Name: Shreeyash Shripad Dongarkar

PRN: 22510025

High Performance Computing Lab (B – 1)

Assignment – 7

Title: Implementation of Matrix – Vector & Matrix – Matrix Multiplication using MPI

[GitHub Repository](https://github.com/TerminatorShri/22510025_HPCL)

Problem Statement 1:  
Implement Matrix-Vector Multiplication using MPI. Use different number of processes and analyse the performance.

1. #include <mpi.h>

2. #include <stdio.h>

3. #include <stdlib.h>

4.

5. static int header\_printed = 0;

6.

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

8.     int rank, size, N;

9.     int \*A = NULL, \*x = NULL, \*local\_rows, \*local\_result, \*final\_result;

10.     int rows\_per\_proc;

11.

12.     MPI\_Init(&argc, &argv);

13.     MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

14.     MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

15.

16.     if (argc < 2) {

17.         if (rank == 0) printf("Usage: %s <matrix\_size>\n", argv[0]);

18.         MPI\_Finalize();

19.         return 0;

20.     }

21.

22.     N = atoi(argv[1]);

23.     if (N % size != 0) {

24.         if (rank == 0)

25.             printf("Error: N (%d) must be divisible by number of processes (%d)\n", N, size);

26.         MPI\_Finalize();

27.         return 0;

28.     }

29.

30.     rows\_per\_proc = N / size;

31.

32.     x = (int \*)malloc(N \* sizeof(int));

33.     local\_rows = (int \*)malloc(rows\_per\_proc \* N \* sizeof(int));

34.     local\_result = (int \*)malloc(rows\_per\_proc \* sizeof(int));

35.

36.     if (rank == 0) {

37.         A = (int \*)malloc(N \* N \* sizeof(int));

38.         final\_result = (int \*)malloc(N \* sizeof(int));

39.

40.         for (int i = 0; i < N; i++) {

41.             x[i] = i + 1;

42.             for (int j = 0; j < N; j++) {

43.                 A[i \* N + j] = (i + j + 1) % 10;

44.             }

45.         }

46.     }

47.

48.     MPI\_Bcast(x, N, MPI\_INT, 0, MPI\_COMM\_WORLD);

49.

50.     double start\_time = MPI\_Wtime();

51.

52.     MPI\_Scatter(A, rows\_per\_proc \* N, MPI\_INT,

53.                 local\_rows, rows\_per\_proc \* N, MPI\_INT,

54.                 0, MPI\_COMM\_WORLD);

55.

56.     for (int i = 0; i < rows\_per\_proc; i++) {

57.         local\_result[i] = 0;

58.         for (int j = 0; j < N; j++) {

59.             local\_result[i] += local\_rows[i \* N + j] \* x[j];

60.         }

61.     }

62.

63.     MPI\_Gather(local\_result, rows\_per\_proc, MPI\_INT,

64.                final\_result, rows\_per\_proc, MPI\_INT,

65.                0, MPI\_COMM\_WORLD);

66.

67.     double end\_time = MPI\_Wtime();

68.     double exec\_time = end\_time - start\_time;

69.

70.     if (rank == 0) {

71.         FILE \*fp = fopen("PS\_1.csv", "a");

72.         if (fp == NULL) {

73.             printf("Error opening file!\n");

74.             MPI\_Abort(MPI\_COMM\_WORLD, 1);

75.         }

76.         fprintf(fp, "%d,%d,%f\n", N, size, exec\_time);

77.         fclose(fp);

78.     }

79.

80.     free(x);

81.     free(local\_rows);

82.     free(local\_result);

83.     if (rank == 0) {

84.         free(A);

85.         free(final\_result);

