Parallel Programming
Lab assignment

Lucia Giorgi

1. Goal of the assignment

Perform image augmentation on a image dataset using the Python library Albumentations (fig. 1). Given an input folder with N images, write M augmented images in an output folder for every input image (N * M augmented images in total).

Figure 1: Examples of image augmentation from Albumentations website

2. Implementation

There are a sequential and two parallel versions of this script.
The script takes 4 inputs (5 in the parallel versions):

- the input folder with the input images
- the output folder which will contain all the augmented images by the end of the execution
- the number of augmentations needed for each input image (M)
- number of parallel processes (parallel versions only)
- name of the output result file

For every image in the input folder, M augmentations are created and saved in the output folder. Every augmentation is created by sampling and combining a random number of transformations from a list, each one with random parameters within a range.

2.1. Files

A file named `common.py` contains the functions and variables that are common for both the sequential script and the parallel scripts. These include:

- the list of transformations
- the function `apply_transformation()` that takes an image, samples a number of transformations, composes the transformations, creates a single augmentation and saves it in the output folder with a name indicating the original image and the augmentation number
- a function for parsing command-line arguments
- a function for writing the experiment result to a file

In each script, every image from the input folder is read and converted into RGB space using OpenCV. Then, `apply_transformation()` is called M times (one for every augmentation).

3. Parallelization

Parallelization is achieved by using a `multiprocessing.Pool` of processes.
There are two possible ways to parallelize this task:

- **on the augmentations**: given an image, a pool of `num_processes` is used in order to create M augmentations. There is an external for-loop on the images and for each image `pool.starmap()` is called on `apply_transformation()`. The same pool is used for every image. Code in fig. 3.
- **on the images**: a pool of *num_processes* is used for going through the images in the input folder. Every process performs the augmentations sequentially (with a for-loop on the number of augmentations). Code in fig. 3.

```
def augment_images(input_dir, output_dir, num_augmented, num_processes, result_file):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    start_time = time.time()
    with multiprocessing.Pool(processes=num_processes) as  pool:
        for filename in os.listdir(input_dir):
            input_path = os.path.join(input_dir, filename)
            image = cv2.imread(input_path)
            image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
            pool.starmap(apply_transformation,
                [(image, output_dir, filename, i) for i in range(num_augmented)])
    end_time = time.time()
    write_output_to_file(end_time, num_processes, num_augmented, result_file, start_time)
```

Figure 2: Parallelization on the augmentations

```
def process_image(filename, input_dir, output_dir, num_augmented):
    input_path = os.path.join(input_dir, filename)
    image = cv2.imread(input_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    for i in range(num_augmented):
        apply_transformation(image, output_dir, filename, i)

def augment_images(input_dir, output_dir, num_augmented, num_processes, result_file):
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)
    start_time = time.time()
     with multiprocessing.Pool(processes=num_processes) as  pool:
        pool.starmap(process_image,
            [(filename, input_dir, output_dir, num_augmented) for filename in os.listdir(input_dir)])
    end_time = time.time()
```

Figure 3: Parallelization on the images

4. Experiments and results

For the experiments I used $N = 40$ and $M = 40$, that is 40 input image and 40 augmentations for each image, and images of size 4000×3000 pixels.
I ran the sequential version of the program and the parallel versions with different number of processes (2 to 24) and the execution times were measured by taking the start time just after the argument parsing and the output directory creation. My machine has 8 physical cores, so I expect to see an increment in the speedup up to 8 parallel processes and then a decrement.
Results are shown in figs. 4 and 5. We see that in both cases we have a maximum speedup for 6 processes, even though the machine has 8 cores. The improvement with parallelization is very mild in both cases. This might be due to some overhead caused by the high dimension of the images, especially in the *parallelization on the augmentations* version where the image is copied for every process (this version is indeed a bit slower than the other one). After the peak, then, the speedup decreases as expected, even below 1.

I ran the experiment again but with smaller images (400×300) with 2 to 30 processes (figs. 6 and 7). This time the speedup is more noticeable than in the previous experiment. The *parallelization on the images* version is still the fastest one and in both versions the speedup peak is at 22 processes.
Some example of augmented images produced by the scripts are shown in fig. 8.

