VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

Duplicate Image Detection using Python Multithreading and Multiprocessing

Prepared by

Jatin Kumar Dhankhar (18BCE0102)

Paurush Batish (18BCE0543)

Parallel and Distributed Systems L5 + L6

Prof. Balamurugan R

Index

Chapter 1- Introduction

- Abstract
- Aim
- Objective
- Motivation

Chapter 2- Literature Review

- Literature survey

Chapter 3- Problem Formulation

- Background
- Existing and Proposed messages
- Description about Dataset and Time Taken for processing
- Algorithms –
  - Serial Execution
  - Multi - Threading
  - Multi – Processors

Chapter 4- Result and Discussion

- Results

Chapter 5-Conclusions and future work

- Conclusion

References

Appendix

Introduction

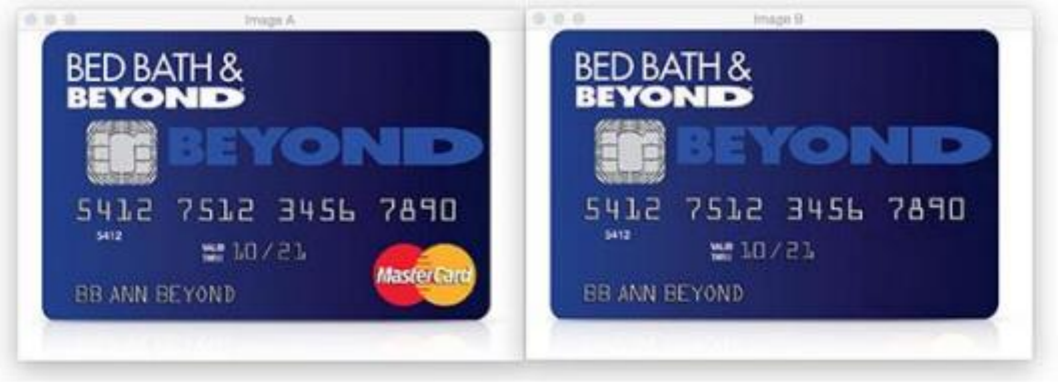


Fig 1. Manually inspecting the difference between two input images

Take a look at these 2 images. If you take a second to study the two credit cards, you’ll notice that the MasterCard logo is present on the left image but has been Photoshopped out from the right image. You may have noticed this difference immediately, or it may have taken you a few seconds. Either way, this demonstrates an important aspect of comparing image 4 differences — sometimes image differences are subtle — so subtle that the naked eye struggles to immediately comprehend the difference. Computing image difference is important for various reasons one of which is phishing. Attackers can manipulate images ever-so-slightly to trick unsuspecting users who don’t validate the URL into thinking they are logging into their banking website — only to later find out that it was a scam.

No doubt you have numerous images on your computer? The trouble with having lots of pictures is that you tend to collect duplicates along the way. It would be a wise thing to manage space efficiently.

Finding similarly pictures and duplicate photos can become an overwhelming project. This is where duplicate photo finders come in. Remember that you should always backup all files before doing any deletion, which is a good practice to take regularly, for example to a DVD or an external hard drive Usually the time taken by such algorithms rise exponentially and thus when put in a serial order, execution is very tough.

Platforms used –

- PyCharm

PyCharm is an integrated development environment (IDE) used in computer programming, specifically for the Python language. It is developed by the Czech company JetBrains. It provides code analysis, a graphical debugger, an integrated unit tester, integration with version control systems (VCSes), and supports web development with Django as well as Data Science with Anaconda. PyCharm is cross-platform, with Windows, macOS and Linux versions. The Community Edition is released under the Apache License, and there is also Professional Edition with extra features – released under a proprietary license.

- Visual Studio Code

Visual Studio Code is a free source-code editor made by Microsoft for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git. Users can change the theme, keyboard shortcuts, preferences, and install extensions that add additional functionality. The source code is free and open-source, released under the permissive MIT License. The compiled binaries are freeware for any use.

Python3

- OpenCV

OpenCV (Open Source Computer Vision Library) is a library of programming functions mainly aimed at real-time computer vision. Originally developed by Intel, it was later supported by Willow Garage then It sees (which was later acquired by Intel). The library is cross-platform and free for use under the open-source BSD license.

- Multiprocessing

multiprocessing is a package that supports spawning processes using an API similar to the threading module. The multiprocessing package offers both local and remote concurrency, effectively sidestepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows the programmer to fully leverage multiple processors on a given machine. It runs on both Unix and Windows. The multiprocessing module also introduces APIs which do not have analogs in the threading module. A prime example of this is the Pool object which offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data parallelism). The following example demonstrates the common practice of defining such functions in a module so that child processes can successfully import that module.

- Threading

This module constructs higher-level threading interfaces on top of the lower level _thread module. This module provides low-level primitives for working with multiple threads (also called lightweight processes or tasks) — multiple threads of control sharing their global data space. For synchronization, simple locks (also called mutexes or binary semaphores) are provided. The threading module provides an easier to use and higher-level threading API built on top of this module.

Abstract

Detecting duplicate images from huge number of images is a tedious task which can be automated to save time and removing duplicate data to save space.

As our usage of mobile grows, the unnecessary duplicate photo and picture files grows in device randomly, ideally every folder of the phone. The duplicate pictures/photos occupy the lots of phone

memory and also reduce the operating speed of phone. Manually it is difficult to find and remove them.

We present the duplicate images detection using parallel processing from which you can scan through your entire phone and finds the duplicate image/picture/photo files for you. You can make judgement yourself if you want to delete by selecting them.

At the end we will also analyze the compute time and power consumed of processing on multiple core v/s single core using threads and provide benchmarks for both along with graphical representations.

Aim

Detecting Duplicate images from a large sample of given images in lesser time compared to Sequential Single Core Execution

Analysis of Processing on Multiple Core v/s Single Core using Threads

Objective

The main aim for us to make this project is that we want to make a one stop platform to detect duplicate images from a large sample of given images in lesser time. The reason we are implementing it with multiple cores is that we want to compare it to sequential single core execution and write about our findings. As our usage of mobile grows, the unnecessary duplicate photo and picture files grows in device randomly, ideally every folder of the phone. The duplicate pictures/photos occupy the lots of phone memory and also reduce the operating speed of phone. Manually it is difficult to find and remove them. We present the duplicate images detection using parallel processing from which you can scan through your entire phone and finds the duplicate image/picture/photo files for you. You can make judgement yourself if you want to delete by selecting them. At the end we will also analyse the compute time and power consumed of processing on multiple core v/s single core using threads and provide benchmarks for both along with graphical representations.

Motivation

Jatin had a recent trip to Pondi, He asks from his friends to upload all clicked images on google drive.

Now here comes the tedious problem, Jatin is a bit late in making his this scheme and his friends have already exchanged images among them but still scattered and few only with all.

Now he needed to get these images, and thus many duplicate images to be deleted [seems like Jatin likes to keep things clean and untidy]. And here he makes a google search: “How to Remove Duplicate Images”.

Now the google search brings in some results, where one result is about Duplicate Cleaner. But Duplicate Cleaner free plan has some limitations, limited speed in free mode, so we thought why not to make our own Duplicate Image Cleaner Pro using Parallelism Fundamentals.

That must be a long motivation story if think about it that’s the story with almost anyone who has gone on a group trip, that’s facts.

In conclusion: We end up with idea for our PDC Project

Literature survey

In [1] “Image quality assessment: from error visibility to structural similarity. IEEE

transactions on image processing” the authors Wang, Z., Bovik, A. C., Sheikh, H.

R., & Simoncelli, E. P. develop a method for assessing perceptual image quality

traditionally attempted to quantify the visibility of errors (differences) between a

distorted image and a reference image using a variety of known properties of the

human visual system. Under the assumption that human visual perception is highly

adapted for extracting structural information from a scene, we introduce an alternative complementary framework for quality assessment based on the degradation of structural information. As a specific example of this concept, we develop a Structural Similarity Index and demonstrate its promise through a set of intuitive examples, as well as comparison to both subjective ratings and state-ofthe-art objective methods on a database of images compressed with JPEG and JPEG2000

In [3] “Fast image inpainting using similarity of subspace method” the authors Hosoi, T., Kobayashi, K., Ito, K., & Aoki, T. propose a model for Image inpainting is a technique for estimating missing pixel values in an image by using the pixel value information obtained from neighbor pixels of a missing pixel or the prior knowledge derived from learning the object class. In this paper, we propose a fast and accurate image inpainting method using similarity of the subspace. The proposed method generates the subspace from many images related to the object class in the learning step and estimates the missing pixel values of the input image belonging to the same object class so as to maximize the similarity between the input image and the subspace in the inpainting step. Through a set of experiments, we demonstrate that the proposed method exhibits excellent performance in terms of both inpainting accuracy and computation time compared with conventional algorithms.

