Concurrent and PARALLEL SYSTEMS COURSEWORK 1

Software Engineering - SET10108

Edinburgh Napier University

40618059

|  |
| --- |
| **Declaration**  I declare, except where explicit reference is made to the contribution of others\*, that this assignment is the result of my own work and has not been submitted for any module or programme degree at the Edinburgh Napier University or any other institution. This is in accordance with Edinburgh Napier University’s Academic Integrity Regulations.  \*IMPORTANT: Contribution of others may include use of Artificial Intelligence (AI) tools (details of which can be found in the Guidelines for Students on AI & Writing Assistant Tools). Please declare here whether you have used such tools, and to what extent:  ☐NO I have not used such tools  ☐YES I have used such tools and I have provided details below and included sample prompts and responses in an appendix.  If you answered YES here, please, in around 100 words, describe how (and at which points) you have used such tools to support your completion of this assessment |

[1. Analysis 3](#_Toc149745892)

[2. Methodology 3](#_Toc149745893)

[3.1 UI 3](#_Toc149745894)

[3.2 Image Loading 4](#_Toc149745895)

[3.3 Calculate Median 4](#_Toc149745896)

[3.4 Sorting 5](#_Toc149745897)

[3.5 Performance Measuring 5](#_Toc149745898)

[3. Results 5](#_Toc149745899)

[3.1 Hardware 5](#_Toc149745900)

[3.2 Optimisations 5](#_Toc149745901)

[Load Images Average: 71.50% increase in Parallelization. 6](#_Toc149745902)

[Calculate Median Hue: 56.7% increase in Parallelization. 6](#_Toc149745903)

[Sort Images By Hue: 72.20% increase in Parallelization. 6](#_Toc149745904)

[4. Conclusion 7](#_Toc149745905)

[5. Appendix 8](#_Toc149745906)

# Analysis

The following functions of the application for loading images and sorting by hue colour can be parallelised for better performance than their sequential alternative:

* Image loading: This function can be made parallel using threads utilising stbi::load. This means rather than one image at a time being loaded, all images can be spread amongst different threads taking in their data all at once. Since this data will include texture data and more, mutex will also be used to stop memory issues from happening.
* Calculating Median Hue: This is a mathematical function that will require most of the system performance regardless of being sequential or parallel. It will, again through threads, take all images and get their median hue through a HSV conversion method using the method of y = frac(x+1/6).
* Image Sorting: Again, rather than adding one image at a time to the list, it would be much faster utilising the std::sort method in order to sort all images at once.
* User Interface: The user interface currently resides inside the main thread of the codebase. This should be rectified straight away with it being moved to its own thread using an async call so that the UI runs in sync with the rest of the program and remains responsive for the user to have a better experience with the application. If left in the main thread, the program would run slower, the UI would not load until the rest of the programs function have completed and would appear as if it was unresponsive or has system lag.

# Methodology

The development of this C++ application used SFML and standard C++ libraries in order to utilise threads and take advantage of the performance enhancements that these libraries provide. Another library such as OpenMP is an alternative to using SFML and threads however requires a level of experience that has not been met yet for the purposes of this project.

## UI

When it comes to the UI, the first modification made was removing the code from the main thread, outside to its own thread. This used std::async in order to make sure the UI is synchronized with the rest of the program at all times. This change makes it so that the UI boots up as soon as the application is launched, otherwise it would rely on all over functions finishing their tasks before the UI boots up. This change allows the user to at the very least understand the application is responsive and running as intended. To accommodate this change and ensure the application does not become unresponsive when the user attempts to press the navigation keys, a std::future call is put into place.

## Image Loading

The UI will then lead into Image loading, this begins by calling for a thread called t\_lImage to get the images data assigned to a vector called imageDataList that will be used throughout the code, then passing that onto thread(s) called getImgData. This will utilise stbi\_load (obtains height, width and channels which may be RGB or RGBA) and stores it into an Image object containing the image filepath and data. As this is dealing with many images and their data that are all being push backed into the imgData object, a mutex lock guard is put into place in order to ensure that the threads don’t conflict when adding imgData. After this, stbi\_image\_free(imgData) is called to refresh memory. Once the threadpool has reached its end, the thread is joined, and image loading is complete. A lock guard is also put into place so that ImageDataList can be accessed with thread protection on concurrent access to the list.

## Calculate Median

After the images are loaded, the next step is calculating the median hue for each of the images stored in imgData. This is done by once again using a stbi async function to send all of the images to send multiple images at the same time to their own threads for calculation. The calculation takes place in a separate C++ file called ImageCalc, it takes the RGB values, converts to HSV and gets the median of the image and sends it back. The file and median hue are assigned to a vector that is a future type used to then add the images and their median hue to a list called SortedImages.

