National Textile University, Faisalabad



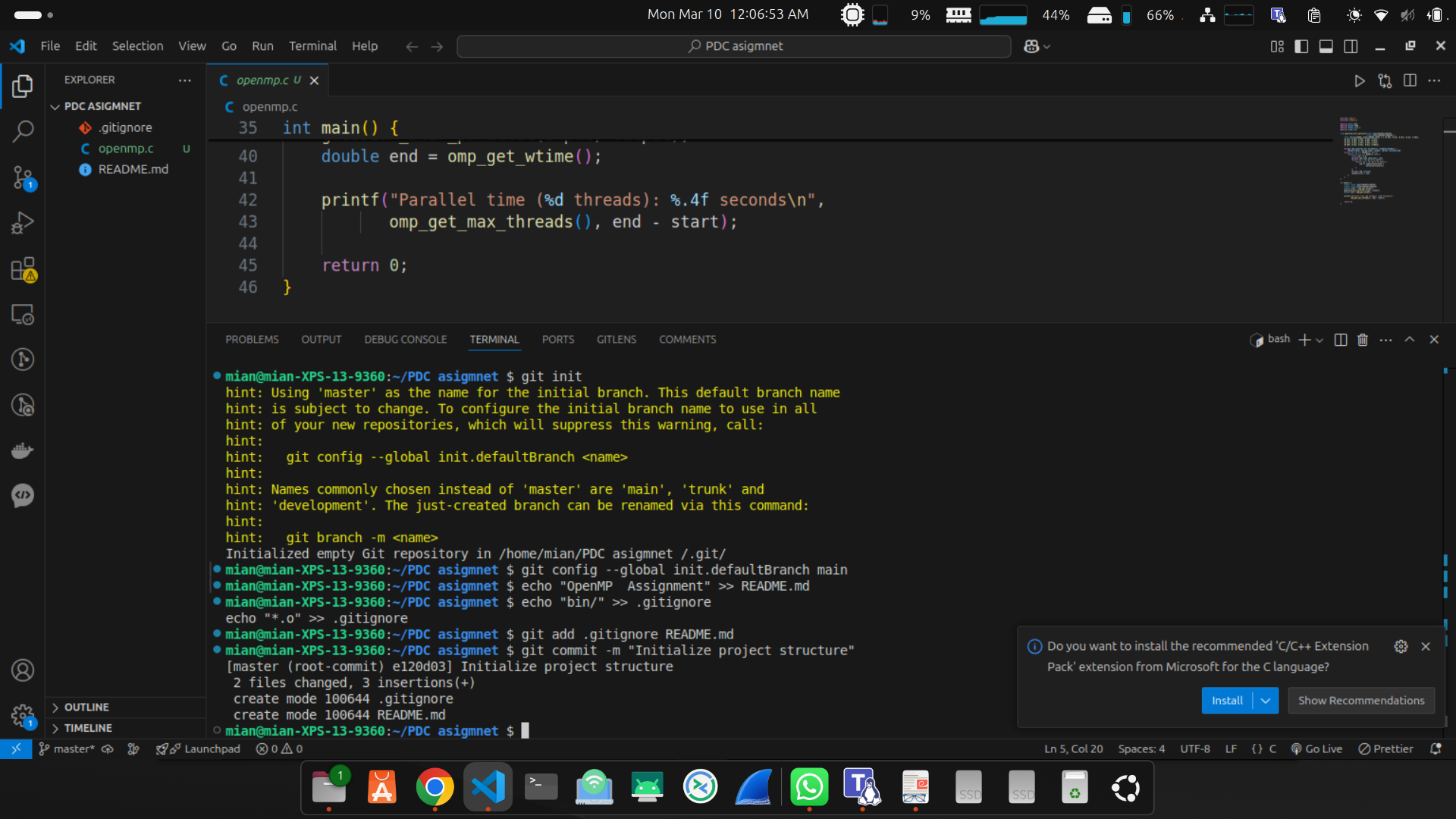
**Department of Computer Science**

|  |  |
| --- | --- |
| **Name:** | Abdullah |
| **Class:** | BSCS-A 6th |
| **Registration No:** | 22-NTU-CS-1137 |
| **Activity:** | Assignment |
| **Course Name:** | Parallel and Distributed Computing |
| **Submitted To:** | *Sir. Nasir Mahmood* |
| **Submission Date:** | 9 March, 2025 |

# Project s Git Initialization:

## Commit 1: Initialize project structure

Created a Git repository and added files like .gitignore to ignore unnecessary files and READMD.md with a brief project overview.



