Comparing the execution of Image Compression on sequential and parallel execution

[1]Aravind Shreyas, [2]Dheeraj Bhat, [3]Divya, [4]Gaurav V

[1]1MS18CS025, [2]1MS18CS040, [3]1MS18CS043, [4]1MS18CS046

Abstract

The shift to multicore computing is pervasive throughout most industries, and the image and machine vision industries are no exception. This could improve throughput and reduce response times for camera systems dealing with growing amounts of data. Images are processed using two or more computer cores in multicore image processing. In other words, the processing of a task from an imaging system is shared among numerous cores. Moving to a multicore system has the overall benefit of reducing response time and increasing throughput in an imaging system. Multicore allows users to make use of the latest PC processor designs, allowing algorithms and software to run quicker and perform more tasks. The increasing computational capacity and programmability of multi-core architectures offer promising opportunities for parallelizing image compression and processing methods.

Introduction

Image compression is a sort of data compression that is used to lower the cost of storing or transmitting digital photos. When compared to generic data compression approaches used for other digital data, algorithms can take advantage of visual perception and the statistical features of picture data to deliver greater outcomes. Image compression is a key stage in the field of image processing before we begin processing larger images or films. An encoder performs image compression and outputs a compressed version of the image. The mathematical transforms play a crucial part in compression operations.

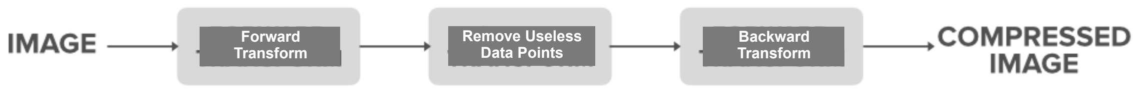


Fig. 1. Flow chart of the process of image compression

Image processing is time-consuming. Real-time applications are frequently time-constrained. As a result, serial image processing does not meet real-time requirements. Parallel computing approaches, particularly multicore and multiprocessing technologies, should be leveraged to overcome this problem.

A type of high-performance computing is parallel processing. Data is broken into small chunks and allocated to different execution units in parallel processing. As a result, data distribution is difficult in parallel processing. Data partitioning should be fine-grained to improve computer system efficiency by minimising execution time and memory bottleneck.

The global interpreter lock (GIL) in Python causes single-CPU utilisation by allowing only one thread to carry the Python interpreter at any given moment. The global interpreter lock was created to address a memory management problem, but as a result, Python can only use one CPU.

