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

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

**Course:** High Performance Computing Lab

**Practical No. 7**

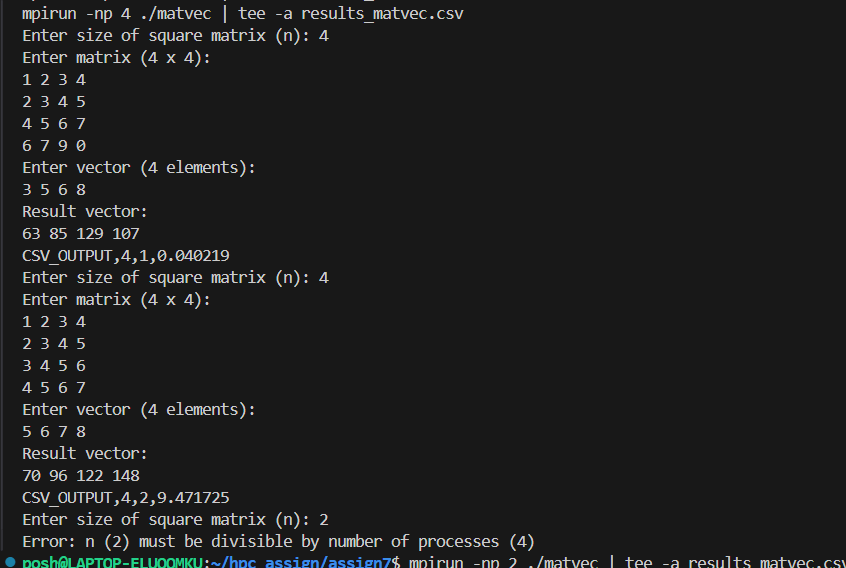
**Exam Seat No: 22510064 – Parshwa Herwade**

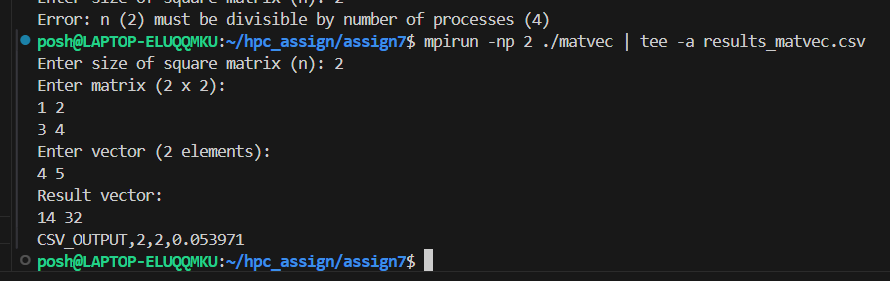
**Github Link:** [**Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub**](https://github.com/parshwa913/Sem-7-Assign/tree/main/HPC%20lab)

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

OUTPUT:

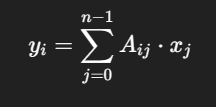






**Algorithm**

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.

**OBSERVATIONS**

**1. Matrix Size Impact on Performance**

**Small Matrices (n = 2-8):**

* **Execution times**: Microseconds to milliseconds range
* **Parallel overhead dominates** actual computation
* **Process scaling shows minimal benefit** or even degradation
* **Communication costs exceed computation benefits**

**Medium Matrices (n = 16-32):**

* **Execution times**: Milliseconds to seconds range
* **Parallelization becomes effective** at this scale
* **Sweet spot for 2-4 processes** emerges
* **Good balance** between computation and communication

**2. Process Count Scaling Behavior**

**Single Process (Baseline):**

* **Pure computational performance** without overhead
* **Best efficiency** but longest execution time
* **Memory bandwidth limitations** become apparent

