Parallel and Distributed Computing

Assignment 1

Question 1: DAXPY Loop

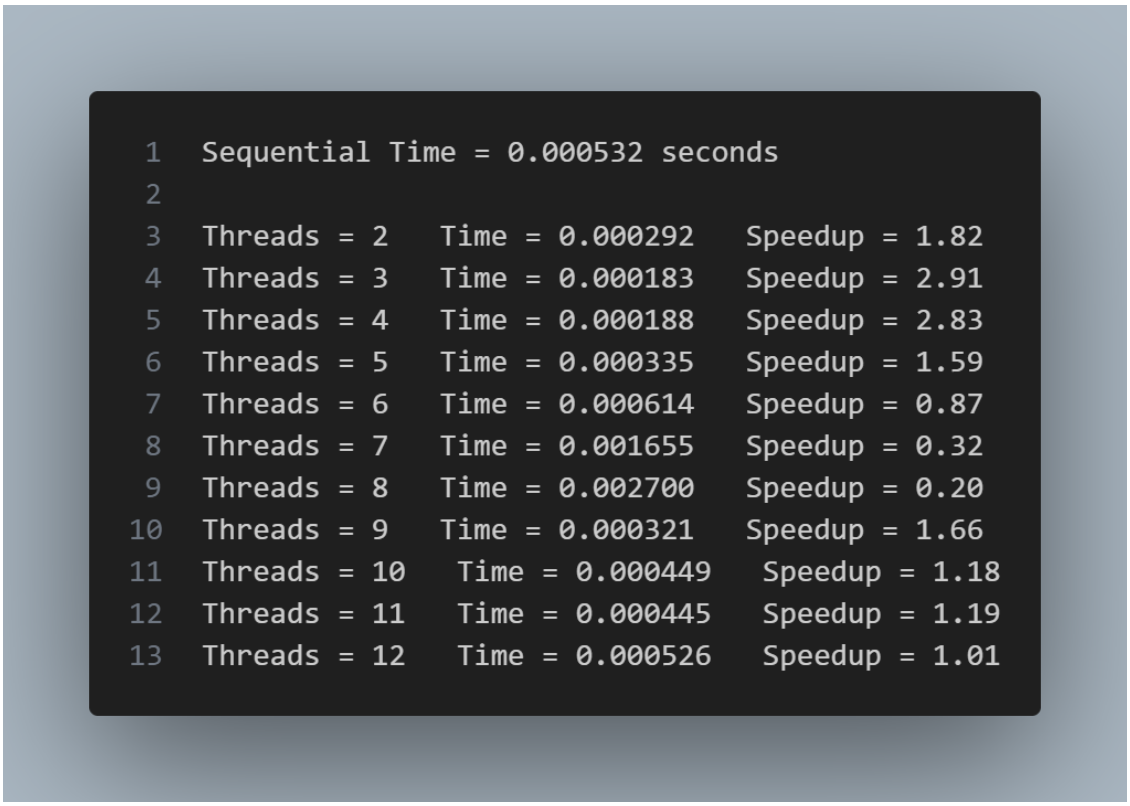
Problem Statement:

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 performed in one iteration is:

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

The objective is to compare the speedup (in execution time) gained by increasing the number of threads starting from a 2-thread implementation.

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

int main() {
    int N = 65536;
    double X[N], Y[N];
    double a = 2.5;
    int i, threads;
    double t1, t2;
    double seq_time;

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

    // measure sequential time
    for (i = 0; i < N; i++)
        X[i] = 1.0;

    t1 = omp_get_wtime();
    for (i = 0; i < N; i++) {
        X[i] = a * X[i] + Y[i];
    }
    t2 = omp_get_wtime();
    seq_time = t2 - t1;

    printf("Sequential Time = %f seconds\n\n", seq_time);

