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

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

**Course:** High Performance Computing Lab

**Practical No. 7**

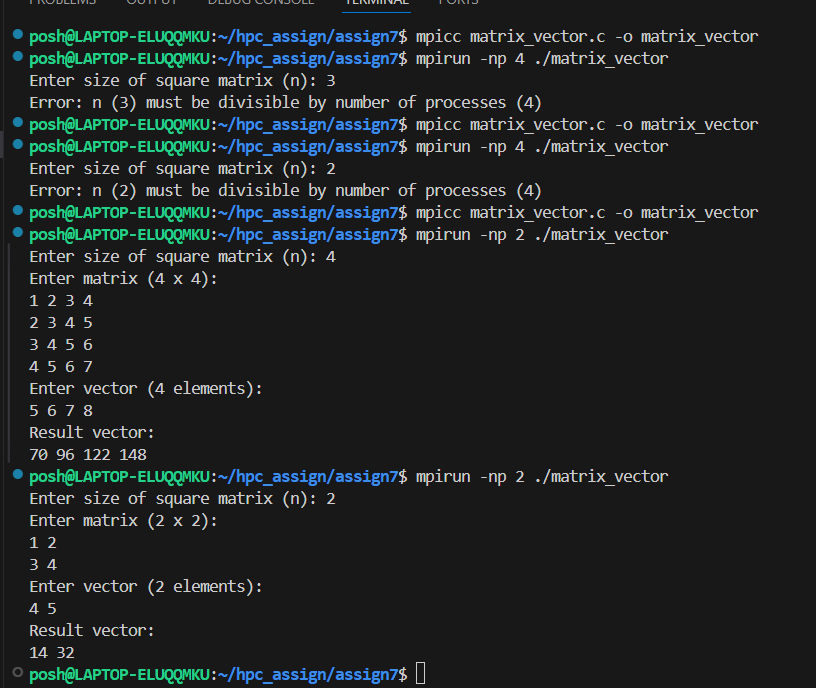
**Exam Seat No: 22510064 – Parshwa Herwade**

**Github Link:** [**Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub**](https://github.com/parshwa913/Sem-7-Assign/tree/main/HPC%20lab)

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

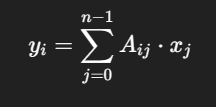
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mpi.h>
4. int main(int argc, char\* argv[]) {
5. int rank, size;
6. int n;
7. int \*matrix = NULL, \*vector = NULL, \*result = NULL;
8. int \*local\_matrix, \*local\_result;
9. int rows\_per\_proc;
10. MPI\_Init(&argc, &argv);
11. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
12. MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);
13. if (rank == 0) {
14. printf("Enter size of square matrix (n): ");
15. fflush(stdout);
16. scanf("%d", &n);
17. }
18. MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
19. if (n % size != 0) {
20. if (rank == 0) {
21. printf("Error: n (%d) must be divisible by number of processes (%d)\n", n, size);
22. }
23. MPI\_Finalize();
24. return 0;
25. }
26. rows\_per\_proc = n / size;
27. if (rank == 0) {
28. matrix = (int\*)malloc(n \* n \* sizeof(int));
29. vector = (int\*)malloc(n \* sizeof(int));
30. result = (int\*)malloc(n \* sizeof(int));
31. printf("Enter matrix (%d x %d):\n", n, n);
32. for (int i = 0; i < n; i++)
33. for (int j = 0; j < n; j++)
34. scanf("%d", &matrix[i \* n + j]);
35. printf("Enter vector (%d elements):\n", n);
36. for (int i = 0; i < n; i++)
37. scanf("%d", &vector[i]);
38. }
39. local\_matrix = (int\*)malloc(rows\_per\_proc \* n \* sizeof(int));
40. local\_result = (int\*)malloc(rows\_per\_proc \* sizeof(int));
41. if (rank != 0) vector = (int\*)malloc(n \* sizeof(int));
42. MPI\_Scatter(matrix, rows\_per\_proc \* n, MPI\_INT,
43. local\_matrix, rows\_per\_proc \* n, MPI\_INT,
44. 0, MPI\_COMM\_WORLD);
45. MPI\_Bcast(vector, n, MPI\_INT, 0, MPI\_COMM\_WORLD);
46. for (int i = 0; i < rows\_per\_proc; i++) {
47. local\_result[i] = 0;
48. for (int j = 0; j < n; j++) {
49. local\_result[i] += local\_matrix[i \* n + j] \* vector[j];
50. }
51. }
52. MPI\_Gather(local\_result, rows\_per\_proc, MPI\_INT,
53. result, rows\_per\_proc, MPI\_INT,
54. 0, MPI\_COMM\_WORLD);
55. if (rank == 0) {
56. printf("Result vector:\n");
57. for (int i = 0; i < n; i++)
58. printf("%d ", result[i]);
59. printf("\n");
60. }
61. if (rank == 0) { free(matrix); free(vector); free(result); }
62. else free(vector);
63. free(local\_matrix);
64. free(local\_result);
65. MPI\_Finalize();
66. return 0;
67. }

OUTPUT:



**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:



for its assigned rows.

