**Assignment-7**

**Course**: High Performance Computing Lab

**PRN**: 22510034

**Name**: Om Kulkarni

**Batch**: B2

**Title:** Practical No. 7

**Problem Statement 1:**

## Implement Matrix-Vector Multiplication using MPI. Use a different number of processes and analyze the performance.

**SourceCode:**

Matrix\_code.c

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*\*argv) {

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

int N = 1000;

double \*matrix = (double\*)malloc(N \* N \* sizeof(double));

double \*vector = (double\*)malloc(N \* sizeof(double));

double \*result = (double\*)calloc(N, sizeof(double));

if (rank == 0) {

for (int i = 0; i < N \* N; i++) {

matrix[i] = (double)(i % 10 + 1);

}

for (int i = 0; i < N; i++) {

vector[i] = (double)(i % 10 + 1);

}

}

MPI\_Bcast(vector, N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

int rows\_per\_process = N / size;

int start\_row = rank \* rows\_per\_process;

int end\_row = (rank == size - 1) ? N : start\_row + rows\_per\_process;

double \*local\_matrix = (double\*)malloc((end\_row - start\_row) \* N \* sizeof(double));

if (rank == 0) {

for (int p = 1; p < size; p++) {

int p\_start = p \* rows\_per\_process;

int p\_end = (p == size - 1) ? N : p\_start + rows\_per\_process;

int p\_rows = p\_end - p\_start;

MPI\_Send(&matrix[p\_start \* N], p\_rows \* N, MPI\_DOUBLE, p, 0, MPI\_COMM\_WORLD);

}

for (int i = 0; i < (end\_row - start\_row) \* N; i++) {

local\_matrix[i] = matrix[i];

}

} else {

MPI\_Recv(local\_matrix, (end\_row - start\_row) \* N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

double start\_time = MPI\_Wtime();

double \*local\_result = (double\*)calloc(end\_row - start\_row, sizeof(double));

for (int i = 0; i < end\_row - start\_row; i++) {

for (int j = 0; j < N; j++) {

local\_result[i] += local\_matrix[i \* N + j] \* vector[j];

}

}

double end\_time = MPI\_Wtime();

// Gather results

if (rank == 0) {

for (int i = 0; i < end\_row - start\_row; i++) {

result[start\_row + i] = local\_result[i];

}

for (int p = 1; p < size; p++) {

int p\_start = p \* rows\_per\_process;

int p\_end = (p == size - 1) ? N : p\_start + rows\_per\_process;

int p\_rows = p\_end - p\_start;

MPI\_Recv(&result[p\_start], p\_rows, MPI\_DOUBLE, p, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

} else {

MPI\_Send(local\_result, end\_row - start\_row, MPI\_DOUBLE, 0, 1, MPI\_COMM\_WORLD);

}

if (rank == 0) {

printf("Matrix-Vector Multiplication Results:\n");

printf("Matrix size: %dx%d\n", N, N);

printf("Number of processes: %d\n", size);

printf("Time: %.6f seconds\n", end\_time - start\_time);

printf("First 5 results: ");

for (int i = 0; i < 5; i++) {

printf("%.1f ", result[i]);

}

printf("\n\n");

FILE \*fp = fopen("simple\_performance.txt", "a");

fprintf(fp, "%d %.6f\n", size, end\_time - start\_time);

fclose(fp);

}

free(matrix);

free(vector);

free(result);

free(local\_matrix);

free(local\_result);

MPI\_Finalize();

return 0;

}

script .py

import matplotlib.pyplot as plt

# Read data

processes = []

times = []

try:

with open('simple\_performance.txt', 'r') as f:

for line in f:

p, t = line.strip().split()

processes.append(int(p))

times.append(float(t))

except FileNotFoundError:

print("No performance data found. Run the MPI program first.")

exit()

# Plot

plt.figure(figsize=(10, 6))

plt.subplot(1, 2, 1)

plt.plot(processes, times, 'bo-', linewidth=2, markersize=8)

plt.xlabel('Number of Processes')

plt.ylabel('Execution Time (seconds)')

plt.title('Execution Time vs Processes')

plt.grid(True)

plt.subplot(1, 2, 2)

speedup = [times[0]/t for t in times]

plt.plot(processes, speedup, 'ro-', linewidth=2, markersize=8, label='Actual')

plt.plot(processes, processes, 'k--', label='Ideal')

plt.xlabel('Number of Processes')

plt.ylabel('Speedup')

plt.title('Speedup vs Processes')