The code snippet is called caclulateMedianHue, and it will extract the median hue from the image data. The section will do the following:

* Input validation: Verifies that the provided image data is valid and not empty, otherwise the image has a default value set to 0.0.
* Conversion: The RGB values are taken and normalized to a range of 0,1 by taking each channel and dividing by 255.
* Hue Calculation: Red, Green, and Blue values will be taken, identifying the channel that has the max value, the min value then gets the difference between the two.
* Using the values found, determines which colour channel is the most dominant and calculate the hue value.
* The median hue is passed to a function called mapHue that will verify the hue is in the expected range.
* The mapped value is returned and normalised to a range of 0,1.

The value is then passed back to the main and used for sorting the image list for output.

## Sorting

Sorting is done simply by using another std::async function to send all images and their median hue from SortedImages and performs an std::sort.

## Performance Measuring

For the three major functions of Loading Images, Calculating Median Hue and then Sorting by Median Hue, there are performance markers for each. Firstly, there is a function called OutputPerf. For each of the functions, right before the process starts, a timer will begin named, after the function has finished, the timer will end immediately. The time it took will be calculated and then the process OutputPerf function will call, and the information will be written to a csv. For example, after loading images it will write “Name of process: Load Images” and in the next cell the time it took. This is a more optimal way rather than writing cout lines that invoke the command console possibly affecting the processing time, and moreover for a standalone exe it will be better writing to a csv rather than having a command box coming up every time it runs.

# Results

## Hardware

Before discussing results, hardware specs should be detailed:

* Custom Built Desktop PC:
  + Ryzen 5 3600 3.6 GHz / 4.2 GHz (Turbo Core) 6 cores / 12 threads.
  + RAM – 16GB DDR4 2600Mhz
  + RTX 4060ti GPU (8GB)
  + 1Tb Seagate SSD (2gbs)
  + B450-A Pro Max Motherboard.

## Optimisations

When it came to optimising the code as much as possible, it was a main focus from the start. Making sure that every line of code is done efficiently, and nothing is redundant or repeated was at high importance. Originally, loading in data would be done at the same time as calling performing the median calculation. Instead, simply obtaining that data while the images have been loaded gave a level of optimisation required.

Originally, the code was created in a sequential format. After finishing the build, 10 runs were performed, and performance was captured. Then, each major function was optimized to run in parallel. Then 10 runs were performed for that with the performance also captured. The average time for each function both optimized and unoptimized were then used to calculate the percentage increase in speed for each function to complete, then an overall speed increase was captured.

For each function, the average over 10 runs for both versions of the code were captured. Below, each result will be shown.

### Load Images Average: 71.50% increase in Parallelization.

#### Figure 1: Load Images Average

### Calculate Median Hue: 56.7% increase in Parallelization.

#### Figure 2: Calculate Median Hue Average

### Sort Images By Hue: 72.20% increase in Parallelization.

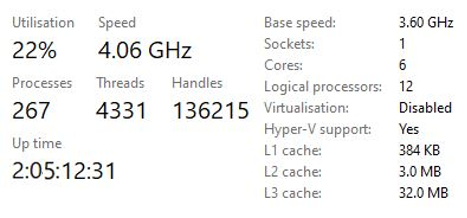
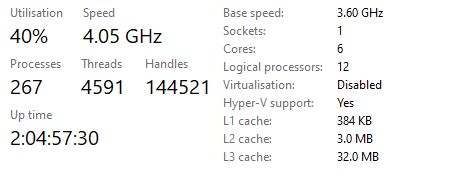
#### Figure 3: Sort Images By Hue Average

The overall speed increase for parallelization: 110.31%. This is a definitive result showing that the system overall runs faster, therefor more effectively, with threads being utilized all over the codebase. The speed is over doubled when the code is run in parallel opposed to sequential. Image loading and hue calculations benefitted the most from the change, as seen from the overall averages. Another performance measure that was captured is CPU utilization while the program is running:

Parallel

Serialized





#### Figure 4: CPU Utilisation

As seen from the task manager screenshot at time of execution, serialized version of the code did not utilize the hardware nearly as much as it’s parallel counterpart. More threads are also used, of course, in the parallel version making the project run smoother and faster.

# Conclusion

Looking back at the analysis, the aim of this project was to create an image processing program that would sort the images by hue using parallelization on the major functions. Now, at the end of the project I can say that I have achieved this goal. All 3 major functions showed improvement when parallel as seen from the results (more results found in the appendix).

This is not to say that this is the most well optimised the code could ever be, other libraries outside of SFML could offer better tools such an OpenMP. However, the main goal was still achieved and at 10 runs for serialization and then parallel functions to obtain a reliable average, obtaining over a 100% speed overall system performance increase, that optimisation goal has been met.

# Appendix

Figure 5: Sequential Runs

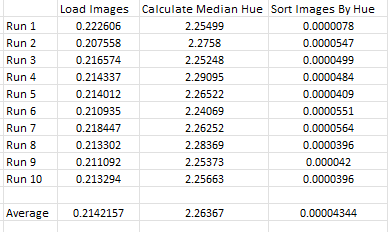


Figure 6: Sequential Run Data

Figure 7: Parallel Runs

A table with numbers and symbols

Description automatically generated

Figure 8: Parallel Run Data