Name: Harshavardhan Bamane

PRN: 22510112

High Performance Computing Lab (B – 1)

Assignment – 5

[Title: Implementation of OpenMP Programs](https://github.com/TerminatorShri/22510025_HPCL)

1. Implementation of Matrix Multiplication

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4.

5. #define N 500

6.

7. int main() {

8.     int i, j, k;

9.     double \*\*A, \*\*B, \*\*C;

10.     double start\_time, end\_time;

11.

13.     A = (double \*\*)malloc(N \* sizeof(double \*));

14.     B = (double \*\*)malloc(N \* sizeof(double \*));

15.     C = (double \*\*)malloc(N \* sizeof(double \*));

16.     for (i = 0; i < N; i++) {

17.         A[i] = (double \*)malloc(N \* sizeof(double));

18.         B[i] = (double \*)malloc(N \* sizeof(double));

19.         C[i] = (double \*)malloc(N \* sizeof(double));

20.     }

21.

23.     for (i = 0; i < N; i++) {

24.         for (j = 0; j < N; j++) {

25.             A[i][j] = rand() % 100;

26.             B[i][j] = rand() % 100;

27.             C[i][j] = 0.0;

28.         }

29.     }

30.

32.     start\_time = omp\_get\_wtime();

33.

35.     #pragma omp parallel for private(i, j, k) shared(A, B, C)

36.     for (i = 0; i < N; i++) {

37.         for (j = 0; j < N; j++) {

38.             double sum = 0.0;

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

40.                 sum += A[i][k] \* B[k][j];

41.             }

42.             C[i][j] = sum;

43.         }

44.     }

45.

47.     end\_time = omp\_get\_wtime();

48.

49.     printf("Matrix multiplication completed in %f seconds.\n", end\_time - start\_time);

50.

51.     // Optional: Print a small matrix

52.     /\*

53.     if (N <= 5) {

54.         printf("\nMatrix C:\n");

55.         for (i = 0; i < N; i++) {

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

57.                 printf("%6.2f ", C[i][j]);

58.             }

59.             printf("\n");

60.         }

61.     }

62.     \*/

63.

65.     for (i = 0; i < N; i++) {

66.         free(A[i]);

67.         free(B[i]);

68.         free(C[i]);

69.     }

70.     free(A);

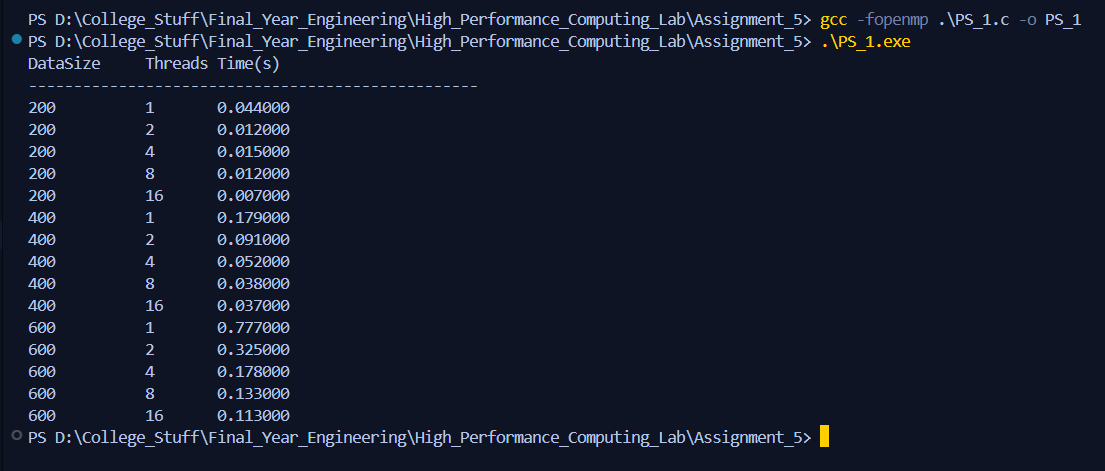
71.     free(B);

72.     free(C);

73.