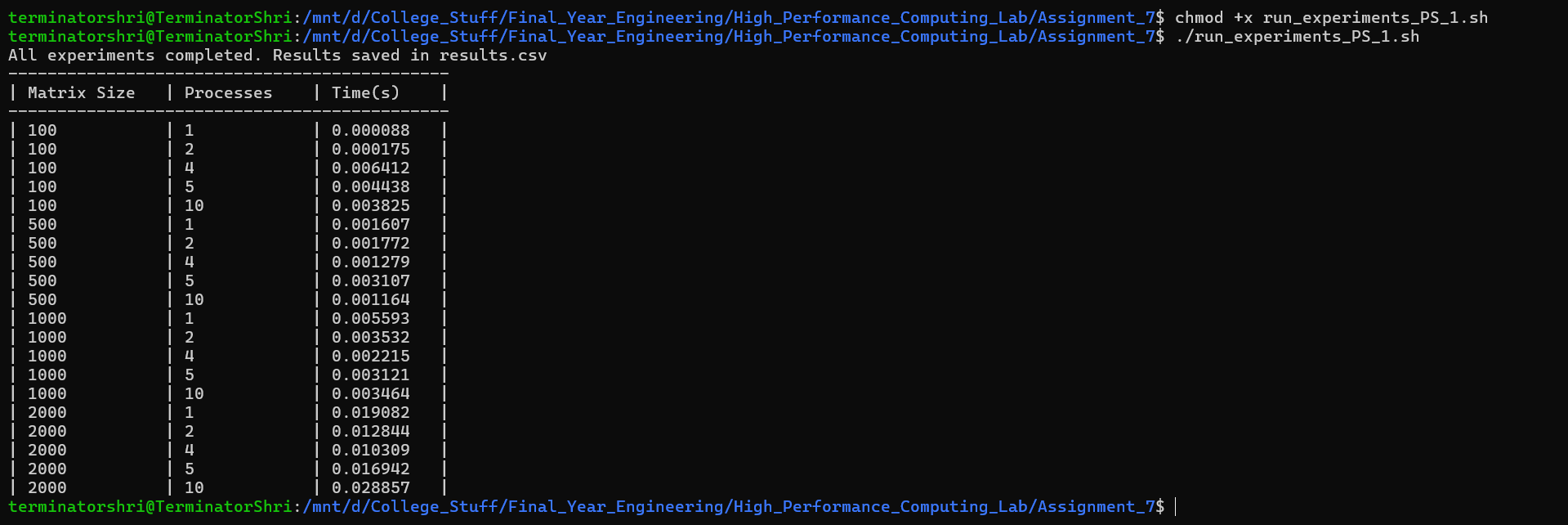
86.     }

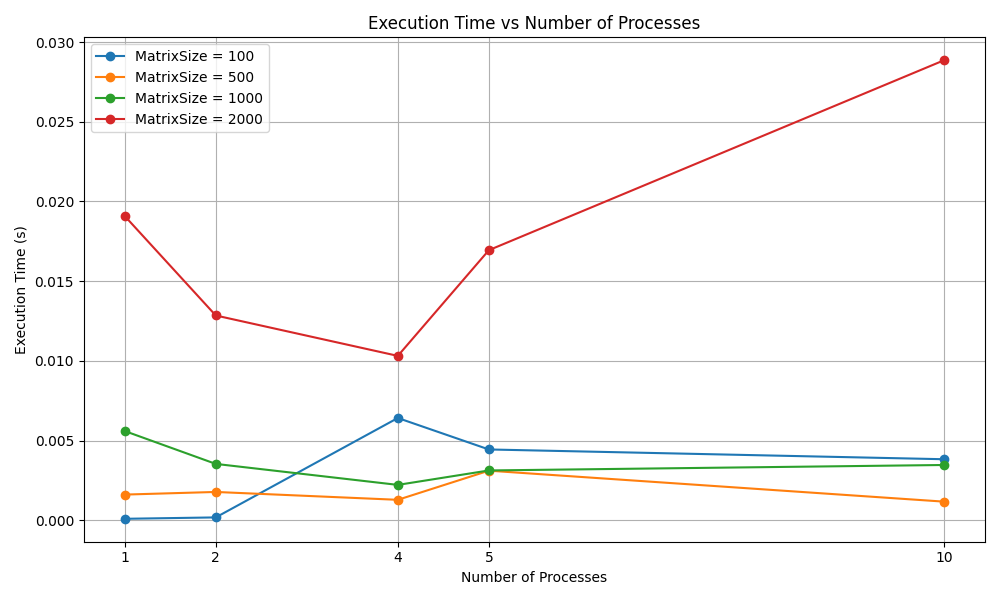
87.

