Image Reader Project Report

Mattia Marilli

Abstract

This project addresses the challenge of optimizing image reading performance in Python. Recognizing the potential bottlenecks in this process, we investigate and evaluate different parallelization techniques to improve efficiency. Our approach involves implementing and comparing various methods, analyzing their impact on the image reading pipeline. The goal is to identify and understand effective strategies for enhancing overall performance in Python image processing applications.

1. Introduction

This project explores the impact of parallelization techniques on improving the efficiency of an *image reader* in Python. The primary focus is on identifying the time-consuming phases of image loading and leveraging Python's capabilities to optimize these processes. Image loading typically involves two critical stages that can significantly affect performance:

- **Memory Loading**: The process of reading image data from storage into memory, which can be slowed down by I/O bottlenecks.
- **Decompression**: The computational task of decoding compressed image data, which can be resource-intensive, especially for large or high-resolution images.

To address these challenges, the project investigates two approaches: a **sequential version** and a **parallel version**. The sequential approach processes images one at a time, while the parallel approach employs advanced techniques to distribute the workload and improve efficiency.

The project was developed in Python, which provides tools to optimize both memory loading and decompression, but it also has inherent limitations, such as the Global Interpreter Lock (GIL), which restricts true parallel execution in certain scenarios. To overcome these limitations, the project leverages:

- **Asynchronous Programming**: For memory loading, asynchronous I/O operations can be used to reduce idle time and improve efficiency in I/O-bound tasks. By allowing non-blocking operations, the program can initiate multiple file reads concurrently, minimizing wait times and making better use of available resources.
- **Multiprocessing**: For decompression, multiprocessing enables the distribution of computational tasks across multiple CPU cores. This is particularly beneficial for CPU-bound tasks, as it bypasses the GIL and allows for true parallel execution, significantly reducing processing time for large datasets.

The sequential approach serves as a baseline, performing tasks without any parallelism. In contrast, the parallel approach aims to maximize performance by utilizing asynchronous programming for memory loading and multiprocessing for decompression. By combining these techniques, the project seeks to demonstrate how Python's capabilities can be effectively used to optimize image reading, despite its limitations. This analysis provides insights into the trade-offs between sequential and parallel execution and highlights strategies for improving performance in both I/O-bound and CPU-bound scenarios. The project will evaluate these strategies through a series of performance metrics, including execution time, speedup, efficiency, to quantify the benefits of parallelization and identify potential bottlenecks.

2. Objectives

The main objective of this project is to design and implement an efficient *image reader* in Python, leveraging parallelization techniques to optimize performance. Specifically, the project aims to:

- **Implement an image reader with parallelization support**: Develop a Python-based image reader that efficiently handles memory loading and decompression, integrating parallel processing techniques.
- **Optimize memory loading with asynchronous programming**: Assess how Async I/O can reduce wait times and enhance efficiency when retrieving images from the storage.

- **Accelerate image decompression using multiprocessing**: Distribute the decompression workload across multiple CPU cores to improve performance and resource utilization.

- **Analyze performance gains of parallelization**: Compare sequential and parallel implementations by measuring key performance metrics such as loading times and decompression speeds.

3. Implementation

To analyze the two parallelization approaches separately, two versions of the image reader were created. Each version implements a different technique to address the two critical phases of image loading: memory loading and decompression.

3.1. Async I/O Image reader

In this implementation, the image loader is designed to load all images into memory as compressed byte streams. The decompression occurs on demand, only when an image is required. This approach makes use of asynchronous programming to perform the image loading concurrently, ensuring that the execution of the program is not blocked.

Moreover, it guarantees that I/O operations can be performed in parallel, enabling multiple images to be loaded simultaneously without waiting for each one to finish before starting the next. While the program waits for the images to be retrieved from disk, it can continue to execute other tasks, maximizing the utilization of available resources.

3.1.1 Sequential Implementation

In this version, the loader reads the images one by one in a blocking way. The `load_images` method iterates over each file in the given directory and opens each image file using standard synchronous file I/O (`open(file_path)`). After reading the contents of the file, it stores the image data in memory as a `BytesIO` object. The program must wait for each image to finish loading before proceeding to the next one.