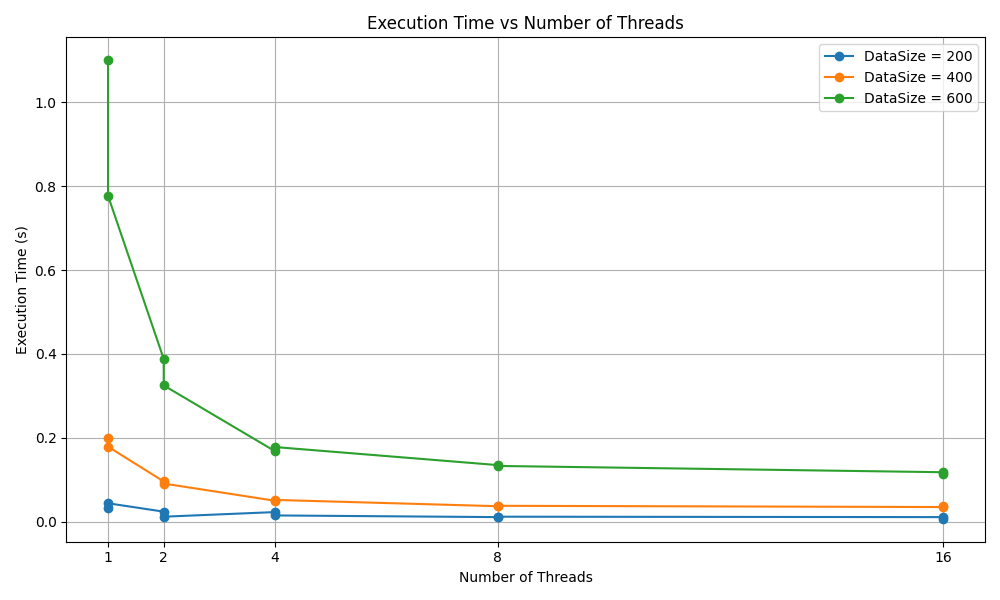
74.     return 0;

75. }

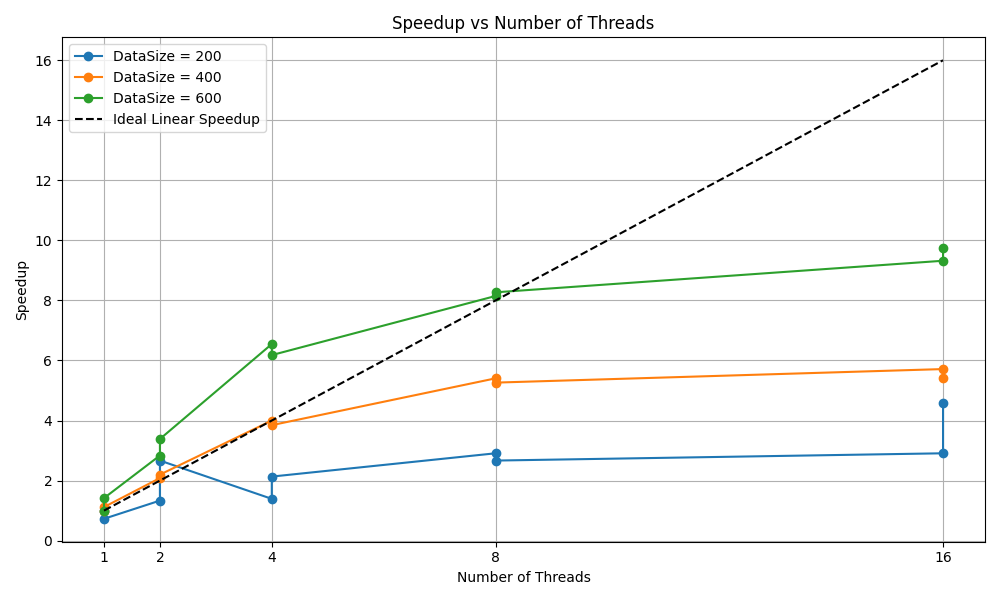


Here we distribute outer loop i across threads, each thread computing a subset of rows of C.

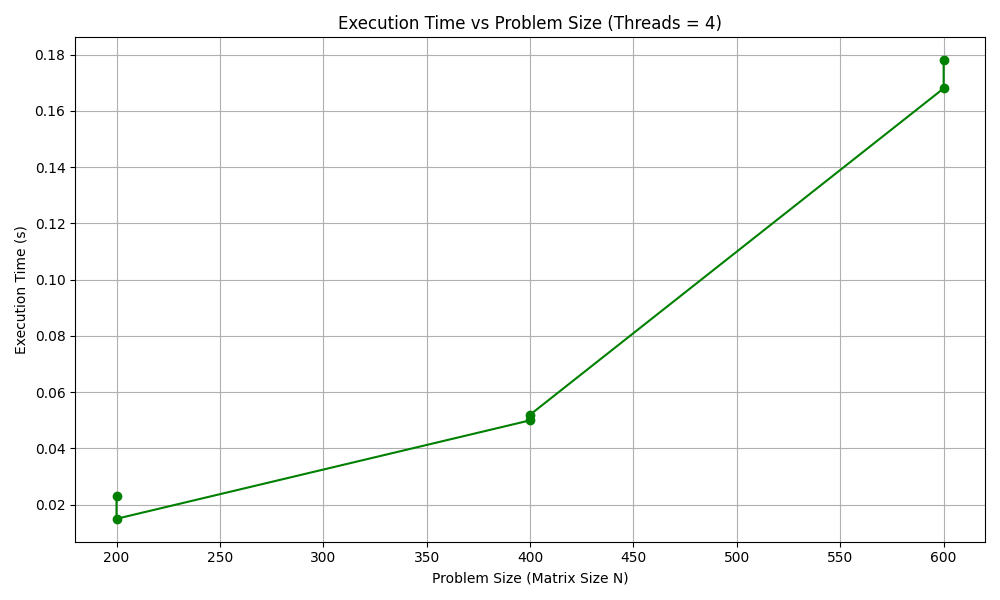
* Also, private clause ensures that each thread has its own loop counters.
* And shared clause ensures that each thread uses same input / output matrices.
* For each (i, j) cell of C, the inner loop (k) performs the **dot product** of row i of A and column j of B.



