Parallel Computing 2023-24.
Image Renderer project using Python multiprocessing

Elizaveta Mochalova
elizaveta.mochalova@edu.unifi.it
The University of Florence

Abstract

This project studies ...

1. Introduction

This project implements image loading and rendering using OpenCV library, multiprocessing, and asynchronous programming. The scope of this project is to compare sequential and parallel image processing approaches and measure performance speedup. This project demonstrates how parallelization and asynchronous programming can optimize image processing tasks.

1.1. Basic approach

This program loads images from a specified folder adding them to an image list. After that it uses at first sequential and later parallel methods to load images, that can then be displayed or flipped. For both methods we compute the rendering and total computation time, calculating the speedup achieved through parallel execution. We iterate over both approaches 10 times for a more accurate result. At the core of this project is the *ImageManager* class, a common feature for both sequential and parallel image loading methods. It provides a way to manage images, apply transformations, and handle concurrent access.

1.2. OpenCV library

OpenCV or Open Source Computer Vision Library is an open-source library designed for real-time computer vision and image processing. This project uses it to handle image loading, manipulation, and display.The key OpenCV functions used are:

- Image loading

  cv2.imread(filepath) loads an image from the given filepath, it returns a NumPy array representing the image.

- Image resizing

  cv2.resize(image, (new_width, new_height)) resizes an image to the specified width and height while maintaining the aspect ratio.

- Image flipping

  cv2.flip(image, flip_code) mirrors the image along a given axis based on flip_code, so 0 for vertical, 1 for horizontal, -1 for both.

- Displaying images

  cv2.imshow(window_name, image) opens a window and displays the given image.

1.3. Image Transformations

There are multiple methods that get applied to the images, some mentioned above.

- Resizing *resize_image()*

  This steps checks that images fit within a predefined display size 800x600.


Figure 1: Example of images


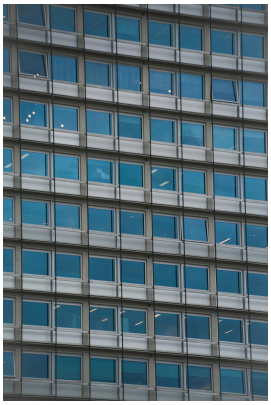

Figure 2: Example of images

- Flipping *flip_image()*

  This method mirrors images based on a specified flip code, in our case -1, so that both transformations are applied.

- Rendering *display_image(), render_image()*

  It displays images using OpenCV.

Some of the images used can be seen in Image 1 and Image 2, their size is between 2 and 10 MB.

1.4. Sequential image loading

The method applies the basic approach of using the given folder and retrieving image file paths. Each image is read and then stored in a shared dictionary. After loading, the images can be displayed sequentially with each image undergoing transformations such as resizing and flipping.

1.5. Parallel image loading

In this step the images are loaded in parallel approach. Parallel execution in this project consists of two techniques - Multiprocessing for Image Loading and AsyncIO for Image Rendering.

Using the first technique, instead of loading images one by one, multiple worker processes are created to load images concurrently. It uses a shared dictionary to store the loaded images and a lock mechanism for safe updates to shared memory.

The process class is used to create parallel workers, each process loads one image and adds it to the shared dictionary.

```
p.start()
p.join()
```

The rendering step is more I/O-bound so it can benefit from asynchronous execution. We use *async def render_image(filename)* for non-blocking execution. This command schedules multiple image rendering tasks concurrently.

```
asyncio.gather(*tasks)
```

2. Results and conclusions

The results that can be seen in Table 1 confirm that using parallel execution improves the performance of both image loading and rendering tasks in this project.

The render speedup is over 2x faster in most runs, while the total speedup is around 2x. The combination of multiprocessing and asynchronous rendering utilizes system resources in an effective way, providing noticeable performance gains without significant complexity.

The render speedup fluctuates across the runs, but generally it can be seen that asynchronous rendering with asyncio provides a notable performance boost. The total speedup values show the overall performance improvement. The total speedup also fluctuates, and there are some runs where the speedup is slightly lower (around 1.6x). The reason for this can be the overhead for managing multiple processes for image loading or

other factors like hardware limitations or system load during execution.

Render Speedup	Total Speedup
2.416	2.080
1.850	1.827
2.124	2.039
2.346	2.241
1.887	1.864
1.936	1.597
2.276	2.218
2.350	2.141
2.802	2.683
2.927	2.514

Table 1: Render and Total Speedup

Overall, while sequential execution is a straightforward approach, it is inefficient for handling large numbers of images, making parallel processing a better alternative for performance optimization. But we also need to consider some of the challenges that come when using multiprocessing and asynchronous execution, such as overhead because managing shared resources requires extra synchronization, more memory usage since each process has its own memory space, complexity.