Parallel Histogram Equalizer in C++ with OpenMP

Giovanni Colombo
Mat. 7092745
giovanni.colombo@edu.unifi.it

Abstract

This report describes the development of a simple Histogram Equalizer in C++ and quantitatively compares its sequential and parallel implementations. The analysis demonstrates the significant potential of parallel computing and multithreading using the OpenMP API. Performance measurements including speedup and efficiency are conducted across various environmental configurations, hyperparameters, and different combinations of directives and clauses. Finalterm Assignment for the Parallel Programming for Machine Learning course, taught by Professor Marco Bertini at the University of Florence, Italy.

1. Introduction

This report presents the development of a Histogram Equalizer implemented in C++, comparing its sequential and parallel implementations using the OpenMP API. The study focuses on examining the performance benefits of parallel computing through measurements of speedup and efficiency across various configurations of OpenMP directives and clauses.

Histogram equalization (Figure 1) is a fundamental digital image processing technique used to enhance the contrast of images by spreading out the most frequent intensity values across the histogram of pixel intensities. This method is particularly effective when applied to images with limited dynamic range or poor contrast distribution, such as underexposed or overexposed images.

Inputs are generally represented by grayscale images with intensity levels in the range [0,255], and the process of Histogram Equalization involves four key steps:

1. **Histogram computation**: calculates the frequency distribution of pixel intensities, and creates a histogram where each bin represents the number of pixels with a specific intensity level.
2. **Cumulative Distribution Function computation**: obtained by computing the running sum of histogram values.

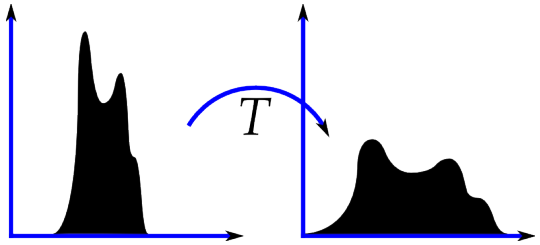

Figure 1: Visual representation of histogram equalization.

3. **Histogram normalization**: transforms the CDF in order to span the full intensity range [0,255]. The normalization formula is

$$h(v) = \left\lfloor \frac{(\text{CDF}(v) - \text{CDFmin}) \times 255}{\text{N} - \text{CDFmin}} \right\rceil$$

4. **Image transformation**: executed by mapping each original pixel intensity to its new value from the normalized CDF.

This algorithm should produce an output image with a more uniform intensity distribution. Its structure presents several opportunities for parallelization, particularly in the histogram computation and image transformation phases. However, it also introduces challenges such as potential race conditions during the histogram computation phase and the need for synchronization between different processing stages.

Moreover, while the implementation primarily focuses on grayscale images, it was later extended to handle and output color images through the HSV color space as an experimental enhancement, applying the equalization to the V (Value) channel while preserving hue and saturation information.

2. Method and Code

2.1. Hardware and Software setup

All tests were performed on an MSI Pulse GL76, with an Intel Core i7-12700H 14-core processor running at 2.7 GHz base frequency, and 16GB of RAM.

2.2. Data

To evaluate the quality of the histogram equalization operation, a dataset of 10 images was used, initially in JPG format and properly converted to PPM format. The dataset consists of two groups:

- 4 test images with specific black and white geometric patterns: checkerboard, circles, gradient, stripes.
- 6 of my photographs which were specifically preprocessed in order to create 3 deliberately overexposed images and 3 deliberately underexposed images.

All images are square and are used in four different sizes: 256, 512, 1024 and 2048 pixels. All images and their respective outputs are illustrated in Figures 9 and 10.

2.3. Code general structure

The project is built around a main **Equalizer** class, which is the core of the implementation as the class' public interface includes the two methods `renderSequential` and `renderParallel`, which implement the sequential and parallel versions of the program respectively. These methods return two structs, `SequentialResult` and `ParallelResult`, which encapsulate essential performance metrics for quantitative analysis of each execution.

Each implementation comprises four main steps. The execution time of each step is measured and recorded to facilitate all speedup calculations.

The `main.cpp` file allows defining several constants, enabling multiple combinations of parameters in the same execution:

- `IMAGE_SIZES`: size of the input images {256, 512, 1024, 2048}
- `NUM_THREADS`: number of forked threads {2, 4, 8, 12, 16, 24, 32}
- `BLOCK_SIZES`: number of blocks/chunks for scheduling {4, 8, 12, 16, 24, 32}

After an initial testing, I observed that the execution times were remarkably short, even for larger images, which complicated the quantitative analysis of performance improvements. To address this problem, I introduced the `NUM_TIMES` constant, so that every image is processed many consecutive times in the same way before moving to the next one. This approach effectively simulates a more substantial workload for each image, making performance differences more measurable. I decided to set this value to 1000, finding it to be an optimal balance between meaningful execution times and practical testing duration.