We can see execution time decreases as number of threads increase which shows clear benefit from parallelism. But after number of threads equal to 8 we see diminishing returns which can be due to overhead of thread synchronization or scheduling outweighing further parallel speedup.



For smaller data size speedup is very poor as overhead dominates due to small workload. For larger workloads scaling behaviour is best indicates the workload is large enough to exploit parallelism.



Execution time still rises sharply with N because of algorithmic complexity. For **larger N**, parallelism gives more benefits, since the computation cost grows much faster than thread management overhead.

2. Implementation of Matrix Scalar Multiplication

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4.

5. #define N 500

6. #define M 500

7.

8. int main() {

9.     int i, j;

10.     double \*\*A;

11.     double scalar = 5.0;

12.     double start\_time, end\_time;

13.

14.     A = (double \*\*)malloc(N \* sizeof(double \*));

15.     for (i = 0; i < N; i++) {

16.         A[i] = (double \*)malloc(M \* sizeof(double));

17.     }

18.

19.     for (i = 0; i < N; i++) {

20.         for (j = 0; j < M; j++) {

21.             A[i][j] = rand() % 100;

22.         }

23.     }

24.

25.     start\_time = omp\_get\_wtime();

26.

27.     #pragma omp parallel for private(i, j) shared(A, scalar)

28.     for (i = 0; i < N; i++) {

29.         for (j = 0; j < M; j++) {

30.             A[i][j] = A[i][j] \* scalar;

31.         }

32.     }

33.

34.     end\_time = omp\_get\_wtime();

35.

36.     printf("Matrix–Scalar multiplication completed in %f seconds.\n", end\_time - start\_time);

37.

38.     // Optional: Print small matrix for verification

39.     /\*

40.     if (N <= 5 && M <= 5) {

41.         printf("\nResult Matrix:\n");

42.         for (i = 0; i < N; i++) {

43.             for (j = 0; j < M; j++) {

44.                 printf("%6.2f ", A[i][j]);

45.             }

46.             printf("\n");

47.         }

48.     }

49.     \*/

50.

51.     for (i = 0; i < N; i++) {

52.         free(A[i]);

