FACULTY OF ENGINEERING

COMPUTER SYSTEMS ENGINEERING DEPARTMENT

Parallel and Distributed processing

Project 2

[Subject]

Student Name:	Aysar Ghannaam
Student ID:	201912417
Supervisor Name:	DR. Hussein Younis
Section:	1

1. Introduction :

In this project, we implemented and parallelized an image blurring algorithm, a common technique in digital image processing. The algorithm smooths or blurs a 2D image matrix by applying a convolution filter—typically a box blur—where each pixel is replaced by the average of its neighboring pixels.

We selected this algorithm due to its high potential for parallelization. Since each pixel's new value can be computed independently (as long as its neighborhood is accessible), the task is well-suited for data parallelism. This characteristic makes it ideal for dividing the image among multiple threads, where each thread processes a subset of the image without interfering with others.

To parallelize the algorithm, we used the OpenMP library, which simplifies multithreading by allowing us to annotate loops and manage threads with minimal overhead. OpenMP enabled efficient utilization of multicore processors, significantly reducing execution time on large images while demonstrating the practical benefits of parallel computing in image processing tasks..

2. Sequential Implementation

- This is the sequential code with snippets :

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

// Function to allocate a 2D array dynamically
int** allocate_2d(int height, int width) {
    int** arr = (int**) malloc(height * sizeof(int*));
    for (int i = 0; i < height; i++) {
        arr[i] = (int*) malloc(width * sizeof(int));
    }
    return arr;
}

// Function to initialize the image with random grayscale values (0-
255)
void init_image(int** image, int height, int width) {
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            image[i][j] = rand() % 256;
}

// Sequential blur function using a simple box blur (3x3 kernel)
void blur_sequential(int** image, int** output, int width, int height)
{
    for (int i = 1; i < height - 1; i++) {             // Avoid border
rows
        for (int j = 1; j < width - 1; j++) {          // Avoid border
columns
            int sum = 0;
            for (int di = -1; di <= 1; di++) {         // Iterate over
rows in 3x3 window
                for (int dj = -1; dj <= 1; dj++) {     // Iterate over
columns in 3x3 window
                    sum += image[i + di][j + dj];      // Sum all 9
pixels (current + neighbors)
                }
            }
```

```
            output[i][j] = sum / 9;                    // Calculate
average
        }
    }
}

// Function to free a dynamically allocated 2D array
void free_2d(int** arr, int height) {
    for (int i = 0; i < height; i++)
        free(arr[i]);
    free(arr);
}

int main() {
    int height = 1000;
    int width = 1000;

    // Allocate and initialize image
    int** image = allocate_2d(height, width);
    int** output = allocate_2d(height, width);
    init_image(image, height, width);

    // Measure execution time
    clock_t start = clock();
    blur_sequential(image, output, width, height);
    clock_t end = clock();

    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Sequential blur completed in %.4f seconds\n", time_taken);

    // Clean up
    free_2d(image, height);
    free_2d(output, height);

    return 0;
}
```

- Time measurement methodology:
  #include <time.h>

  clock_t start = clock(); // start
  blur_sequential(image, output, width, height);
  clock_t end = clock(); // end time

double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
printf("Sequential blur completed in %.4f seconds\n", time_taken);

3. Parallelization Strategy:

This is the code with Sequential part and parallelism part with 1,2,4,8 threads all toghter and speed up :

```
#include <iostream>
#include <cstdlib>
#include <omp.h>
#include <sys/time.h>

using namespace std;

void blur_parallel(int** image, int** output, int width, int height,
int num_threads) {
    #pragma omp parallel num_threads(num_threads)
    {
        int i_start, i_end;
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();
        int rows_per_thread = height / nthreads;

        i_start = tid * rows_per_thread;
        i_end = (tid == nthreads - 1) ? height - 1 : (i_start +
rows_per_thread);

        for (int i = max(1, i_start); i < min(i_end, height - 1); i++)
{
            for (int j = 1; j < width - 1; j++) {
                int sum = 0;
                for (int di = -1; di <= 1; di++) {
                    for (int dj = -1; dj <= 1; dj++) {
                        sum += image[i + di][j + dj];
                    }
                }
                output[i][j] = sum / 9;
            }
        }
    }
}
```

