64.sh

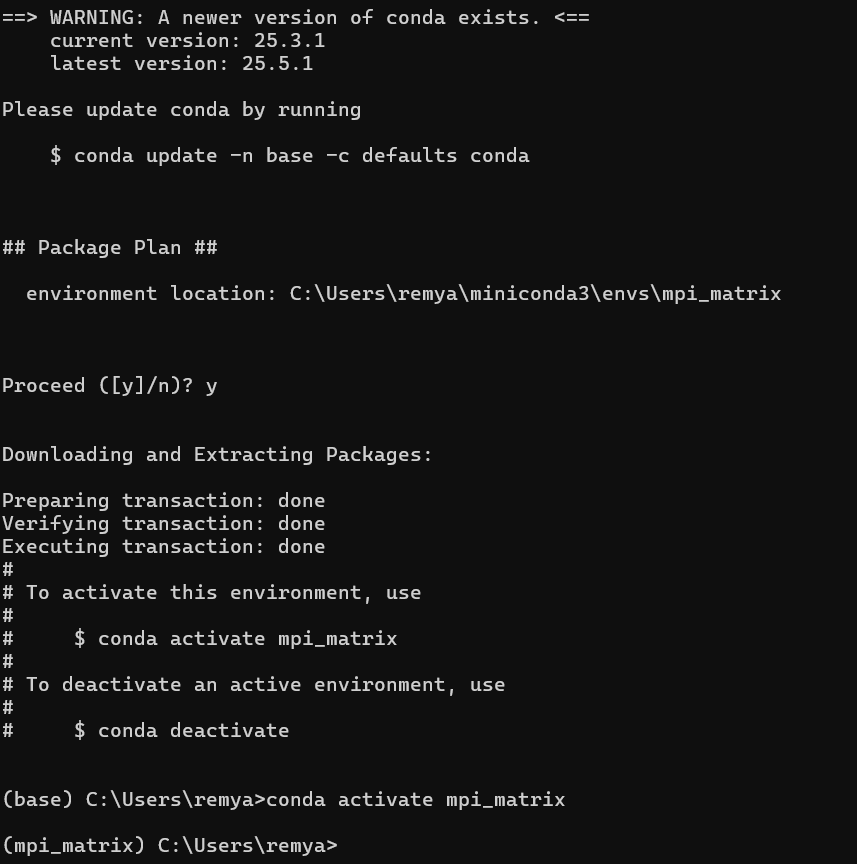
bash Miniconda3-latest-Linux-x86\_64.sh

conda create -n mpi-env python=3.11 -y

conda activate mpi-env

pip install mpi4py psutil matplotlib pandas

sudo apt install openmpi-bin libopenmpi-dev

  
  
  
Executing transaction: done

entry\_point.py:256: DeprecationWarning: Python 3.14 will, by default, filter extracted tar archives and reject files or modify their metadata. Use the filter argument to control this behavior.

installation finished.

Do you wish to update your shell profile to automatically initialize conda?

This will activate conda on startup and change the command prompt when activated.

If you'd prefer that conda's base environment not be activated on startup,

run the following command when conda is activated:

conda config --set auto\_activate\_base false

You can undo this by running `conda init --reverse $SHELL`? [yes|no]

[no] >>> yes

### 2. Matrix Multiplication (Serial)

Implemented in serial\_matrix\_multiplication.py, the program:

* Generates two random N x N matrices (A and B)
* Multiplies them using three nested loops
* Measures execution time using time.time()
* Purpose: Baseline for benchmarking parallel performance.

Key Code:

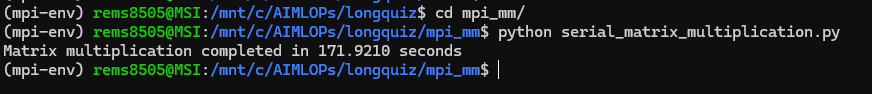
for i in range(N):

for j in range(N):

for k in range(N):

C[i][j] += A[i][k] \* B[k][j]

Multiplication of the matrices with the size of 600x600 took around 172 seconds to comple.



### 3. Distributed Implementation with MPI

Implemented in mpi\_matrix\_multiplication\_metrics.py using mpi4py.