    // parallel execution with 2 to 12 threads
    for (threads = 2; threads <= 12; threads++) {
        double par_time, speedup;

        omp_set_num_threads(threads);

        for (i = 0; i < N; i++)
            X[i] = 1.0;

        t1 = omp_get_wtime();
        #pragma omp parallel for
        for (i = 0; i < N; i++) {
            X[i] = a * X[i] + Y[i];
        }
        t2 = omp_get_wtime();

        par_time = t2 - t1;
        speedup = seq_time / par_time;

        printf("Threads = %d   Time = %f   Speedup = %.2f\n", threads, par_time, speedu
p);}

    return 0;
}

```



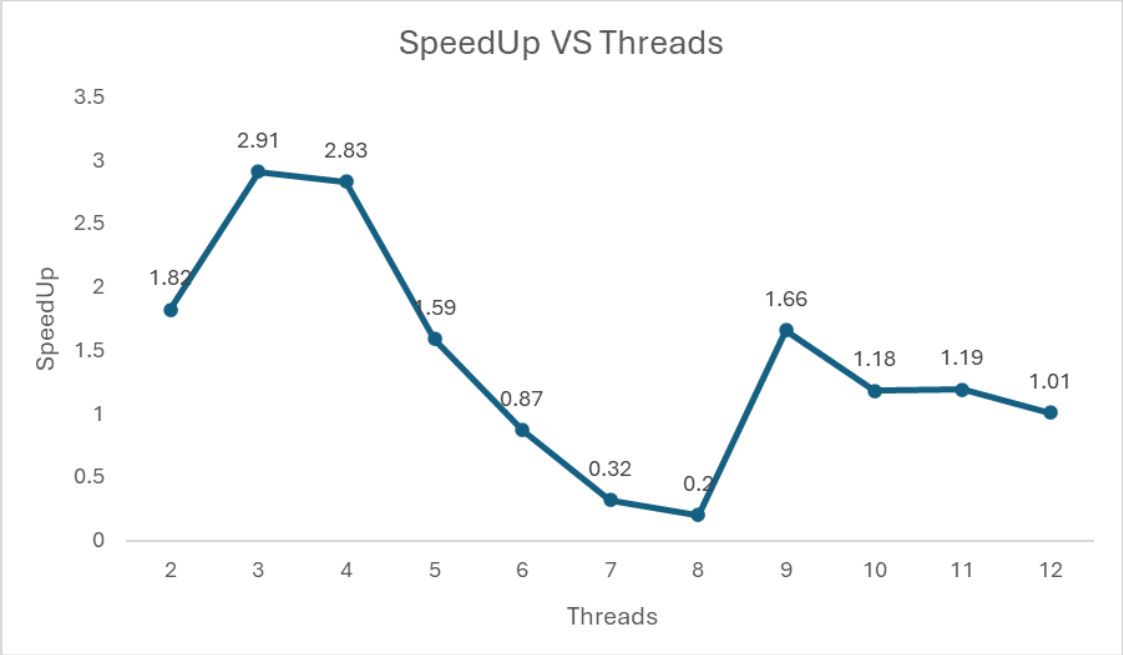
Observation

Speedup rises till 4 threads and achieves maximum speedup at around 3–4 cores (approximately 2.9). The synchronization overhead is small. Since the laptop has 4 physical cores, execution remains within the physical core limit.

After 4 threads (5–8 threads), a stable dip in speedup is observed. This occurs because execution goes beyond physical cores and hyperthreading is used. Hyperthreading introduces thread creation and synchronization overhead, leading to memory bottlenecks and cache conflicts.

At around 9 threads, slight recovery is observed as the scheduler adapts, but speedup remains lower than the maximum achieved.

The DAXPY loop is memory bound and executes very fast, hence overhead dominates computation.



Conclusion

Best performance is achieved using 1–4 threads corresponding to physical cores. Beyond this point, parallelization overhead outweighs computation.

Question 2: Matrix Multiplication

Problem Statement:

Build a parallel implementation of multiplication of large matrices (e.g., 1000×1000). Repeat the experiment from Question 1 and analyze work partitioning among threads using:

- 1D threading
- 2D threading

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

double A[1000][1000], B[1000][1000], C[1000][1000];

int main()
{
    int N = 1000;
    int i, j, k, threads;
