# Lab 3
# Thread-level parallelism: Parallelization via OpenMP and asynchronous scheduling of digital photo processing

**Group Members:** George Hope, Naima Ontiveros, Alicja Wagner, Songul Yasin

**The individual tasks (0 and 1) can be found in separate documents.**

# Parallelization of digital photo development

## Task 2.1: Analyze the code and identify the different processes

This C++ code is an image processing pipeline that takes a raw image file as input, performs various image processing operations, and saves the final result as an output image file. The pipeline includes operations such as debayering, denoising, gamma correction, color balance correction, equalization, enhancing details, bloom effect, and a screen mode merge. The code uses the LibRaw library for working with raw image files and the OpenCV library for image processing.

The code includes several C++ and library headers, such as “<stdio.h>”, “<string.h>”, “<math.h>”, “<iostream>”,” <chrono>”, LibRaw (libraw/libraw.h), and OpenCV.

The “std” and “std::chrono” namespaces are declared for using standard C++ and time-related functionalities. A macro „SQR(x)“ is defined for squaring a value.

Prototypes for various image processing functions are declared, including colorBalance, gammaCorrection, sharpening, enhanceDetails, bloom, denoise, equalization, debayer, and screenMerge. The gammaCurve function generates a gamma correction curve, which is a lookup table used in gamma correction.

The following different processes are applied at the main functions to the RAW image

The *debayer* function converts the RAW image into a color image by applying the Debayering algorithm. The function has no dependency.

The *denoise* function is used for noise reduction, It applies median filtering to reduce noise in the YCrCb color space. The function has a dependency on Debayer.

The *gammaCorrection* function adjusts the gamma value of the image to convert it from a linear to a non-linear representation. The function has a dependency on denoise.

The *colorBalance* function balances the color of the image based on percentiles and is dependent on gamma correction.

The *equalization* function performs histogram equalization. It equalizes luminance values and increases saturation and has a dependency on color balance.

The *enhanceDetails* function enhances high-frequency details in an image by subtracting a blurred version and amplifying the differences. the function has a dependency on equalization

The *bloom* function generates a bloom effect by thresholding, blurring, and combining with the original image and has a dependency on equalization.

The *screenMerge* function combines enhanced details with the bloom mask using screen mode merge. It has dependencies to enhanceDetails and Bloom.

The *main* function reads a RAW image, processes it through the pipeline of functions, and saves the final image and has dependencies to all the above processes.

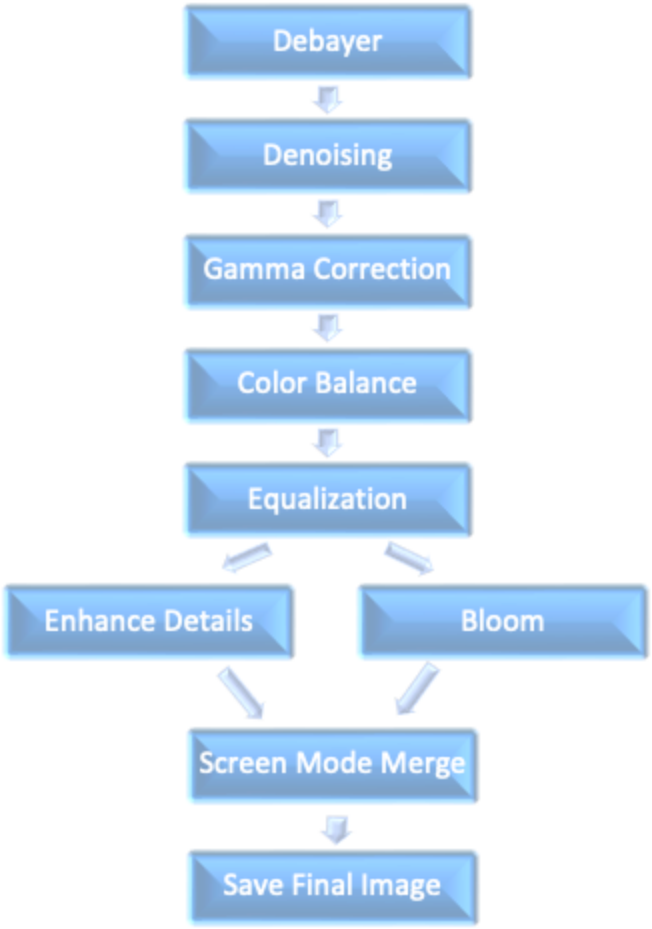



**Figure 1 - Dependency diagram**

Parallelization Opportunities:

