Practical No 9

PRN: 22520005

Name: Aftab Imtiyaj Bhadgaonkar

Batch: B6

Course: High Performance Computing Lab

Title of practical:

Matrix-Matrix and Matrix-Vector Multiplication

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

Code:

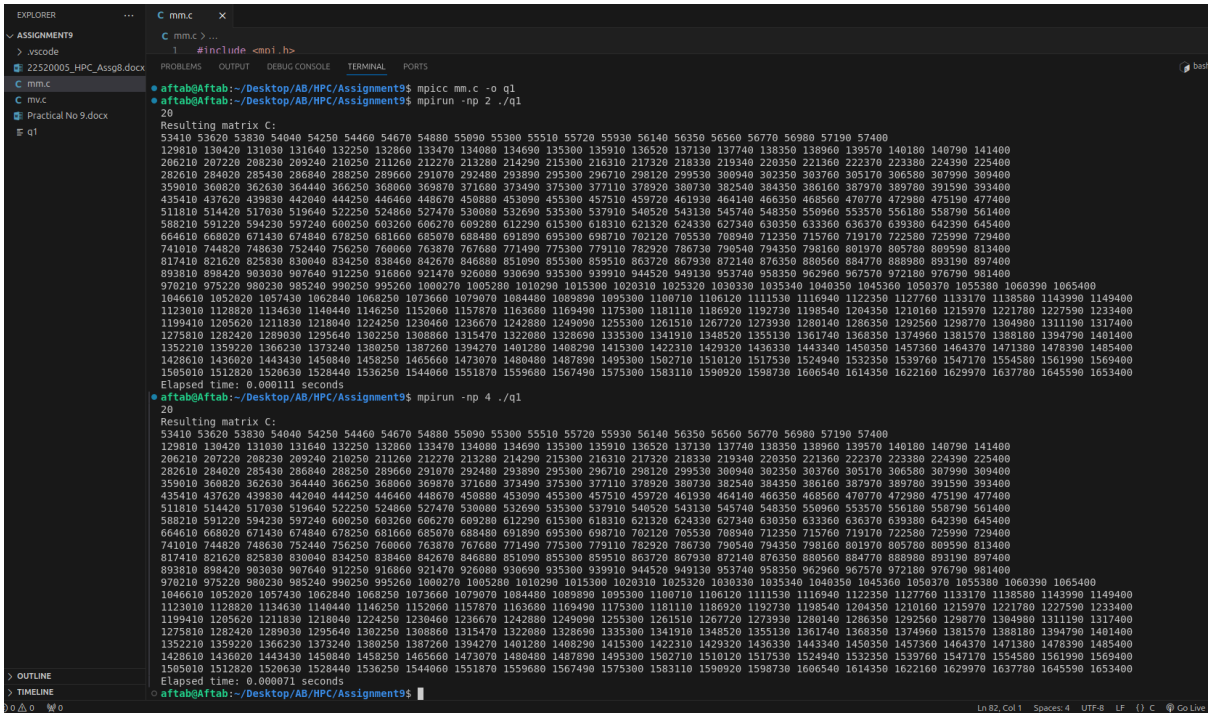
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
MPI_Init(&argc, &argv);

int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int n;
if (world_rank == 0) {
scanf("%d", &n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n % world_size != 0 && world_rank == 0) {
printf("Error: Matrix size must be divisible by the number of processes.\n");
MPI_Abort(MPI_COMM_WORLD, 1);
}

int (*A) = malloc(n * n * sizeof(int));
int (*B) = malloc(n * n * sizeof(int));
int (*C) = malloc(n * n * sizeof(int));
if (world_rank == 0) {
```