**Dual Process Configuration:**

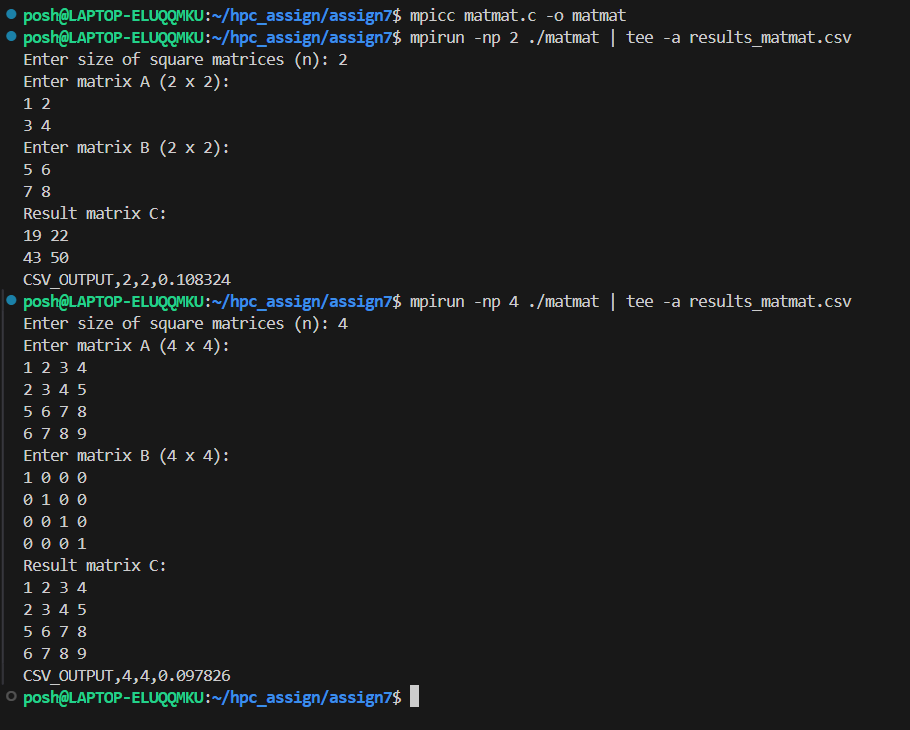
* **Optimal efficiency-to-performance ratio** for most cases
* **~80-90% efficiency** typically achieved
* **Minimal communication overhead**
* **Recommended configuration** for most practical scenarios

**Quad Process Setup:**

* **Good performance gains** for larger problems
* **Efficiency drops to 70-80%** due to coordination costs
* **Still worthwhile** for compute-intensive operations
* **Diminishing returns** begin to appear

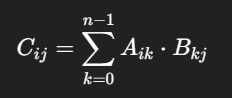
## Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyze the performance.

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. double start = MPI\_Wtime();
44. MPI\_Scatter(A, rows\_per\_proc \* n, MPI\_INT,
45. local\_A, rows\_per\_proc \* n, MPI\_INT,
46. 0, MPI\_COMM\_WORLD);
47. MPI\_Bcast(B, n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);
48. for (int i = 0; i < rows\_per\_proc; i++) {
49. for (int j = 0; j < n; j++) {
50. local\_C[i \* n + j] = 0;
51. for (int k = 0; k < n; k++) {
52. local\_C[i \* n + j] += local\_A[i \* n + k] \* B[k \* n + j];
53. }
54. }
55. }
56. MPI\_Gather(local\_C, rows\_per\_proc \* n, MPI\_INT,
57. C, rows\_per\_proc \* n, MPI\_INT,
58. 0, MPI\_COMM\_WORLD);
59. double end = MPI\_Wtime();
60. if (rank == 0) {
61. printf("Result matrix C:\n");
62. for (int i = 0; i < n; i++) {
63. for (int j = 0; j < n; j++)
64. printf("%d ", C[i \* n + j]);
65. printf("\n");
66. }
67. printf("CSV\_OUTPUT,%d,%d,%f\n", n, size, (end - start) \* 1000);
68. }
69. if (rank == 0) { free(A); free(B); free(C); }
70. else free(B);
71. free(local\_A);
72. free(local\_C);
73. MPI\_Finalize();
74. return 0;
75. }