Since it doesn't depend on any other process, debayering can be parallelized.After parallelized debayering the denoising can be parallelized.The gamma correction depends on the denoising process, so after parallelizing this the gamma correction can be parallelized. After this the color Balance can be parallelized. The Equalization can be parallelized after color balance. After parallelizing equalization the Enhance Details and bloom can be parallelized. The functions bloom and enhance details don't depend on each other, so they can be done in parallel. The Screen Mode Merge requires results from both Enhance Details and bloom, so these need to be completed before starting screen mode merge. Saving Final Image can be done after the entire image processing pipeline is complete. Suggested Parallelization Strategy is to use OpenMP directives to parallelize the independent processes (debayering, denoising, gamma correction, color balance, equalization, enhance details, bloom) at the thread level.

## Task 2.2: Analyze the code of each process

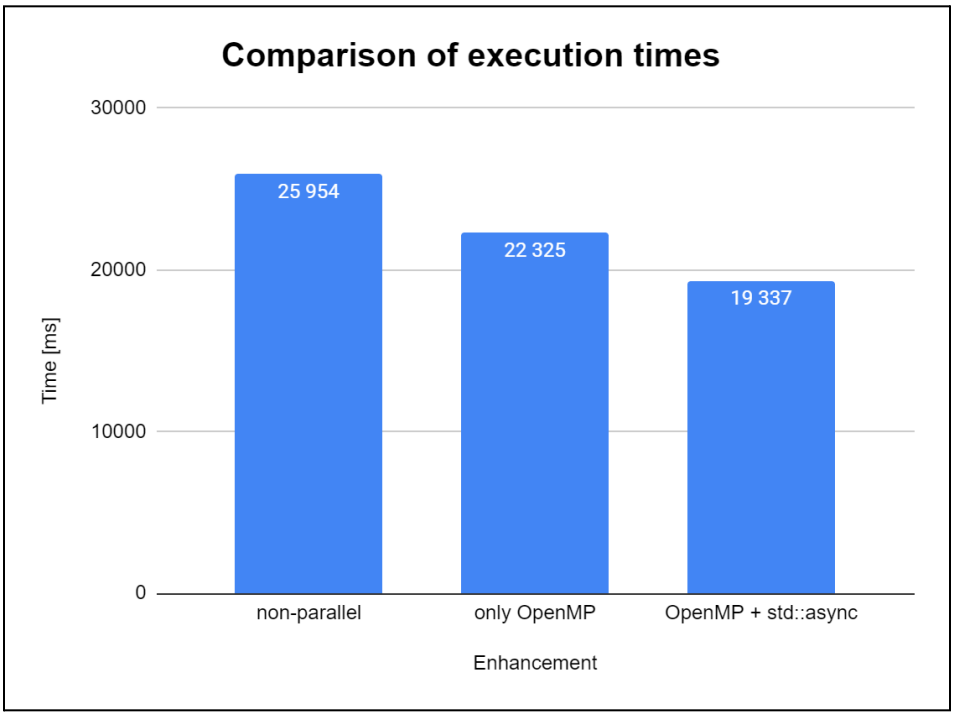
According to the teacher's instructions, this task does not have to be completed. The flow diagram is not needed.

## Task 2.3 Parallelization

### Parallelization of the code

The parallelized code can be found in the main.cc file.

To parallelize the code I used two concurrency mechanisms: “OpenMP” for parallelism and “std::async” for asynchrony. The best results, when it comes to the program execution time, were obtained using both of them, which can be seen in the bar graph below.




From the OpenMP library I chose

*#pragma omp parallel for.*

This pragma is used to parallelize for loops which can significantly improve the performance of the application. However, in certain functions adding this pragma didn’t minimize the execution time.

1. The “gammaCurve” function without any enhancement takes only 3ms to complete. Adding “#pragma omp parallel for” doesn't result in an improvement probably because the overhead time is too big. Activities, such as thread creation, synchronization, and management take too much time in comparison to the execution of the function itself.

