Comparison of image processing techniques in serial and parallel

^{1}Aravind S, ^{2}Arun JK , ^{3}Ashik MP and ^{4}Chinmay RB)
(USN: ^{1}1MS18CS024,^{2}1MS18CS029,^{3}1ms18cs030 and ^{4}1ms18cs036)

Abstract

In general Image processing requires working on all the pixels of the image. An hd image of resolution of 720p (1280 x 720 pixels) has around a million pixels (921,600 pixels) and a full hd image of resolution of 1080p(1920 x 1080 pixels) has around 2 million pixels. Since the number of pixels is huge, working on each pixel sequentially is very inefficient and requires a lot of time. Hence working on multiple pixels simultaneously using threading and/or multi-processing is beneficial. Moreover, general consumer grade computers have multi-core, multi-thread CPU's which provides enough computation power for simple to intermediate image processing.

This paper compares the performance of sequential and parallel methods of image processing (by applying various filters).

Introduction

Image processing deals with transforms and operations on images. These functions are applied on images which gives us a desired output. The operations can range from operations like simple image enhancement, contrast/colour correction, sharpening to more scientific operations like, noise removal, edge detection, Fourier analysis, image segmentation etc. These operations involve manipulation of the pixels of the source image, applying certain transforms on them which produces a resultant output image. Such operations find many applications. Edge detection and object detection are operations that are frequently used in self-driving cars, in automated assembly lines, quality control etc. The operations like image enhancement, histogram equalisation, colour correction are also common operations used in video editing and other applications. Software like photoshop and other editing software rely heavily on performing such transforms on images to get the desired output.

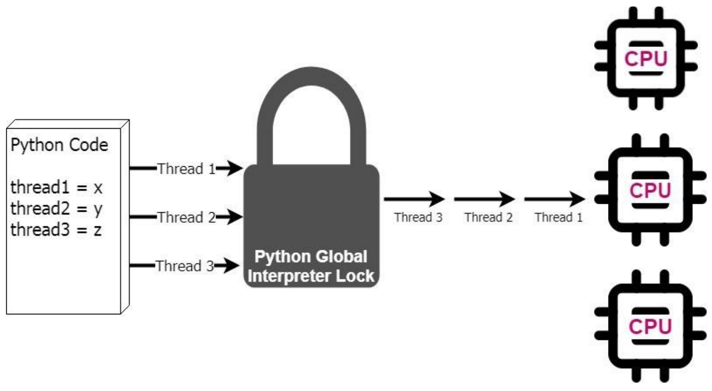
Since these operations have such a widespread use, there is also a need to optimise such tasks, to make them faster and efficient. One approach to make such operations faster is to use parallelisation techniques. Since many of the image processing operations and transforms are not dependent on intermediate results, it is an ideal candidate for using parallelisation to achieve speedup.

Parallelisation can be achieved in an application by using concepts like multi-threading and multi-processing. Both methods make use of the multiple cores in a CPU. By using such techniques, instead of performing all the operations on a single core, the operations are performed simultaneously on multiple cores which provides a vast increase in speed.

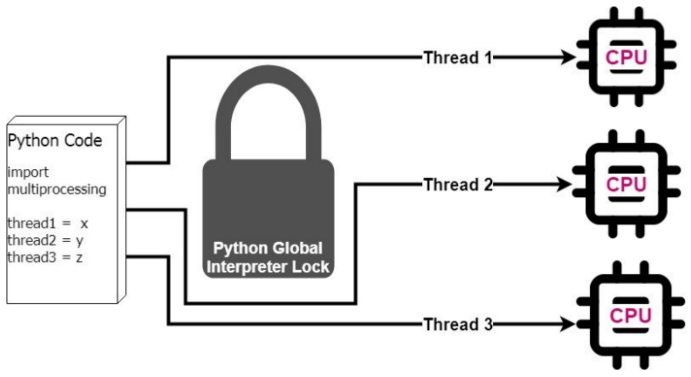
Another method is to use GPUs to achieve this speedup. GPUs are specialised hardware that have many cores (much more than a CPU). Their main purpose is to perform tasks in parallel much faster and in an efficient manner compared to CPUs. GPUs are used in areas like rendering graphics, geometry, video editing and also in areas like machine learning and deep learning.

The method that we have used here, explores the former idea to achieve a performance gain. We have used python to perform these operations since it offers libraries like OpenCV which make handling images and applying operations on them very easy. multi-threading in python however uses only 1 core and performs context switching between the different "virtual threads" which is effectively the same as performing the operation on a single core. This is due to the **GIL (Global interpreter lock)** that python uses.

