THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

PARALLEL AND DISTRIBUTED PROCESSING

Parallel And Distributed Processing Project 1:

Histogram-image processing

Student: Hashem Erainat

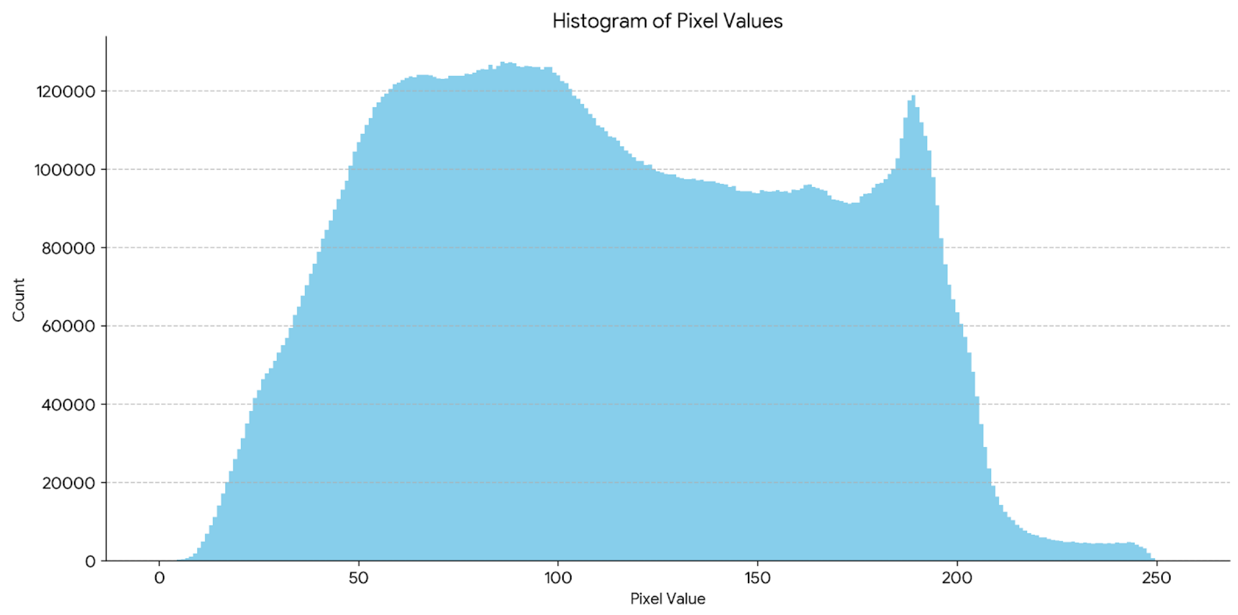
Instructor: Hussein Younes

1. Introduction

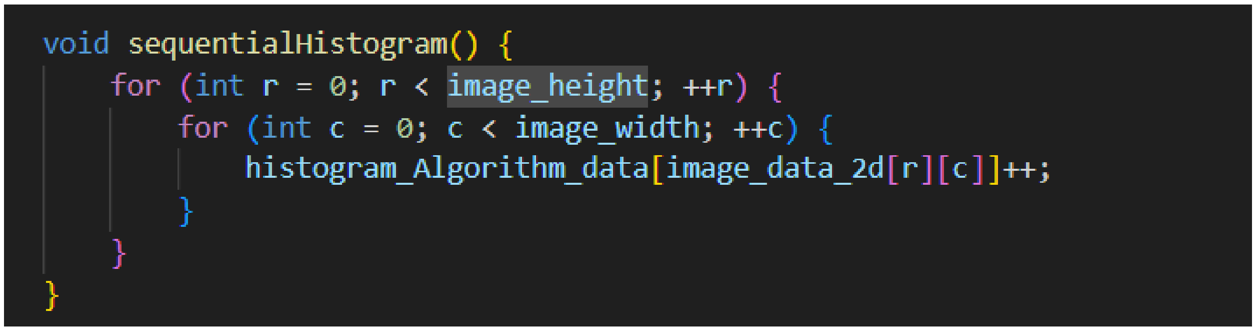
With the increasing need for fast and efficient image analysis, especially in real-time applications, optimizing performance has become more important than ever. In this report, we focus on the Histogram Calculation algorithm, a widely-used technique in grayscale image processing. Its main purpose is to analyze and represent how often each pixel intensity value appears in an image. The goal of this work is to convert this algorithm from a sequential to a multithreaded version, comparing its performance when executed sequentially versus in parallel, and to quantify the benefits of parallel processing for this specific task.

Histogram calculation is an ideal choice for this kind of comparison because it's naturally easy to parallelize. Each pixel in the image can be processed on its own, without needing to know anything about the others. This makes it simple to break the work into parts that can be handled separately. In our implementation, we load standard image files like JPG or PNG using the lightweight and popular stb_image library. In our implementation, we load standard image files like JPG or PNG using the lightweight and popular stb_image library, converting them to grayscale upon loading. The image data is then stored in memory as a 2D vector where each pixel's brightness is represented by a value between 0 (black) and 255 (white).

