<u>**Final Year B. Tech., Sem VII 2024-25**</u>

High_Performance_Computing_Lab

Practical No. 9

PRN : 21510111

Name : Siddharth Salunkhe

---

1. **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>
#include <time.h>

#define N 4  // Matrix size (NxN)

void matrix_vector_multiply(int rank, int size, int n, double*
A, double* x, double* local_result) {
   int local_rows = n / size;  // Number of rows for each
process
   for (int i = 0; i < local_rows; i++) {
       local_result[i] = 0;
       for (int j = 0; j < n; j++) {
           local_result[i] += A[i * n + j] * x[j];
       }
   }
}

// Sequential version of matrix-vector multiplication for
single-process time
void sequential_matrix_vector_multiply(int n, double* A, double*
```

```
x, double* result) {
    for (int i = 0; i < n; i++) {
        result[i] = 0;
        for (int j = 0; j < n; j++) {
            result[i] += A[i * n + j] * x[j];
        }
    }
}

int main(int argc, char** argv) {
    int rank, size;
    int n = N;  // Matrix size
    double *A = NULL, *x = NULL, *result = NULL;
    double *local_A, *local_result;
    int local_rows;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    local_rows = n / size;  // Number of rows per process

    // Allocate memory for local matrices and vectors
    local_A = (double*) malloc(local_rows * n * sizeof(double));
    local_result = (double*) malloc(local_rows * sizeof(double));

    // Master process initializes the matrix and vector
    if (rank == 0) {
        A = (double*) malloc(n * n * sizeof(double));
        x = (double*) malloc(n * sizeof(double));
        result = (double*) malloc(n * sizeof(double));

        // Initialize matrix A and vector x with random values
        srand(time(0));
        for (int i = 0; i < n * n; i++) {
            A[i] = rand() % 10;
```

```
        }
        for (int i = 0; i < n; i++) {
            x[i] = rand() % 10;
        }

        printf("Matrix A:\n");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                printf("%6.2f ", A[i * n + j]);
            }
            printf("\n");
        }

        printf("\nVector x:\n");
        for (int i = 0; i < n; i++) {
            printf("%6.2f\n", x[i]);
        }
    }

    // Sequential execution to calculate T1
    double T1_start, T1_end, T1;
    if (rank == 0) {
        T1_start = MPI_Wtime();  // Start timing for sequential
execution
        double* seq_result = (double*) malloc(n *
sizeof(double));
        sequential_matrix_vector_multiply(n, A, x, seq_result);
        T1_end = MPI_Wtime();  // End timing for sequential
execution
        T1 = T1_end - T1_start;

        printf("\nSequential Result vector:\n");
        for (int i = 0; i < n; i++) {
            printf("%6.2f\n", seq_result[i]);
        }
        free(seq_result);
```

```
    }

    // Scatter the matrix rows to all processes
    MPI_Scatter(A, local_rows * n, MPI_DOUBLE, local_A,
local_rows * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Broadcast the vector to all processes
    if (rank == 0) {
        MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    } else {
        x = (double*) malloc(n * sizeof(double));
        MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    // Parallel execution to calculate Tp
    double Tp_start, Tp_end, Tp;
    Tp_start = MPI_Wtime();  // Start timing for parallel
execution

    // Perform local matrix-vector multiplication
    matrix_vector_multiply(rank, size, n, local_A, x,
local_result);

    // Gather the local results into the global result
    MPI_Gather(local_result, local_rows, MPI_DOUBLE, result,
local_rows, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    Tp_end = MPI_Wtime();  // End timing for parallel execution
    Tp = Tp_end - Tp_start;

    // Master process prints the parallel result and calculates
speedup
    if (rank == 0) {
        printf("\nParallel Result vector:\n");
        for (int i = 0; i < n; i++) {
            printf("%6.2f\n", result[i]);
```

```c
        }

        // Calculate speedup: Speedup = T1 / Tp
        double speedup = T1 / Tp;
        printf("\nTime (Sequential - T1): %f seconds\n", T1);
        printf("Time (Parallel - Tp): %f seconds\n", Tp);
        printf("Speedup: %f\n", speedup);
    }

    // Clean up
    free(local_A);
    free(local_result);
    if (rank == 0) {
        free(A);
        free(x);
        free(result);
    } else {
        free(x);
    }

    MPI_Finalize();
    return 0;
}
```

Output :

```
● *[main][~/acad/hpc-lab/as9]$ mpirun --oversubscribe -np 4 ./1
Matrix A:
  2.00   7.00   1.00   1.00
  3.00   4.00   2.00   4.00
  8.00   7.00   6.00   1.00
  3.00   8.00   2.00   5.00

