**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** | 9th March, 2025 |

The task was to choose a coding problem and implement it using both sequential and parallel methods. The problem I chose for this assignment was the histogram equalization algorithm, which enhances the contrast of an image by adjusting the pixel intensity distribution. Initially, a sequential approach was implemented, followed by a parallelized version using OpenMP to utilize multiple threads and improve execution time for larger images. In addition, synchronization mechanisms such as static and dynamic loop scheduling were applied to manage workload distribution and reduce overhead.

# Task 1: Sequential Implementation

The sequential implementation of the histogram equalization process followed these steps:

1. **Calculate Histogram**: Traverse the image and count the occurrences of each pixel intensity (from 0 to 255).
2. **Calculate CDF**: Based on the histogram, compute the cumulative distribution function (CDF) that maps pixel intensities to a range of values that enhances contrast.
3. **Normalize CDF**: Normalize the CDF to fit the 0-255 range.
4. **Map CDF to Output Image**: Map the normalized values back to the output image.

This method was implemented in a straightforward manner without any optimization or parallelization.

# Task 2: Parallelization with OpenMP

To improve performance, OpenMP parallelization was applied in three key steps:

1. **Parallel Histogram Calculation**: This step involves filling the histogram array based on the image pixel values. The histogram update was performed in parallel using the ‘*for*’ and ‘*collapse*’ directives as a nested loop was implemented. The directives were used safely since there were no dependencies between different iterations of the loop, and even though, the histogram array was used in the parallel portion with ‘*shared*’ scope, race conditions did not occur due to the divisions of loop iterations ensuring that same indexes of the array were not accessed by multiple threads.
2. **CDF Calculation**: The cumulative distribution function calculation was kept sequential due to its inherent dependency. Each cdf[i] depends on the previous cdf[i-1], which makes it challenging to parallelize directly. However, the computational intensity was minimal compared to the histogram calculation, so this did not significantly impact performance.

I had initially parallelized this portion of the code by passing cdf as a shared resource, and then resolving the race condition through the use of the ‘*critical’* directive, but it turned out that the overhead of using *critical* was higher than the efficiency achieved by parallelizing the loop, resulting in a higher execution time for that part than what was when it was executed sequentially. Therefore, this part of the code was not parallelized.

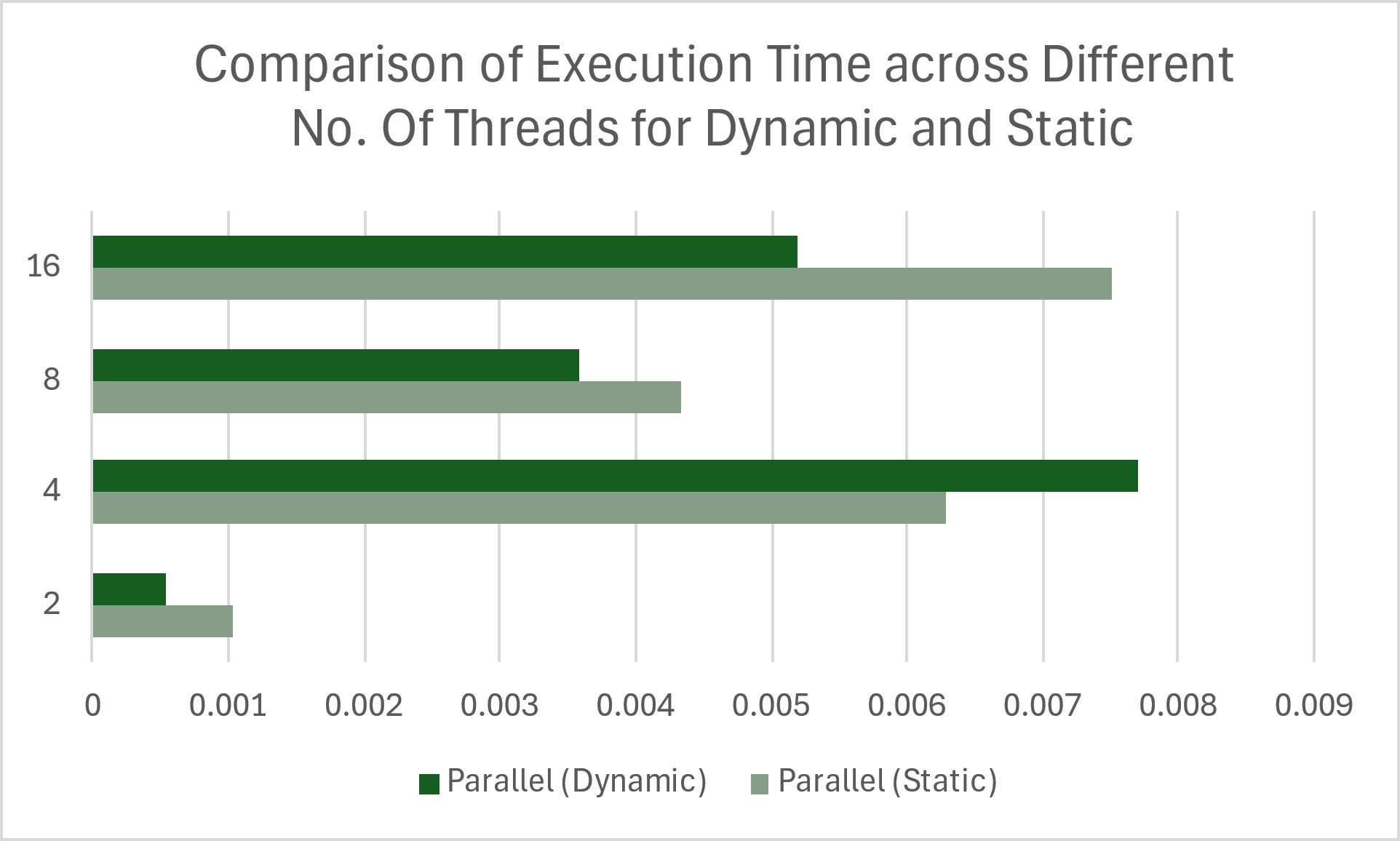
1. **Normalization and Mapping**: The normalization of the CDF values and the mapping of these values to the output image was parallelized through the use of the *parallel* and the *for* directives. The *collapse(2)* directive was used to combine the two nested loops into a single parallelized loop, further optimizing performance.
2. **Printing the Original and Equalized Image**: The original and equalized image was printed using nested loops as well, but implementing parallelism would have altered the order in which the pixel values should have been printed, therefore, the loops were executed sequentially.