#### Design:

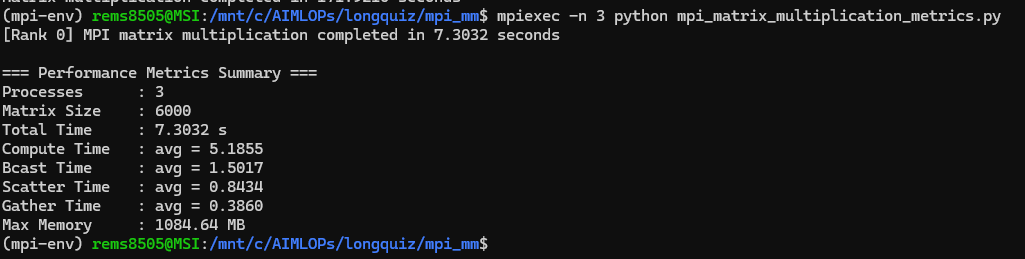
* Matrix A is partitioned row-wise across all ranks
* Matrix B is broadcast to all processes
* Each process computes its partial result (C\_chunk)
* The root process gathers all partial results into full matrix C

#### MPI Operations Used:

* comm.Bcast(B, root=0)
* comm.Scatter(A, A\_chunk, root=0)
* comm.Gather(C\_chunk, C, root=0)

Matrix size N must be divisible by number of processes.

MPI multiplication of matrix **size 6000 x 6000** with mpiexec command with 3 processes took only around ***7 seconds*** to complete.



**5. Scalability Testing**

Metrics collected and logged in metrics\_summary.csv:

| Metric | Description |
| --- | --- |
| TotalTime | Full execution time (wall-clock) |
| ComputeAvg | Average compute time across ranks |
| BcastAvg | Average time spent in Bcast |
| ScatterAvg | Average time spent in Scatter |
| GatherAvg | Average time spent in Gather |
| MaxMemoryMB | Peak memory usage per process (MB) |

#### Tooling:

* time.time() for timing
* psutil for memory usage

### 5. Scalability Testing

Conducted via benchmark\_mpi\_metrics.py, which:

* Runs the MPI version with different process counts: 2, 3, 4
* Captures execution logs and performance metrics
* Appends results to:
  + benchmark\_log.txt
  + benchmark\_metrics.csv

Plots generated using plot\_mpi\_metrics.py:

* Speedup vs #Processes
* Efficiency
* Max Memory Usage

### 6. Benchmarking

The parallel version was benchmarked against the serial implementation to evaluate:

* Speedup:

Speedup=Tserial/Tparallel

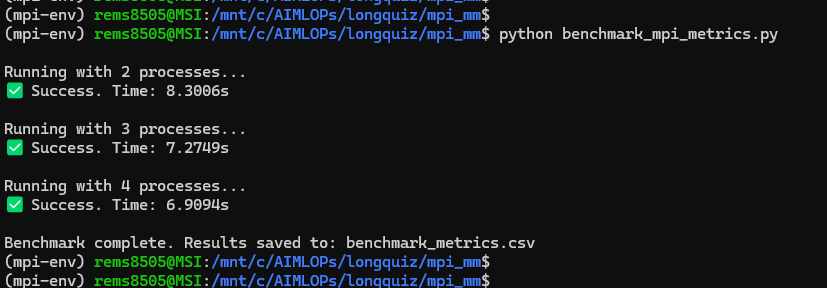
* Efficiency:

Efficiency=Speedup/Number of Processes

Findings:

* For small matrices (N=600), speedup is modest due to communication overhead.
* As matrix size increases (e.g., N=1000+), compute time dominates and speedup improves.
* Efficiency decreases with more processes if workload per process is too small.

benchmark: [2,3,4] processors



**7.Plotting Performance Metrics**

Performance results from benchmarking are visualized using plot\_mpi\_metrics.py. This script reads benchmark\_metrics.csv (generated by the benchmark script) and produces several plots that analyze how the program scales with different numbers of MPI processes.

  
  
  
Generated Plots

#### 1. **Speedup vs Number of Processes**

* Y-axis: Speedup = SerialTime / ParallelTime
* X-axis: Number of MPI processes
* Helps evaluate how effectively the program speeds up with more processes.

#### 2. **Efficiency vs Number of Processes**

* Y-axis: Efficiency = Speedup / Process Count
* Indicates how efficiently the processes are being used.
* Values close to 1 (100%) mean perfect scaling.

#### 3. **Max Memory Usage per Process**

* Tracks the highest memory used by any process (in MB).
* Helps identify if any memory scaling issues occur when increasing processes.