2. In the “enhanceDetails” function there’s no speed-up as well. In this situation the lack of improvement may be also caused by too big overhead time or by the fact that there are many variables which need to be private for the function to give the proper results (to not damage the output image).

Nonetheless, there were 3 functions in which I noticed a significant speed-up.

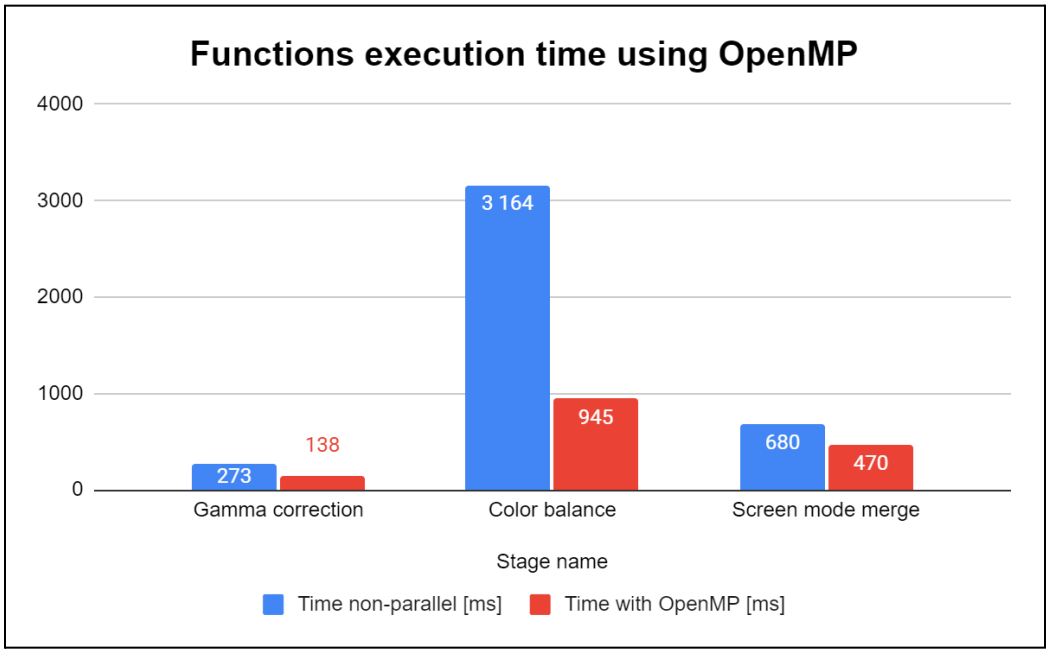
1. In the “colorBalance” function both the time difference and the speed-up were the greatest. Executing the program using 4 cores, I achieved the speed-up equal to 3,35. The time difference was equal to 2 219 ms.

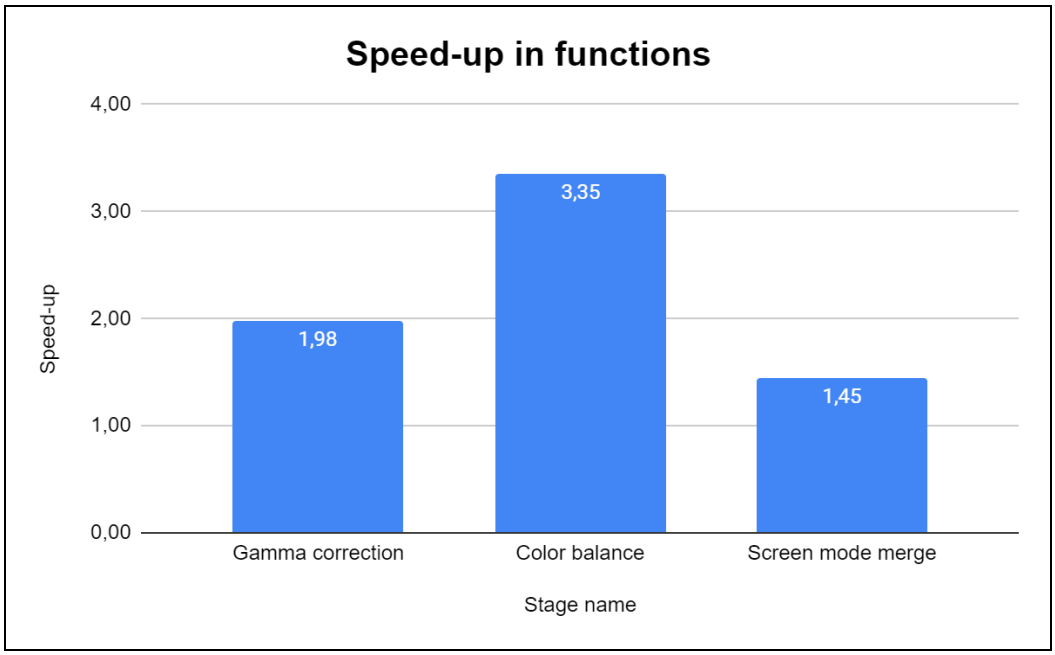
2. In the “gammaCorrection” function, besides using “#pragma omp parallel for”, I also needed to specify which variables should be private and which should be shared to keep the output image intact. To do this I used “private” and “shared” clauses.

