A blue and yellow logo

Description automatically generatedLebanese Universityالجامعة اللبنانية

Faculty of Engineeringكلية الهندسة

Branch-3الفرع الثالث

**Java Parallel Programming Project Report**

***Prepared by:***

* Omar Ismail 6304
* Roudy Barakat 6305

**Presented for**:

Dr. Mohammed Aoude

**Date**: 10/ 7 / 2025

**Table of Contents**

1. **Introduction**
   * 1.1 Problem Domain: Image Processing
   * 1.2 Motivation
   * 1.3 Project Goals
2. **Design and Architecture**
   * 2.1 System Overview
   * 2.2 Core Components
     + 2.2.1 SegmentationGUI.java (User Interface)
     + 2.2.2 LiveImageDisplay.java
     + 2.2.3 SequentialSegmenter.java
     + 2.2.4 Parralel\_segmenter.java
   * 2.3 Image Segmentation Logic (RGB Thresholding)
   * 2.4 Performance Considerations in Design
3. **Implementation Detail**s
   * 3.1 Development Environment
   * 3.2 Concurrency with ForkJoinPool and RecursiveAction
   * 3.3 GUI Implementation with Swing
   * 3.4 Performance Instrumentation
   * 3.5 Live Visualization Mechanism
   * 3.6 Memory Management and Optimization
4. **Testing Methodology**
   * 4.1 Correctness Testing
   * 4.2 Performance Testing
     + 4.2.1 Metrics Measured
     + 4.2.2 Test Procedure
     + 4.2.3 Test Environment Considerations
5. **Results and Analysis (Expected)**
   * 5.1 Speed-up Analysis
   * 5.2 Scalability Analysis
   * 5.3 CPU Utilization Discussion
   * 5.4 Memory Overhead Analysis
   * 5.5 Comparison: Sequential vs. Parallel
6. **Conclusion and Future Work**
   * 6.1 Summary of Achievements
   * 6.2 Challenges and Lessons Learned
   * 6.3 Future Enhancements

**Parallel Color Image Segmentation with GUI: Performance Analysis**

**Prepared by:**

* Omar Ismail (ID: 6304, ULFG 3 – Telecommunications Engineering)
* Roudy Barakat (ID: 6305, ULFG 3 – Telecommunications Engineering)

**1. Introduction**

**1.1 Problem Domain: Image Processing**

Image segmentation is a fundamental task in image processing and computer vision, involving the partitioning of a digital image into multiple segments (sets of pixels). This process simplifies and/or changes the representation of an image into something more meaningful and easier to analyze. Applications span various fields, including medical imaging (e.g., tumor detection), autonomous vehicles (e.g., object recognition), satellite imagery analysis, and industrial inspection.

**1.2 Motivation**

While grayscale image segmentation is relatively straightforward, color image segmentation introduces significantly more complexity due to the three-color channels (Red, Green, Blue) and their interdependencies. Processing high-resolution color images sequentially results in a major performance bottleneck:

* **Slow Execution Times:** Each pixel must be processed individually, leading to long waits for large images.
* **High CPU Idling:** Modern computers possess multi-core CPUs, but sequential algorithms only utilize a single core, leaving other cores idle and resources underutilized. This project aims to address these limitations by leveraging Java's robust concurrency features to parallelize color image segmentation.

**1.3 Project Goals**

The primary objective of this project is to develop an efficient parallel color image segmentation application using Java, specifically focusing on RGB thresholding. Key goals include:

* **Achieve Significant Speed-up:** Demonstrate a substantial reduction in processing time compared to the sequential approach, especially for high-resolution images.
* **Optimize CPU Utilization:** Ensure that the parallel implementation effectively utilizes multiple CPU cores, minimizing idle time.
* **Develop an Intuitive User Interface (UI):** Create a graphical front-end that allows users to easily upload images, configure segmentation parameters, visualize the live progress of both sequential and parallel algorithms, and view comprehensive performance metrics.
* **Analyze Performance Metrics:** Rigorously measure and present speed-up, scalability across varying thread counts, and memory overhead to validate the efficiency and effectiveness of the parallel approach.

