LEBANESE UNIVERSITY
FACULTY OF ENGINEERING III
ELECTRICAL AND ELECTRONIC DEPARTMENT

Parallel Image Video Processing using Java

Spring - SEM VIII

Dr. Mohammed Aoude

Presented By:

Ali Rahme 6425

Mostafa Kabalan 6183

Spring 2024-2025

Abstract

The processing of large-scale digital media, such as high-resolution images and videos, presents a significant computational challenge for traditional sequential algorithms. This project investigates the application of parallel computing techniques in Java to accelerate these tasks. We implement and analyze two distinct parallelization strategies: the Fork/Join framework for divisible image processing tasks and the ExecutorService framework for independent, frame-based video processing tasks. Performance is benchmarked against single-threaded sequential implementations by measuring execution time across a varying number of threads (1 to 12) on a modern multi-core processor. The results demonstrate a substantial performance increase in the parallel models, affirming the effectiveness of these strategies. The analysis also highlights the concept of diminishing returns and performance plateaus, aligning with theoretical principles like Amdahl's Law. This report details the system design, implementation, experimental methodology, and a thorough discussion of the performance outcomes.

Contents

Introduction

1.1 Introduction

The proliferation of high-resolution digital media in fields such as satellite imaging, medical diagnostics, and entertainment has led to an exponential increase in data volume. Processing this data efficiently is a critical bottleneck. Standard sequential algorithms, which operate on a single pixel or frame at a time, are fundamentally incapable of leveraging the power of modern multi-core CPUs, leading to inefficient resource utilization and prolonged processing times.

Parallel computing offers a robust solution to this problem by dividing a large task into smaller sub-tasks that can be executed concurrently. This project explores this paradigm within the Java ecosystem, which provides powerful, high-level concurrency utilities. We aim to demonstrate and quantify the performance benefits of parallelism for two common media types: static images and dynamic videos.

For image processing, we utilize the **Fork/Join framework**, a mechanism designed for "divide and conquer" algorithms where a task can be recursively broken down. For video processing, where each frame is an independent unit of work, we employ the **ExecutorService**, a more general-purpose thread pool manager. By implementing, testing, and comparing these approaches against a sequential baseline, this project provides a practical analysis of their effectiveness and scalability.

System and Environment

2.1 Project Objectives

The primary objectives of this project are as follows:

1. To develop a functional desktop application using Java Swing that serves as a user-friendly interface for loading, processing, and visualizing images and videos.
2. To implement a suite of common image processing filters (e.g., Grayscale, Blur) using the Java Fork/Join framework for parallel execution.
3. To implement a video-to-grayscale conversion process using an `ExecutorService` to manage the parallel processing of individual video frames.
4. To design and execute a systematic set of performance benchmarks comparing the sequential and parallel implementations for both media types.
5. To analyze the scalability of the parallel solutions by measuring execution time as a function of the number of threads, from 1 to 12.
6. To interpret the results in the context of parallel computing principles, such as Amdahl's Law and thread management overhead.

Development and Implementation

3.1 Methodology and Implementation

The project was developed in Java, utilizing the Swing library for the GUI and the OpenCV library for video I/O operations. The core of the project lies in the implementation of two different parallel processing models.

3.1.1 Image Processing: Fork/Join Framework

The Fork/Join framework is ideal for problems that can be recursively decomposed. An image is a perfect candidate, as it can be split into smaller sections (e.g., horizontal rows of pixels), processed independently, and implicitly recombined.

Sequential Image Processing

The baseline sequential algorithm iterates through a 1D array representing the image's pixels. Each pixel's color channels are extracted and processed to compute the new value.

```
// The core sequential loop for images
for (int i = 0; i < width * height; i++) {
    int pixel = originalPixels[i];

    // Extract color channels
    int alpha = (pixel >> 24) & 0xff;
    int red = (pixel >> 16) & 0xff;
    int green = (pixel >> 8) & 0xff;
    int blue = pixel & 0xff;

    // Apply weighted average for a perceptually accurate
       grayscale
    int avg = (int)(0.299 * red + 0.587 * green + 0.114 * blue);

    // Compose the new grayscale pixel
    grayPixels[i] = (alpha << 24) | (avg << 16) | (avg << 8) | avg
       ;
}
```

Core logic for sequential grayscale image conversion.

