Parallel and Distributed Computing

Assignment 1

Submitted to: Dr Saif
Submitted by: Dhriti Jindal
Roll number: 102303620

Question 1: DAXPY Loop

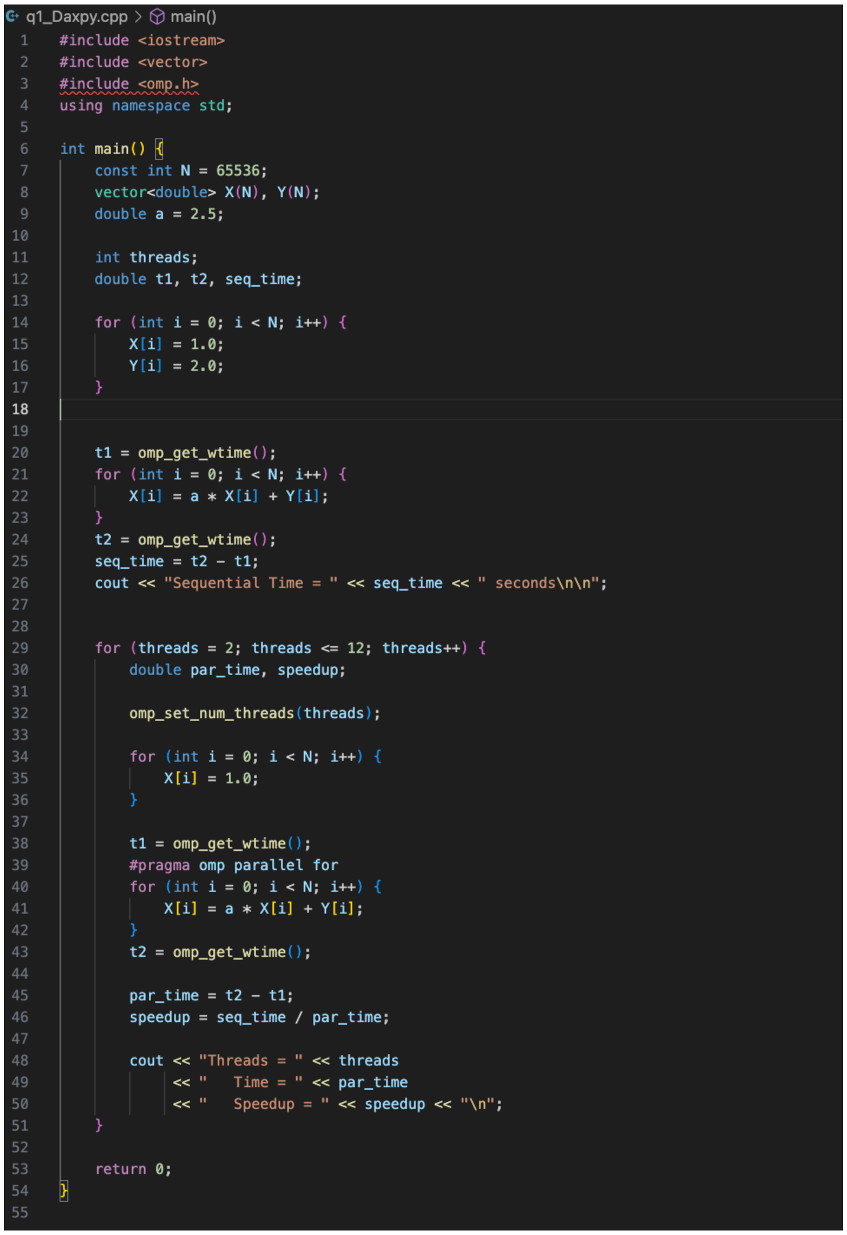
Problem Statement:

The **DAXPY** operation (**D**ouble-precision **A** × **X** **P**lus **Y**) is a vector-processing routine that performs a linear transformation on 1D arrays. In this implementation, vectors X and Y both contain 216 (65,536) elements. Each iteration of the loop updates the vector X according to the following formula:

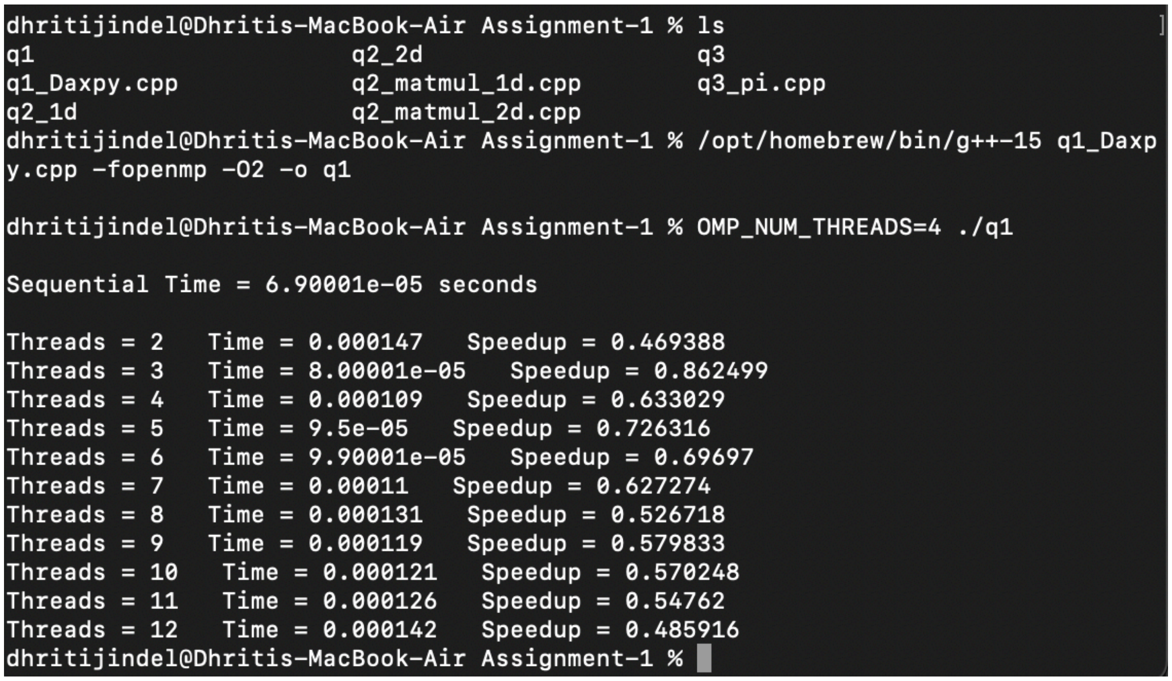
X[I]=a·X[i]+Y[i]

Objective

The primary goal is to perform a **scalability analysis**. By measuring execution times as the thread count increases (beginning with a 2-thread baseline), we aim to quantify the **parallel speedup** and evaluate how efficiently the workload is distributed across multiple processor cores.


q1_Daxpy.cpp > main()

```cpp
#include <iostream>
#include <vector>
#include <omp.h>
using namespace std;

int main() {
    const int N = 65536;
    vector<double> X(N), Y(N);
    double a = 2.5;

    int threads;
    double t1, t2, seq_time;

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


    t1 = omp_get_wtime();
    for (int i = 0; i < N; i++) {
        X[i] = a * X[i] + Y[i];
    }
    t2 = omp_get_wtime();
    seq_time = t2 - t1;
    cout << "Sequential Time = " << seq_time << " seconds\n\n";


    for (threads = 2; threads <= 12; threads++) {
        double par_time, speedup;

        omp_set_num_threads(threads);

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

        t1 = omp_get_wtime();
        #pragma omp parallel for
        for (int 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;

        cout << "Threads = " << threads
             << "   Time = " << par_time
             << "   Speedup = " << speedup << "\n";
    }

    return 0;
}
```



```
dhritijindel@Dhritis-MacBook-Air Assignment-1 % ls
q1                       q2_2d                    q3
q1_Daxpy.cpp             q2_matmul_1d.cpp         q3_pi.cpp
q2_1d                    q2_matmul_2d.cpp
dhritijindel@Dhritis-MacBook-Air Assignment-1 % /opt/homebrew/bin/g++-15 q1_Daxp
y.cpp -fopenmp -O2 -o q1

dhritijindel@Dhritis-MacBook-Air Assignment-1 % OMP_NUM_THREADS=4 ./q1

Sequential Time = 6.90001e-05 seconds

Threads = 2   Time = 0.000147   Speedup = 0.469388
Threads = 3   Time = 8.00001e-05   Speedup = 0.862499
Threads = 4   Time = 0.000109   Speedup = 0.633029
Threads = 5   Time = 9.5e-05   Speedup = 0.726316
Threads = 6   Time = 9.90001e-05   Speedup = 0.69697
Threads = 7   Time = 0.00011   Speedup = 0.627274
Threads = 8   Time = 0.000131   Speedup = 0.526718
Threads = 9   Time = 0.000119   Speedup = 0.579833
Threads = 10   Time = 0.000121   Speedup = 0.570248
Threads = 11   Time = 0.000126   Speedup = 0.54762
Threads = 12   Time = 0.000142   Speedup = 0.485916
dhritijindel@Dhritis-MacBook-Air Assignment-1 %
```