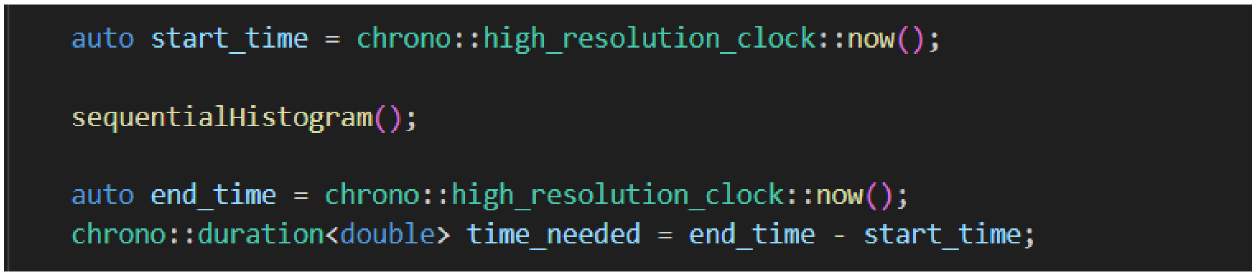
To take advantage of multiple CPU cores, we use Pthreads to split the image into equal parts, assigning each thread a group of rows to work on. Each thread independently calculates its own local histogram, a crucial design choice that eliminates the need for complex thread synchronization mechanisms like mutexes during the main processing phase. This approach significantly reduces overhead and helps maintain high performance by avoiding contention and data races.. Once all the threads finish their part, we combine the results in a final step by merging the local histograms into one global histogram. This approach efficiently uses the power of modern processors and can lead to noticeable speedups, especially when dealing with large, high-resolution images.



2. Sequential Implementation



```
void sequentialHistogram() {
    for (int r = 0; r < image_height; ++r) {
        for (int c = 0; c < image_width; ++c) {
            histogram_Algorithm_data[image_data_2d[r][c]]++;
        }
    }
}
```



```
auto start_time = chrono::high_resolution_clock::now();

sequentialHistogram();

auto end_time = chrono::high_resolution_clock::now();
chrono::duration<double> time_needed = end_time - start_time;
```

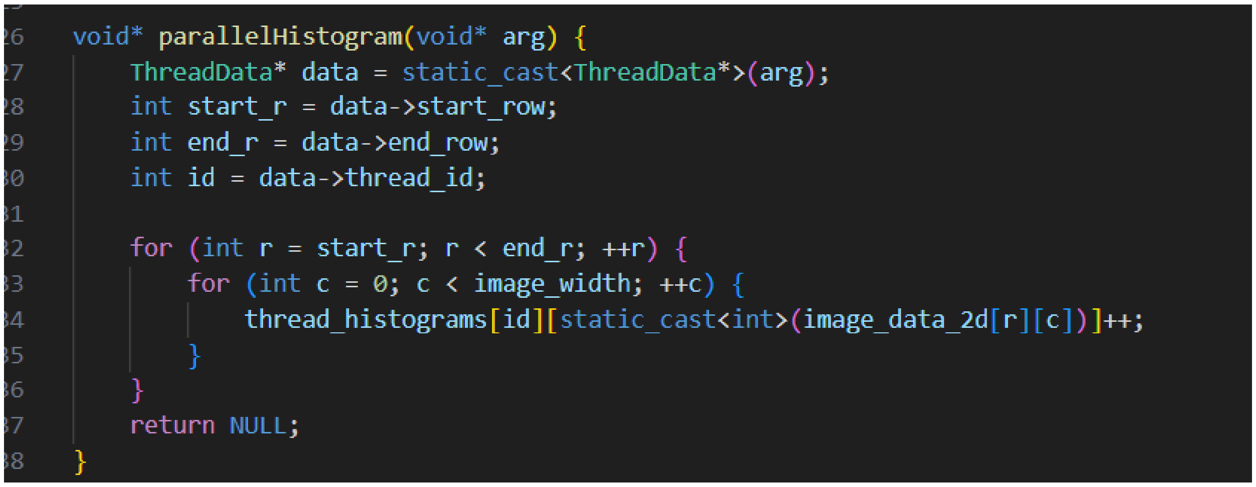
In this part of the project, the histogram algorithm is implemented in a simple, sequential way to operate on grayscale images. The idea is to determine how many times each pixel intensity value (from 0 to 255) appears in the image.

Initially, the image is loaded from a file and converted to grayscale with the help of a basic image-loading library. After being loaded, the image data is stored in memory in a way that makes it convenient for us to loop over every pixel separately.

