Name: Shreeyash Shripad Dongarkar

PRN: 22510025

High Performance Computing Lab (B – 1)

Assignment – 8

Title: Implementation of Convolution and Dot Product using MPI

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

Problem Statement 1:  
Study and implement 2D Convolution using MPI. Use a different number of processes and analyse the performance.

1. #include <mpi.h>

2. #include <stdio.h>

3. #include <stdlib.h>

4. #include <string.h>

5.

6. static int header\_printed = 0;

7.

8. void apply\_convolution(int \*input\_chunk, int \*output\_chunk, int \*kernel,

9.                       int chunk\_rows, int cols, int kernel\_size,

10.                       int start\_row, int total\_rows) {

11.     int kernel\_center = kernel\_size / 2;

12.

13.     for (int i = 0; i < chunk\_rows; i++) {

14.         for (int j = 0; j < cols; j++) {

15.             int sum = 0;

16.

17.             for (int ki = 0; ki < kernel\_size; ki++) {

18.                 for (int kj = 0; kj < kernel\_size; kj++) {

19.                     int row\_idx = (start\_row + i) + ki - kernel\_center;

20.                     int col\_idx = j + kj - kernel\_center;

21.

22.                     if (row\_idx >= 0 && row\_idx < total\_rows &&

23.                         col\_idx >= 0 && col\_idx < cols) {

24.                         int target\_process = row\_idx / chunk\_rows;

25.                         int local\_row = row\_idx % chunk\_rows;

26.

27.                         if (target\_process \* chunk\_rows + local\_row < total\_rows) {

28.                             sum += input\_chunk[(local\_row \* cols) + col\_idx] \*

29.                                    kernel[ki \* kernel\_size + kj];

30.                         }

31.                     }

32.                 }

33.             }

34.             output\_chunk[i \* cols + j] = sum;

35.         }

36.     }

37. }

38.

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

40.     int rank, size, N, kernel\_size;

41.     int \*input\_matrix = NULL, \*output\_matrix = NULL;

42.     int \*local\_input, \*local\_output, \*kernel;

43.     int rows\_per\_proc;

44.

45.     MPI\_Init(&argc, &argv);

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

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

48.

49.     if (argc < 2) {

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

51.         MPI\_Finalize();

52.         return 0;

53.     }

54.

55.     N = atoi(argv[1]);

56.     kernel\_size = (argc > 2) ? atoi(argv[2]) : 3;

57.

58.     if (N % size != 0) {

59.         if (rank == 0)

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

61.         MPI\_Finalize();

62.         return 0;

63.     }

64.

65.     rows\_per\_proc = N / size;

66.

67.     // Allocate memory for local data

68.     local\_input = (int \*)malloc(rows\_per\_proc \* N \* sizeof(int));

69.     local\_output = (int \*)malloc(rows\_per\_proc \* N \* sizeof(int));

70.     kernel = (int \*)malloc(kernel\_size \* kernel\_size \* sizeof(int));

71.

72.     if (rank == 0) {

73.         input\_matrix = (int \*)malloc(N \* N \* sizeof(int));

74.         output\_matrix = (int \*)malloc(N \* N \* sizeof(int));

75.

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

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

78.                 input\_matrix[i \* N + j] = (i + j) % 10 + 1;

79.             }

80.         }

81.

82.         if (kernel\_size == 3) {

83.             int edge\_kernel[9] = {

84.                 -1, -1, -1,

85.                 -1,  8, -1,

86.                 -1, -1, -1

87.             };

88.             memcpy(kernel, edge\_kernel, 9 \* sizeof(int));

89.         } else {

90.             for (int i = 0; i < kernel\_size \* kernel\_size; i++) {

91.                 kernel[i] = 1;

92.             }

93.         }

94.

95.         FILE \*fp = fopen("PS\_1.csv", "r");

96.         if (fp == NULL) {

97.             fp = fopen("PS\_1.csv", "w");

98.             if (fp != NULL) {

99.                 fprintf(fp, "MatrixSize,NumProcesses,TimeTaken\n");

100.                 fclose(fp);

101.             }

102.         } else {

103.             fclose(fp);

104.         }

105.     }

106.