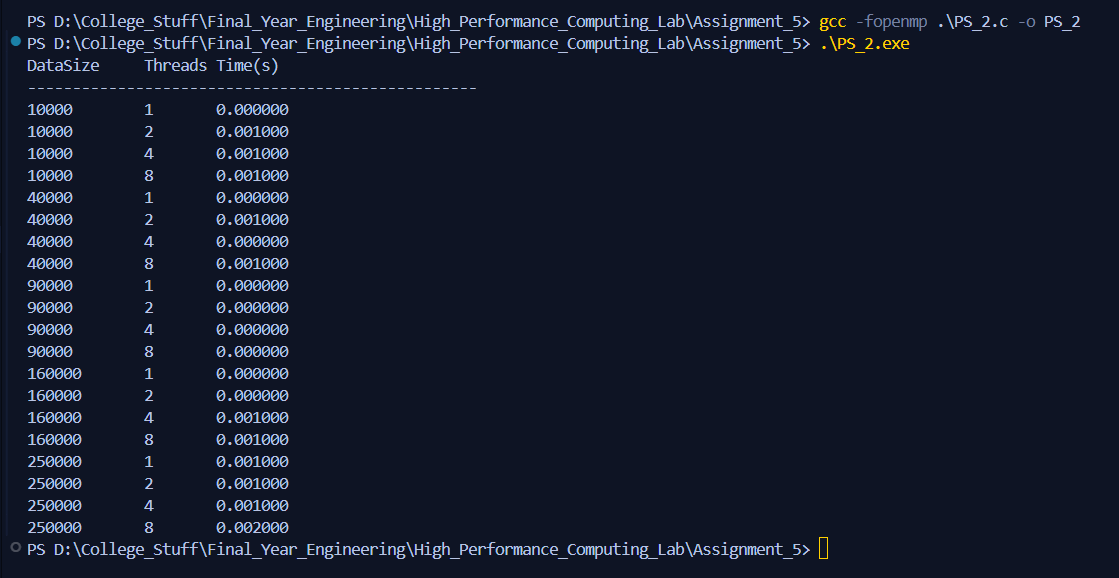
53.     }

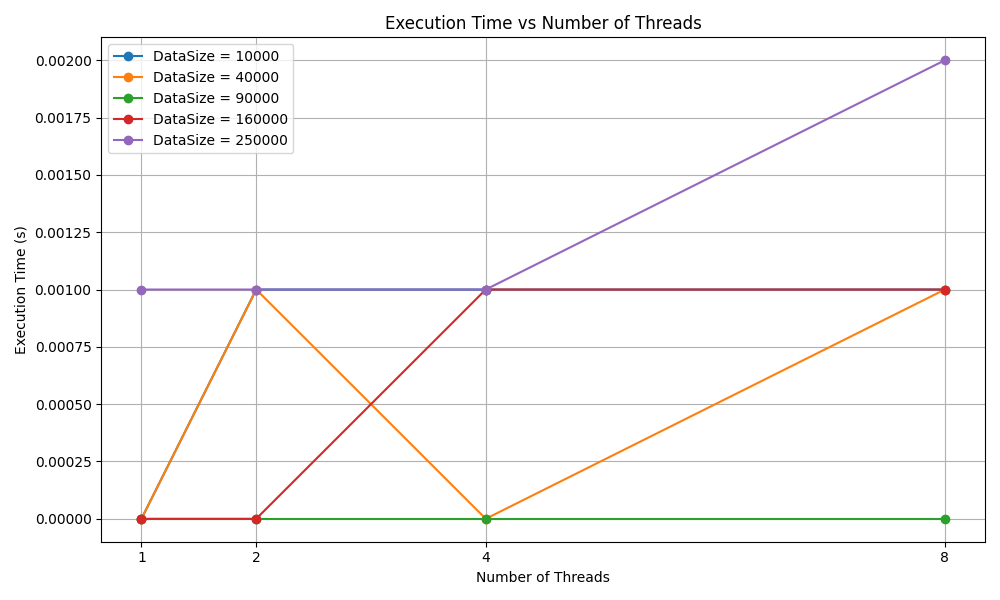
54.     free(A);

55.

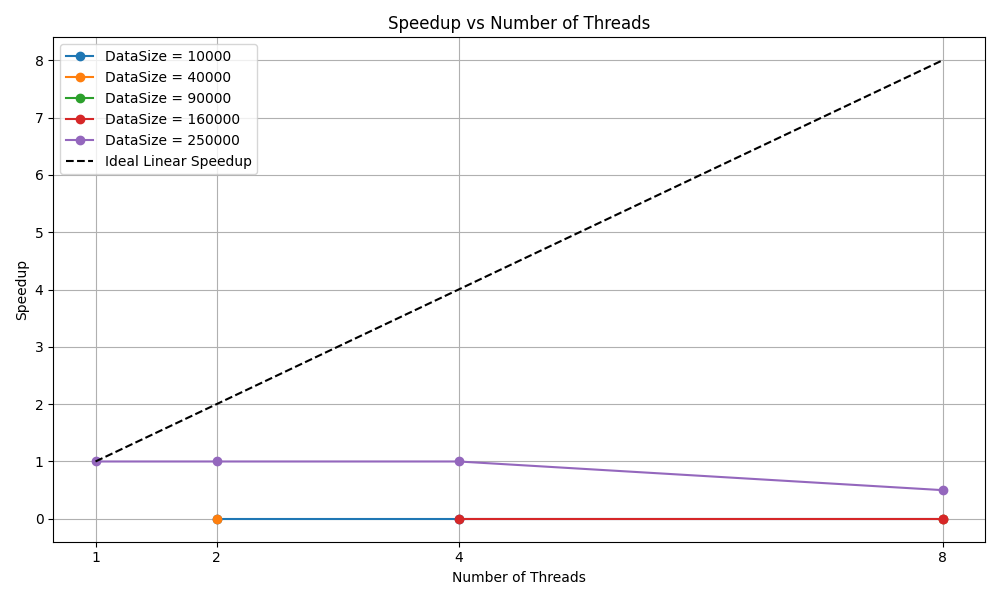
56.     return 0;