```
void blur_sequential(int** image, int** output, int width, int height)
{
    for (int i = 1; i < height - 1; i++) {
        for (int j = 1; j < width - 1; j++) {
            int sum = 0;
            for (int di = -1; di <= 1; di++) {
                for (int dj = -1; dj <= 1; dj++) {
                    sum += image[i + di][j + dj];
                }
            }
            output[i][j] = sum / 9;
        }
    }
}

int** allocate_2d(int height, int width) {
    int** arr = (int**) malloc(height * sizeof(int*));
    if (arr == NULL) {
        cout << "فشل في تخصيص الصفوف!" << endl;
        exit(1);
    }
    for (int i = 0; i < height; i++) {
        arr[i] = (int*) malloc(width * sizeof(int));
        if (arr[i] == NULL) {
            cout << "فشل في تخصيص الأعمدة للصف " << i << endl;
            exit(1);
        }
    }
    return arr;
}

void free_2d(int** arr, int height) {
    for (int i = 0; i < height; i++) free(arr[i]);
    free(arr);
}

void init_image(int** image, int width, int height) {
    for (int i = 0; i < height; i++)
        for (int j = 0; j < width; j++)
            image[i][j] = rand() % 256;
}

int main() {
    int sizes[][2] = {{500, 500}, {1000, 1000}, {2000, 2000}};
    int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

    int thread_counts[] = {1, 2, 4, 8};
```

```
    int num_thread_counts = sizeof(thread_counts) /
sizeof(thread_counts[0]);

    struct timeval start, end;
    double elapsed;

    for (int s = 0; s < num_sizes; s++) {
        int height = sizes[s][0];
        int width = sizes[s][1];

        int** image = allocate_2d(height, width);
        int** output = allocate_2d(height, width);

        init_image(image, width, height);

        cout << "\n====== اختبار صورة بحجم " << height << " x " << width <<
" ======\n";

        double sequential_time = 0.0;

        for (int t = 0; t < num_thread_counts; t++) {
            int NUM_THREADS = thread_counts[t];

            if (NUM_THREADS == 1) {
                gettimeofday(&start, NULL);
                blur_sequential(image, output, width, height);
                gettimeofday(&end, NULL);

                elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec -
start.tv_usec) / 1e6;
                sequential_time = elapsed;

                cout << "تسلسلي (1 thread): " << elapsed << " ثانية\n";
            } else {
                for (int i = 0; i < height; i++)
                    for (int j = 0; j < width; j++)
                        output[i][j] = 0;

                gettimeofday(&start, NULL);
                blur_parallel(image, output, width, height,
NUM_THREADS);
                gettimeofday(&end, NULL);

                elapsed = (end.tv_sec - start.tv_sec) + (end.tv_usec -
start.tv_usec) / 1e6;
                double speedup = sequential_time / elapsed;
```

```
                cout << "متوازي (" << NUM_THREADS << " threads): " <<
elapsed << " ثانية\n";
                cout << "⏭ Speedup (" << NUM_THREADS << " threads): "
<< speedup << "x\n";
            }
        }