88.     MPI\_Finalize();

89.     return 0;

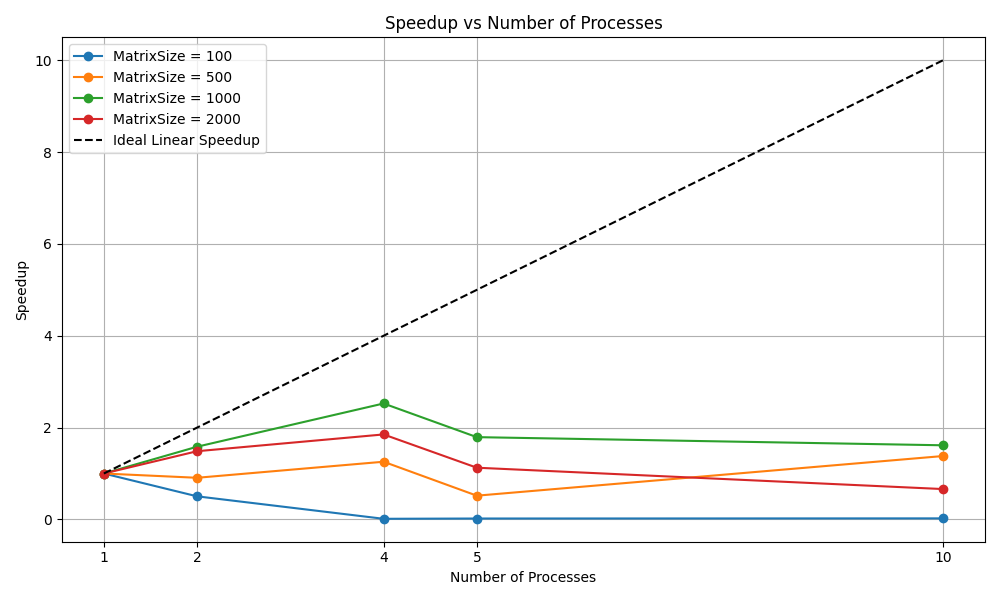
90. }



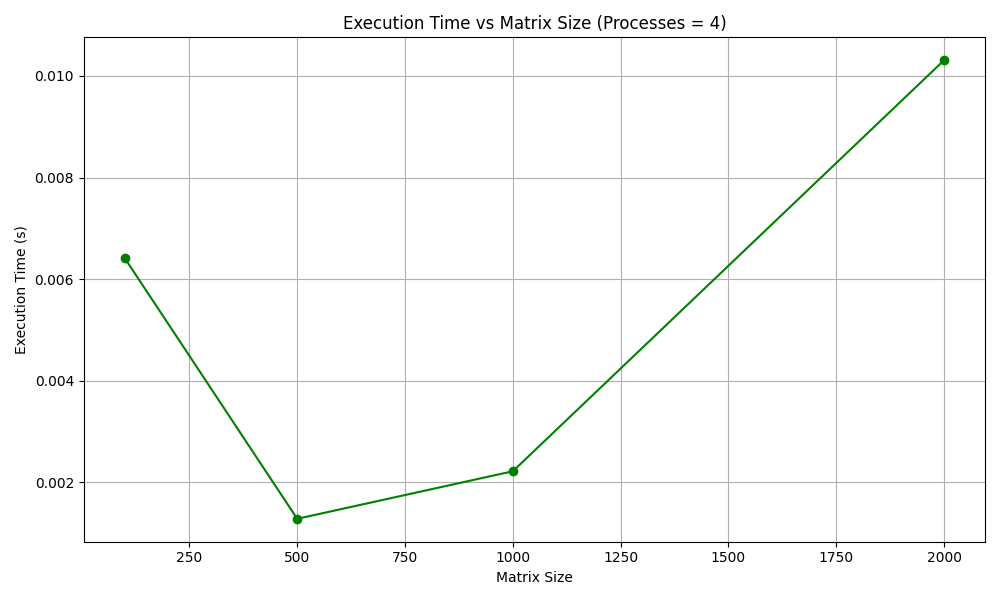


For small matrix sizes (e.g., 100, 500), the execution time is extremely low, almost negligible. As matrix size increases (1000, 2000), execution times increase, which is expected since computation grows with matrix size. Beyond 4 – 5 processes, execution time start increasing indicating diminishing returns and communication overhead incurred by MPI Scatter and Gather

Matrix-Vector multiplication is **compute-light** for small matrices; parallelism only helps for larger matrices.



Speedup actually drops to near zero as soon as more than 2 processes are used. This means communication overhead dominates computation when matrix size is very small. For large matrix sizes there is some speedup but it again drops on increasing number of processes likely due to communication overhead and load imbalance.