57. }

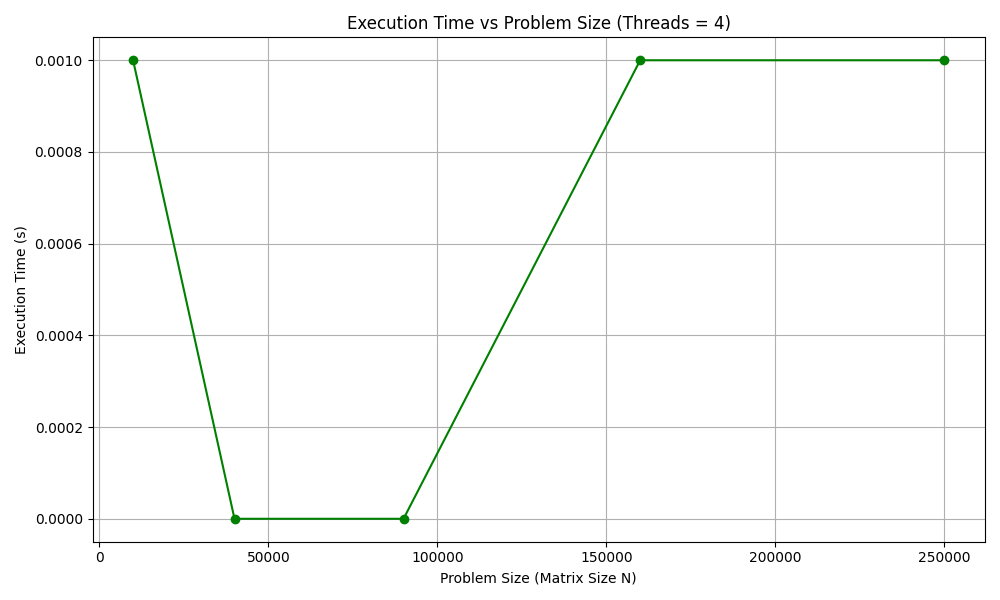




Execution time barely changes and, in some cases, it increases as number of threads increase which means parallel overhead dominates. This happens because there is not enough workload and creating and managing threads costs more than benefiting from parallelization.



Here speedup is very minimal which means parallel computation is as fast as serial which can be due to computation being minimal and operation being I/O bound and not CPU bound. Even at large sizes, **parallelism doesn’t scale** because the memory system saturates quickly.



Matrix–scalar multiplication is so fast that for moderate sizes, execution time becomes **too small to measure accurately** with standard timers. At 4 threads, parallelism doesn’t provide visible benefit, since the workload is tiny compared to OpenMP overhead.

3. Implementation of Matrix Vector Multiplication

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4.

5. #define N 500

6. #define M 500

7.

8. int main() {

9.     int i, j;

10.     double \*\*A, \*x, \*y;

11.     double start\_time, end\_time;

12.

13.     A = (double \*\*)malloc(N \* sizeof(double \*));

14.     for (i = 0; i < N; i++) {

15.         A[i] = (double \*)malloc(M \* sizeof(double));

16.     }

17.     x = (double \*)malloc(M \* sizeof(double));

18.     y = (double \*)malloc(N \* sizeof(double));

19.

20.     for (i = 0; i < N; i++) {

21.         for (j = 0; j < M; j++) {

22.             A[i][j] = rand() % 100;

23.         }

24.     }

25.     for (j = 0; j < M; j++) {

26.         x[j] = rand() % 100;

27.     }

28.

29.     start\_time = omp\_get\_wtime();

30.

31.     #pragma omp parallel for private(i, j) shared(A, x, y)

32.     for (i = 0; i < N; i++) {

33.         double sum = 0.0;

34.         for (j = 0; j < M; j++) {

35.             sum += A[i][j] \* x[j];

36.         }

37.         y[i] = sum;

38.     }

39.

40.     end\_time = omp\_get\_wtime();

41.

42.     printf("Matrix–Vector multiplication completed in %f seconds.\n", end\_time - start\_time);

43.

44.     // Optional: Print small result

45.     /\*

46.     if (N <= 5) {

47.         printf("\nResult Vector y:\n");

48.         for (i = 0; i < N; i++) {

49.             printf("%6.2f\n", y[i]);

50.         }

51.     }

52.     \*/

53.

54.     for (i = 0; i < N; i++) {

55.         free(A[i]);

56.     }

57.     free(A);

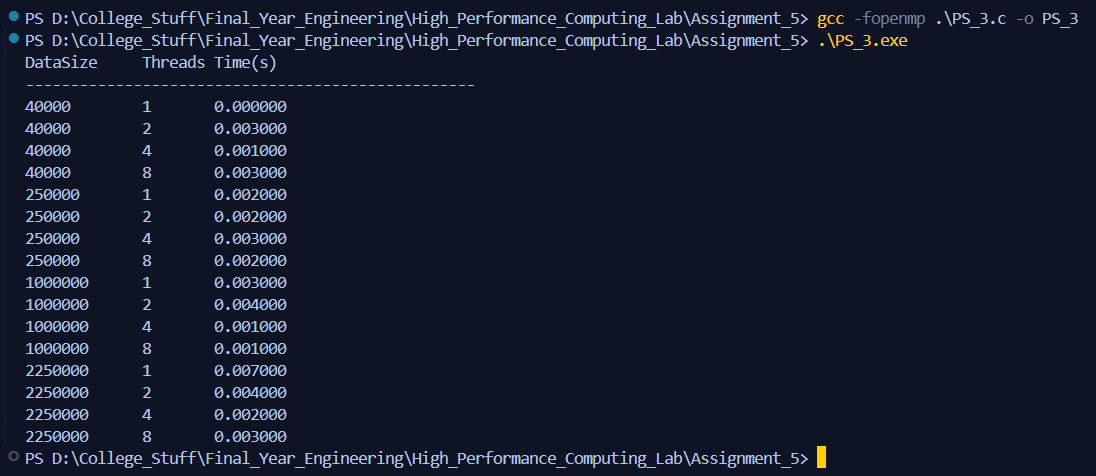
58.     free(x);