The `get_image` method can then be used to retrieve an image from memory, where it is decompressed when required. Although simple and easy to implement, this approach does not take full advantage of the system's resources, as the CPU is idle while waiting for the I/O operations to complete. As a result, the sequential implementation is less efficient for larger datasets where multiple images need to be loaded simultaneously.

3.1.2 Parallel Implementation

On the contrary, this version leverages asynchronous programming to load images in parallel. The `load_images` method defines asynchronous tasks for loading image files using the `load_image` coroutine, which reads files asynchronously with `aiofiles.open(file_path)`. This allows multiple image files to be read at the same time, without blocking the execution of other tasks. The `asyncio.gather(*tasks)` method is used to launch all image loading tasks, allowing the program to perform other operations while some images are being read from disk. This implementation is particularly beneficial for I/O-bound tasks because it ensures that while waiting for I/O operations to complete (such as disk reads), the program can continue processing other tasks instead of sitting idle.

After all images are loaded into memory as byte streams, the 'get_image' method can be used to access each image, decompressing it only when necessary.

3.1.3 Expectations of Asynchronous Implementation

In the asynchronous implementation, the key advantage comes from the ability to launch multiple tasks that can run in parallel during I/O operations like reading image files from disk.

When the `load_images` function is called, it creates a task for each image file. These tasks are launched sequentially, but as soon as one task reaches the point where it needs to wait for I/O (such as reading data from a file), the event loop can move on to the next task without delay. Since each iteration of the sequential loop we aim to parallelize with 'asyncio', is entirely I/O-bound, every task is also predominantly I/O-bound. This means that while one task is waiting for I/O, other tasks can run concurrently, all attempting to load images in parallel, leading to a scenario like the one shown in Figure 1.

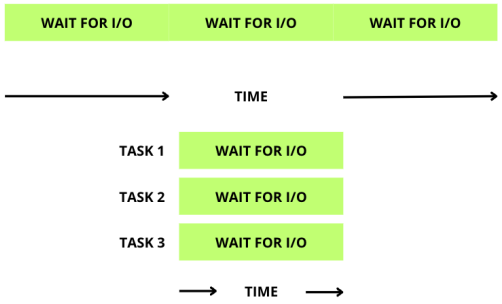

Figure 1. Illustration of I/O parallelism in asynchronous image loading.

Once all the tasks are launched, they are essentially working in parallel during their I/O waiting periods. While one task waits for an image to be read from disk, other tasks can continue their own I/O operations, loading their respective images. This should reduce the total time spent waiting for I/O to complete because multiple image-loading operations are happening simultaneously. As a result, the program can load multiple images at once, hopefully,making the process faster than the sequential approach, where each image is loaded one by one.

3.2. Multiprocessing Image Reader

In this implementation, the image loader is designed to take the file path of each image and directly decompress it using the **multiprocessing** module. Unlike the Async I/O version, which loads the entire image dataset into memory before decompression, this approach leverages multiple processes to perform decompression in parallel, effectively bypassing the Global Interpreter Lock (GIL).

Since image decompression is a computationally intensive task, utilizing **multiprocessing** allows the workload to be distributed across multiple CPU cores. Each core handles a separate image decompression task independently, maximizing CPU utilization and significantly reducing the total processing time compared to a sequential approach. The ability of **multiprocessing** to run processes in parallel ensures that large sets of images can be loaded and decompressed efficiently.

3.2.1 Sequential Implementation

In the sequential implementation, each image is processed one at a time. The `decompress_images` method iterates through the list of image paths provided as input. For each image, it opens the file using the `Image.open(file_path)` method from the Pillow library, ensuring that the file is properly closed after processing by using a `with` statement. This ensures resource management is handled efficiently. After opening the image, the method converts it to RGB format with `img.convert("RGB")`.