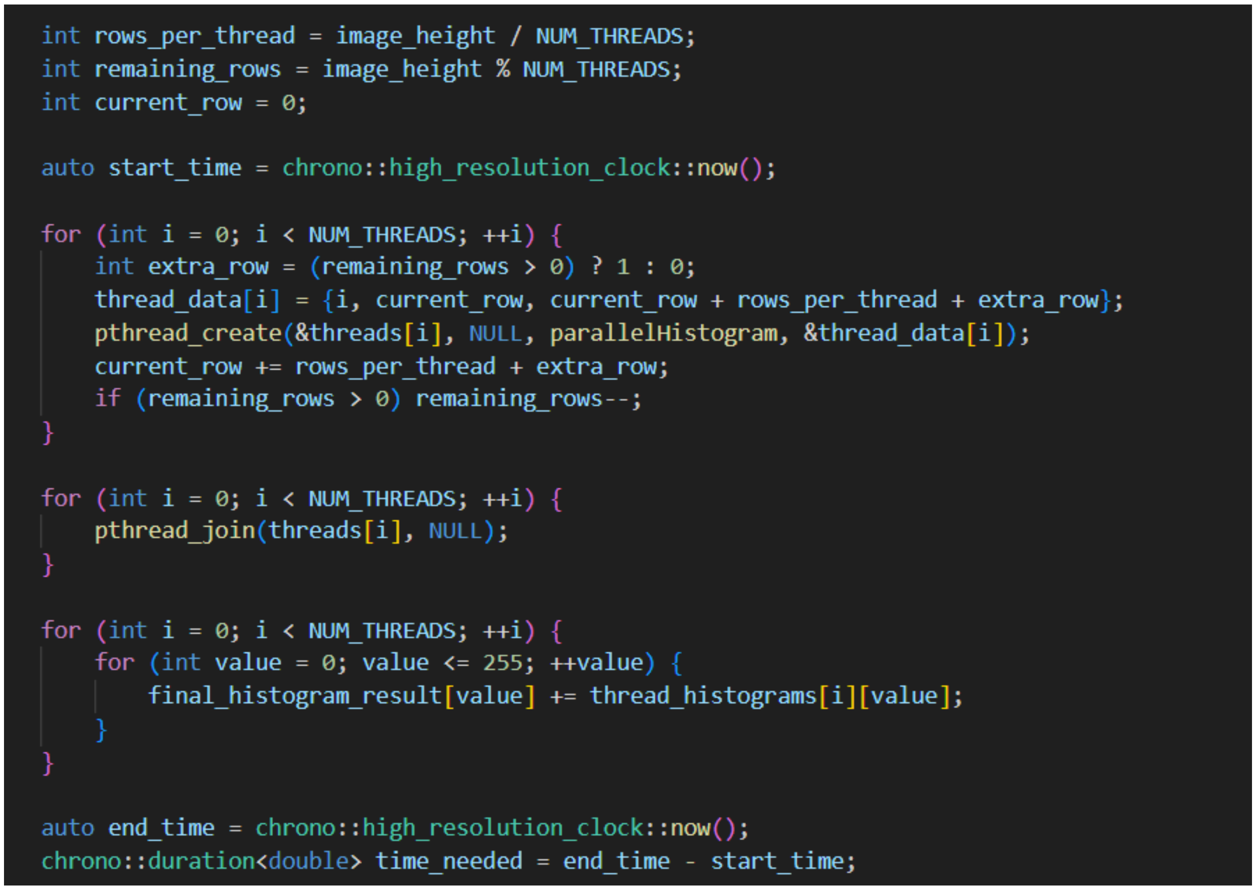
The majority of the algorithm loops over every pixel in the image using nested loops. For every pixel, it increments the counter of its intensity in a histogram array. There is no parallelism here, so the entire image is processed by one thread, so this version is easier to debug and implement but not the most performing.

To determine how long the sequential version takes, execution time is measured before and after the computation. This is for later comparison with the parallel version to see how much speedup is achieved by using multiple threads.

3. Parallelization Strategy



```
void* parallelHistogram(void* arg) {
    ThreadData* data = static_cast<ThreadData*>(arg);
    int start_r = data->start_row;
    int end_r = data->end_row;
    int id = data->thread_id;

    for (int r = start_r; r < end_r; ++r) {
        for (int c = 0; c < image_width; ++c) {
            thread_histograms[id][static_cast<int>(image_data_2d[r][c])]++;
        }
    }
    return NULL;
}
```



```
int rows_per_thread = image_height / NUM_THREADS;
int remaining_rows = image_height % NUM_THREADS;
int current_row = 0;

auto start_time = chrono::high_resolution_clock::now();

for (int i = 0; i < NUM_THREADS; ++i) {
    int extra_row = (remaining_rows > 0) ? 1 : 0;
    thread_data[i] = {i, current_row, current_row + rows_per_thread + extra_row};
    pthread_create(&threads[i], NULL, parallelHistogram, &thread_data[i]);
    current_row += rows_per_thread + extra_row;
    if (remaining_rows > 0) remaining_rows--;
}

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

for (int i = 0; i < NUM_THREADS; ++i) {
    for (int value = 0; value <= 255; ++value) {
        final_histogram_result[value] += thread_histograms[i][value];
    }
}

auto end_time = chrono::high_resolution_clock::now();
chrono::duration<double> time_needed = end_time - start_time;
```

In this part of the project, the goal was to improve performance by parallelizing the histogram calculation using Pthreads. By running the algorithm on multiple threads, we aim to reduce execution time and take full advantage of multi-core processors.