107.     MPI\_Bcast(kernel, kernel\_size \* kernel\_size, MPI\_INT, 0, MPI\_COMM\_WORLD);

108.

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

110.

111.     MPI\_Scatter(input\_matrix, rows\_per\_proc \* N, MPI\_INT,

112.                 local\_input, rows\_per\_proc \* N, MPI\_INT,

113.                 0, MPI\_COMM\_WORLD);

114.

115.     int \*extended\_input = (int \*)malloc((rows\_per\_proc + kernel\_size - 1) \* N \* sizeof(int));

116.

117.     int \*all\_data = NULL;

118.     if (rank == 0) {

119.         all\_data = (int \*)malloc(N \* N \* sizeof(int));

120.     }

121.

122.     MPI\_Gather(local\_input, rows\_per\_proc \* N, MPI\_INT,

123.                all\_data, rows\_per\_proc \* N, MPI\_INT,

124.                0, MPI\_COMM\_WORLD);

125.

126.     if (rank == 0) {

127.         int kernel\_center = kernel\_size / 2;

128.

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

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

131.                 int sum = 0;

132.

133.                 for (int ki = 0; ki < kernel\_size; ki++) {

134.                     for (int kj = 0; kj < kernel\_size; kj++) {

135.                         int row\_idx = i + ki - kernel\_center;

136.                         int col\_idx = j + kj - kernel\_center;

137.

138.                         if (row\_idx >= 0 && row\_idx < N &&

139.                             col\_idx >= 0 && col\_idx < N) {

140.                             sum += all\_data[row\_idx \* N + col\_idx] \*

141.                                    kernel[ki \* kernel\_size + kj];

142.                         }

143.                     }

144.                 }

145.                 output\_matrix[i \* N + j] = sum;

146.             }

147.         }

148.     }

149.

150.     MPI\_Scatter(output\_matrix, rows\_per\_proc \* N, MPI\_INT,

151.                 local\_output, rows\_per\_proc \* N, MPI\_INT,

152.                 0, MPI\_COMM\_WORLD);

153.

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

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

156.

157.     if (rank == 0) {

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

159.         if (fp == NULL) {

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

161.         }

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

163.         fclose(fp);

164.     }

165.

166.     free(local\_input);

167.     free(local\_output);

168.     free(kernel);

169.     free(extended\_input);

170.

171.     if (rank == 0) {

172.         free(input\_matrix);

173.         free(output\_matrix);

174.         if (all\_data) free(all\_data);

175.     }

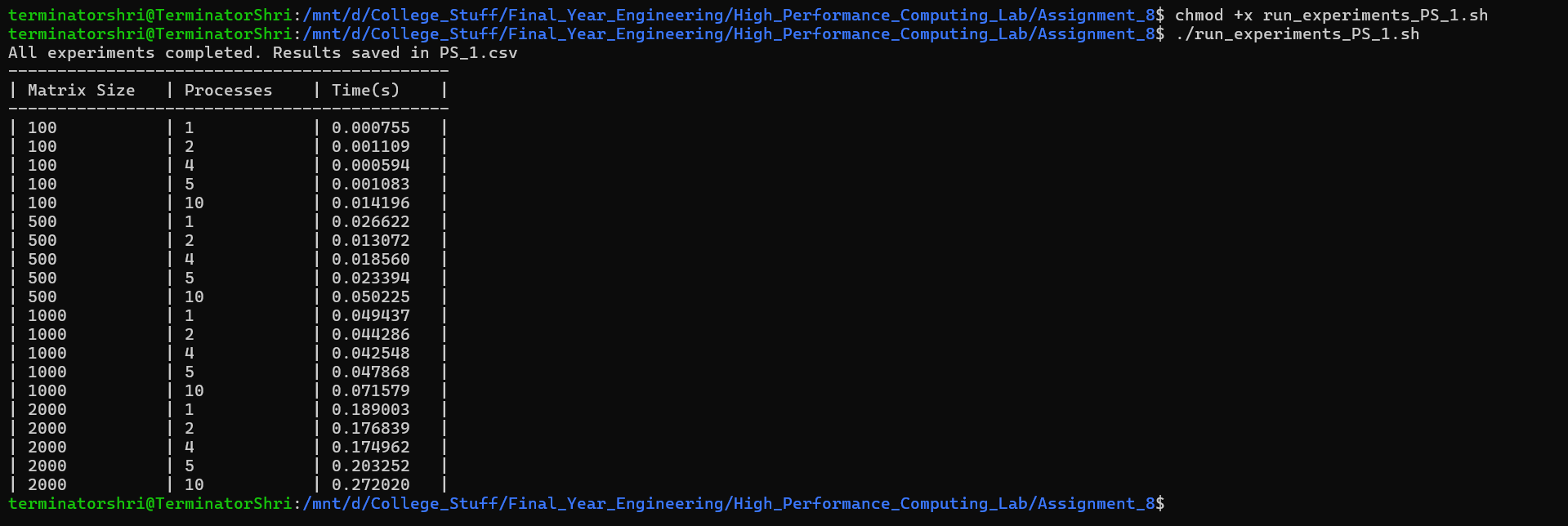
176.