In [4] “Image similarity using Fourier transform” the authors Narayanan, S., & Thirivikraman, P. K. use a similarity measure for images based on values from their respective Fourier Transforms is proposed for image registration. The approach uses image content to generate signatures and is not based on image annotation and therefore does not require human assistance. It uses both, the real and complex components of the FFT to compute the final rank for measuring similarity. Any robust approach must accurately represent all objects in an image and depending on the size of the image data set, diverse techniques may need to be followed. This paper discusses implementation of a similarity rating scheme through the Open CV library and introduces a metric for comparison, carried out by considering Intersection bounds of a covariance matrix of two compared images with normalized values of the Magnitude and Phase spectrum. Sample results on a test collection are given along with data using existing methods of image histogram comparison. Results have shown that this method is particularly advantageous in images with varying degrees of lighting.

In [6] “Image Edge Detection Based On Opencv” the authors Xie, G., & Lu, W. introduce a method of image edge detection to determine the exact number of the copper core in the tiny wire based on OpenCV with rich computer vision and image processing algorithms and functions. Firstly, we use high-resolution camera to take picture of the internal structure of the wire. Secondly, we use OpenCV image processing functions to implement image preprocessing. Thirdly we use morphological opening and closing operations to segment image because of their blur image edges. Finally the exact number of copper core can be clearly distinguished through contour tracking. By using of Borland C++ Builder 6.0, experimental results show that OpenCV based image edge detection methods are simple, high code integration, and high image edge positioning accuracy.

In [7] “Parallel astronomical data processing with Python: Recipes for multicore machines” the authors Singh, N., Browne, L. M., & Butler, R. propose parallel processing recipes for multicore machines for astronomical data processing. The target audience is astronomers who use Python as their preferred scripting language and who may be using PyRAF/IRAF for data processing. Three problems of varied complexity were benchmarked on three different types of multicore processors to demonstrate the benefits, in terms of execution time, of parallelizing data processing tasks. The native multiprocessing module available in Python makes it a relatively trivial task to implement the parallel code. We have also compared the three multiprocessing approaches—Pool/Map, Process/Queue and Parallel Python.

Our test codes are freely available and can be downloaded from our website.

In [9] “Composable multi-threading for Python libraries.” the author Malakhov, A.

tells that python is popular among numeric communities that value it for easy to use

number crunching modules like [NumPy], [SciPy], [Dask], [Numba], and many

others. These modules often use multi-threading for efficient multi-core parallelism

in order to utilize all the available CPU cores. Nevertheless, their threads can

interfere with each other leading to overhead and inefficiency if used together in

one application. The loss of performance can be prevented if all the multi-threaded

parties are coordinated. This paper describes usage of Intel® Threading Building

Blocks (Intel® TBB), an open-source cross-platform library for multi-core

parallelism [TBB], as the composability layer for Python modules. It helps to

unlock additional performance for numeric applications on multi-core systems

.

In [10] “PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms” the authors Meier, L., Honegger, D., & Pollefeys, M. present a novel, deeply embedded robotics middleware and programming environment. It uses a multithreaded, publish-subscribe design pattern and provides a Unix-like software interface for micro controller applications. We improve over the state of the art in deeply embedded open source systems by providing a modular and standards-oriented platform. Our system architecture is centered around a publish-subscribe object request broker on top of a POSIX application programming interface. This allows to reuse common Unix knowledge and experience, including a bash-like shell. We demonstrate with a vertical takeoff and landing (VTOL) use case that the system modularity is well suited for novel and experimental vehicle platforms. We also show how the system architecture allows a direct interface to ROS and to run individual processes either as native ROS nodes

on Linux or nodes on the micro controller, maximizing interoperability. Our

microcontroller-based execution environment has substantially lower latency and

better hardware connectivity than a typical Robotics Linux system and is therefore

well suited for fast, high rate control tasks.

Background

Let’s pretend that we have a huge dataset of stamp images. And we want to take two arbitrary stamp images and compare them to determine if they are identical, or near identical in some way.

In general, we can accomplish this in two ways.

- The first method is to use locality sensitive hashing, which I’ll cover in a

later blog post.

- The second method is to use algorithms such as Mean Squared Error (MSE)

or the Structural Similarity Index (SSIM).

We can use Python to compare two images using Mean Squared Error and

Structural Similarity Index.

Let’s take a look at the Mean Squared error equation:

In order to remedy some of the issues associated with MSE for image comparison,

we have the Structural Similarity Index, developed by Wang et al.:

$$MSE = \frac{1}{m\,n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2$$

Using this method, we were able to easily determine if two images were identical

or had differences due to slight image manipulations, compression artifacts, or

purposeful tampering.

We are going to extend the SSIM approach so that we can visualize the differences

between images using OpenCV and Python. Specifically, we’ll be drawing

bounding boxes around regions in the two input images that differ.

In order to compute the difference between two images we’ll be utilizing the

Structural Similarity Index, first introduced by Wang et al. in their 2004 paper,

$$\mathrm{SSIM}(x,y)=\frac{(2\mu_x\mu_y+c_1)(2\sigma_{xy}+c_2)}{(\mu_x^2+\mu_y^2+c_1)(\sigma_x^2+\sigma_y^2+c_2)}$$

Image Quality Assessment: From Error Visibility to Structural Similarity. This

method is already implemented in the scikit-image library for image processing.

The trick is to learn how we can determine exactly where, in terms of (x, y)-

coordinate location, the image differences are.

To accomplish this, we'll first need to make sure our system has Python, OpenCV,

scikit-image, and imutils.

Existing and proposed methods

Existing Solutions →

- Duplicate Photos Fixer Pro
- Duplicate Photo Finder

- Anti-Twin
- VisiPics
- Similar Image Search
- Awesome Duplicate Photo Finder

Proposed Solution →

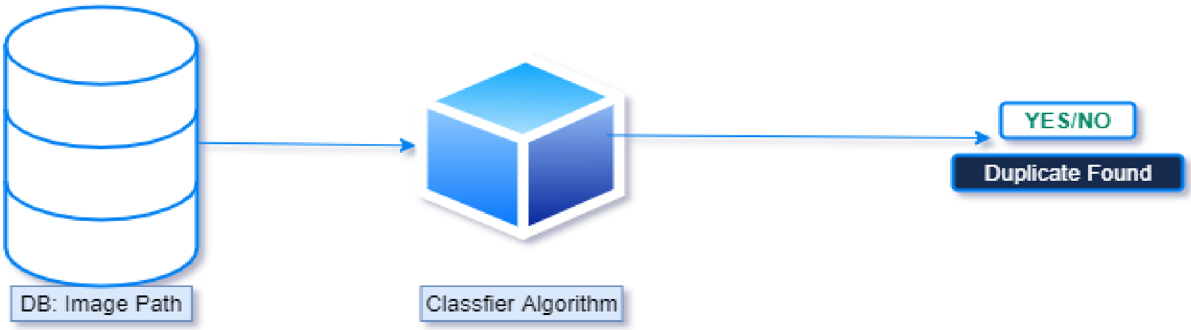
- An open source Python package to remove duplicate Images

Modules used →

- import cv2
- import NumPy as np
- import glob
- import time
- import multiprocessing
- import threading
- import sys

Algorithms

➢ Serial execution



In computer science, a sequential algorithm or serial algorithm is an algorithm that is executed sequentially – once through, from start to finish, without other processing executing – as opposed to concurrently or in parallel. The term is primarily used to contrast with concurrent algorithm or parallel algorithm; most standard computer algorithms are sequential algorithms, and not specifically identified as such, as sequential is a background assumption. Concurrency and parallelism are in general distinct concepts, but they often overlap – many distributed algorithms are both concurrent and parallel – and thus "sequential" is used to contrast with both, without distinguishing which one. If these need to be distinguished, the opposing pairs sequential/concurrent and serial/parallel may be used. "Sequential algorithm" may also refer specifically to an algorithm for decoding a convolutional code.