2.4. Sequential implementation

The histogram computation and image transformation steps (1 and 4) both have an $O(N)$ computational complexity, where N is the total number of the image pixels. In contrast, the CDF computation and normalization steps (2 and 3) remain constant $O(1)$ regardless of image size, as they always operate on 256 intensity levels.

The total computational complexity of the algorithm is then $O(N)$.

2.5. Parallel implementation

The histogram equalization algorithm is divided into 4 steps, whose parallelization must be studied individually.

The histogram computation step is *embarrassingly parallel*, as threads can work independently on different portions of the image pixels: each thread creates and computes its own local histogram which will eventually flow into the global image histogram. While managing in a multithreaded way introduces many advantages in the local computation phase, it requires careful handling during the accumulation of each histogram into the global one, due to potential race conditions that could emerge in case of simultaneous access to the global histogram by different threads. Therefore, it becomes necessary to create a critical section, or to use an atomic directive to address this problem.

The CDF computation step, on the contrary, is inherently sequential, as the sum must follow a precise order on the global histogram values. Though, the impact of this step is minimal as it always operates on just 256 values, resulting in extremely fast execution times regardless of the image size.

The CDF normalization step starts with a sequential part for finding the minimum non-zero value, and then it consists of the same operation performed independently on all the 256 histogram bins, so parallelization using a Map would theoretically be possible. However, it was observed that parallelizing this step actually causes a reduction in speedup, probably because the computational overhead generated by thread creation exceeds the benefits of parallelization.

The image transformation step is therefore the only other step where parallelization could provide real benefits. For each pixel, its intensity value in the input image is passed to the CDF, which returns the value to be assigned in the generation of the output image. As in the first step, each pixel can be processed independently, and since the CDF at this stage is only read and not modified, parallelization is possible without the need for synchronization between threads.

3. Performance Analysis and Results

The first analyses were aimed at verifying how effective the parallelization of each step would have been. Since the

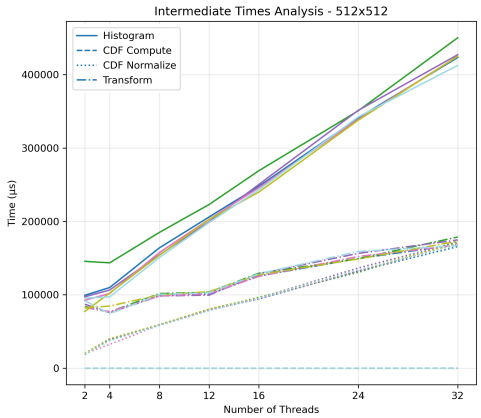

Figure 2: Execution times of intermediate steps on 512x512 images.

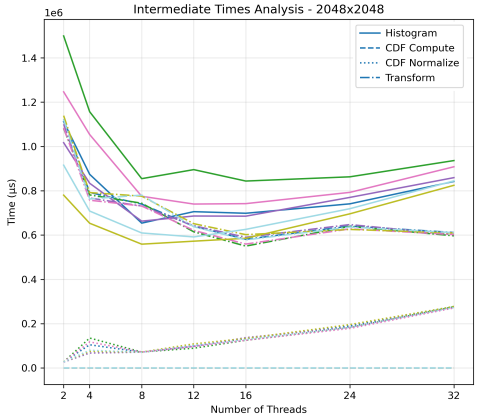

Figure 3: Execution times of intermediate steps on 2048x2048 images.

steps are independent of each other, it was possible to directly apply the same combination of OpenMP directives and clauses to each step.

Figures 2 and 3 clearly show how the impact of parallelization varies significantly with the image size. On small images (512×512), parallelization brings no benefits, as execution times of each step grow proportionally with the number of threads, indicating that the overhead of thread creation and management exceeds the potential benefits of parallel computation.

Conversely, on larger images (2048×2048), parallelization becomes effective as the execution times of histogram computation and image transformation steps (1 and 4) do not grow with the number of threads. However, parallelization of the CDF normalization step (3) is proved to be counterproductive as no enhancements are seen even for larger images.

As expected, the CDF computation step (2), which must be sequential, does not represent a computational bottleneck as its execution time is little.

3.1. Thread Scaling Analysis

The analysis of speedup and efficiency metrics across different image sizes reveals significant patterns in the performance scaling of this parallel implementation.