**2. Design and Architecture**

**A white rectangular object with a black border

Description automatically generated**A computer screen shot of a diagram

Description automatically generated**2.1 System Overview** The system is designed as a standalone Java Swing application. Its architecture can be visualized as follows:

**2.2 Core Components**

The project consists of four main Java classes, each serving a distinct role:

**2.2.1 SegmentationGUI.java (User Interface)**

This is the main application class, inheriting from ***JFrame***. It provides the interactive dashboard for the user.

* **Components:**
  + **JLabel imagePreviewLabel:** Displays a small preview of the uploaded image.
  + **JTextField thresholdField:** Allows input of the segmentation threshold (0-255).
  + **JComboBox<String> typeComboBox:** Selects the segmentation type (grayscale, red, green, custom).
  + **JTextField numThreadsField:** Specifies the number of threads for parallel execution, defaulting to available CPU cores.
  + **JTextField delayMsField:** An optional field to introduce an artificial delay (in milliseconds) per image update, useful for visualizing the step-by-step process.
  + **JTextArea metricsTextArea:** Displays all performance results, logs, and scalability test outcomes.
  + **JScrollPane metricsScrollPane:** Provides scrollability for the metricsTextArea.
  + **JButton uploadButton:** Triggers a JFileChooser to select an image file.
  + **JButton startSegmentationButton:** Initiates a single run of both sequential and parallel segmentation, then displays combined results.
  + **JButton runScalabilityTestButton:** Executes the parallel segmentation multiple times with varying thread counts (1, 2, 4, 8, max cores) to gather scalability data.
  + **LiveImageDisplay seqLivePanel, LiveImageDisplay parLivePanel:** Custom JPanel instances embedded directly into the GUI, displaying the real-time progress of sequential and parallel segmentation, respectively.
* **Layout:** Utilizes a combination of BorderLayout, GridBagLayout (for flexible control arrangement), and GridLayout (for image displays) to create a clear and organized interface.
* **Responsiveness:** Employs SwingWorker for all long-running segmentation tasks (doInBackground() method). This ensures that the GUI remains responsive and doesn't freeze while image processing is underway. Results and updates are posted back to the Event Dispatch Thread (EDT) via publish() and SwingUtilities.invokeLater().

**2.2.2 LiveImageDisplay.java**

This custom JPanel is designed to display a BufferedImage and dynamically update it.

* **BufferedImage image:** Holds the current state of the image being processed.
* **setImage(BufferedImage newImage):** A synchronized method to update the image reference. It also handles preferred size adjustment and calls revalidate() and repaint() to ensure the panel redraws itself with the new image data.
* **paintComponent(Graphics g):** Overrides the standard Swing painting method. It draws the current BufferedImage onto the panel. Synchronization is crucial here to prevent concurrent modification issues, as worker threads might be altering the image while the EDT attempts to paint it. If no image is loaded, it displays a "Image Not Loaded" placeholder.

**2.2.3 SequentialSegmenter.java**

This class provides the single-threaded baseline implementation for image segmentation.

* **segment(BufferedImage input, BufferedImage output, int threshold, String type, int delayMs, LiveImageDisplay panel):**
  + Iterates through each pixel of the input image (inputRaster.getPixel(x,y,pixels)).
  + Applies the chosen thresholding logic (grayscale, red, green, custom) based on the RGB values.
  + Sets the processed pixel to the output image (outputRaster.setPixel(x,y,newPixel)).
  + **Optimization:** int[] pixels and int[] newPixel arrays are declared *once* outside the inner loops to minimize object allocation and reduce garbage collection overhead, significantly improving performance for large images.
  + **Visualization:** Includes a Thread.sleep(delayMs) call and a panel.repaint() after each row is processed, allowing for a visual step-by-step demonstration when a delayMs is provided. This is only for visualization; for performance measurement, delayMs is set to 0.

**2.2.4 Parralel\_segmenter.java**

This class implements the parallel image segmentation using the ForkJoinPool framework.