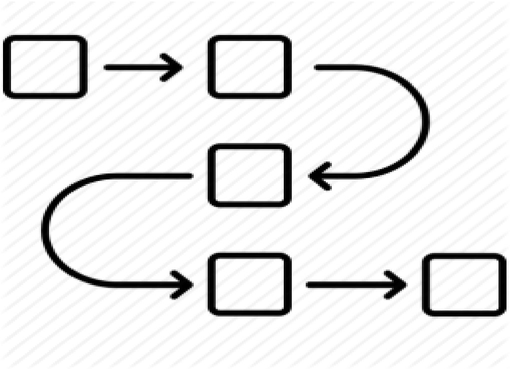
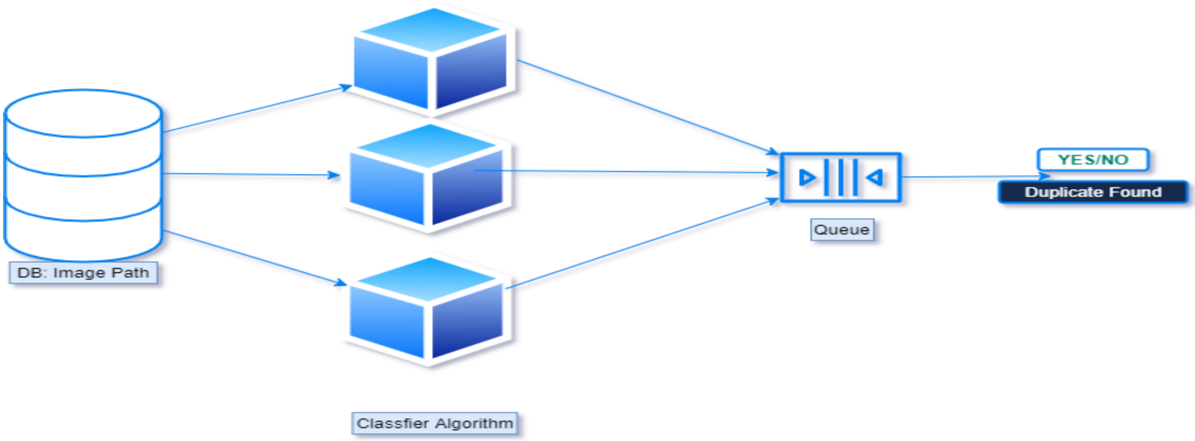


Fig 2.1 Sequential Algorithm

➢ Parallel execution



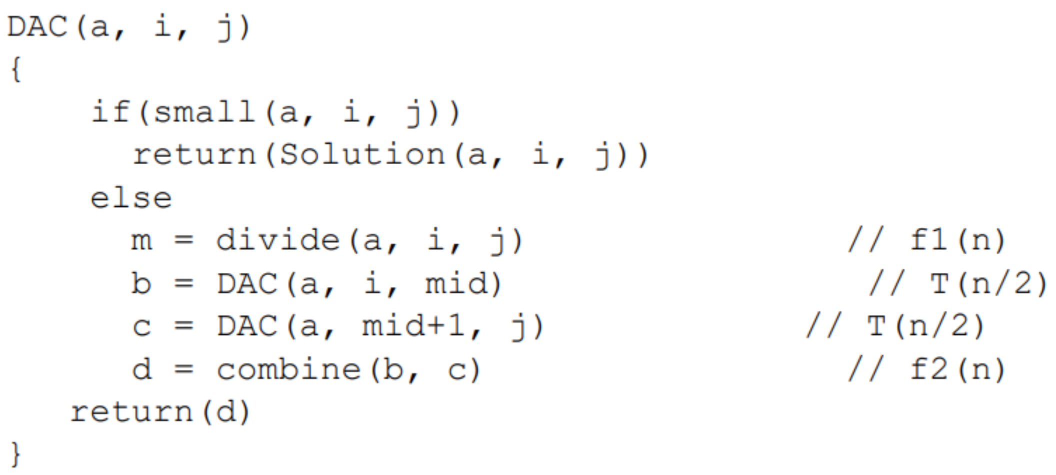
Here divide and conquer strategy is used in parallel programs.

Divide and Conquer

This technique can be divided into the following three parts:

- Divide: This involves dividing the problem into some sub problem.
- Conquer: Sub problem by calling recursively until sub problem solved.
- Combine: The Sub problem Solved so that we will get find problem solution.

Divide and Conquer algorithm:



```
DAC(a, i, j)
{
    if(small(a, i, j))
      return(Solution(a, i, j))
    else
      m = divide(a, i, j)                // f1(n)
      b = DAC(a, i, mid)                  // T(n/2)
      c = DAC(a, mid+1, j)              // T(n/2)
      d = combine(b, c)                  // f2(n)
   return(d)
}
```

Recurrence Relation - This is recurrence relation for above program.

```
O(1) if n is small
T(n) =     f1(n) + 2T(n/2) + f2(n)
```

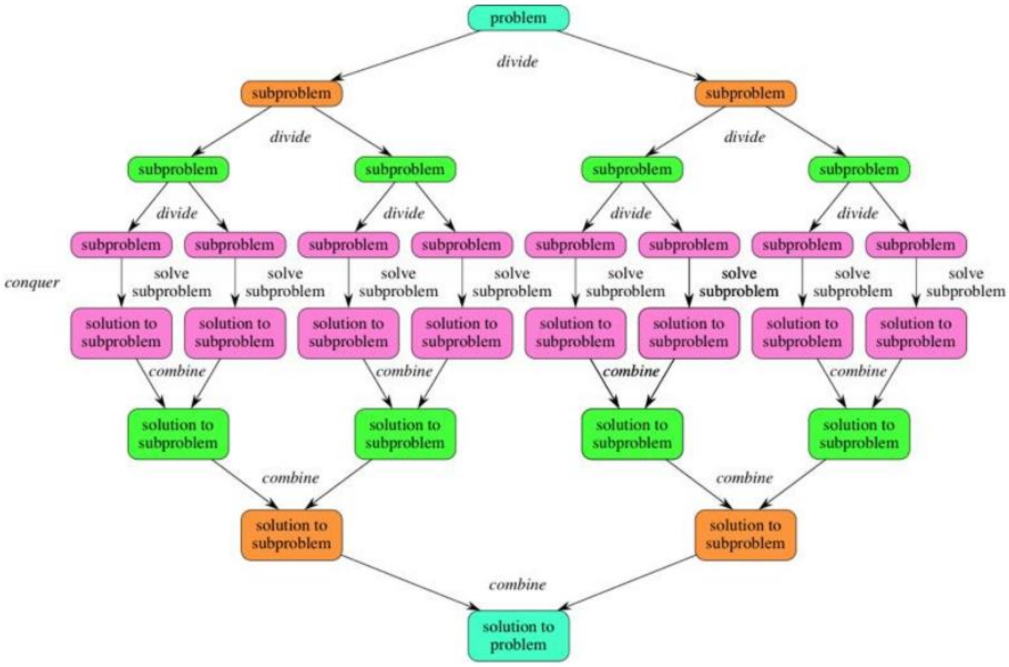


FIg 4 Divide and Conquer Algorithm

Basic Algorithm

```
def find_duplicates(image_):

    duplicates = ''

     try:

        image_to_compare = cv2.imread(image_)

        if original.shape == image_to_compare.shape:


            difference = cv2.subtract(original, image_to_compare)
```

```
                b, g, r = cv2.split(difference)

                if cv2.countNonZero(b) == 0 and cv2.countNonZero(g) == 0 and cv2.countNonZero(r) == 0:

                    duplicates = image_

                    return duplicates

        except Exception as e:
            pass
```

Explanation of the Algorithm

We have created a working Script to detect duplicate images which identifies duplicate images recursively using multiprocessing library of python

In order to compute the difference between two images we'll be utilizing the Structural Similarity Index, first introduced by Wang et al. in their 2004 paper, Image Quality Assessment: From Error Visibility to Structural Similarity.

This method is already implemented in the scikit-image library for image processing.

The trick is to learn how we can determine exactly where, in terms of (x, y)- coordinate location, the image differences are.

To accomplish this, we'll first need to make sure our system has Python, OpenCV, scikit-image, and imutils.

Serial Algorithm Vs Parallel Algorithm

Search for Images Recursively from a given starting directory

Sequential –

- Compare two images by their RGB values and find image similarity using cv2 library sequentially

Multiple-Processors

- Compare two images by their RGB values and find image similarity using cv2 library by spawning processes using multiprocessing library API

Threading –

- Compare two images by their RGB values and find image similarity using cv2 library by passing each comparison to a new thread

Multiprocessing vs Multithreading

Multiprocessing adds CPUs to increase computing power. Multiple processes are executed concurrently. Creation of a process is time-consuming and resource intensive. Multiprocessing can be symmetric or asymmetric. The multiprocessing library in Python uses separate memory space, multiple CPU cores, bypasses GIL limitations in CPython, child processes are killable (ex. function calls in program) and is much easier to use. Some caveats of the module are a larger memory footprint and IPC's a little more complicated with more overhead.	Multithreading creates multiple threads of a single process to increase computing power. Multiple threads of a single process are executed concurrently. Creation of a thread is economical in both sense time and resource. The multithreading library is lightweight, shares memory, responsible for responsive UI and is used well for I/O bound applications. The module isn't killable and is subject to the GIL. Multiple threads live in the same process in the same space, each thread will do a specific task, have its own code, own stack memory, instruction

	pointer, and share heap memory. ✚ If a thread has a memory leak it can damage the other threads and parent process.

