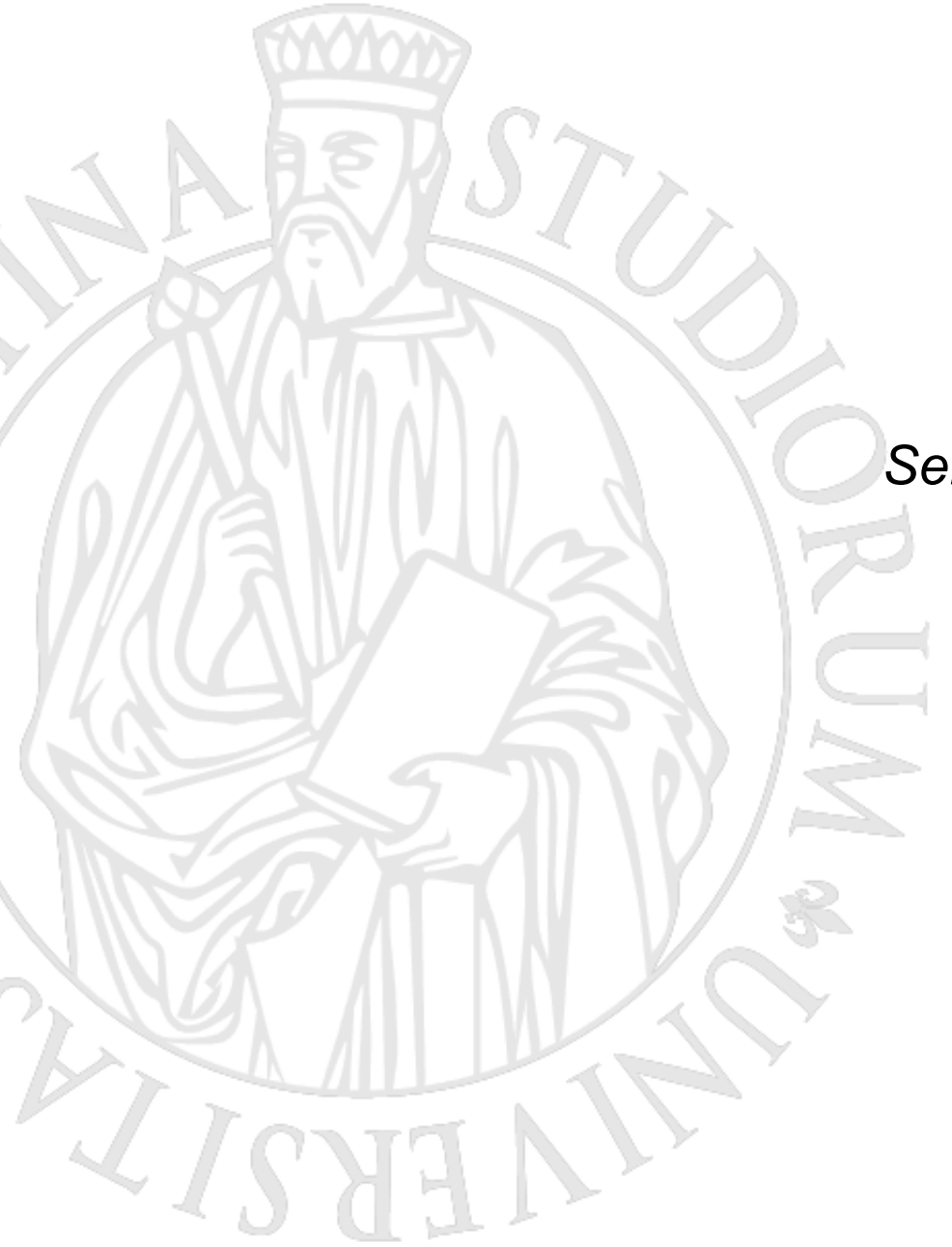


Image Reader

Serial vs Parallel Approach with Python for Image Reading

Mattia Marilli

A.A. 2024/2025

Table of Content

INTRODUCTION

Optimizing Image Loading with Parallelization in Python

Critical Phases in Image Reading

Image reading involves memory loading (reading data into memory) and decompression (decoding compressed data). Both stages can slow performance due to I/O bottlenecks and intensive computation.