    double t1, t2, seq_time;

    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            A[i][j] = 1.0;
            B[i][j] = 2.0;
            C[i][j] = 0.0;
        }
    }

    // sequential
    printf("Sequential:\n");
    t1 = omp_get_wtime();
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            for (k = 0; k < N; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    t2 = omp_get_wtime();
    seq_time = t2 - t1;
    printf("Time = %f\n\n", seq_time);

    // 1D parallel
    printf("1D Parallel:\n");
    for (threads = 2; threads <= 10; threads++)
    {
        double par_time, speedup;

        omp_set_num_threads(threads);

        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                C[i][j] = 0.0;

        t1 = omp_get_wtime();
#pragma omp parallel for private(j, k)
        for (i = 0; i < N; i++)
```

```
{
            for (j = 0; j < N; j++)
            {
                for (k = 0; k < N; k++)
                {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        t2 = omp_get_wtime();

        par_time = t2 - t1;
        speedup = seq_time / par_time;

        printf("Threads=%d  Time=%f  Speedup=%.2f\n", threads, par_time, speedup);
    }

    // 2D parallel
    printf("\n2D Parallel:\n");
    for (threads = 2; threads <= 10; threads++)
    {
        double par_time, speedup;

        omp_set_num_threads(threads);

        for (i = 0; i < N; i++)
            for (j = 0; j < N; j++)
                C[i][j] = 0.0;

        t1 = omp_get_wtime();
#pragma omp parallel for collapse(2) private(k)
        for (i = 0; i < N; i++)
        {
            for (j = 0; j < N; j++)
            {
                for (k = 0; k < N; k++)
                {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        t2 = omp_get_wtime();

        par_time = t2 - t1;
        speedup = seq_time / par_time;

        printf("Threads=%d  Time=%f  Speedup=%.2f\n", threads, par_time, speedup);
    }

    return 0;
}

