**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** | 23th 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

### Comparison Between Sequential, Static and Dynamic Execution:

Three versions of the code were compared based on their execution times for the same number of threads (12 threads):

|  |  |  |  |
| --- | --- | --- | --- |
| **Execution Time for Sequential And Parallel** | | | |
|  | **Sequential** | **Static** | **Dynamic** |
|  | 2.684503 | 0.782838 | 1.993353 |
|  | 4.974233 | 0.769057 | 0.902408 |
|  | 2.182456 | 0.809353 | 1.196917 |
|  | 2.069434 | 1.046829 | 1.25453 |
|  | 2.614927 | 0.757473 | 1.3379 |
|  | 2.518812 | 0.759928 | 1.235415 |
|  | 3.169371 | 0.979525 | 1.848767 |
|  | 2.062875 | 0.806165 | 1.265136 |
|  | 3.148981 | 0.853681 | 0.813674 |
|  | 2.44196 | 0.774685 | 0.813777 |
| **Average** | **2.7867552** | **0.8339534** | **1.2661877** |

**Analysis:**

Static scheduling provides the greatest reduction in execution time. On average, it cuts down the sequential time (2.79s) to roughly **0.83s**, which is a significant speedup. Dynamic scheduling still improves over sequential, but not as effectively as static. Its average (1.27s) sits between sequential and static.

This is becausethe workload is split evenly among threads at the start in static scheduling, and this tends to minimize overhead and thread idle time for tasks with uniform workload. The reason why dynamic scheduling did not perform as well as static was due to the overhead caused during dynamic scheduling and because the workload assigned was uniform, meaning the overhead was more than the efficiency provided by dynamic scheduling.

The sequential times vary more widely, possibly due to varying system load or caching effects. However, even the slowest parallel static runs are still much faster than most sequential runs.

### Comparison Between Different Numbers of Threads:

A table with numbers and lines

AI-generated content may be incorrect.

|  |  |  |  |
| --- | --- | --- | --- |
| **No. of Threads** | **4** | **8** | **12** |
| **Dynamic** | 1.0681469 | 0.8762315 | 1.0186433 |
| **Static** | 1.1091052 | 0.6879895 | 0.6402083 |

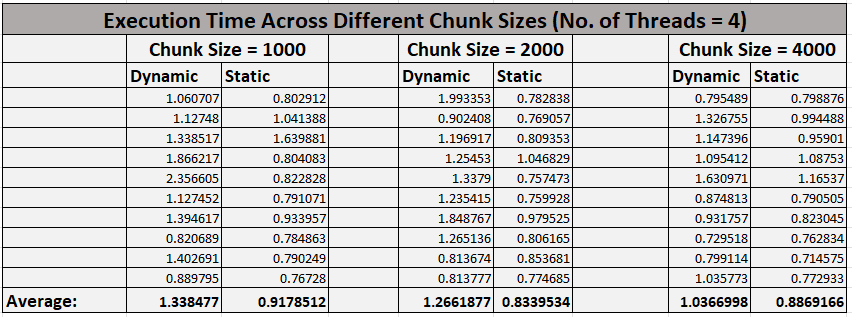
Table 1 Average Execution Times for Different No. Of Threads

**Analysis:**

With dynamic, increasing threads from 4 to 8 improves performance, but adding more (up to 12) actually makes it a bit slower — probably because of the extra overhead of managing all those threads. For static, it just keeps getting better as you increase threads since the work is split evenly and there's less back-and-forth.

This shows that using too few threads doesn't give you much parallel benefit, but using too many, especially with dynamic scheduling, adds extra overhead (like threads waiting around or constantly being reassigned tasks). Static avoids that by just dividing the work up once and letting each thread handle its chunk.

### Comparison Between Different Chunk Sizes:



|  |  |  |  |
| --- | --- | --- | --- |
| **No. of Threads** | **4** | **8** | **12** |
| **Dynamic** | 1.0681469 | 0.8762315 | 1.0186433 |
| **Static** | 1.1091052 | 0.6879895 | 0.6402083 |

Table 2 Average Execution Times For Different Number of threads

**Analysis:**

* Static scheduling consistently delivered better performance across all chunk sizes tested. The most optimal result was seen with a chunk size of 2000, where the average execution time dropped to approximately 0.83 seconds, outperforming both smaller and larger chunk sizes within the same scheduling strategy.
* Dynamic scheduling showed noticeable improvement as chunk size increased. Starting with an average of around 1.34 seconds at a chunk size of 1000, it improved to about 1.04 seconds at 4000. However, even at its best, dynamic scheduling was still slower than static, likely due to the added overhead of managing task redistribution between threads.
* The performance advantage of static scheduling can be attributed to its low-overhead, one-time division of work, which is especially effective when the workload is evenly distributed. In contrast, dynamic scheduling introduces extra management cost that doesn’t always pay off unless there is significant imbalance in workload distribution.

Overall, the data suggests that static scheduling with moderate-to-large chunk sizes (around 2000 or more) is the most effective approach for achieving faster execution times in scenarios where the workload is uniform and predictable.

# 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(Enhanced)) + (F(Enhanced)/Speedup(Enhanced))**

Calculated for 12 threads and static scheduling:

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

Speedup of enhanced portion = (Execution time of sequential) / (Execution time of parallel) = 5.895522388059701

**Speedup = 2.24**

# Task 5: Conclusion

The experimentation with histogram equalization using both sequential and parallel approaches revealed some important insights about performance under different parallelization strategies.

* Static scheduling consistently outperformed both dynamic and sequential approaches across all tests. It delivered the lowest average execution times in nearly every scenario, thanks to its low-overhead, evenly divided workload distribution — making it highly suitable for tasks like histogram equalization where the processing is uniform across pixels.
* The number of threads had a noticeable impact on performance, but only to a certain extent. For static scheduling, increasing threads from 4 to 12 led to steady improvements, with 12 threads achieving the best results. In contrast, dynamic scheduling performed best at 8 threads, but performance declined slightly at 12 due to increased task management overhead. This highlights that while more threads can speed things up, excessive threading may introduce inefficiencies, especially with dynamic load balancing.
* Chunk size played a key role in performance, particularly for dynamic scheduling. Larger chunks reduced the overhead in dynamic mode, improving its speed from 1.34s (chunk size 1000) to 1.04s (chunk size 4000). However, even with larger chunks, dynamic couldn’t outperform static. Static scheduling performed consistently well across all chunk sizes, with the best average time achieved at a chunk size of 2000.

Combining all observations, the most optimal configuration for this specific task is static scheduling with a thread count of 12 and a chunk size of around 2000. This setup minimizes execution time by maximizing core utilization while keeping scheduling overhead low — making it ideal for parallel histogram equalization of uniformly structured data.