How was threading used

Search for Images Recursively from a given starting directory

Compare two images by their RGB values and find image similarity using cv2

library by passing each comparison to a new thread

Python on the CPython interpreter does not support true multi-core execution via

multithreading. However, Python DOES have a Threading library. So what is the

benefit of using the library if we (supposedly) cannot make use of multiple cores?

Many programs, particularly those relating to network programming or data

input/output (I/O) are often network-bound or I/O bound. This means that the

Python interpreter is awaiting the result of a function call that is manipulating data

from a "remote" source such as a network address or hard disk. Such access is far

slower than reading from local memory or a CPU-cache.

Hence, one means of speeding up such code if many data sources are being accessed

is to generate a thread for each data item needing to be accessed.

For example, consider a Python code that is scraping many web URLs. Given that

each URL will have an associated download time well in excess of the CPU

processing capability of the computer, a single-threaded implementation will be

significantly I/O bound.

By adding a new thread for each download resource, the code can download

multiple data sources in parallel and combine the results at the end of every

download. This means that each subsequent download is not waiting on the

download of earlier web pages. In this case the program is now bound by the bandwidth limitations of the client/server(s) instead.

However, many financial applications ARE CPU-bound since they are highly numerically intensive. They often involve large-scale numerical linear algebra solutions or random statistical draws, such as in Monte Carlo simulations. Thus as far as Python and the GIL are concerned, there is no benefit to using the Python Threading library for such tasks.

Python offers four possible ways to handle that. First, you can execute functions in parallel using the multiprocessing module. Second, an alternative to processes are threads. Technically, these are lightweight processes, and are outside the scope of this article. For further reading you may have a look at the Python threading module.

Third, you can call external programs using the system() method of the os module, or methods provided by the subprocess module, and collect the results afterwards.

The multiprocessing module covers a nice selection of methods to handle the

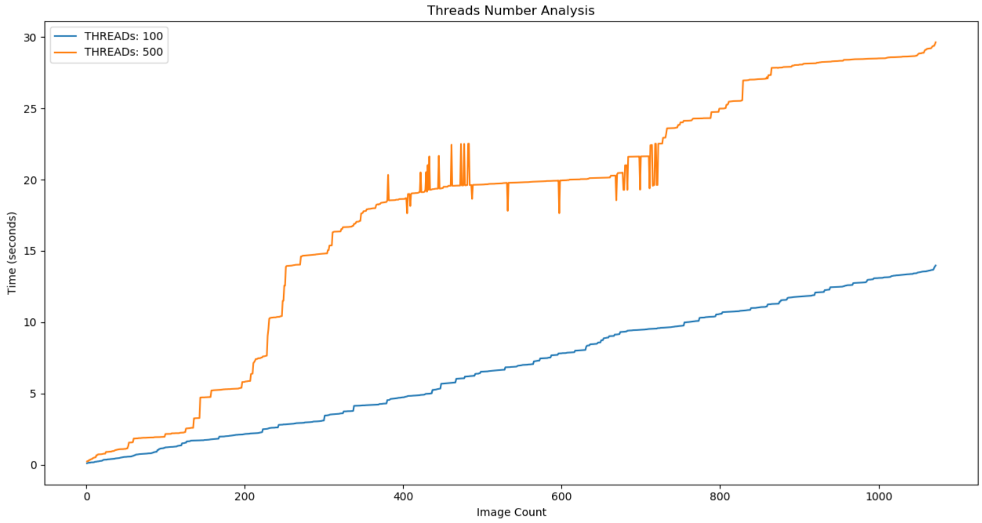
parallel execution of routines. This includes processes, pools of agents, queues, and

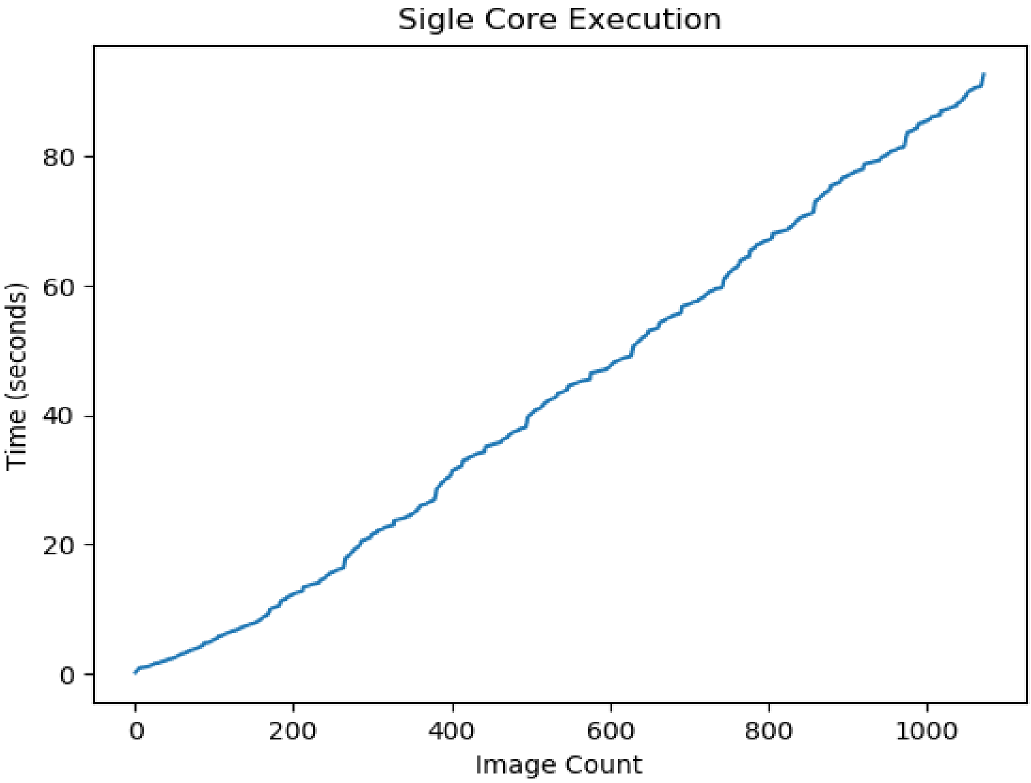
pipes.