3.1 How Work Is Divided Among Threads

The image is divided based on rows. Each thread is assigned a specific range of rows to process. For example, if the image has 1000 rows and there are 4 threads, each thread will handle around 250 rows. Within their assigned portion, threads calculate a local histogram independently from the others.

This approach avoids the need for synchronization (like using mutexes), because threads don’t write to a shared histogram while processing. Each one works on its own copy. After all threads finish, their local histograms are combined into the final result in a separate step. This final merge is fast and doesn’t affect overall performance much.

3.2 Pthread Functions and Structures Used

To implement multithreading, the following tools from the Pthread library were used:

- pthread_t to create thread objects.
- pthread_create to launch each thread and assign it a task.
- pthread_join to wait for all threads to complete before moving on.
- A custom struct is used to pass necessary information to each thread, like:

- Start and end row indices.
- A pointer to the image data.
- A pointer to that thread’s local histogram.

This setup keeps things organized and avoids conflicts between threads. Overall, the parallel version of the algorithm is much faster—especially with large images—and shows how effective parallel programming can be for this kind of task.

4. Experiments

4.1 Hardware Specifications

The performance tests were conducted on a system with the following hardware and software configuration:

Processor (CPU): Intel Core i7-12700H

Number of Cores/Threads: 14 Physical Cores / 20 Logical Threads

4.2 Input Sizes and Thread Counts Tested

For testing, grayscale images of various resolutions were used. The main test image was a JPG file with a resolution of 3474x5441.

The parallel version of the algorithm was tested using different thread counts to observe how performance scales. The following thread counts were used:

1 thread (baseline for sequential comparison)