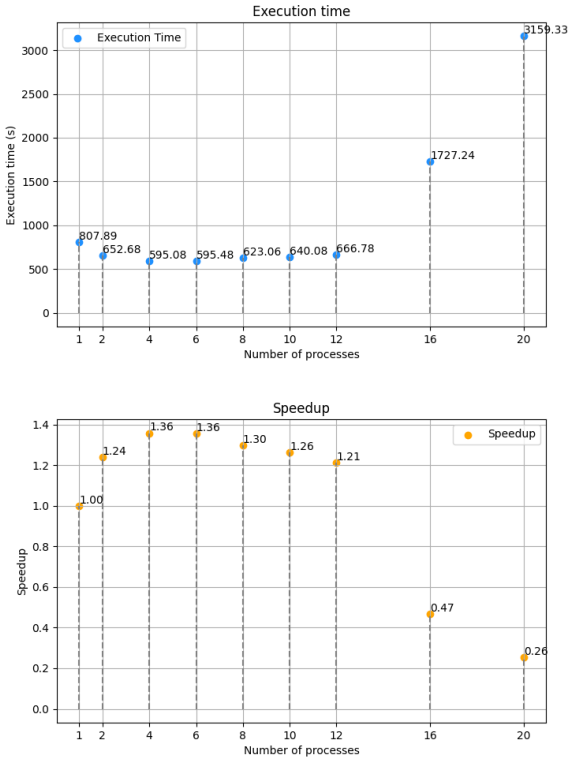

Figure 4: Parallelization on the augmentations: execution time and speedup for different number of processes (4000×3000)

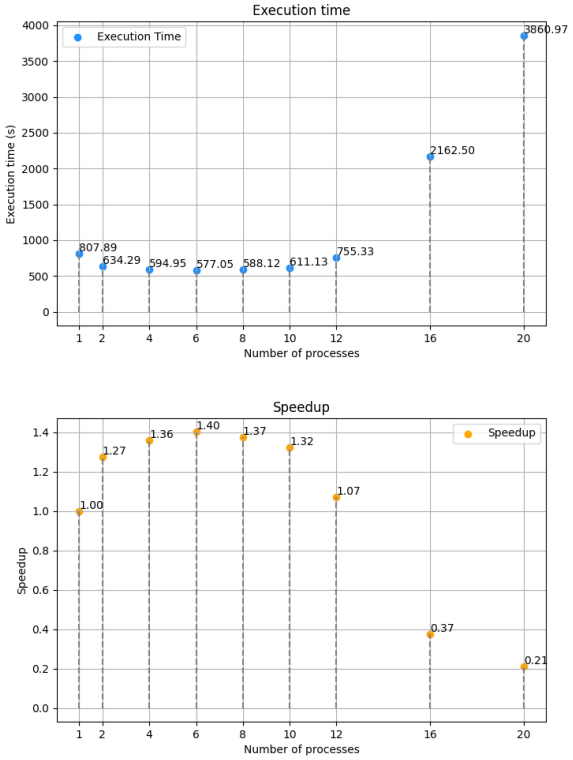

Figure 5: Parallelization on the images: execution time and speedup for different number of processes (4000×3000)

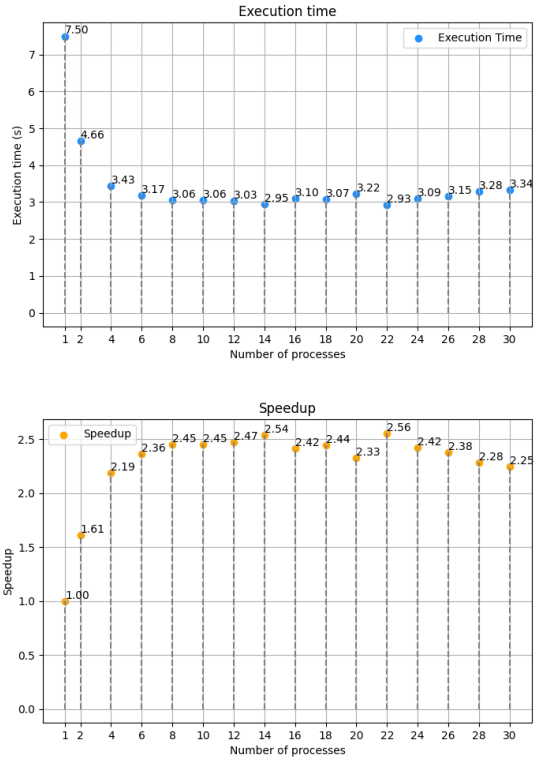

Figure 6: Parallelization on the augmentations: execution time and speedup for different number of processes (400×300)

