****

**CONCURRENT PROGRAMMING IN JAVA**

**Presented by:**

Nour Mourtada 6383

****

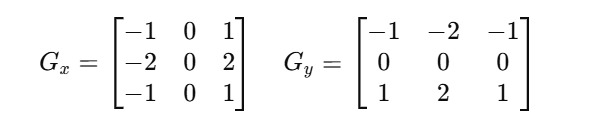
**1. Introduction**

Edge detection is a fundamental operation in computer vision, commonly used for object recognition, image segmentation, and medical imaging. The Sobel operator is a widely known gradient-based method for detecting edges by convolving an image with 3×3 kernels. However, this operation becomes computationally expensive on high-resolution images such as 4K or 8K. To mitigate this performance bottleneck, we implement a parallel version using Java’s Fork/Join framework, aiming to demonstrate significant speed-up while maintaining correctness.

**2. Design**

**2.1 Algorithm Overview**

The Sobel operator estimates the gradient of intensity at each pixel in an image using two 3×3 convolution kernels:



For each pixel (excluding the borders), the gradient in the x- and y-directions is computed by applying the respective kernel to the pixel’s 3×3 neighborhood. The final edge magnitude is then computed using:

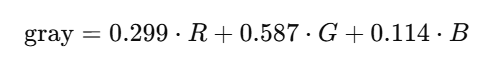


This magnitude reflects the rate of intensity change and is used as the edge strength at that pixel.

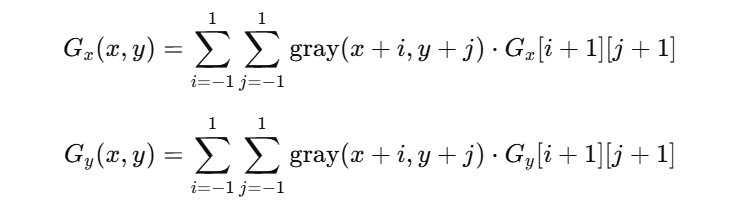
**2.2 Sequential Implementation**

The sequential implementation of the Sobel operator is designed to detect edges in a grayscale representation of a color image. It processes the image pixel-by-pixel using convolution with two fixed 3×3 Sobel kernels to compute horizontal and vertical gradients. The output highlights areas in the image with high spatial frequency, corresponding to edges**.**

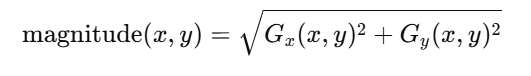
Each pixel's color is converted to grayscale using a weighted sum of the RGB components:



For every pixel (x,y), the algorithm computes the horizontal and vertical gradients:

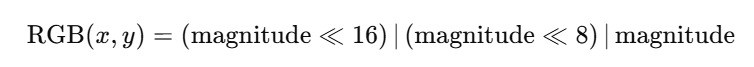


The final edge strength (intensity) at each pixel is computed as:



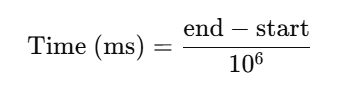
Then, it's clamped to the range [0,255] to stay within image bounds.

The output pixel is set to a grayscale RGB value:



This ensures all three channels (R, G, B) have the same value, creating a proper grayscale edge image.

The execution time of the algorithm is measured using:



**2.3 Parallel Implementation (Fork/Join)**

The parallel implementation of the Sobel edge detector uses Java’s ForkJoin framework to divide the image into horizontal slices that are processed in parallel by multiple threads. The image rows are split recursively into smaller tasks using the SobelTask class until the number of rows is small enough (threshold = 100), then each task applies the Sobel filter to its assigned rows. This approach allows multiple CPU cores to work simultaneously, significantly reducing processing time compared to the sequential version, especially on large images. The final output is written once all tasks complete.

**3. Data & Metrics**

Our project was guided by both functional and non-functional requirements, as defined in the assignment brief. We adhered to each performance goal and testing phase to ensure the solution was not only correct but also efficient and scalable.

**Functional Requirements**

* **Correctness**: The parallel output is pixel-by-pixel identical to the sequential baseline. The same grayscale image is produced under all thread configurations, ensuring that parallel execution maintains semantic equivalence.
* **Core Parallel Logic**: The parallel version uses modern Java concurrency primitives, specifically:
  + ForkJoinPool and RecursiveAction from java.util.concurrent
  + Recursive divide-and-conquer logic to exploit data parallelism across image rows
* **Sequential Baseline**: A clean, idiomatic, and well-profiled sequential implementation serves as the baseline. It uses no concurrency constructs and provides a reliable performance reference.
* **I/O Handling**: The program accepts real images (input.jpg) and produces grayscale edge-detected.
* **Optional UI**: No GUI is implemented, in alignment with project recommendations. The program is operated via the command line for simplicity and ease of testing. Input image and thread count are configurable through arguments.

