# Parallel And Distributed Computing

Assignment–1

Submitted to: Dr. Saif Nalband

Submitted by: Nitin Gupta
Roll No.: 102303525

February 1, 2026

# Q1. DAXPY Loop

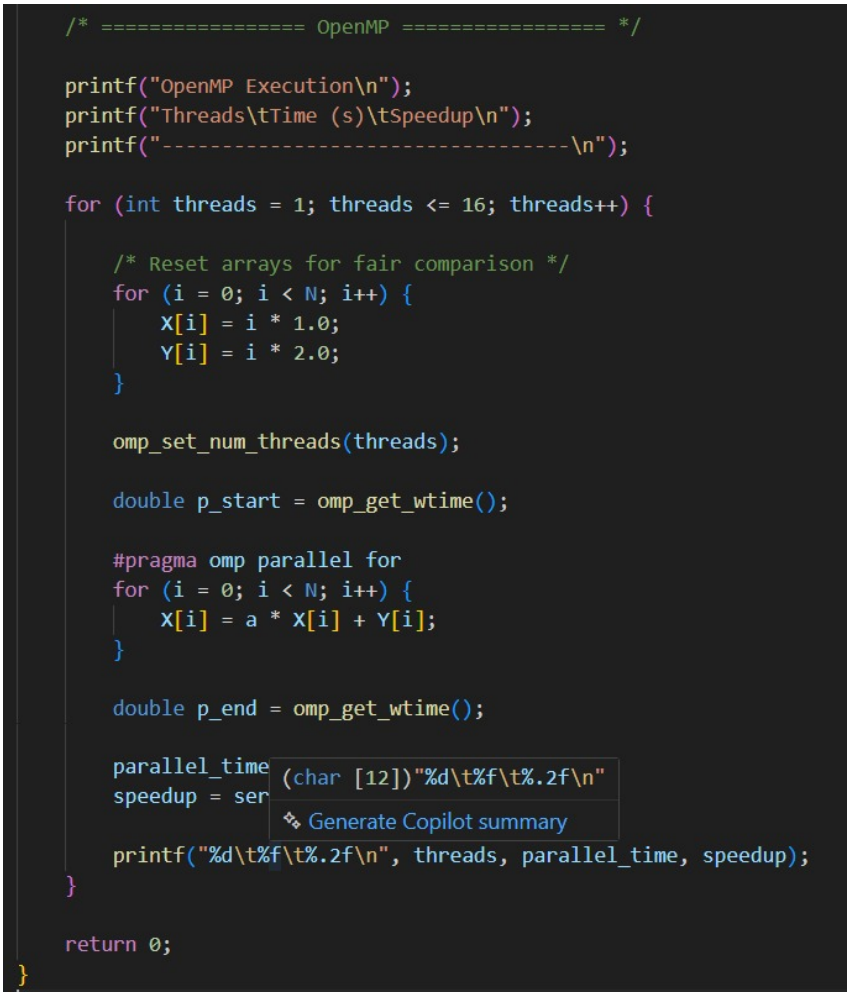
D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size $2^{16}$ each, and P stands for Plus. The operation to be completed in one iteration is:

$$X[i] = a \times X[i] + Y[i]$$

The task is to compare the speedup in execution time gained by increasing the number of threads.

## Solution

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>

#define N (1 << 24)

double X[N], Y[N];

int main() {
    double a = 2.5;
    int i;

    double serial_time;
    double parallel_time;
    double speedup;

    /* ================ SERIAL (NO OpenMP) ================ */

    for (i = 0; i < N; i++) {
        X[i] = i * 1.0;
        Y[i] = i * 2.0;
    }

    clock_t s_start = clock();

    for (i = 0; i < N; i++) {
        X[i] = a * X[i] + Y[i];
    }

    clock_t s_end = clock();

    serial_time = (double)(s_end - s_start) / CLOCKS_PER_SEC;