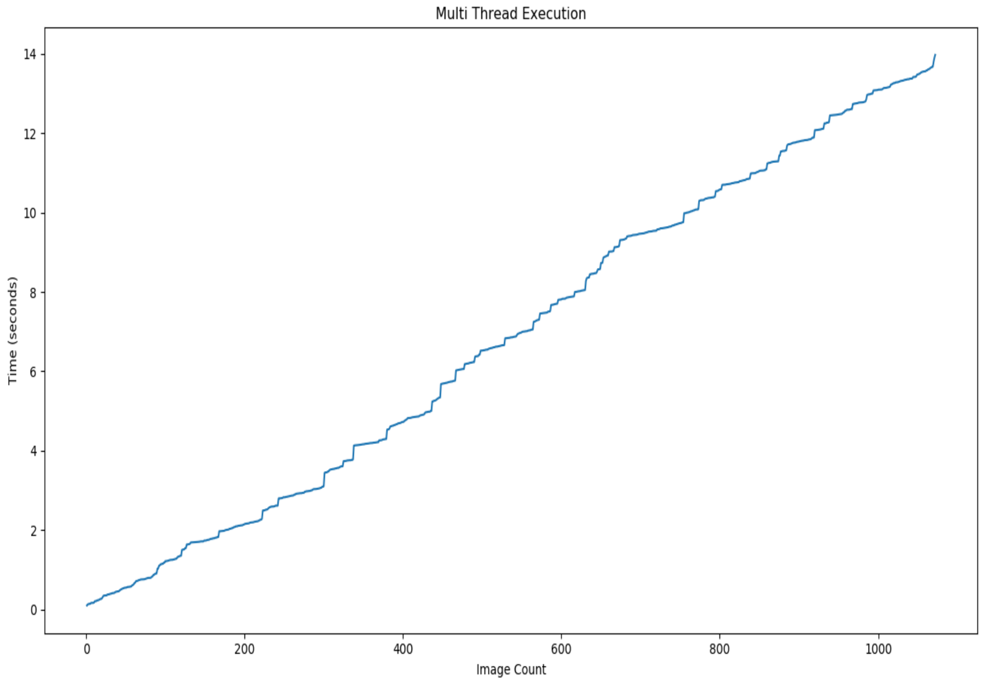
Important Details

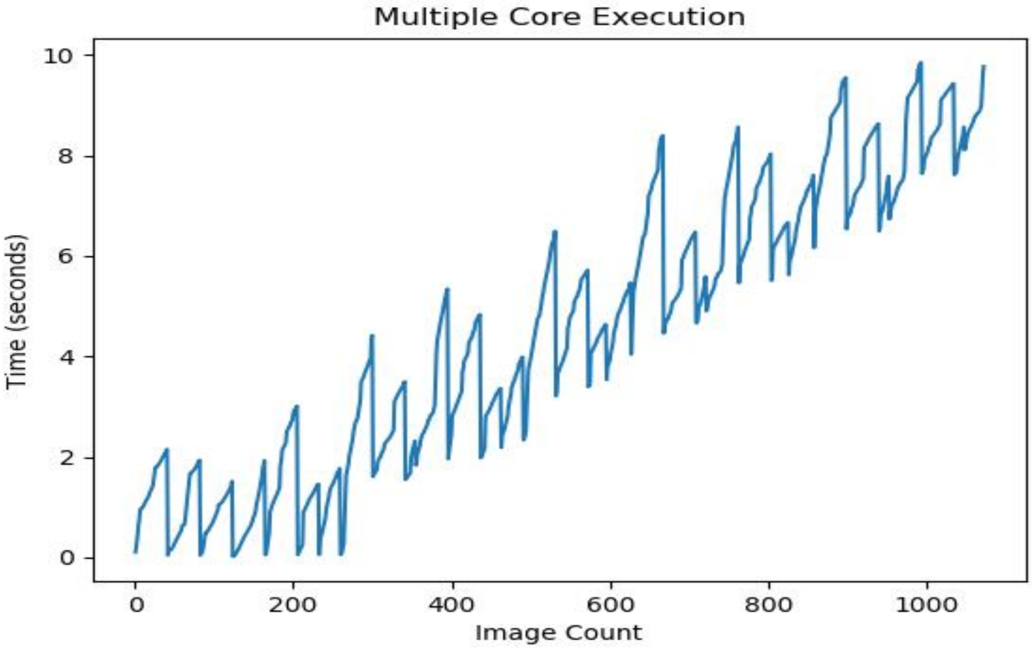
- For Smaller Sample Thread works better but on creation of too many threads then comes context switch in play which acts as a barrier
- But Creating a Thread is a cheap process when creating limited number of threads as they all are sharing same resources and hence quite faster than Multiple Processing at that time.
- Multiple Processors have there own resource bucket, they are GIL safe and can work parallelly because of this reason with increase in data they worked better than threading

Results (Output graphs)

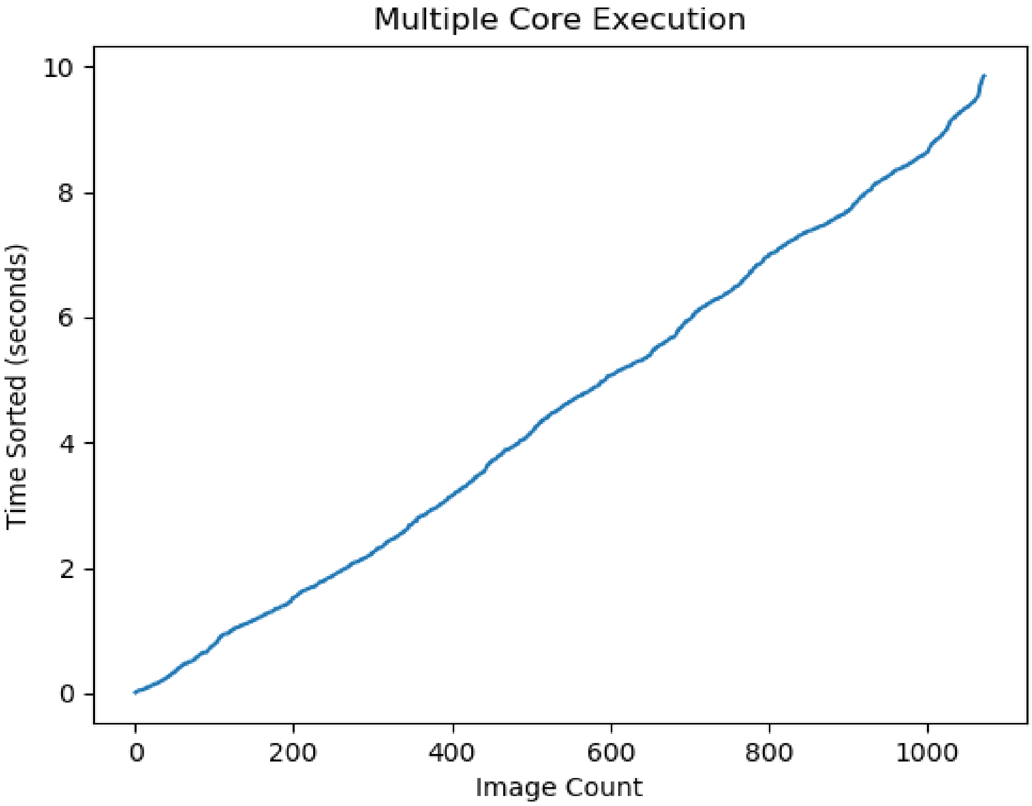


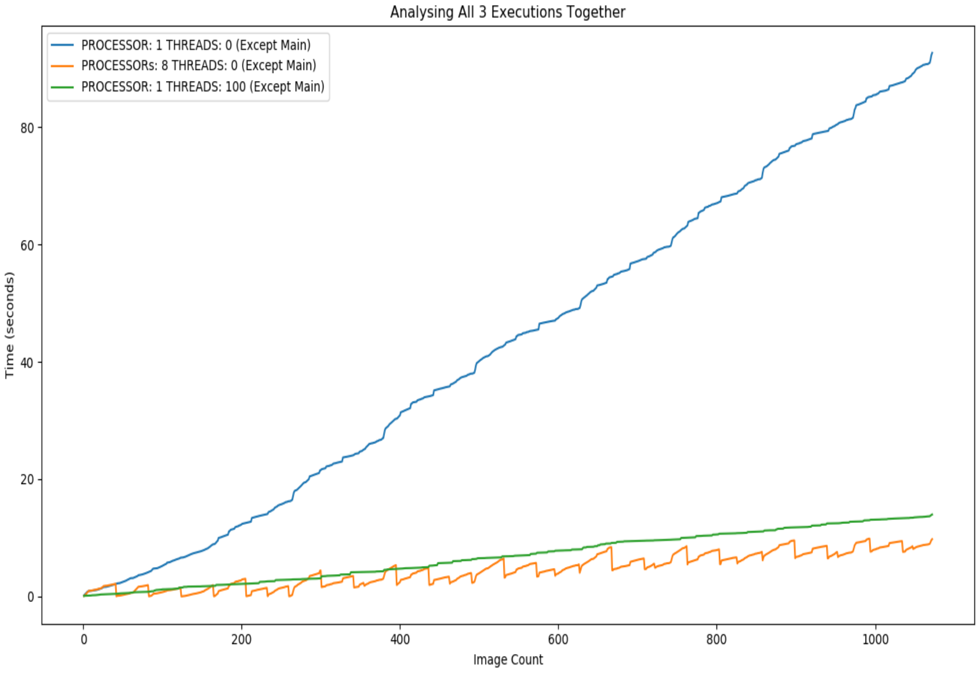






Here we are having this pattern because each core had an equal amount of work divided, now they all merging their chunked worked output together which is creating this wave-like pattern. It is observable this line graph can be broken down into chunks of the group of 8 cores generating output together for their particular work


Multiple Core Execution
Time Sorted (seconds)
10
8
6
4
2
0
0
200
400
600
800
1000
Image Count


Analysing All 3 Executions Together
PROCESSOR: 1 THREADS: 0 (Except Main)
PROCESSORs: 8 THREADS: 0 (Except Main)
PROCESSOR: 1 THREADS: 100 (Except Main)
Time (seconds)
80
60
40
20
0
0
200
400
600
800
1000
Image Count