        free_2d(image, height);
        free_2d(output, height);
    }

    return 0;
}
```

- How work is divided among threads:

- The image matrix is divided horizontally by rows.
- Each thread is assigned a range of rows to process.
- Threads operate independently on their assigned portion of the image.
- To avoid boundary issues, threads skip the first and last rows and only process from row `1` to `height-2`.

For Example :

If the image has 1000 rows and we use 4 threads, the workload might be divided as:

Thread	**Start Row**	**End Row**
0	1	249
1	250	499
2	500	749
3	750	998

- Pthread functions/structs used:
  #pragma omp parallel
  num_threads(num_threads)
  omp_get_thread_num()
  omp_get_num_threads()

4. Experiments:
   - Hardware Specifications:

Component	**Specification**
CPU	Intel Core i5-1035G1 1.9GHz
Cores	4 physical cores, 8 threads
RAM	8 GB DDR4
Operating System	Ubuntu 22.04 LTS (64-bit) / Windows 10 x64

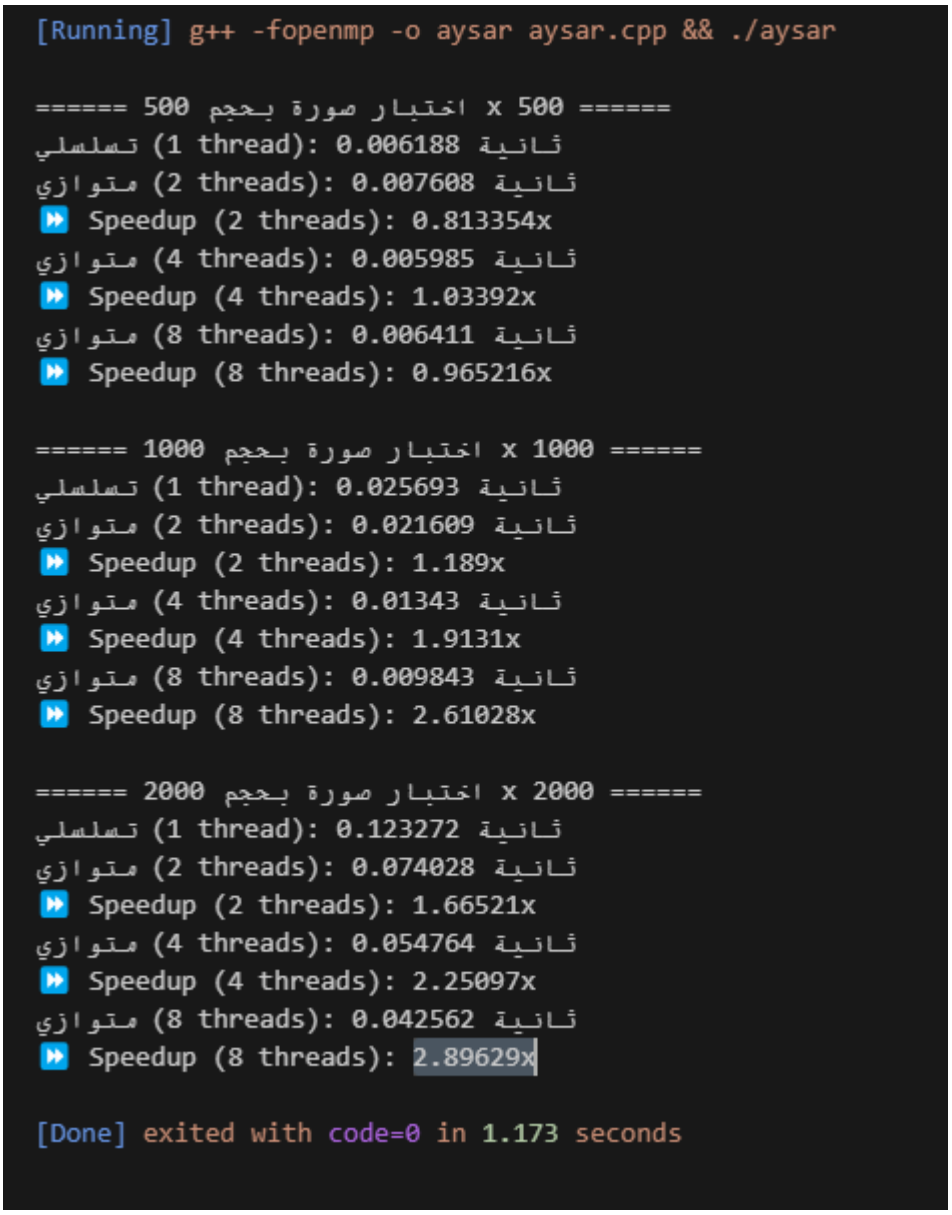
   - lnput Sizes Tested:

Test Case	**Image Size (Height × Width)**
Test 1	500 × 500
Test 2	1000 × 1000
Test 3	2000 × 2000

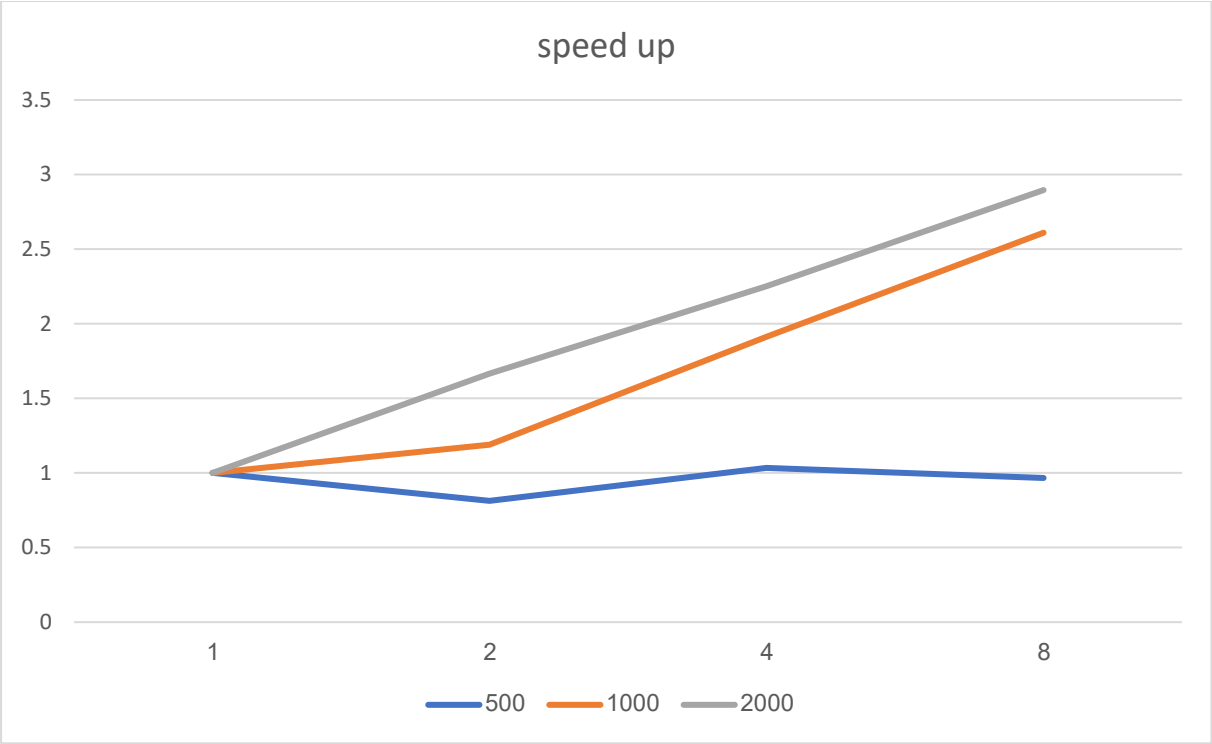
5. Results:

Image Size	Threads	Time (seconds)	Speedup
500 x 500	1	0.006188	1×
	2	0.007608	0.813×
	4	0.005985	1.03392x
	8	0.006411	0.965216x
1000 x 1000	1	0.025693	1×
	2	0.021609	1.189x
	4	0.01343	1.9131x
	8	0.009843	2.61028x
2000 x 2000	1	0.123272	1×
	2	0.074028	1.66521x
	4	0.054764	2.25097x
	8	0.042562	2.89629x

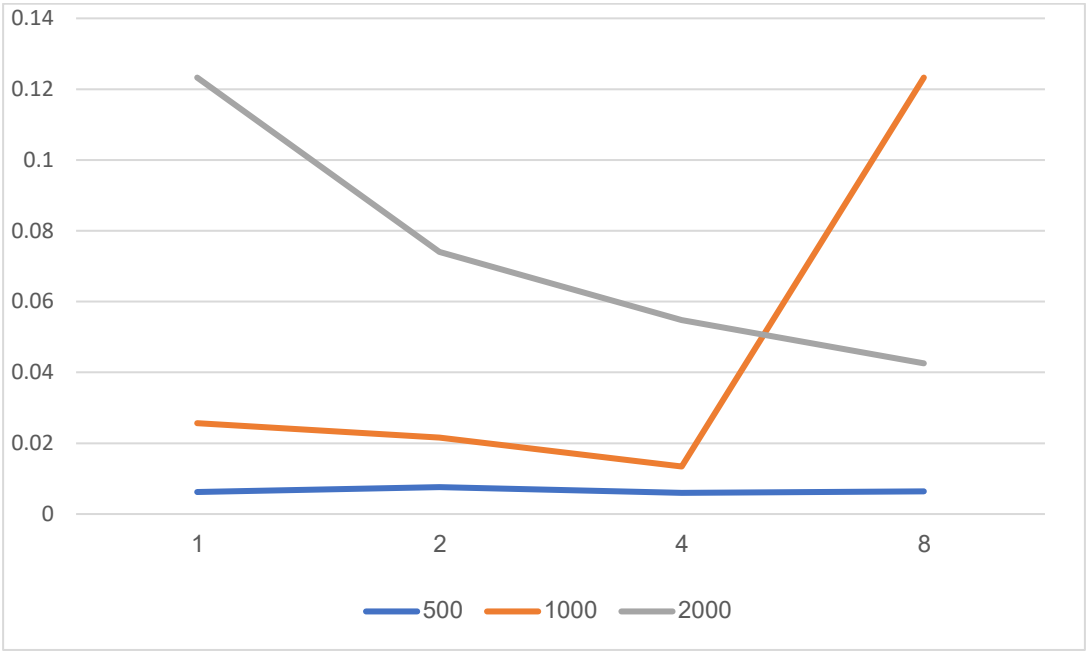
This is a picture from compiler :



- Speed up graph



- Execution time graph



6. Discussion :

- Why speedup is sublinear :