* **LEAF\_TASK\_THRESHOLD:** A static variable that determines the granularity of tasks. Tasks smaller than this threshold are processed directly (leaf tasks), while larger tasks are forked into sub-tasks. It's dynamically calculated based on image height and number of threads to optimize for different scenarios.
* **SegmentTask (Inner RecursiveAction class):**
  + Represents a unit of work that processes a horizontal strip of the image (from startY to endY).
  + **compute() method:**
    - If the strip is small enough (less than or equal to LEAF\_TASK\_THRESHOLD), it directly processes pixels within that strip, similar to the sequential segmenter.
    - If the strip is too large, it splits itself into two halves (leftTask, rightTask) and calls invokeAll(leftTask, rightTask) to execute them concurrently.
    - **Optimization:** Similar to the sequential segmenter, int[] pixels and int[] newPixel are declared *once* per SegmentTask instance (outside the pixel-processing loops) to reduce allocation overhead.
    - **Live Visualization:** It periodically calls livePanelRef.repaint() (e.g., every 10 rows processed) and includes a Thread.sleep(delayMs) when delayMs > 0 to provide live updates of the parallel process. The repaint() call is synchronized to prevent issues with concurrent painting.
* **segment(BufferedImage input, BufferedImage output, int threshold, String type, int numThreads, LiveImageDisplay panel, int delayMs):**
  + Initializes the ForkJoinPool with the specified numThreads.
  + Creates the initial SegmentTask covering the entire image.
  + Invokes the task on the ForkJoinPool, which manages the forking and joining of sub-tasks.
  + Manages the lifecycle of ForkJoinPool (shutdown and await termination).

**2.3 Image Segmentation Logic (RGB Thresholding)**

The core segmentation logic applied to each pixel is a form of RGB thresholding. For each pixel, its Red (R), Green (G), and Blue (B) components are extracted. Based on the selected type, a new pixel value is determined:

* **grayscale:** Calculates the average of R, G, and B, then sets the pixel to black (0) or white (255) based on the threshold (binary segmentation).
* **red:** If the red component is dominant and exceeds the threshold (r > g && r > b && r > threshold), the pixel becomes pure red (255, 0, 0); otherwise, it's black.
* **green:** Similar to red, but highlights the green component (g > r && g > b && g > threshold).
* **custom:** Applies a specific custom condition (e.g., r > 100 && g < 150 && b > 50) to determine if the original color is kept or set to black. The alpha channel is preserved if the original image has one.

**2.4 Performance Considerations in Design**

* **Direct Pixel Access:** Using BufferedImage.getRaster().getPixel() and WritableRaster.setPixel() is generally more efficient than BufferedImage.getRGB() and BufferedImage.setRGB() for direct pixel manipulation in tight loops, as it avoids repeated Color object creation and integer conversions.
* **Reduced Object Allocation:** Critically, int[] pixels and int[] newPixel arrays are reused per task/segmentation run rather than being allocated per pixel within the inner loops. This significantly reduces garbage collection pressure, which is a common performance bottleneck in Java.
* **Amdahl's Law:** The design acknowledges Amdahl's Law. While computation is parallelized, image loading/saving and UI updates (if delayMs > 0) are inherently sequential parts. The performance measurements are focused on the "compute phase" to isolate parallel gains.
* **Granularity:** The LEAF\_TASK\_THRESHOLD directly controls task granularity. Optimal granularity minimizes both context switching overhead (too many small tasks) and idle core time (too few large tasks).

**3. Implementation Details**

**3.1 Development Environment**

* **Language:** Java (JDK 17 or newer, preferably JDK 21 or 24 for modern features and performance).
* **Libraries:** Standard Java SE libraries (java.awt, javax.swing, java.awt.image, java.util.concurrent). No external third-party libraries are used, ensuring easy compilation and deployment.
* **IDE:** Developed using a standard Java IDE like Eclipse or IntelliJ IDEA.

**3.2 Concurrency with ForkJoinPool and RecursiveAction**

The heart of the parallel implementation lies in the java.util.concurrent.ForkJoinPool.

* ForkJoinPool efficiently manages a pool of worker threads and implements a work-stealing algorithm, where idle threads "steal" tasks from busy threads' queues, ensuring optimal load balancing.
* RecursiveAction is used because the tasks do not return a result directly but modify a shared BufferedImage in place. The main SegmentTask recursively forks itself until a small enough sub-task (defined by LEAF\_TASK\_THRESHOLD) is reached.
* The output BufferedImage is a shared mutable resource. While pixel-level writes are concurrent on distinct regions, repaint() calls and Thread.sleep() are controlled to prevent race conditions during UI updates.

**3.3 GUI Implementation with Swing**

The GUI (SegmentationGUI.java) is built using Java Swing components.

