Matriculation Number: 40646460

Coursework 2

SET10108: Concurrent & Parallel Systems

Contents

[Chapter 1: Introduction 2](#_Toc183964382)

[1.1 Overview of Task 2](#_Toc183964383)

[1.2 Hardware Specifications 2](#_Toc183964384)

[Chapter 2: Analysis 3](#_Toc183964385)

[2.1 Performance Issues Identified 3](#_Toc183964386)

[2.1.1 User Interface Blocking 3](#_Toc183964387)

[2.1.2 Sequential Colour Temperature Calculation 3](#_Toc183964388)

[2.1.3 Inefficient Sorting Algorithm 3](#_Toc183964389)

[2.2 Parallelisation and Concurrency Opportunities 3](#_Toc183964390)

[2.2.1 Running the Sorting Function Concurrently 3](#_Toc183964391)

[2.2.2 Parallelising Colour Temperature Calculations 4](#_Toc183964392)

[2.2.3 Parallelising the Sorting Algorithm 4](#_Toc183964393)

[2.3 Identification of Shared Resources 4](#_Toc183964394)

[Chapter 3: Methodology 5](#_Toc183964395)

[3.1 Introducing Task Parallelism 5](#_Toc183964396)

[3.1.1 Background Sorting 5](#_Toc183964397)

[3.1.2 Justification 5](#_Toc183964398)

[3.2 Introducing Data Parallelism 5](#_Toc183964399)

[3.2.1 Parallelising Colour Temperature Calculation 5](#_Toc183964400)

[3.2.2 Justification 6](#_Toc183964401)

[3.3 Parallelising the Sorting Algorithm 6](#_Toc183964402)

[3.3.1 Parallel Sort Implementation 6](#_Toc183964403)

[3.3.2 Justification 7](#_Toc183964404)

[Chapter 4: Results 8](#_Toc183964405)

[4.1 Performance Analysis(See appendix 1, 2, and 3, for full performance measurements) 8](#_Toc183964406)

[4.2 Quantitative Analysis 9](#_Toc183964407)

[4.3 Qualitative Analysis 9](#_Toc183964408)

[4.4 Final Discussion 9](#_Toc183964409)

[Appendices 10](#_Toc183964410)

[Appendix 1: Testing Performances (1) 10](#_Toc183964411)

[Appendix 2: Testing Performances (2) 10](#_Toc183964412)

[Appendix 3 Testing Performances (3) 11](#_Toc183964413)

[Appendix 4: Excel Spreadsheet Data 11](#_Toc183964414)

[Appendix 5: Graph – Average Runtime (Bar Chart) 12](#_Toc183964415)

[Appendix 6: Graph – Relative Improvement (line chart) 13](#_Toc183964416)

# Chapter 1: Introduction

## Overview of Task

This report focuses on optimising an image viewer application which sorts images based on their colour temperature, and displays them in order. The objective is to document improvements that resolve user interface (UI) blocking, accelerate processing through parallelism, and enhance user experience.

## 1.2 Hardware Specifications

Testing was performed on a system with:

GPU: NVIDIA GeForce RTX 3060 Laptop GPU (6 GB VRAM, 30 multiprocessors)

CPU: AMD Ryzen 7 5800H (8 cores, 16 threads)

Performance may vary on different hardware.

# Chapter 2: Analysis

## 2.1 Performance Issues Identified

### 2.1.1 User Interface Blocking

The main performance issue was UI blocking during the sorting process. Since the sorting function was executed on the main thread, it monopolised the central processing unit (CPU), preventing the UI from updating, and freezing the application. This made it impossible for users to interact with the application (e.g., navigating images) until the sorting task was complete. This delay was especially problematic for large datasets, where sorting took considerable time.

### 2.1.2 Sequential Colour Temperature Calculation

Colour temperature calculations were performed sequentially, processing each pixel one at a time. This approach limited the use of multi-core processors, as only one core was utilised for the entire process. As a result, the application suffered from significant performance degradation on large datasets. This bottleneck became particularly noticeable when dealing with high-resolution images or large image directories.

### 2.1.3 Inefficient Sorting Algorithm

The sorting process used std::sort, a sequential algorithm that sorts elements one by one on a single thread. The sorting task was computationally intensive, and running it sequentially on a single core delayed the overall execution time. This sorting operation is disproportionately slow, contributing significantly to the sluggish performance of the application.