2 threads

4 threads

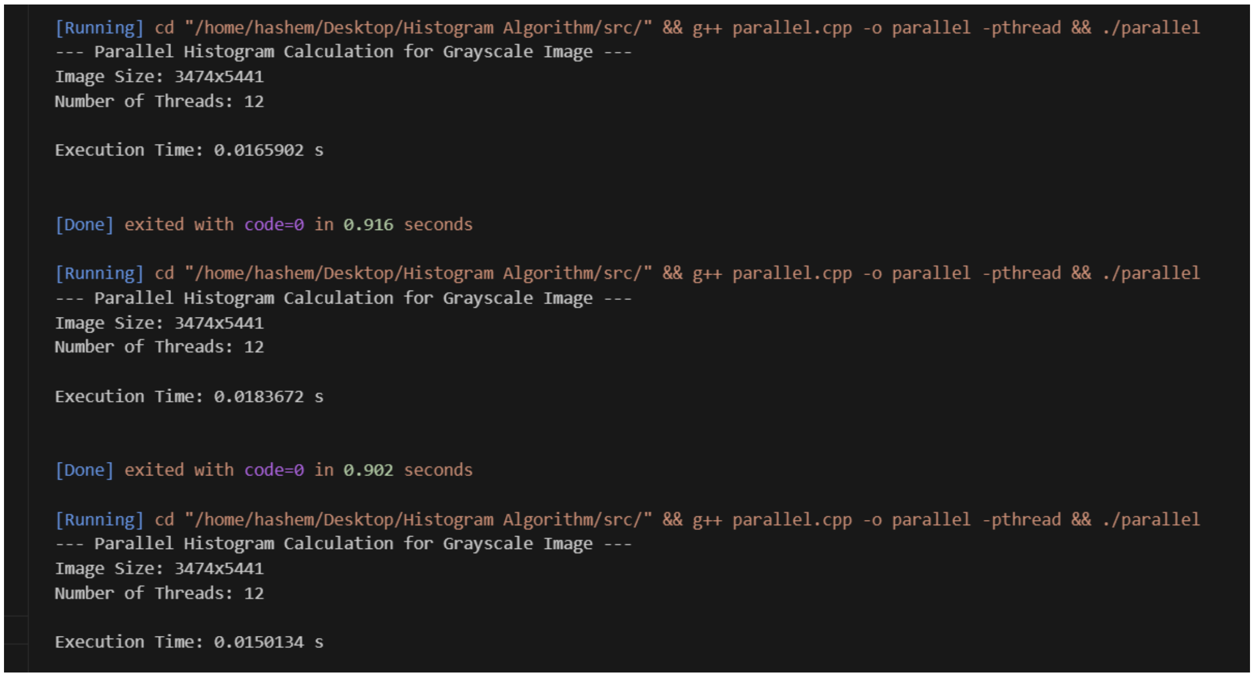
8 threads

16 threads

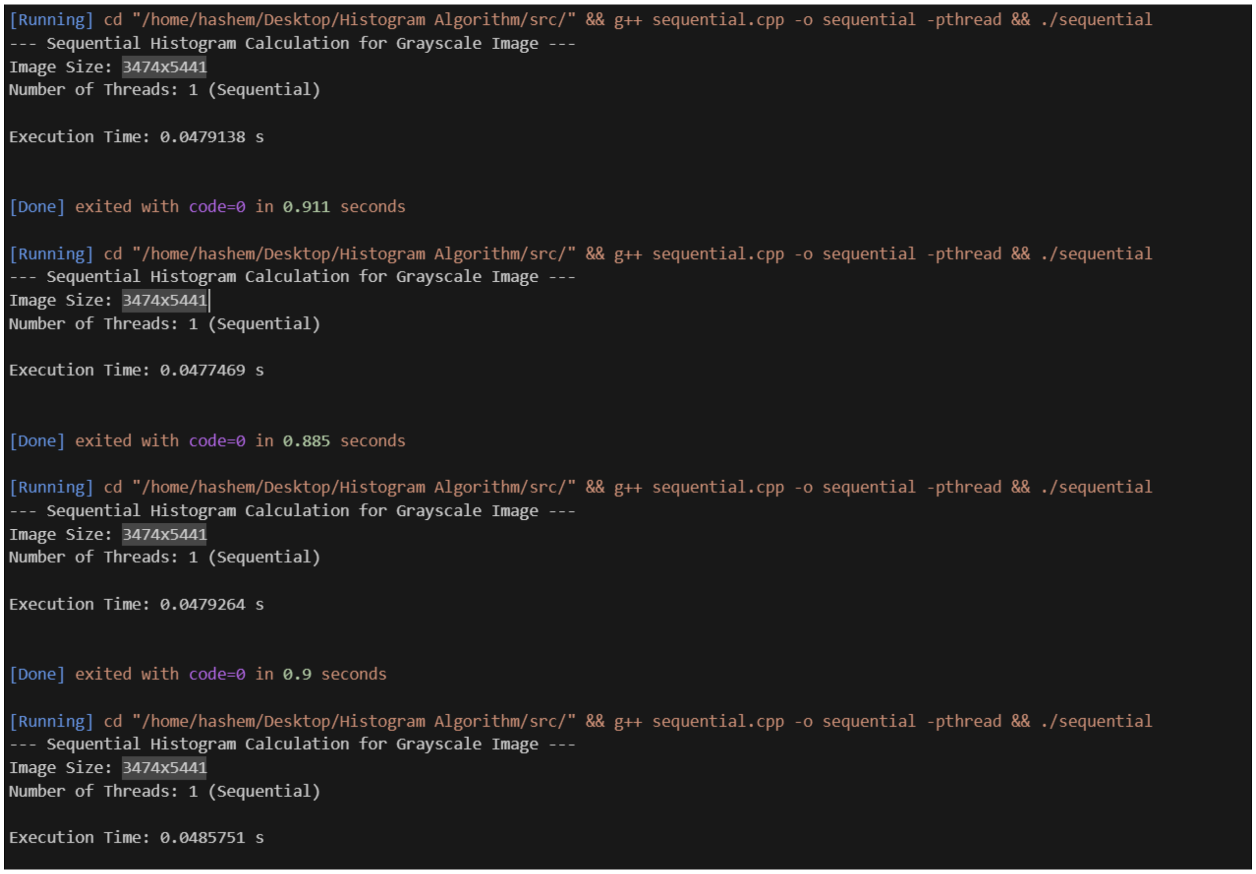
For each thread, the execution time was measured, and the speedup was calculated by comparing it with the time taken by the sequential version.