In simple terms, the Python Global Interpreter Lock (GIL) is a mutex (or a lock) that allows just one thread to control the Python interpreter. This means that at any given time, only one thread may be in a state of execution. The GIL has no effect on developers writing single-threaded applications, but it can cause performance issues in CPU-bound and multi-threaded code.



a. Default multithreading in python



b. Multiprocessing bypassing GIL

The GIL has earned a reputation as a "infamous" element of Python since it permits only one thread to operate at a time, even in multi-threaded architectures with many CPU cores.

To circumvent this problem, python offers the multi-processing library which helps us leverage the multiple CPU cores by spawning separate processes instead of multiple threads within the same process. In this way, each process gets is own **GIL** and code that executes concurrently can be made to use the multiple cores simultaneously. For communication between the different processes (**IPC-inter process communication**), we have used shared memory which acts as a buffer and stores the final image.

Method

For achieving the increase in speed for the image processing applications, the approach we have used is to split the image into multiple sections and have an individual process compute the necessary calculations for each portion of the image. The output of the operation on one portion of the image doesn't depend on the output of another portion of the image. We can exploit fact which makes parallelizing a suitable method to gain speedup. There is also no need for any locking as each process operates on a different portion of the memory and no two processes access the same portion of an image.

For demonstrating the advantage of using parallelization to perform the image processing operations we have used common image processing operations such as edge-detection, applying image point transformations and image noise removal. These are common operations that are applied during the pre-processing stage and can greatly reduce time in the pre-processing pipeline.

To illustrate the differences, we have performed the same operations serially and parallelly and compared the different execution times. To achieve parallelism, we have used shared memory to communicate between the different processes. Each individual process performs the required operation on each pixel of the input image and the result of the operation is written to a specified portion of the shared memory. In this way, using shared memory for inter-process communication helps to reduce the runtime drastically.

The edge detection and noise removal and blurring operations use a kernel which performs a convolution operation across the entire image and the log transformation uses point transformation on each pixel to decrease the contrast gradient of the image, enhancing darker areas while also slightly decreasing the contrast of the brighter areas.

Code explanation

Python uses a **Global Interpreter Lock (GIL)** which means that at any time only one core can execute python instructions. The core obtains the lock and executes the instructions preventing other cores from executing python instructions concurrently. To circumvent this, we have used the multiprocessing library from python. By spawning an individual process, each individual process gets its own GIL and it is now possible to execute python instructions concurrently. Communication between the processes is done using shared memory. The final image that is being written to is declared as a shared memory buffer which multiple processes can write to. Because of the nature of the operations that are being performed, coupled with the fact that each process writes to a different portion of memory, there is no need to obtain a lock before writing into the shared memory and all the processes can write to the shared memory concurrently.

The code for declared memory as shared memory looks like this

```
shm = shared_memory.SharedMemory(
        create=True, size=final_image.nbytes, name="shr_mem")
```

while creating the memory, we set the initial size of the memory (which is equal to the size of the image) and provide the shared memory with a name so that other processes can access the memory through its name.

The function that uses the shared memory must have this piece of code which uses the shared memory.

```
existing_shm = shared_memory.SharedMemory(name='shr_mem')


final_image = np.ndarray(
        (image.shape[0], image.shape[1]), dtype=np.uint8,
buffer=existing_shm.buf)
```

The structure of the main program looks like this -

```
num_process = 8
    rows_n = int(img.shape[0]/num_process)

    process_pool = []





    for i in range(num_process):
        process_pool.append(Process(target=convolution.convolv
e_multi_process, args=(
            img, i*rows_n, (i+1)*rows_n, 1, img.shape[1]-1,
laplace)))

    for i in process_pool:
        i.start()

    for i in process_pool:
        i.join()
```