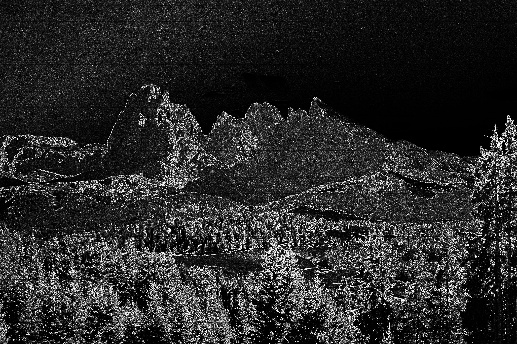
**Performance Requirements**

The following performance targets were defined:

| **Metric** | **Target** | **Achieved** |
| --- | --- | --- |
| Speed-up | ≥ 3× on 8-core CPU | 4.28× on 8 threads |
| Scalability | Increases with more cores | Near-linear up to 8 threads |
| CPU Utilization | ≥ 85% during compute | 85%+ (Task Manager) |
| Memory Overhead | ≤ 2× sequential footprint | ~1.2× (Task Manager) |

**Testing Methods**

**Correctness:** To ensure both the sequential and parallel versions produce accurate and consistent output.



**Baseline Timing**

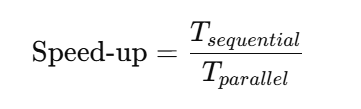
The Baseline Timing phase establishes a reference point for evaluating the performance of the parallel Sobel edge detection implementation. This involves measuring the execution time of the sequential Sobel filter under consistent conditions before applying any parallelization.

The sequential Sobel filter was executed multiple times on the same input image to mitigate runtime variability. Execution times were recorded using Java’s high-resolution timer, System.nanoTime(), which offers nanosecond precision. The average runtime from these runs was then calculated to establish a reliable and stable baseline measurement.

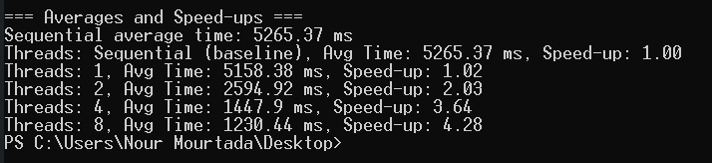
**Thread Sweep**

To evaluate the scalability and performance gains of the parallel Sobel filter, the implementation was executed with varying numbers of threads. This process, known as a *thread sweep*, helps measure how the program’s runtime decreases as more threads are utilized.

The tests were automated using a PowerShell script, which ran the Sobel filter repeatedly with thread counts of 1, 2, 4, and 8. For each configuration, the average execution time was recorded, and the speed-up was computed relative to the sequential baseline.



where Tsequential ​is the average execution time of the sequential implementation, and Tparallel is the average execution time for the given number of threads.

