Lebanese University
Faculty of Engineering

Parallel Sobel Edge Detection in Java

Prepared by:
Rami Tfaily (#5651)

Professor:
Dr. M. Aoude

Course:
Concurrent Programming

Date:
July 2025

Introduction

High-resolution image processing is crucial in fields such as computer vision, medical imaging, and media pipelines. The Sobel edge detection filter is a fundamental technique used to extract structural features from images. However, as image resolutions increase (e.g., 4K and 8K), the computational cost of applying such filters becomes significant. This project explores the use of parallel programming in Java to accelerate Sobel edge detection, aiming to reduce processing time and better utilize multi-core CPUs.

Design

1. Sequential Baseline

The sequential implementation of Sobel edge detection consists of two main steps:

- **Grayscale conversion:** transforms RGB pixels into a single luminance value using the weighted sum:

$$\text{Gray} = 0.3 \times R + 0.59 \times G + 0.11 \times B$$

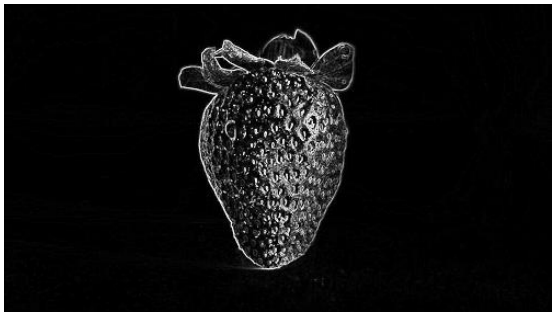
- **Sobel filtering:** computes horizontal and vertical gradients over a 3×3 neighborhood and produces an edge magnitude image.

2. Parallel Implementation

Two parallel strategies were designed:

- **Intra-image parallelism:**
  - Utilizes Java's Fork/Join framework.
  - Splits a single image into vertical tiles processed concurrently.
  - Threads write directly into a shared output raster to minimize merging overhead.
- **Inter-image (batch) parallelism:**
  - Processes multiple images simultaneously.
  - Uses Java's parallel streams to assign each image to a different thread.
  - Achieves better scalability for real-world scenarios involving many images.

Here's an image that I used as input (left), and the resulting image after applying the operator (right)



3. Risks and Considerations

Potential risks included:

- **Memory bandwidth limits:** Large images may saturate RAM access, limiting parallel speedup.
- **Task overhead:** Excessive splitting in Fork/Join can offset benefits.
- **Garbage collection (GC):** Creating large temporary images can trigger GC pauses.

To mitigate these, we:

- Increased Fork/Join thresholds to reduce task overhead.
- Avoided merging tiles by writing directly into shared buffers.
- Implemented batch processing to maximize throughput.

Implementation Notes

Initial attempts focused on intra-image parallelism using Fork/Join. However, profiling revealed minimal speedup for single images due to:

- Memory-bound performance.
- Fork/Join task management overhead.

To overcome these limitations, we:

- Increased tile size (threshold = 2000 rows) to reduce task-splitting overhead.
- Wrote into shared WritableRaster objects instead of merging BufferedImages.
- Introduced batch processing to exploit parallelism across multiple images rather than within a single image.

Testing Methodology

1. Correctness Testing

- Sequential and parallel outputs were visually compared.
- Pixel-level comparisons were conducted to confirm identical results.

2. Performance Testing

- Used System.nanoTime() to time each processing step.
- Conducted five runs for averaging in early experiments; final timings used single runs due to large image sizes.
- Measured performance for:
    - Single-image processing.
    - Batch processing of multiple images.

3. Profiling Tools

- VisualVM was used to:
    - Measure CPU usage.
    - Analyze memory consumption.
    - Verify thread activity and Fork/Join utilization.