5. Results

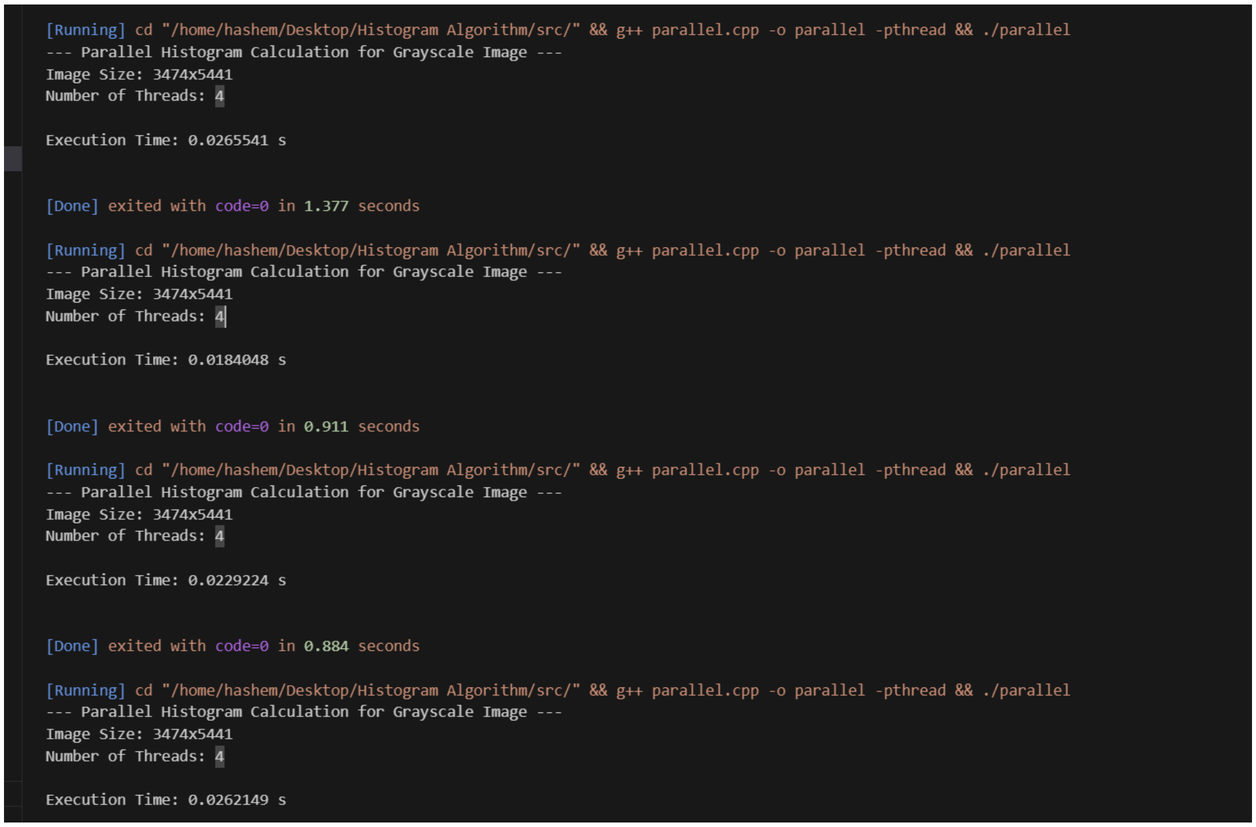
Some screenshot:



```
[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 12

Execution Time: 0.0165902 s


[Done] exited with code=0 in 0.916 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 12

Execution Time: 0.0183672 s


[Done] exited with code=0 in 0.902 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 12

Execution Time: 0.0150134 s
```



```
[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ sequential.cpp -o sequential -pthread && ./sequential
--- Sequential Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 1 (Sequential)

Execution Time: 0.0479138 s


[Done] exited with code=0 in 0.911 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ sequential.cpp -o sequential -pthread && ./sequential
--- Sequential Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 1 (Sequential)

Execution Time: 0.0477469 s


[Done] exited with code=0 in 0.885 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ sequential.cpp -o sequential -pthread && ./sequential
--- Sequential Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 1 (Sequential)

Execution Time: 0.0479264 s


[Done] exited with code=0 in 0.9 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ sequential.cpp -o sequential -pthread && ./sequential
--- Sequential Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 1 (Sequential)

Execution Time: 0.0485751 s
```



```
[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 4

Execution Time: 0.0265541 s


[Done] exited with code=0 in 1.377 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 4

Execution Time: 0.0184048 s


[Done] exited with code=0 in 0.911 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 4

Execution Time: 0.0229224 s


[Done] exited with code=0 in 0.884 seconds

[Running] cd "/home/hashem/Desktop/Histogram Algorithm/src/" && g++ parallel.cpp -o parallel -pthread && ./parallel
--- Parallel Histogram Calculation for Grayscale Image ---
Image Size: 3474x5441
Number of Threads: 4

Execution Time: 0.0262149 s
```

