Name: Shreeyash Shripad Dongarkar

PRN: 22510025

High Performance Computing Lab (B – 1)

Assignment – 2

Title: Study and Implementation of Basic OpenMP Clauses

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

Implement following Programs with OpenMP in C

1) Vector Scalar Addition

Code:

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <omp.h>

4.

5. void vector\_scalar\_addition(float \*A, float \*C, float scalar, int N, int num\_threads) {

6.     #pragma omp parallel for num\_threads(num\_threads)

7.     for (int i = 0; i < N; i++) {

8.         C[i] = A[i] + scalar;

9.     }

10. }

11.

12. int main() {

13.     int N\_values[] = {1000000, 10000000, 50000000};

14.     int thread\_counts[] = {1, 2, 4, 8};

15.     float scalar = 3.5;

16.

17.     printf("%-12s %-7s %s\n", "DataSize", "Threads", "Time(s)");

18.     printf("--------------------------------------------------\n");

19.

20.     for (int ni = 0; ni < sizeof(N\_values)/sizeof(N\_values[0]); ni++) {

21.         int N = N\_values[ni];

22.

23.         float \*A = (float \*)malloc(N \* sizeof(float));

24.         float \*C = (float \*)malloc(N \* sizeof(float));

25.

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

27.             A[i] = i \* 0.5;

28.         }

29.

30.         for (int t = 0; t < sizeof(thread\_counts)/sizeof(thread\_counts[0]); t++) {

31.             int num\_threads = thread\_counts[t];

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

33.

34.             vector\_scalar\_addition(A, C, scalar, N, num\_threads);

35.

36.             double end = omp\_get\_wtime();

37.             printf("%-12d %-7d %.6f\n", N, num\_threads, end - start);

38.         }

39.

40.         free(A);

41.         free(C);

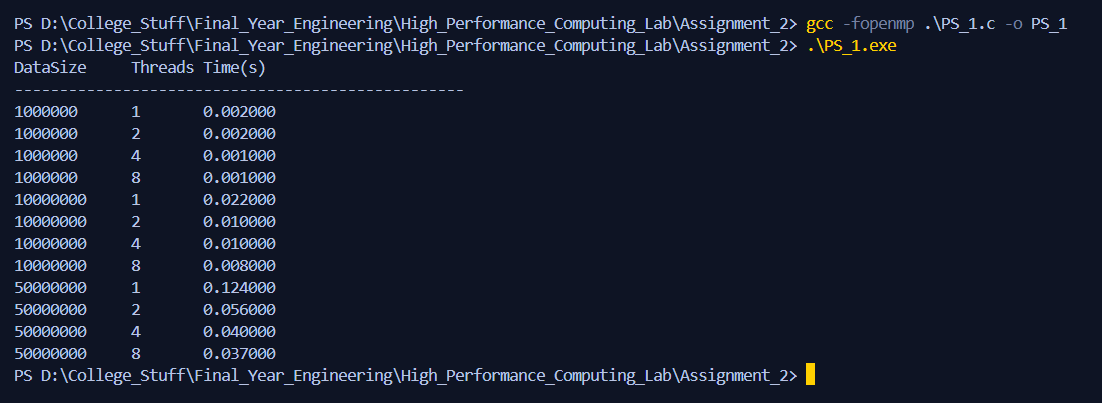
42.     }

43.

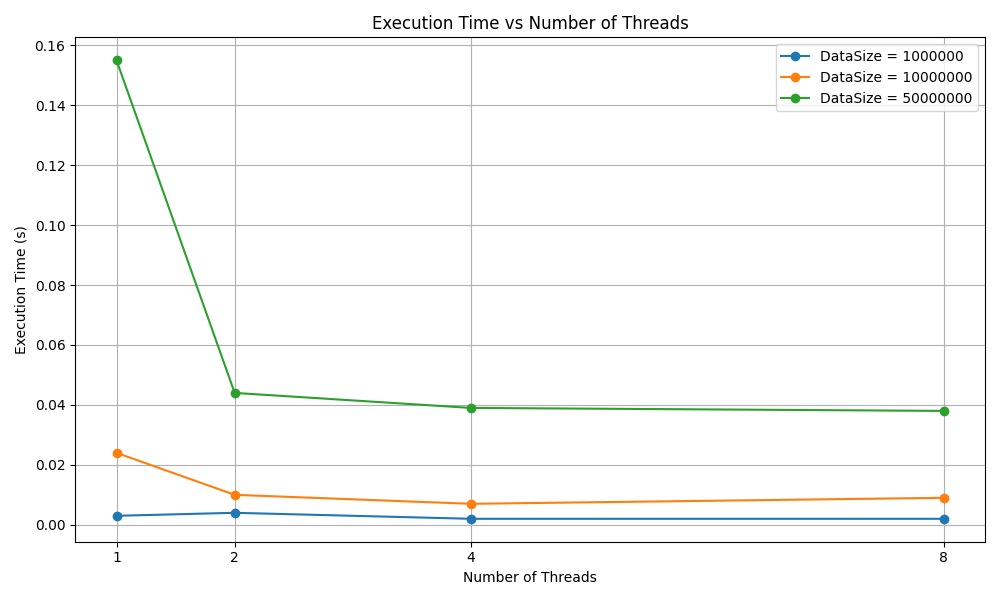
44.     return 0;