   *#pragma omp parallel for private(p, tp) shared(in, tmp, a, b, curve)*

3. In the “screenMerge” function the private clause was also necessary.

   *#pragma omp parallel for private(pIn1, pIn2, pTmp)*






To use asynchronous processing, we needed to identify which stages do not depend on one another. In the "main" function such functions were "enhanceDetails' and "bloom". After executing them in parallel, it was obligatory to wait for both of them to complete before proceeding to the next step. To do this, *task1.wait();* and *task2.wait();* were used.

```
// enhance high frequency details
std::future<void> task1 = std::async(std::launch::async, [&image, &enhanced]() {
        enhanceDetails(image, enhanced, 20, 1.25);
});
// compute bloom mask
std::future<void> task2 = std::async(std::launch::async, [&image, &bloomed]() {
        bloom(image, bloomed, 70, 0.9);
});

task1.wait();
task2.wait();
```

Although each of the times of the two functions individually increased (which was due to a decrease in the number of computer resources for each function), the total application execution time was reduced.

## About the implemented code

**Aspects that define the size of the problem:**

- **Image Dimensions**
  - In this case, the size of the image inputted, which includes its width and height results in a change in the computational workload. Images that are on a larger scale will have more pixels and this results in more processing time because of the increase in complexity.

- **Image Processing Operations**
  - Different operations, such as denoising, gamma correction, and debayering contribute to the size of the problem at hand. As the complexity of the program and images increases, more computational resources will be needed.

**Code control structures of special interest in the solution to the problem:**

- **Sequential Control Structures**
  - The examples of sequential control structures used in the main.cc file are the loops in the "gammaCorrection" or "colorBalance" function that go over each pixel and or color channel.

  - Example in gammaCorrection Function:

```
for(int i = 0; i < in.rows; ++i) {
   for (int j = 0; j < in.cols; ++j) {
      // Apply gamma correction to each pixel
   }
}
```

- **Parallel Control Structures**
  - The same loops are parallelized by utilizing "#pragma omp parallel for". This results in the workload for processing the pixels being divided among many threads.

- In the image below, a snippet of the code in the gammaCorrection function is provided. The "#pragma omp parallel for private(p, tp) shared( in, tmp, a, b, curve) is the statement that parallelizes the loop that processes the image "in" and stores the result in "tmp". Each thread has its own "p" and "tp" respectively that go through certain sections of the image. The other variables "a","b", and "curve" hold the rest of the data for gamma correction which is shared.

```
    // for each pixel, apply the computed LUT
#pragma omp parallel for private(p, tp) shared(in, tmp, a, b, curve)
    for(int i = 0; i < in.rows; ++i)
    {
        p = in.ptr<unsigned short>(i);
        tp = tmp.ptr<unsigned short>(i);
        for (int j = 0; j < in.cols; ++j)
        {
            tp[j*3] = a * curve[p[j*3]] + b;
            tp[j*3+1] = a * curve[p[j*3+1]] + b;
            tp[j*3+2] = a * curve[p[j*3+2]] + b;
        }
    }
```

**Justification:**

- **OpenMP Integration:**
  - Integrating OpenMP directives in certain for-loops allows for data parallelism, which in turn processes the pixels more efficiently without changing any of the logic of the original sequential code. In this case, it should be used because "gammaCorrection" and "colorBalance" are independent of each other, so it is favorable in terms of parallelization. Another justification would be that images with many pixels have a longer processing time, so by splitting the computational workload across several cores, the processing time is reduced immensely.