While this approach is straightforward and easy to implement, it does not leverage any form of parallelism. Consequently, when dealing with large datasets, the performance may degrade, increasing the overall processing time. Unlike parallel approaches, this implementation does not take advantage of multiple CPU cores, which limits its efficiency in scenarios requiring rapid image decompression.

3.2.2 Parallel Implementation

In the parallel implementation, we try to improve performance utilizing multiple CPU cores to process images concurrently. The `decompress_images` method in this class leverages the `ProcessPoolExecutor` from the `concurrent.futures` module to distribute decompression tasks across multiple processes. Specifically, the `executor.map(self._decompress_image, image_paths)` function maps each image path from the list of `image_paths` to the `_decompress_image` method, which is executed in parallel by separate processes.

The `_decompress_image` method is responsible for handling the decompression of each individual image. For each image, the method opens the file using `Image.open(file_path)` and then converts it to RGB format with `img.convert("RGB")`. This method is designed to be executed by a separate process, meaning that each image is decompressed in isolation, which allows for true parallel execution.

The `ProcessPoolExecutor` manages a pool of worker processes, with the number of processes controlled by the `max_workers` parameter. By setting `max_workers=self.num_processes`, the implementation distributes the decompression workload across the available CPU cores, maximizing resource utilization.

3.3. Expectations of Multiprocess Implementation

The main expectation from the multiprocessing implementation is a significant reduction in decompression time, especially when handling large volumes of images. By leveraging multiple CPU cores, the system can process several images simultaneously, making it ideal for workloads that are computationally intensive, such as image decompression. This parallelization is expected to speed up the overall process, allowing for more efficient handling of large datasets compared to the sequential approach.

However, it is important to note that while multiprocessing provides a performance boost, it also introduces some overhead. Each process requires its own memory space and incurs the cost of inter-process communication. This overhead can become more noticeable as the number of processes increases, especially when working with smaller datasets.

In summary, the multiprocessing approach is expected to deliver the best results for large volumes of images due to its ability to exploit multiple CPU cores, but it does come with additional overhead compared to multithreading. This

trade-off needs to be considered when deciding the most suitable parallelization strategy for different workloads.

4. Experimentation and Results

4.1. Async I/O Image Reader

4.1.1 Test Setup

To ensure a rigorous and reliable performance evaluation, the benchmark was conducted using a progressively increasing number of jpeg images, specifically testing with sets of 50, 100, 300, 500, 700, and 900 images (with a medium size of 4MB for each image). This step was crucial in analyzing how the system scales when handling varying workloads, allowing us to better understand the efficiency and behavior under different conditions.

Each test was executed ten times for every dataset size to mitigate the impact of potential outliers and fluctuations in performance due to external factors. By averaging the results across multiple runs, we aimed to obtain a more accurate and consistent measurement of execution times.

A key factor in the benchmarking process was minimizing the influence of filesystem caching mechanisms, which could otherwise distort the results. To address this issue, we implemented a cache saturation step before each test execution. This was achieved by creating, writing, and subsequently reading a temporary 3 GB file, thereby forcing the operating system to evict previously cached data. This approach aimed to simulate a real-world scenario in which images are loaded from disk rather than being retrieved from a cache.

For each execution, we evaluate the average execution time for each dataset size, along with the corresponding speedup factor. The collected data highlights the relative efficiency of different loading strategies and serves as a foundation for further analysis.

4.1.2 Performance Results

The results of our benchmarking experiments are summarized in Table 1, which presents the average execution times for both sequential and asynchronous image loading approaches across different dataset sizes. The table also reports the computed speedup.

Contrary to initial expectations, the asynchronous approach does not provide a substantial speedup over the sequential method. The recorded speedup values remain close to 1.0 in most cases, indicating that the two approaches yield nearly identical performance.

# Images	Seq. (s)	Async (s)	Speedup
50	0.19	0.19	1.02x
100	0.44	0.41	1.08x
300	1.41	1.39	1.02x
500	2.13	2.05	1.04x
700	3.04	2.90	1.05x
900	4.00	4.31	0.93x

Table 1. Comparison of execution times between sequential and asynchronous image loading.