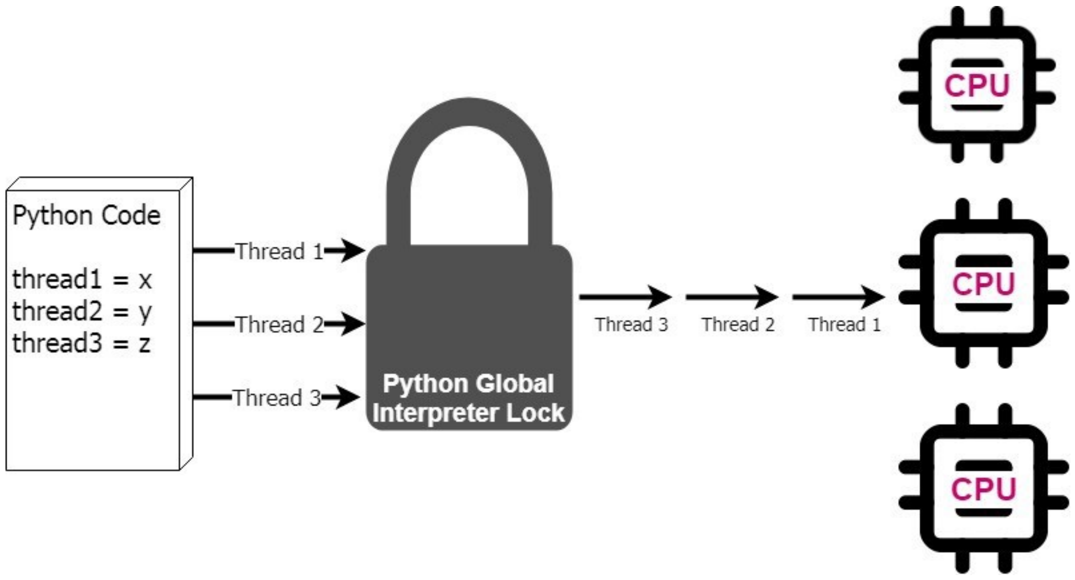


Fig. 2. Representation of traditional serial python global interpreter lock

Multiprocessing can significantly speed up processing. Bypassing the global interpreter lock when running Python code allows us to take advantage of multiprocessing, which allows the function to run quicker. We can select some areas of code to circumvent the global interpreter lock and transmit the code to many processors for simultaneous execution using Python's built-in multiprocessing module. Multiprocessing has three prerequisites:-

- The code must not be reliant on previous outcomes
- Data does not need to be executed in a particular order
- The program does not return anything that would need to be accessed later in the code

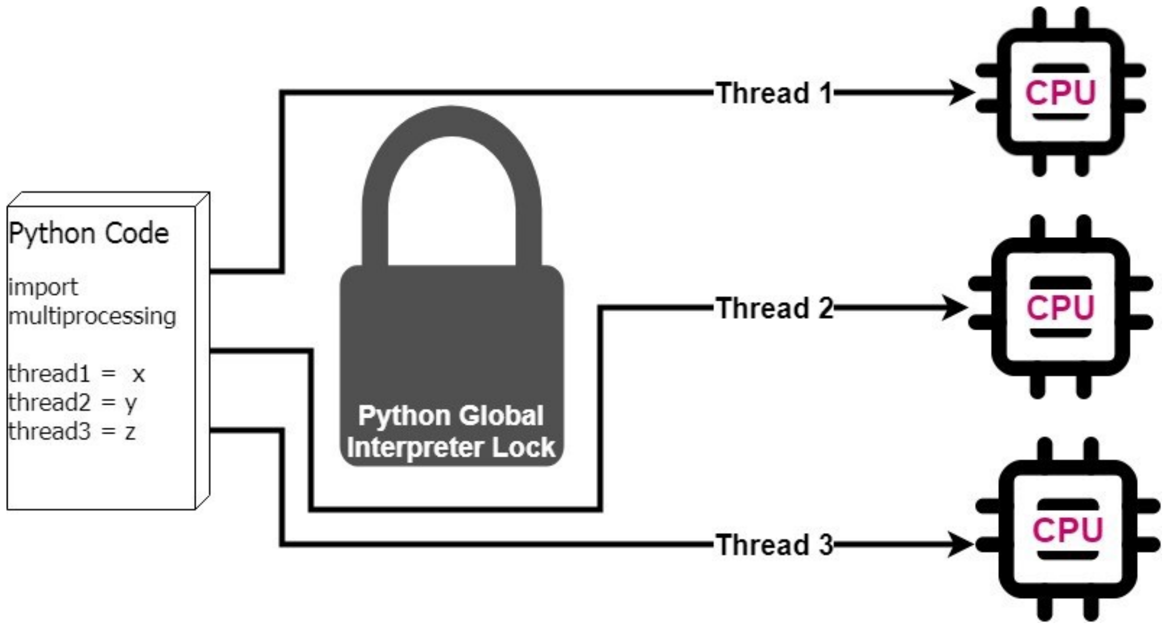


Fig. 3. Representation of bypassing python global interpreter lock using multiprocessing