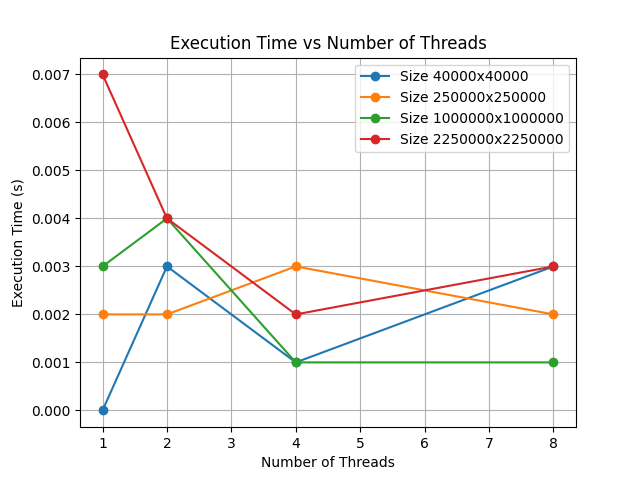
59.     free(y);

60.

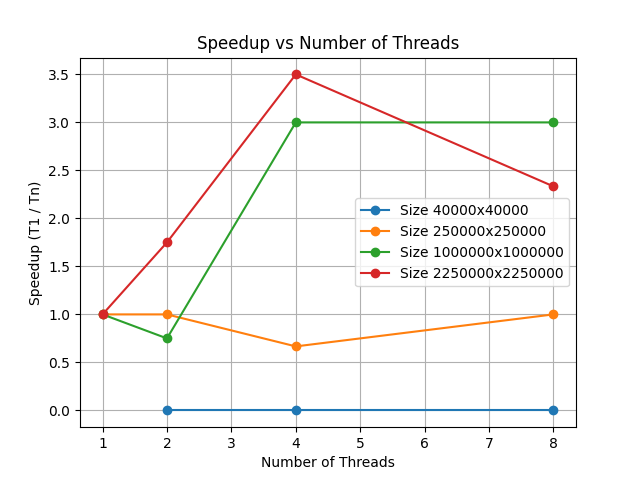
61.     return 0;

62. }

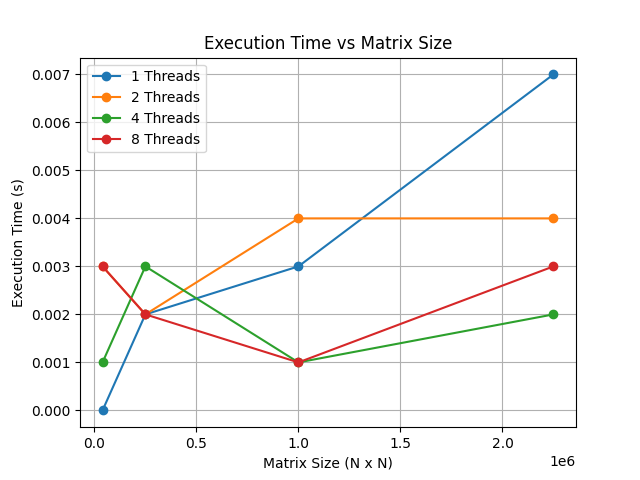




For larger sizes we see reduction in execution time and but after number of threads equal to 4 we get diminishing returns. For smaller size we can see synchronization overhead dominating as workload is not greater as compared to matrix-matrix multiplication.



This operation is memory bound and for smaller sizes overhead dominates and thus speedup is negligible. For larger workloads speedup increases significantly up to some point but then we get diminishing returns.



For **Matrix–Vector Multiplication**, the **sweet spot is 4 threads**, giving the lowest execution times across large matrix sizes. Beyond that, extra threads cause contention, not speedup.

4. Implementation of Prefix Sum

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4.

5. #define N 1000000

6.

7. int main() {

8.     int i, tid, num\_threads;

9.     double \*arr, \*prefix;

10.     double start\_time, end\_time;

11.

12.     arr = (double \*)malloc(N \* sizeof(double));

13.     prefix = (double \*)malloc(N \* sizeof(double));

14.

15.     for (i = 0; i < N; i++) {

16.         arr[i] = 1;

17.     }

18.

19.     start\_time = omp\_get\_wtime();

20.

21.     #pragma omp parallel

22.     {

23.         int id = omp\_get\_thread\_num();

24.         int nthreads = omp\_get\_num\_threads();

25.         int chunk\_size = N / nthreads;

26.         int start = id \* chunk\_size;

27.         int end = (id == nthreads - 1) ? N : start + chunk\_size;

28.

