**Class:** Final Year (Computer Science and Engineering)

**Year:** 2022-23 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 7**

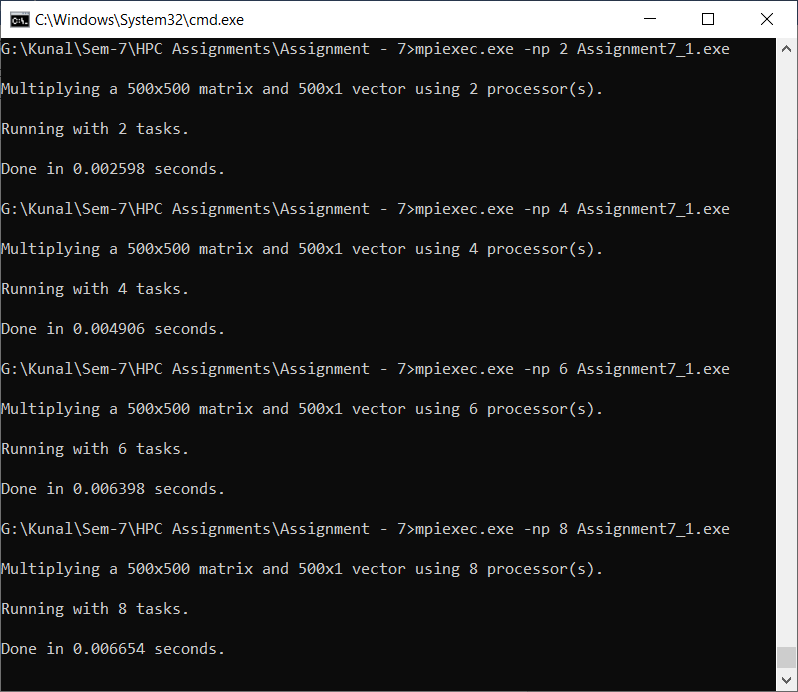
**Exam Seat No: 2019BTECS00064**

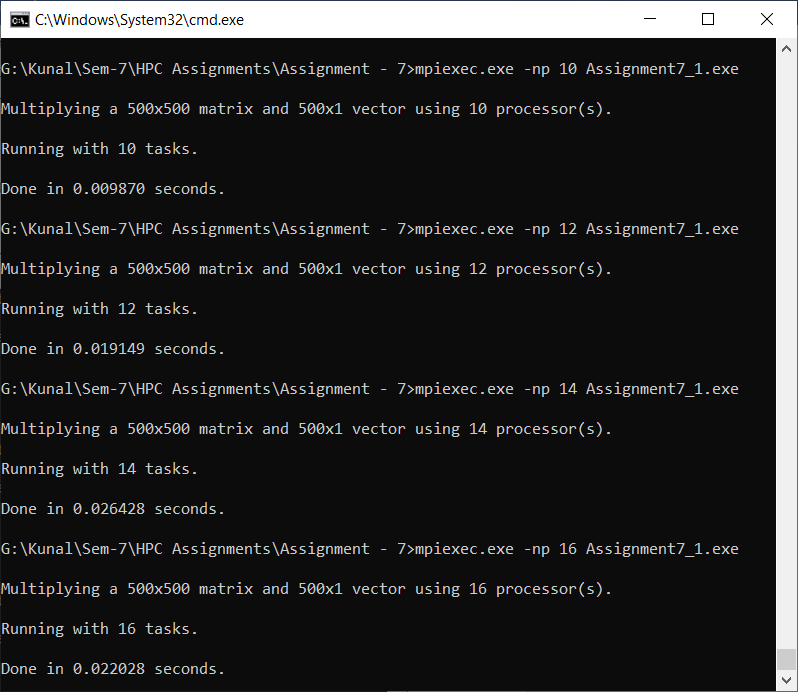
**Name – Kunal Santosh Kadam**

**Problem Statement 1:**

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

**Screenshot #:**





|  |  |
| --- | --- |
| **Processors (Size 500)** | **Execution Time (sec)** |
| 2 | 0.002598 |
| 4 | 0.004906 |
| 6 | 0.006398 |
| 8 | 0.006654 |
| 10 | 0.00987 |
| 12 | 0.019149 |
| 14 | 0.026428 |
| 16 | 0.022028 |

**Information #:**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

// size of matrix

#define N 500

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

{

int np, rank, numworkers, rows, i, j, k;

// a\*b = c

double a[N][N], b[N], c[N];

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

numworkers = np - 1; // total process - 1 ie process with rank 0

// rank with 0 is a master process

int dest, source;

int tag;

int rows\_per\_process, extra, offset;

// master process, process with rank = 0

if (rank == 0)

{

printf("\nMultiplying a %dx%d matrix and %dx1 vector using %d processor(s).\n\n", N, N, N, np);

printf("Running with %d tasks.\n", np);