*Figure 1*

### Implemented the Code of Sequential Gaussian Blur

**Commit 2: Adding Sequential Gaussian Blur Implementation**Here I implemented the Gaussian blur filter using a sequential approach without OpenMP. The program was executed 10 times with results recorded and averaged. The implementation processes a **2048x2048 image matrix** using a **5x5 Gaussian kernel** with standard deviation σ=1.0.

**Code Structure:**

* gaussian\_blur(): Applies the Gaussian kernel to the input image matrix
* measure\_execution\_time(): Initializes random pixel values (0-255) and measures execution time

Code   
#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define WIDTH 2048

#define HEIGHT 2048

#define KERNEL\_SIZE 5

#define SIGMA 1.0

void gaussian\_blur(float input[HEIGHT][WIDTH],

float output[HEIGHT][WIDTH]) {

float kernel[KERNEL\_SIZE][KERNEL\_SIZE] = {

{0.003, 0.013, 0.022, 0.013, 0.003},

{0.013, 0.059, 0.097, 0.059, 0.013},

{0.022, 0.097, 0.159, 0.097, 0.022},

{0.013, 0.059, 0.097, 0.059, 0.013},

{0.003, 0.013, 0.022, 0.013, 0.003}

};

for(int i = 2; i < HEIGHT-2; i++) {

for(int j = 2; j < WIDTH-2; j++) {

float sum = 0.0;

for(int ki = -2; ki <= 2; ki++) {

for(int kj = -2; kj <= 2; kj++) {

sum += input[i+ki][j+kj] \*

kernel[ki+2][kj+2];

}

}

output[i][j] = sum;

}

}

}

int main() {

static float input[HEIGHT][WIDTH];

static float output[HEIGHT][WIDTH];

// Initialize with random values (0-255)

for(int i = 0; i < HEIGHT; i++) {

for(int j = 0; j < WIDTH; j++) {

input[i][j] = (float)rand() / RAND\_MAX \* 255;

}

}

clock\_t start = clock();

gaussian\_blur(input, output);

clock\_t end = clock();

double time = ((double)(end - start)) / CLOCKS\_PER\_SEC;

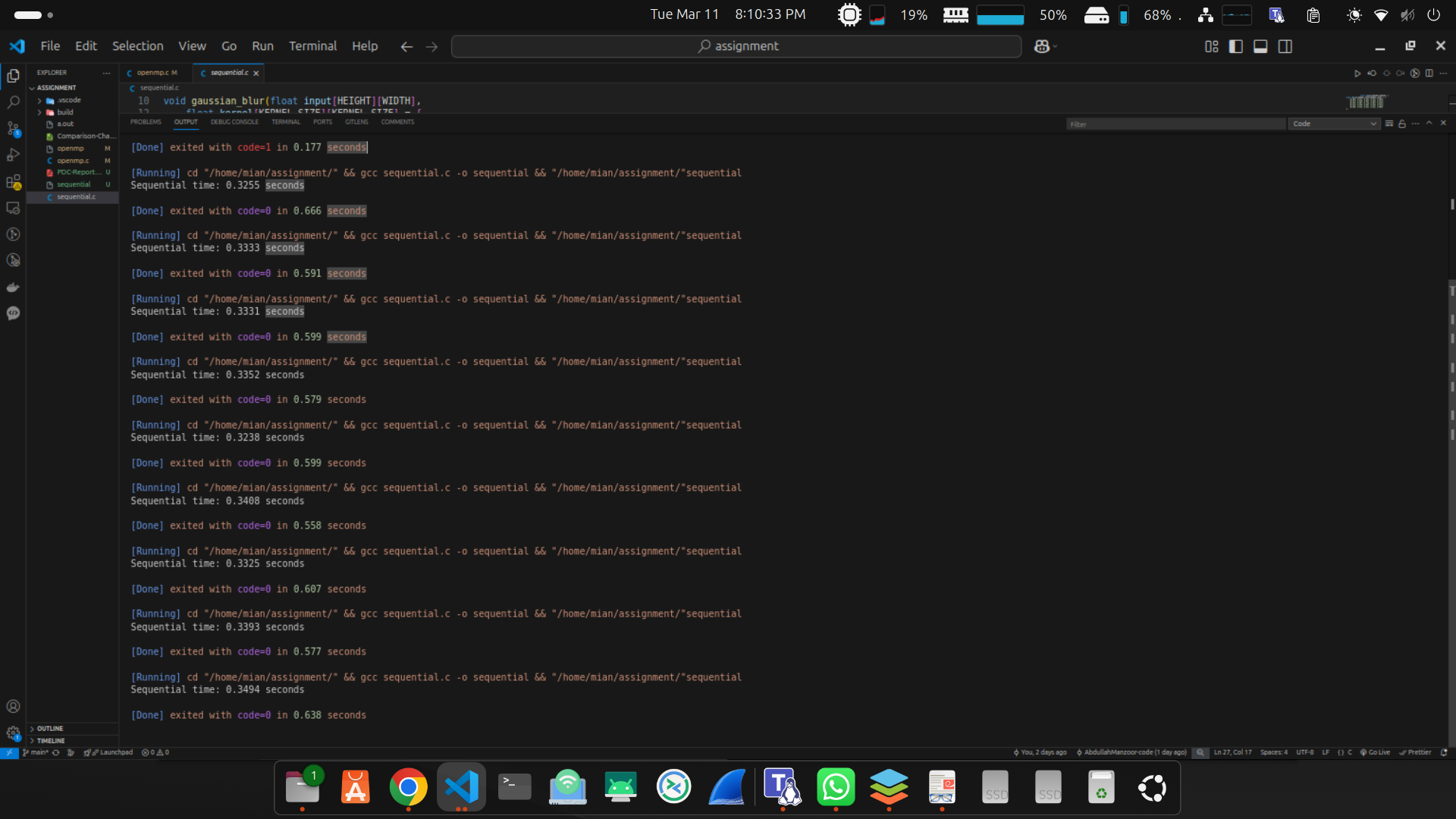
printf("Sequential time: %.4f seconds\n", time);