Optimization Techniques

Asynchronous programming speeds up memory loading by allowing concurrent file reads, while multiprocessing speeds up decompression by using multiple CPU cores and bypassing Python's GIL.

Sequential and Parallel Versions

Implement both a sequential and a parallel version. The performance of both approaches will be analyzed to assess the impact of parallelization.

OBJECTIVES

Project Goals and Key Tasks

O1

Develop a Python-based image reader that efficiently handles memory loading and decompression, integrating parallel processing techniques.

O2

Assess how Async I/O can reduce wait times and enhance efficiency when retrieving images from the storage.

O3

Distribute the decompression workload across multiple CPU cores to improve performance and resource utilization.

O4

Compare sequential and parallel implementations by measuring key performance metrics such as loading times and decompression speeds.

IMPLEMENTATION

Async I/O Image Loader

On-Demand Decompression

Images are loaded as compressed byte streams to save memory and reduce initial load time. Decompression happens only when an image is needed, ensuring efficient resource usage.

Asynchronous Execution

Image loading runs concurrently, preventing the main program from blocking. This ensures smooth performance by handling multiple images simultaneously without delays.

Efficient I/O Operations

Parallel disk access allows multiple images to be retrieved at once. While images are being loaded, the program can continue executing other tasks, maximizing overall resource utilization.

Sequential Implementation

Sequential Image Reading

- The load_images method loads each image one by one.
- It uses standard synchronous file I/O operations (open(file path))
- Images are stored as BytesIO objects in memory

Inefficient Resource Utilization

- Does not fully utilize system resources.
- CPU remains idle while waiting for I/O operations to complete.
- Less efficient with larger datasets, where simultaneous image loading would be beneficial.

Parallel Implementation

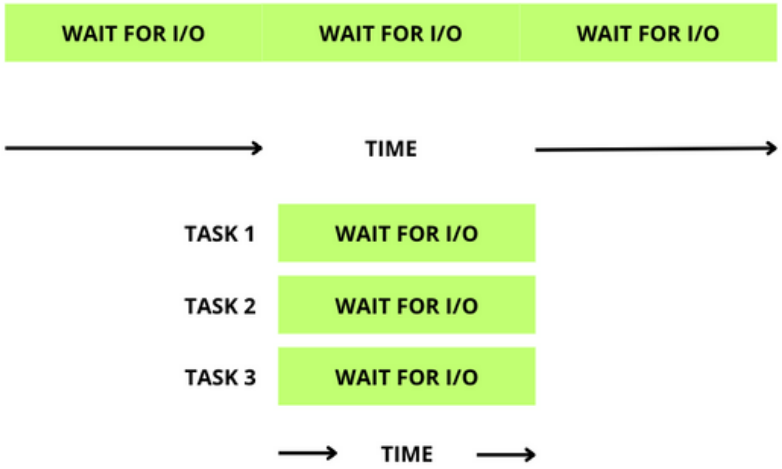
Concurrent Image Loading

- The load_images method defines asynchronous tasks for loading image files.
- It uses the load_image coroutine with aiofiles.open(file_path) for asynchronous file reading.
- Multiple image files are read concurrently without blocking other tasks

Advantages and Benefits

- Leverages asyncio.gather(*tasks) to launch all tasks concurrently.
- Allows the program to perform other operations while waiting for I/O tasks (e.g., disk reads) to complete.
- Ideal for I/O-bound tasks, as it avoids idle CPU time.

Expectations of Asynchronous Implementation



Parallel Task Execution

In the asynchronous implementation, the load_images function creates a task for each image file, which are launched sequentially. When a task reaches an I/O operation, it allows the event loop to switch to the next task, preventing idle time.