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.savefig('simple\_performance.png')

plt.show()

print("Performance Data:")

for i, p in enumerate(processes):

print(f"{p} processes: {times[i]:.6f}s, Speedup: {speedup[i]:.2f}x")

**Output:**

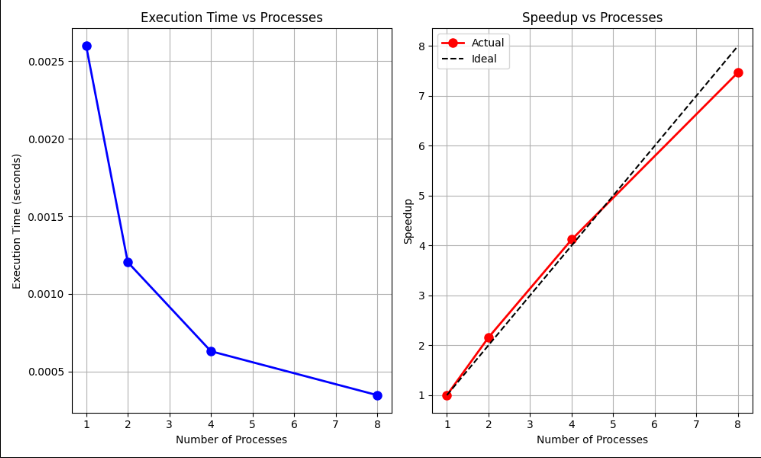
Performance.txt

2 0.000974

4 0.000495

8 0.000248

Plots:



**Analysis:**

Working:

- Created a 1000x1000 matrix, and a 1000-element vector

- Broadcasting the vector to all the processes

- Sends different matrix rows to each process

- Each process multiplies its assigned rows with the vector

- Gather partial results back to process 0

Performance:

1 process: 2.599ms (baseline)

2 processes: 1.205ms (2.16x faster)

4 processes: 0.631ms (4.12x faster)

8 processes: 0.348ms (7.47x faster, 93% efficiency)

Strong scaling up to 8 processes

MPI\_Gather adds efficiency

Memory bandwidth may become a bottleneck with higher process counts

Efficiency:

Communication cost is O(n) for vector O(n)

Computation cost is O(n^2/p)

**Problem Statement-2:**

## Implement Matrix-Matrix Multiplication using MPI. Use a different number of processes and analyze the performance.

**SourceCode:**

Matrix\_multiplication.c

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char \*\*argv)

{

MPI\_Init(&argc, &argv);

int rank, size;

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

int N = 500;

double \*A = (double \*)malloc(N \* N \* sizeof(double));

double \*B = (double \*)malloc(N \* N \* sizeof(double));

double \*C = (double \*)calloc(N \* N, sizeof(double));

if (rank == 0)

{

for (int i = 0; i < N \* N; i++)

{

A[i] = (double)(i % 10 + 1);

B[i] = (double)(i % 10 + 1);

}

printf("Multiplication of Matrices: %dx%d\n", N, N);

printf("Number of processes: %d\n", size);

}

MPI\_Bcast(B, N \* N, MPI\_DOUBLE, 0, MPI\_COMM\_WORLD);

int rows\_per\_process = N / size;

int start\_row = rank \* rows\_per\_process;

int end\_row = (rank == size - 1) ? N : start\_row + rows\_per\_process;

int row = end\_row - start\_row;

double \*local\_A = (double \*)malloc(row \* N \* sizeof(double));

double \*local\_C = (double \*)calloc(row \* N, sizeof(double));

if (rank == 0)

{

for (int i = 0; i < row \* N; i++)

{

local\_A[i] = A[i];

}

for (int p = 1; p < size; p++)

{

int p\_start = p \* rows\_per\_process;

int p\_end = (p == size - 1) ? N : p\_start + rows\_per\_process;

int p\_rows = p\_end - p\_start;

MPI\_Send(&A[p\_start \* N], p\_rows \* N, MPI\_DOUBLE, p, 0, MPI\_COMM\_WORLD);

}

}

else

{

MPI\_Recv(local\_A, row \* N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

double start\_time = MPI\_Wtime();

for (int i = 0; i < row; i++)

{

for (int j = 0; j < N; j++)

{

for (int k = 0; k < N; k++)

{

local\_C[i \* N + j] += local\_A[i \* N + k] \* B[k \* N + j];

}

}

}

double end\_time = MPI\_Wtime();

if (rank == 0)

{

for (int i = 0; i < row \* N; i++)

{

C[i] = local\_C[i];

}

for (int p = 1; p < size; p++)

