**National Textile University, Faisalabad**



|  |  |
| --- | --- |
| **Assignment** | 1 |
| **Name** | Hadiah Iftikhar |
| **Registration No** | 22-NTU-CS-1159 |
| **Semester** | BSCS-6th (Section A) |
| **Course** | Parallel and Distributed Computing |
| **Submitted To** | Sir Nasir Mahmood |
| **Submission Date** | 11th March, 2025 |

# Problem Description

In order to parallelize a sequential code problem and compare the results, I implemented the histogram equalization algorithm, which is used to enhance the contrast of an image by redistributing its pixel intensities.

# Task 1: Sequential Implementation

In the sequential implementation, the process is divided into four steps:

* **Calculate Histogram:**  
  Traverse the entire image to count the frequency of each pixel intensity (0–255).
* **Calculate Cumulative Distribution Function (CDF):**  
  Compute the cumulative sum of the histogram values to obtain the CDF. This step accumulates frequencies so that each intensity level is mapped to a cumulative value.
* **Normalize the CDF:**  
  Adjust the CDF values so that they scale between 0 and 255. This normalization improves the contrast by stretching the intensity range.
* **Map CDF to Output Image:**  
  Replace each pixel in the input image with its corresponding normalized CDF value, thus generating the equalized image.

The sequential code straightforwardly processes each pixel and performs the above steps without any parallelization. Its simplicity ensured correctness but left performance gains on multi-core processors largely untapped, resulting in a high execution time.

# Task 2: Parallel Implementation

To exploit parallel processing and reduce execution time, the sequential algorithm was restructured using OpenMP. The following outlines which sections were parallelized and why:

## Parallelized Regions

### Histogram Calculation:

**Challenge:**  
Had to parallelize the nested loop which was used to fill the histogram array, but updating a shared histogram array in parallel can lead to race conditions.

**Solution:**  
Used the ‘*for’* and ‘*collapse’* directives to parallelize the nested loops, and then to ensure that the threads safely accessed the critical section, OpenMP’s reduction clause was used to allow each thread to maintain a private copy of the histogram array and then safely combine them.

**Code Snippet:**

#pragma omp parallel for reduction(+: histogram[:256]) collapse(2)

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

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

histogram[ img[i][j] ]++;

}

}

Here, collapse(2) flattens the nested loops, and reduction(+: histogram[:256]) ensures thread-safe accumulation by splitting the initial histogram array and allowing each thread to work on its portion and then adding all the portions at the end to form the complete, filled histogram.

### Mapping the Normalized CDF to the Output Image:

**Reasoning:**  
Each pixel’s new value is computed independently, making it an ideal candidate for parallelization. Since, nested loops are used here as well, therefore *for* and *collapse* were used to parallelize the code block.

**Code Snippet:**

#pragma omp parallel for collapse(2)

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

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

output\_img[i][j] = cdf[ img[i][j] ];

}

}

## Regions Kept Sequential

1. CDF Calculation:  
   The computation of the cumulative distribution is sequential by nature since each element depends on the previous one. If we had parallelized this portion, then we would have probably run into a segmentation fault or calculated wrong answers, if while computing the value of a certain cdf[i[, cdf[i-1] had not been computed yet. Moreover, given only 256 iterations, parallelization would not provide significant benefit and might even increase overhead.
2. Image Generation and Output Printing:  
   Generating the image (especially when using random numbers) and printing the result were kept sequential. For image generation, thread safety and reproducibility (by using a fixed seed) are more critical than performance, and printing must maintain a defined order which would have been disrupted if the loop was parallelized as the order of printing would have depended on the different threads’ performance..