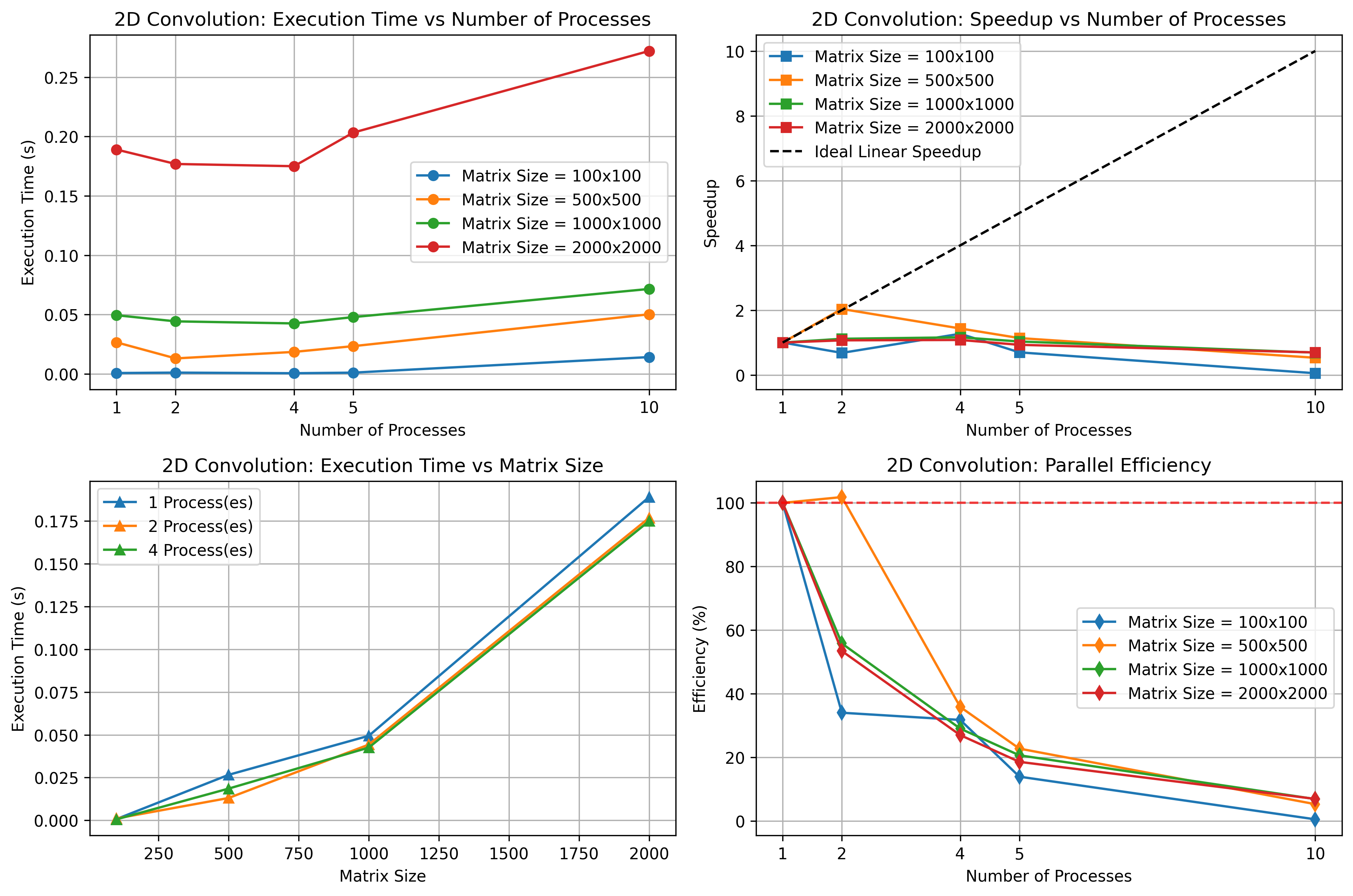
177.     MPI\_Finalize();