Vector x:
  5.00
  9.00
  5.00
  9.00

Sequential Result vector:
 87.00
 97.00
142.00
142.00

Parallel Result vector:
 87.00
 97.00
142.00
142.00

Time (Sequential - T1): 0.000001 seconds
Time (Parallel - Tp): 0.000055 seconds
Speedup: 0.011074
```

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>
#include <time.h>

#define N 4   // Matrix size (NxN)

void matrix_multiply(int rank, int size, int n, double* A,
double* B, double* local_C) {
   int local_rows = n / size;   // Number of rows assigned to
each process

   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] += A[i * n + k] * B[k * n +
j];
            }
        }
    }
}

// Sequential version for time comparison
void sequential_matrix_multiply(int n, double* A, double* B,
double* C) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            C[i * n + j] = 0;
            for (int k = 0; k < n; k++) {
                C[i * n + j] += A[i * n + k] * B[k * n + j];
            }
        }
    }
}

int main(int argc, char** argv) {
    int rank, size;
    int n = N;  // Matrix size
    double *A = NULL, *B = NULL, *C = NULL;
    double *local_A, *local_C;
    int local_rows;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    local_rows = n / size;  // Number of rows per process

    // Allocate memory for local matrices
```

```c
local_A = (double*) malloc(local_rows * n * sizeof(double));
local_C = (double*) malloc(local_rows * n * sizeof(double));

// Master process initializes matrices A and B
if (rank == 0) {
    A = (double*) malloc(n * n * sizeof(double));
    B = (double*) malloc(n * n * sizeof(double));
    C = (double*) malloc(n * n * sizeof(double));

    // Initialize matrix A and B with random values
    srand(time(0));
    for (int i = 0; i < n * n; i++) {
        A[i] = rand() % 10;
        B[i] = rand() % 10;
    }

    printf("Matrix A:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%6.2f ", A[i * n + j]);
        }
        printf("\n");
    }

    printf("\nMatrix B:\n");
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            printf("%6.2f ", B[i * n + j]);
        }
        printf("\n");
    }
}

// Broadcast matrix B to all processes
if (rank == 0) {
    MPI_Bcast(B, n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

```
    } else {
        B = (double*) malloc(n * n * sizeof(double));
        MPI_Bcast(B, n * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }

    // Scatter rows of matrix A to all processes
    MPI_Scatter(A, local_rows * n, MPI_DOUBLE, local_A,
local_rows * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Perform local matrix multiplication
    double start_time = MPI_Wtime();  // Start parallel time
    matrix_multiply(rank, size, n, local_A, B, local_C);
    double end_time = MPI_Wtime();     // End parallel time

    // Gather the results back into matrix C in the master
process
    MPI_Gather(local_C, local_rows * n, MPI_DOUBLE, C, local_rows
* n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    // Master process prints the final matrix C
    if (rank == 0) {
        printf("\nMatrix C (A * B):\n");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                printf("%6.2f ", C[i * n + j]);
            }
            printf("\n");
        }
        printf("\nParallel execution time: %f seconds\n",
end_time - start_time);
    }

    // Sequential execution to calculate T1 for speedup
comparison
    if (rank == 0) {
        double* seq_C = (double*) malloc(n * n * sizeof(double));
```

```
        double seq_start_time = MPI_Wtime();
        sequential_matrix_multiply(n, A, B, seq_C);
        double seq_end_time = MPI_Wtime();
        double sequential_time = seq_end_time - seq_start_time;

        printf("\nSequential execution time: %f seconds\n",
sequential_time);

        // Speedup calculation
        double speedup = sequential_time / (end_time -
start_time);
        printf("\nSpeedup: %f\n", speedup);

        free(seq_C);
    }

    // Clean up
    free(local_A);
    free(local_C);
    if (rank == 0) {
        free(A);
        free(B);
        free(C);
    } else {
        free(B);
    }

    MPI_Finalize();
    return 0;
}
```

Output :

```
● *[main][~/acad/hpc-lab/as9]$ mpirun --oversubscribe -np 4 ./2
  Matrix A:
    4.00   8.00   2.00   9.00
    1.00   9.00   3.00   6.00
    9.00   5.00   6.00   5.00
    5.00   9.00   1.00   7.00

  Matrix B:
    1.00   3.00   0.00   9.00
    4.00   4.00   4.00   5.00
    4.00   4.00   0.00   1.00
    9.00   4.00   7.00   8.00

  Matrix C (A * B):
  125.00  88.00  95.00 150.00
  103.00  75.00  78.00 105.00
   98.00  91.00  55.00 152.00
  108.00  83.00  85.00 147.00

  Parallel execution time: 0.000001 seconds

  Sequential execution time: 0.000001 seconds

  Speedup: 1.980315
```