PC-2024/25 Laboratory Assignment

D'Alò Gabriele
gabriele.dalo@edu.unifi.it

Abstract

This assignment deals with the creation of an image augmentation system for object detection. The main goal is to evaluate the difference in performance of the sequential and parallel versions.

1. Introduction

1.1. Assignment

In this assignment we want to create an image augmentation system for object detection using Python. It is required to implement the sequential and parallel version of the code in order to evaluate the difference in performances. It is also required to carry out experiments on different cases, for example we want to test with datasets of different sizes and with different parallelization methods.

1.2. Objectives

The goal of the assignment is to experiment with parallelizing Python code.

1.3. Development environment

All experiments were carried out on a machine with a macOS Sonoma operating system with a 2,6 GHz Intel Core i7 6 core processor and a 16 GB 2400 MHz DDR4 memory. The code has been written using the Python language and PyCharm was used as the IDE.

2. Implementation

2.1. General overview

This project provides a method to perform image augmentation in order to increase the images present in a dataset and also increase its diversity. In particular, we focus on object detection so when we augment the image we also have to take into account the bounding boxes of the objects present in the images.
The library used for augmentation is Albumentations (`https://albumentations.ai/`) while the image dataset is the Teledyne FLIR ADAS Thermal Dataset v2 (`https://www.kaggle.com/datasets/samdazel/teledyne-flir-adas-thermal-dataset-v2`).
The project allows experimenting with different dataset sizes and different parallelization techniques in Python. The images created with augmentation are saved in memory for demonstration purposes, as in a real situation one would prefer to do augmentation on-the-fly, using the augmented images immediately (e.g. for training).
For augmentation, a simple transformation pipeline was created following Albumentation's pipeline creation guidelines.
The code execution flow is as follows:

1. You are asked for the size of the dataset to use;
2. The parallelization technique to be used is asked (see 2.3);
3. If the sequential version is not chosen, you are also asked for the number of cores to use;
4. The dataset is loaded;
5. The dataset is augmented using the chosen parallelization version;

6. The augmented images are saved in a folder.

2.2. Main project

Let's now move on to the main part of the project, the implementation of the augmentation.
First of all it is necessary to say that Python suffers from some **limitations regarding parallelism**. The CPython interpreter uses reference counting to manage memory. In a multithreaded program, updating these reference counts must be thread-safe. To prevent corruption, Python only allows one thread to execute Python bytecode at a time. The **GIL** (Global Interpreter Lock) is a mutex that ensures only one thread runs Python code at a time, even on multi-core machines. This means Python threads cannot execute CPU-bound tasks in true parallel. Python supports multithreading, but because of the GIL, threads are limited to concurrency, not parallelism for CPU-bound tasks. The GIL is released during low-level I/O system calls (like reading from disk or network). So threads performing I/O can run in parallel, since these operations happen outside the Python runtime. This means the GIL does not significantly impact I/O-bound multithreaded programs.
However, many Python libraries written in C/C++ (like NumPy, OpenCV, or deep learning frameworks) explicitly release the GIL during heavy computations. This allows other Python threads to run in parallel, achieving real concurrency on multi-core CPUs. For example, Albumentations uses OpenCV, so many image transformations happen in C++ with the GIL released. This means you can often see real multithreaded speedups despite Python's GIL.
In this project, three different techniques are used to parallelize image augmentation (and image storage) in order to verify the performance differences (see 4 for a glimpse of che code used).
The first technique uses the **ProcessPoolExecutor** to parallelize image augmentation across multiple CPU cores. It builds a list of samples, splits them into chunks, and distributes them to worker processes. It creates separate processes, each with its own Python interpreter and GIL. This allows true parallel execution on multiple CPU cores, unlike threads, which are limited by Python's GIL. We use a dynamic chunksize to reduce the overhead of sending tiny tasks. Larger chunks mean fewer dispatch calls, but risk less load balancing. In general, this approach is better for CPU-bound tasks (heavy computations like image processing in Python). For I/O tasks, threads are usually better (see 4.1).
The second version, uses **joblib with the threading backend** to run the augmentation in parallel across multiple threads. Threads share the same memory space and are useful for I/O-bound tasks (like downloading files or waiting on network / disk). In Python, threads do not bypass the GIL, so for CPU-bound work, you get no real parallel speedup. However, often the GIL is released during Albumentations heavy operations, because many transformations (like scaling, rotations, filters) use OpenCV or NumPy, which are implemented in C/C++ and release the GIL when they do computations. This means that even with the threading backend, you can get some real parallelism, because Python threads can concurrently execute underlying C code that doesn't need the GIL. But the effect depends on how much work Albumentations does outside Python, and how much is pure Python overhead (see 4.2).
The latest version, instead, uses **joblib with loky backend** which performs real parallelism through processes (not threads). It is similar to multiprocessing, but handled in a more robust and cross-platform way. GIL is not a problem here because each process has its own Python interpreter so they can execute bytecodes at the same time. It is good for CPU-bound or mixed (CPU + I/O) workloads but has more overhead than threads: each process has its own memory and data must be serialized/deserialized, so it can be slower if the exchanged data is large (e.g. HD images) (see 4.3).

2.3. Implementation choices

Let's now talk about some implementation choices that have been made.
First of all, for memory reasons it was decided to use a maximum size of 10000 images.
Regarding the augmentation pipeline, it is necessary to specify that, given the nature of the grayscale images, only transformations that make sense for such images were used. Therefore, all color transformations were avoided.

While doing some tests, a degradation in performance was observed when running the sequential version and the parallel version in the same code launch (probably due to the increase in temperature of the machine), therefore it was decided to run each test separately from the others.
Finally, batch augmentation of images instead of a single image at a time was tested but no significant performance improvements were observed so it was decided to omit it from the final implementation.