For 4 processes 500 is the optimal matrix size where computation is large enough to mask MPI overhead, but not too large to cause excessive data transfer and computation load. There’s an **optimal number of processes** depending on matrix size; beyond that, parallel overhead dominates.

Problem Statement 2:  
Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyse the performance.

1. #include <mpi.h>

2. #include <stdio.h>

3. #include <stdlib.h>

4.

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

6.     int rank, size, N;

7.     int \*A = NULL, \*B = NULL, \*local\_A, \*local\_C, \*C = NULL;

8.     int rows\_per\_proc;

9.

10.     MPI\_Init(&argc, &argv);

11.     MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

12.     MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

13.

14.     if (argc < 2) {

15.         if (rank == 0) printf("Usage: %s <matrix\_size>\n", argv[0]);

16.         MPI\_Finalize();

17.         return 0;

18.     }

19.

20.     N = atoi(argv[1]);

21.     if (N % size != 0) {

22.         if (rank == 0)

23.             printf("Error: N (%d) must be divisible by number of processes (%d)\n", N, size);

24.         MPI\_Finalize();

25.         return 0;

26.     }

27.

28.     rows\_per\_proc = N / size;

29.

30.     local\_A = (int \*)malloc(rows\_per\_proc \* N \* sizeof(int));

31.     local\_C = (int \*)malloc(rows\_per\_proc \* N \* sizeof(int));

32.     B = (int \*)malloc(N \* N \* sizeof(int));

33.

34.     if (rank == 0) {

35.         A = (int \*)malloc(N \* N \* sizeof(int));

36.         C = (int \*)malloc(N \* N \* sizeof(int));

37.

38.         for (int i = 0; i < N; i++) {

39.             for (int j = 0; j < N; j++) {

40.                 A[i \* N + j] = (i + j + 1) % 10;

41.                 B[i \* N + j] = (i + j + 2) % 10;

42.             }

43.         }

44.     }

45.

46.     MPI\_Bcast(B, N \* N, MPI\_INT, 0, MPI\_COMM\_WORLD);

47.

48.     MPI\_Scatter(A, rows\_per\_proc \* N, MPI\_INT,

49.                 local\_A, rows\_per\_proc \* N, MPI\_INT,

50.                 0, MPI\_COMM\_WORLD);

51.

52.     double start\_time = MPI\_Wtime();

53.

54.     for (int i = 0; i < rows\_per\_proc; i++) {

55.         for (int j = 0; j < N; j++) {

56.             local\_C[i \* N + j] = 0;

57.             for (int k = 0; k < N; k++) {

58.                 local\_C[i \* N + j] += local\_A[i \* N + k] \* B[k \* N + j];

59.             }

60.         }

61.     }

62.

63.     MPI\_Gather(local\_C, rows\_per\_proc \* N, MPI\_INT,

64.                C, rows\_per\_proc \* N, MPI\_INT,

65.                0, MPI\_COMM\_WORLD);

66.

67.     double end\_time = MPI\_Wtime();

68.     double exec\_time = end\_time - start\_time;

69.

70.     if (rank == 0) {

71.         FILE \*fp = fopen("PS\_2.csv", "a");

72.         if (!fp) {

73.             printf("Error opening file!\n");

74.             MPI\_Abort(MPI\_COMM\_WORLD, 1);

75.         }

76.         fprintf(fp, "%d,%d,%f\n", N, size, exec\_time);

77.         fclose(fp);

78.     }

79.

80.     free(local\_A);

81.     free(local\_C);

82.     free(B);

83.     if (rank == 0) {

84.         free(A);

85.         free(C);