```

```
Sequential:
Time = 7.773309

1D Parallel:
Threads=2 Time=4.312744 Speedup=1.80
Threads=3 Time=3.151255 Speedup=2.47
Threads=4 Time=3.574226 Speedup=2.17
Threads=5 Time=2.714597 Speedup=2.86
Threads=6 Time=2.642308 Speedup=2.94
Threads=7 Time=2.780996 Speedup=2.80
Threads=8 Time=2.466476 Speedup=3.15
Threads=9 Time=2.566586 Speedup=3.03
Threads=10 Time=2.686164 Speedup=2.89

2D Parallel:
Threads=2 Time=4.730801 Speedup=1.64
Threads=3 Time=3.271672 Speedup=2.38
Threads=4 Time=3.160151 Speedup=2.46
Threads=5 Time=2.605118 Speedup=2.98
Threads=6 Time=2.594382 Speedup=3.00
Threads=7 Time=2.527724 Speedup=3.08
Threads=8 Time=2.466996 Speedup=3.15
Threads=9 Time=2.576557 Speedup=3.02
Threads=10 Time=2.643251 Speedup=2.94
```

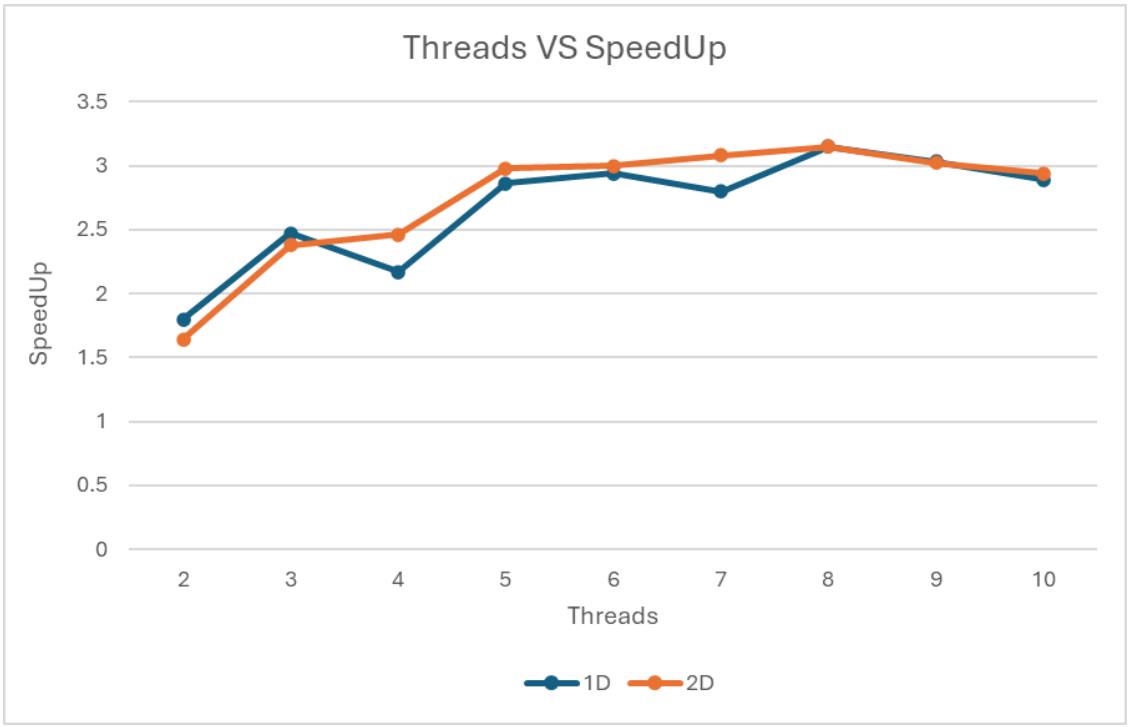
Observation

The sequential version uses a triple nested loop to compute each output element.

In the 1D parallel implementation, the outer loop is parallelized and each thread computes a subset of rows, resulting in low overhead.

In the 2D parallel implementation, both (i, j) indices are parallelized. This distributes work more evenly but increases scheduling overhead.

For 2–3 threads, overhead dominates and 1D performs better. For 4–8 threads, 2D slightly outperforms 1D due to improved load balancing. Cache effects and memory bandwidth limit perfect scaling.



Conclusion

Both 1D and 2D implementations are correct and scale well up to logical cores. Similar performance is observed due to moderate workload size.