- **Application of "std::aysnc":**
  - The application of "std::async" enables concurrent execution of the processing of the different image tasks. In the image below, it shows how "enhanceDetails" and "bloom" are executed concurrently using "std::async", which should reduce the total time to finish both tasks.

```
// enhance high frequency details
std::future<void> task1 = std::async(std::launch::async, [&image, &enhanced]() {
    enhanceDetails(image, enhanced, 20, 1.25);
});
// compute bloom mask
std::future<void> task2 = std::async(std::launch::async, [&image, &bloomed]() {
    bloom(image, bloomed, 70, 0.9);
});
```

**OpenMP or std::async instructions and blocks used for code parallelization:**

- OpenMP Blocks include the “pragma omp parallel for” directive, which is used in functions such as “gammaCorrection” and “colorBalance” in order to do the task of parallel processing all the image pixels.

## On the exploited parallelism

**Types of parallelism used:**

The code implemented in the main.cc file uses “data parallelism” and “task parallelism”.

- **Data Parallelism:**

  This type of parallelism was implemented using OpenMP, which includes “#pragma omp parallel for” and is also used in functions such as “gammaCorrection” and “colorBalance”

- **Task Parallelism:**

  Task parallelism was used with “std::async” and specifically with “enhanceDetails” and the “bloom” function. “Std::async” allows for certain tasks to be parallelized. This allows for the idle time to be reduced and a more effective management of resources.

Parallel programming mode:

- **Multi-threading programming**
  - In the main.cc file, OpenMP, and “std::async” are utilized to create several threads, which allows for different portions of the program to be parallelized.

**Communication alternatives:**

- Implicit communication in OpenMP:
  - The communication between the threads is mainly implicit and the memory is shared while using the directive “#pragma omp parallel for”. This is because all threads can modify or access shared variables. An example of this would be “in”, “tmp”, “a,”b”, or “curve” that is utilized in the “gammaCorrection” portion of the code.
  - OpenMP reduces the explicit communication needed because it automatically handles loop iterations and synchronization.

- Explicit Communication with “std::async”
  - Explicit forms of communication are used by running the tasks independently and using “std::future” to wait for tasks to be completed and by using “std::async” as

discussed in the previous sections with the “enhanceDetailss” and “bloom” functions.

Parallel programming style:

- Imperative Programming
    - The programming style used in the program is imperative because it is necessary to state which sections are going to be parallelized using either OpenMP directive or “std::async.

Type of parallel program structure:

- Mixed Parallel Structure:
    - There is a mixed parallel structure because both task and data parallelism are implemented as discussed previously. It uses the strengths of both structures to efficiently parallelize the image processing tasks.