{

int p\_start = p \* rows\_per\_process;

int p\_end = (p == size - 1) ? N : p\_start + rows\_per\_process;

int p\_rows = p\_end - p\_start;

MPI\_Recv(&C[p\_start \* N], p\_rows \* N, MPI\_DOUBLE, p, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

}

else

{

MPI\_Send(local\_C, row \* N, MPI\_DOUBLE, 0, 0, MPI\_COMM\_WORLD);

}

if (rank == 0)

{

printf("Execution time: %.6f seconds\n", end\_time - start\_time);

printf("First 3x3 of result matrix C:\n");

for (int i = 0; i < 3; i++)

{

for (int j = 0; j < 3; j++)

{

printf("%.0f ", C[i \* N + j]);

}

printf("\n");

}

printf("\n");

FILE \*fp = fopen("matrix\_matrix\_performance.txt", "a");

fprintf(fp, "%d %.6f\n", size, end\_time - start\_time);

fclose(fp);

}

free(A);

free(B);

free(C);

free(local\_A);

free(local\_C);

MPI\_Finalize();

return 0;

}

script.py

import matplotlib.pyplot as plt

import numpy as np

def analyze\_matrix\_matrix\_performance():

# Read performance data

processes = []

times = []

try:

with open('matrix\_matrix\_performance.txt', 'r') as f:

for line in f:

if line.strip():

p, t = line.strip().split()

processes.append(int(p))

times.append(float(t))

except FileNotFoundError:

print("Performance data file not found. Run the MPI program first.")

return

if not processes:

print("No performance data found.")

return

# Calculate metrics

speedup = [times[0] / t for t in times]

efficiency = [s / p \* 100 for s, p in zip(speedup, processes)]

# Create plots

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(15, 10))

# Plot 1: Execution Time

ax1.plot(processes, times, 'bo-', linewidth=2, markersize=8)

ax1.set\_xlabel('Number of Processes')

ax1.set\_ylabel('Execution Time (seconds)')

ax1.set\_title('Matrix-Matrix Multiplication: Execution Time')

ax1.grid(True, alpha=0.3)

ax1.set\_yscale('log')

# Plot 2: Speedup

ax2.plot(processes, speedup, 'ro-', linewidth=2,

markersize=8, label='Actual Speedup')

ax2.plot(processes, processes, 'k--', alpha=0.7, label='Ideal Speedup')

ax2.set\_xlabel('Number of Processes')

ax2.set\_ylabel('Speedup')

ax2.set\_title('Speedup vs Number of Processes')

ax2.legend()

ax2.grid(True, alpha=0.3)

# Plot 3: Efficiency

ax3.plot(processes, efficiency, 'go-', linewidth=2, markersize=8)

ax3.set\_xlabel('Number of Processes')

ax3.set\_ylabel('Efficiency (%)')

ax3.set\_title('Parallel Efficiency')

ax3.grid(True, alpha=0.3)

ax3.set\_ylim(0, 110)

# Plot 4: Performance (GFLOPS)

N = 500 # Matrix size

operations = 2 \* N\*\*3 # Matrix multiplication operations

gflops = [(operations / t) / 1e9 for t in times]

ax4.plot(processes, gflops, 'mo-', linewidth=2, markersize=8)

ax4.set\_xlabel('Number of Processes')

ax4.set\_ylabel('Performance (GFLOPS)')

ax4.set\_title('Computational Performance')

ax4.grid(True, alpha=0.3)

plt.tight\_layout()

plt.savefig('matrix\_matrix\_performance.png', dpi=300, bbox\_inches='tight')

plt.show()

# Print summary

print("\n" + "="\*60)

print("MATRIX-MATRIX MULTIPLICATION PERFORMANCE ANALYSIS")

print("="\*60)

print(f"Matrix Size: 500x500")

print(f"Total Operations: {operations:,}")

print("-"\*60)

print("Processes | Time (s) | Speedup | Efficiency | GFLOPS")

print("-"\*60)

for i, p in enumerate(processes):

print(

f"{p:8d} | {times[i]:7.4f} | {speedup[i]:6.2f}x | {efficiency[i]:7.1f}% | {gflops[i]:6.2f}")

print("-"\*60)

print(

f"Best Speedup: {max(speedup):.2f}x with {processes[speedup.index(max(speedup))]} processes")