    printf("\n===== DAXPY Performance =====\n");
    printf("Serial Execution Time: %f seconds\n\n", serial_time);
```




## Serial execution (no OpenMP)

The entire DAXPY loop is executed by a single CPU core. This means that all vector elements are processed one after another, which results in a higher execution time for large vector sizes. In this experiment, the serial execution time was measured first and used as a baseline for calculating speedup. This baseline represents how long the computation takes when no parallelism is used.

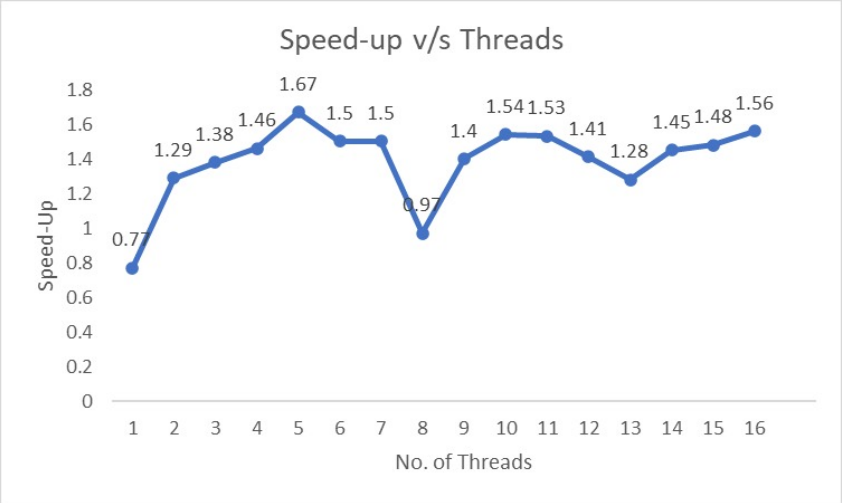
## OpenMP execution (parallel)

When OpenMP is introduced, the loop is parallelized using the `#pragma omp parallel for` directive. OpenMP divides the loop iterations among multiple threads, and each thread processes a different portion of the vector simultaneously. These threads run on different CPU cores, allowing multiple elements of the vector to be updated at the same time. The overall computation remains the same, but the workload is shared among threads, which reduces execution time.

## Observation