Maximizing I/O Efficiency

Since the tasks are predominantly I/O-bound, while one task waits for a file to be read from disk, other tasks can continue their own I/O operations. This ensures that multiple images are loaded in parallel.

Improved Performance with Parallel Loading

By allowing multiple image loading operations to happen in parallel, the overall time spent waiting for I/O to complete should be minimized. This should results in faster image loading compared to the sequential approach.

Multiprocessing Image Loader

Parallel Decompression

Uses the multiprocessing module to decompress images in parallel, distributing the workload across multiple CPU cores.

Maximized CPU Utilization

Each image is processed by a separate process, enhancing resource usage and significantly reducing total processing time.

Parallel Processing

Unlike the Async I/O version, multiprocessing bypasses the GIL, enabling more efficient handling of computationally intensive tasks.

Sequential Implementation

Simple and Easy to Implement

- Each image is processed one by one, making the code straightforward and easy to understand.
- The image is opened using the Image.open(file_path) method from the Pillow library, ensuring it is ready for further processing.
- After opening, the image is converted to the RGB format using img.convert("RGB"), ensuring consistency in color representation.

Limitations

- The approach operates sequentially, meaning each image is processed one after another, which can lead to slower performance when dealing with large datasets.
- Unlike parallel implementations, this method does not take advantage of multiple CPU cores.

Parallel Implementation

Improved Performance

- Utilizes the ProcessPoolExecutor from the concurrent.futures module to distribute image decompression tasks across multiple processes.
- The executor.map(self.decompress_image, image_paths) function maps each image path to the decompress_image method, which is executed in parallel by separate processes.
- Each image is processed in isolation, maximizing parallelism and improving performance.

Benefits and Resource Optimization

- The decompress_image method handles the decompression of each image, opening the file with Image.open(file_path) and converting with img.convert("RGB").
- The ProcessPoolExecutor controls the number of worker processes via the max_workers parameter.

Expectations of Multiprocess Implementation

Significant Performance Improvement

By loading images in a separate task, we can fully utilize the system's multi-core capabilities. This approach distributes the computational load across multiple cores, reducing processing time and enhancing the overall time of image processing.

Efficient for Large Datasets

The system can handle large volumes of images efficiently, leveraging multiple CPU cores to speed up processing compared to sequential methods.

Overhead and Trade-Offs

While multiprocessing boosts performance, it introduces memory overhead and inter-process communication costs, especially with smaller datasets. These factors should be considered when choosing parallelization strategies.

