Parallel Programming: Bulk Image Enhancer

1st Joshue Salvador A. Jadie
Manila, Philippines
joshue_jadie@dlsu.edu.ph

2nd Andre Dominic H. Ponce
Manila, Philippines
andre_ponce@dlsu.edu.ph

***Abstract*—This document covers how parallel programming concepts were applied in the implementation of a *bulk image enhancer program* in Python. Both thread-based and process-based implementations were presented. The performance of the program was also assessed based on various factors with respect to its non-parallelized implementation.**

***Index Terms*—Image Enhancement, Parallelism, Python, Pillow**

I. Introduction

A. General Specification

In a usual sequential image enhancer program, when given a set of images to enhance, it would go through the images and enhance them one by one. The problem here is that when the number of images increases, the time it takes for the program to sift through all the provided images increases as well. This is where *parallelism* becomes useful.

This project requires the implementation of a *bulk image enhancer* program that can effectively enhance multiple images simultaneously with the use of parallel programming techniques. In enhancing an image, the **brightness**, **sharpness**, and **contrast** of the images will be adjusted based on the user's input. The program will only support and enhance images with **.jpg**, **.gif**, or **.png** as file formats.

The program can be implemented in any programming language as long as it incorporates parallel programming techniques. The use of libraries or application programming interfaces (API) for image enhancement such as Pillow is also allowed.

B. Program Input

The program must be able to take the following arguments as input:

- Input folder directory of the images to be enhanced
- Output folder directory of the enhanced images
- Enhancing time in minutes
- Brightness enhancement factor
- Sharpness enhancement factor
- Contrast enhancement factor
- *Optional: Number of threads or processes to use*

C. Program Output

The program should also output the following once it finishes executing:

- The **enhanced images** in the specified output folder location
- A **text file** that contains the statistics of the processed files (i.e., number of images enhanced, output folder location)

II. Program Implementation

A. Parallelism

The programming language used for implementing the program is **Python**. The main implementation uses the `threading` module to create multiple **threads** that will load and enhance the input images simultaneously. A `multiprocessing`-based implementation that uses multiple **processes** instead will also be introduced later on as an optimization.

An `ImageEnhancer` class was created to hold all thread-related functionalities. This is where all the processes related to image enhancement will occur. This class also inherits the attributes and methods from the `threading.Thread` class.

The **lock** variable `num_enhanced_global_lock` was used to impose mutual exclusion on a shared variable named `num_enhanced_global` - which is intended for counting the total number of images that all the threads were able to enhance within the given time limit.

The **queue** variable `images` was used to store all the image data needed for the threads to be able to load and enhance the input images. Since the variable is a `queue.Queue` object, which is thread-safe, there is already no need to use locks or semaphores for this variable.

Since additional threads are spawned within the address space of the main thread, these new threads already have access to the same set of global variables. With this, there is also no need to coordinate the use of these variables unlike when using multiple processes.

B. Image Enhancement

This implementation uses the `ImageEnhance` module of Pillow - a fork of the Python Imaging Library (PIL) - to enhance the input images. This module contains various classes that also contain methods for enhancing images according to a certain attribute (i.e., color, contrast). With this, to address the brightness, sharpness, and contrast enhancement requirements of this project, the following `ImageEnhance` classes' `enhance(image)` methods were used: `Brightness`, `Sharpness`, and `Contrast`.

C. Implementation

When running the program, the arguments passed from the command line interface (CLI) will be parsed first. To do this, the `ArgumentParser` class of the `argparse` module was utilized. Upon parsing the input arguments, the function `get_images()` is then called to get the data of the

input images in the specified folder. Algorithm 1 shows the pseudocode of how the input images' data are retrieved by the program.

