Assignment 1: Parallel Image Filter

IMAGE1.JPG

FilteredIMAGE1.JPG

The Problem Description

The *ImageFilter* class is designed to perform a sequential, iterative nine-point image convolution filter on a linearized (2D) image. The process involves computing the average RGB-value of each pixel 'p' in the source array during each of the NRSTEPS (=100) iteration steps. This computation takes into account 'p' and its eight neighboring pixels in the 2D space. Subsequently, the computed values are written to the destination array. The challenge at hand involves parallelizing this operation to optimize its efficiency.

The Solution Approach

Retaining the sequential blurring steps due to their interdependency, the approach concentrates on parallelizing surrounding independent computations.

Approach 1: Task Separation by Height

This approach focuses on dividing the image processing task based on its height. By segmenting the image into distinct height segments, threads are allocated to handle specific segments independently. Each thread operates on its designated height range, performing computations without significant interference from other threads. This segmentation minimizes the need for synchronization between threads since they work on separate, non-overlapping height segments. Thus, it enables parallel processing of image regions based on their height, optimizing efficiency by reducing contention among threads.

Fig 1. Height-based Task Separation

Approach 2: Nested Threads for Width

In this approach, the focus shifts to further subdividing the workload within each height segment based on the width of the image. Threads are utilized to handle narrower width

segments within the previously allocated height segments. This nesting of threads enables concurrent processing of both height and width, enhancing parallelism. By breaking down the processing into smaller, more manageable chunks of width within the height segments, it allows for finer control over the workload distribution. Consequently, this approach maximizes parallelism by enabling simultaneous computation of specific width sections across the image.

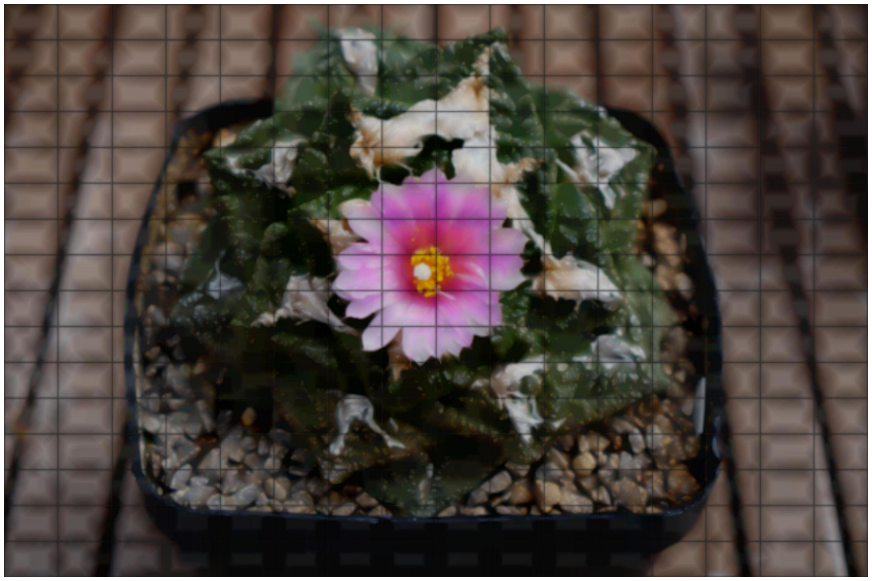

Fig 2. The Additional nested task separation for width

Approach 3: Balanced Load Distribution

The emphasis lies on ensuring a balanced distribution of workload among threads and nested threads. The height segments are evenly distributed among threads, and within each thread's assigned height segment, the width segments are equally divided among nested threads. This balanced load distribution minimizes thread idling, optimizing performance by efficiently utilizing system resources.

Approach 4: Unbalanced Load Illustration

This approach serves as an illustration of handling unbalanced workloads. It deliberately showcases scenarios where the allocation of height or width segments among threads is uneven, resulting in varying workloads for different threads. In this case we give the first thread twice the amount of work, while the others’ is reduced.

Fig 3. The first thread takes on twice the amount of work and the other threads' work gets reduced leading to an unbalanced load.

Analysis

There have been a few approaches to solve the problem. The main approach is balanced load with task separation by height. This is because:

- the problem in this case is embarrassingly parallel, so all the threads can have the same load, and,
- Having nested threads for width, in this case, only creates overhead.

Below there has been an attempt to modify the code so we can show a visualization of how these approaches could look like.

Framework and Configuration

- *ExecutorService is a JDK API that simplifies running tasks in asynchronous mode. Generally speaking, ExecutorService automatically provides a pool of threads and an API for assigning tasks to it.*
- In order to try the various approaches the configuration run should contain these three arguments:
    - *Image_path*
    - *balanced (true or false)*
    - *width*

Limitations and Results

The code was run on both the Mac M2 Pro and the Core i7 12700 processors, exploring their performance in terms of efficiency and speedup during image processing.

