# Parallel DataLoader
## Parallel Programming for Machine Learning
## Project Work in Artificial Intelligence Programming

Niccolò Arati
niccolo.arati@edu.unifi.it

### Abstract

*This project aims to analyze the benefits of implementing a Dataloader with parallelization. In particular, we want to observe the **speedup**, calculated as the ratio between sequential execution time and parallel execution time, while varying different configurations for the Dataloader.*
*The results will also be analyzed changing the number of worker processes associated to parallelization.*



### 1. Introduction

A Dataloader is a class that provides a flexible and efficient way to load data. Its primary application involves loading data into a model for training or inference.
In this work, we will focus on loading images and applying to them data augmentation techniques.
The purpose of the project is to measure the speedup achieved by parallelizing the dataloader and comparing it with its sequential version. The programming language used is Python, and for parallelization Multiprocessing was exploited.
All experiments were conducted on a PC with Windows 10 as the operating system and an Intel Pentium Gold G5400 CPU.

### 2. Code Structure

The IDE employed for this project is PyCharm. The code consists of multiple files:

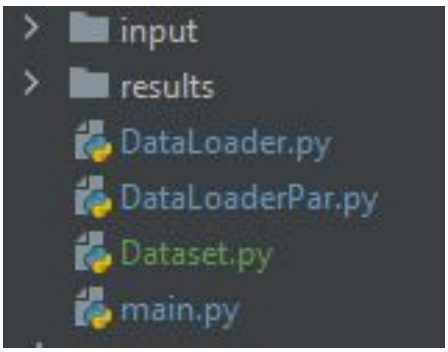


Figure 1. Code structure.

- **DataLoader.py**: contains the sequential implementation of the DataLoader class, defined as an iterable;
- **DataLoaderPar.py**: contains the parallelized implementation of the DataLoaderPar class, which extends the DataLoader class;
- **Dataset.py**: contains the implementation of the Dataset class;
- **main.py**: contains functions related to the tests, as well as the main() function.

The **inputs** folder is also considered, it contains 111 subfolders with 1000 images each on which the Dataloader will be called later.
Since we will use images as data to be loaded with the Dataloader, as specified in Sec. 1, the implementation only considers images and does not generalize to other types of data.

#### 2.1. Class DataLoader

The DataLoader class accesses the dataset of images to be loaded and for each iteration it re-

turns a certain number of them in batches.
Four functions are defined for DataLoader objects:


Figure 2. Dataloader Class declaration.

- **__init__** initializes the dataloader and its attributes, requiring as parameters the dataset (an object of the Dataset class), the batch size to load, and an optional boolean to randomize the order of the images. Additionally, it defines an integer index set to 0, which will be used to iterate through the images.
- **__iter__** returns the object (DataLoader) to be iterated over.
- **__next__** loads a batch of images by calling the get method for a number of times equal to the value of the batch_size attribute, and returns it in output.
- **get** loads an image from the dataset and increments by one the index attribute.

### 2.2. Class DataLoaderPar

Subclass of DataLoader, for the method __init__ it requires additional parameters compared to the DataLoader class. This method also initializes some data structures that will be discussed in Sec. 3 with other functions defined for this class.

### 2.3. Class Dataset

The Dataset class represents a wrapper for the actual dataset to be loaded. In our case, it receives the paths relative to the location of the images and, when requested from the DataLoader, loads them into memory and returns them.
The class has 3 methods:

- **__init__** initializes the dataset, it requires parameters such as size, which represents the dataset's dimension, and im_paths, which is a list containing the paths of the images to be loaded. These two parameters are then saved as attributes of the class.
- __len__ simply returns the size of the dataset.
- **__getitem__** takes as input the index of the image to be loaded and then, by that index, it accesses the path of the desired image and loads it (using the cv2 package).

### 2.4. main.py

In this file there are tests related to the parallelization of the Dataloader (presented in Sec. 4) along with the main function, whose purpose is to execute the tests defined in the file.
To generate strings containing the paths of the images, the pathlib library was used.

## 3. Parallelization with Multiprocessing

The code of the DataLoaderPar class defines a function called **worker_funct**, which will be executed by the Worker processes defined through Multiprocessing.
The processes are generated through the __init__ method of the class, and then there are other methods for their management, which will be presented in the following subsections.
The idea behind the parallelization of the Dataloader is that when a batch of images is loaded, it is assumed that these are subjected to a transformation by a certain process (with Albumentations, Sec. 4). While the images are going through transformation, we want to use Worker processes to prefetch the next batch so that it (or most of it) is already ready in memory for transformation.