```
===== DAXPY Performance =====
Serial Execution Time: 0.030235 seconds

OpenMP Execution
Threads Time (s)        Speedup
---------------------------------
1       0.039387        0.77
2       0.023398        1.29
3       0.021943        1.38
4       0.020763        1.46
5       0.018148        1.67
6       0.020210        1.50
7       0.020210        1.50
8       0.031251        0.97
9       0.021596        1.40
10      0.019591        1.54
11      0.019715        1.53
12      0.021426        1.41
13      0.023624        1.28
14      0.020900        1.45
15      0.020490        1.48
16      0.019397        1.56
```

Speedup increases initially as the number of threads increases and reaches a maximum value of approximately 1.67 at 5 threads. This indicates effective utilization of all 10 physical cores. Beyond this point, speedup fluctuates and sometimes decreases as the number of threads increases further. This occurs due to thread management overhead, memory bandwidth limitations, and cache contention. Since the DAXPY operation is memory-bound and executes very quickly, parallelization overhead dominates after a certain number of threads.




## Conclusion

Maximum performance is achieved at around 5 threads. Increasing the number of threads beyond this does not improve performance due to memory bottlenecks and parallelization overhead.

# Q2. Matrix Multiplication

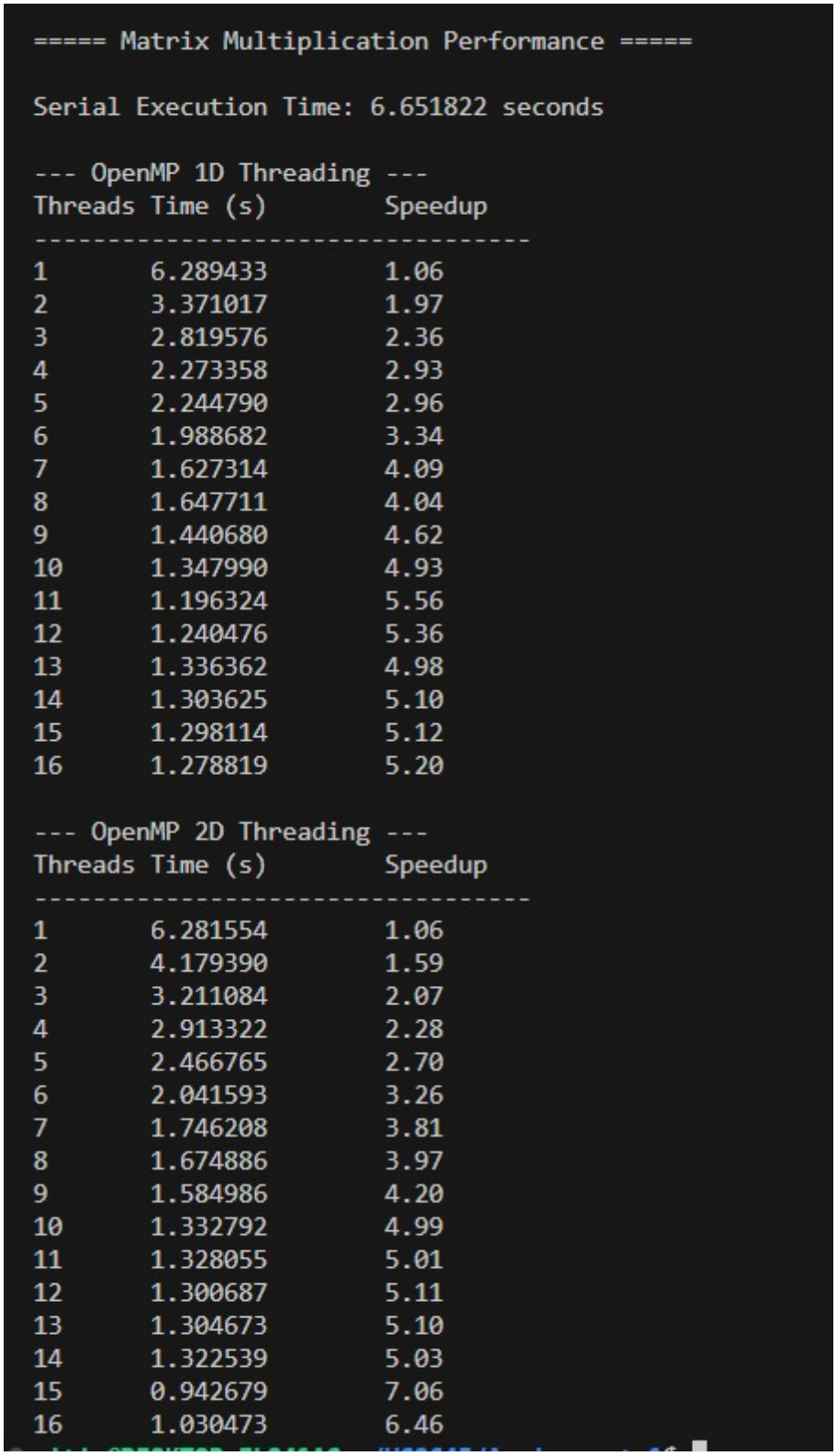
Build a parallel implementation of multiplication of large matrices (size $1000 \times 1000$). Repeat the experiment from the previous question and analyze the performance.

## Solution

```c
/* -------- SERIAL MATRIX MULTIPLICATION -------- */
double matmul_serial() {
    initialize();
    clear_C();

    double start = omp_get_wtime();

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];

    double end = omp_get_wtime();
    return (end - start);
}
```

```c
/* -------- OPENMP 1D THREADING -------- */
double matmul_1D(int threads) {
    initialize();
    clear_C();
    omp_set_num_threads(threads);

    double start = omp_get_wtime();

    #pragma omp parallel for
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];

    double end = omp_get_wtime();
    return (end - start);
}

/* -------- OPENMP 2D THREADING -------- */
double matmul_2D(int threads) {
    initialize();
    clear_C();
    omp_set_num_threads(threads);

    double start = omp_get_wtime();

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];

    double end = omp_get_wtime();
    return (end - start);
}
```

```c
int main() {

    printf("\n===== Matrix Multiplication Performance =====\n");

    /* -------- SERIAL BASELINE -------- */
    double serial_time = matmul_serial();
    printf("\nSerial Execution Time: %f seconds\n", serial_time);

    /* -------- OPENMP 1D THREADING -------- */
    printf("\n--- OpenMP 1D Threading ---\n");
    printf("Threads\tTime (s)\tSpeedup\n");
    printf("---------------------------------\n");

    for (int threads = 1; threads <= 16; threads++) {
        double time_taken = matmul_1D(threads);
        double speedup = serial_time / time_taken;
        printf("%d\t%f\t%.2f\n", threads, time_taken, speedup);
    }

