# PC24/25: Parallel integral image in freethreaded Python

Lorenzo Pasquini
E-mail address
lorenzo.pasquini3@edu.unifi.it

## Abstract

*The integral image is an algorithm which, given a matrix of integers, calculates the integral of the said matrix by doing a cumulative sum of the elements in the matrix. This calculation can be very time–consuming with a sequential approach, but parallelizing it can be a challenge, especially in python where, usually, only IO-bound expression can be computed in parallel. Using the newly dispatched GIL-free version of Python CPU-intensive tasks can be parallelized, thus making this implementation effectively faster.*

## 1. Introduction

The integral Image algorithm outputs an integral matrix from the input one. Let M be the starting matrix and M' the integral one based on M. M' is defined so that $M'_{x,y}$ is calculated by the sum of every item in a sub-matrix of M of size x, y as shown in the figure 1. The formula to obtain M' is so described:

$$M'(x,y) = \sum_{x'=0}^{x} \sum_{y'=0}^{y} M(x',y') \tag{1}$$

It should be noted that this approach can be expensive in terms of both memory usage and time consumption, as parallelizing this many calculations can be of little help, having a big enough dataset. Another approach is to do a cumulative sum in the same matrix, while this method can become difficult to parallelize given the dependencies between the various elements in the matrix, it is easier to program and can better use Python's libraries, furthermore, it is a net gain in free threads, which can be used to spread the dataset between multiple threads. Being that the cumulative sum is defined as:

$$M_{\text{sum}}(x,y) = \sum_{x'=0}^{x} M(x',y) \tag{2}$$

it is clear that by running the sums twice, once for the x and then one for the y of the resulting matrix $M_{\text{sum}}$ we obtain the same result as formula (1).

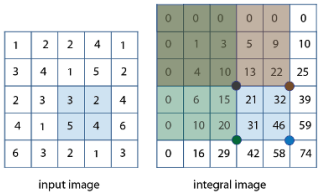


Figure 1. How an integral image is calculated

### 1.1. Dataset and Parallelization

To convey one of the uses of the integral image algorithm, the dataset used constitutes of RBG images, which, once loaded into the program are computed as three matrices each, one for the value of red in that pixel, one for the value of green and one for the value of blue. These are saved in a single 3D matrix for each image as shown in figure 2. Then for each color the integral image is calculated and printed. While the original project used to calculate each matrix in a single thread this implementation used to add an heavy overhead to the program, slowing the sum calculation. In the actual program, each thread computes an image and prints it's own integral matrix.

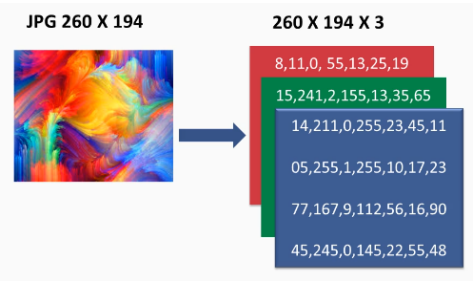


Figure 2. RGB Matrices of an image

## 2. Implementation

In this section will be explained how the algorithm was implemented, what choices were made and which other choices were discarded and the motivations behind those decisions. It will be also explained how the differences between the GIL and the freethreaded version of python can matter when developing or running a program.

### 2.1. To GIL or not to GIL

The Global Interpreter Lock has always been one of the biggest limitations in python, the interpreter could only run one CPU-bound operation at a time, unless it was in different, dedicated processes, in short, multithreading was used only to compute I/O operations, making parallel python programs hard to find and to code. In the lastest version of python things changed, the GIL was deactivated in an experimental version of the 3.13, release, while effectively permitting parallelization of CPU-bound operations, this new version has some serious limitations. First of all the new version introduced some serious compatibility issues, numpy and pandas were initially completely unusable and some other libraries (notoriously, openGL and pandas still) are not even installable in a free-threaded environment. Moreover, this changes created an immortalization problem, where function object, module descriptors and classes can't be deallocated interpreter while the interpreter is running. Lastly, the single-threaded performances are actually worse if compared with the normal version with GIL, this due to the disabling of the specialized adaptive interpreter, an interpreter that heavily specializes the code it runs. This change can increment the overhead up to 40% in single threaded instances.