Small images (256×256) show consistently worse scaling behavior (Figure 4). The speedup never goes above 1, indicating that the parallel implementation actually performs worse than its sequential counterpart. The degradation becomes more evident as the number of threads increases, with extremely low efficiency values.

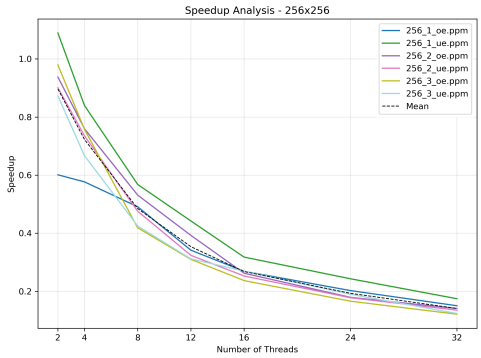

Figure 4: Speedup on 256x256 images.

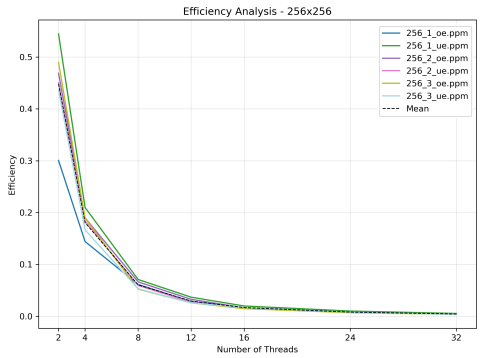

Figure 5: Efficiency on 256x256 images.

In contrast, for large images (2048×2048) results are promising (Figure 6), and suggest that even larger images can lead to even better performance. The speedup shows a consistent growth reaching a peak of 3.5x at 12 threads. Beyond this point, performance begins to degrade gradually. As expected, efficiency starts high (above 100% with 2 threads), and then declines progressively, though maintaining acceptable levels (around 30%) even at 12 threads.

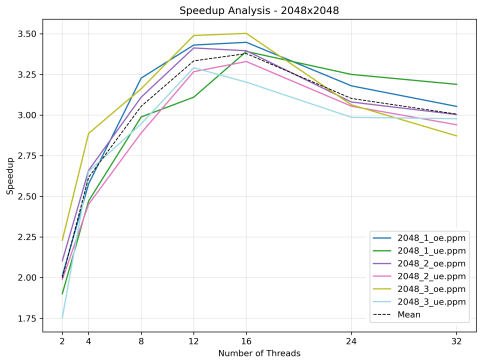

Figure 6: Speedup on 2048x2048 images.

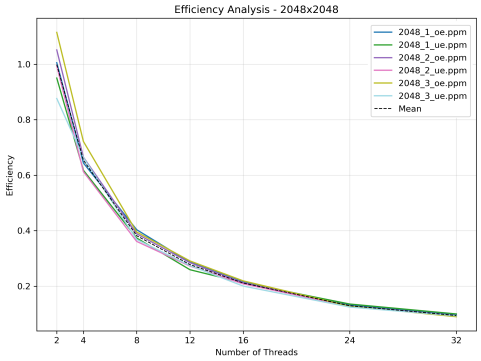

Figure 7: Efficiency on 2048x2048 images.

3.2. Scheduling Strategy Evaluation

The analysis of scheduling strategies reveals a clear superiority of static scheduling over dynamic (Figures 8 and **??**).

This difference is primarily motivated by the inherent characteristics of our histogram equalization algorithm. The workload at each step is generally uniform across the image, making dynamic work distribution unnecessarily complex.

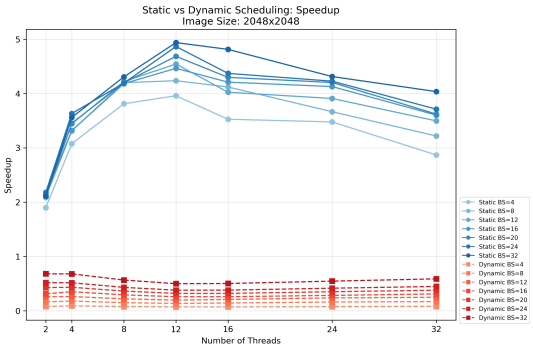

Figure 8: Speedup of different scheduling strategies on 2048x2048 images.