29.         prefix[start] = arr[start];

30.         for (i = start + 1; i < end; i++) {

31.             prefix[i] = prefix[i - 1] + arr[i];

32.         }

33.

34.         double block\_sum = prefix[end - 1];

35.

36.         #pragma omp barrier

37.         double offset = 0.0;

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

39.             int prev\_end = (i + 1) \* chunk\_size;

40.             offset += prefix[prev\_end - 1];

41.         }

42.

43.         if (id != 0) {

44.             for (i = start; i < end; i++) {

45.                 prefix[i] += offset;

46.             }

47.         }

48.     }

49.

50.     end\_time = omp\_get\_wtime();

51.

52.     printf("Prefix sum completed in %f seconds.\n", end\_time - start\_time);

53.

54.     // Optional: Print small prefix array

55.     /\*

56.     for (i = 0; i < 10; i++) {

57.         printf("%6.2f ", prefix[i]);

58.     }

59.     printf("\n");

60.     \*/

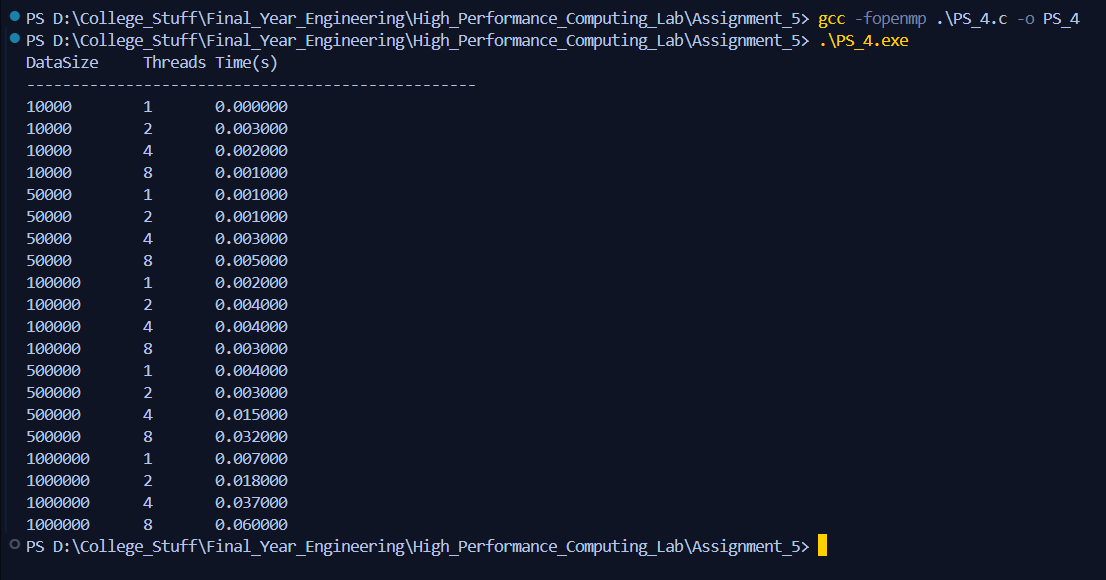
61.

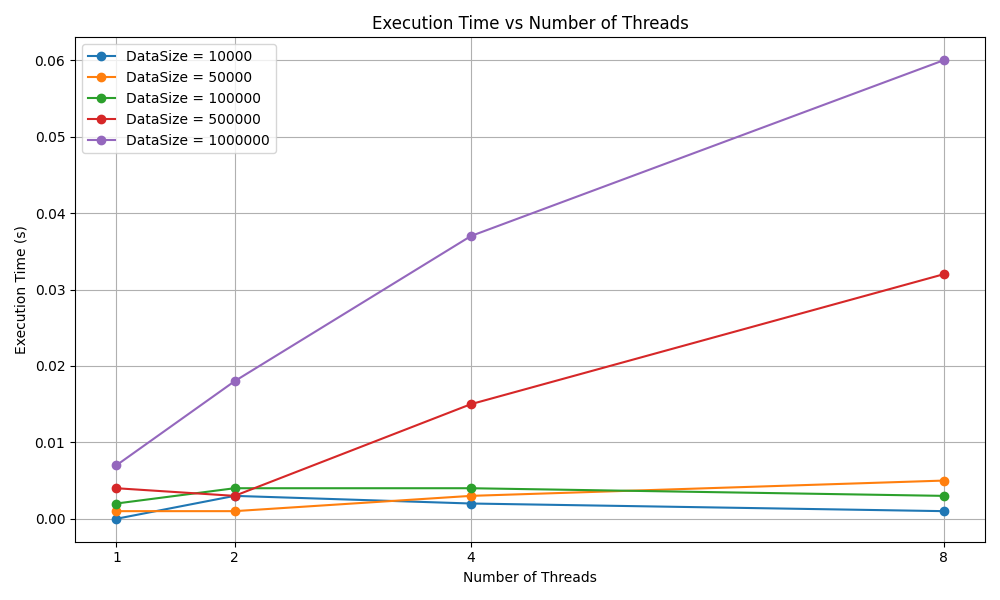
62.     free(arr);