Algorithm 1 Images Retrieval

```
images ← global queue of image data
path ← input folder directory
num_images_input ← no. of input global counter
for file in path do
    if existing file and valid format then
        append image_data to images
    end if
end for
num_images_input ← size of images
```

This will populate the global `images` queue variable with image data. Each resulting element of the queue contains three attributes: *input path*, *image filename*, and *image format*.

input path contains the directory of the image file, *image filename* contains the name of the image itself, and *image format* contains the format of the image (i.e., .jpg, .png, .gif). These will be used by the threads later on when loading and enhancing the input images.

Afterwards, the threads are created and started in accordance to the number of threads entered by the user. If none was supplied, this defaults to five (5) threads. The `ImageEnhancer` class' constructor was also used here. Algorithm 2 shows the creation of multiple threads for the bulk image enhancing task.

Algorithm 2 Thread Creation

```
threads ← list of threads
num_threads ← number of threads to create
for i = 0 to num_threads do
    thread ← ImageEnhancer(i)        ▷ Instantiate object
    append thread to threads
    Start(thread)
end for
for thread in threads do
    Join(thread)
end for
```

Algorithm 3 shows the pseudocode that the threads follow when executing. First, it will retrieve an image's data from the *images* queue. It will then use this data to load the actual image from the local machine. Afterwards, it will enhance the image using the methods from the `ImageEnhance` Pillow module. Once done, it will save the enhanced image to the specified output folder of the local machine. A counter that is local to the thread is then incremented to signify that an image has been enhanced by it. The process repeats until (1) the queue becomes empty or (2) the time limit has been exceeded.

After all these, the thread's local counter of enhanced images will be added to the global total. To ensure mutual exclusion in this scenario, the lock `num_enhanced_global_lock` is used to restrict access to the global counter `num_enhanced_global`.

Algorithm 3 Thread Image Enhancing

```
images ← global queue of image data
curr_time ← current time
time_limit ← time limit input
num_enhanced ← no. of images enhanced by this thread
num_enhanced_global ← global no. of images enhanced by all
num_enhanced_global_lock ← global lock
while images not empty do
    if curr_time ≥ time_limit then
        break loop
    end if
    image_data ← images
    enhanced_image ← EnhanceImage(image_data)
    SaveImage(enhanced_image)
    num_enhanced ← num_enhanced + 1
end while
acquire(num_enhanced_global_lock)
num_enhanced_global ← num_enhanced_global + num_enhanced
release(num_enhanced_global_lock)
```

Algorithm 4 shows the pseudocode of the `enhance_image()` function as seen and called in Algorithm 3 as **EnhanceImage()**. In enhancing an input image, the file format is checked first to determine whether the image is a GIF or not. If the image is a GIF, which consists of many frames (images), each frame is enhanced through a loop. Else, if the format is JPG or PNG, the image is enhanced only once which is the same as enhancing a single frame when it comes to GIFs. The final enhanced image is then appended to the image data to be saved afterwards.

Algorithm 4 Image Enhancement

```
brightness ← global brightness enhancement factor
sharpness ← global sharpness enhancement factor
constrast ← global contrast enhancement factor
image_data ← data of image
image ← LoadImage(image_data)
if format is 'gif' then                    ▷ Image is a gif
    enhanced_frames ← list of enhanced frames
    for frame in image do                  ▷ Enhance each frame
        frame ← Brightness(frame, brightness)
        frame ← Sharpness(frame, sharpness)
        frame ← Contrast(frame, contrast)
        append frame to enhanced_frames
    end for
    append enhanced_frames to image_data
else                                       ▷ Image is a jpg or png
    image ← Brightness(image, brightness)
    image ← Sharpness(image, sharpness)
    image ← Contrast(image, contrast)
    append image to image_data
end if
```

Once all threads have exited already, the main thread will then print out the required statistics onto the console and onto a text file. The following statistics are included:

- Number of enhanced images (required)
- Output directory of enhanced images (required)
- Time elapsed since start of program
- Number of input images
- Number of image enhancement threads used
- Brightness, sharpness, and contrast enhancement factors used

D. Optimizations

Some enhancements were also implemented to improve the overall performance of the program:

1) Initially, the global counter `num_enhanced_global` was being incremented by one per iteration inside the threads' loop. This caused some performance overhead when the number of threads increased by a significant amount. This is because all the threads are acquiring the `num_enhanced_global_lock` in each of its iterations - which is causing some delay at times. To resolve this, the local thread variable `num_enhanced` was created, and this is the one being incremented by one instead in each iteration. This removes the need to acquire a lock for every loop iteration. When the loop ends, the value of this local thread variable is then added to the global counter - achieving the same goal as the initial implementation, but more efficiently since only one lock acquisition is needed per thread.
2) According to the ImageEnhance documentation, for its enhancement functions (i.e., brightness, sharpness, contrast), if the enhancement factor given is 1.0 then these functions will just return the original image. In the initial implementation of the program, regardless if the enhancement factor input is equal to 1.0 or not, the program will still call the corresponding enhancement function. This caused some performance overhead because there were unnecessary function calls. To resolve this, some conditions were added to prevent the program from calling the corresponding enhancement function if the given factor is equal to 1.0. This is because the resulting image is still the same either way, but this time, the process is more computationally efficient.
3) Another optimization that has been applied is when it comes to the loading of the image files. Initially, before all the threads are even started, the program would first load all the actual images - along with their file name and format - and store them inside the global `images` queue with the use of the `get_images(path)` function. As this loading process sometimes causes delays when loading a large number of images, it is also causing some performance overhead. To resolve this, instead of loading the images before the threads are started, this task is delegated to the threads themselves instead. With this, the `get_images(path)` function now loads the directory of the image file in place of the actual image matrix itself. As for the actual images, these are loaded later on one by one in each thread as defined in Algorithm 3.
4) With the initial implementation, it was observed that when given a large number of images to enhance, increasing the number of threads does not improve the program's performance anymore. This is mainly due to the **Global Interpreter Lock (GIL)** of Python which only allows one thread at a time to hold the control of the Python interpreter. Given that the threads in the program are CPU-bound as they enhance images which require many computations, the GIL restricts the program from using the processor's full potential especially if it has multiple cores - creating a performance bottleneck. With this said, the implementation was changed to use multiple **processes** instead of threads, bypassing the limitations of the GIL as each process has its own GIL. In exchange for larger memory consumption, this significantly increased the performance speed of the program as true parallelism occurs with multiple processes. Changes were made to the new program implementation such as using the `multiprocessing` module of Python instead of the `threading` module and moving the location of the global variables as needed. The program's algorithms that were discussed in the Program Implementation section still remained the same and were used for the new implementation.

III. Result

The experiment setup involves testing the task first on a single thread/process only followed by two implementations - one that uses multiple threads and the other with multiple processes. The setup used was as follows:

- Number of images (mix of jpgs, pngs, and gifs) - **5978**
- Brightness enhancement factor - **1.5**
- Sharpness enhancement factor - **0.2**
- Contrast enhancement factor - **0.1**
- Enhancing time minutes - **[1.0, 3.5, 10.0]**
- Number of threads/processes - **[1, 2, 4, 8, 12]**

From the setup, the number of images and the values of all enhancement factors are constants. Meanwhile, the enhancing time minutes and the number of threads/processes change through the given range of values per test. Table I shows the program's performance results for each test when a single thread/process was used only. The input format as shown in each row of the table is as follows: **number of threads/processes, enhancing time in minutes, if brightness is modified (b), if sharpness is modified (s),** and **if contrast is modified (c)**. Each performance result shows the elapsed time and the number of images enhanced.