45. }

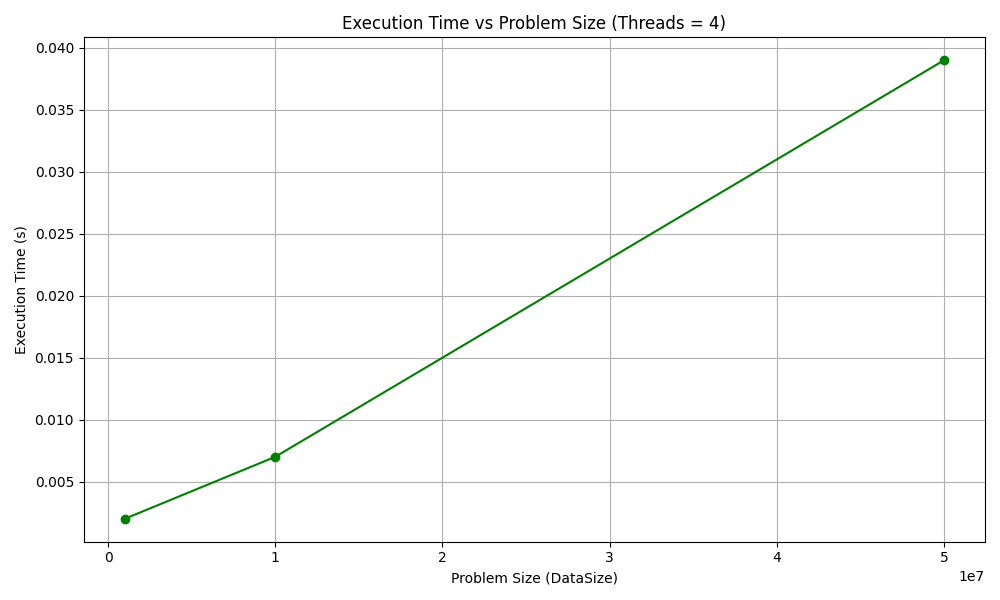
Output:

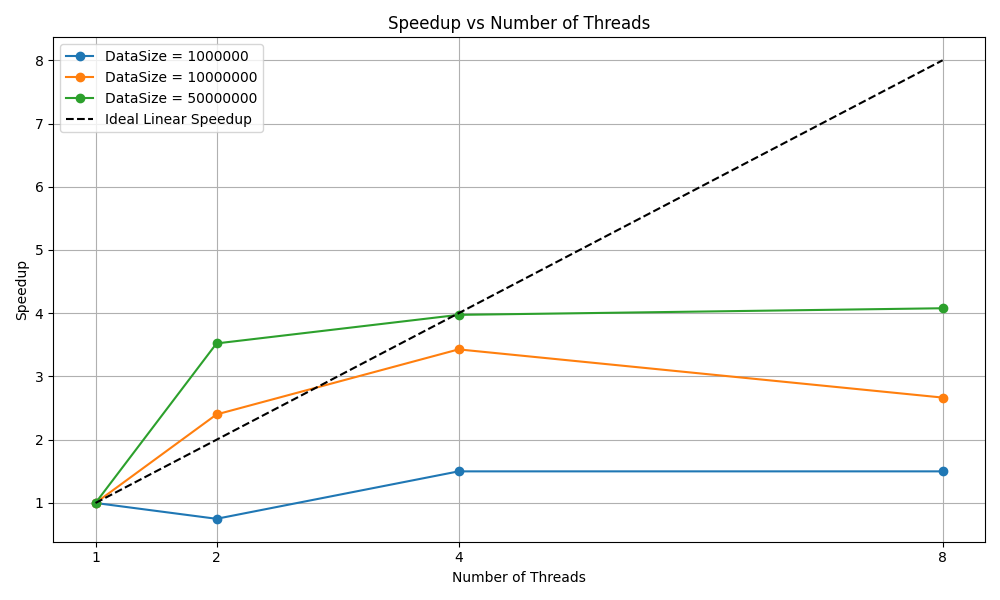


Analysis:



* We can see that for all data sizes, increasing the number of threads **reduces execution time**. The performance boost is especially noticeable when going from **1 to 2 threads**.
* After **4 threads**, the performance gain becomes minimal or even **slightly increases in time**, especially for smaller data sizes. This suggests **overhead from thread management** begins to outweigh the benefits for small workloads.
* We can also see for the **largest data size (50,000,000)**, execution time drops **significantly** proving parallelism is **more effective** with larger data sizes.





* Larger Datasets Yield Better Speedup as more work is done effectively, whereas smaller threads suffer from parallel overhead such that thread spawning and synchronization.
* But none of the data sizes reach ideal linear speedup which can be due to overheads of thread management and memory bandwidth limitations.
* Diminishing returns beyond 4 threads is observed because of plateau.

2) Calculation of Value of

We know, the Arctangent Function is:

Multiplying both sides by 4 we get,

Now we can approximate area under the curve over the interval by splitting it into small rectangles each of width

So, if we divide it into N small subintervals then,

Then for each subinterval we calculate

Hence total sum approximation becomes

For all intervals,

Code:

1. #include <stdio.h>

2. #include <omp.h>

3.

4. double calculate\_pi(int num\_steps, int num\_threads) {

5.     double step = 1.0 / (double)num\_steps;

6.     double sum = 0.0;

7.

8.     #pragma omp parallel for num\_threads(num\_threads) reduction(+:sum)

9.     for (int i = 0; i < num\_steps; i++) {

10.         double x = (i + 0.5) \* step;

11.         sum += 1.0 / (1.0 + x \* x);

12.     }

13.

14.     return 4.0 \* step \* sum;

15. }

16.

17. int main() {

18.     int steps[] = {1000000, 10000000, 50000000};

19.     int threads[] = {1, 2, 4, 8};

20.

21.     printf("%-12s %-8s %-10s %-20s\n", "Steps", "Threads", "Time(s)", "Approx. Pi");

22.

23.     for (int s = 0; s < sizeof(steps)/sizeof(steps[0]); s++) {

24.         int num\_steps = steps[s];

25.

26.         for (int t = 0; t < sizeof(threads)/sizeof(threads[0]); t++) {

27.             int num\_threads = threads[t];

28.             double start = omp\_get\_wtime();

29.

30.             double pi = calculate\_pi(num\_steps, num\_threads);

31.

32.             double end = omp\_get\_wtime();

33.             printf("%-12d %-8d %-10.6f %-20.15f\n", num\_steps, num\_threads, end - start, pi);

34.         }

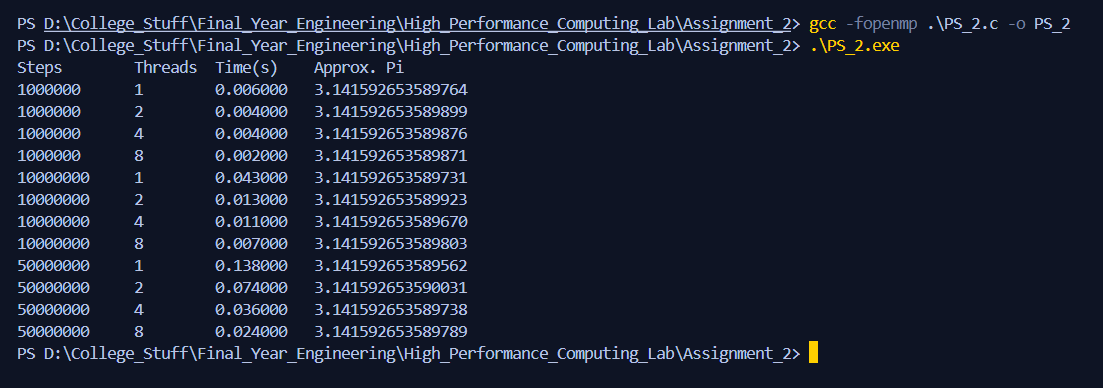
35.     }

36.