1. The partial results are gathered at the root process using MPI\_Gather.
2. Root process prints the result vector.
3. Finalize MPI.

Observations (Sample Outputs)

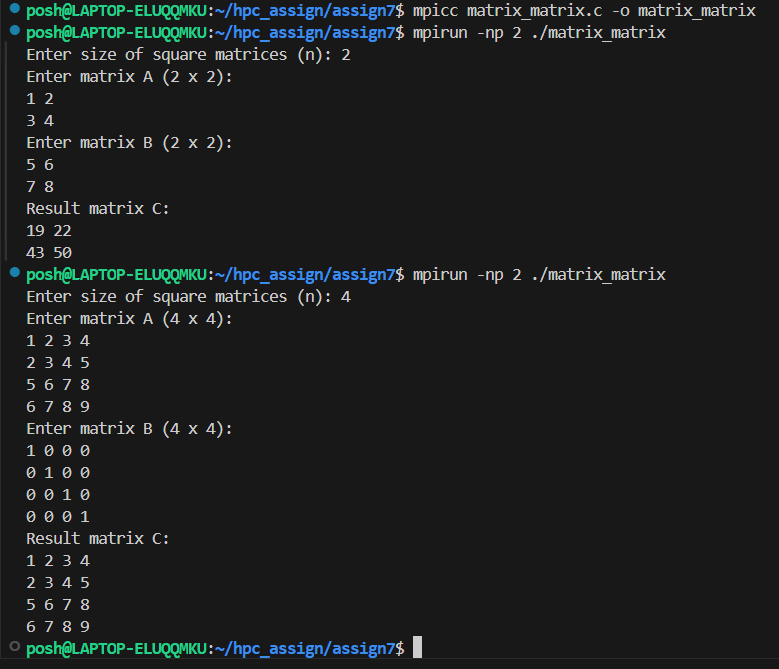
* Execution time decreases as the number of processes increases (for large matrices).
* For small n, communication overhead may dominate, giving no real speedup.

**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.

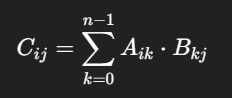
## Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyze the performance.

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <mpi.h>
4. int main(int argc, char\* argv[]) {
5. int rank, size;
6. int n;
7. int \*A = NULL, \*B = NULL, \*C = NULL;
8. int \*local\_A, \*local\_C;
9. int rows\_per\_proc;
10. MPI\_Init(&argc, &argv);
11. MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);
12. MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);
13. if (rank == 0) {
14. printf("Enter size of square matrices (n): ");
15. fflush(stdout);
16. scanf("%d", &n);
17. }
18. MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);
19. if (n % size != 0) {
20. if (rank == 0) {
21. printf("Error: n (%d) must be divisible by number of processes (%d)\n", n, size);
22. }
23. MPI\_Finalize();
24. return 0;
25. }
26. rows\_per\_proc = n / size;
27. if (rank == 0) {
28. A = (int\*)malloc(n \* n \* sizeof(int));
29. B = (int\*)malloc(n \* n \* sizeof(int));
30. C = (int\*)malloc(n \* n \* sizeof(int));
31. printf("Enter matrix A (%d x %d):\n", n, n);
32. for (int i = 0; i < n; i++)
33. for (int j = 0; j < n; j++)
34. scanf("%d", &A[i \* n + j]);
35. printf("Enter matrix B (%d x %d):\n", n, n);
36. for (int i = 0; i < n; i++)
37. for (int j = 0; j < n; j++)
38. scanf("%d", &B[i \* n + j]);
39. }
40. local\_A = (int\*)malloc(rows\_per\_proc \* n \* sizeof(int));
41. local\_C = (int\*)malloc(rows\_per\_proc \* n \* sizeof(int));
42. if (rank != 0) B = (int\*)malloc(n \* n \* sizeof(int));
43. MPI\_Scatter(A, rows\_per\_proc \* n, MPI\_INT,
44. local\_A, rows\_per\_proc \* n, MPI\_INT,
45. 0, MPI\_COMM\_WORLD);
46. MPI\_Bcast(B, n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);
47. for (int i = 0; i < rows\_per\_proc; i++) {
48. for (int j = 0; j < n; j++) {
49. local\_C[i \* n + j] = 0;
50. for (int k = 0; k < n; k++) {
51. local\_C[i \* n + j] += local\_A[i \* n + k] \* B[k \* n + j];
52. }
53. }
54. }
55. MPI\_Gather(local\_C, rows\_per\_proc \* n, MPI\_INT,
56. C, rows\_per\_proc \* n, MPI\_INT,
57. 0, MPI\_COMM\_WORLD);
58. if (rank == 0) {
59. printf("Result matrix C:\n");
60. for (int i = 0; i < n; i++) {
61. for (int j = 0; j < n; j++)
62. printf("%d ", C[i \* n + j]);
63. printf("\n");
64. }
65. }
66. if (rank == 0) { free(A); free(B); free(C); }
67. else free(B);
68. free(local\_A);
69. free(local\_C);
70. MPI\_Finalize();
71. return 0;
72. }

OUTPUT:  


**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:



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.