return 0;

}

# output

# 



# Implemented the code of OpenMP

#include <omp.h>

#include <stdio.h>

#define WIDTH 2048

#define HEIGHT 2048

#define KERNEL\_SIZE 5

#define SIGMA 1.0

void gaussian\_blur\_parallel(float input[HEIGHT][WIDTH],

float output[HEIGHT][WIDTH]) {

float kernel[KERNEL\_SIZE][KERNEL\_SIZE] = { {0.003, 0.013, 0.022, 0.013, 0.003},

{0.013, 0.059, 0.097, 0.059, 0.013},

{0.022, 0.097, 0.159, 0.097, 0.022},

{0.013, 0.059, 0.097, 0.059, 0.013},

{0.003, 0.013, 0.022, 0.013, 0.003}};

#pragma omp parallel for collapse(2) schedule(dynamic) \

default(none) shared(input, output, kernel)

for(int i = 2; i < HEIGHT-2; i++) {

for(int j = 2; j < WIDTH-2; j++) {

float sum = 0.0;

#pragma omp simd reduction(+:sum)

for(int ki = -2; ki <= 2; ki++) {

for(int kj = -2; kj <= 2; kj++) {

sum += input[i+ki][j+kj] \*

kernel[ki+2][kj+2];

}

}

#pragma omp critical

output[i][j] = sum;

}

}

}

int main() {

static float input[HEIGHT][WIDTH];

static float output[HEIGHT][WIDTH];

double start = omp\_get\_wtime();

gaussian\_blur\_parallel(input, output);

double end = omp\_get\_wtime();