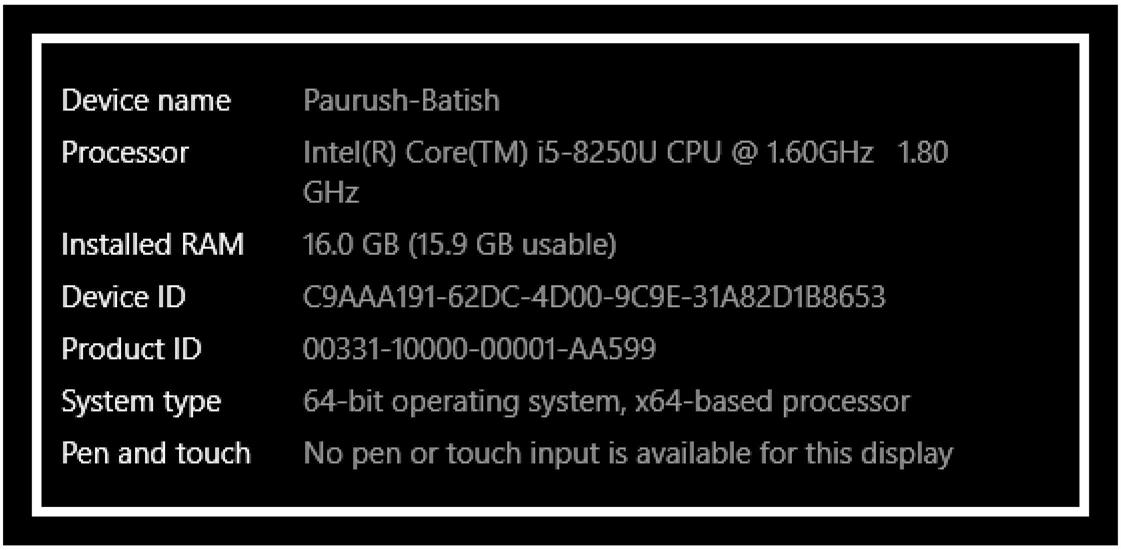
Explanation

We can clearly observe here that the Time Taken for Single Core, Single Threaded application increases exponentially with Time. But for MultiProcessing and MultiThreading, it increases with a little slope value. An addition to the observation is the fact that MultiProcessing Library is beating the thread as data is increased this is because MultiProcessing is resource extensive and so takes higher time to start it's processing but then it processes the data like a charm whereas Thread have interrupts which are continuously degrading the performance of the application and it is not able to perform any better.

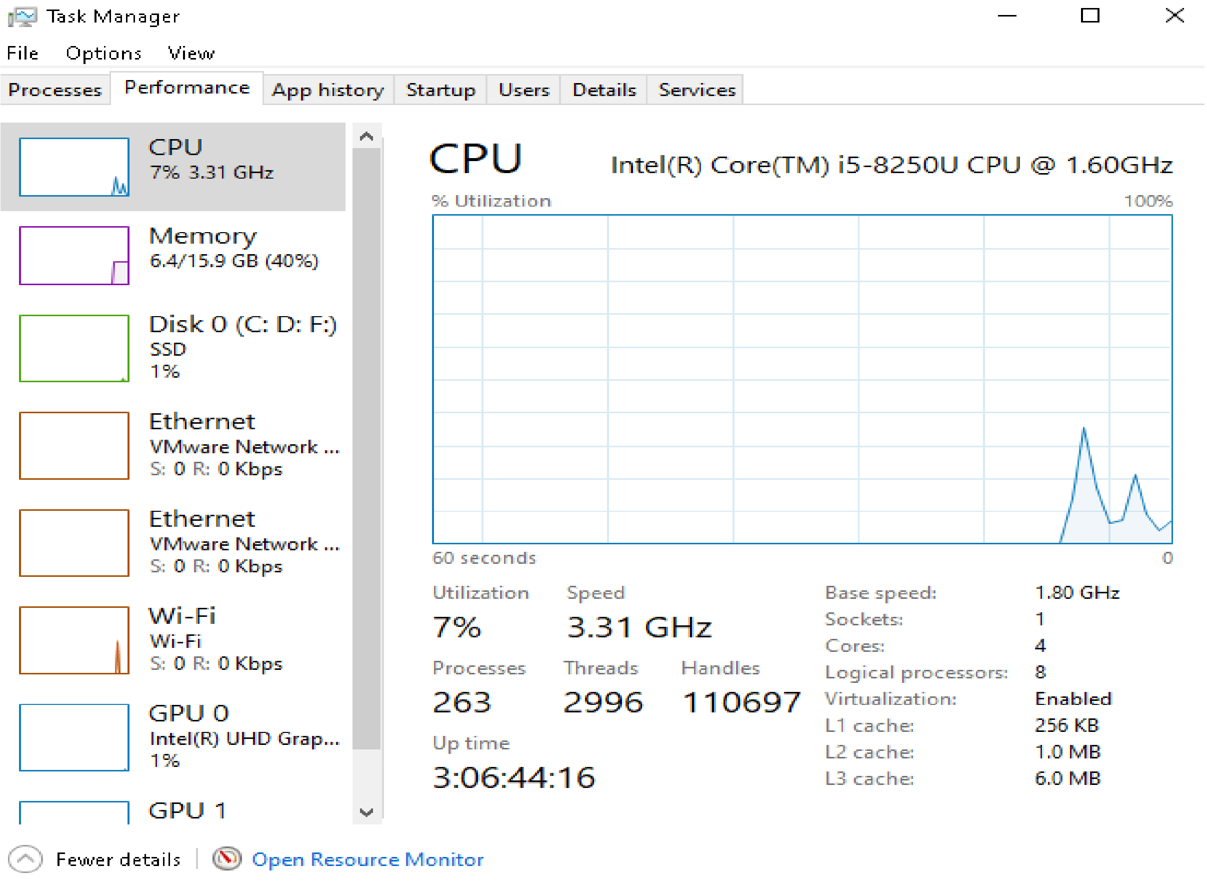
Here we see that serial execution of the processes takes up exponential time to process the code while on the other hand multithreading and multiprocessing both take up very less time for the execution.

Another thing that we found is that multiprocessing works slow at first because generation of threads is faster than generation of processors but later on since every processor has its own GIL so parallelisation is observed in a real-time scenario.

Environment used –



Device name	Paurush-Batish
Processor	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz 1.80 GHz
Installed RAM	16.0 GB (15.9 GB usable)
Device ID	C9AAA191-62DC-4D00-9C9E-31A82D1B8653
Product ID	00331-10000-00001-AA599
System type	64-bit operating system, x64-based processor
Pen and touch	No pen or touch input is available for this display

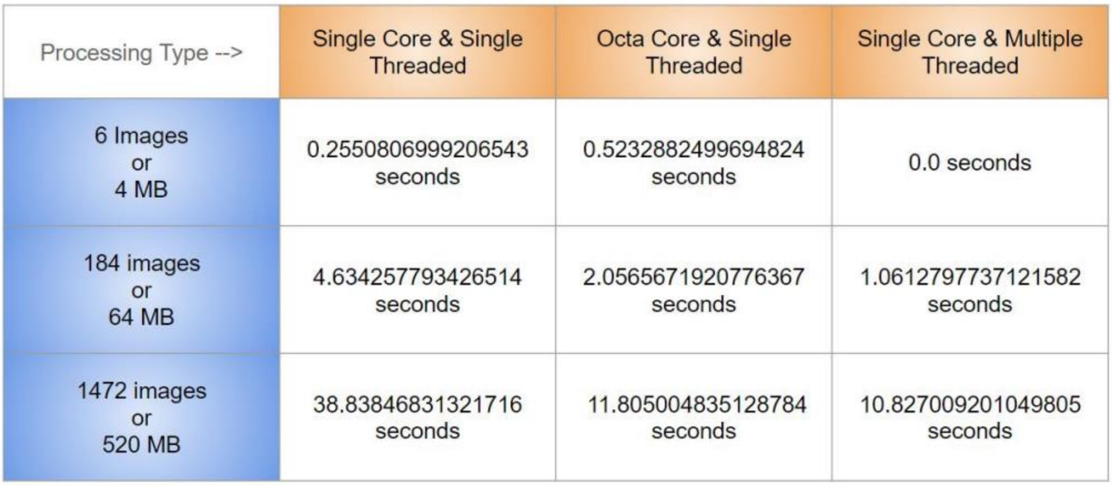


Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz

Utilization 7%
Speed 3.31 GHz
Processes 263
Threads 2996
Handles 110697
Up time 3:06:44:16