37.     return 0;

38. }

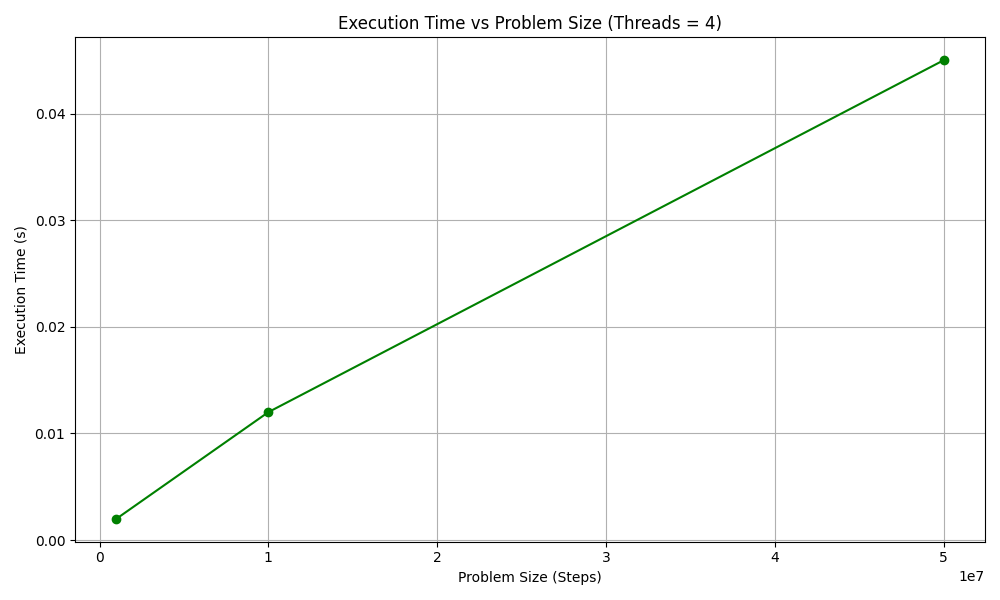
Output:

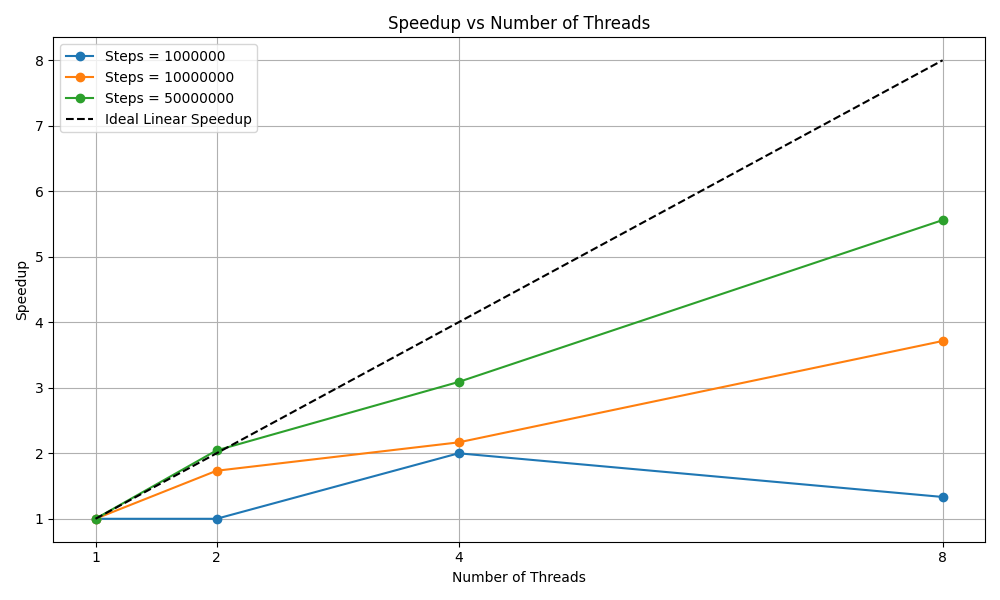


Analysis:



* There is clear reduction in execution time which shows successful parallelization.
* For smaller data size there are diminishing returns which is seen by plateau which indicates domination of parallel overheads. But for larger data size we see significant reduction.





* Larger step counts benefit from parallelization and we can see speedup increasing almost linearly for larger data sizes which implies there is more computation per thread reducing relative impact of overhead.
* Meanwhile for smaller data sizes speedup drops on increasing number of threads which means there is more overhead that parallel gain.