Question 3: Calculation of π

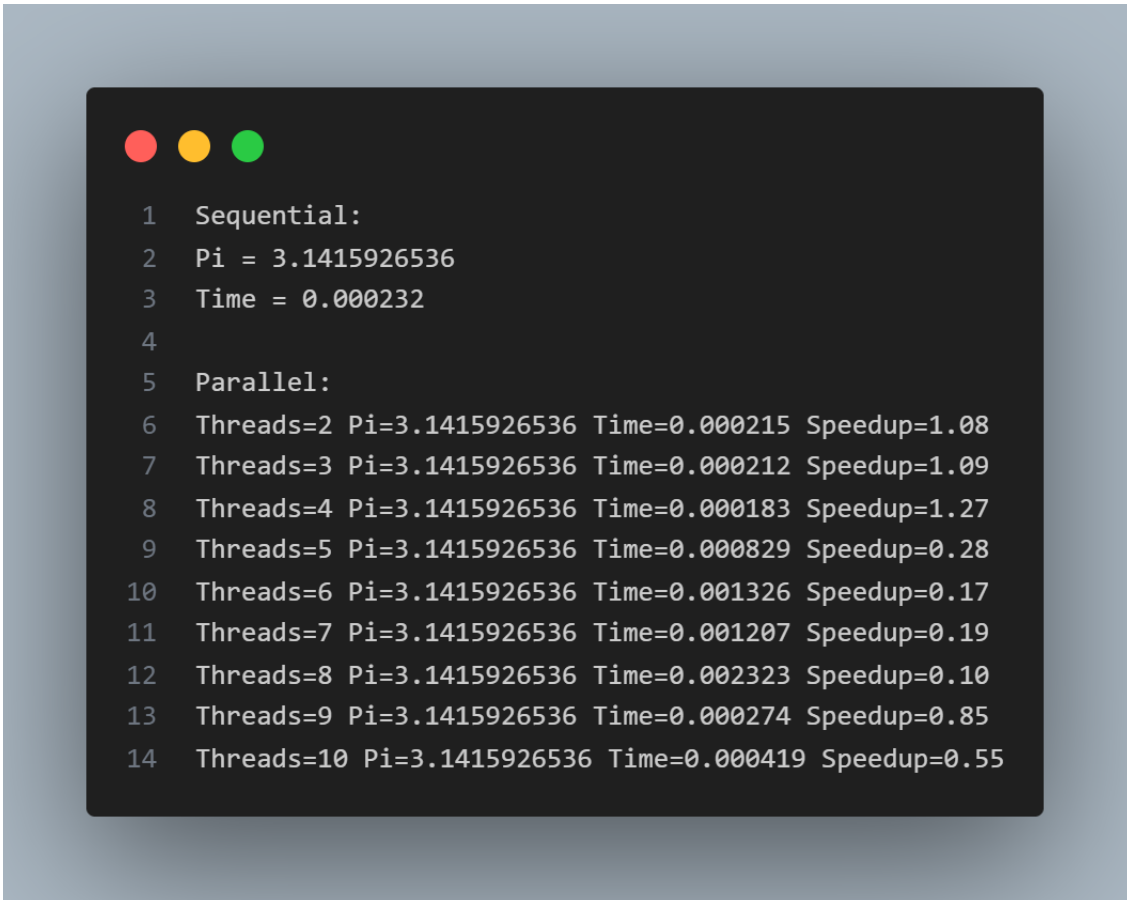
Problem Statement:

The value of π is computed using numerical integration:

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

The integral is approximated using the rectangle method and parallelized using OpenMP reduction.

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

int main() {
    long num_steps = 100000;
    double step;
    int i, threads;
    double x, pi, sum;
    double t1, t2, seq_time;

    step = 1.0 / (double)num_steps;

    // sequential
    sum = 0.0;
    t1 = omp_get_wtime();
    for (i = 0; i < num_steps; i++) {
        x = (i + 0.5) * step;
        sum = sum + 4.0 / (1.0 + x * x);
   }
    pi = step * sum;
    t2 = omp_get_wtime();
    seq_time = t2 - t1;

    printf("Sequential:\n");
    printf("Pi = %.10f\n", pi);
    printf("Time = %f\n\n", seq_time);

    // parallel
    printf("Parallel:\n");
    for (threads = 2; threads <= 10; threads++) {
        double par_time, speedup;

        omp_set_num_threads(threads);

        sum = 0.0;
        t1 = omp_get_wtime();
        #pragma omp parallel for private(x) reduction(+:sum)
        for (i = 0; i < num_steps; i++) {
            x = (i + 0.5) * step;
            sum = sum + 4.0 / (1.0 + x * x);
       }
        pi = step * sum;
        t2 = omp_get_wtime();

        par_time = t2 - t1;
        speedup = seq_time / par_time;

        printf("Threads=%d  Pi=%.10f  Time=%f  Speedup=%.2f\n", threads, pi, par_time, speedu
p);}