Method

The following section describes our code for image compression which has been performed serially and parallelly.

```
import os
import time
import multiprocessing
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np

imagesPath = os.path.join(os.getcwd(), "Images")

try:
    os.mkdir(os.path.join(os.getcwd(), "Serial Compressed"))
    os.mkdir(os.path.join(os.getcwd(), "Parallel Compressed"))
except OSError as error:
    pass

serialTime = {}
parallelTime = {}

def compressImage(file, type, verbose = False):
    formats = ('.jpg', '.jpeg', '.png')
    if os.path.splitext(file)[1].lower() in formats:
        filepath = os.path.join(imagesPath, file)
        picture = Image.open(filepath)
        compressedPath = os.path.join(os.getcwd(),type)
        picture.save(os.path.join(compressedPath, file), "JPEG", optimize = True,
quality = 10)
    return

def serialProcess():
    print("==== Serial Image Compression ===\n")
    starttime = time.time()
    for index,file in enumerate(os.listdir(imagesPath)):
        print("\t"+str(index+1)+". Serial Compressing "+file)
        compressImage(file,"Serial Compressed")
        if(index % 50 == 0 and index!= 0):
            serialTime[index] = time.time() - starttime
    global serialTimeTotal
    serialTimeTotal = time.time() - starttime
    print("\n\tSerial Image Compression Completed.")

def parallelProcess():
    print("\n=== Started Parallel Image Compression ===\n")
    starttime = time.time()
    global processes
    processes = []
    for index,file in enumerate(os.listdir(imagesPath)):
        print("\t"+str(index+1)+". Parallel Compressing "+file)
        if(index%50 == 0 and index!= 0):
            parallelTime[index] = time.time() - starttime
        p = multiprocessing.Process(target=compressImage, args=(file,"Parallel
Compressed",))
        processes.append(p)
        p.start()
    global parallelTimeTotal
    parallelTimeTotal = time.time() - starttime
    print("\n\tParallel Image Compression Completed.")

def plotGraph():
    x = list(serialTime.keys())
    y1 = list(serialTime.values())
    y2 = list(parallelTime.values())
    X_axis = np.arange(len(x))
    plt.bar(X_axis - 0.2, y1, 0.4, label = 'Serial Processing')
    plt.bar(X_axis + 0.2, y2, 0.4, label = 'Parallel Processing')
    plt.xticks(X_axis, x)
    plt.xlabel("Images")
    plt.ylabel("Time Taken(s)")
    plt.title("Serial vs Parallel processing for Image Compression")
    plt.legend()
    plt.show()

if __name__ == '__main__':
    serialProcess()
    parallelProcess()
    print("\n\n === Compression Statistics ===")
    print("\nNo. of images : 268")
    print("\nTotal size of images before compression : 276.5 MB")
    print("Total size of images after compression : 11.4 MB")
    print('\nSerial Compression took {} seconds'.format(serialTimeTotal))
    print('Parallel Compression took {} seconds'.format(parallelTimeTotal))

    plotGraph()

    print("Terminating all processes, This might take a while ...")

    for process in processes:
        process.join()
```

Function Descriptions:-

compressImage(file, type): This function is used for compressing a given image using the python Pillow library. The function takes in 2 parameters, the first being the filename which is further used to fetch the file path. The second parameter is the type of processing being performed i.e. serial or parallel, the type parameter is used to save the resultant compressed file in the correct directory.