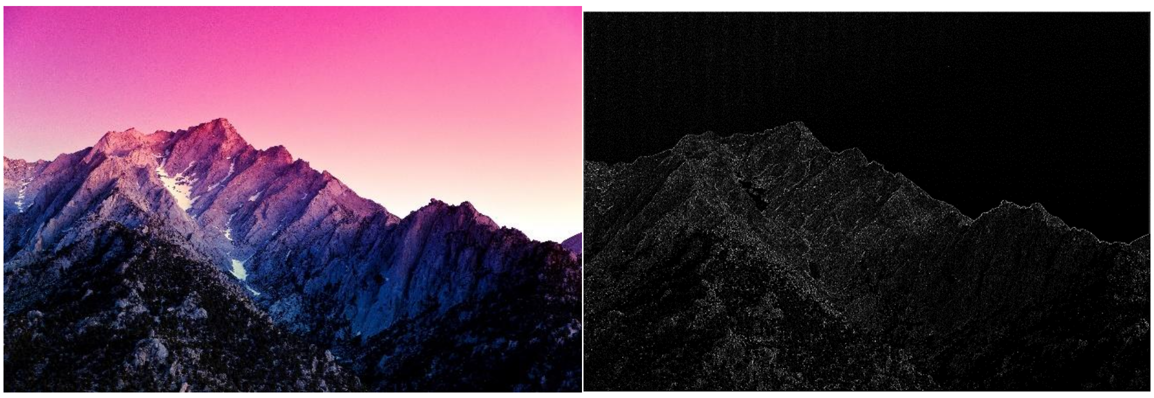
Since the machine on which we are executing the programs has 8 cores, (effectively has 8 logical processors but 4 actual cores) we have chosen the number of processes to be 8. We create an empty list of processes which is the process pool. Then, for each process in the process pool, we assign a different portion of the image for each process to handle. Each process in the process pool is then started calling its corresponding start() method. The join() method is called for each process to wait for all the process to finish to completion.

Results

The programs were run on intel core i5 – 8th generation CPU which has 4 physical cores and 8 logical cores. The image processing operations were performed serially and parallelly a number of times and an average of the results were taken.

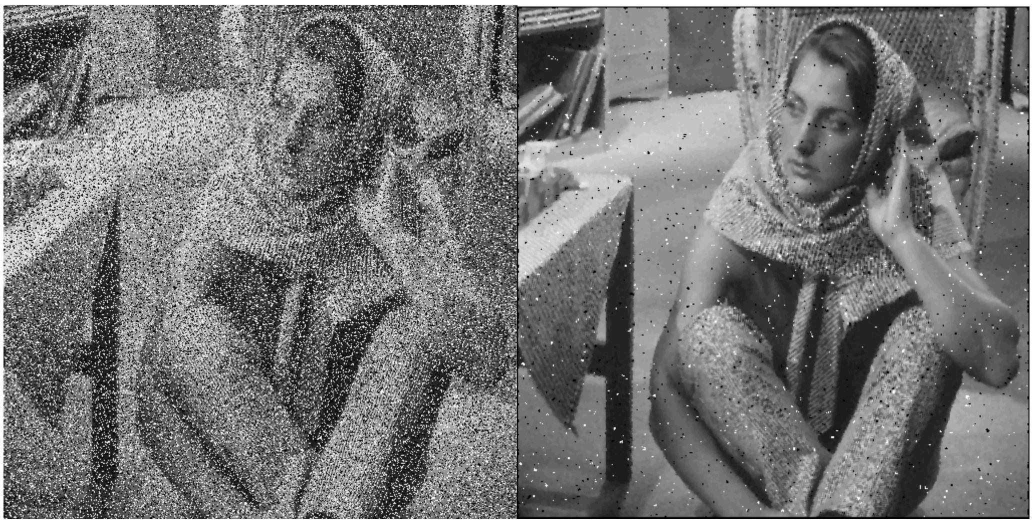
The following are the results of the transforms that were used.