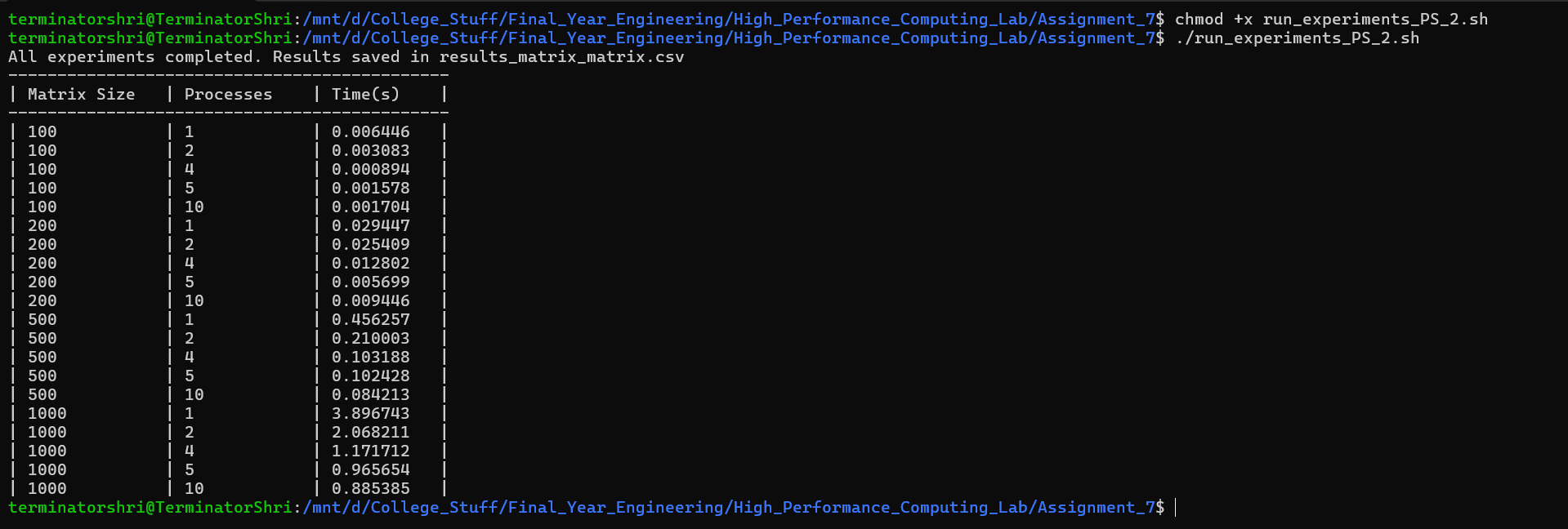
86.     }

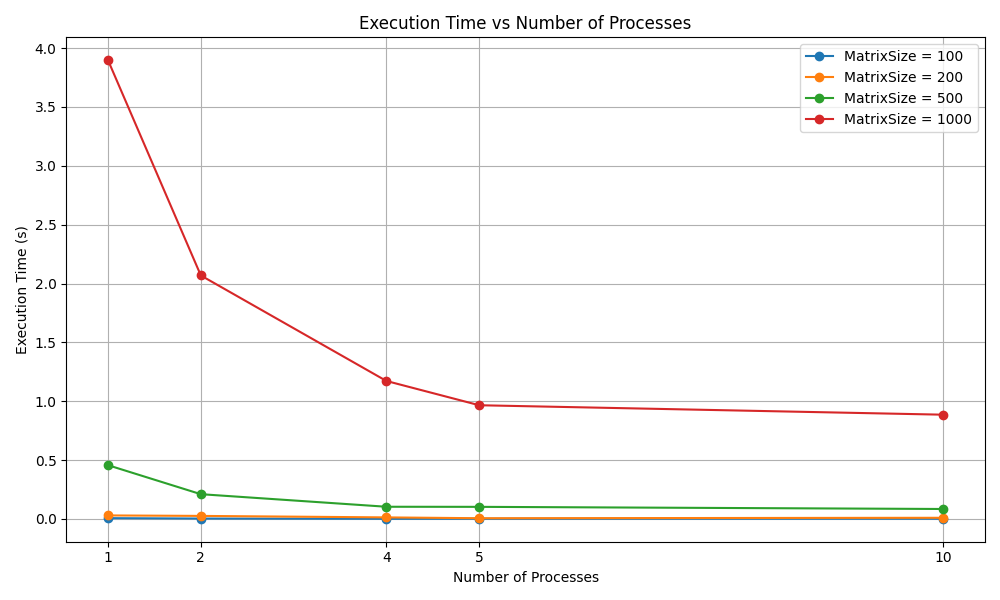
87.