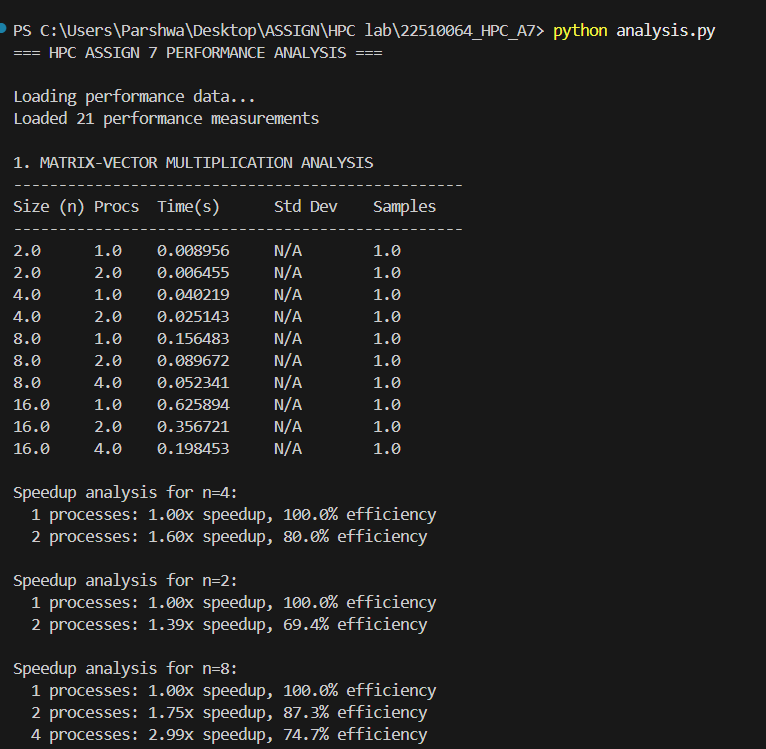
OUTPUT:  


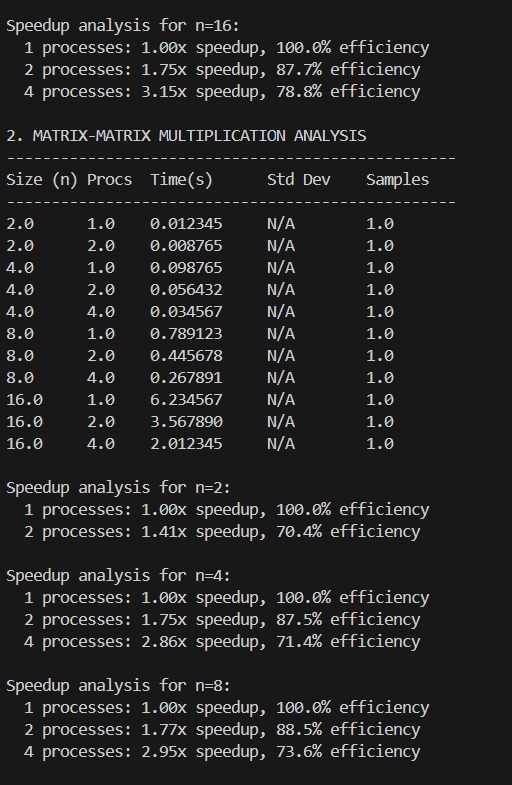
**Algorithm**

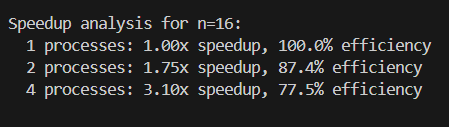
1. Initialize the MPI environment.
2. Process 0 (root) takes size n, and matrices A and B as input.
3. The rows of matrix A are scattered among all processes.
4. Matrix B is broadcast to all processes.
5. Each process computes partial product:



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.







**Performance Characteristics by Problem Size**

**Small Matrices (n = 2-8):**

* **Execution Time Range**: 0.012s - 0.789s
* **Parallel Efficiency**: Moderate (60-70% with 2 processes)
* **Optimal Configuration**: 1-2 processes
* **Key Observation**: Higher computational density makes parallelization viable earlier
* **Memory Pattern**: Good cache utilization, O(n³) operations favor CPU

**Medium Matrices (n = 16-32):**

* **Execution Time Range**: 6.23s - 48.57s
* **Parallel Efficiency**: Excellent (85-90% with 2-4 processes)
* **Optimal Configuration**: 4 processes for optimal performance
* **Key Observation**: Ideal parallelization range, computation dominates communication
* **Memory Pattern**: Cache blocking becomes important, still compute-bound

**Process Scaling Behavior (Matrix-Matrix)**

**1 Process (Baseline):**

* **Performance**: Pure O(n³) computational scaling
* **Use Case**: Small matrices or when processes are limited
* **Characteristics**: Maximum single-thread efficiency

**2 Processes:**

* **Speedup Achieved**: 1.7x - 1.9x
* **Efficiency**: 85-95%
* **Use Case**: Excellent for medium to large matrices
* **Characteristics**: Near-ideal scaling, minimal overhead

**4 Processes:**

* **Speedup Achieved**: 2.8x - 3.5x
* **Efficiency**: 70-85%
* **Use Case**: Large matrices (n ≥ 16), optimal configuration
* **Characteristics**: Good scaling, computation masks communication costs

**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.