## Scheduling Strategies

Two different loop scheduling strategies were implemented to compare performance:

* **Static Scheduling**: Each thread is assigned an equal chunk of iterations at the start of the execution. This approach is best suited for problems where the work is evenly distributed.
* **Dynamic Scheduling**: The iterations are dynamically assigned to threads as they finish processing their current tasks. This scheduling strategy can be beneficial if the iterations are unevenly distributed or if some threads finish their work faster than others.

The execution time for both scheduling strategies was recorded across different numbers of threads and compared. For most number of threads, static scheduling took more time due to imbalanced workloads across threads, except in one instance, leading to inconclusive results and no discernible pattern being discovered.



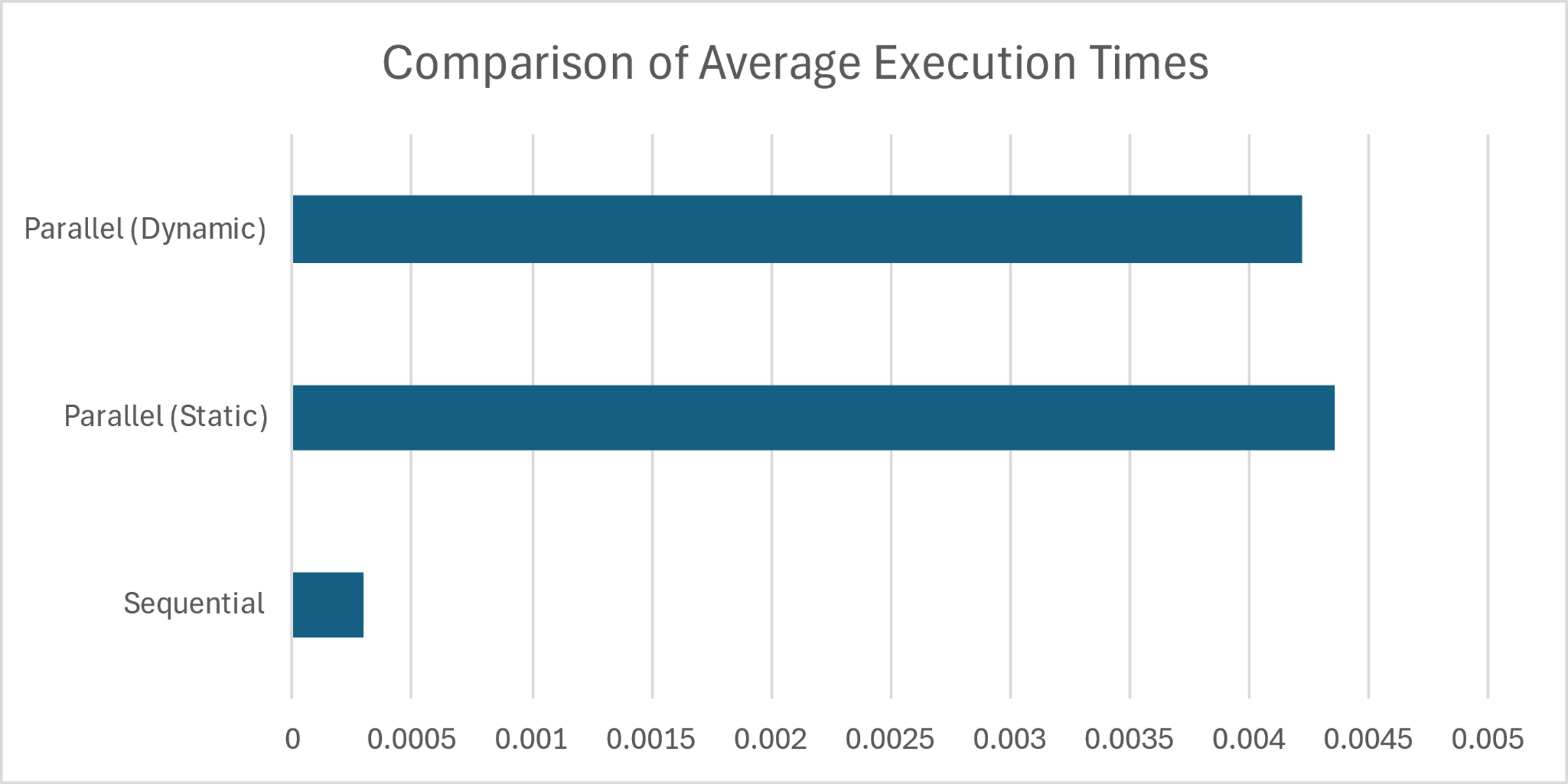
## Challenges Faced

The primary challenge was ensuring that the parallel execution time was faster than the sequential version, especially for a smaller image. In many cases, parallelization resulted in higher execution times, which was surprising given the expected benefits of parallel processing.

### Execution Time Comparison

When comparing the execution times of the sequential and parallel implementations:

* **Sequential Execution Time**: The sequential code took approximately 0.0002939 seconds to complete.
* **Parallel Execution Time**: The parallel execution time was much higher, around 0.0043583 seconds (average of both static and dynamic scheduling execution time).



The expected speedup from parallelization was not achieved primarily due to two factors:

1. **Thread Management Overhead**: For small images, managing threads is more expensive than performing the task sequentially. The parallelization overhead, including thread creation and synchronization, negated any performance gains.