Mac M2 Pro

The Mac M2 Pro, despite having eight cores, encountered challenges achieving the required efficiency with eight threads. Two primary factors potentially influenced this outcome:

1. Core Distribution: Out of the eight cores, the Mac M2 Pro comprises four performance cores and four efficiency cores. This configuration might have impacted the parallelization efficiency.
2. Initial Speed: The initial speed of image processing without parallelization on the Mac M2 Pro was notably rapid, completing the task in 18 seconds. This swift execution suggests that the overhead from thread creation might have impacted the efficiency of parallelization.

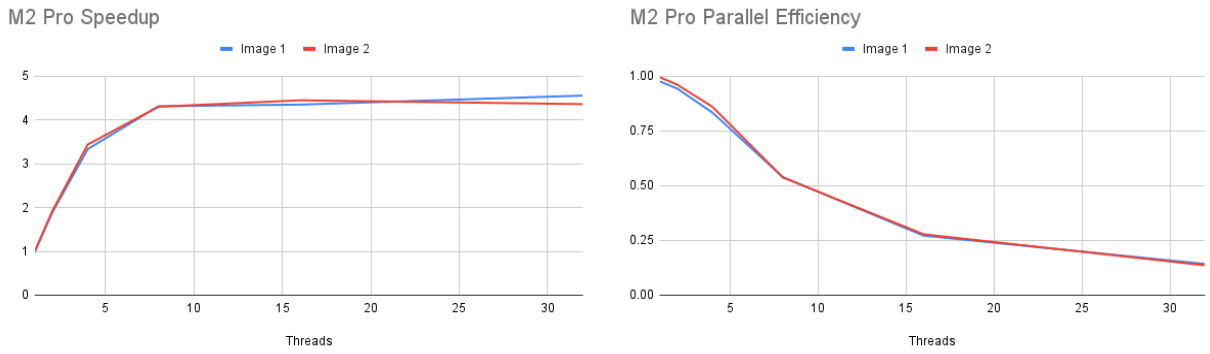

Fig 4. M2 Pro: Visualization of Speedup increase per number of Threads (left), and Parallel Efficiency per number of Threads (right)

Core i7 12700

When the same code was executed on the Core i7 12700, parallelization exhibited similar speedup results compared to the Mac M2 Pro. However, the crucial distinction emerged in terms of efficiency curves.

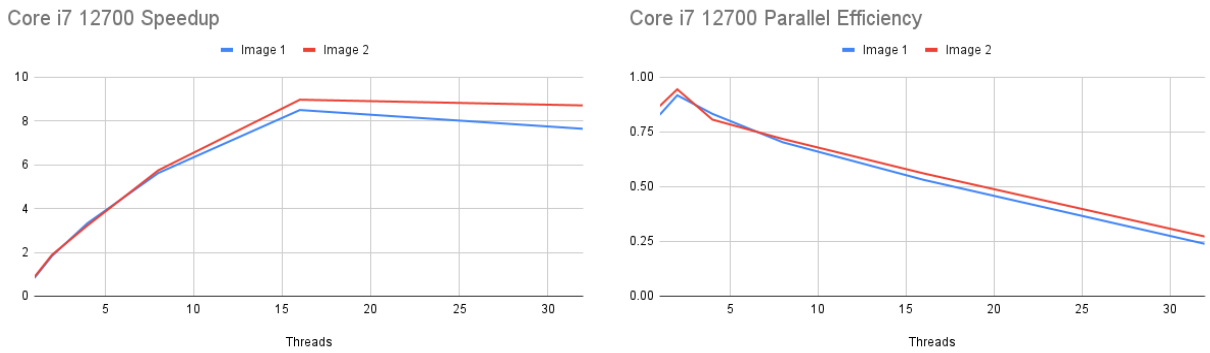

Fig 5. Core i7 12700: Visualization of Speedup increase per number of Threads (left), and Parallel Efficiency per number of Threads (right)

Comparison

To compare the performance of each processor we take the average results of both Images to compare speed, speedup, and parallel efficiency, respectively.