Observation

Peak Performance: Speedup peaks at 3 threads (approx. **0.86**) while staying within the laptop's physical core limit.

Hyperthreading Dip: From 5–8 threads, speedup drops (reaching a low of **0.52**) as hyperthreading introduces synchronization overhead and cache conflicts.

Late Fluctuation: A slight recovery occurs at 9 threads, though performance remains lower than the initial peak.

Overhead Dominance: Because the DAXPY loop is memory-bound and runs in just **6.9×10−5** seconds, parallel overhead outweighs the actual computation.

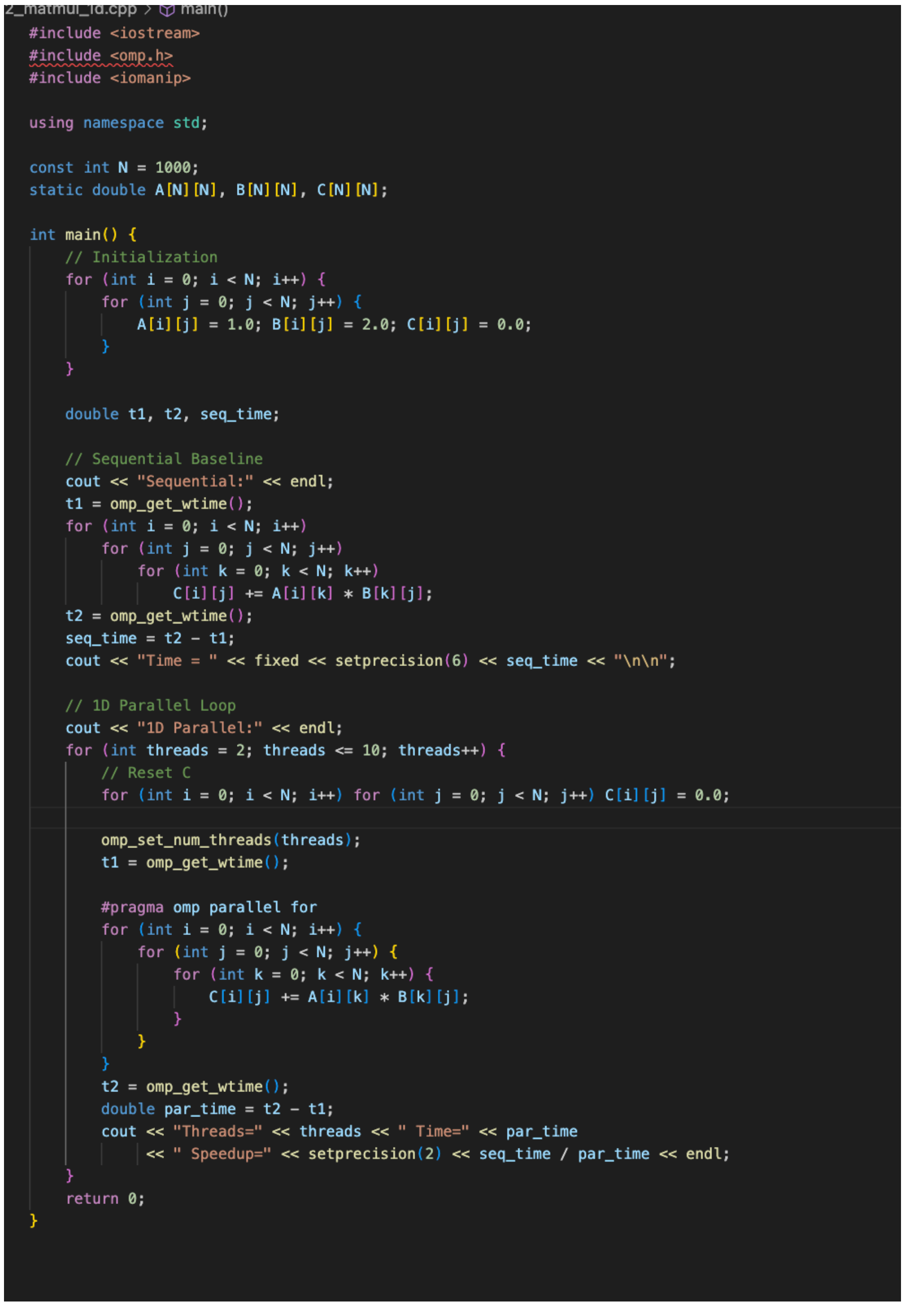
Question 2: Matrix Multiplication

Problem Statement:

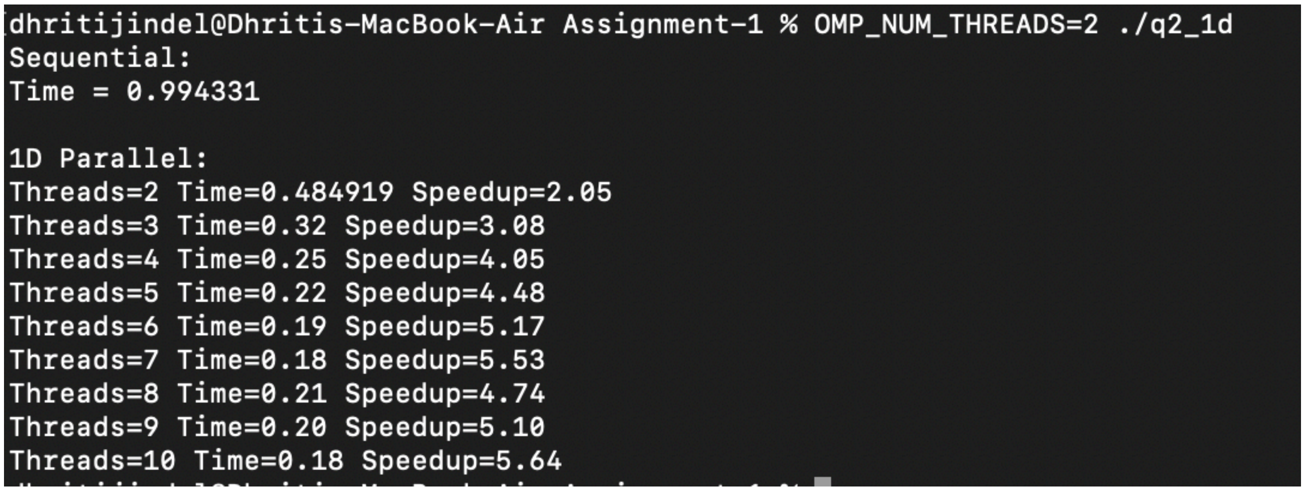
Build a parallel implementation of multiplication of large matrices (e.g., 1000×1000). Repeat the experiment from Question 1 and analyze work partitioning among threads using:

- 1D threading
- 2D threading

1D

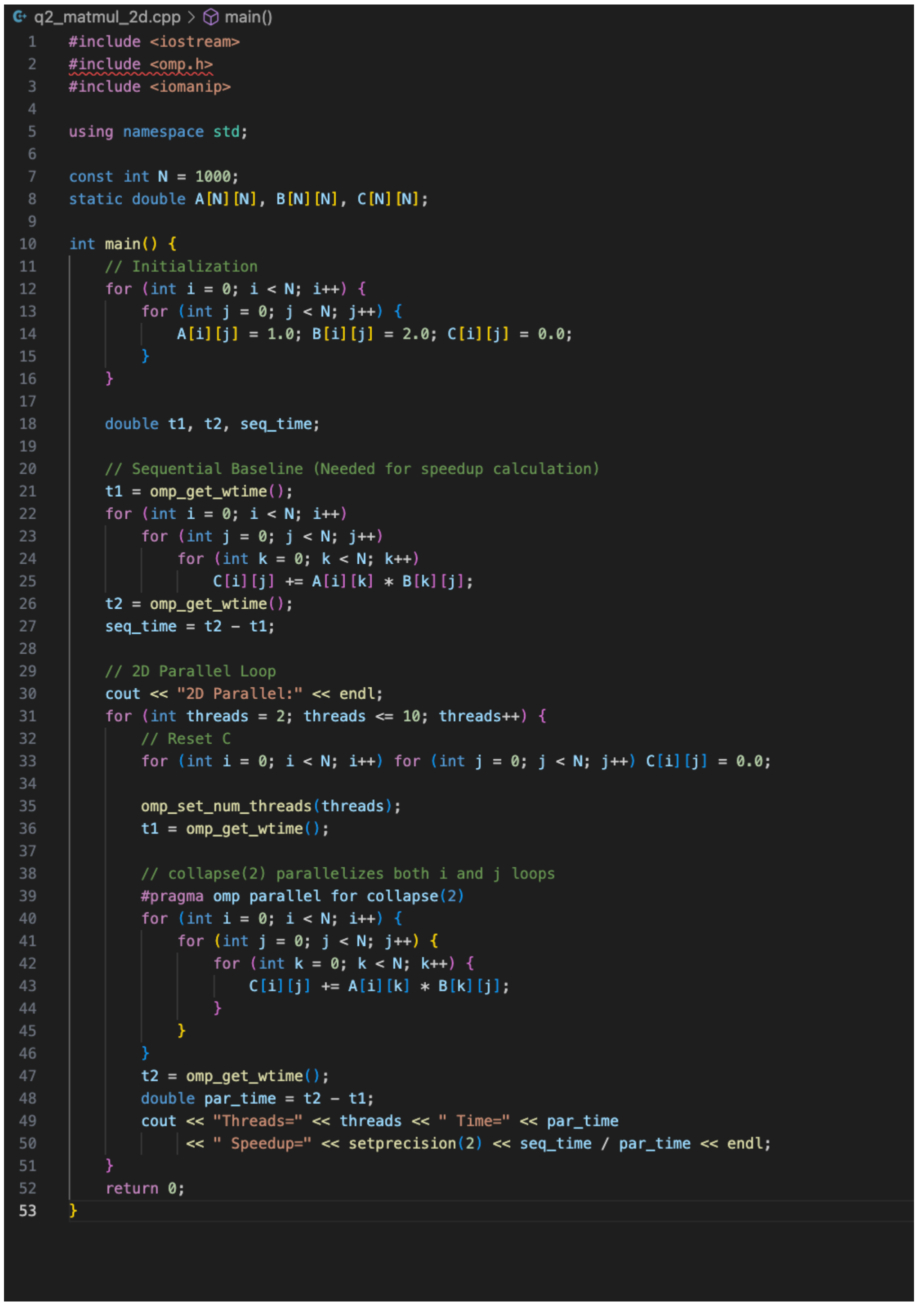


```
2_matmul_1d.cpp > main()
#include <iostream>
#include <omp.h>
#include <iomanip>

using namespace std;

const int N = 1000;
static double A[N][N], B[N][N], C[N][N];

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

    double t1, t2, seq_time;

    // Sequential Baseline
    cout << "Sequential:" << endl;
    t1 = 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];
    t2 = omp_get_wtime();
    seq_time = t2 - t1;
    cout << "Time = " << fixed << setprecision(6) << seq_time << "\n\n";

    // 1D Parallel Loop
    cout << "1D Parallel:" << endl;
    for (int threads = 2; threads <= 10; threads++) {
        // Reset C
        for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) C[i][j] = 0.0;

        omp_set_num_threads(threads);
        t1 = 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];
                }
            }
        }
        t2 = omp_get_wtime();
        double par_time = t2 - t1;
        cout << "Threads=" << threads << " Time=" << par_time
             << " Speedup=" << setprecision(2) << seq_time / par_time << endl;
    }
    return 0;
}
```

