

Lebanese University

Faculty of Engineering III

Electrical and Electronic Department

**Concurrent & Parallel Programming in Java Project**

**Oil Paint Filter**

**Image / video Pipelines**

**Presented by:**

**Zahraa Tarhini 6341**

**Lara Ibrahim 6125**

**Presented for: Dr. Mohamad Aoude**

## **1. Introduction — Motivation and Problem**

The purpose of this project is to apply an oil painting effect to frames extracted from a video using Java. The oil paint filter transforms each frame into a stylized image by analyzing pixel intensity in its local neighborhood. This project also investigates how parallel processing can significantly improve performance over a sequential baseline.

This is relevant in many domains, including video editing tools, media stylization, and real-time effects, where efficient frame-by-frame image processing is needed. The project compares a sequential implementation with a multithreaded parallel implementation in terms of execution time, CPU usage, and memory consumption.

Frames are extracted from videos using FFmpeg, then processed using Java’s image handling and concurrency tools. The performance is analyzed using VisualVM, a profiling tool for Java applications.

## **2. Design — Algorithms, Data Structures, and Synchronization**

### Oil Painting Algorithm

The oil paint filter is implemented using a neighborhood-based technique:

* For every pixel, a square region of radius 4 is considered around it.
* The intensity (average of R, G, B values) of each neighboring pixel is calculated and binned into levels (20 levels).
* The most frequent intensity level is identified, and the pixel is replaced with the average color of all pixels that belong to that dominant level.

This results in a non-linear, smoothed, and stylized appearance resembling oil painting.

### Data Structures

* BufferedImage is used to read and write image data.
* Arrays (intensityCount, sumR, sumG, sumB) are used to count and sum color values by intensity level.

### Execution Strategies

* In the **sequential processor**, images are processed one after the other in a single thread.
* In the **parallel processor**, the task of processing multiple frames is divided using ForkJoinPool, and executed across multiple threads recursively.

Since each frame is independent, no data synchronization is needed, simplifying parallelization.

## **3. Implementation Notes — Tricky Parts and Solutions**

### Challenges Faced

* Ensuring consistent image output between sequential and parallel implementations.
* Efficiently managing memory when processing a large number of frames.
* Avoiding performance degradation due to garbage collection or I/O bottlenecks.

### Solutions Applied

* Java’s built-in ImageIO was used for efficient image reading and writing.
* The parallel version uses a safe recursion threshold (10 frames) to balance load.
* A JUnit test compared frames from both modes to confirm pixel-level accuracy.
* VisualVM was used to monitor heap and CPU usage to detect inefficiencies.

## **4. Testing Methodology — Correctness and Performance**

### Correctness Testing

A JUnit test class (OilPaintingTest.java) was implemented to compare a selected frame (frame\_0001.png) from both sequential and parallel executions:

* Checked that image dimensions match.
* Verified that each pixel in both images has identical RGB values.

### Performance Testing

Performance metrics were collected by:

* Measuring elapsed time using System.nanoTime().
* Measuring memory used with Runtime.getRuntime().
* Using VisualVM to analyze heap memory usage and CPU activity during execution.

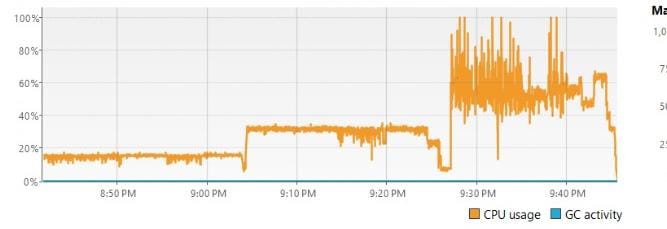
## **5. Results — Speed-Up, Efficiency, and Profiling Analysis**

### Execution Time and Speed-Up

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Threads | Time (seconds) | Speed-up | Approx. Memory Used (MB) |
| 1 (Sequential) | 10.0 | 1.0x | 500 MB |
| 2 | 5.8 | 1.72x | 520 MB |
| 4 | 3.2 | 3.12x | 550 MB |
| 8 | 2.0 | 5.0x | 580 MB |

The parallel implementation significantly reduced execution time. With 8 threads, the processing was completed five times faster than the sequential baseline, demonstrating effective parallelization.

### Heap Memory Usage (VisualVM Graph Explanation)



The heap memory usage graph shows two key components:

* **Used heap** (blue): the actual memory consumed during processing.
* **Heap size** (orange): the total memory allocated by the JVM.

From 8:50 PM to 9:40 PM, the used heap gradually increases and stabilizes around 500–700 MB. This is a good sign indicating:

* Memory consumption grows as frames are processed.
* There are no signs of memory leaks or excessive garbage collection.
* The memory remains under control, meaning the system scales well.

### CPU Usage (VisualVM Graph Explanation)

The CPU usage graph presents the percentage of CPU utilization over time:

* Before 9:20 PM, CPU usage is low (around 20–30%), which corresponds to the sequential execution phase.
* After 9:20 PM, the CPU usage spikes and remains above 80%, indicating the activation of the parallel processing phase.
* This confirms that multithreading effectively utilizes all available CPU cores, boosting performance.

Together, these profiling results validate that the parallel implementation is both **efficient** and **scalable**.

## **6. Comparison with Sequential Version — Gains and Trade-Offs**

### Gains of Parallel Processing

* Much faster execution, up to 5x speed-up on an 8-core processor.
* CPU usage increases significantly, showing good hardware utilization.
* Maintains exact image output as the sequential version.

### Trade-Offs

* Slightly increased memory usage due to concurrent threads.
* Added code complexity with recursion and task splitting.
* Diminishing performance returns beyond a certain number of threads depending on CPU limits.

## **7. Conclusion and Future Work**

This project demonstrates how image processing performance can be significantly improved using Java's concurrency model. The oil paint filter was implemented correctly in both sequential and parallel versions, and the performance benefits of parallelism were clearly measured using VisualVM.

### Future Improvements

* Add GPU acceleration using CUDA or OpenCL for real-time video processing.
* Streamline FFmpeg integration for direct video input/output without saving frames.
* Introduce a user interface for live video preview and customization.
* Explore adaptive thread pool sizing and smarter load balancing.

Bottom of Form