## 2.2 Parallelisation and Concurrency Opportunities

2.2.1 Running the Sorting Function Concurrently

One option to address UI blocking is to move sorting to a background thread (e.g., multithreading), thus ensuring the UI remains responsive. Alternatively, using a thread pool would distribute the sorting task across multiple cores, thereby improving performance. Both approaches require managing shared resources, like the image filename vector. A mutex or read-write lock can protect the vector during access to avoid race conditions. Lock-free data structures are an alternative but more complex option.

2.2.2 Parallelising Colour Temperature Calculations

Colour temperature calculations are naturally parallel, as each pixel is independent. Single Instruction Multiple Data (SIMD) framework allows processing multiple pixels simultaneously but requires hardware-specific optimisations, making it less portable. OpenMP, however, is more flexible and easier to implement, distributing the task across multiple threads and scaling with the number of CPU cores. OpenMP simplifies parallelisation with minimal code changes, making it a practical solution for this task.

2.2.3 Parallelising the Sorting Algorithm  
Parallelising the sorting process can be achieved using a divide-and-conquer approach, where the dataset is split into subarrays, sorted independently, and merged in parallel. OpenMP is ideal for managing parallel threads during both sorting and merging. Alternatively, task parallelism with a thread pool could allow more dynamic load balancing, but with added complexity. OpenMP offers a balance between performance, ease of use, and scalability across multi-core systems.

## 2.3 Identification of Shared Resources

Shared resources included the image filename vector and pixel temperature data. Access to the vector by both the main thread (UI updates) and the sorting thread required synchronisation to prevent race conditions. A mutex was used to protect the vector, thus ensuring safe access. For temperature data, OpenMP’s management of thread workloads prevented conflicts, as each thread worked on distinct data sections.

# Chapter 3: Methodology

## 3.1 Introducing Task Parallelism

### 3.1.1 Background Sorting

To address UI blocking, the sorting operation was offloaded to a background thread using std::thread. This allowed the main thread to remain responsive to user inputs (e.g., image navigation while the sorting task ran concurrently). By moving the sorting to a separate thread, the UI thread was freed from waiting for the completion of the sorting process, thereby improving user experience. A critical challenge was ensuring that the shared vector of image filenames, accessed by both the UI thread (for updates) and the sorting thread (for modification), was protected. To prevent race conditions, where multiple threads could read and write to the vector simultaneously, a std::mutex was employed. This mutex locked the vector during write operations, ensuring that only one thread could access it at any given time, thus preventing data corruption. The mutex was strategically applied around the sorting code, guaranteeing thread safety while limiting performance overhead to the minimum requirement for synchronisation.

### 3.1.2 Justification

The decision to use std::thread was based on its simplicity, low overhead, and compatibility across different platforms. By using the C++ Standard Library's thread management system, the background thread could be easily integrated into the existing application. std::thread provided an efficient means of offloading the sorting task without needing a complex threading framework. Additionally, sorting is a CPU-intensive operation, and offloading it to a single background thread was deemed sufficient for this task, as the sorting itself was the bottleneck. Further splitting the sorting task into multiple threads was not necessary, as it would have added extra complexity with minimal performance gains. Meanwhile, std::thread offered an optimal balance of simplicity and functionality for the task at hand, making it an ideal choice.

## 3.2 Introducing Data Parallelism

### 3.2.1 Parallelising Colour Temperature Calculation

The colour temperature calculation, which was the most computationally expensive part of the application, was parallelised using OpenMP. Since the calculation for each pixel is independent, this provided a perfect opportunity for parallel execution. OpenMP's #pragma omp parallel for directive was used to divide the loop processing the image's pixels into smaller chunks, each of which was handled by a different thread. This ensured that multiple pixels were processed simultaneously across multiple cores, which significantly reduced execution time. The parallelised loop allowed each thread to calculate the colour temperature for its assigned set of pixels, without any need for complex thread synchronisation between iterations, as each iteration was independent. This approach made full use of multi-core CPU capabilities, ensuring that each core was actively engaged in performing computations, rather than leaving them idle.