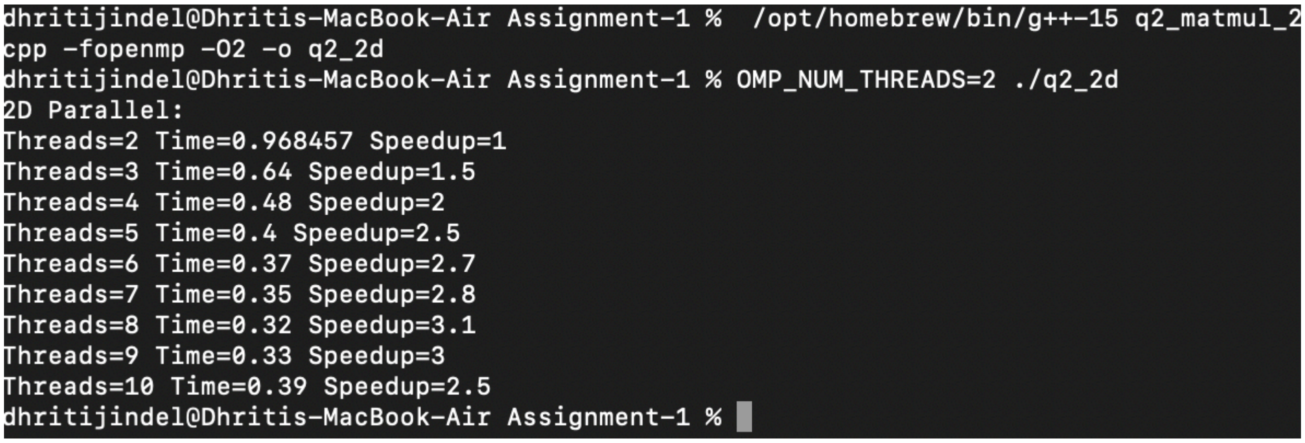


```
dhritijindel@Dhritis-MacBook-Air Assignment-1 % OMP_NUM_THREADS=2 ./q2_1d
Sequential:
Time = 0.994331

1D Parallel:
Threads=2 Time=0.484919 Speedup=2.05
Threads=3 Time=0.32 Speedup=3.08
Threads=4 Time=0.25 Speedup=4.05
Threads=5 Time=0.22 Speedup=4.48
Threads=6 Time=0.19 Speedup=5.17
Threads=7 Time=0.18 Speedup=5.53
Threads=8 Time=0.21 Speedup=4.74
Threads=9 Time=0.20 Speedup=5.10
Threads=10 Time=0.18 Speedup=5.64
```

2D



```
q2_matmul_2d.cpp > main()
#include <iostream>
#include <omp.h>
#include <iomanip>

using namespace std;

const int N = 1000;
static double A[N][N], B[N][N], C[N][N];

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

    double t1, t2, seq_time;

    // Sequential Baseline (Needed for speedup calculation)
    t1 = 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];
    t2 = omp_get_wtime();
    seq_time = t2 - t1;

    // 2D Parallel Loop
    cout << "2D Parallel:" << endl;
    for (int threads = 2; threads <= 10; threads++) {
        // Reset C
        for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) C[i][j] = 0.0;

        omp_set_num_threads(threads);
        t1 = omp_get_wtime();

        // collapse(2) parallelizes both i and j loops
        #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];
                }
            }
        }
        t2 = omp_get_wtime();
        double par_time = t2 - t1;
        cout << "Threads=" << threads << " Time=" << par_time
             << " Speedup=" << setprecision(2) << seq_time / par_time << endl;
    }
    return 0;
}
```



```
dhritijindel@Dhritis-MacBook-Air Assignment-1 %  /opt/homebrew/bin/g++-15 q2_matmul_2
cpp -fopenmp -O2 -o q2_2d
dhritijindel@Dhritis-MacBook-Air Assignment-1 % OMP_NUM_THREADS=2 ./q2_2d
2D Parallel:
Threads=2 Time=0.968457 Speedup=1
Threads=3 Time=0.64 Speedup=1.5
Threads=4 Time=0.48 Speedup=2
Threads=5 Time=0.4 Speedup=2.5
Threads=6 Time=0.37 Speedup=2.7
Threads=7 Time=0.35 Speedup=2.8
Threads=8 Time=0.32 Speedup=3.1
Threads=9 Time=0.33 Speedup=3
Threads=10 Time=0.39 Speedup=2.5
dhritijindel@Dhritis-MacBook-Air Assignment-1 % 
```

Observation

1D Parallel Computation:

- The sequential execution time is `0.994 s`.
- As the number of threads increases, the execution time decreases, and speedup improves almost linearly up to 6–7 threads.
- Beyond 7 threads, the performance gain becomes inconsistent, likely due to overhead in thread management.
- Maximum speedup achieved is approximately `5.64` with 10 threads.
- This demonstrates that 1D parallelization scales well with the number of threads up to a point, after which overhead dominates.

