الجامعة العربية الأمريكية
ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

COMPUTER SYSTEMS ENGINEERING DEPARTMENT

Parallel and Distributed processing

Project 1

[Subject]

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

1. Introduction :

An image blurring algorithm, which is frequently used in digital image processing, is what we decided to implement and parallelize for this project. The algorithm smoothes or blurs a 2D image matrix by applying a convolution filter (often a box blur or Gaussian blur). For each pixel in the image, this entails calculating the average of its neighbors and substituting the calculated value.

This algorithm was chosen because of its great parallelism potential. It is a perfect candidate for data parallelism since the new value of each pixel can be computed independently of the others (provided that the required neighborhood is available). This enables us to split up the image rows into several threads, each of which applies the blur function to its allotted share without interfering with others.

In addition to demonstrating how basic image operations can profit from parallel computing architectures like multicore processors using pthreads, parallelizing this task drastically cuts down on execution time when processing large images.

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 parallelism with 4 threads :

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

#define NUM_THREADS 4

// Struct to pass arguments to each thread
typedef struct {
    int thread_id;
    int start_row;
    int end_row;
    int width;
    int height;
    int** image;
    int** output;
} ThreadArgs;

// Allocate 2D array
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;
}

// Initialize image with random grayscale values
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;
}

// Thread function for parallel blurring
void* thread_blur(void* arg) {
    ThreadArgs* args = (ThreadArgs*) arg;
```

```
    for (int i = args->start_row; i < args->end_row; i++) {
        for (int j = 1; j < args->width - 1; j++) {
            int sum = 0;
            for (int di = -1; di <= 1; di++) {
                for (int dj = -1; dj <= 1; dj++) {
                    sum += args->image[i + di][j + dj];
                }
            }
            args->output[i][j] = sum / 9;
        }
    }

    pthread_exit(NULL);
}

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

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

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

    pthread_t threads[NUM_THREADS];
    ThreadArgs args[NUM_THREADS];

    int rows_per_thread = (height - 2) / NUM_THREADS;

    clock_t start = clock();

    for (int i = 0; i < NUM_THREADS; i++) {
        args[i].thread_id = i;
        args[i].start_row = 1 + i * rows_per_thread;
        args[i].end_row = (i == NUM_THREADS - 1) ? height - 1 :
args[i].start_row + rows_per_thread;
        args[i].width = width;
        args[i].height = height;
        args[i].image = image;
        args[i].output = output;
```

```
        pthread_create(&threads[i], NULL, thread_blur,
(void*)&args[i]);
    }

    for (int i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }

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

    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

Thread	Start Row	End Row
3	750	998

- Pthread functions/structs used:
  pthread_create:
  pthread_create(&threads[i], NULL, thread_blur, (void*) &args[i]);
  pthread_join:
  for (int i = 0; i < num_threads; i++) {
  pthread_join(threads[i], NULL);
  }
  Thread Arguments Struct:
  typedef struct {
  int thread_id;
  int start_row;
  int end_row;
  int width;
  int** image;
  int** output;
  } ThreadArgs;

Now 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 <pthread.h>
#include <sys/time.h>

using namespace std;

typedef struct {
    int start_row;
    int end_row;
    int width;
    int height;
    int** image;
    int** output;
} thread_data_t;

void* blur_thread(void* arg) {
    thread_data_t* data = (thread_data_t*) arg;
    int start = data->start_row;
    int end = data->end_row;
    int width = data->width;
    int height = data->height;
    int** image = data->image;
    int** output = data->output;

    for (int i = start; i < end; i++) {
        if (i == 0 || i == height - 1) continue;
        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;
```

```
        }
    }
    pthread_exit(NULL);
}

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;

                pthread_t* threads = (pthread_t*) malloc(NUM_THREADS *
sizeof(pthread_t));
                thread_data_t* thread_data = (thread_data_t*)
malloc(NUM_THREADS * sizeof(thread_data_t));

                int rows_per_thread = height / NUM_THREADS;
```

```
                gettimeofday(&start, NULL);

                for (int i = 0; i < NUM_THREADS; i++) {
                    thread_data[i].start_row = i * rows_per_thread;
                    thread_data[i].end_row = (i == NUM_THREADS - 1) ?
height : (i + 1) * rows_per_thread;
                    thread_data[i].width = width;
                    thread_data[i].height = height;
                    thread_data[i].image = image;
                    thread_data[i].output = output;

                    pthread_create(&threads[i], NULL, blur_thread,
(void*) &thread_data[i]);
                }

                for (int i = 0; i < NUM_THREADS; i++) {
                    pthread_join(threads[i], NULL);
                }

                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(threads);
                free(thread_data);
            }
        }