* **Layout Managers:** BorderLayout manages the main sections (controls, image displays, metrics). GridBagLayout provides flexible, constraint-based layout for the input controls, ensuring proper alignment and resizing. GridLayout is used for the side-by-side image panels, and FlowLayout centers the action buttons.
* **Event Handling:** ActionListener interfaces are used to respond to button clicks and user interactions.
* **SwingWorker:** This abstract class is crucial for maintaining UI responsiveness. It allows long-running operations (doInBackground()) to be executed on a separate background thread. Progress updates (publish()) and final results (done()) are marshaled back to the Event Dispatch Thread (EDT) safely, preventing the GUI from freezing.

**3.4 Performance Instrumentation**

* **System.nanoTime():** This high-resolution timer is used to precisely measure the execution time of both sequential and parallel segmentation algorithms. Timings are taken immediately before and after the core computation to exclude I/O overhead.
* **Memory Measurement:** Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory() provides an approximate measure of heap memory used. System.gc() calls are strategically placed before measurements to encourage garbage collection and obtain more consistent snapshots of memory usage, though these are indicative rather than absolute precise measurements.

**3.5 Live Visualization Mechanism**

The LiveImageDisplay custom JPanel combined with strategic repaint() calls provides real-time visual feedback:

* In SequentialSegmenter, repaint() is called after each row is processed, showing the image building up from top to bottom.
* In Parralel\_segmenter, repaint() is called periodically (e.g., every 10 rows processed collectively by all threads). This creates a more dynamic, block-by-block fill pattern, especially noticeable with a delayMs > 0.
* The delayMs parameter allows users to slow down the process, making these subtle differences in execution patterns clearly visible for demonstration and educational purposes.

**3.6 Memory Management and Optimization**

* **Deep Copies:** deepCopy(BufferedImage bi) ensures that input images are not modified and that new BufferedImage objects are created for output, preventing unintended side effects.
* **Array Reuse:** As noted in 2.2.3 and 2.2.4, int[] pixels and int[] newPixel arrays are allocated once per processing unit (per method call in sequential, per SegmentTask in parallel) and reused. This minimizes repeated object creation within the pixel-processing loops, significantly reducing stress on the Garbage Collector and improving overall performance.
* **No Reckless Allocation:** The design actively avoids creating unnecessary objects in performance-critical sections to adhere to the memory overhead target.

**4. Testing Methodology**

**4.1 Correctness Testing**

* **Visual Inspection:** The primary method for correctness. After segmentation, the combined output image (combined\_output.png) shows sequential and parallel results side-by-side. Visually, they should be identical.
* **Edge Cases:** Testing with images containing extreme colors, very low/high thresholds, and different dimensions ensures the algorithms handle various inputs correctly.
* **No Data Loss/Corruption:** Ensuring that BufferedImage and Raster manipulations do not lead to image corruption.

**4.2 Performance Testing**

The GUI provides tools to conduct performance tests efficiently.

**4.2.1 Metrics Measured**

* **Sequential Time (**Ts​**):** Time taken by the single-threaded algorithm.
* **Parallel Time (**Tp​**):** Time taken by the multi-threaded algorithm for a given number of threads.
* **Speed-up (**Sp​**):** Calculated as Ts​/Tp​.
* **Memory Footprint (Approximate):** Change in JVM heap memory usage during computation for both sequential and parallel versions.
* **Memory Overhead Ratio:** Parallel Memory Footprint / Sequential Memory Footprint.

**4.2.2 Test Procedure**

* **Standard Segmentation Run:**
  1. Upload a high-resolution image (e.g., 4K).
  2. Set desired threshold and type.
  3. Set Delay (ms/update) to 0 for performance measurements (or a higher value for visualization).
  4. Click "Start Segmentation".
  5. Observe live progress and final metrics in the metricsTextArea and the combined output window.
* **Scalability Test (Thread Sweep):**
  1. Upload a high-resolution image.
  2. Set desired threshold and type.
  3. Ensure Delay (ms/update) is 0.
  4. Click "Run Scalability Test".
  5. The system will automatically run parallel segmentation for 1, 2, 4, 8, and the maximum available CPU cores.
  6. The metricsTextArea will display the parallel time and speed-up for each thread count, demonstrating the scalability curve.