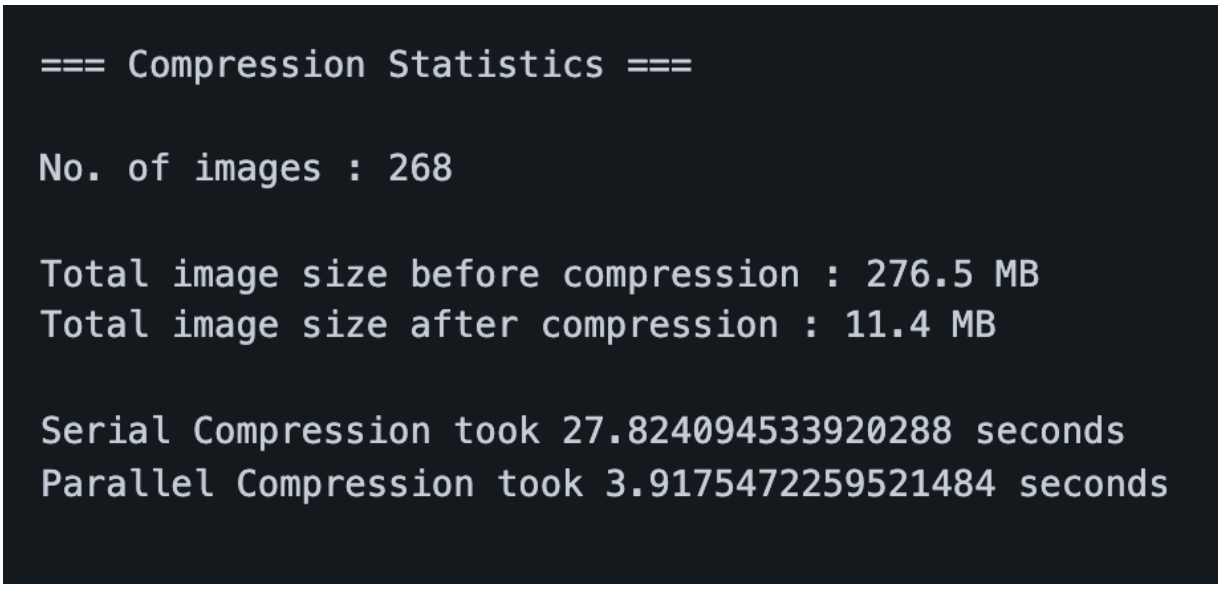
serialProcess(): This function uses a simple for loop and serially calls the compressImage() function to compress a given image from the selected directory. The function also captures the execution time for serial compression of the images. The traditional for-loop iteration goes through the list one by one and performs the function on each item individually.

parallelProcess(): This function uses the multiprocessing python library to send each task to a different processor. We use the multiprocessing module to create a new process for each list item, and trigger each process in one call. We keep track of all processes by making a list and adding each process to it. After creating all the processes, the separate output of each core is combined and displayed together. The function also captures the execution time for parallel compression of the images.

plotGraph(): This function makes use of the data stored in the 2 dictionaries initialized at the top. The dictionary consists of a key pair value where the key holds the number of images processed (n) and the value holds the time it took to run n no. of processes. Using this data the function plots a simple bar graph to compare the results of serial and parallel execution using the matplotlib python library.

Results

A sample set of 268 images were used in the program. Before compression, the combined size of the images amounted to 276.5 MB, after compression the combined image size drastically reduced to 11.4 MB. The above program gave the following output.



```
=== Compression Statistics ===

No. of images : 268

Total image size before compression : 276.5 MB
Total image size after compression : 11.4 MB

Serial Compression took 27.824094533920288 seconds
Parallel Compression took 3.9175472259521484 seconds
```

Fig. 4. Program output

The multi processed code doesn't execute in the same order as serial execution. There's no guarantee that the first process to be created will be the first to start or complete. As a result, multi processed code usually executes in a different order each time it is run, even if each result is always the same. The following table shows the time taken for image compression run serially and parallelly for every 50 consecutive images.

No. Of Images	**Time taken for Serial Processing (s)**	**Time taken for Parallel Processing (s)**
50	5.31	0.66
100	14.03	1.18
150	17.48	1.65
200	22.38	2.19
250	26.67	3.56

Table 1. Time taken for Serial vs Parallel processing for n given images

As we can see, serial processing takes a considerably longer time to finish execution when compared to parallel processing. The total time taken to compress 268 images serially was 27.82 seconds, whereas the total time taken for parallel compression was just 3.91 seconds. Another observation made was that as the number of images increased, parallel processing performed significantly better.