EXPERIMENTATION AND RESULTS

Async I/O Image Loader

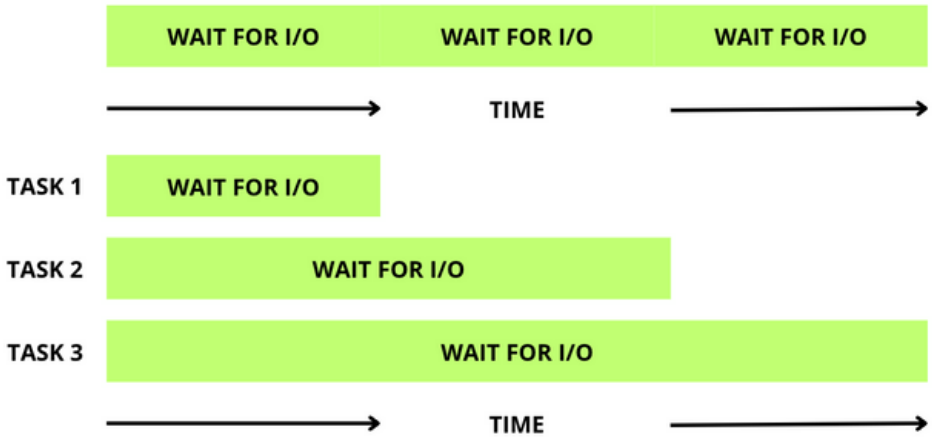
Test Setup

- The benchmark tests were conducted with **progressively increasing image sets**: 50, 100, 300, 500, 700, and 900 images, in order to evaluate how the system performs and scales when handling different workloads.
- Each test executed 10 times per dataset size to **minimize the impact of outliers** and performance fluctuations, ensuring more reliable and consistent results.
- To eliminate the potential influence of filesystem caching, a **cache saturation step** was performed before each test. This was done by creating, writing, and reading a temporary 3 GB file, which forces the operating system to evict previously cached data and simulate real-world image loading scenarios.
- The **performance metrics**, such as average execution times and speedup factors, were collected for each dataset size, allowing for a detailed comparison of the efficiency and effectiveness of the different image loading strategies.

Performance Results

# Images	Sequential (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**

Results Analysis



No Speedup in Asynchronous Approach

The asynchronous image loading method did not provide any noticeable speedup over the sequential method. In fact, for the largest dataset (900 images), it was slightly slower than the sequential approach.

Disk I/O Limitation

Performance was constrained by disk I/O operations, as the storage medium serialized concurrent read requests, preventing any improvement from parallel processing.

Potential Speedup with Other I/O Types

While disk I/O limits performance, other I/O-bound tasks, such as network requests or database queries, could see significant speedup with asynchronous processing due to less reliance on storage read performance.

EXPERIMENTATION AND RESULTS

Multiprocessing Image Loader

Test Setup

- Benchmarking was performed using **datasets of 50, 100, 300, 500, 700, and 900 images** (with medium size of 4MB for each image), allowing for a comprehensive analysis of system scalability as the number of images increased.
- Each test was **executed 10 times** per dataset size, with the sequential implementation being repeated multiple times to ensure reliable results and account for variability.
- Parallel decompression tests were conducted using 1 to 16 processes to **assess the impact of parallelism** on performance, analyzing how different levels of CPU utilization affected execution times.
- All tests were executed on a **Mac M1 with 4 high-efficiency and 4 high-performance cores**.

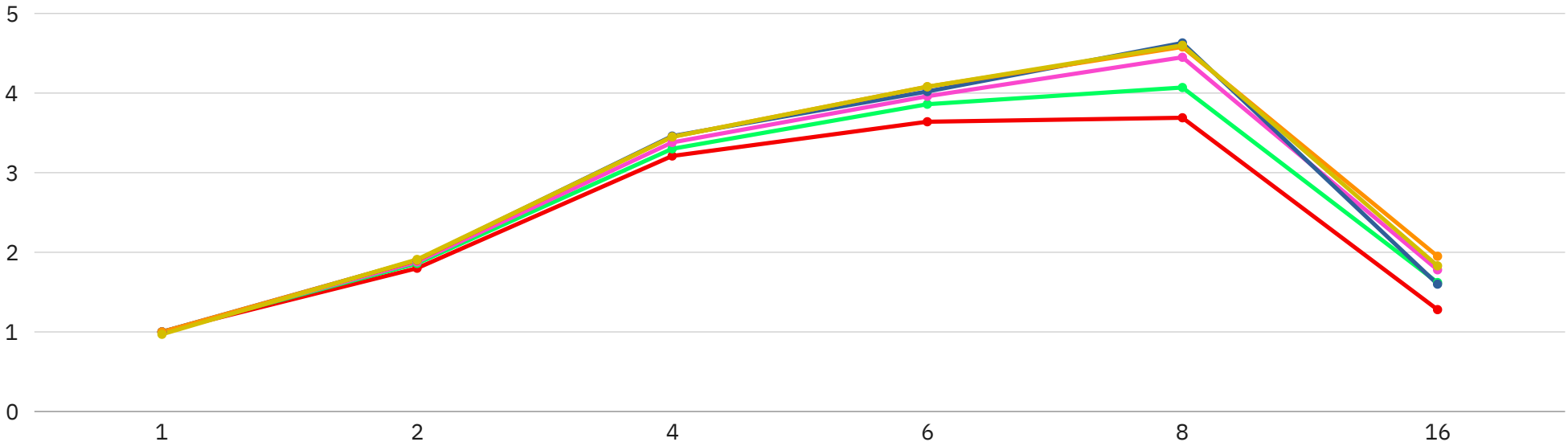
Speedup

The following graph illustrates the speedup achieved with increasing numbers of processes for different amount of images.