    /* -------- OPENMP 2D THREADING -------- */
    printf("\n--- OpenMP 2D Threading ---\n");
    printf("Threads\tTime (s)\tSpeedup\n");
    printf("---------------------------------\n");

    for (int threads = 1; threads <= 16; threads++) {
        double time_taken = matmul_2D(threads);
        double speedup = serial_time / time_taken;
        printf("%d\t%f\t%.2f\n", threads, time_taken, speedup);
    }

    return 0;
}
```



```
===== Matrix Multiplication Performance =====

Serial Execution Time: 6.651822 seconds

--- OpenMP 1D Threading ---
Threads Time (s)        Speedup
---------------------------------
1       6.289433        1.06
2       3.371017        1.97
3       2.819576        2.36
4       2.273358        2.93
5       2.244790        2.96
6       1.988682        3.34
7       1.627314        4.09
8       1.647711        4.04
9       1.440680        4.62
10      1.347990        4.93
11      1.196324        5.56
12      1.240476        5.36
13      1.336362        4.98
14      1.303625        5.10
15      1.298114        5.12
16      1.278819        5.20

--- OpenMP 2D Threading ---
Threads Time (s)        Speedup
---------------------------------
1       6.281554        1.06
2       4.179390        1.59
3       3.211084        2.07
4       2.913322        2.28
5       2.466765        2.70
6       2.041593        3.26
7       1.746208        3.81
8       1.674886        3.97
9       1.584986        4.20
10      1.332792        4.99
11      1.328055        5.01
12      1.300687        5.11
13      1.304673        5.10
14      1.322539        5.03
15      0.942679        7.06
16      1.030473        6.46
```

## Serial Implementation

In the serial implementation, each element of the result matrix is calculated using a triple nested loop. A single CPU core computes every element of the result matrix one by one. This results in a high execution time since the total number of operations is very large.

## 1D Threading

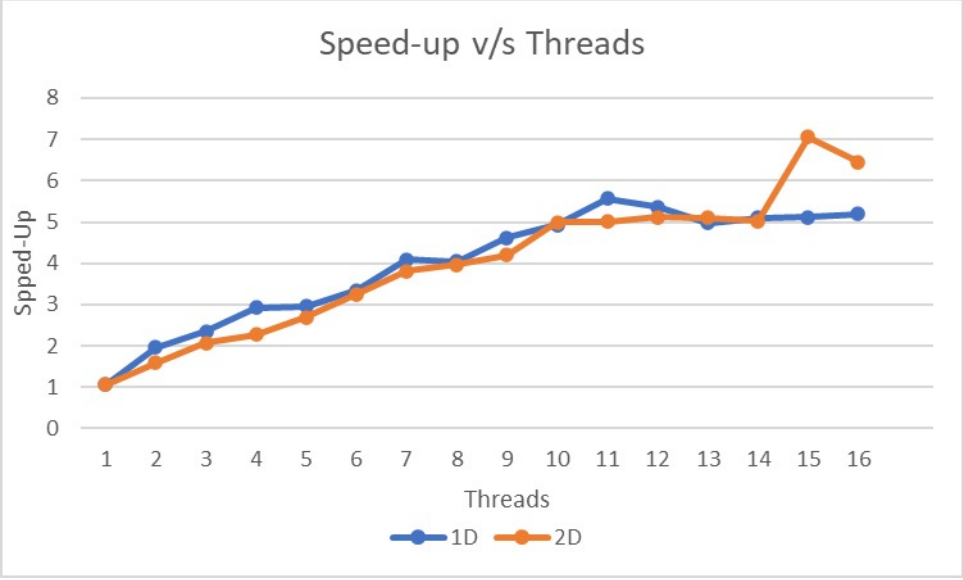
Only the outer loop of the matrix multiplication is parallelized. This means that each thread is assigned a set of rows of the result matrix to compute. Each thread processes all columns and inner-loop calculations for its assigned rows. As the number of threads increases, execution time decreases steadily, showing good speedup up to higher thread counts.

## 2D Threading

Two nested loops (row and column indices) are parallelized using OpenMP's `collapse(2)` clause. In this approach, the computation of individual elements of the result matrix is distributed more evenly among threads. This leads to better load balancing, especially when a larger number of threads is used.

## Observation

The serial execution time was approximately 6.65 seconds, which served as the baseline for speedup calculation. In the 1D OpenMP implementation, speedup increased gradually with the number of threads and reached a maximum value of around 5.56 at 11 threads. Beyond this point, the speedup fluctuated slightly but did not increase significantly, indicating that hardware limits were being reached. In the 2D OpenMP implementation, speedup also increased with the number of threads and achieved an even higher maximum speedup of approximately 7.06 at 15 threads. This shows that 2D threading provides better scalability at higher thread counts due to improved load balancing.