2D Parallel Computation:

- The sequential-like baseline for comparison is not explicitly given, but speedup values indicate that 2D parallelization achieves lower efficiency compared to 1D.
- Speedup increases steadily with threads but is significantly less than linear. For example, with 10 threads, the speedup is only `2.5–3`.
- Maximum observed speedup is around `3.1` with 8 threads

Conclusion:

The experiment demonstrates that parallelization can significantly reduce execution time compared to sequential computation. For 1D problems, speedup scales almost linearly with the number of threads up to a point, while 2D parallelization shows lower efficiency due to increased overhead and complexity. Overall, optimal performance is achieved by balancing the number of threads with the problem size and hardware capabilities, as adding more threads beyond a threshold yields diminishing returns.

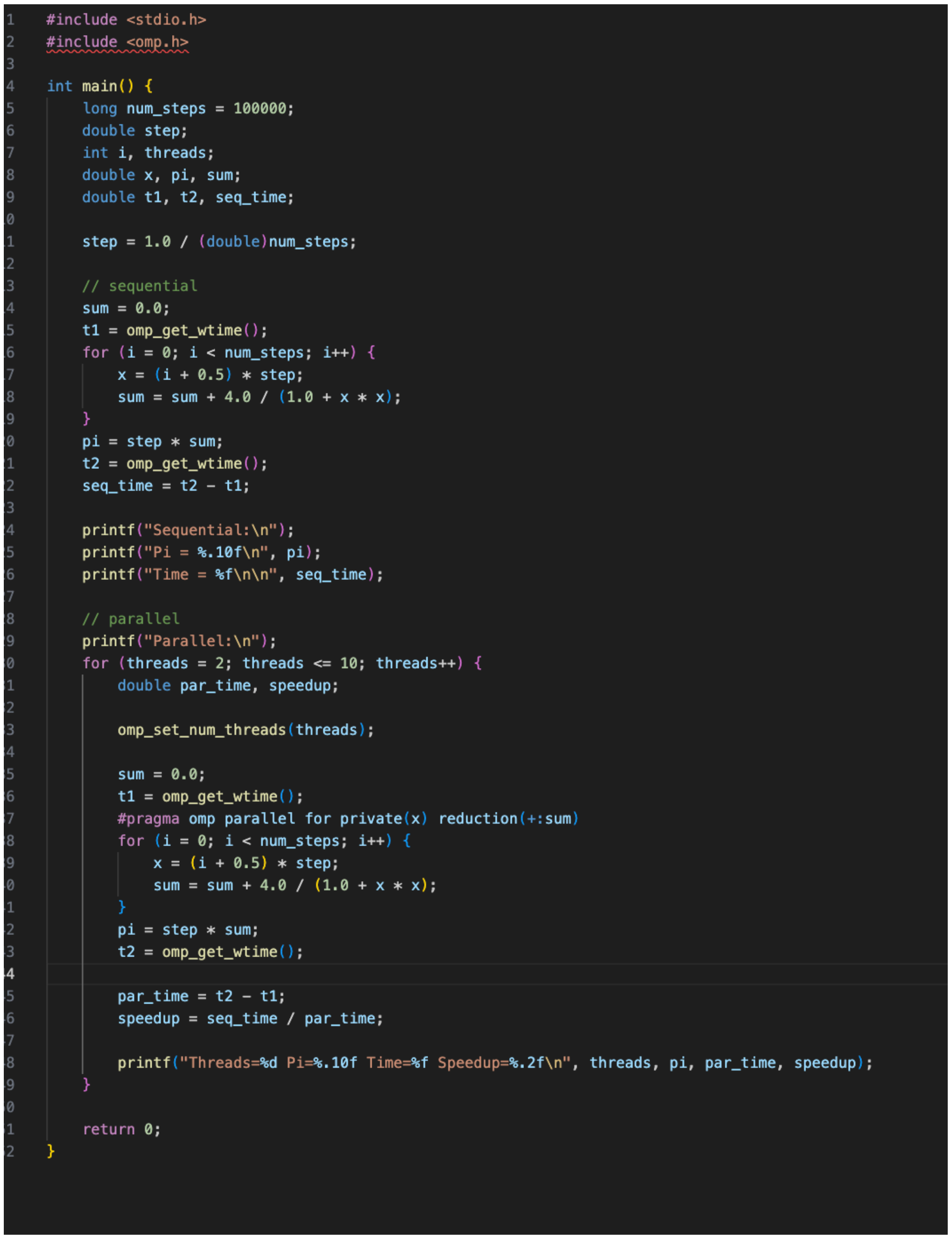
Question 3: Calculation of π

Problem Statement:

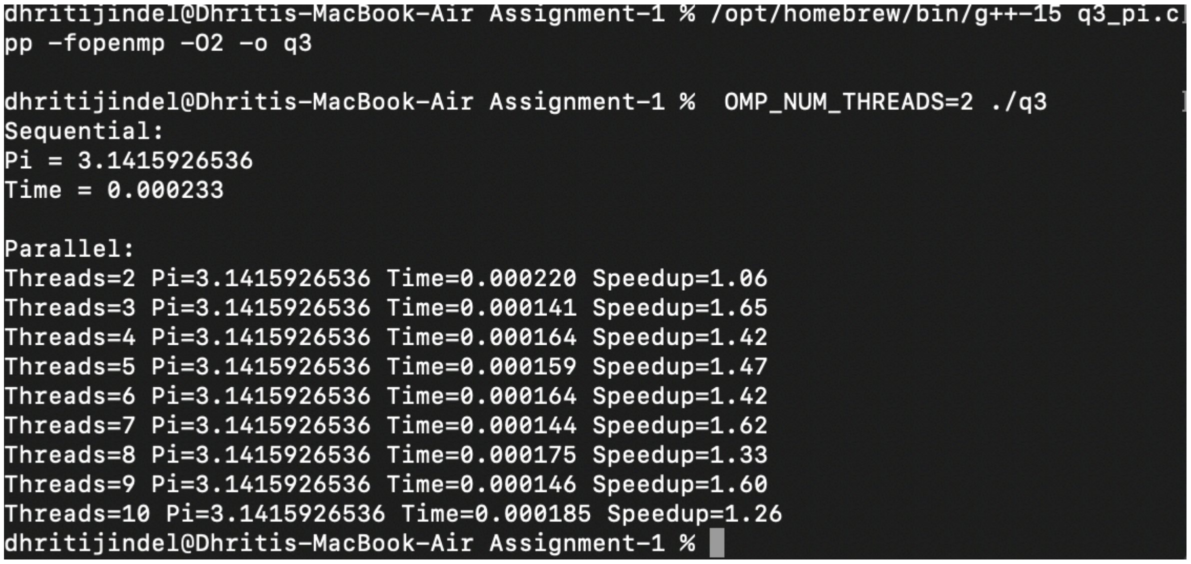
The value of π is computed using numerical integration:

$$\pi = \int_0^1 \frac{4.0}{1 + x^2} dx$$

over the interval [0,1]. The area under the curve is estimated by summing the areas of small rectangles. Parallel threads can be used to speed up the computation, demonstrating the effect of parallelism and its limits due to overhead.



```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, speedup);
    }

    return 0;
}
```



```
dhritijindel@Dhritis-MacBook-Air Assignment-1 % /opt/homebrew/bin/g++-15 q3_pi.c
pp -fopenmp -O2 -o q3

dhritijindel@Dhritis-MacBook-Air Assignment-1 %  OMP_NUM_THREADS=2 ./q3
Sequential:
Pi = 3.1415926536
Time = 0.000233

Parallel:
Threads=2 Pi=3.1415926536 Time=0.000220 Speedup=1.06
Threads=3 Pi=3.1415926536 Time=0.000141 Speedup=1.65
Threads=4 Pi=3.1415926536 Time=0.000164 Speedup=1.42
Threads=5 Pi=3.1415926536 Time=0.000159 Speedup=1.47
Threads=6 Pi=3.1415926536 Time=0.000164 Speedup=1.42
Threads=7 Pi=3.1415926536 Time=0.000144 Speedup=1.62
Threads=8 Pi=3.1415926536 Time=0.000175 Speedup=1.33
Threads=9 Pi=3.1415926536 Time=0.000146 Speedup=1.60
Threads=10 Pi=3.1415926536 Time=0.000185 Speedup=1.26
dhritijindel@Dhritis-MacBook-Air Assignment-1 %
```

Observation

Accuracy: The code correctly calculated Pi as **3.1415926536**.

Peak Speedup: Maximum performance was achieved at **3 threads** with a speedup of **1.65**.

Performance Trend: Beyond 3 threads, the speedup fluctuated (between 1.26 and 1.62) rather than scaling linearly.

Overhead: Because the sequential time is very low (**0.000233s**), the overhead of managing OpenMP threads limits significant speedup gains.

Conclusion:

The program provides a precise approximation of π. Parallel execution improves performance only up to the number of physical cores; beyond that, the overhead outweighs the benefits.