In fact, for the largest dataset of 900 images, the asynchronous approach is slightly slower than the sequential one, resulting in a speedup below 1.0.

This outcome can be attributed to the way disk I/O operations are handled. While it is true that an asynchronous approach allows multiple tasks to request and process image loading in parallel, the overall read performance is still constrained by the disk's capabilities. The time taken by the i-th task to complete its read operation is influenced by the previous $i - 1$ reads, as all requests must still be served by the same underlying storage medium. Consequently, the total execution time remains largely unchanged, regardless of whether the operations are scheduled sequentially or concurrently. Figure 2 provides a visual representation of this phenomenon, illustrating how concurrent read requests are ultimately serialized by the disk, leading to minimal performance improvements.

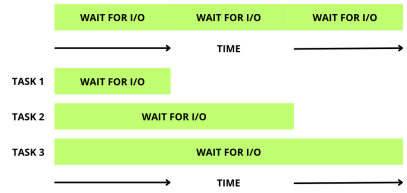

Figure 2. Illustration of disk I/O behavior under sequential and asynchronous approaches.

This behavior stands in contrast to other I/O-bound tasks, such as handling HTTP requests, where an asynchronous approach can provide significant speedup.When reading images from a single storage device, the disk remains the limiting factor, preventing full exploitation of asynchronous processing.

Furthermore, the overhead introduced by the asynchronous framework, may contribute additional delays. In some scenarios, as observed in the case of 900 images, this overhead can outweigh any potential benefits, leading to a slight degradation in performance.

4.2. Multiprocessing Image Loader

4.2.1 Test Setup

To ensure a comprehensive and accurate performance evaluation, the benchmarking was performed with a progressively increasing number of images, specifically testing datasets containing 50, 100, 300, 500, 700 and 900 images (with medium size of 4MB for each image). This varied dataset size allowed for a detailed analysis of how the system scales with an increasing number of images, providing insights into the efficiency and scalability of both sequential and parallel loading mechanisms.

Each test was executed ten times for every dataset size, with the benchmarking process for the sequential implementation being repeated multiple times to account for variability. The average execution time for the sequential decompression was recorded to establish a baseline performance. Subsequently, parallel decompression tests were carried out with varying numbers of processes, ranging from 1 to 16 processes. This approach aimed to evaluate the impact of parallelism on performance under different levels of CPU utilization.

For each execution, the average execution time for each dataset size was evaluated for both sequential and parallel implementations, alongside the corresponding speedup factor and efficiency. This data offers valuable insights into the relative performance of sequential and parallel decompression approaches and provides a foundation for further analysis and optimization of image loading strategies.

All tests were executed on a Mac M1 with 4 high-efficiency cores and 4 high-performance cores. This is an important aspect as the heterogeneous architecture of the M1 chip—combining high-performance and high-efficiency cores—impacts the test results and highlights how the program leverages these resources. The effects of this architecture will be visible in the tests, particularly in scenarios with varying thread counts.

Images	Proc	Seq. (s)	Par. (s)	Speedup	Efficiency
50	1	5.97	6.30	0.95x	0.95
50	2	5.80	3.22	1.80x	0.90
50	4	5.80	1.81	3.21x	0.80
50	6	5.80	1.59	3.64x	0.61
50	8	5.80	1.57	3.69x	0.46
50	16	5.73	4.47	1.28x	0.08
100	1	12.91	14.22	0.91x	0.91
100	2	12.56	6.75	1.86x	0.93
100	4	12.56	3.80	3.30x	0.83
100	6	12.56	3.25	3.86x	0.64
100	8	12.56	3.08	4.07x	0.51
100	16	12.49	7.72	1.62x	0.10
300	1	44.18	44.27	1.00x	1.00
300	2	43.05	22.86	1.88x	0.94
300	4	43.05	12.74	3.38x	0.84
300	6	43.05	10.86	3.96x	0.66
300	8	43.05	9.67	4.45x	0.56
300	16	42.57	23.90	1.78x	0.11
500	1	64.48	64.83	0.99x	0.99
500	2	64.15	33.77	1.90x	0.95
500	4	64.15	18.55	3.46x	0.86
500	6	64.15	15.94	4.02x	0.67
500	8	64.15	13.86	4.63x	0.58
500	16	64.14	40.21	1.60x	0.10
700	1	88.63	88.84	1.00x	1.00
700	2	88.21	46.42	1.90x	0.95
700	4	88.21	25.54	3.45x	0.86
700	6	88.21	21.62	4.08x	0.68
700	8	88.21	19.28	4.58x	0.57
700	16	88.56	45.46	1.95x	0.12
900	1	106.55	110.12	0.97x	0.97
900	2	106.49	55.68	1.91x	0.96
900	4	106.49	30.84	3.45x	0.86
900	6	106.49	26.10	4.08x	0.68
900	8	106.49	23.14	4.60x	0.58
900	16	105.74	57.48	1.83x	0.11