Base speed:	1.80 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB

Calculation



Processing Type -->	Single Core & Single Threaded	Octa Core & Single Threaded	Single Core & Multiple Threaded
6 Images or 4 MB	0.2550806999206543 seconds	0.5232882499694824 seconds	0.0 seconds
184 images or 64 MB	4.634257793426514 seconds	2.0565671920776367 seconds	1.0612797737121582 seconds
1472 images or 520 MB	38.83846831321716 seconds	11.805004835128784 seconds	10.827009201049805 seconds

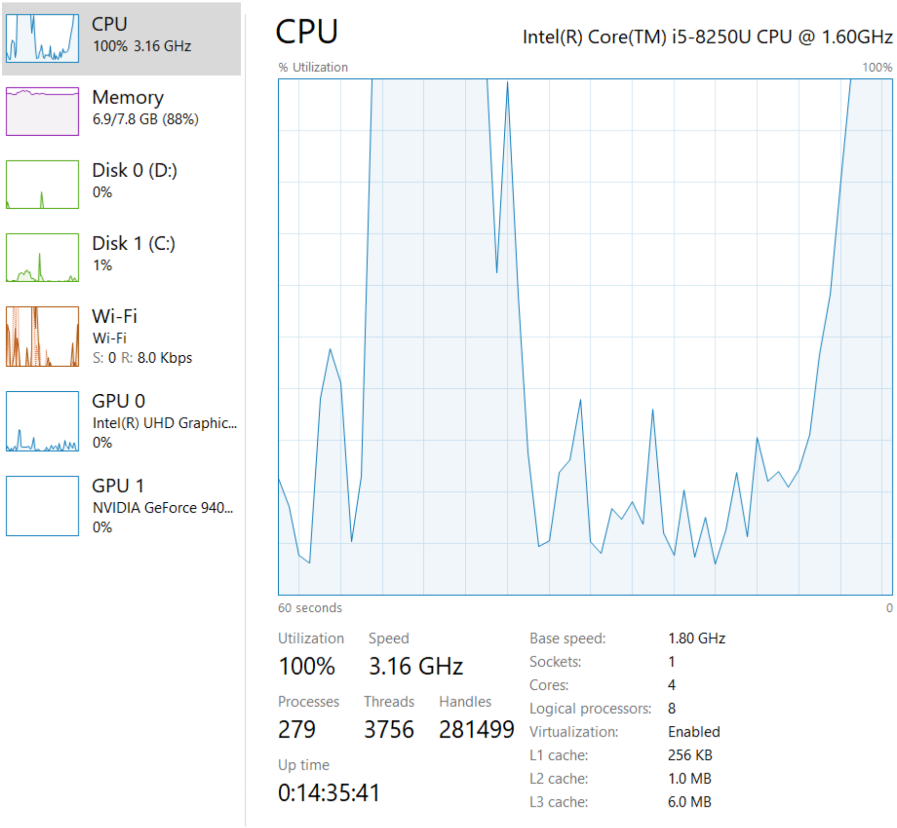
Fig 5. Analysis: Execution Time vs Data Size

Note: Threads are not synchronized, after sync time taken for 520MB is 13.5 sec using 100 Threads

How is CPU influenced

We found that

- CPU Utilization becomes 88% for both Multiprocessing and Multithreading(depends upon the Hardware sometimes a 100%)
- CPU utilization remains lower and around 26% which is normal for Single Processing and Single Threaded application