Edge Detection



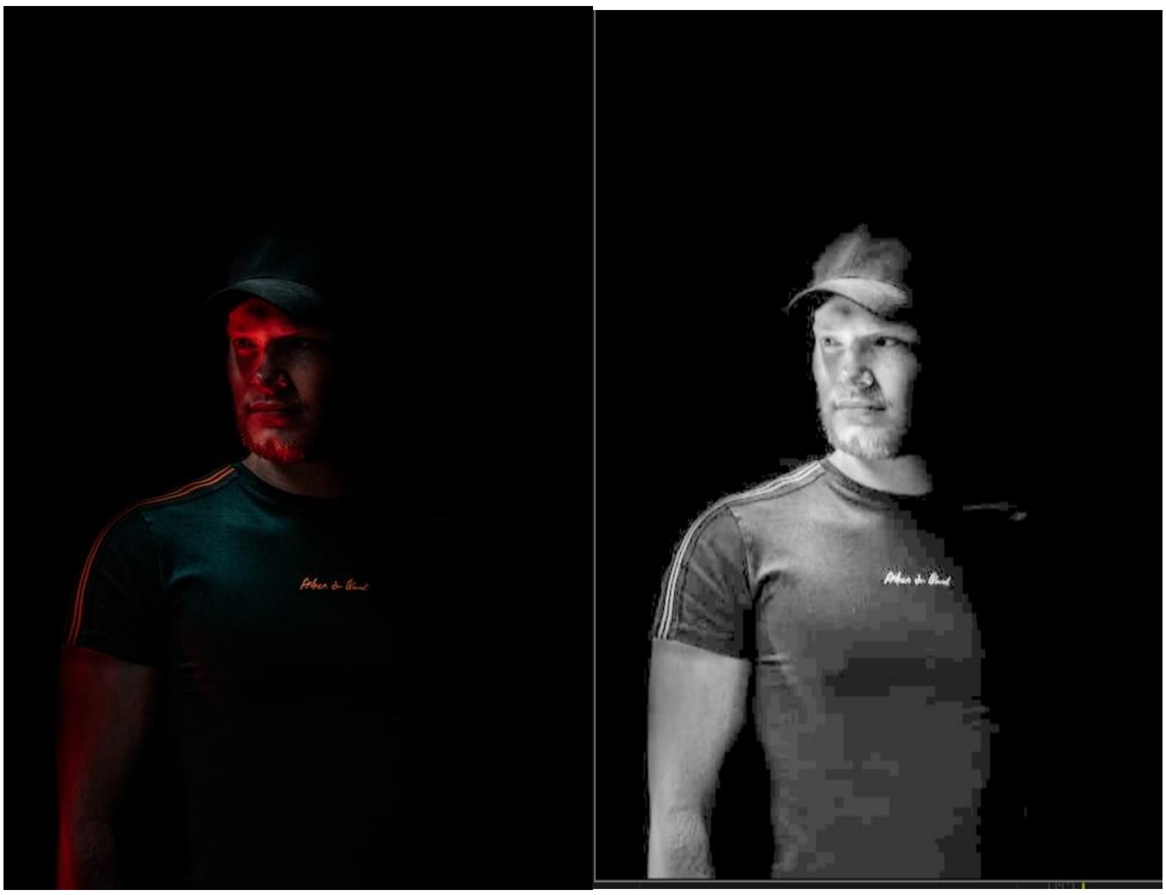
The image on the left represents the original image and the image on the right is the result of the edge detection operation which detects the edges of the image.

Median noise removal



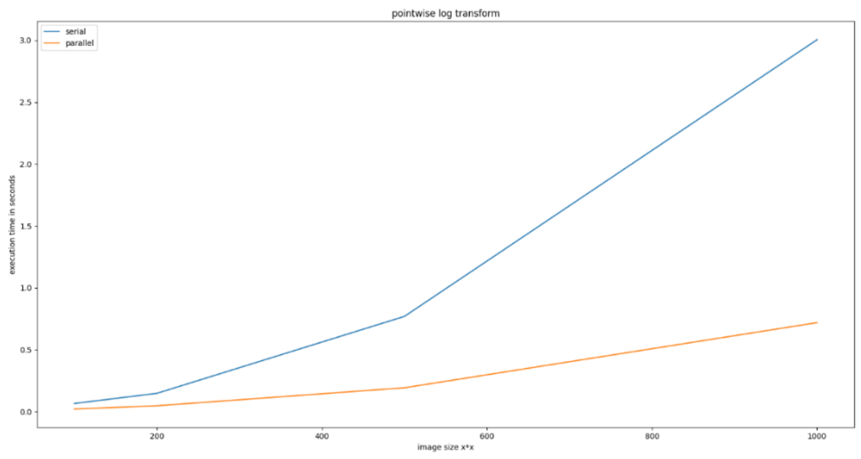
The image on the left represents the input image which has a lot of noise. While the image on the right is the output after applying the median filter on the noisy image.