// matrix a and b initialization

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

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

a[i][j] = 1;

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

b[i] = 1;

// start time

double start = MPI\_Wtime();

// Send matrix data to other worker processes

rows\_per\_process = N / numworkers;

extra = N % numworkers;

offset = 0;

tag = 1;

// send data to other nodes

for (dest = 1; dest <= numworkers; dest++)

{

rows = (dest <= extra) ? rows\_per\_process + 1 : rows\_per\_process;

MPI\_Send(&offset, 1, MPI\_INT, dest, tag, MPI\_COMM\_WORLD);

MPI\_Send(&rows, 1, MPI\_INT, dest, tag, MPI\_COMM\_WORLD);

MPI\_Send(&a[offset][0], rows \* N, MPI\_DOUBLE, dest, tag, MPI\_COMM\_WORLD);

MPI\_Send(&b, N, MPI\_DOUBLE, dest, tag, MPI\_COMM\_WORLD);

offset = offset + rows;

}

// receive data from other nodes and add it to the ans matrix c

tag = 2;

for (i = 1; i <= numworkers; i++)

{

source = i;

MPI\_Recv(&offset, 1, MPI\_INT, source, tag, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, source, tag, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&c[offset], N, MPI\_DOUBLE, source, tag, MPI\_COMM\_WORLD, &status);

}

// print multiplication result

// printf("Result Matrix:\n");

// for (i = 0; i < N; i++)

// {

// printf("%6.2f ", c[i]);

// }

printf("\n");

double finish = MPI\_Wtime();

printf("Done in %f seconds.\n", finish - start); // total time spent

}

// all other process than process with rank = 0

if (rank > 0)

{

tag = 1;

// receive data from process with rank 0

MPI\_Recv(&offset, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&a, rows \* N, MPI\_DOUBLE, 0, tag, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&b, N, MPI\_DOUBLE, 0, tag, MPI\_COMM\_WORLD, &status);

// calculate multiplication of given rows

for (i = 0; i < rows; i++)

{

c[i] = 0.0;

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

c[i] = c[i] + a[i][j] \* b[j];

}

// send result back to process with rank 0

tag = 2;

MPI\_Send(&offset, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD);

MPI\_Send(&rows, 1, MPI\_INT, 0, tag, MPI\_COMM\_WORLD);

MPI\_Send(&c, N, MPI\_DOUBLE, 0, tag, MPI\_COMM\_WORLD);

}

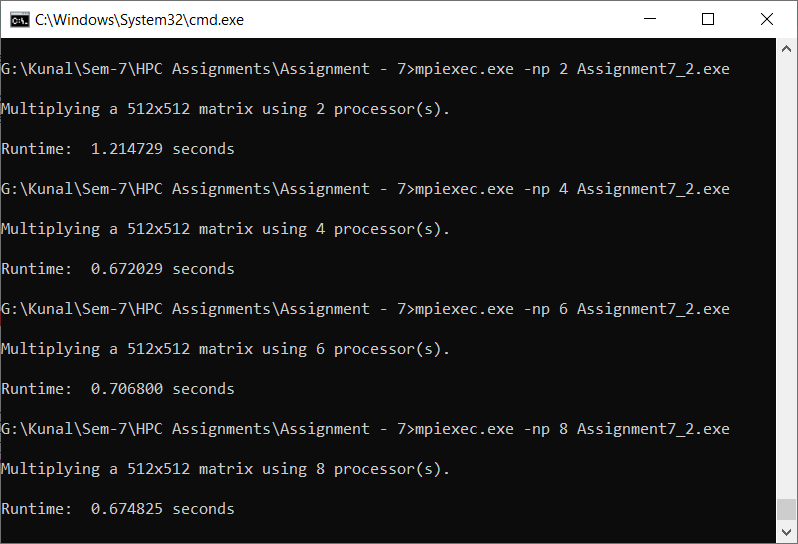
MPI\_Finalize();