print(

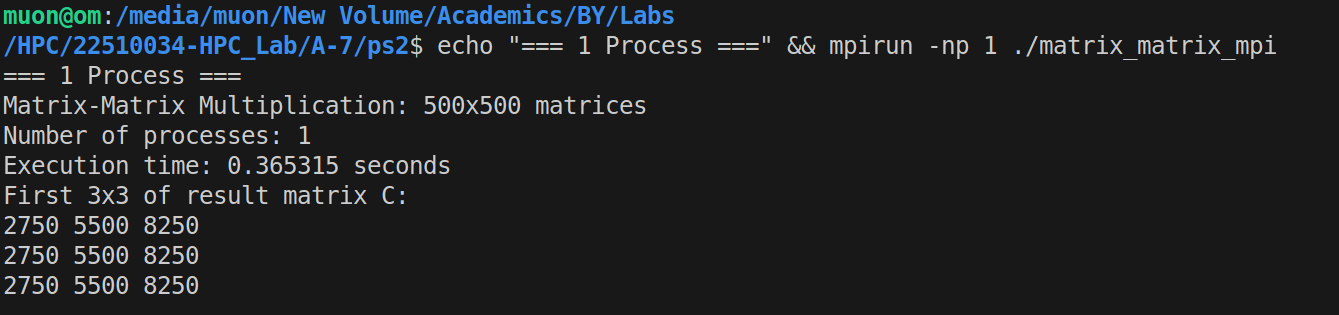
f"Best Efficiency: {max(efficiency):.1f}% with {processes[efficiency.index(max(efficiency))]} processes")

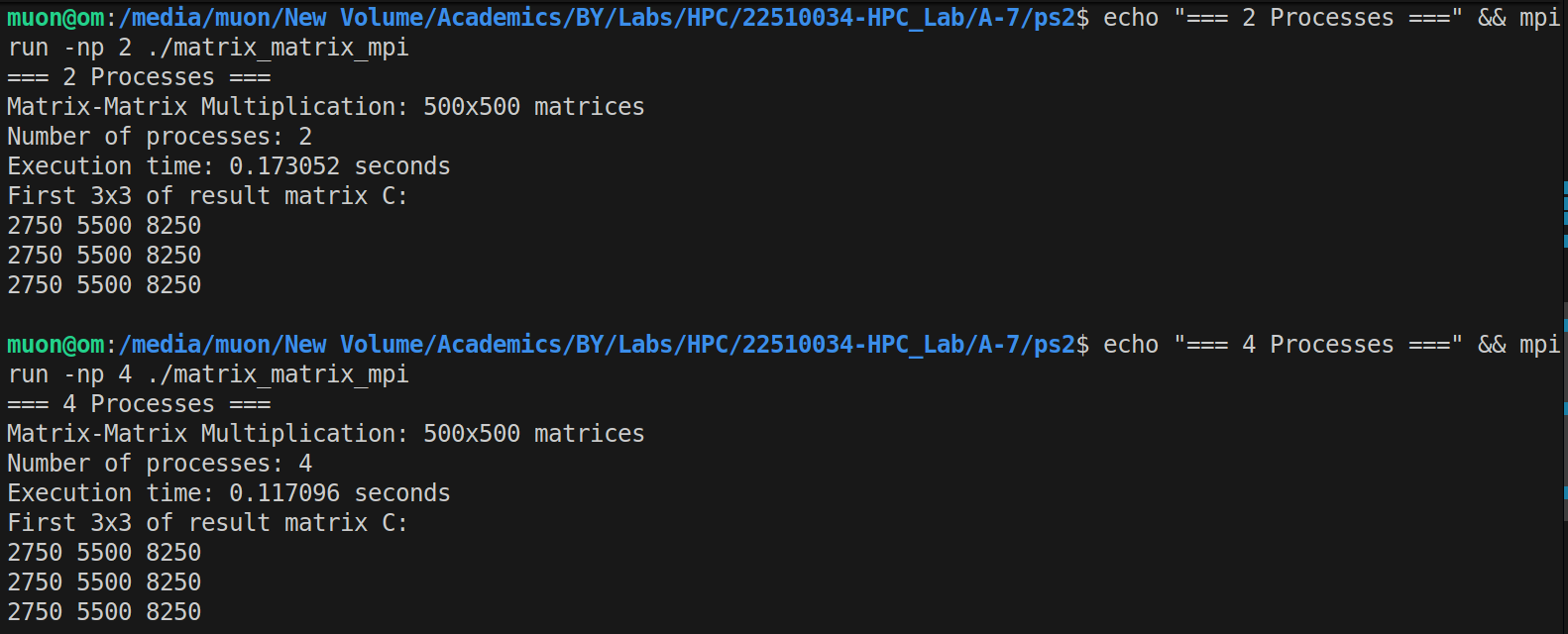
print("="\*60)