```
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
A[i * n + j] = i * n + j + 1;
B[i * n + j] = i * n + j + 1;
}
}
}
double start_time = MPI_Wtime();
int local_rows = n / world_size;
int *local_A = malloc(local_rows * n * sizeof(int));
int *local_C = malloc(local_rows * n * sizeof(int));
MPI_Scatter(A, local_rows * n, MPI_INT, local_A, local_rows * n, MPI_INT, 0,
MPI_COMM_WORLD);
MPI_Bcast(B, n * n, MPI_INT, 0, MPI_COMM_WORLD);
for (int i = 0; i < local_rows; i++) {
for (int j = 0; j < n; j++) {
local_C[i * n + j] = 0;
for (int k = 0; k < n; k++) {
local_C[i * n + j] += local_A[i * n + k] * B[k * n + j];
}
}
}
MPI_Gather(local_C, local_rows * n, MPI_INT, C, local_rows * n, MPI_INT, 0,
MPI_COMM_WORLD);
double end_time = MPI_Wtime();
double elapsed_time = end_time - start_time;
if (world_rank == 0) {
printf("Resulting matrix C:\n");
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
printf("%d ", C[i * n + j]);
}
printf("\n");
}
printf("Elapsed time: %f seconds\n", elapsed_time);
}
free(A);
free(B);
free(C);
free(local_A);
free(local_C);

MPI_Finalize();
return 0;
}
```

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
MPI_Init(&argc, &argv);

int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int n;
if (world_rank == 0) {
scanf("%d", &n);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (n % world_size != 0 && world_rank == 0) {
printf("Error: Matrix size must be divisible by the number of processes.\n");
MPI_Abort(MPI_COMM_WORLD, 1);
}

int (*A) = malloc(n * n * sizeof(int));
int (*B) = malloc(n * n * sizeof(int));
int (*C) = malloc(n * n * sizeof(int));
if (world_rank == 0) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
A[i * n + j] = i * n + j + 1;
B[i * n + j] = i * n + j + 1;
}
}
}

double start_time = MPI_Wtime();

int local_rows = n / world_size;
int *local_A = malloc(local_rows * n * sizeof(int));
int *local_C = malloc(local_rows * n * sizeof(int));

MPI_Scatter(A, local_rows * n, MPI_INT, local_A, local_rows * n, MPI_INT, 0,
MPI_COMM_WORLD);

MPI_Bcast(B, n * n, MPI_INT, 0, MPI_COMM_WORLD);

for (int i = 0; i < local_rows; i++) {
for (int j = 0; j < n; j++) {
```

```
local_C[i * n + j] = 0;
for (int k = 0; k < n; k++) {
local_C[i * n + j] += local_A[i * n + k] * B[k * n + j];
}
}
}

MPI_Gather(local_C, local_rows * n, MPI_INT, C, local_rows * n, MPI_INT, 0,
MPI_COMM_WORLD);

double end_time = MPI_Wtime();
double elapsed_time = end_time - start_time;

if (world_rank == 0) {
printf("Resulting matrix C:\n");
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
printf("%d ", C[i * n + j]);
}
printf("\n");
}
printf("Elapsed time: %f seconds\n", elapsed_time);
}

free(A);
free(B);
free(C);
free(local_A);
free(local_C);

MPI_Finalize();
return 0;
}
```

Output:



Analysis:

Input Size (n)	Cores	Time
20	2	0.000111
	4	0.000071
40	2	0.000230
	4	0.000149

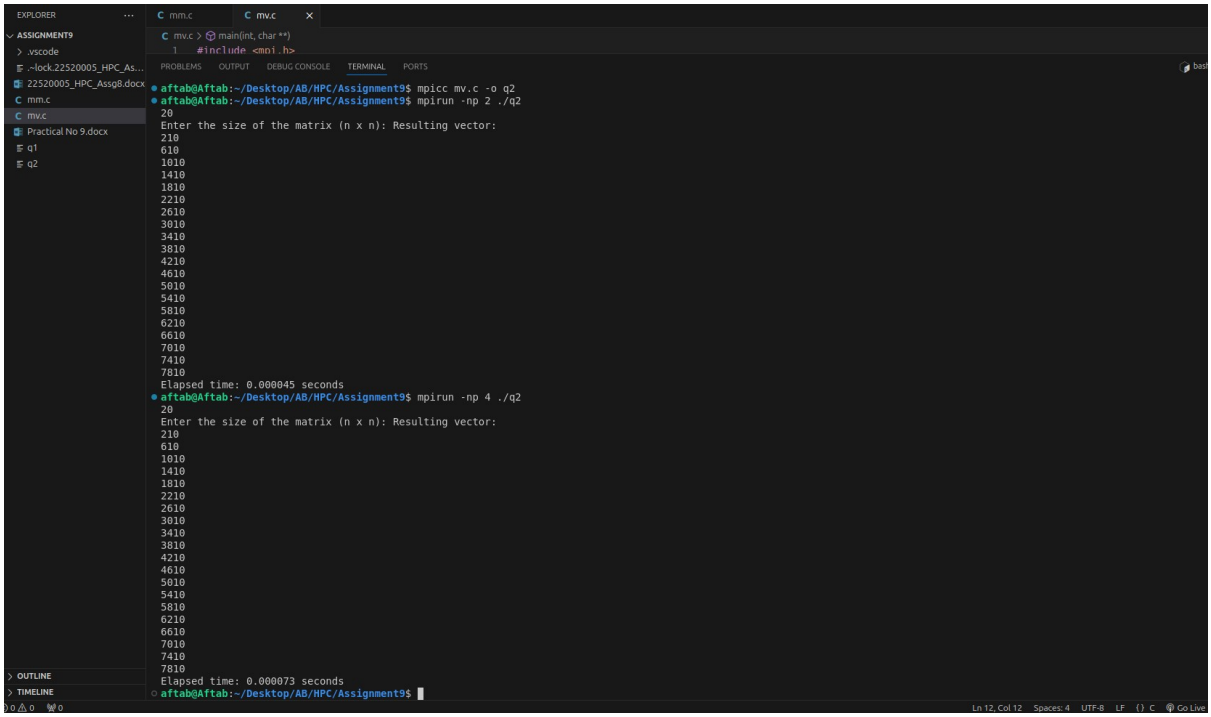
2) Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyze the performance.

Code:

```
#include <mpi.h>

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
MPI_Init(&argc, &argv);

int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int n;
if (world_rank == 0) {
printf("Enter the size of the matrix (n x n): ");
scanf("%d", &n);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

if (n % world_size != 0 && world_rank == 0) {
printf("Error: Matrix size must be divisible by the number of processes.\n");
MPI_Abort(MPI_COMM_WORLD, 1);
}

int* matrix = malloc(n * n * sizeof(int));
int* vector = malloc(n * sizeof(int));
int* result = malloc(n * sizeof(int));
if (world_rank == 0) {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
matrix[i * n + j] = i * n + j + 1;
}
vector[i] = 1;
}
}

double start_time = MPI_Wtime();

int rows_per_process = n / world_size;
int* local_matrix = malloc(rows_per_process * n * sizeof(int));
int* local_result = malloc(rows_per_process * sizeof(int));
```

```
MPI_Scatter(matrix, rows_per_process * n, MPI_INT, local_matrix, rows_per_process * n,
MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(vector, n, MPI_INT, 0, MPI_COMM_WORLD);

for (int i = 0; i < rows_per_process; i++) {
local_result[i] = 0;
for (int j = 0; j < n; j++) {
local_result[i] += local_matrix[i * n + j] * vector[j];
}
}

MPI_Gather(local_result, rows_per_process, MPI_INT, result, rows_per_process, MPI_INT, 0,
MPI_COMM_WORLD);

double end_time = MPI_Wtime();
double elapsed_time = end_time - start_time;

if (world_rank == 0) {
printf("Resulting vector:\n");
for (int i = 0; i < n; i++) {
printf("%d\n", result[i]);
}
printf("Elapsed time: %f seconds\n", elapsed_time);
}

free(matrix);
free(vector);
free(result);
free(local_matrix);
free(local_result);

MPI_Finalize();
return 0;
}
```

Output:



Analysis:

Input Size (n)	Cores	Time
20	2	0.000045
	4	0.000073
40	2	0.000039
	4	0.000073