88.     MPI\_Finalize();

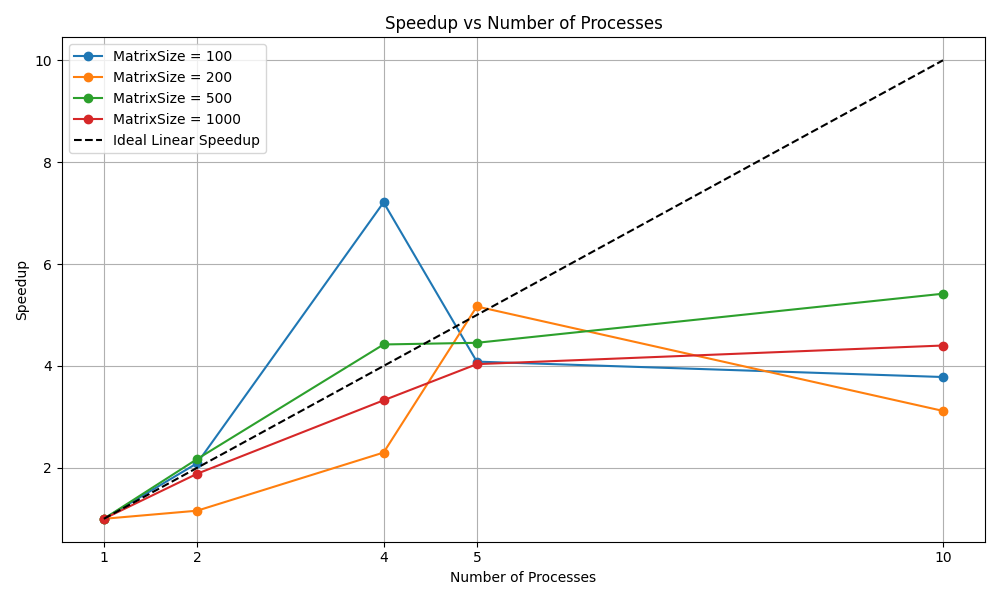
89.     return 0;

90. }

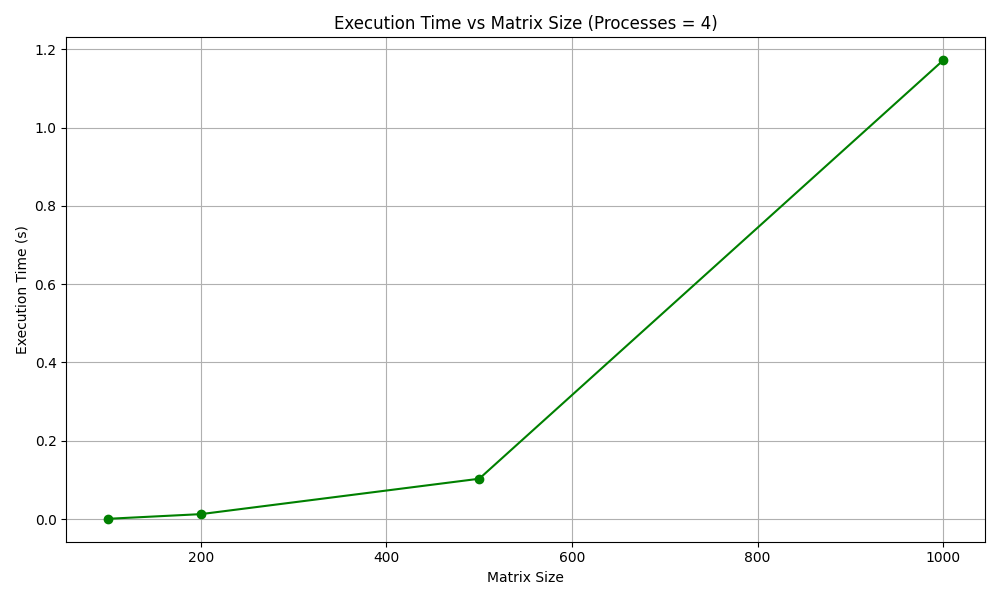




For smaller matrices parallelization is not effective but scalability is good for larger matrices as computation is large enough to amortize communication costs.



We can see speedup up to certain number of processed then drops as high communication overhead is incurred for smaller matrices. For medium matrix size is reasonably close to ideal can be best scaling case. For larger sizes there is gradual, consistent improvement but performance gain is lower suggesting growing overhead with larger communication demands, limiting efficiency.



With 4 processes, execution time scales roughly cubic with matrix size. Parallelization reduces absolute execution time (compared to 1 process), but the fundamental growth (O(n³)) remains.