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

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 11**

**Name:** Sumit Narake

**PRN:**2020BTECS00023

**Batch :** B2

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

**Code:**

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

// size of matrix

#define N 100

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("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);

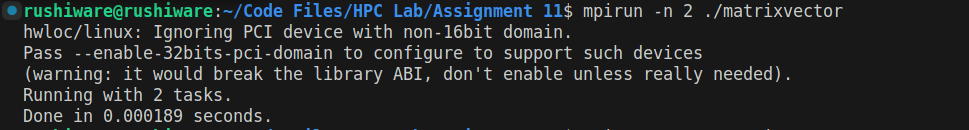
    }

    MPI\_Finalize();

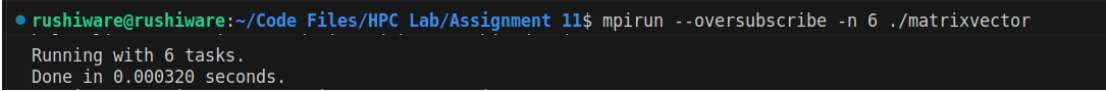
}

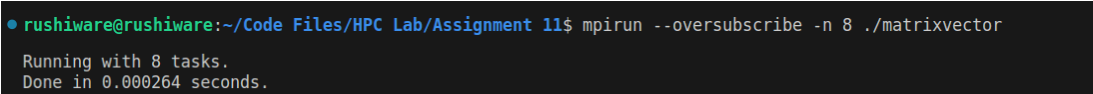
**Output Screenshot:**





## 





## 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>

#define SIZE 4 /\* Size of matrices \*/

int A[SIZE][SIZE], B[SIZE][SIZE], C[SIZE][SIZE];

void fill\_matrix(int m[SIZE][SIZE])

{

    static int n = 1;

    int i, j;

    for (i = 0; i < SIZE; i++)

        for (j = 0; j < SIZE; j++)

            m[i][j] = n++;

}

void print\_matrix(int m[SIZE][SIZE])

{

    int i, j = 0;

    for (i = 0; i < SIZE; i++) {

        printf("\n\t| ");

        for (j = 0; j < SIZE; j++)

            printf("%2d ", m[i][j]);

        printf("|");

    }

}

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

{

    int myrank, P, from, to, i, j, k;

    int tag = 666; /\* any value will do \*/

    MPI\_Status status;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myrank); /\* who am i \*/

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &P);      /\* number of processors \*/

    /\* Just to use the simple variants of MPI\_Gather and MPI\_Scatter we \*/

    /\* impose that SIZE is divisible by P. By using the vector versions, \*/

    /\* (MPI\_Gatherv and MPI\_Scatterv) it is easy to drop this restriction. \*/

    if (SIZE % P != 0) {

        if (myrank == 0)

            printf("Matrix size not divisible by number of processors\n");

        MPI\_Finalize();

        exit(-1);

    }

    from = myrank \* SIZE / P;

    to = (myrank + 1) \* SIZE / P;

    /\* Process 0 fills the input matrices and broadcasts them to the rest \*/

    /\* (actually, only the relevant stripe of A is sent to each process) \*/

    if (myrank == 0) {

        fill\_matrix(A);

        fill\_matrix(B);

    }

    double start = MPI\_Wtime();

    MPI\_Bcast(B, SIZE \* SIZE, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Scatter(A[to], SIZE \* SIZE / P, MPI\_INT, A[from], SIZE \* SIZE / P, MPI\_INT, 0, MPI\_COMM\_WORLD);

    printf("computing slice %d (from row %d to %d)\n", myrank, from, to - 1);

    for (i = from; i < to; i++)

        for (j = 0; j < SIZE; j++) {

            C[i][j] = 0;

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

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

        }

    MPI\_Gather(C[from], SIZE \* SIZE / P, MPI\_INT, C[to], SIZE \* SIZE / P, MPI\_INT, 0, MPI\_COMM\_WORLD);

    if (myrank == 0) {

        double finish = MPI\_Wtime();

        // printf("\n\n");

        // print\_matrix(A);

        // printf("\n\n\t   \* \n");

        // print\_matrix(B);

        // printf("\n\n\t   = \n");

        // print\_matrix(C);

        // printf("\n\n");

        printf("Exection Time: %f\n", finish - start);

    }

    MPI\_Finalize();

    return 0;

}

**Output Screenshots:**