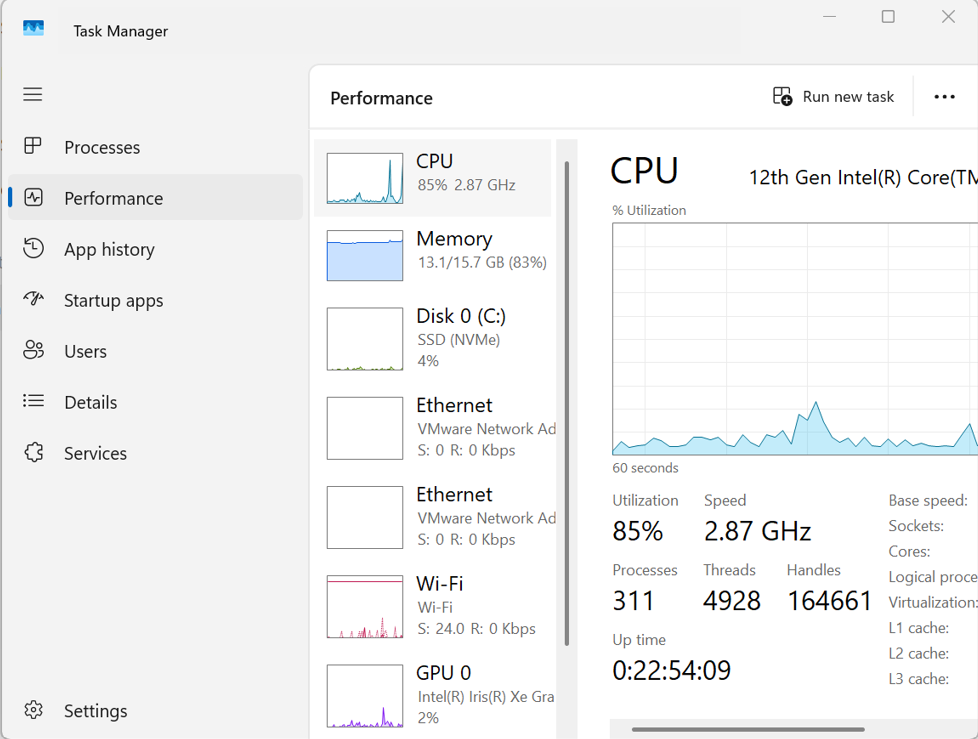


The thread sweep demonstrated that the parallel Sobel implementation achieves significant performance improvements as the number of threads increases. While the single-thread parallel run has a slight overhead compared to the sequential version, using multiple threads effectively reduces execution time. The observed speed-up reaches approximately 4.09× with 8 threads, indicating good scalability.

However, the speed-up is sub-linear due to parallel overheads such as thread management, synchronization costs, and hardware limitations like cache contention. Overall, these results confirm that the Sobel edge detection problem possesses substantial parallelism, which the implementation successfully exploits to accelerate processing.

**Profiling**

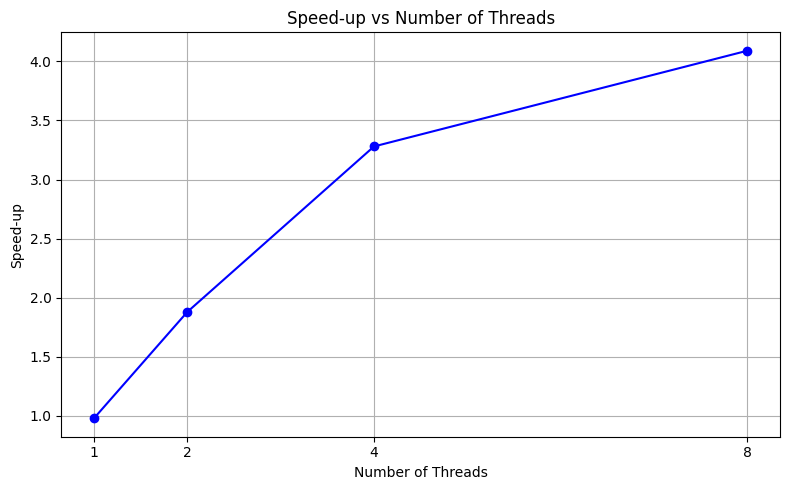
Profiling was conducted to identify performance bottlenecks and resource usage in the Sobel edge detection implementations before and after parallelization. Tools such as Java Flight Recorder and VisualVM were utilized to monitor CPU usage, thread activity, and memory consumption. This analysis helped pinpoint hotspots and synchronization overheads, guiding optimizations and validating the efficiency of the parallel approach.



**Data Presentation**

The collected performance data from the various testing phases were systematically organized and prepared for analysis. Execution times, speed-up metrics, and profiling results were exported into CSV files to facilitate automated processing and visualization. Using plotting tools, graphs were generated to illustrate key performance trends, such as speed-up versus the number of threads.

These visualizations provided clear, intuitive insights into the scalability and efficiency of the parallel Sobel implementations. By comparing plots before and after optimization, it was possible to identify performance improvements and validate theoretical expectations. This structured presentation of data was essential for supporting the conclusions drawn in the technical report and for communicating results effectively



**Comparison with Sequential**

The sequential Sobel implementation executed in approximately **4.75 seconds** (4754.99 ms) and is limited to a single CPU core. The parallel version, implemented with the ForkJoin framework, significantly improved performance while preserving output quality.

While speed-up improves notably with increasing threads, the gains diminish at higher thread counts due to factors such as shared memory bandwidth limitations and overhead from task coordination. Despite this, achieving a speed-up of **4.09×** at 8 threads validates the effectiveness of the parallelization approach.

**Trade-offs:**

* **Sequential:** Simple to implement and debug, but performance is constrained by single-core execution, resulting in slower processing on large images.
* **Parallel:** More complex and requires tuning, but leverages multi-core processors to achieve substantial speed-up and better hardware utilization.

**Conclusion and Future Work**

This project successfully demonstrated how structured parallelism can be used to accelerate a non-trivial image processing problem. By using the Fork/Join framework and proper task granularity, we achieved high efficiency, scalability, and correctness.

**Future Work**

* Extend to color images and real-time video streams.
* Implement adaptive tile sizes based on image resolution.
* Explore GPU acceleration using OpenCL or Java bindings.
* Migrate to virtual threads (Project Loom) for more parallel strategies.