178.     return 0;

179. }

180.





For all matrix sizes, **execution time increases with number of processes.** Expected behaviour is reduced time with more processes, but here overhead (communication, synchronization, data distribution) dominates.

Ideal speedup is far above the actual speedup. This confirms communication and distribution overhead dominates computation, meaning parallel execution can be slower than sequential.

Efficiency for 2 processes, drops sharply for small matrices. For more than 2 processes, efficiency plummets below 20% across all matrix sizes which indicates poor scalability due to MPI overhead.

Problem Statement 2:  
Implement dot product using MPI. Use different number of processes and analyze the performance.

1. #include <mpi.h>

2. #include <stdio.h>

3. #include <stdlib.h>

4. #include <time.h>

5.

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

7.     int rank, size, N;

8.     double \*vector\_a = NULL, \*vector\_b = NULL;

9.     double \*local\_a, \*local\_b;

10.     double local\_dot\_product, global\_dot\_product;

11.     int elements\_per\_proc;

12.

13.     MPI\_Init(&argc, &argv);

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

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

16.

17.     if (argc < 2) {

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

19.         MPI\_Finalize();

20.         return 0;

21.     }

22.

23.     N = atoi(argv[1]);

24.

25.     if (N % size != 0) {

26.         if (rank == 0)

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

28.         MPI\_Finalize();

29.         return 0;

30.     }

31.

32.     elements\_per\_proc = N / size;

33.

34.     local\_a = (double \*)malloc(elements\_per\_proc \* sizeof(double));

35.     local\_b = (double \*)malloc(elements\_per\_proc \* sizeof(double));

36.

37.     if (rank == 0) {

38.         vector\_a = (double \*)malloc(N \* sizeof(double));

39.         vector\_b = (double \*)malloc(N \* sizeof(double));

40.

41.         srand(time(NULL));

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

43.             vector\_a[i] = (double)(rand() % 100) / 10.0;

44.             vector\_b[i] = (double)(rand() % 100) / 10.0;

45.         }

46.

47.         FILE \*fp = fopen("PS\_2.csv", "r");

48.         if (fp == NULL) {

49.             fp = fopen("PS\_2.csv", "w");

50.             if (fp != NULL) {

51.                 fprintf(fp, "VectorSize,NumProcesses,TimeTaken\n");

52.                 fclose(fp);

53.             }

54.         } else {

55.             fclose(fp);

56.         }

57.     }

58.

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

60.

61.     MPI\_Scatter(vector\_a, elements\_per\_proc, MPI\_DOUBLE,

62.                 local\_a, elements\_per\_proc, MPI\_DOUBLE,

63.                 0, MPI\_COMM\_WORLD);

64.

65.     MPI\_Scatter(vector\_b, elements\_per\_proc, MPI\_DOUBLE,

66.                 local\_b, elements\_per\_proc, MPI\_DOUBLE,

67.                 0, MPI\_COMM\_WORLD);

68.

69.     local\_dot\_product = 0.0;

70.     for (int i = 0; i < elements\_per\_proc; i++) {

71.         local\_dot\_product += local\_a[i] \* local\_b[i];

72.     }

73.

74.     MPI\_Reduce(&local\_dot\_product, &global\_dot\_product, 1, MPI\_DOUBLE,

75.                MPI\_SUM, 0, MPI\_COMM\_WORLD);

76.

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

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

79.