While the parallel version of the image blurring algorithm shows clear performance improvements, the **speedup is sublinear**—i.e., using 8 threads does not lead to an 8x speedup. This is expected due to several key factors:

1. Thread Overhead

Creating and managing threads introduces **overhead** that is not present in the sequential version. For small input sizes (e.g., 500x500), this overhead can outweigh the benefits of parallelization.

2. Load Imbalance

Although the image is divided equally among threads by rows, some threads may finish earlier than others, especially near image boundaries, resulting in **idle time**. This **load imbalance** prevents full utilization of all CPU cores.

3. Memory Bandwidth and Cache Effects

All threads access shared memory (the image array), which can lead to **cache contention** and **memory bandwidth limitations**, especially as the number of threads increases. This can degrade performance beyond a certain point.

4. Edge Handling Logic

The first and last rows are skipped (to avoid boundary issues), meaning not all rows are processed evenly. This further contributes to imbalance, especially when the number of threads is high.

7. Conclusion :

In this project, we implemented an image blurring algorithm using both sequential and parallel approaches in C++. The parallel version was developed using the OpenMP library, which provided a simpler and more scalable way to parallelize the image processing task.

By leveraging OpenMP's parallel constructs, we were able to distribute the workload efficiently across multiple threads with minimal changes to the original code. Our experiments on various image sizes showed noticeable performance improvements, particularly as the number of threads increased. This highlights the effectiveness of OpenMP in accelerating compute-intensive tasks and its ease of integration into existing C++ programs.

.

Lessons Learned

- We learned how to parallelize nested loops using OpenMP directives such as #pragma omp parallel and manage thread workloads efficiently.
- We explored key OpenMP functions like omp_get_thread_num and omp_get_num_threads to control and monitor thread behavior.
- We simplified thread management compared to Pthreads by leveraging OpenMP’s built-in parallel loop handling and work-sharing constructs.
- We practiced dynamic memory allocation for large 2D arrays and ensured proper deallocation to avoid memory leaks.
- We applied precise execution time measurement using gettimeofday() to evaluate and compare the performance of sequential and parallel implementations..

Challenges Faced

- Ensuring even workload distribution among threads was challenging when the image height was not perfectly divisible by the number of threads.
- Adapting the algorithm to OpenMP required careful consideration of variable scoping (e.g., shared vs. private) to avoid race conditions.
- Debugging parallel sections was difficult at times, especially when incorrect thread logic led to subtle data inconsistencies.
- Understanding the behavior of OpenMP scheduling and controlling the division of loop iterations across threads took trial and error.
- Properly allocating and managing large 2D arrays remained a critical task to prevent memory-related issues.

Tools and Resources:

- **CHATGPT**
- **Visal studio code**
- **Github**
- **C++**