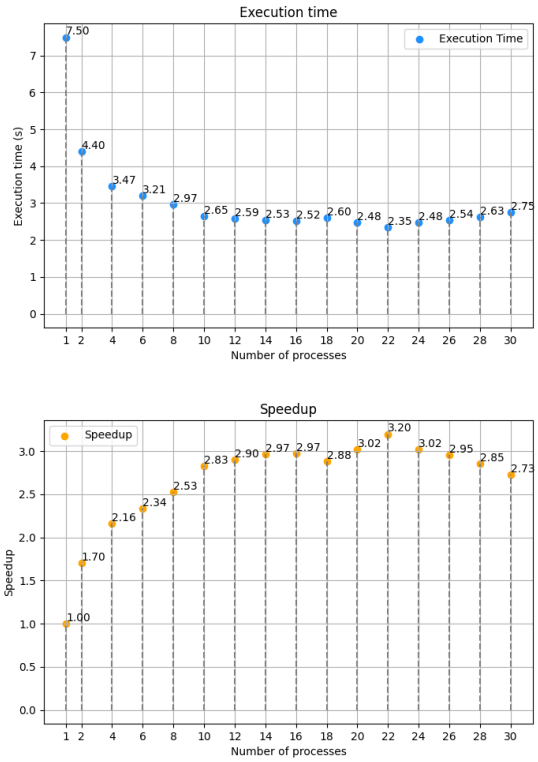

Figure 7: Parallelization on the images: execution time and speedup for different number of processes (400×300)

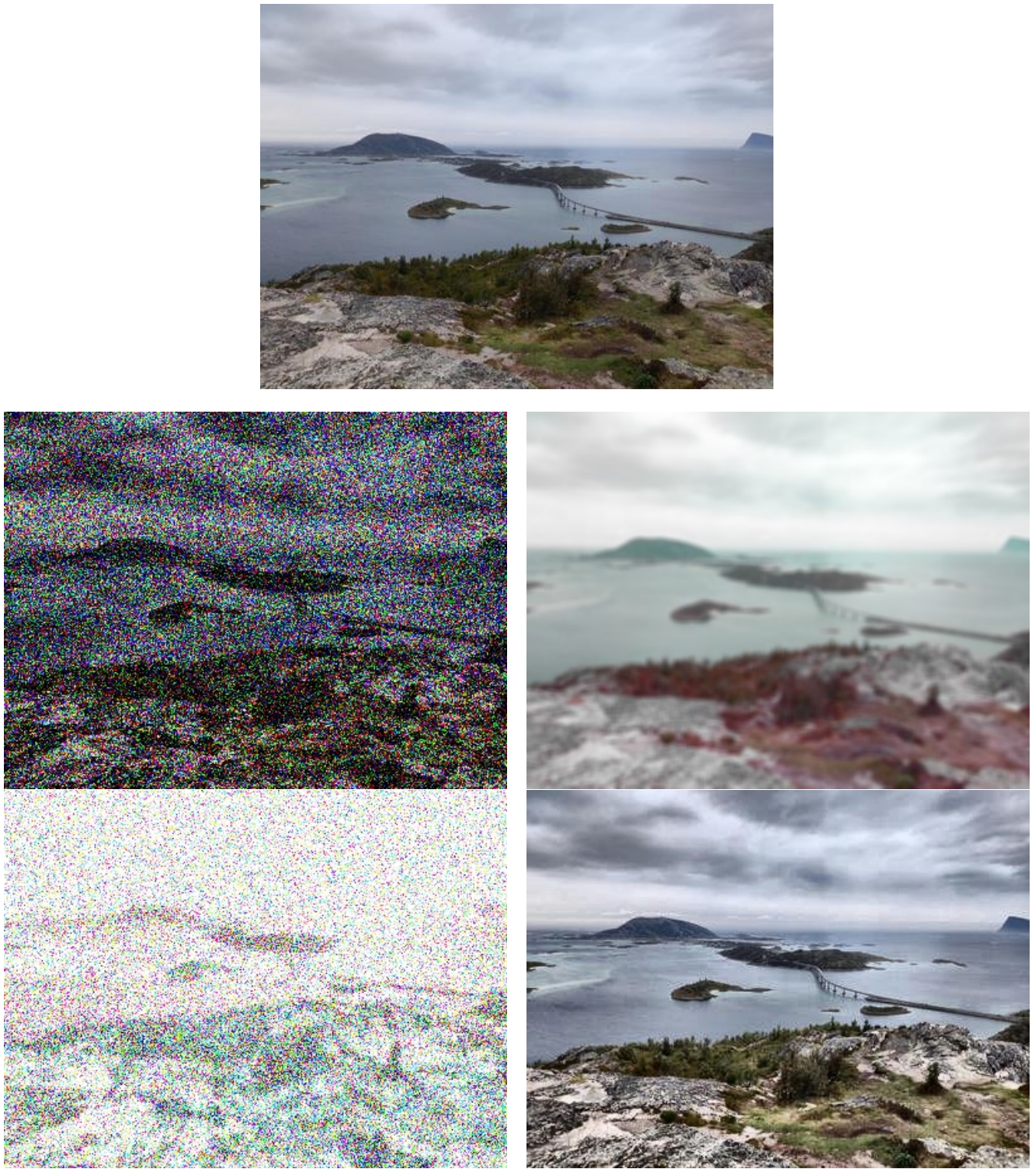

Figure 8: Examples of augmentations produced by the script from an input image