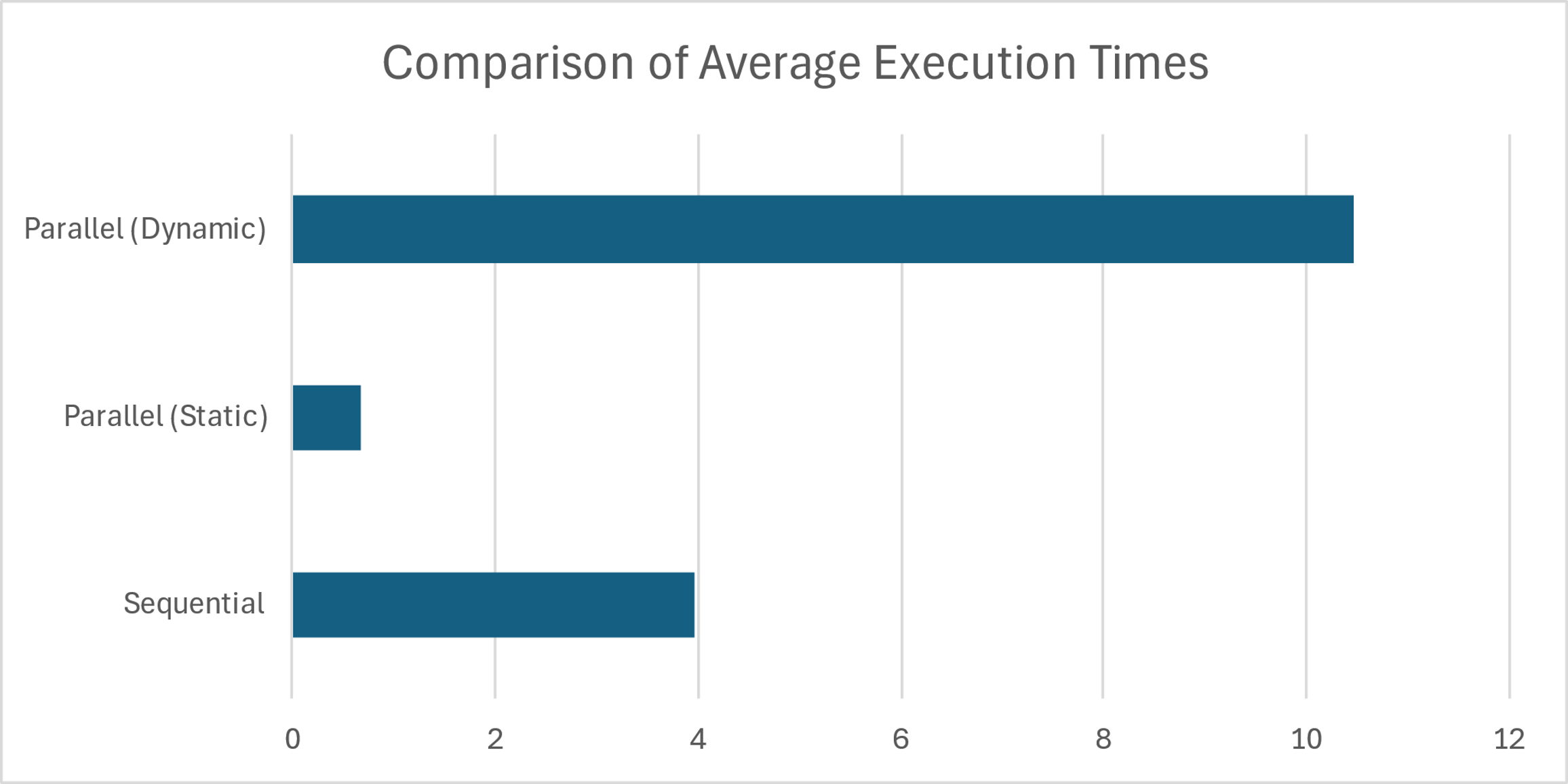
### Normalization of the CDF:

### While the normalization of the CDF is a simple operation with only 256 iterations, it was also parallelized in the beginning. However, the benefit is negligible given the small iteration count. Thus, this portion was ultimately kept sequential in the final version to avoid parallel overhead.

# Task 3: Results

Three versions of the code were compared based on their execution times:

* **Sequential Code:**  
  Processes each pixel one at a time. For large images (e.g., 17000×17000 pixels), this version had an average execution time of 3.9545254 seconds.
* **Parallel Code with Static Scheduling:**  
  Using OpenMP’s static scheduling, which divides the loop iterations evenly among the threads, resulted in a significant speedup. The static version’s execution time averaged about 0.6712101 seconds.
* **Parallel Code with Dynamic Scheduling:**  
  In contrast, dynamic scheduling—which assigns loop iterations on-the-fly—introduced excessive overhead due to the large number of iterations and uniform workload (which dynamic scheduling is not suitable for). The dynamic version averaged around 10.46 seconds, which is considerably slower than both the sequential and static scheduling versions.



### Key Observations:

* **Static Scheduling** is highly effective for this problem because the workload per iteration (processing a pixel) is uniform. The overhead is minimal, and each thread processes a contiguous block of data.
* **Dynamic Scheduling** suffers from high scheduling overhead in this context, as its benefit in balancing load is not required when all iterations take approximately the same time.
* **Sequential vs. Parallel:**  
  While the parallel static version is clearly advantageous on large images, the benefits might diminish for smaller images due to thread management overhead.

# Task 4: Performance Analysis Through Amdahl’s

The result exhibits that the execution time and, thus, the performance of the code improved after it was parallelized, but to show how much speedup was achieved, Amdahl’s law will be used:

**Speedup (S) = 1 / (1-F(Parallel)) + (F(Parallel)/N)**

Parallel Portion = Parallel Work/ Total Work ​= 578000000 / 867000511 = 0.6667

N (number of cores) = 4

**Speedup = 2.0001**

# Task 5: Conclusion

In summary, the problem demonstrated that:

* **Problem Suitability:**  
  The type of the problem determines whether the code can be parallelized or not. If the code is such that statements that manipulate data depend on each other then the code is sequential by nature and should not be parallelized. Similarly, loop scheduling type must also be determined on the basis of the uniformity of the workload, as choosing the wrong scheduling type might decrease efficiency instead of enhancing it.
* **Effective Parallelization:**  
  The parallel implementation using OpenMP with static scheduling significantly reduces execution time over the sequential version by evenly dividing the workload among threads.
  + The **static scheduling** approach yielded a real performance gain, while
  + The **dynamic scheduling** approach introduced unnecessary overhead and degraded performance.
* **Conditions for Parallelization:**
  + **Uniform Workload:** Static scheduling is preferred when each iteration requires nearly the same computation, as in pixel processing.
  + **Overhead Considerations:** For tasks with low computational intensity or small data sizes, the overhead of parallelization (thread management, synchronization) may outweigh the benefits.

Thus, the experiment confirms that the selection of an appropriate scheduling strategy is crucial. For histogram equalization on large images, static scheduling allows us to effectively harness the power of multi-core processors, whereas dynamic scheduling is better reserved for tasks with irregular workloads.