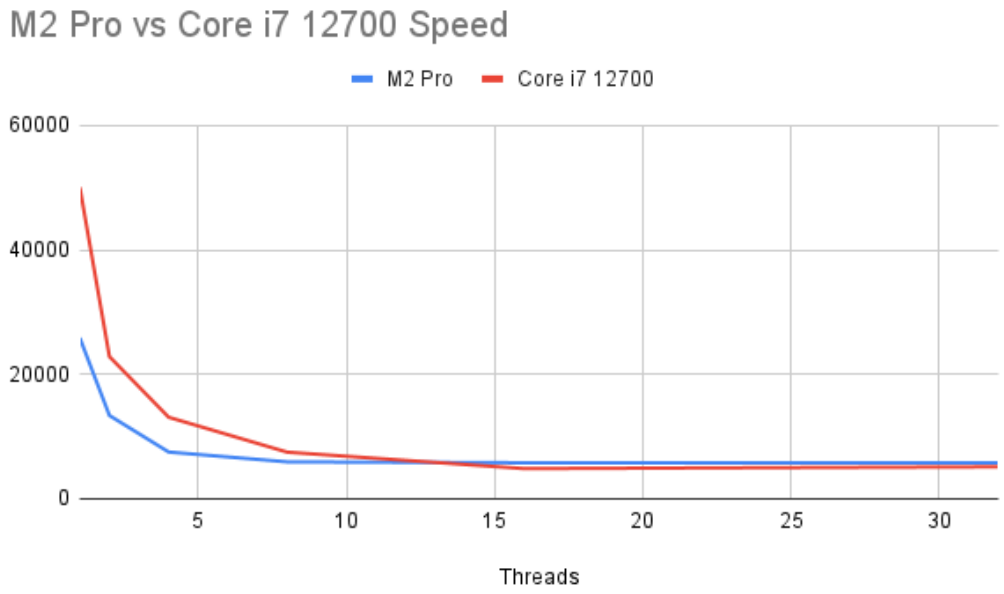

Fig 6. Speed Comparison

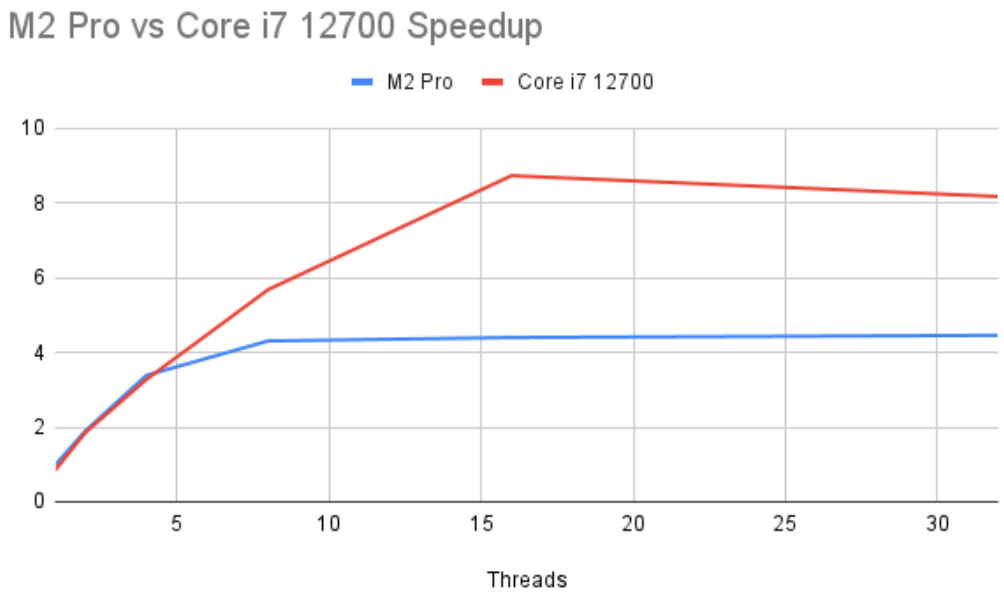

Fig 7. Speedup Comparison

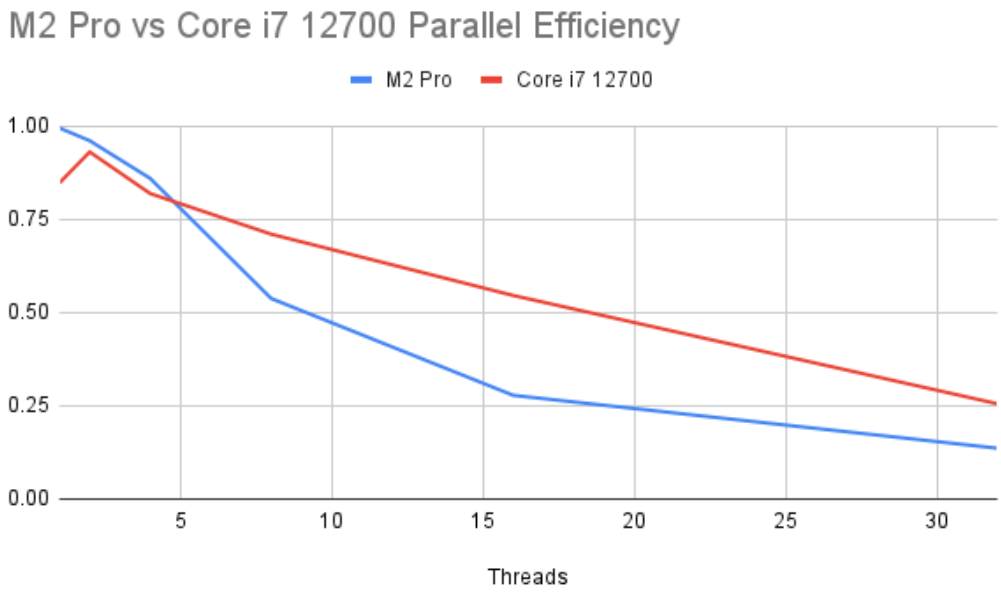

Fig 8. Parallel Efficiency Comparison