It is observed that within 1 minute, the main thread/process alone can enhance **4134** out of 5978 images. This is considering that the brightness is the only enhancement factor adjusted. When the other factors are adjusted as well, the number of images enhanced began to lower as seen in tests #2 and #3. Overall, it would take at least **6** minutes for a single thread to

#	Input	With main thread/process only
1	1, 1.0, b	1 min (60.00 secs), 4134
2	1, 1.0, b, s	1 min (60.33 secs), 2194
3	1, 1.0, b, s, c	1 min (60.02 secs), 1774
4	1, 10.0, b, s, c	5.9 mins (357.50 secs), 5978

TABLE I
SINGLE THREAD/PROCESS PERFORMANCE

enhance a total of 5978 images if all 3 enhancement factors are adjusted, as seen in test #4. Now with multiple threads and processes implemented, Table II shows the performance results of both implementations for each test.

#	Input	With threads	With processes
1	2, 1.0, b	1 min (60.10 secs), 5543	1 min (60.57 secs), 5574
2	2, 1.0, b, s	1 min (60.01 secs), 3683	1 min (60.15 secs), 4228
3	2, 1.0, b, s, c	1 min (60.02 secs), 2990	1 min (60.24 secs), 3029
4	2, 3.5, b, s, c	3.5 mins (210.21 secs), 5882	3.2 mins (190.07 secs), 5978
5	4, 1.0, b, s	1 min (60.03 secs), 4706	1 min (60.28 secs), 5587
6	4, 1.0, b, s, c	1 min (60.02 secs), 3220	1 min (60.13 secs), 4940
7	4, 3.5, b, s, c	3.5 mins (210.37 secs), 5949	2.4 mins (146.01 secs), 5978
8	8, 1.0, b, s, c	1 min (60.12 secs), 3414	1 min (61.69 secs), 5614
9	8, 3.5, b, s, c	3.5 mins (211.55 secs), 5938	1.8 mins (105.29 secs), 5978
10	12, 1.0, b, s, c	1 min (60.10 secs), 3443	1 min (61.77 secs), 5574
11	12, 3.5, b, s, c	3.5 mins (212.18 secs), 5917	1.5 min (92.28 secs), 5978

TABLE II
MULTIPLE THREADS AND PROCESSES PERFORMANCE

Based on the results, it is observed that using multiple processes produced better results compared to using multiple threads. This became more evident when the number of threads and processes were increased. Especially in tests #8 and #11 where both implementations were given 3.5 minutes to enhance all 5978 images, it took only under **2** minutes for multiple processes to enhance all images while for multiple threads, they were not able to enhance all images even after 3.5 minutes. It shows that bypassing the constraint imposed by GIL, and having true parallelism with the usage of multiple processes, further improves the program's performance. Nonetheless, both implementations performed better than a single thread/process. The results proved that these techniques could improve and speed up the task and other tasks that are similar to it.

IV. CONCLUSION

In conclusion, parallel programming techniques were used to speed up the task by enhancing multiple images simultaneously instead of one image at a time. In exchange for a larger use of resources, the program's performance improved drastically. These techniques are one of the ways to speed up the program's performance along with the optimization of algorithms. With this said, it should be ensured however that all the threads and processes are synchronized together to avoid interference. This is done by using synchronization objects such as locks and queues.

As Python was the programming language used, using multiple processes instead of multiple threads further improved the performance as it bypasses the constraint imposed by the GIL of Python. With multiple processes, true parallelism is achieved thus further speeding up the program's performance.

REFERENCES

[1] ImageEnhance Module. Pillow Documentation. Retrieved from https://pillow.readthedocs.io/en/stable/reference/ImageEnhance.html.

[2] multiprocessing - Process-based parallelism. Python 3 Documentation. Retrieved from https://docs.python.org/3/library/multiprocessing.html.

[3] queue - A synchronized queue class. Python 3 Documentation. Retrieved from https://docs.python.org/3/library/queue.html.

[4] threading - Thread-based parallelism. Python 3 Documentation. Retrieved from https://docs.python.org/3/library/threading.html.

[5] time - Time access and conversions. Python 3 Documentation. Retrieved from https://docs.python.org/3/library/time.html.

[6] What Is the Python Global Interpreter Lock (GIL)? Real Python. Retrieved from https://realpython.com/python-gil.