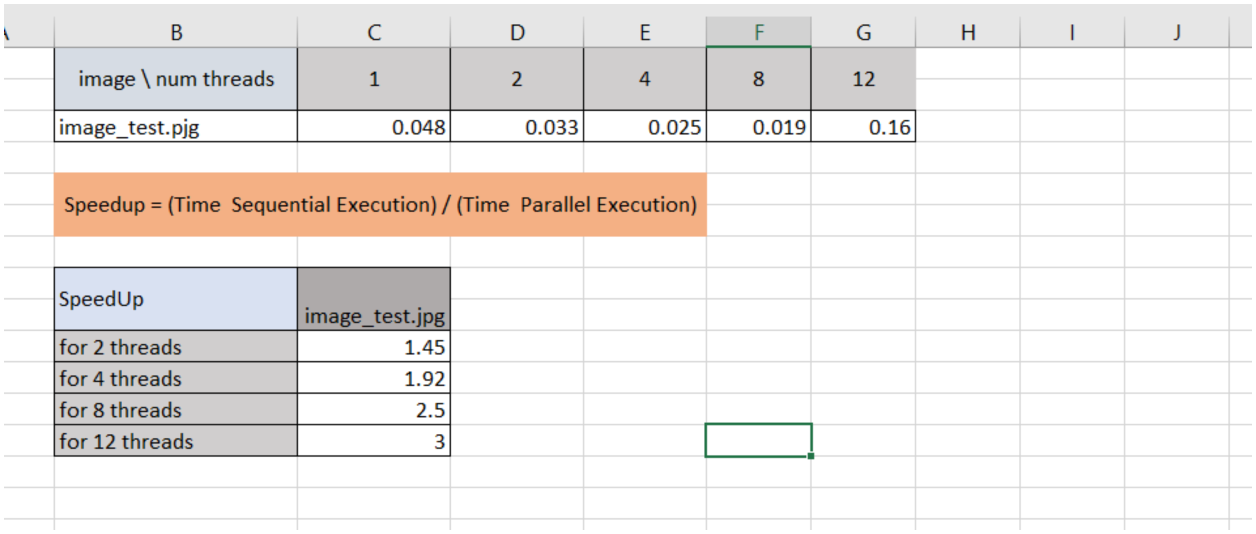
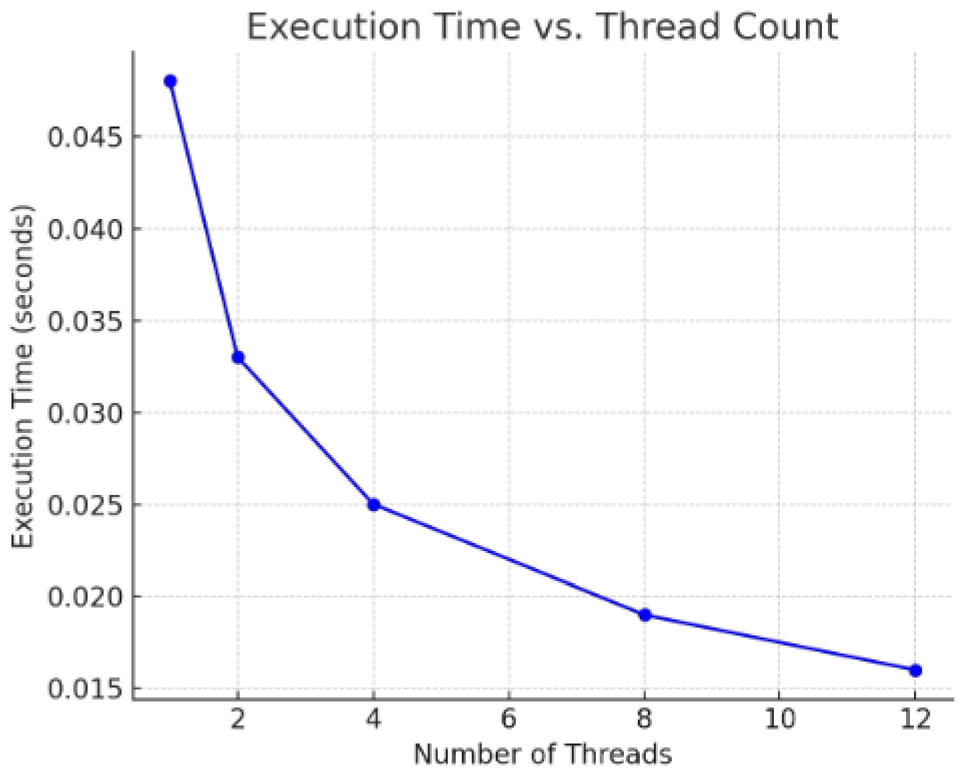
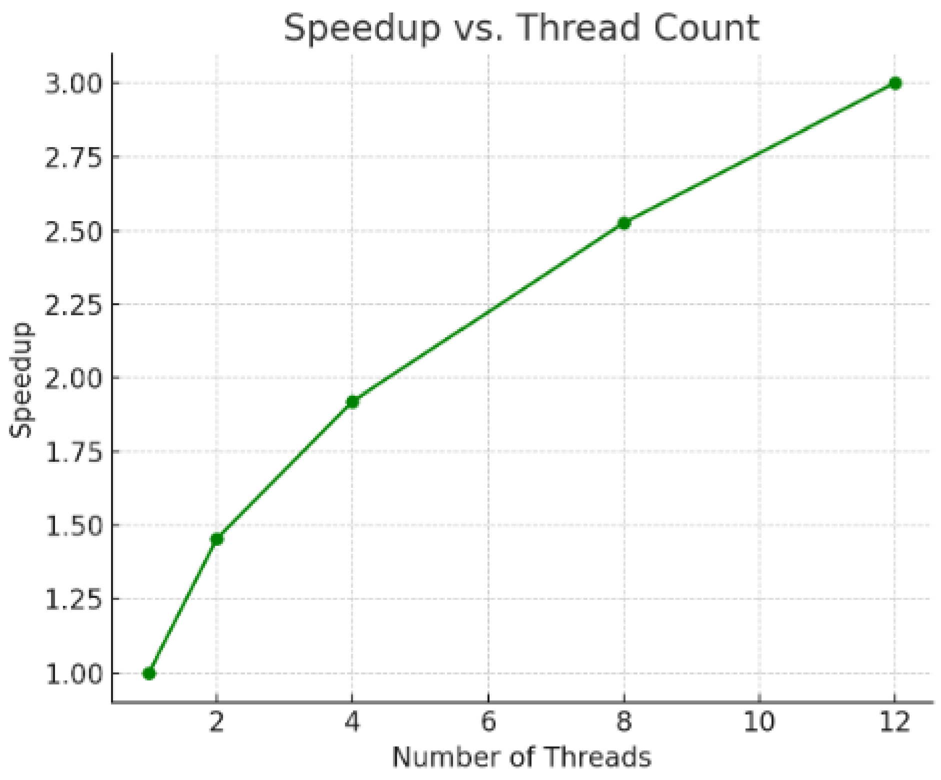