### 3.1. Function associated with Worker Processes

From the code presented in Fig. 3, the function requires 3 parameters: the dataset to load, an index_queue, and an output_queue (both are multiprocessing.Queue objects). The output_queue is shared among all processes, while each process

```
def worker_funct(dataset, index_queue, output_queue):
    while True:
        try:
            index = index_queue.get(timeout = 0)
        except queue.Empty:
            continue
        if index is None:
            break
        output_queue.put((index, dataset[index]))
```

Figure 3. Code of function worker_funct

has its own private index_queue associated with it.
All the index_queue objects contain the indices of the elements to be loaded. The Worker process, following the function, takes an index from the queue (if it is not empty) and inserts the corresponding image from the dataset into the output_queue, paired with the index.

### 3.2. Method __init__

The __init__ method requires the same parameters as the DataLoader class, as well as a num_workers parameter specifying the number of Worker processes to generate, and a prefetch_batches parameter specifying the maximum number of batches to prefetch.
In addition to initializing the attributes corresponding to the input parameters, this method also initializes the output_queue (Sec. 3.1), a list that will contain the index_queues associated with the Worker processes, another list that will contain the Worker processes themselves, an attribute called worker_cycle that will be used to iterate through the processes in order to assign the indices of the images to be loaded to their index_queues, a list called cache (Sec. 3.3), and a prefetch_index attribute that keeps track of the prefetched elements (different from the index attribute as index keeps track of the images actually processed by whoever is using the DataLoaderPar).
Subsequently, the method initializes the Worker processes and their respective index_queues:

As shown in Fig. 4, a multiprocessing.Process is generated with the function specified earlier as its target, and as its attributes the dataset and the two queues (a private index_queue and a shared output_queue).
The Worker process is specified as a **daemon**, as we want it to act in background compared to the image transformation process with Albumentations.
Finally, the prefetch function, presented in Sec. 3.3, is called.

```
for _ in range(num_workers):
    index_queue = multiprocessing.Queue()
    worker = multiprocessing.Process(target = worker_funct,
                                     args = (self.dataset, index_queue,
                                             self.output_queue))
    worker.daemon = True
    worker.start()
    self.workers.append(worker)
    self.index_queues.append(index_queue)
```

Figure 4. Code related to the initialization of Worker processes and their associated index_queues.

### 3.3. Image loading and Dataloader management

The function **prefetch** handles the management of image prefetching. First of all, it checks whether the end of the dataset has been reached or if it is more than two batches ahead of the images actually loaded. Then, it assigns the indices of the images to be prefetched to the index_queues associated with the Worker processes.
To do this, it leverages the worker_cycle attribute, and consequently updates the prefetch index.
On the other hand, the **get** method handles the actual loading of images and thus the current batch.

Observing the code in Fig. 5, we can summarize it into 3 points:

- It checks if the index of the image to be loaded is present in the list defined by the cache attribute. If it is present, it is removed from the list and returned as output.
- Otherwise, it checks the output_queue. If the element obtained has an index corresponding to the one of the image to be loaded, it is returned as output.
- Otherwise, the element is added to the cache, waiting when the index of the elements to be

```
def get(self):
    self.prefetch()
    if self.index in self.cache:
        item = self.cache[self.index]
        del self.cache[self.index]
    else:
        while True:
            try:
                (index, data) = self.output_queue.get(timeout=0)
            except queue.Empty:
                continue
            if index == self.index:
                item = data
                break
            else:
                self.cache[index] = data

    self.index += 1
    return item
```

Figure 5. Code of method get

loaded reaches the index of the element in question (first point).

The **__iter__** method, as for the DataLoader class, returns the object to be iterated over, thus reinitializing the attributes related to the index of the image to be loaded (index), the index of the image to be prefetched (prefetch_index), and the cache list.

Finally, the **__delete__** method terminates the Worker processes.

## 4. Experiments conducted

Returning to the main.py file, the functions written for the tests are as follows:

- **testDataLoaderWT**: executed by the main function, calculates the speedup between the sequential dataloader and the parallel one, varying some configurations which consider the number of workers used for parallelization and the complexity of the transformations applied to the images in each batch.
- **testDataLoaderB**: also executed by the main function, but this time the speedups are calculated mainly considering a specified batch size.