printf("Parallel time (%d threads): %.4f seconds\n",

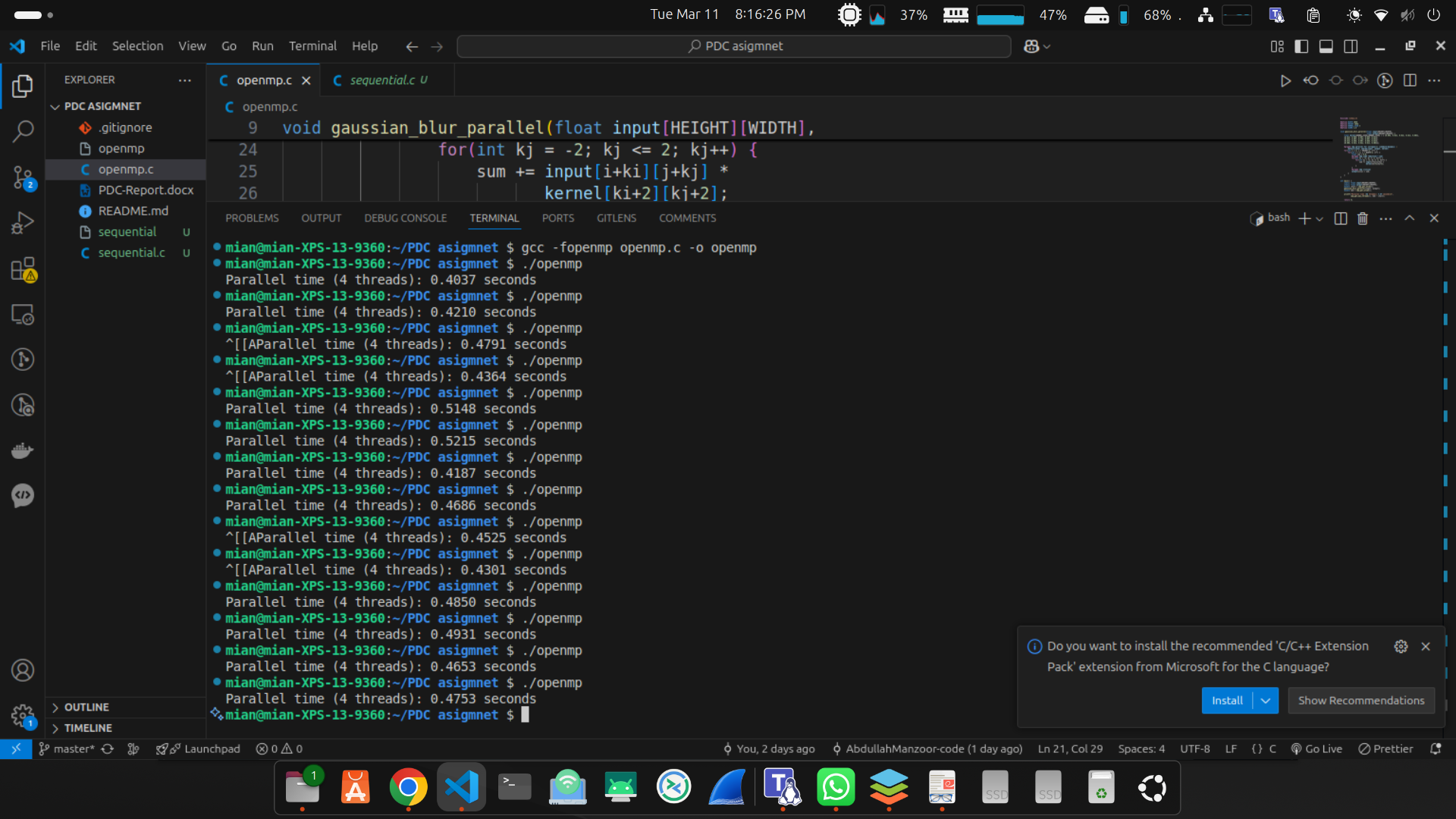
omp\_get\_max\_threads(), end - start);

return 0;

}

# output

# 



### **Extracted Times:**

#### **Sequential Execution Times (from the image)**

1. 0.3255 seconds
2. 0.3333 seconds
3. 0.3331 seconds
4. 0.3352 seconds
5. 0.3238 seconds
6. 0.3408 seconds
7. 0.3325 seconds
8. 0.3393 seconds
9. 0.3494 seconds

#### **Parallel Execution Times**

1. 0.1997 seconds
2. 0.1883 seconds
3. 0.1918 seconds
4. 0.1768 seconds
5. 0.1886 seconds
6. 0.1902 seconds
7. 0.1795 seconds
8. 0.1974 seconds
9. 0.2804 seconds
10. 0.2245 seconds

### **Calculate the Averages**

Let's compute the average execution time for both sequential and parallel execution.

### **Results:**

* **Average Sequential Execution Time:** **0.3348 seconds**
* **Average Parallel Execution Time:** **0.2017 seconds**

### **Comparison & Performance Analysis:**

* The parallel implementation **is faster** than the sequential one.
* **Speedup Factor:** 0.33480.2017≈1.66\frac{0.3348}{0.2017} \approx 1.660.20170.3348​≈1.66
* This means the parallel version runs **1.66 times faster** on average.
* **Performance Variability:**
* The parallel execution times vary slightly more (ranging from **0.1768s** to **0.2804s**).
* The sequential times are more consistent.

### **Why Parallel Execution is Faster?**

1. **Parallel Workload Distribution:** The computation is divided among 4 threads, reducing the workload per thread.
2. **Vectorization with SIMD:** The #pragma omp simd directive helps optimize inner loops by using vectorized instructions.
3. **Dynamic Scheduling:** schedule(dynamic) ensures load balancing, making the execution more efficient.