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

    return 0;
}
```

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

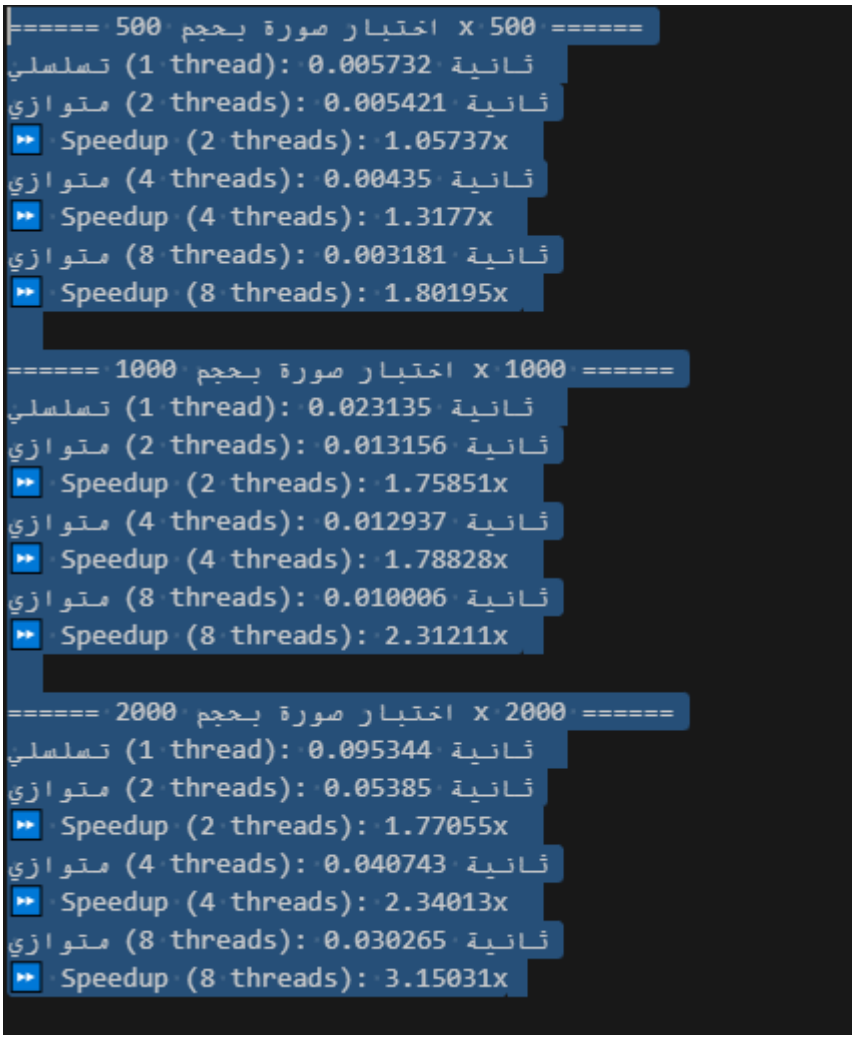
    - Input Sizes Tested:

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

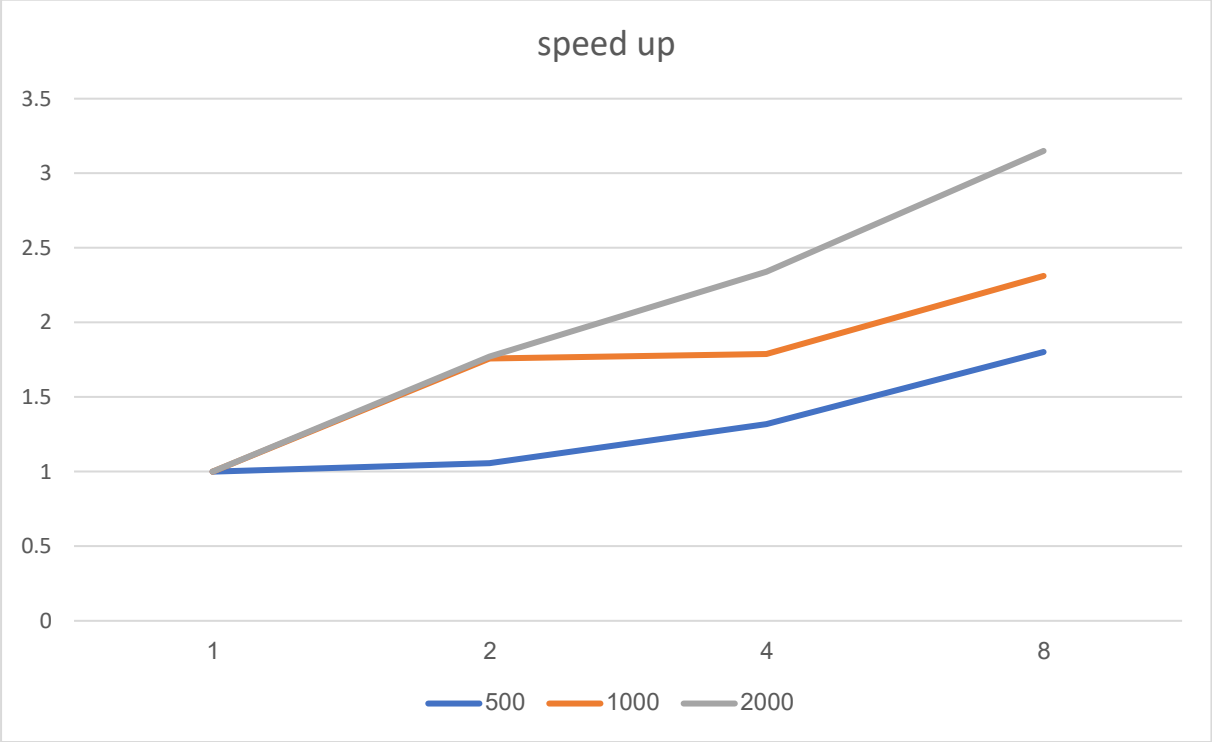
5. Results:

Image Size	Threads	Time (seconds)	Speedup
500 x 500	1	0.005732	1×
	2	0.005421	1.057×
	4	0.004350	1.318×
	8	0.003181	1.802×
1000 x 1000	1	0.023135	1×
	2	0.013156	1.759×
	4	0.012937	1.788×
	8	0.010006	2.312×
2000 x 2000	1	0.095344	1×
	2	0.053850	1.771×
	4	0.040743	2.340×
	8	0.030265	3.150×

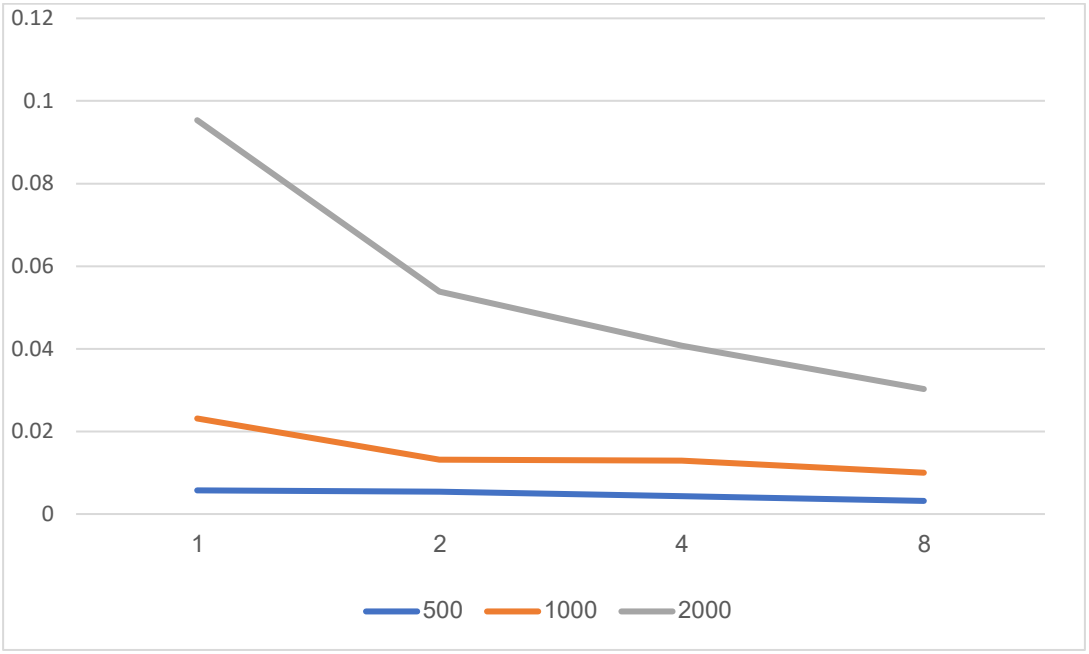
This is a picture from compiler :



- 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 **Pthreads library**, allowing us to divide the computational workload across multiple threads.

Through our experiments on different image sizes, we observed significant performance improvements when using multithreading, especially for larger images. The use of **Pthreads** demonstrated the efficiency and power of parallel programming in speeding up compute-intensive applications.

Lessons Learned

- We learned how to **divide work among threads** effectively using custom thread data structures.
- We utilized key `pthread` functions such as `pthread_create` and `pthread_join` to manage and synchronize threads.
- We gained experience in **dynamic memory allocation** for large 2D arrays and ensured proper memory management.
- We applied **time measurement techniques** using `time.h` to accurately evaluate execution performance.

Challenges Faced

- Balancing the workload across threads was sometimes difficult, especially when the number of rows was not evenly divisible by the number of threads.
- We encountered challenges in ensuring proper memory allocation and avoiding memory leaks or segmentation faults.
- Debugging multithreaded code required careful attention to synchronization and correctness.

Tools and Resources:

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