Table 2. Comparison of execution times between sequential and parallel image loading.

4.2.2 Analysis of the Results

The results of our benchmarking experiments are summarized in Table 2, which presents the average execution times for both sequential and parallel processing approaches across different dataset sizes and numbers of processes. The table also reports the average speedup and the average efficiency.

As expected, the parallel approach shows clear improvements in performance compared to the sequential method, especially as the number of processes increases. For the smallest dataset of 50 images, the speedup is modest, but as the dataset size grows, parallel processing provides significantly better results. For instance, the speedup for the 500-image dataset with 8 processes reaches up to 4.63x, showing substantial reductions in processing time.

The speedup values demonstrate diminishing returns as the number of processes increases. This trend is further reflected in the efficiency metric, which decreases as more processes are added. For example, the efficiency of 0.95 for 2 processes drops to 0.46 for 8 processes with the 50-image dataset, suggesting that while parallelization improves performance, adding more processes beyond a certain point does not result in a proportional increase in performance.

Figure 3 visually represents the speedup observed for different dataset sizes and process counts, showing how efficiency decreases as the number of processes grows.

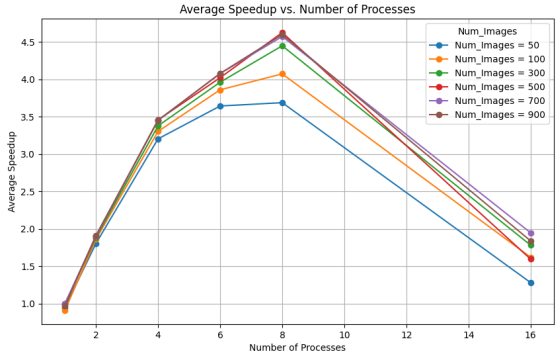

Figure 3. Speedup of parallel image loading over sequential loading across different dataset sizes.

These results are consistent with the architecture on which the benchmarking experiments were conducted. The system's computational resources, including the number of cores available, align with the observed diminishing efficiency. In particular, the performance gains observed with larger datasets support the notion that the benefits of parallelization are more pronounced when sufficient resources are available to handle the additional processes. However, with 16 processes, performance degrades significantly due to the overhead introduced by excessive thread management and resource contention, especially since this utilizes double the available cores.

These results confirm that parallelization is particularly effective for larger datasets, as it makes better use of available computational resources. However, the diminishing efficiency indicates that, as the number of processes increases, the overhead from task management and coordination starts to reduce the performance benefits.

5. Conclusions

The project implemented and evaluated both asynchronous I/O and multiprocessing techniques for optimizing image reading in Python. While the asynchronous I/O approach, intended to improve I/O-bound performance, yielded minimal gains due to inherent limitations in disk I/O parallelism and the overhead of the asynchronous framework, the multiprocessing implementation demonstrated significant performance improvements, particularly for larger datasets. Multiprocessing effectively leveraged multiple CPU cores for the computationally intensive decompression task.

These findings underscore the effectiveness of multiprocessing for CPU-bound tasks like image decompression and provide valuable insights into the trade-offs between different parallelization strategies in Python for image loading optimization.