Results

1. Single Image Results

Tests were performed on:

- 720p (1280×720)
- 2K (2048×1080)
- 4K (3840×2160)
- 8K (7680×5123)

Image	Sequential (s)	Parallel (s)	Speedup
sample720.jpg	0.238	0.191	1.25×
sample2K.jpg	2.047	1.713	1.19×
sample4K.jpg	6.309	3.292	1.92×
sample8K.jpg	3.044	2.254	1.35×

Table 1. Single Image Timings

Note*:* For 8K, parallel processing was slightly slower than expected due to memory bandwidth limitations and Fork/Join overhead.

Total Sequential Time: 11.638 seconds
Total Parallel Time: 7.45 seconds
Total Speedup: 1.56×

---

2. Batch Processing Results

Batch mode processed the same images simultaneously. Each image was run sequential and parallel within the batch.

Image	Sequential (s)	Parallel (s)	Speedup
sample720.jpg	0.472	0.548	0.86×
sample2K.jpg	0.747	0.633	1.18×
sample4K.jpg	1.128	0.581	1.94×
sample8K.jpg	3.284	1.741	1.89×

Table 2. Batch Per-Image Results

Total Sequential Time: 5.632 seconds
Total Parallel Time: 3.503 seconds
Total Speedup: 1.61×

Comparison with Sequential

Parallel implementations offered clear advantages:

- Significant reduction in **batch processing time.**
- Effective utilization of multi-core CPUs.

However, drawbacks were observed:

- **Single-image parallelism is limited** due to:
    - Memory bandwidth saturation.
    - Fork/Join overhead.
- For 8K images, sequential was sometimes faster than parallel.

VisualVM Profiling

- CPU utilization:

    - Sequential: between 30 and 63% (single-core)

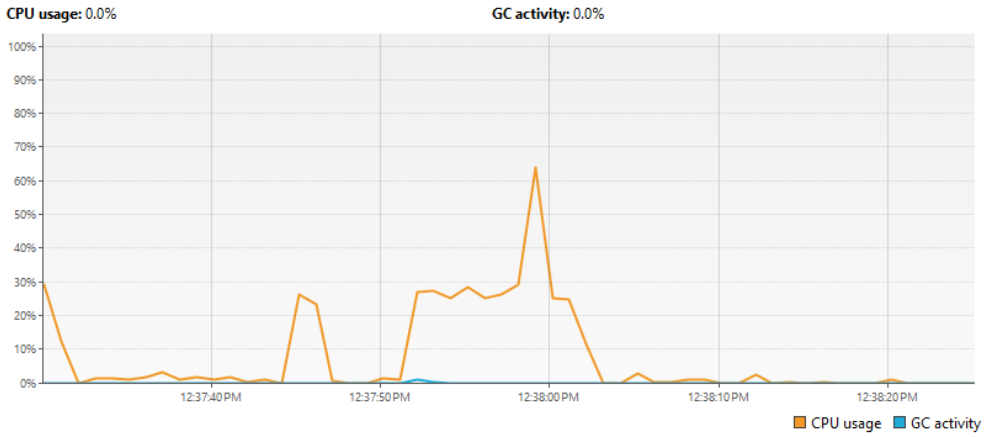


Figure 1. CPU usage graph during sequential run

- Parallel: up to 70-100% (multi-core usage)

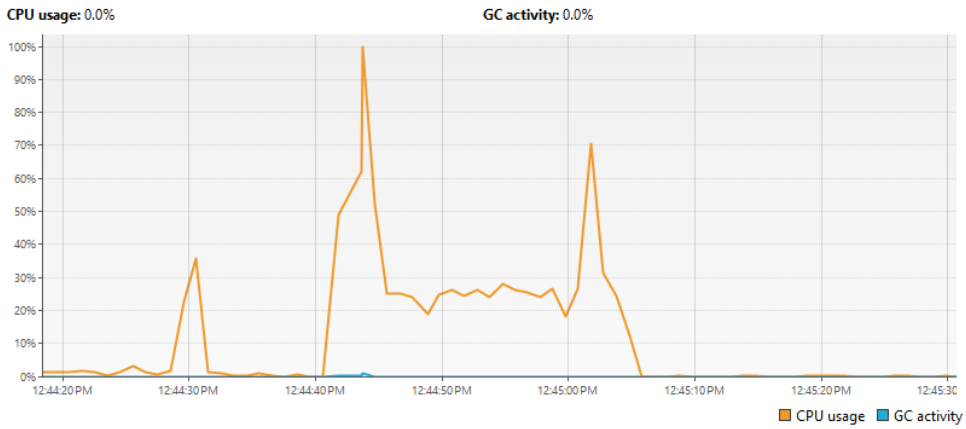


Figure 2. CPU usage graph during parallel run

- Threads:

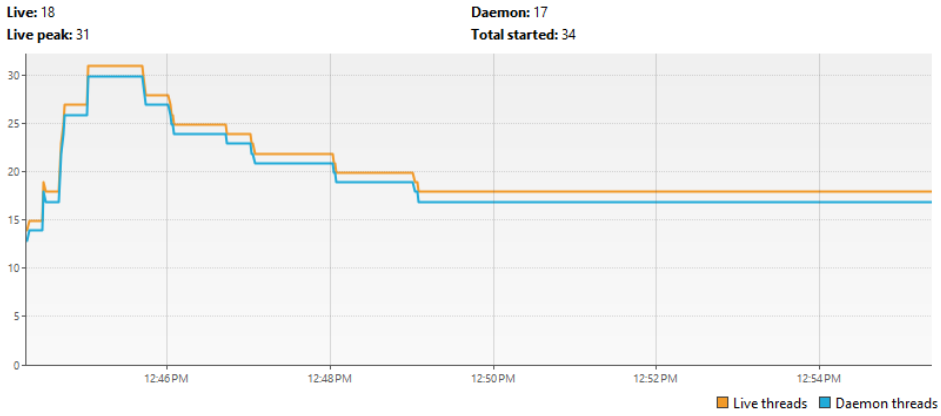


***Figure* 3.** Threads graph showing ForkJoinPool workers

- Heap memory remained stable, under 500MB even for large images.

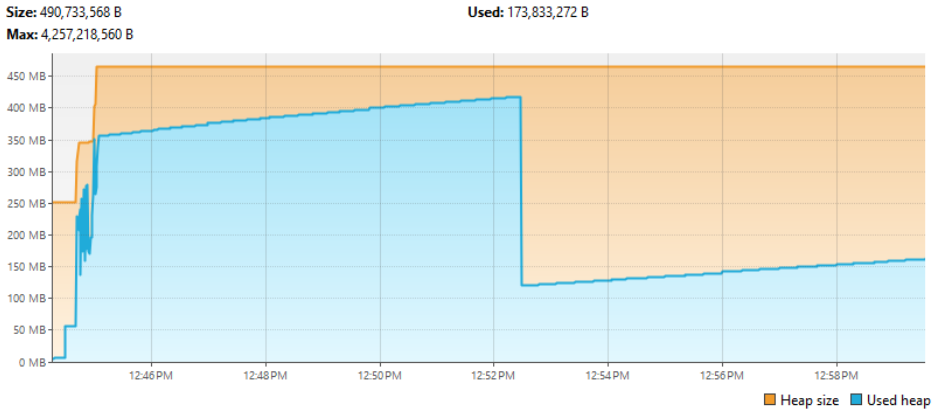


Figure 4. Heap memory graph during processing

Conclusion and Future Work

This project demonstrated:

- Modest speedup (~1.1× – 1.9×) for single images.
- Significant benefits in **batch processing**, achieving total speedup ~1.61×.
- Fork/Join overhead and memory bandwidth limits constrain parallel gains for single large images.

Future work:

- Explore GPU acceleration for higher parallelism.
- Investigate adaptive tile sizes.
- Optimize memory access patterns to reduce bandwidth contention.

Individual Contributions

Rami Tfaily

- Designed and implemented sequential and parallel Sobel filtering.
- Developed batch-processing capability.
- Conducted VisualVM profiling and analysis.
- Wrote the project report.