    return 0;
}

```



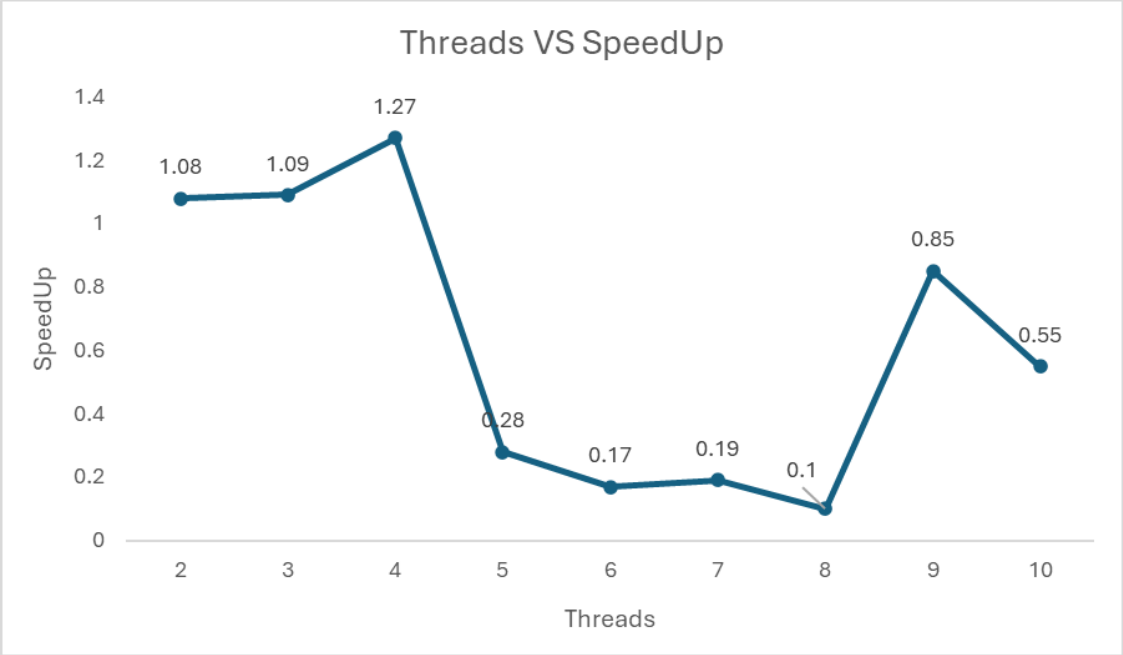
```
Sequential:
Pi = 3.1415926536
Time = 0.000232

Parallel:
Threads=2 Pi=3.1415926536 Time=0.000215 Speedup=1.08
Threads=3 Pi=3.1415926536 Time=0.000212 Speedup=1.09
Threads=4 Pi=3.1415926536 Time=0.000183 Speedup=1.27
Threads=5 Pi=3.1415926536 Time=0.000829 Speedup=0.28
Threads=6 Pi=3.1415926536 Time=0.001326 Speedup=0.17
Threads=7 Pi=3.1415926536 Time=0.001207 Speedup=0.19
Threads=8 Pi=3.1415926536 Time=0.002323 Speedup=0.10
Threads=9 Pi=3.1415926536 Time=0.000274 Speedup=0.85
Threads=10 Pi=3.1415926536 Time=0.000419 Speedup=0.55
```

Observation

The loop iterations are divided among threads and partial sums are combined using `reduction(+:sum)`, avoiding race conditions.

Maximum speedup is small (~ 1.27) because the sequential execution time is extremely small. For 2–4 threads, speedup increases gradually.

For 5–8 threads, speedup drops due to synchronization overhead, cache contention, and reduction costs. A slight recovery at higher thread counts is attributed to scheduling effects.



Conclusion

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

Submitted to : Dr Saif Nalband

Submitted by : Arpan Goyal

Roll No. : 102303479