## Conclusion

Parallel matrix multiplication significantly improves performance compared to serial execution. The 1D threading approach performs well with lower overhead and is effective for moderate thread counts, while the 2D threading approach provides better scalability and higher speedup at larger thread counts.

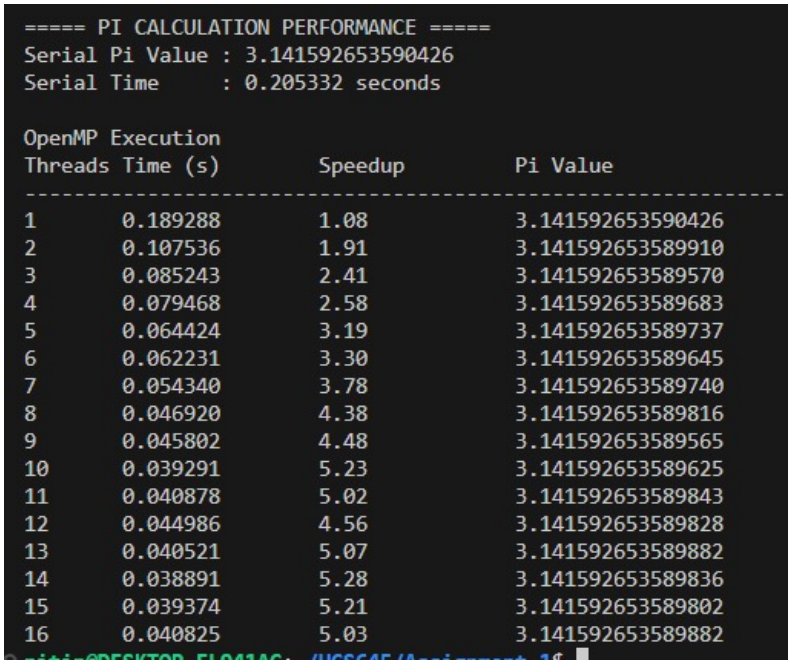
# Q3. Calculation of $\pi$

This experiment demonstrates how multiple threads cooperate to calculate an approximate value of $\pi$. The value of $\pi$ is given by the following integral:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

## Solution

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>

static long NUM_STEPS = 100000000;

int main() {

    double step = 1.0 / (double)NUM_STEPS;
    double sum, pi;
    double serial_time, parallel_time, speedup;

    /* ================ SERIAL (NO OpenMP) ================ */

    sum = 0.0;
    clock_t s_start = clock();

    for (long i = 0; i < NUM_STEPS; i++) {
        double x = (i + 0.5) * step;
        sum += 4.0 / (1.0 + x * x);
    }

    pi = step * sum;
    clock_t s_end = clock();

    serial_time = (double)(s_end - s_start) / CLOCKS_PER_SEC;

    printf("\n===== PI CALCULATION PERFORMANCE =====\n");
    printf("Serial Pi Value : %.15f\n", pi);
    printf("Serial Time     : %f seconds\n\n", serial_time);

    /* ================ OpenMP ================ */

    printf("OpenMP Execution\n");
    printf("Threads\tTime (s)\tSpeedup\t\tPi Value\n");
    printf("-------------------------------------------------------------\n");

    for (int threads = 1; threads <= 16; threads++) {

        sum = 0.0;
        omp_set_num_threads(threads);

        double p_start = omp_get_wtime();

        #pragma omp parallel for reduction(+:sum)
        for (long i = 0; i < NUM_STEPS; i++) {
            double x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }

        pi = step * sum;
        double p_end = omp_get_wtime();

        parallel_time = p_end - p_start;
        speedup = serial_time / parallel_time;

        printf("%d\t%f\t%.2f\t\t%.15f\n",
               threads, parallel_time, speedup, pi);
    }

    return 0;
}
```



```
===== PI CALCULATION PERFORMANCE =====
Serial Pi Value : 3.141592653590426
Serial Time     : 0.205332 seconds

OpenMP Execution
Threads Time (s)        Speedup         Pi Value
-----------------------------------------------------------------
1       0.189288        1.08            3.141592653590426
2       0.107536        1.91            3.141592653589910
3       0.085243        2.41            3.141592653589570
4       0.079468        2.58            3.141592653589683
5       0.064424        3.19            3.141592653589737
6       0.062231        3.30            3.141592653589645
7       0.054340        3.78            3.141592653589740
8       0.046920        4.38            3.141592653589816
9       0.045802        4.48            3.141592653589565
10      0.039291        5.23            3.141592653589625
11      0.040878        5.02            3.141592653589843
12      0.044986        4.56            3.141592653589828
13      0.040521        5.07            3.141592653589882
14      0.038891        5.28            3.141592653589836
15      0.039374        5.21            3.141592653589802
16      0.040825        5.03            3.141592653589882
```