80.     if (rank == 0) {

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

82.         if (fp == NULL) {

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

84.         }

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

86.         fclose(fp);

87.     }

88.

89.     free(local\_a);

90.     free(local\_b);

91.

92.     if (rank == 0) {

93.         free(vector\_a);

94.         free(vector\_b);

95.     }

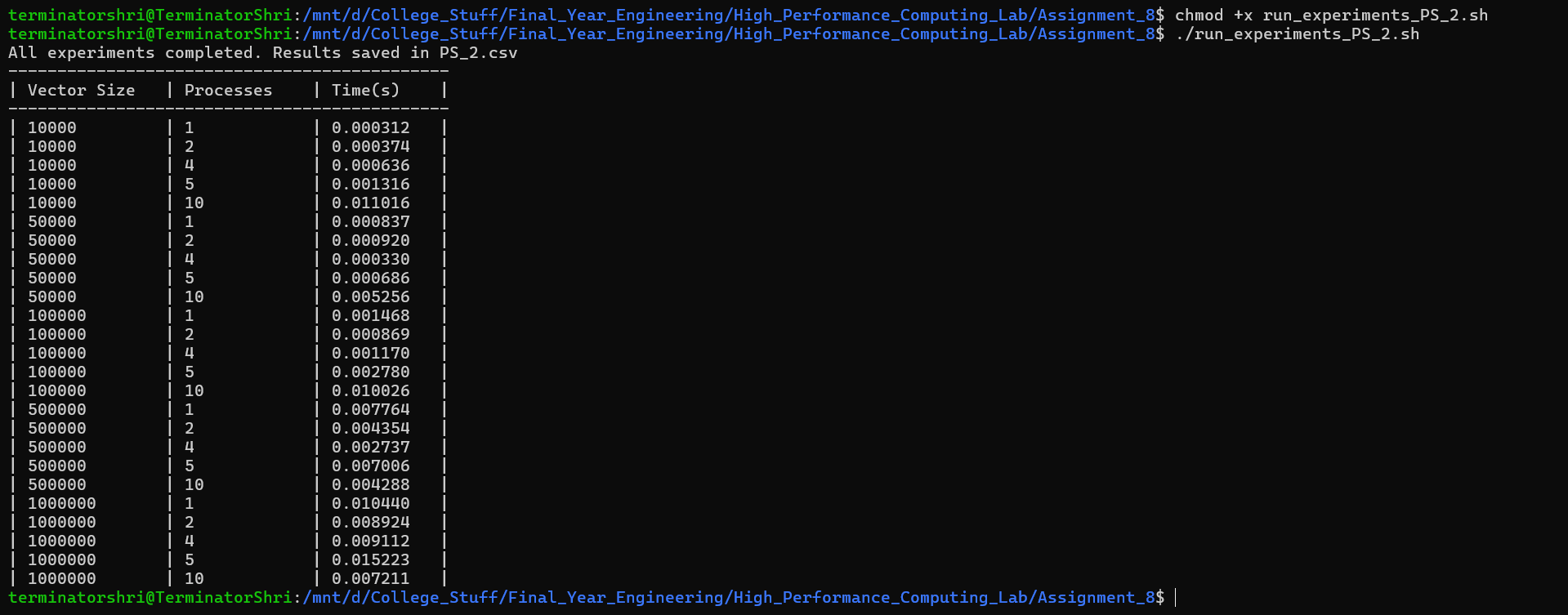
96.