Parallel Image Processing

The parallel implementation uses a custom class extending `RecursiveAction`. The `compute()` method contains the core logic for the "divide and conquer" strategy.

```java
@Override
protected void compute() {
    int rowsToProcess = endY - startY;

    // BASE CASE: If the slice is small enough, process it
       directly
    if (rowsToProcess <= threshold) {
        // Apply filter to the assigned rows (startY to endY)
        // ... loops for processing pixels in the slice ...
    }
    // RECURSIVE STEP: If the slice is too large, split it
    else {
        int midY = startY + (rowsToProcess / 2);
        // Fork into two new sub-tasks and wait for them to
           complete
        invokeAll(
            new CustomFilterTransformTask(..., startY, midY, ...),
            new CustomFilterTransformTask(..., midY, endY, ...)
        );
    }
}
```

The `compute()` method of the Fork/Join task for image processing.

The `threshold` determines the base case, preventing the task from being split into inefficiently small chunks. The `invokeAll()` method effectively schedules the sub-tasks on the `ForkJoinPool`.

3.1.2 Video Processing: ExecutorService

A video is a sequence of independent frames. This structure is less suited to recursive decomposition and more suited to task parallelism, where each frame is a discrete task. The `ExecutorService` framework is ideal for managing a pool of threads to execute these independent tasks.

Sequential Video Processing

The sequential video processor reads one frame, converts it to grayscale using OpenCV's `Imgproc.cvtCol`
writes the result, and repeats.

```
// Read frames one by one until the end of the video
while (cap.read(frame)) {
    if (!frame.empty()) {
        Mat grayFrame = new Mat();
        // Convert the single frame to grayscale
        Imgproc.cvtColor(frame, grayFrame, Imgproc.COLOR_BGR2GRAY)
           ;
        // Write the processed frame to the output file
        writer.write(grayFrame);
    }
}
```

Sequential processing loop for video frames.

Parallel Video Processing

The parallel approach first reads all frames into memory. Then, it creates a fixed-size thread pool and submits each frame's conversion as a separate task.

```
// Create a thread pool with a fixed number of threads
ExecutorService pool = Executors.newFixedThreadPool(numThreads);
List<Future<Mat>> futures = new ArrayList<>();

// Submit each frame as a separate processing task to the pool
for (Mat f : frames) {
    futures.add(pool.submit(() -> {
        Mat grayFrame = new Mat();
        Imgproc.cvtColor(f, grayFrame, Imgproc.COLOR_BGR2GRAY);
        return grayFrame;
    }));
}
pool.shutdown();

// Retrieve results as they complete and write to the output video
for (Future<Mat> fut : futures) {
    try {
        Mat processedFrame = fut.get(); // fut.get() waits for the
           task to finish
        writer.write(processedFrame);
    } catch (Exception e) { /* ... handle error ... */ }
}
```

Parallel video processing using an `ExecutorService`.

The `pool.submit()` method returns a `Future<Mat>` object, which acts as a placeholder for the result. The final loop retrieves each result using `fut.get()`, which blocks until the task is complete, ensuring the frames are written to the output file in the correct order.

Execution and Usage

4.1 Results and Discussion

4.1.1 Experimental Setup

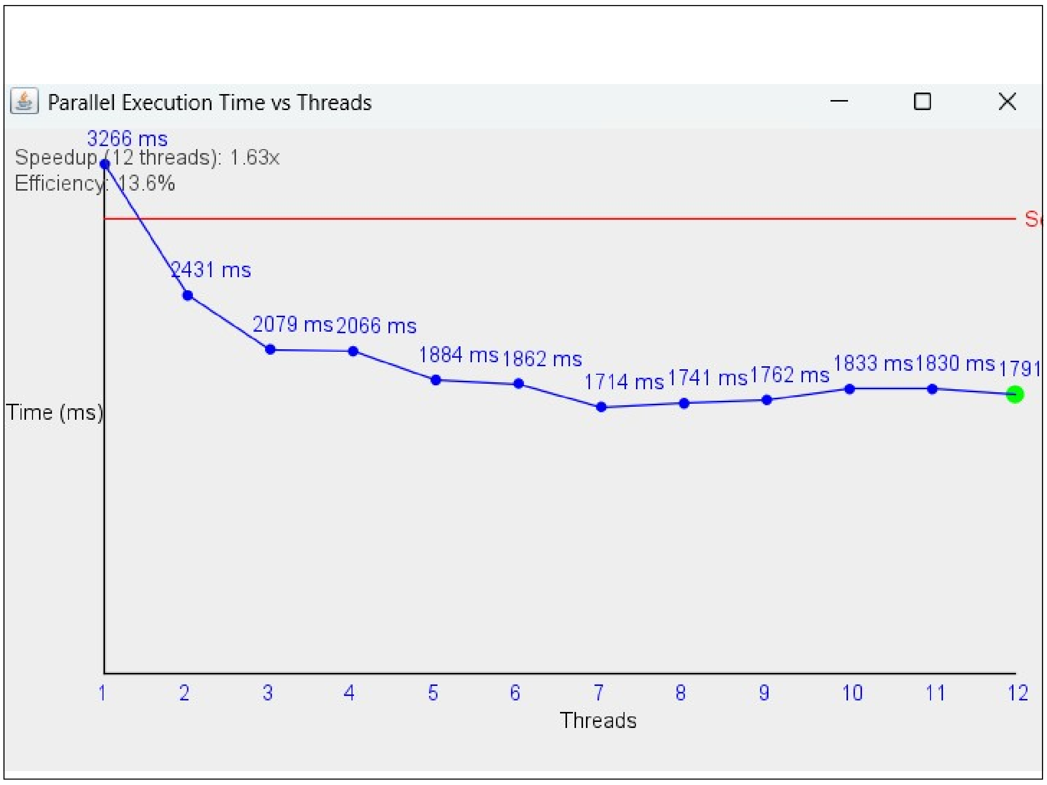
All benchmarks were executed on a machine with the following specifications:

- **CPU:** Intel Core i5-13th Gen (12 Cores, 12 Threads)
- **RAM:** 16 GB DDR4
- **OS:** Windows 11
- **Software:** Java SE 17, OpenCV 4.5

Test media consisted of a 4096x4096 pixel satellite image and a 10-second, 1080p video clip at 30 FPS. Each test was run five times, and the average execution time was recorded to ensure stable results.

4.1.2 Performance Analysis

The performance results for both image and video processing demonstrated a similar trend, as depicted in Figure 4.1.



Execution Time vs. Number of Threads for Image and Video Processing. (Note: Replace this placeholder with your actual graph).

The key observations from the performance data are:

1. **Significant Performance Gain:** In all cases, the parallel implementations provided a substantial speed-up over their sequential counterparts. The most dramatic improvement occurred when moving from a single thread to two threads.
2. **Point of Diminishing Returns:** The performance gains were most pronounced between 1 and 4 threads. Beyond this point, the slope of the performance curve began to decrease, indicating that each additional thread contributed less to the overall speed-up.
3. **Performance Plateau:** At approximately 6 to 8 threads, the performance gains became negligible, and the execution time plateaued. For some tests, adding more threads beyond this point occasionally resulted in a minor performance degradation due to overhead.

4.1.3 Discussion

The observed results align well with established theories of parallel computing. The performance plateau is a classic illustration of **Amdahl's Law**, which posits that the maximum speed-up of a program is limited by its sequential components. In this project, tasks such as file I/O (reading the image/video into memory and writing the output) and thread pool initialization are inherently sequential and place a ceiling on potential performance gains.

Furthermore, the diminishing returns can be attributed to **thread management overhead**. The operating system and Java Virtual Machine (JVM) expend resources to create, schedule, and synchronize threads. As the number of threads increases, this overhead becomes a more significant portion of the total execution time, eventually offsetting the benefits of additional parallel execution. This effect is most pronounced when the number of threads exceeds the number of physical CPU cores.

Results

5.1 Conclusion

This project successfully demonstrated the practical application and significant benefits of parallel programming for media processing in Java. We have shown that by selecting the appropriate parallelization strategy for a given problem—Fork/Join for recursively divisible tasks and ExecutorService for independent task parallelism—it is possible to achieve substantial reductions in execution time on modern multi-core hardware.

The key finding is that while parallelism offers a powerful tool for performance enhancement, its benefits are not limitless. The performance gains are ultimately constrained by sequential bottlenecks and the overhead of thread management, leading to a point of diminishing returns. The optimal level of parallelism was consistently found to be near the number of physical cores of the test machine's CPU.

5.1.1 Future Work

This project serves as a solid foundation for further exploration in high-performance media processing. Potential avenues for future work include:

- **GPU Acceleration:** The most logical next step is to offload the computational workload to a Graphics Processing Unit (GPU) using libraries like JOCL (Java Bindings for OpenCL) or JCuda. GPUs, with their thousands of specialized cores, are exceptionally well-suited for these types of data-parallel tasks and would likely yield an order-of-magnitude greater performance increase.

- **Real-Time Stream Processing:** The current video processing model reads all frames into memory first. A more advanced implementation could process frames from a live video stream (e.g., from a webcam or network feed) in real-time, using a producer-consumer pattern to decouple frame capture from frame processing.

- **Hybrid Parallelism Model:** For video processing, a hybrid model could be explored. An `ExecutorService` could be used to distribute individual frames to different threads, and within each thread, the `Fork/Join` framework could be used to further parallelize the processing of that single frame by splitting it into sections. This could prove beneficial for ultra-high-resolution video (e.g., 4K or 8K).