## Serial Implementation

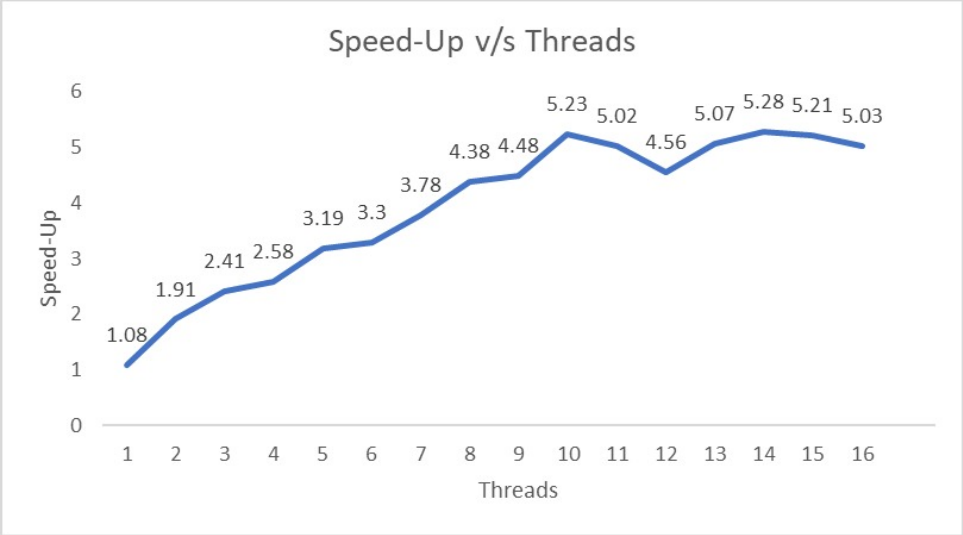
In the serial implementation, the value of $\pi$ is computed by evaluating the numerical integral using a single CPU core, where all iterations are executed one after another. This approach gives an accurate value of $\pi$ but takes more time for a large number of steps.

## OpenMP Implementation

In the OpenMP implementation, the loop iterations are divided among multiple threads, and each thread computes a partial sum. These partial sums are combined using the reduction clause to avoid race conditions. As the number of threads increases, the execution time decreases and speedup improves. However, beyond the number of physical CPU cores, the speedup saturates due to thread management overhead.

## Observation

The serial implementation computes $\pi$ accurately with an execution time of about 0.20 seconds. Using OpenMP, execution time decreases as the number of threads increases, and speedup improves from about 1.08 (1 thread) to a maximum of approximately 5.28 at 14 threads. Beyond this point, speedup shows only minor improvement due to thread management overhead. The computed value of $\pi$ remains close to 3.14159 for all thread counts, confirming correct parallel execution.




## Conclusion

The program computes an accurate approximation of $\pi$. Parallelism is beneficial only up to physical CPU cores, after which overhead dominates.