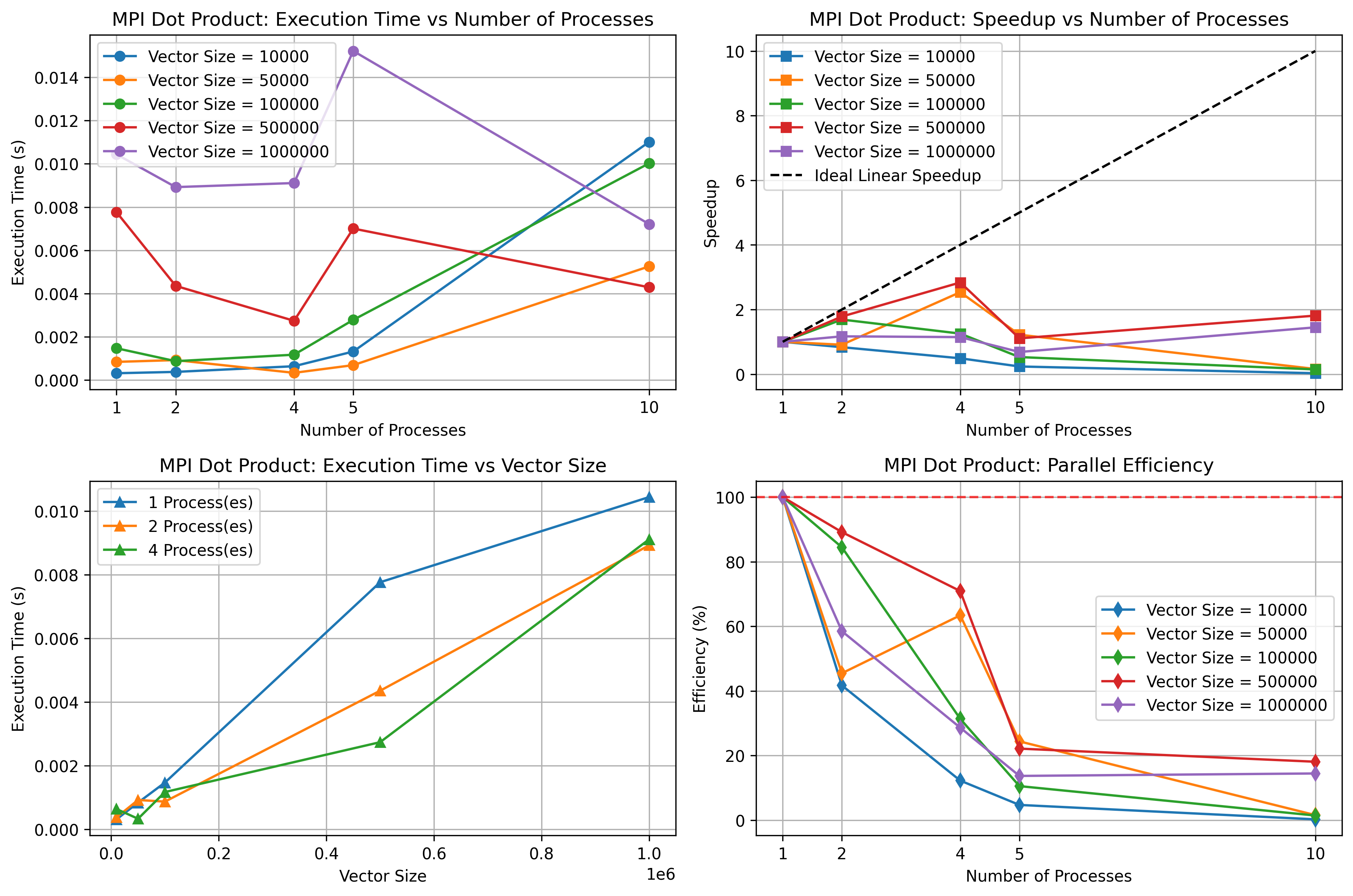
97.     MPI\_Finalize();

98.     return 0;

99. }

100.





For small vectors execution time is already tiny, and adding processes increases time due to MPI overhead. For larger vectors, time decreases slightly up to 4 processes, but then grows again. Hence is seen that, computation benefits only appear for larger vectors, otherwise overhead dominated.

Ideal speedup is far above actual results. Only for vector size 500k and 1M we see some useful speedup. Beyond 4 processes, speedup declines sharply, meaning parallel scaling is poor.

Execution time grows linearly with vector size, as expected. **Parallel versions (2 and 4 processes)** slightly reduce execution time for larger vectors, but difference is small. Confirms MPI is only beneficial when the dataset is large enough to amortize communication overhead.

For **small vectors**, efficiency collapses rapidly (<20% at 4 processes). For **larger vectors (500k, 1M)**, efficiency holds up a bit better (~60–70% with 2 processes, ~40% with 4 processes). Still, efficiency drops significantly with more than 4 processes, showing MPI communication cost dominates.