### 2.2. Reading the data and computing it

The program was implemented using Pillow to read the images from the *images* folder and convert them into image arrays composed of the RGB colors, then converted into numpy arrays, creating the tridimensional matrix, the parallel and sequential implementation depend on the number of threads used. Initially the threads from the Threading library were used, mainly because they offer more control on how the threads are started (and stopped) and with the barriers threading offer those threads could be controlled in be most precise way, moreover threading is one of the most compatible libraries with the freethreaded version of python (so much so that the usage of threading.lock() is also suggested in the documentation of the version). In the end however, the threadpool form concurrent.futures was revealed the most legible and concurrent.future's way of running threads was not burdensome to the development so the code was adapted to this library. The code to actually compute the matrices was initially hand-written and parallelized, but this created way more threads than the machine could handle and the payoff was little, so the computation of each matrix was left to numpy's cumsum function, that automatizes the cumulative sum of an array it was then coded so that for each color the thread dedicated to the image would run a cumulative sum for the rows and then for the columns. The results would later be printed in matrix style but, since the data is a massive read, it was decided to just print the completion message for each thread before terminating it.

## 3. Execution

This section will be dedicated to the performance part of the program, analyzing the speedup with various threads, differences between GIL and freethreaded version and how the number of images to compute actually matters in the various cases. The writing and testing of the code were made on Visual Studio Code version 1.99.3, Windows 11 OS on a laptop with a 13th Gen Intel core I7-1360P processor running 4 performance cores and 8 efficient cores and a 16 thread limit and Python 3.13.3 experimental version for the multithreaded runs, 3.13.3 for the singlethreaded executions.

### 3.1. Differences between GIL and single-threaded

When the code was run with a single thread and when it was run sequentially with the gil results were different, as mentioned, this is due the extra overhead given by the deactivation of the specialized adaptive interpreter, in fact, when run on the full 64 images, the freethreaded interpreter (run sequentially with a single thread) needed 28.83 seconds where the GIL locked interpreter needed 25.47, even after some more tests this difference stands, proving that in fact the extra overhead is caused by the interpreter and not some noise.

### 3.2. Differences between datasets

Analyzing the datasets used can show how well the parallelization went. As shown in figure 3, there are three lines, the blue one represents the 64 images dataset, the orange one the 32 and the blue one the 16, same applies for figure 4, where the times of execution are shown. The speedup is directly linked with the size of the dataset, having the best speedup with 64 images while the speedup for the 16 images starts slowing at the 8th thread and starts increasing more slowly compared to 32 and 64 images. Interestingly, the execution time for the smaller datasets slows down at the 8th and 14th threads, while it could be just noise caused by the multithreading, it could also be caused by the CPUs hyperthreading.

### 3.3. Conclusions

After parallelizing the computations, python revealed itself a valid language to use multithreading, the results are valid and the reading simplicity are another positive point, the language is therefore now usable to parallelize even CPU-bound operations and while it still shows the experimental nature of the project, the freethreading python can (and hopefully will) be still developed to further enhance the performance it can output, maybe reactivating the specialized adaptive interpreter or simply fixing more compatibility issues and in the long run, this language could even rival the pure speed of openMP.

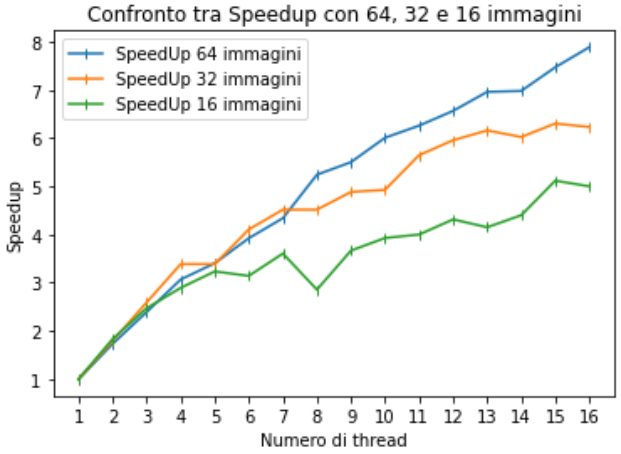


Figure 3. Plot of the speedup differences between datasets

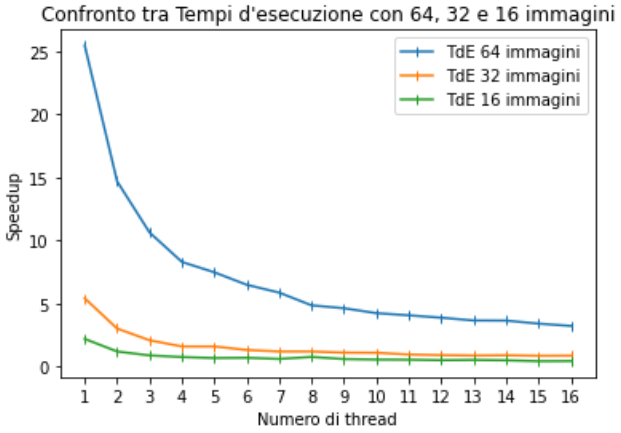


Figure 4. Plot for the execution times between datasets