Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

Year: 2025-26 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 7

Exam Seat No: 22510064 – Parshwa Herwade

Github Link: [Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub]()

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

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

int main(int argc, char* argv[]) {
    int rank, size;
    int n;
    int *matrix = NULL, *vector = NULL, *result = NULL;
    int *local_matrix, *local_result;
    int rows_per_proc;

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

    if (rank == 0) {
        printf("Enter size of square matrix (n): ");
        fflush(stdout);
        scanf("%d", &n);
    }

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

    if (n % size != 0) {
        if (rank == 0) {
```

```
            printf("Error: n (%d) must be divisible by number of processes (%d)\n", n, size);
        }
        MPI_Finalize();
        return 0;
    }

    rows_per_proc = n / size;

    if (rank == 0) {
        matrix = (int*)malloc(n * n * sizeof(int));
        vector = (int*)malloc(n * sizeof(int));
        result = (int*)malloc(n * sizeof(int));

        printf("Enter matrix (%d x %d):\n", n, n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                scanf("%d", &matrix[i * n + j]);

        printf("Enter vector (%d elements):\n", n);
        for (int i = 0; i < n; i++)
            scanf("%d", &vector[i]);
    }

    local_matrix = (int*)malloc(rows_per_proc * n * sizeof(int));
    local_result = (int*)malloc(rows_per_proc * sizeof(int));
    if (rank != 0) vector = (int*)malloc(n * sizeof(int));

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

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

    for (int i = 0; i < rows_per_proc; 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_proc, MPI_INT,
               result, rows_per_proc, MPI_INT,
               0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Result vector:\n");
        for (int i = 0; i < n; i++)
            printf("%d ", result[i]);
        printf("\n");
    }

    if (rank == 0) { free(matrix); free(vector); free(result); }
    else free(vector);
    free(local_matrix);
    free(local_result);

    MPI_Finalize();
    return 0;
}

```

OUTPUT:

```
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpicc matrix_vector.c -o matrix_vector
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpirun -np 4 ./matrix_vector
Enter size of square matrix (n): 3
Error: n (3) must be divisible by number of processes (4)
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpicc matrix_vector.c -o matrix_vector
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpirun -np 4 ./matrix_vector
Enter size of square matrix (n): 2
Error: n (2) must be divisible by number of processes (4)
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpicc matrix_vector.c -o matrix_vector
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpirun -np 2 ./matrix_vector
Enter size of square matrix (n): 4
Enter matrix (4 x 4):
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
Enter vector (4 elements):
5 6 7 8
Result vector:
70 96 122 148
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpirun -np 2 ./matrix_vector
Enter size of square matrix (n): 2
Enter matrix (2 x 2):
1 2
3 4
Enter vector (2 elements):
4 5
Result vector:
14 32
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$
```

Algorithm

1. Initialize the MPI environment.
2. Process 0 (root) takes the size n, the matrix A, and the vector x as input.
3. The rows of matrix A are divided among the processes (block row distribution).
    - Each process gets n/p rows (if n divisible by p).
4. The vector x is broadcast to all processes.
5. Each process computes its partial product:

$$y_i = \sum_{j=0}^{n-1} A_{ij} \cdot x_j$$

for its assigned rows.

6. The partial results are gathered at the root process using MPI_Gather.
7. Root process prints the result vector.
8. Finalize MPI.

Observations (Sample Outputs)

- Execution time decreases as the number of processes increases (for large matrices).
- For small n, communication overhead may dominate, giving no real speedup.

Conclusion

- Matrix–vector multiplication parallelizes well because rows can be distributed independently.
- Speedup is noticeable for larger matrices.
- For small matrices, MPI overhead reduces efficiency.

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

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

```

int main(int argc, char* argv[]) {
    int rank, size;
    int n;
    int *A = NULL, *B = NULL, *C = NULL;
    int *local_A, *local_C;
    int rows_per_proc;

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

    if (rank == 0) {
        printf("Enter size of square matrices (n): ");
        fflush(stdout);
        scanf("%d", &n);
    }

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

    if (n % size != 0) {
        if (rank == 0) {
            printf("Error: n (%d) must be divisible by number of processes (%d)\n", n, size);
        }
        MPI_Finalize();
        return 0;
    }

    rows_per_proc = n / size;

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

        printf("Enter matrix A (%d x %d):\n", n, n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
```

```c
                scanf("%d", &A[i * n + j]);

        printf("Enter matrix B (%d x %d):\n", n, n);
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                scanf("%d", &B[i * n + j]);
    }

    local_A = (int*)malloc(rows_per_proc * n * sizeof(int));
    local_C = (int*)malloc(rows_per_proc * n * sizeof(int));
    if (rank != 0) B = (int*)malloc(n * n * sizeof(int));

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

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

    for (int i = 0; i < rows_per_proc; 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, rows_per_proc * n, MPI_INT,
               C, rows_per_proc * n, MPI_INT,
               0, MPI_COMM_WORLD);

    if (rank == 0) {
        printf("Result 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");
        }
```

```c
    }

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

    MPI_Finalize();
    return 0;
}

```

OUTPUT:

```
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpicc matrix_matrix.c -o matrix_matrix
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpirun -np 2 ./matrix_matrix
Enter size of square matrices (n): 2
Enter matrix A (2 x 2):
1 2
3 4
Enter matrix B (2 x 2):
5 6
7 8
Result matrix C:
19 22
43 50
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$ mpirun -np 2 ./matrix_matrix
Enter size of square matrices (n): 4
Enter matrix A (4 x 4):
1 2 3 4
2 3 4 5
5 6 7 8
6 7 8 9
Enter matrix B (4 x 4):
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
Result matrix C:
1 2 3 4
2 3 4 5
5 6 7 8
6 7 8 9
posh@LAPTOP-ELUQQMKU:~/hpc_assign/assign7$
```

Algorithm

1. Initialize the MPI environment.
2. Process 0 (root) takes size n, and matrices A and B as input.
3. The rows of matrix A are scattered among all processes.
4. Matrix B is broadcast to all processes.
5. Each process computes partial product:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj}$$

1. for its assigned rows.
2. Partial results are gathered back at the root process.
3. Root process prints the result matrix.
4. Finalize MPI.

Observations (Sample Outputs)

- Speedup increases with larger matrices but communication overhead affects small cases.

Conclusion

- Matrix–matrix multiplication is highly parallelizable, as computations for rows can be distributed.
- MPI provides good scalability for large n.
- Communication and gathering steps are bottlenecks when n is small.