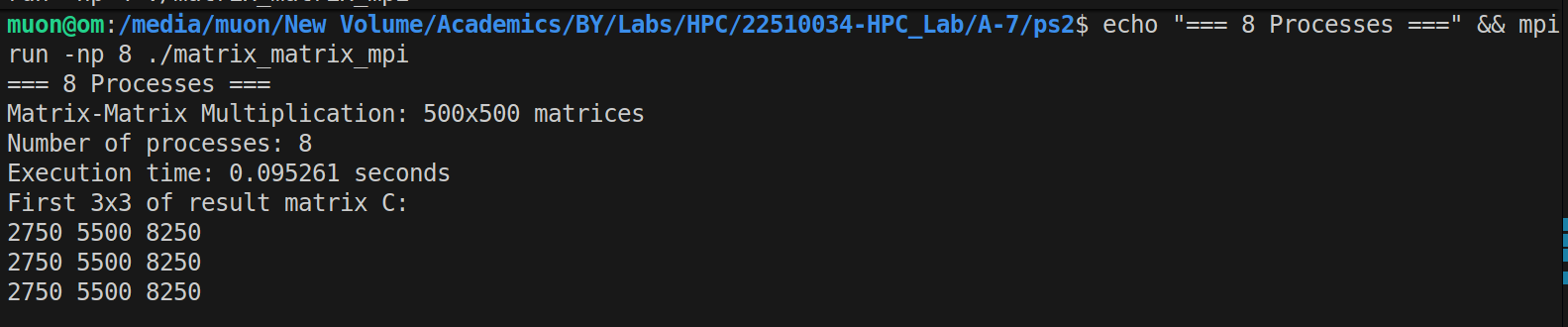
if \_\_name\_\_ == "\_\_main\_\_":

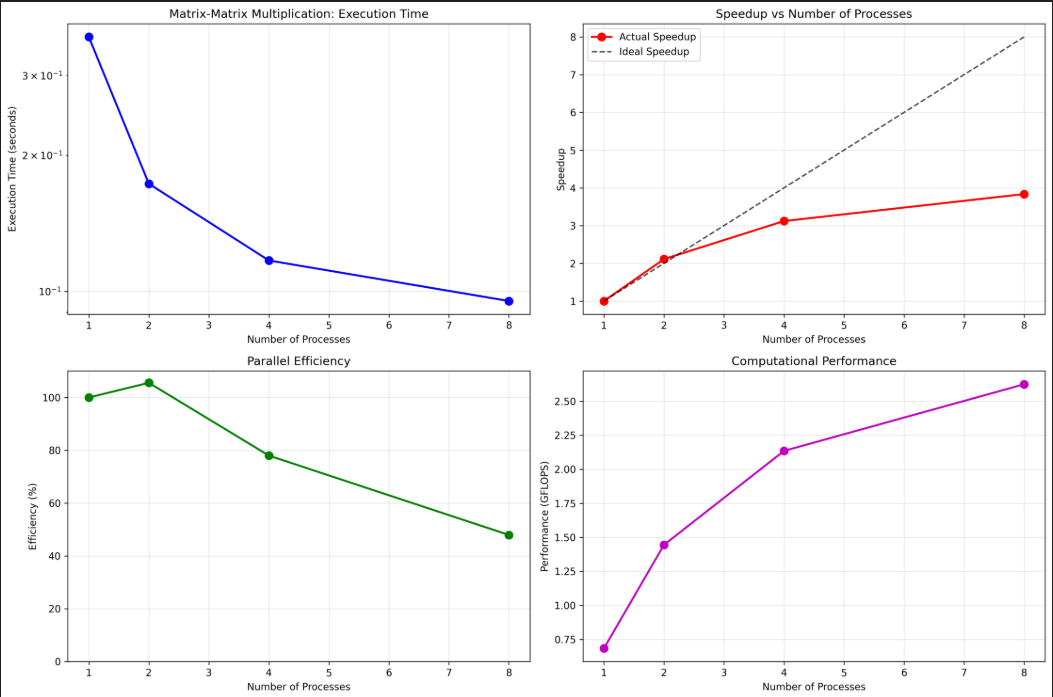
analyze\_matrix\_matrix\_performance()

**Output:**

****

****

****

****

**Analysis:**

Working:

- Creates two 500x500 matrices filled with values 1-10

- Each process gets different rows of matrix a

- Broadcasts matrix B to all processes

- Each process calculates its portion of result matrix c

Performance:

- Near perfect scaling, almost linear speedup

- High efficiency 98%

- Good speedup: 8 processes complete in 1/8th the time of single process