3.2.2 Justification

OpenMP was selected for parallelising the colour temperature calculation due to its ease of use and low implementation overhead. With minimal changes to the existing code, OpenMP was able to manage the creation, distribution, and management of threads automatically. The #pragma omp parallel for directive made it easy to divide the workload, ensuring that each chunk of pixels was processed in parallel. OpenMP's scalability was a major factor in its selection: it allowed the parallelisation to scale with the number of CPU cores, in order that performance would improve with more powerful hardware. Furthermore, OpenMP’s inherent portability meant the parallelised code would run efficiently across a variety of system architectures, without needing low-level hardware-specific optimisations (e.g., those required for SIMD). Given its simplicity, scalability, and cross-platform capabilities, OpenMP was the most appropriate choice for parallelising this data-intensive task.

## 3.3 Parallelising the Sorting Algorithm

### 3.3.1 Parallel Sort Implementation

To parallelise the sorting operation, the dataset of image filenames (and their corresponding colour temperature values) was divided into smaller chunks. Each chunk was sorted independently using std::sort within separate OpenMP threads. The choice to use std::sort was based on its highly optimised sequential implementation, which ensured fast individual sorting operations. Once all chunks were sorted in parallel, they were merged together to form the fully sorted dataset. This merging operation was also parallelised, further reducing execution time. By breaking down the sorting task into smaller, manageable chunks, and processing them in parallel, the algorithm was able to make full use of the multi-core CPU, thus improving performance significantly, especially with larger datasets. OpenMP’s thread management system handled the distribution of chunks to threads, thus automatically balancing the workload across available cores.

3.3.2 Justification

OpenMP was chosen for parallelising the sorting algorithm due to its simplicity in managing multi-threaded operations. The divide-and-conquer approach used for sorting allowed the task to be split into independent subarrays, which could be sorted concurrently. The final step (merging these subarrays into a single sorted array) was also done in parallel, thereby further enhancing performance. OpenMP’s ease of integration into the existing codebase and its support for thread synchronisation (e.g., managing the merge process without race conditions) made it an ideal choice for this task. OpenMP’s ability to handle thread distribution and load balancing automatically, allowed for the efficient use of system resources without requiring complex manual thread management. While more specialised parallel sorting algorithms (like parallel merge sort) could offer marginally better performance, OpenMP provided a good balance between performance improvement and ease of implementation. The ability to scale the parallel sorting operation based on the number of CPU cores available, ensured that the system could handle larger datasets more efficiently, while also keeping the implementation relatively simple.

# Chapter 4: Results

## 4.1 Performance Analysis (cf. [Appendix 1](#_Appendix_1:_Testing)-[4](#_Appendix_4:_Excel), for full performance measurements)

The performance results were measured based on the average execution times and relative improvement percentages, calculated as follows:

1. Average Time Calculation

The average execution time for each optimisation approach was determined by recording the results of five consecutive runs (from the second run onward), then calculating the mean:

This approach excluded the first run, to minimise the influence of initialisation overheads and warm-up effects on the results.

1. Relative Improvement Calculation

The relative improvement percentage was calculated to assess the performance gain over the unoptimised version:

-------------------------------------------------

Results:

1. Original Application (Unoptimised)

* Average Time:**60.26 seconds**
* Relative Improvement: **0%**
* The initial version was slow and blocked the UI during the sorting process.

1. Sorting Moved to a Separate Thread

* Average Time:**59.54 seconds**
* Relative Improvement: **1.20%**
* The UI no longer froze during sorting, but the performance remained nearly identical to the unoptimised version.

1. Colour Temperature Calculation Parallelised Using OpenMP

* Average Time:**4.37 seconds**
* Relative Improvement: **92.75%**
* Parallelising the colour temperature calculation showed a huge speedup. This step dramatically reduced the sorting time, likely because the colour temperature calculation was a bottleneck, and because using OpenMP allowed for concurrent computation.

1. Parallel Sorting Using OpenMP (Mutex Implementation)

* Average Time:**2.99 seconds**
* Relative Improvement: **95.04%**
* This step improved sorting by parallelising the median calculation as well. The sorting process itself was parallelised with mutex synchronisation, to prevent race conditions when updating the list of image filenames.

