On Optimisation of Parallel Blurring

Aniket Gupta - CID: 02241962

Introduction

This report will explore variations in the performance characteristics of five parallel picture blurring algorithms in C. The results depend on the execution time of these algorithms to blur the same image. We will also discuss some technical issues that arose during experimentation, as well as reasoning of why a particular algorithm is quicker or slower than another. All tests were run on: HP EliteDesk 800 G6 TWR, Nvidia GPUs (Geforce GT 1030 2GB) Intel Core i7-10700 2.90GHz. These machines have 16GB of RAM, a 512GB SSD Disk and run on Linux.

Experimentation

The experiment is conducted by running each algorithm 100 times on an image and then taking the average execution time for each algorithm. Without cause, the following image has been selected as candidate for experimentation:

Fig. 1. Test Image, 600 x 600 pixels

The blurring algorithms implemented for testing are:

1) **Sequential**, where each row and column is traverse and every pixel is blurred in order
2) **By pixel**, where each pixel is assigned to an individual thread to be blurred
3) **By column**, where each column is assigned to an individual thread to be blurred
4) **By row**, where each row is assigned to an individual thread to be blurred
5) **By sector (quartered)**, where the image is divided into quarters and each quarter is assigned to an individual thread to be blurred

Results

The following image is the blurred version of Fig 1, as outputted by the experiment. The effect of the blur is clearly observable and consistent with all algorithms:

Fig. 2. Outputted Image, 600 x 600 pixels

The results of the experiment can be represented by:

Algorithm	Average execution time (ms)
Sequential	137
By pixel	7706
By column	26
By row	23
By sector (quartered)	35

Table 1: Average execution time (ms) of each algorithm

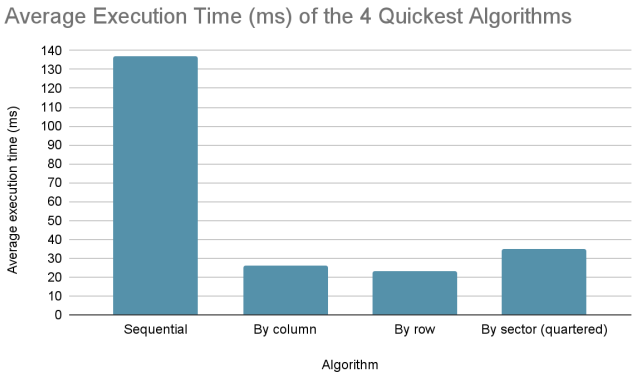

Fig. 3. Average execution time (ms) of the 4 quickest algorithms after 100 iterations of each on Fig 1

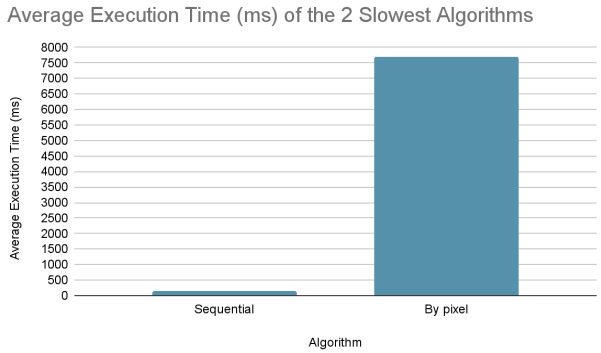

Fig. 4. Average execution time (ms) of the 3 slowest algorithms after 100 iterations of each on Fig 1

Using Fig. 3 and Fig. 4 as a medium to understand Table 1, we can conclude that the quickest parallel blurring is **by row**, followed very closely by **column**. The next quickest is **by sector (quartered)**, while the **sequential** algorithm is penultimate in speed. Finally, one can assert with certainty that **by pixel** is the slowest algorithm.

Discussion and Evaluation

Technical Issues

The primary difficulty when carrying out this study was with managing all threads concurrently. Each blurring algorithm sporadically needs to wait on previous threads to join current running threads. This happens because at times the process will run out of free threads. I chose to retain parallelism and made use of the *pthread_try_join* function provided by the *pthreads* library. This is a non-standard, non-portable version of the conventional *pthread_join* function that attempts to join with a terminated thread without blocking. Essentially, if the callee thread has not terminated, it returns immediately with an error code. If the thread has terminated, it behaves similarly to *pthread_join*. Thus, in my implementation of each algorithm when a new thread creation fails, *pthreads_tryjoin_np* is called on all previously deployed threads. Thus, the algorithm can free resources while parallelised. Note that as *pthreads_tryjoin_np* is non-portable, this experiment can only be run on a GNU Linux machine.

Findings and Reasoning

There are a few things to keep in mind when evaluating any algorithm that processes large volumes of data.

1) Thread Management

The overhead associated with creating and managing threads can significantly impact performance. Some parallelisation frameworks may handle this more efficiently than others.

2) Temporal and Spatial Locality

Algorithms with good temporal locality can benefit from caching mechanisms. Parallel algorithms need to be designed to minimise redundant memory accesses, allowing multiple threads to reuse data from the cache rather than fetching it from main memory. Additionally, algorithms with good spatial locality are more likely to benefit from parallelisation, as threads can work on nearby pixels without frequently accessing distant memory locations.

We can now explore potential reasons for the speed of each algorithm being ranked in this order.

By row: Neighbouring pixels are likely to be accessed together, improving cache efficiency. However, if there are many more rows than columns this could lead to poor thread management and hence slower blurring than **by column**. We have avoided this issue here by using a square image. Processing pixels by row reduces dependencies between neighbouring pixels. This allows for increased parallelism, as different threads can work independently without significant contention for shared resources.

By column: As in the case of rows, processing neighbouring pixels improves cache efficiency. The reverse argument for poor thread management and minimising dependencies holds for columns.

By sector (quartered): This ensures that there is a maximum of 4 threads running at the same time. Furthermore, the layout of an image will not affect the efficacy of this algorithm, as it unvaryingly sections the image and runs threads the same way.

Sequential: Serves as a baseline, with only one thread being active at a time. This leads to poor concurrency. Each pixel's computation depends on the previous pixels' results in the same row or column. However, one could argue that this saves CPU time that would otherwise be spent on parallelisation.

By pixel: Assigning each pixel to an individual thread results in a significantly higher execution time. This is likely due to the overhead of managing a large number of threads, potentially leading to increased contention for resources and synchronisation issues, negatively impacting performance. Moreover, we may have poor spatial locality if neighbouring pixels are assigned to different threads. Furthermore, as there is a cap on the number of threads that can run at a time and this algorithm will generally require the maximum number, it will periodically need to wait for threads to terminate.