3. Resutls analysis

To conclude, let's analyze the results obtained from our experiments. As regards performance, the main parameter we are interested in analyzing is certainly the speedup which allows us to understand how quickly our parallel code performs compared to the sequential version. Remember that:

$$Speedup = \frac{Execution\ time\ sequential}{Execution\ time\ parallel}$$

Furthermore, another parameter that can be evaluated is efficiency, which depends on the number of cores used and the speedup obtained and indicates how efficiently we are using the cores at our disposal:

$$Efficiency = \frac{Speedup}{Ncores}$$

3.1. Results

The tables below show the speedup (and efficiency) obtained compared to the sequential version on each of the three datasets for each parallelization technique used. For simplicity, we report the best results after testing with different numbers of cores (by best we mean both in terms of speedup and efficiency). Tests were performed with: 2, 4, 6, 8, 10 and 12 cores.
Performance was only evaluated for the image augmentation (and image saving) part. For the initial data loading part, both the sequential and parallel versions were tested, but the sequential version was found to be faster in most cases. This is probably due to the fact that there is relatively little data to load and that the actual images are only really acquired during the image augmentation phase.

3.1.1 Small Dataset

Technique	Cores	Speedup	Efficiency
ProcessPoolExecutor	4	1,56	0,39
Joblib_Threading	4	3,06	0,77
Joblib_Loky	4	1,32	0,33

Table 1. Parallelization speedup on small dataset

In the first dataset it is clearly seen that Joblib with Threading backend achieves the highest speedup (3.06) and efficiency (0.77). This is likely because:

- The workload is relatively light, so threads avoid the overhead of process spawning and data serialization;
- Albumentations releases the GIL during underlying OpenCV operations, allowing threads to run in parallel effectively.

3.1.2 Medium Dataset

Technique	Cores	Speedup	Efficiency
ProcessPoolExecutor	6	2,79	0,47
Joblib_Threading	4	2,90	0,73
Joblib_Loky	4	2,39	0,60

Table 2. Parallelization speedup on medium dataset

In the second dataset Joblib_Threading still performs best (2.90 speedup, 0.73 efficiency). This suggests that for medium workloads, the benefits of threading (low overhead, good concurrency with I/O) continue to outweigh the extra costs of multiprocessing.

3.1.3 Big Dataset

Technique	Cores	Speedup	Efficiency
ProcessPoolExecutor	6	3,00	0,50
Joblib_Threading	4	2,68	0,67
Joblib_Loky	4	2,48	0,62

Table 3. Parallelization speedup on big dataset

Interestingly, regarding the last dataset, ProcessPoolExecutor slightly surpasses threading (speedup 3.00 vs. 2.68). This indicates that:

- With larger workloads, the GIL and thread scheduling overhead start to limit threading performance;
- Processes benefit from true parallelism on CPU cores, despite higher inter-process communication costs.

Although loky also uses processes (like ProcessPoolExecutor), it didn't outperform threading on the small and medium datasets, probably because its serialization/deserialization overhead is more noticeable on smaller tasks and the batches of work may have been too small to offset the costs.

3.2. Performances evaluation

Although parallelization clearly reduces execution time, the speedup is sub-linear due to parallel overhead, GIL constraints for Python-level code, and resource contention as cores increase. Threading performed best overall for small to medium datasets thanks to low overhead and GIL-friendly native code. Process-based approaches start to close the gap on large datasets, benefiting from true multi-core execution, but with higher serialization costs.
To summarize, the results show decent but not linear scaling, which is actually very typical in Python workloads. This can be due to various factors, including:

- Parallel overheads: task scheduling, thread/process creation, data splitting and merging results all add overhead. For ProcessPoolExecutor and loky, there's also the serialization cost (pickling/unpickling) when sending data between processes. This is more noticeable on small datasets;
- GIL impact (for threading): even though Albumentations releases the GIL during many heavy C/OpenCV operations, your workload still contains Python-side logic (loops, ecc...) which can't run fully in parallel. That limits Joblib_threading from perfect scaling;
- Diminishing returns with more cores: as you added more cores (8, 10, 12), speedup grew much slower. This happens due to contention on shared resources (RAM, filesystem, Python interpreter locks) and more context switching overhead. At some point, adding cores splits the same work into smaller slices, adding overhead without benefit;
- Possibly not fully CPU-bound: since image augmentation often calls into optimized libraries (like OpenCV), which run outside Python, threading can help a lot up to a point. However, these libraries still compete for CPU cores, memory bandwidth, or even disk I/O when loading/saving files.

4. Code fragments

4.1. Code version ProcessPoolExecutor

```
args = [(idx, train_samples[idx]) for idx in sample_indices]

# Calcolo dinamico di un chunksize
# ad es. tot_samples diviso (n_processi * fattore)
tot = len(args)
factor = 25
chunkSize = max(1, math.ceil(tot / (nCores * factor)))
print(f"\n[INFO] Chunksize = {chunkSize}")

with ProcessPoolExecutor(max_workers=nCores) as executor:
    results = list(executor.map(process_sample, args, chunksize=chunkSize))
```

4.2. Code version joblib with threadnig

```
# Usa il backend 'threading'
with parallel_backend('threading', n_jobs=nCores):
    results = Parallel(verbose=10)(
        delayed(process_sample)((idx, train_samples[idx])) for idx in
        ↪  sample_indices
    )
```

4.3. Code version joblib with loky

```
# Usa il backend 'loky'
with parallel_backend('loky', n_jobs=nCores):
    results = Parallel(verbose=10)(
        delayed(process_sample)((idx, train_samples[idx])) for idx in
        ↪  sample_indices
    )
```