CPU Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz

Utilization 100%

Speed 3.16 GHz

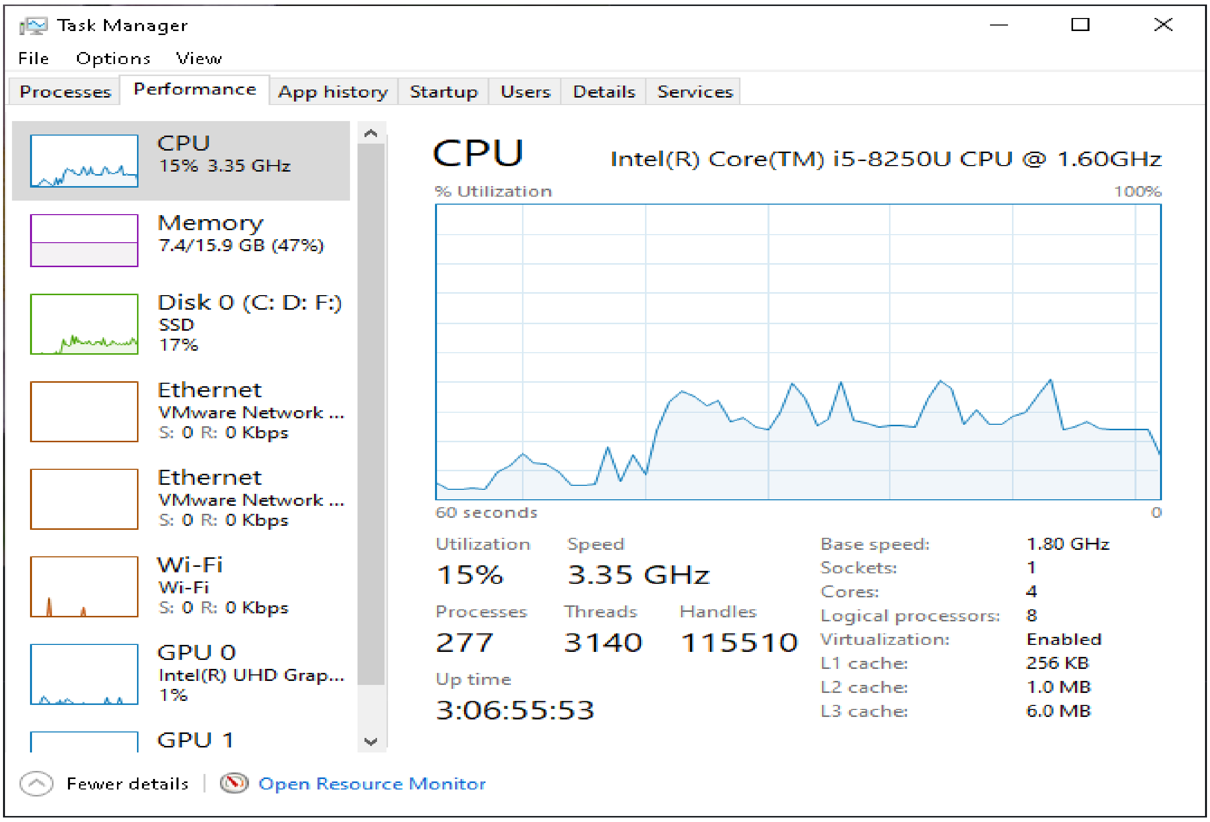
Processes 279

Threads 3756

Handles 281499

Up time 0:14:35:41

Base speed:	1.80 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB



CPU Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz

Utilization 15%

Speed 3.35 GHz

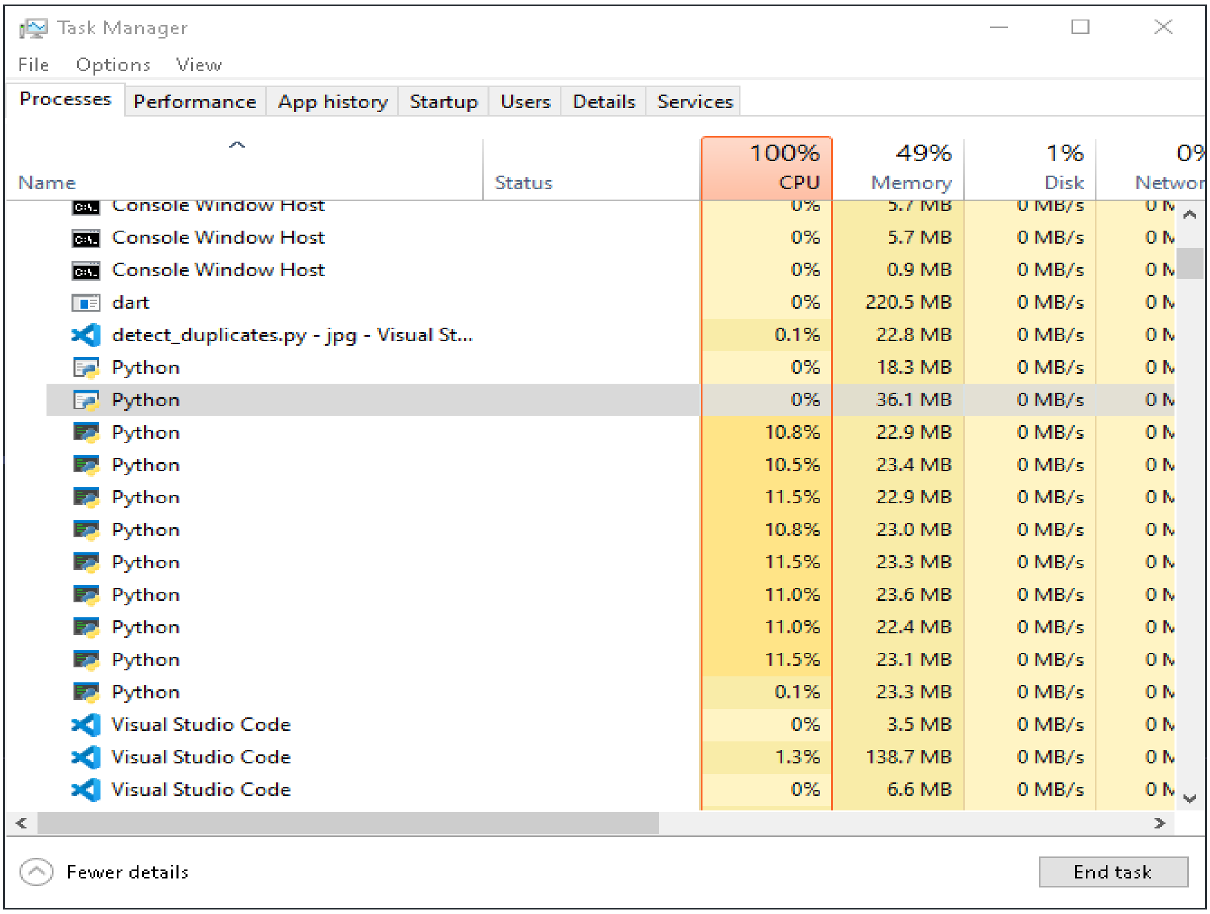
Processes 277

Threads 3140

Handles 115510

Up time 3:06:55:53

Base speed:	1.80 GHz
Sockets:	1
Cores:	4
Logical processors:	8
Virtualization:	Enabled
L1 cache:	256 KB
L2 cache:	1.0 MB
L3 cache:	6.0 MB



Name	Status	100% CPU	49% Memory	1% Disk	0% Networ
Console Window Host		0%	5.7 MB	0 MB/s	0 M
Console Window Host		0%	5.7 MB	0 MB/s	0 M
Console Window Host		0%	0.9 MB	0 MB/s	0 M
dart		0%	220.5 MB	0 MB/s	0 M
detect_duplicates.py - jpg - Visual St...		0.1%	22.8 MB	0 MB/s	0 M
Python		0%	18.3 MB	0 MB/s	0 M
Python		0%	36.1 MB	0 MB/s	0 M
Python		10.8%	22.9 MB	0 MB/s	0 M
Python		10.5%	23.4 MB	0 MB/s	0 M
Python		11.5%	22.9 MB	0 MB/s	0 M
Python		10.8%	23.0 MB	0 MB/s	0 M
Python		11.5%	23.3 MB	0 MB/s	0 M
Python		11.0%	23.6 MB	0 MB/s	0 M
Python		11.0%	22.4 MB	0 MB/s	0 M
Python		11.5%	23.1 MB	0 MB/s	0 M
Python		0.1%	23.3 MB	0 MB/s	0 M
Visual Studio Code		0%	3.5 MB	0 MB/s	0 M
Visual Studio Code		1.3%	138.7 MB	0 MB/s	0 M
Visual Studio Code		0%	6.6 MB	0 MB/s	0 M

When multiprocessing is done then we can see clearly that the number of processors increase rapidly to the no of logical processors in our project

Conclusion

Use of Threading is enough and more suited in Applications like GUI and Networking as it's less resource intensive, Both Multiprocessing and Multithreading have similar performance in I/O bound operations.

Since most online servers provide with single core only so threaded apps are more used but for Research purposes we have need more computation power and parallel execution with less complication and there comes Multiple Processors in to rescue.

Future Scope:

- Added functionality of removing the images as soon as we find it.
- Adding a GUI
- Web Application/Mobile Application
- Scaling the algorithm with time and making it more feasible for even larger datasets

References

- *https://docs.python.org/3/library/multiprocessing.html*
- *https://docs.python.org/3/library/threading.html*
- *https://opencv.org/*
- *[1] Wang, Z., Bovik, A. C., Sheikh, H. R., & Simoncelli, E. P. (2004). Image quality assessment: from error visibility to structural similarity. IEEE transactions on image processing, 13(4), 600-612.*
- *[2] Narayanan, S., & Thirivikraman, P. K. (2015). Image similarity using fourier transform. Journal Impact Factor, 6(2), 29-37.*

- *[3] Xie, G., & Lu, W. (2013). Image Edge Detection Based On Opencv. International Journal of Electronics and Electrical Engineering, 1(2), 104-6.*
- *[4] Singh, N., Browne, L. M., & Butler, R. (2013). Parallel astronomical data processing with Python: Recipes for multicore machines. Astronomy and Computing, 2, 1-10.*
- *[5] Malakhov, A. (2016, July). Composable multi-threading for Python libraries. In Proceedings of the Python in Science Conferences.*
- *[6] Meier, L., Honegger, D., & Pollefeys, M. (2015, May). PX4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In 2015 IEEE international conference on robotics and automation (ICRA) (pp. 6235-6240). IEEE.*

Appendix – (Source code)

```
CLASS DUPLICATEFINDER(THREADING.THREAD):


 DEF _INIT_(SELF, QUEUE, QUEUE2):

 """INITIALIZE THE THREAD"""

 THREADING.THREAD._INIT_(SELF)

 SELF.LOCK = THREADING.LOCK()

 SELF.QUEUE = QUEUE

 SELF.QUEUE2 = QUEUE2


 DEF RUN(SELF):

 # GLOBAL IMAGE_COUNT

 """RUN THE THREAD"""

 WHILE TRUE:

 IMAGE_PATH = SELF.QUEUE.GET()

 SELF.CHECK_DUPLICATE(IMAGE_PATH)

 # SEND A SIGNAL TO THE QUEUE THAT THE JOB IS DONE

 SELF.QUEUE.TASK_DONE()



 DEF CHECK_DUPLICATE(SELF, IMAGE_):

 GLOBAL IMAGE_COUNT

 """PRINTS DUPLICATE IMAGE NAME"""

 TRY:

 IMAGE_TO_COMPARE = CV2.IMREAD(IMAGE_)

 IF ORIGINAL.SHAPE == IMAGE_TO_COMPARE.SHAPE:

 DIFFERENCE = CV2.SUBTRACT(ORIGINAL,

IMAGE_TO_COMPARE)

 B, G, R = CV2.SPLIT(DIFFERENCE)

 IF CV2.COUNTNONZERO(B) == 0 AND CV2.COUNTNONZERO(G)
```

```
== 0 and cv2.countNonZero(r) == 0:
 # print('Duplicate Found')
sift = cv2.xfeatures2d.SIFT_create()
kp_1, desc_1 = sift.detectAndCompute(original,
None)
 kp_2, desc_2 =
sift.detectAndCompute(image_to_compare, None)
 index_params = dict(algorithm=0, trees=5)
search_params = dict()
flann = cv2.FlannBasedMatcher(index_params,
search_params)
 matches = flann.knnMatch(desc_1, desc_2, k=2)
 good_points = []
 for m, n in matches:
 if m.distance < 0.6*n.distance:
 good_points.append(m)
 # Define how similar they are
number_keypoints = 0
if len(kp_1) <= len(kp_2):
 number_keypoints = len(kp_1)
 else:
 number_keypoints = len(kp_2)


 self.queue2.put([image_count, round(time.time() -
start_time, 5)]) # we can skip image count but that's not fair
 self.lock.acquire()
 try:
 image_count += 1
 finally:
 self.lock.release()
 except Exception as e:
```

```
PASS
DEF MAIN(IMAGE_DIR):
"""
RUN THE PROGRAM
"""
QUEUE = QUEUE()
QUEUE2 = QUEUE() # QUEUE TO TAKE PLOTTING DATA
# CREATE A THREAD POOL AND GIVE THEM A QUEUE
FOR I IN RANGE(5000): # I DON'T WON'T MY VIRTUAL MEMORY TO GET
BUNKED UP
T = DUPLICATEFINDER(QUEUE, QUEUE2)
T.SETDAEMON(TRUE)
T.START()
#GIVE THE QUEUE SOME DATA | CONSUMER AND PRODUCER CODE HUH
FOR IMAGE_ IN IMAGE_DIR:
QUEUE.PUT(IMAGE_)


# WAIT FOR THE QUEUE TO FINISH
QUEUE.JOIN()
# QUEUE2.JOIN()
PRINT(LIST(QUEUE2.QUEUE))
```