63.     free(prefix);

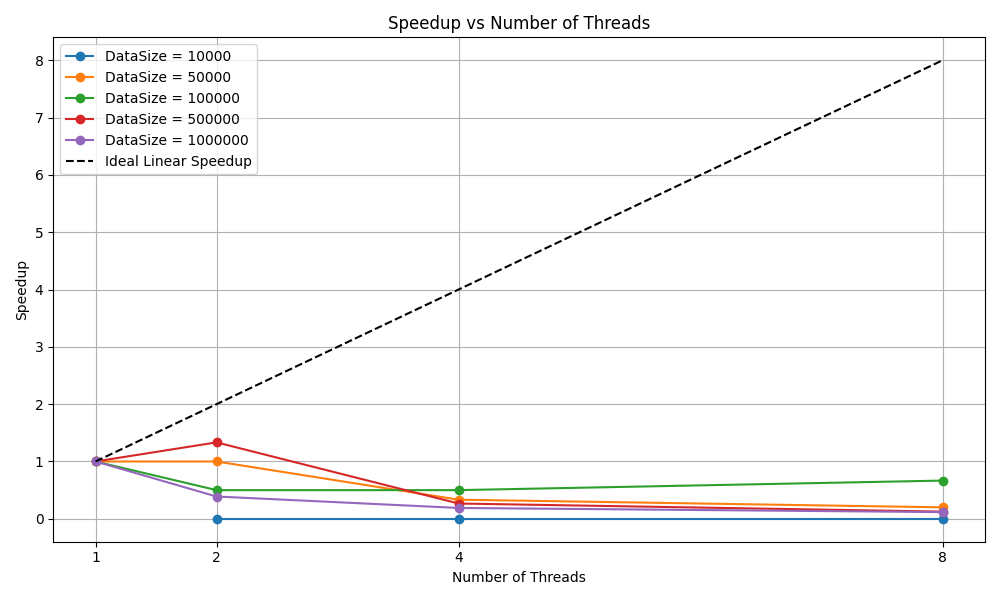
64.     return 0;

65. }

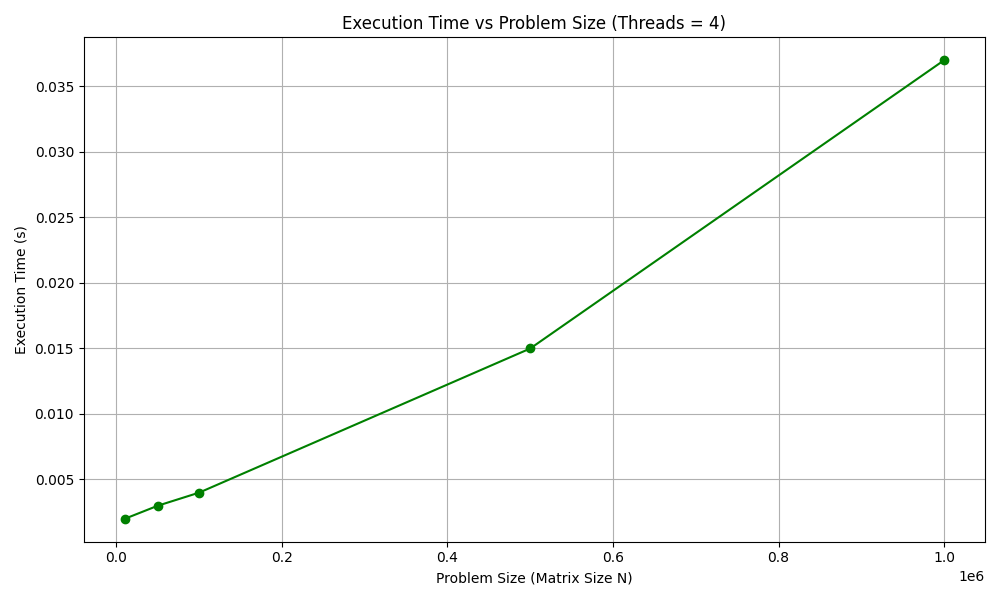




For smaller workloads performance fluctuates which indicate that **thread creation and synchronization overhead dominates** for small problem sizes. The work per thread is too small to benefit from parallelism. Also, for larger workloads execution time increases as number of threads increase which can be due to synchronization overhead as prefix sum problem is not embarrassingly parallel and a lot of time is spend in synchronization overhead.



Prefix sum having inherent data dependencies we can see there is very poor scaling performance and overhead dominates likely making things even worse by false sharing as more threads touch adjacent memory.



This means this prefix sum is **work-efficient in terms of asymptotic complexity (O(n))**, but it doesn’t benefit much from parallelism.