1. Manually Setting Thread Count for Sorting

* Thread Number = 2: Average Time:**2.72 seconds**
  + Relative Improvement: **95.48%**
* Thread Number = 4: Average Time:**2.71 seconds**
  + Relative Improvement: **95.51%**
* Thread Number = 6: Average Time:**2.71 seconds**
  + Relative Improvement: **95.51%**
* Thread Number = 8: Average Time:**2.66 seconds** *(most optimal)*
  + Relative Improvement: **95.60%**
* Thread Number = 10: Average Time:**2.68 seconds**
  + Relative Improvement: **95.55%**
* Thread Number = 12: Average Time:**2.68 seconds**
  + Relative Improvement: **95.55%**
* Thread Number = 14: Average Time:**2.69 seconds**
  + Relative Improvement: **95.54%**
* Thread Number = 16: Average Time:**2.70 seconds**
  + Relative Improvement: **95.52%**

## 4.2 Analysing Trends in the Data

The table/graphs summarise the average execution time ([cf. Appendix 5](#_Appendix_5:_Graph)) and relative improvement percentages ([cf. Appendix 6](#_Appendix_6:_Graph)) for various approaches to optimising the original program. The original, unoptimised application exhibited a high average execution time of **60.26** **seconds**.

* Sorting on Worker Thread: Moving sorting to a worker thread reduced the average time to **59.54 seconds**, representing a minimal improvement of **1.20%**. While this ensured the UI was no longer blocked, it did not significantly improve performance.
* Colour Temperature Calculation Parallelised: Parallelising the bottlenecked colour temperature calculation with OpenMP resulted in a significant reduction to **4.37 seconds**, achieving a relative improvement of **92.75%**.
* Parallelised Sorting: Incorporating OpenMP to parallelise both sorting and colour temperature calculations reduced the time further to **2.99 seconds,** yielding an improvement of **95.04%.**
* Thread Count Adjustments: Experimenting with the number of threads showed marginal gains at higher thread counts. The lowest time of **2.66 seconds** was achieved with **8 threads**, representing a relative improvement of **95.60%**. Further increases in thread count showed diminishing returns.

## 4.3 Interpreting the Data

Strengths of the Optimisation Strategies

1. Substantial Performance Gains:

Parallelisation using OpenMP achieved significant reductions in execution time, particularly in the colour temperature calculation, which was identified as the primary bottleneck.

1. Enhanced User Experience:

Moving sorting to a worker thread, while yielding minimal computational benefits, significantly improved the responsiveness of the application by preventing UI freezes.

1. Scalability and Adaptability:

The introduction of multithreading allowed for scalability across systems with different hardware configurations, while the flexibility to adjust thread counts facilitated fine-tuning for optimal performance.

Limitations and Challenges

1. Diminishing Returns with High Thread Counts:

Beyond 8 threads, the performance gains plateaued due to increased overhead from thread synchronisation and resource contention.

1. Hardware Dependence:

The observed performance improvements were closely tied to the underlying hardware. Systems with fewer cores or limited parallel processing capabilities would likely experience smaller performance gains.

## 4.4 Final Discussion

The optimisation strategies adopted in this report demonstrate the transformative potential of parallelisation in enhancing application performance. The substantial improvements in execution time (up to **95.60%**) validate the efficacy of targeting bottlenecks and leveraging multithreading.

Despite these successes, the analysis highlights key limitations, including diminishing returns at higher thread counts, and dependency on specific hardware configurations. Moving forward, integrating dynamic thread management techniques could address these limitations, thus enabling the application to adapt to varying workloads and environments.

In conclusion, this report underscores the importance of balancing computational efficiency, scalability, and user experience when optimising software applications.

# Appendices

## A table of information with text Description automatically generated with medium confidenceAppendix 1: Testing Performances (1)

## A screenshot of a computer Description automatically generatedAppendix 2: Testing Performances (2)

## A table with text and numbers Description automatically generated with medium confidenceAppendix 3 Testing Performances (3)

## A screenshot of a computer Description automatically generatedAppendix 4: Excel Spreadsheet Data

## A graph with blue and white text Description automatically generatedAppendix 5: Graph – Average Runtime (Bar Chart)

## Appendix 6: Graph – Relative Improvement (Line Chart)

A graph of a bar graph

Description automatically generated