image \ num threads	1	2	4	8	12
image_test.pjg	0.048	0.033	0.025	0.019	0.16

Speedup = (Time Sequential Execution) / (Time Parallel Execution)

SpeedUp	image_test.jpg
for 2 threads	1.45
for 4 threads	1.92
for 8 threads	2.5
for 12 threads	3


Execution Time vs. Thread Count
Execution Time (seconds)
0.045
0.040
0.035
0.030
0.025
0.020
0.015
2
4
6
8
10
12
Number of Threads


Speedup vs. Thread Count
Speedup
3.00
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
2
4
6
8
10
12
Number of Threads

6. Discussion

Why Speedup Is Sublinear

While the execution time decreases with an increasing number of threads, the speedup is sublinear—meaning that doubling the number of threads does not halve the execution time. This is due to several reasons:

Thread Management Overhead:

- Creating and managing threads involves overhead.
- Synchronization, memory allocation, and combining partial histograms at the end all consume time.

Load Imbalance:

- The image rows might not divide evenly among threads, especially if the number of rows isn't a perfect multiple of the thread count.
- Some threads may finish earlier and stay idle while others continue working.

Hardware Limitations:

- Beyond a certain number of threads (e.g., number of physical/virtual cores), threads compete for CPU resources, leading to diminishing returns.

- At 12 threads, you see execution time slightly increase (0.016), likely due to context switching and CPU contention.

Memory Access Bottlenecks:

- Multiple threads accessing shared memory (like image data or histogram arrays) can cause cache thrashing or contention, slowing down execution.

Comparison to Amdahl’s Law

According to **Amdahl’s Law**, the maximum speedup achievable using multiple threads is fundamentally limited by the portion of the program that cannot be parallelized. The

$$\lim_{n\to\infty} Speedup = \frac{1}{(1 - F_{parallel})}$$

This implies that no matter how many threads are used, the speedup will eventually plateau due to non-parallelizable parts such as image loading, thread creation, or final histogram merging.

Looking at the actual results:

- With 2 threads, the speedup is 1.45
- With 4 threads, the speedup increases to 1.92
- With 8 threads, it reaches 2.5
- With 12 threads, the maximum observed speedup is 3.0

These results demonstrate a sublinear speedup — the performance improves with more threads but at a decreasing rate. This trend aligns with Amdahl’s Law, which predicts diminishing returns as thread count increases. For example, the jump from 2 to 4 threads results in a noticeable gain (1.45 → 1.92), but from 8 to 12 threads the gain is smaller (2.5 → 3.0), illustrating the impact of the sequential portion that cannot benefit from parallelization.

7.Conclusion

Lessons Learned and Challenges Faced

We gained some great insights into the strengths and limitations of parallel programming throughout this project. One of them is that parallelization offers an extremely significant performance improvement, especially when working with huge sets of data such as images. Work parallelization through splitting it into several threads helped us reduce execution time and achieve up to 3x speedup compared to the sequential version.

We also encountered several challenges:

- Diminishing returns: As may be observed from our results, the performance gain reduces with an increase in the number of threads. This is due to thread management overhead and the presence of non-parallel code, as explained by Amdahl's Law.

- Load balancing: Distribution of image rows amongst threads in equal proportions was not always even, especially where the image height was not evenly divisible by the thread count. This required specific treatment to correctly distribute the leftover rows.

- Thread overhead: The creation and management of numerous threads involve added overhead, which negates the benefit of parallelism unless efficiently managed.

- Data merging: Merging the thread histograms into a final result involved a sequential step and had little impact on overall speedup.

In conclusion, while parallelization is defined well to yield performance advantages, achieving maximum benefit includes designing with caution to maintain low overhead, offer workload balance, and reduce the impact of sequential stages.

8. AI Assistance

- ChatGPT was used to understand how to load a grayscale image using stb_image.h and convert it into a 2D vector representation in C++ for easier pixel access during histogram computation.

- Gemini was also used to verify that the total of histogram values was equal to the number of image pixels to ensure accuracy of implementation.As a second verification check, this step was performed to continue to confirm the validity of the histogram calculation.
- Both tools were also used to refine the wording and improve the clarity of several texts..