The parameters considered for the experiments are therefore the number of workers used for Multiprocessing, the batch size of images, and the complexity of the transformations for the images. In particular, for this last part, Albumentations was used. In Fig. 6, the simplest transformation defined is shown (for the more complex ones, essentially the transformation shown in the figure is applied multiple times).

```
transform1 = albumentations.Compose([
    resize.Resize(height=256, width=256, p=1),
    albumentations.RandomCrop(width=200, height=200, p=1)
])
```

Figure 6. Base transform defined with Albumentations

The test results are saved in specific CSV files located in the **results** folder.
Below there are the tests conducted for the Dataloader.

### 4.1. Test1: changing n_workers and albumentations

With the first test, executed by the testDataLoaderWT function, we aim to assess the impact of the number of worker processes while varying the complexity of the transformations applied with Albumentations between batch loads.
The batch_size value is kept constant at 300.
The expected outcome of the test is that, as the complexity of the image transformation process increases, the sequential version of the Dataloader will increase the computational time required to process all batches. The parallelized version should instead still maintain better performance, because while the image transformation process is running there are also worker processes responsible for prefetching the next batch.

| | 1 w | 2 ws | 4 ws | 6 ws | 8 ws |
|---|---|---|---|---|---|
| 1stAlb | 1.03472 | 1.38500 | 1.87490 | 2.14634 | 1.9381 |
| 2ndAlb | 3.55765 | 5.02616 | 6.39762 | 7.16192 | 4.50477 |
| 3rdAlb | 4.29092 | 5.94258 | 6.81652 | 4.65382 | 2.74523 |
| 4thAlb | 4.52743 | 6.68475 | 5.05657 | 3.58080 | 2.15300 |

Table 1. Table with speedup values varying with the number of worker processes and the complexity of the transformations applied to the images. Remember from Sec. 3 that the total number of processes running is given by the number of workers plus one, corresponding to Albumentation.

The number of workers considered for the experiment varies between the values of 1, 2, 4, 6,

and 8, while 4 possible image transformations are considered in increasing order of complexity.
Observing the results presented in Tab. 1, we notice that the predictions were accurate. By already looking at the first column, so just utilizing a single worker process working in parallel with the Albumentations process, we see significant improvements in the speedup value as the complexity of the image transformations increases.
However, there are limitations to the performance improvement, likely due to the hardware used for the experiment. It is observed that using 8 worker processes consistently leads to performance degradation compared to the same configuration with 6 worker processes.
Furthermore, observing the rows of the table, there is a loss of the speedup value with less number of workers as the complexity of the Albumentations process increases. Sure enough, with the first and second image transformation processes the speedup decreases only with 8 workers (compared to the previous value of workers), while with the third transformation, it decreases from 4 to 6 workers, and with the fourth transformation, it decreases from 2 to 4 workers.
This may be due to CPU saturation, as it has to handle an increasingly complex transformation process and the images in the considered batch, in addition to a potentially larger number of worker processes.
In any case, observing the speedup values, there is always an improvement in performance compared to the sequential version. The only case where performance is practically equivalent is when the simplest transformation is applied while a single Worker process prefetches the next two batches (row 1, column 1).

### 4.2. Test2: varying batch_size

In this second test, executed by the testDataLoaderB function, speedup values are calculated by varying the sizes of the loaded batches, while keeping the number of Worker processes at 2 and the image transformation process constant (equivalent to the second process applied in the previous test).
The number of workers is kept relatively low and the transformation is not overly complex in order to isolate the impact of batch_size compared to the other parameters.
The expectations for this experiment are to observe an increase in speedup as the batch size grows, exploiting the parallel Dataloader's ability to prefetch images for the next batch while those of the current batch are processed with Albumentations.
The batch_size is varied between the values of 10, 50, 100, 300, 500, and 1000.

| | SpeedUp |
|---|---|
| batch_size = 10 | 1.54175 |
| batch_size = 50 | 1.59989 |
| batch_size = 100 | 1.56747 |
| batch_size = 300 | 1.55132 |
| batch_size = 500 | 1.48342 |
| batch_size = 1000 | 1.72745 |

Table 2. Table with speedup values varying with the batch size for the images.

Observing the results in Tab. 2, it can be inferred that the parameter related to the size of the batch of loaded images may not have much influence on the speedup value.
In fact, all speedup values are comparable, the only one that deviates more from the others being the speedup related to a batch_size of 1000.
This is still an acceptable result, as the batch_size value is usually based on the task in which the Dataloader is utilized, or on the size of the available data, and it makes sense that we shouldn't be forced to increase the batch size just to achieve better performance in terms of speedup (and therefore computational time).