## On the results of the tests performed and their context

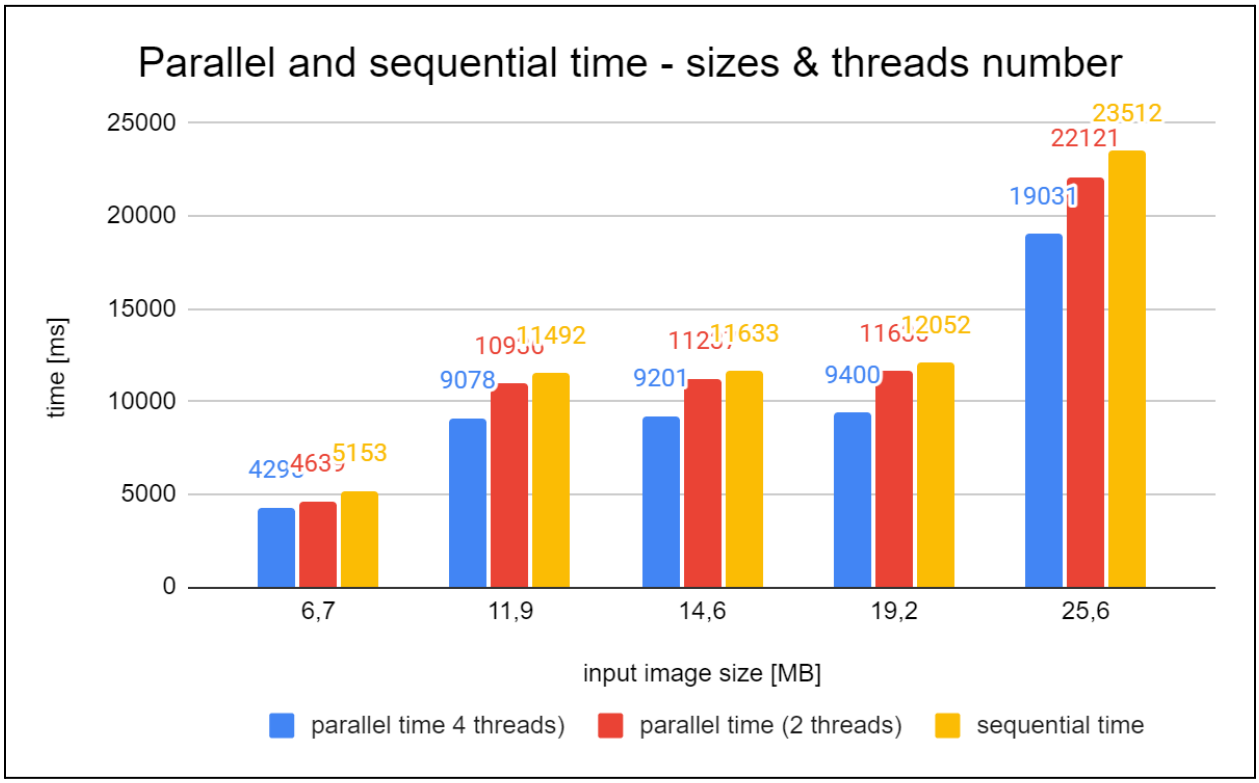
### Characterization of the parallel machine:

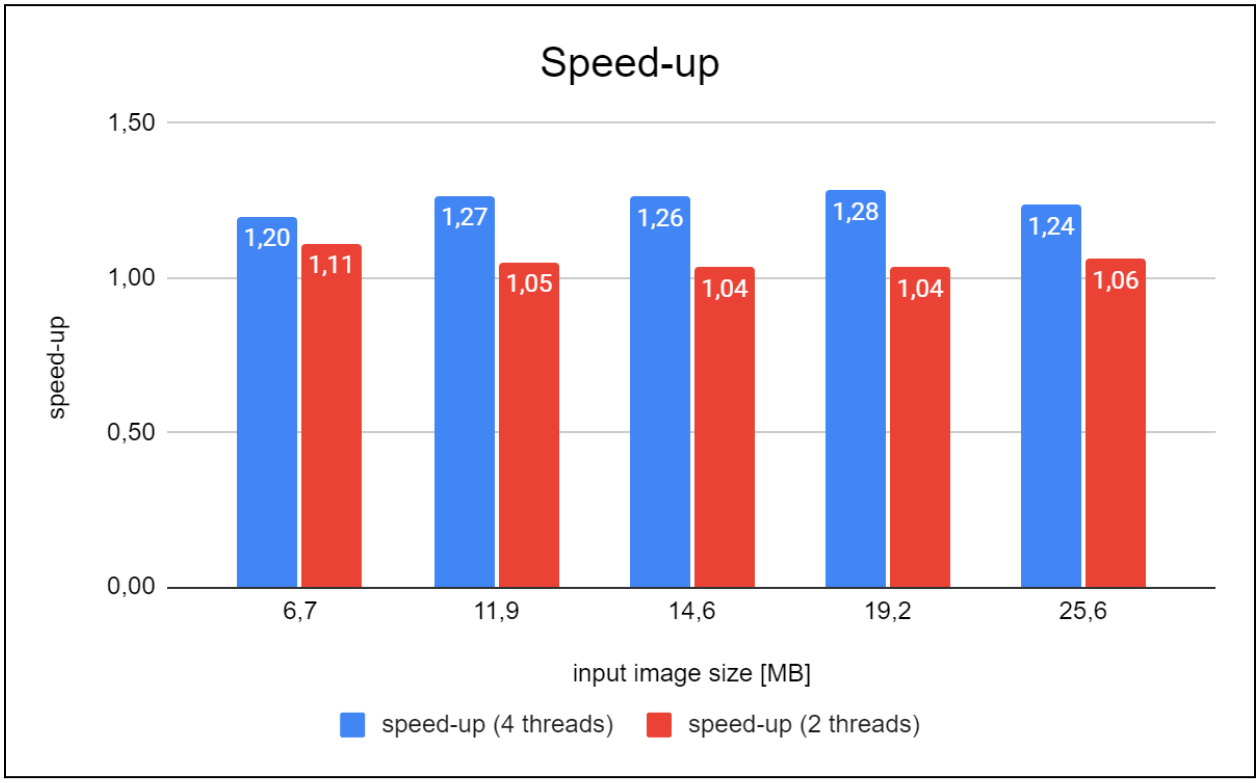
The parallel machine on which the program runs is a virtual machine. It is a single socket system with an Intel Core i5-1035G1 CPU. It has 4 cores and supports 4 threads, with a total of 4 online CPUs. The architecture is x86_64, and the system uses little-endian byte order. The cache system includes L1d cache of 192 KiB, L1i cache of 128 KiB, L2 cache of 2 MiB, and L3 cache of 24 MiB. The system operates in a virtualized environment using KVM as the hypervisor.