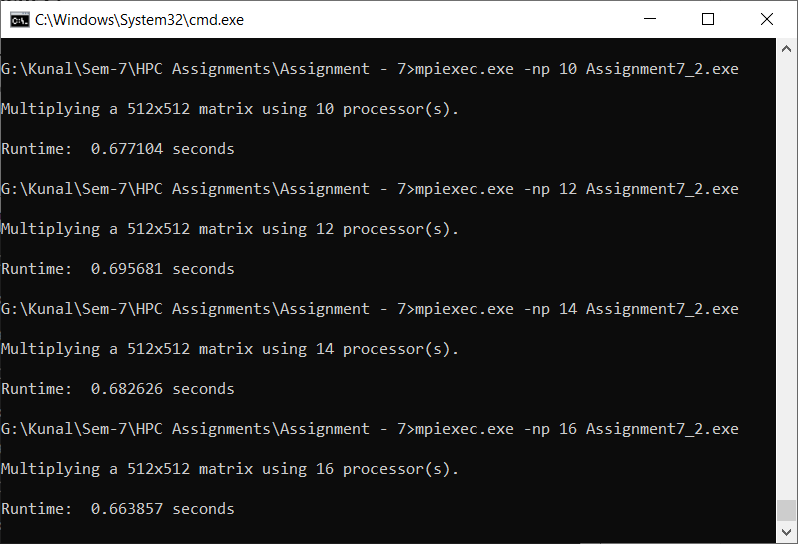
}

**Problem Statement 2:**

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

**Screenshot #:**





|  |  |
| --- | --- |
| **Processors (Size 512)** | **Execution Time (sec)** |
| 2 | 1.214726 |
| 4 | 0.672029 |
| 6 | 0.7068 |
| 8 | 0.674825 |
| 10 | 0.677104 |
| 12 | 0.695681 |
| 14 | 0.682626 |
| 16 | 0.663857 |

**Information #:**

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include <stdbool.h>

#include <mpi.h>

// Size of the matrix (NxN)

#define N 512

MPI\_Status status;

// Define matrices

int matrix1[N][N];

int matrix2[N][N];

int productMatrix[N][N];

// Counter variables

int i, j, k;

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

{

int numberOfProcessors;

int processorRank;

int numberOfWorkers;

// Processor sending data

int sourceProcessor;

// Processor to receive data

int destinationProcessor;

// The number of rows for a worker processor to process

int rows;

// The subset of a matrix to be processed by workers

int matrixSubset;

// Initialize MPI environment

MPI\_Init(&argc, &argv);

// Determine number of processors available

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

// Determine rank of calling process

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

numberOfWorkers = numberOfProcessors - 1;

double stime,etime;

/\* ---------- Manager Processor Code ---------- \*/

if (processorRank == 0)

{

// Initialize a timer

stime = MPI\_Wtime();

printf("\nMultiplying a %dx%d matrix using %d processor(s).\n\n", N, N, numberOfProcessors);

// Populate the matrices with values

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

{

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

{

matrix1[i][j] = (rand() % 6) + 1;

matrix2[i][j] = (rand() % 6) + 1;

}

}

/\* Send the matrix to the worker processes \*/

rows = N / numberOfWorkers;

matrixSubset = 0;

// Iterate through all of the workers and assign work

for (destinationProcessor = 1; destinationProcessor <= numberOfWorkers; destinationProcessor++)

{

// Determine the subset of the matrix to send to the destination processor

MPI\_Send(&matrixSubset, 1, MPI\_INT, destinationProcessor, 1, MPI\_COMM\_WORLD);

// Send the number of rows to process to the destination worker processor

MPI\_Send(&rows, 1, MPI\_INT, destinationProcessor, 1, MPI\_COMM\_WORLD);

// Send rows from matrix 1 to destination worker processor

MPI\_Send(&matrix1[matrixSubset][0], rows \* N, MPI\_INT, destinationProcessor, 1, MPI\_COMM\_WORLD);

// Send entire matrix 2 to destination worker processor

MPI\_Send(&matrix2, N \* N, MPI\_INT, destinationProcessor, 1, MPI\_COMM\_WORLD);

// Determine the next chunk of data to send to the next processor

matrixSubset = matrixSubset + rows;

}

// Retrieve results from all workers processors

for (i = 1; i <= numberOfWorkers; i++)

{

sourceProcessor = i;

MPI\_Recv(&matrixSubset, 1, MPI\_INT, sourceProcessor, 2, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, sourceProcessor, 2, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&productMatrix[matrixSubset][0], rows \* N, MPI\_INT, sourceProcessor, 2, MPI\_COMM\_WORLD, &status);

}

// Stop the timer

etime = MPI\_Wtime();

// Determine and print the total run time

printf("Runtime: %f seconds\n",etime-stime);

}

/\* ---------- Worker Processor Code ---------- \*/

if (processorRank > 0)

{

sourceProcessor = 0;

MPI\_Recv(&matrixSubset, 1, MPI\_INT, sourceProcessor, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, sourceProcessor, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&matrix1, rows \* N, MPI\_INT, sourceProcessor, 1, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&matrix2, N \* N, MPI\_INT, sourceProcessor, 1, MPI\_COMM\_WORLD, &status);

/\* Perform matrix multiplication \*/

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

{

for (i = 0; i < rows; i++)

{

productMatrix[i][k] = 0.0;

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

{

productMatrix[i][k] = productMatrix[i][k] + matrix1[i][j] \* matrix2[j][k];

}

}

}

MPI\_Send(&matrixSubset, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

MPI\_Send(&rows, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

MPI\_Send(&productMatrix, rows \* N, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

}

**GitHub Link:**