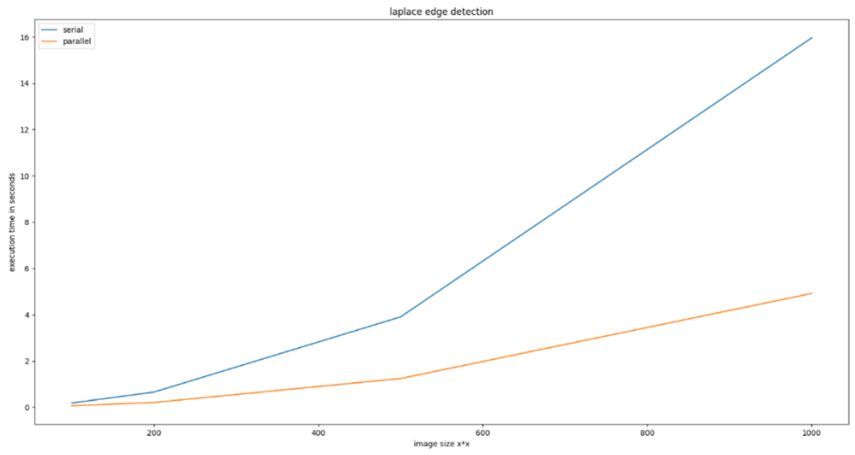
Log transformation

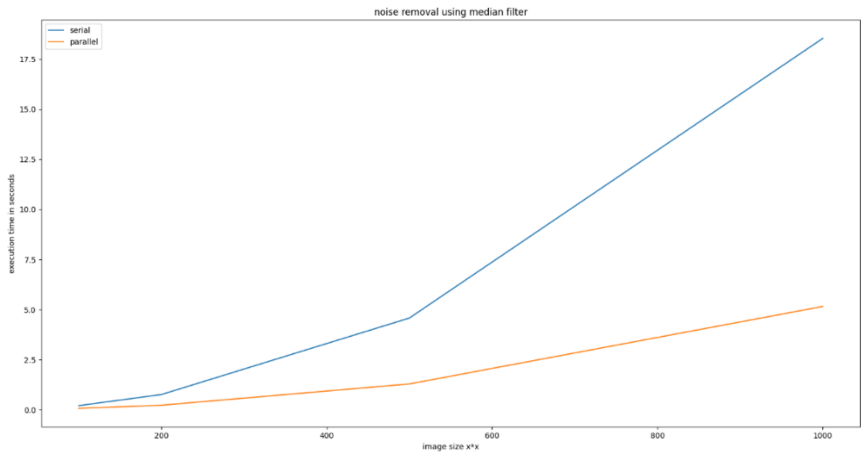


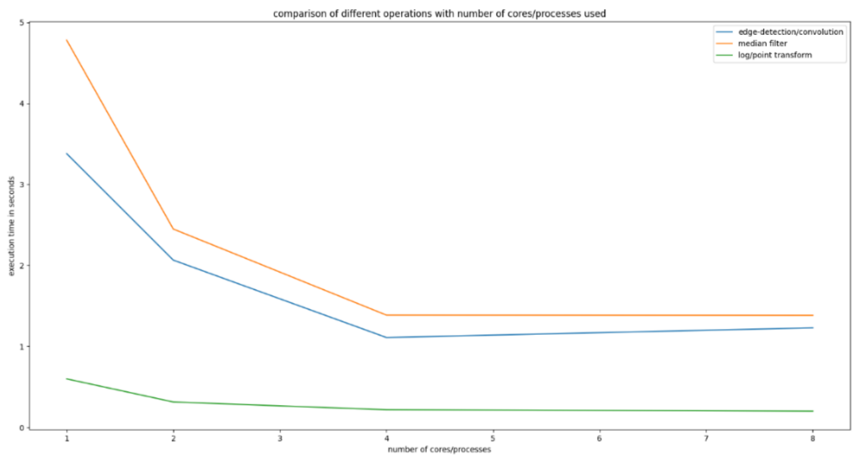
The left image is the input image where the face of the person is barely visible. On applying the log transform, the dynamic range of the pixel values decreases and the overall image pixel intensity is enhanced.

Plot of the Results









The above images show a plot of the results of the experiment. The first 3 images show a plot of the results of the log transform, convolution and the median filtering. As expected, the runtime of the programs in parallel is lesser than serial.

The last plot shows the comparison between the different operations on the same size image but the number of cores used is varying. As the number of cores increase, the execution time decreases.

In all of the cases. The gain in speedup has been close to about 3.15-3.3 times speedup. This is expected as the number of physical cores is 4 and one would expect a speed up of 4, but inter-process communication and factors decreases this factor and realistically we only get a speed up of about 3.2 times at an average

Limitations

Although the parallelization helps us to achieve speedup, further techniques that are unique to image processing can also be applied to make the process even faster. Techniques such as using integral images, which is an alternate representation to store images, can be used but require more careful handling and operations. Signal processing techniques such as Fourier transforms and inverse Fourier transforms can also be used. The Fourier convolution, makes the multiplication operations turn into addition operations in the frequency domain from which the original image can be obtained by doing an inverse transform.

Conclusion

In this paper, we have explored parallelization as a solution to optimise and make image processing operations faster. We have used multiprocessing along with shared memory for inter-process communication as a means to achieve parallelisation. The paper compares the runtimes of the different methods serially and parallelly, and from the plots, it is evident that indeed, there is a speedup in the operations. The idea of using a GPU to enhance such operations is also an idea that is worth being pursued. But GPUs are not common in all computers however, using specialised hardware to perform such tasks would provide an enormous amount of speedup due to the massive parallel processing capabilities of the GPU over the CPU.