3.3. Race Condition management

In the histogram computation step (1), where multiple threads need to update the same global histogram concurrently, it was necessary to manage the possibility of race conditions. To ensure correct results, I implemented and tested two different synchronization approaches:

- Creating a critical section using the `critical` directive

```
#pragma omp parallel
    {...
        #pragma omp critical
        {
            for(int i = 0; i < 256;
                ↪ i++) {
                inputImage.histogram.counts[i]
                    ↪ += local_hist[i];
            }
        }
    }
```

- Using atomic operations with the `atomic` directive

```
#pragma omp parallel
    {...
        for(int i = 0; i < 256; i++) {
            #pragma omp atomic
            inputImage.histogram.counts[i]
                ↪ += local_hist[i];
        }
    }
```

The critical section implementation protects the entire histogram accumulation block, while the atomic approach provides more granular synchronization.

Performance analysis of these two approaches on 2048×2048 images shows similar results, with both implementations achieving maximum speedup around 3.3x with 16 threads, and efficiencies progressively getting lower, setting the best number of threads at 2-4.

After that, I decided to test also the implementation without any synchronization mechanism. Similar performance metrics were observed, reaching speedup values comparable with the ones measured when using the `critical` directive. However, while the unsynchronized version appears to work, it threats the correctness of the results due to race conditions that could occur.

4. Conclusion

This report presented the development of a Histogram Equalization algorithm in its sequential and parallel implementations using OpenMP. Through careful management of each step, the effectiveness of parallelization was demonstrated in terms of speedup and efficiency.

The performance analysis evaluated the algorithm performance across different number of threads and scheduling strategies. Particular attention was given to race condition threats in the histogram computation phase, leading to different synchronization implementations.

The best configuration proved to be the parallelization of the first and last steps using static scheduling, and implementing a critical section for addressing race conditions. This configuration achieved a speedup up to 3.5x with 12 threads and maintained efficiency between 100% and 80% when using 2-4 threads.

Additionally, I briefly tested a color version by converting RGB images to HSV and equalizing only the V channel, obtaining colorful equalized images.