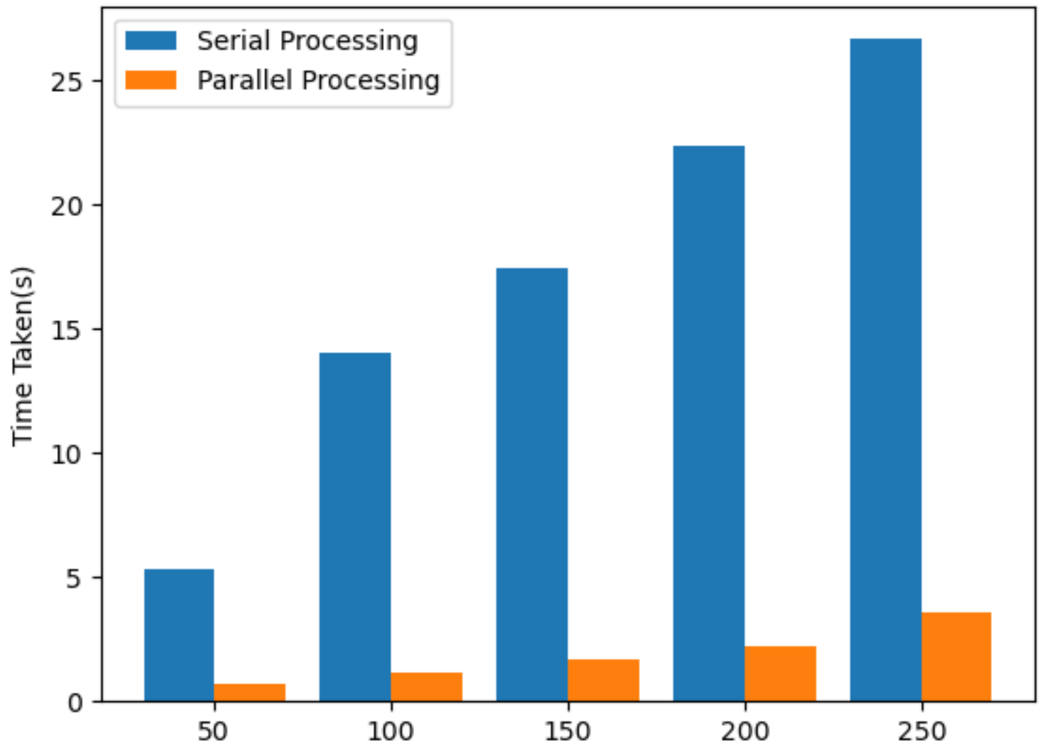


Fig. 5. Serial vs Parallel processing for image compression

The few limitations observed were that the program didn't show any significant difference between serial and parallel processing for dual-core machines but showed a significant change for computers with a higher core count.

Fig. 5. Represents Table 1. In the form of a comparative bar chart. Overall for parallel execution, there was a time decrease of 23.91seconds. Thanks to multiprocessing, we cut down the runtime by 85.95% when compared to the serial runtime. From this, we can conclude parallel processing significantly improves the run time of our program.

References

1. Nagargoje, Priya & Jagtap, Vandana. (2016). Parallel Processing of image in multi-core System.

2. Singh, Sharanjit & Kaur, Parneet & Kaur, Kamaldeep. (2015). Parallel computing in digital image processing. IJARCCE. 183-186. 10.17148/IJARCCE.2015.4139.

3. Liu, Jia & Feld, Dustin & Xue, Yong & Garcke, Jochen & Soddemann, Thomas. (2015). Multicore Processors and Graphics Processing Unit Accelerators for Parallel Retrieval of Aerosol Optical Depth From Satellite Data: Implementation, Performance, and Energy Efficiency. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing. 8. 2306-2317. 10.1109/JSTARS.2015.2438893.

4. Jabar, Moamal. (2021). Image Compression.

5. Inmon, W.H. & Linstedt, Daniel & Levins, Mary. (2019). Parallel Processing. 10.1016/B978-0-12-816916-2.00012-7.

6. https://urban-institute.medium.com/using-multiprocessing-to-make-python-code-faster-23ea5ef996ba

7. https://www.geeksforgeeks.org/how-to-compress-images-using-python-and-pil/

8. https://towardsdatascience.com/10x-faster-parallel-python-without-python-multiprocessing-e5017c93cce1