### What does the word ht mean?

In the output of the lscpu command on Linux, "ht" stands for Hyper-Threading. Hyper-Threading is a technology that makes one physical CPU core appear as two logical cores to the operating system. It helps improve multitasking performance by allowing the CPU to work on multiple tasks simultaneously. The "ht" flag appears in our output, so it indicates that Hyper-Threading is enabled on our CPU.

**Tests**


Parallel and sequential time - sizes & threads number
time [ms]
25000
20000
15000
10000
5000
0
23512
22121
19031
11492
9078
11633
9201
12052
9400
5153
6,7
11,9
14,6
19,2
25,6
input image size [MB]
parallel time 4 threads)
parallel time (2 threads)
sequential time


Speed-up
speed-up
1,50
1,00
0,50
0,00
1,20
1,11
1,27
1,05
1,26
1,04
1,28
1,04
1,24
1,06
6,7
11,9
14,6
19,2
25,6
input image size [MB]
speed-up (4 threads)
speed-up (2 threads)

The above graphs detail the effectiveness of paralellisation of image processing depending on the input image size. The paralellisation experiment has been performed using both 2-thread paralellisation and 4-thread paralellisation. The second graph, showing the speed-up as a fraction, suggests that effect of paralellisation correlates directly to the input image size, as the speed up doesn't seem to be affected by the input image size. The second graph does, however, highlight the improvement in speedup achieved by using 4-threads instead of 2-threads, which is to be expected. What is interesting, though, is the non-linear nature of the first graph. There are many processes that go into performing these entire operations, and maybe it is a subprocess that is causing the flatline in processing times from an image size of 11.9 MB to 19.2 MB. This is something that could be considered for further experimentation.

**What is the theoretical maximum speed gain?**

The theoretical maximum speed-up is equal to the number of threads, which in our case is 4. Nonetheless it is unreachable, due to the fact that not all the code is parallelized (and not all of it can be parallelized) and there is some overhead time. However, in the colorBalance function the speed-up equals 3,35, which is quite a good result.

**Which is the more efficient implementation of the 2?**

An almost 1,3 speed-up in a parallel implementation compared to a sequential one indicates an improvement in processing time. In general, a speed-up greater than 1 suggests that the parallel implementation is more efficient. However, the decision between parallel and sequential depends on factors such as the nature of the task, scalability, and resource utilization. Sequential deployment, characterized by a gradual step-by-step execution, becomes inefficient as any delay in a task causes a system-wide wait. Consequently, parallel implementation proves superior, especially for sizable programs. In the context of parallel processing, it significantly enhances the overall system performance. Opting for parallelization is particularly advantageous for programs with numerous independent loops, as in the case of our program, where the second option proves more efficient due to the lack of interdependence among loop results.

# Bibliography


[1] https://learn.microsoft.com/en-us/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170

[2] https://lsi2.ugr.es/jmantas/ppr/ayuda/omp_ayuda.php?idioma=en

[3] https://stackoverflow.com/questions/