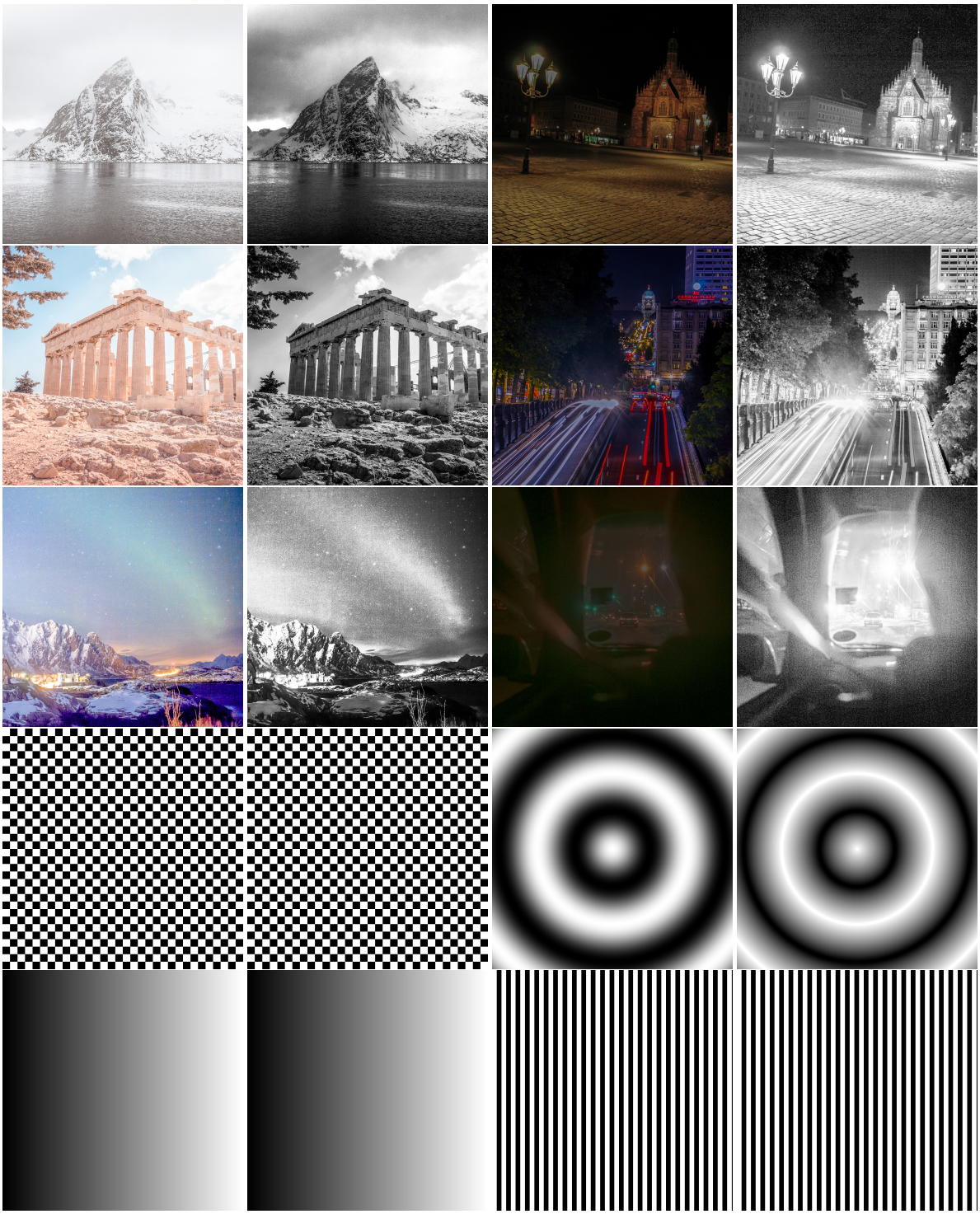

Figure 9: Input images and their respective grayscale outputs.

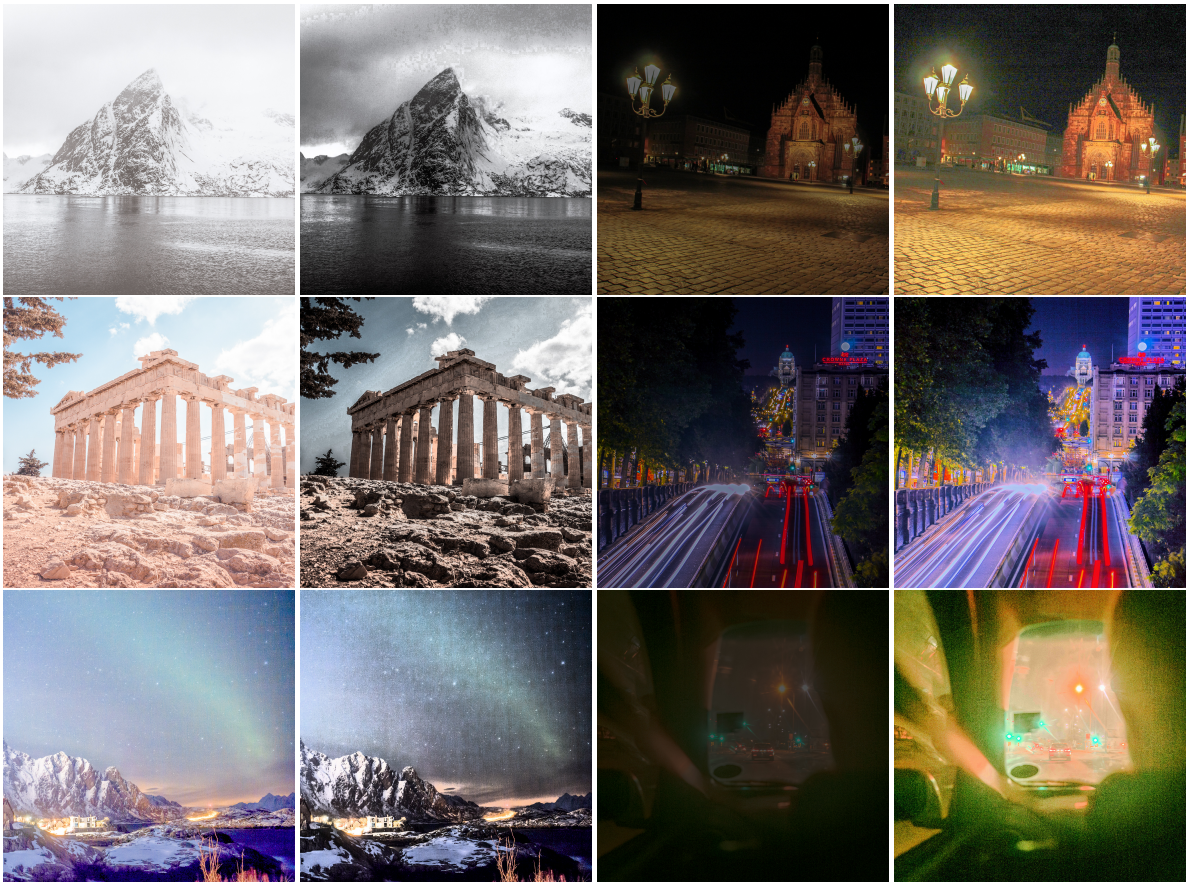

Figure 10: Input images and their respective color outputs.