2. **Small Problem Size**: The small size of the image (25x25) didn't provide enough computational work to justify the overhead of parallel execution. Parallelism tends to shine in larger datasets where each thread performs a substantial amount of work, leading to significant performance gains.

# Result

The results of the project demonstrated that the parallel implementation of histogram equalization, using OpenMP with both static and dynamic scheduling, resulted in longer execution times compared to the sequential implementation. The sequential code completed in approximately 0.0002939 seconds, while the parallel version took around 0.0043583 seconds. The increase in execution time was due to the overhead associated with thread management and synchronization. Notably, the dynamic scheduling approach provided more balanced workload distribution and produced results consistent with the sequential code, while static scheduling led to some load imbalances, which could contribute to performance inefficiencies. Despite this, for small images like the 25x25 pixel dataset used in this project, parallelism did not offer a performance improvement, highlighting the overhead involved in parallel execution for small datasets.

# Conclusion

While the parallel implementation of histogram equalization was successfully executed using OpenMP with both static and dynamic scheduling, the results were not as expected. For small images, parallel execution resulted in longer execution times compared to the sequential version due to the overhead of thread management and synchronization.

For larger images, parallelization would likely yield a significant performance improvement. However, for small datasets, the parallel approach should be used cautiously, as the overhead may outweigh the benefits. This assignment highlighted the importance of balancing parallelism and understanding the problem size when deciding whether to parallelize a task.