Complete Data

Click Here

It shows the relationship between the number of processes (from 1 to 16) and the corresponding speedup for various numbers of images (50, 100, 300, 500, 700, 900).



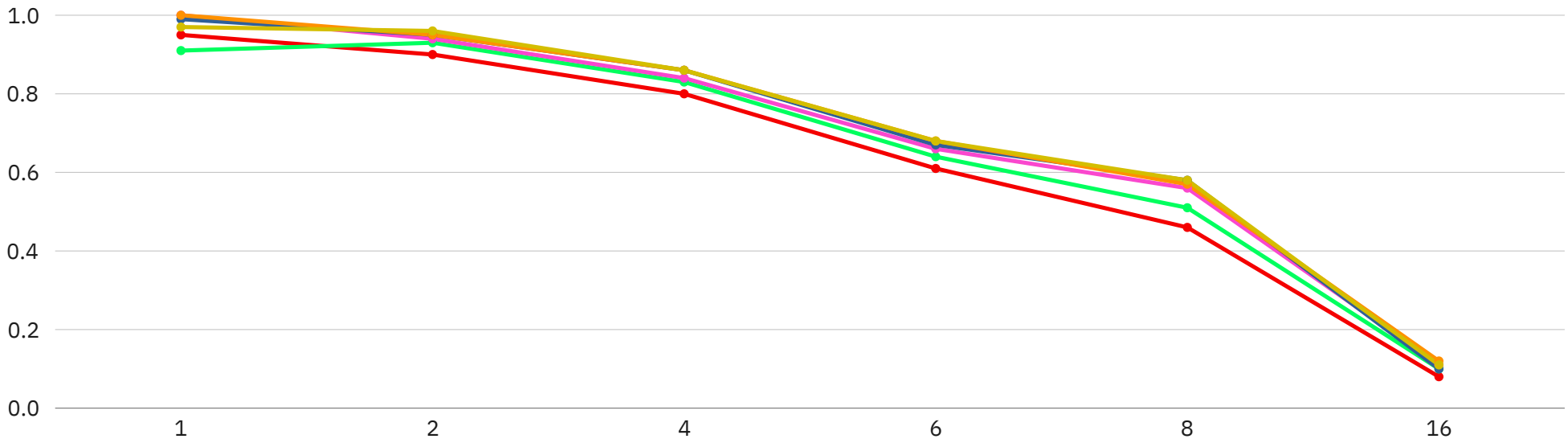
Efficiency

The following graph illustrates the efficiency achieved with increasing numbers of processes for different amount of images.

Complete Data

Click Here

It shows the relationship between the number of processes (from 1 to 16) and the corresponding efficiency for various numbers of images (50, 100, 300, 500, 700, 900).



Results Analysis

Parallel Processing Improvement

As dataset size increases, parallel processing yields significant performance gains, with up to 4.63x speedup for the 500-image dataset with 8 processes.

Diminishing Returns

While speedup improves with more processes, efficiency decreases. For instance, with 50 images, efficiency drops from 0.95 (2 processes) to 0.46 (8 processes), indicating diminishing returns with additional processes.

Resource Utilization

The efficiency reduction aligns with system architecture and available resources. Parallelization is most effective for larger datasets, with diminishing benefits beyond a certain number of processes (e.g., with 16 processes we basically have no efficiency).

Thank you!

Project Repository

Click Here