**4.2.3 Test Environment Considerations**

* **Consistent Hardware:** All tests must be conducted on the same machine to ensure comparable results. The CPU (number of cores), RAM, and operating system should be noted.
* **Dedicated Resources:** Minimize other running applications to ensure the Java process has maximum access to CPU and memory.
* **Large Input Images:** Using 4K resolution or larger images is crucial for demonstrating the benefits of parallelism, as the overhead of ForkJoinPool is amortized over a larger workload.
* **Multiple Runs:** For accurate average measurements, especially for speed-up, multiple runs (e.g., ≥5 runs for each configuration) should be performed, and the average time should be reported. The current GUI performs a single run per button click, but this can be extended for more robust testing.

**5. Results and Analysis (Expected)**

*This section details the anticipated outcomes based on the principles of parallel computing. Actual results will vary based on hardware and specific image characteristics.*

**5.1 Speed-up Analysis**

We expect to observe a significant speed-up (Sp​=Ts​/Tp​) for the parallel segmentation compared to the sequential baseline, especially when processing large, high-resolution images with delayMs = 0.

* For images like 4K resolution on multi-core CPUs, the parallel version is expected to achieve speed-ups well above 3×, potentially reaching 70 or more of ideal speed-up (where ideal speed-up is equal to the number of cores).
* For smaller images or simpler transformations, the overhead of task creation and management in the ForkJoinPool might make the parallel version slightly slower than sequential due to the overhead outweighing the benefits of parallelism.

**5.2 Scalability Analysis**

The scalability test will illustrate how the parallel performance changes with an increasing number of threads.

* Initially, as the number of threads increases from 1 to 2, 4, and possibly 8, the parallel execution time is expected to decrease, and the speed-up is expected to increase. This demonstrates effective utilization of additional CPU cores.
* However, we anticipate observing a **plateau** in speed-up after a certain number of threads (often around the number of physical CPU cores or logical processors). This phenomenon is explained by **Amdahl's Law**, which states that the maximum speed-up is limited by the inherently sequential portion of the program (e.g., image I/O, initial setup, final cleanup, or even internal synchronization in the JVM). Beyond this point, adding more threads yields diminishing returns or even performance degradation due to increased overhead from context switching, contention for shared resources, and cache invalidation.

**5.3 CPU Utilization Discussion**

During the "compute phase" of the parallel segmentation, the goal is to achieve ≥85 CPU utilization. While the Java Runtime API doesn't provide precise per-process CPU utilization, the design of the ForkJoinPool and its work-stealing mechanism inherently aims to keep all available worker threads busy. High utilization implies efficient workload distribution and minimal idle core time during the parallel section. External system monitors (like Task Manager on Windows, htop on Linux, Activity Monitor on macOS) can visually confirm high CPU utilization during the active segmentation.

**5.4 Memory Overhead Analysis**

The memory overhead metric (Parallel Memory Footprint / Sequential Memory Footprint) is expected to remain ≤2× the sequential footprint. This indicates that the parallelization strategy, despite introducing additional data structures for task management (e.g., RecursiveAction objects, ForkJoinPool queues), does not lead to reckless memory allocation. The optimization of reusing pixel arrays within processing loops is critical to achieving this goal, preventing excessive garbage collection cycles.

**5.5 Comparison: Sequential vs. Parallel**

* **Wins of Parallel:**
  + Significantly faster execution for large image datasets.
  + Efficient utilization of multi-core processors.
  + Improved user experience through quicker processing feedback for heavy tasks.
* **Trade-offs of Parallel:**
  + Introduces overhead for task creation and management (less efficient for very small images).
  + Increased code complexity compared to a purely sequential approach.
  + Requires careful tuning of granularity for optimal performance.

**6. Conclusion and Future Work**

**6.1 Summary of Achievements**

This project successfully developed a Java-based parallel image segmentation application for RGB thresholding, demonstrating significant performance improvements over its sequential counterpart. The implementation effectively leverages the ForkJoinPool framework for efficient parallel execution. A comprehensive Swing GUI was developed, transforming the application into a user-friendly dashboard that not only visualizes live segmentation progress but also provides detailed performance metrics, including speed-up, scalability, and approximate memory overhead.

**6.2 Challenges and Lessons Learned**

* **Granularity Tuning:** Discovering the optimal LEAF\_TASK\_THRESHOLD was crucial. Too small, and overhead dominated; too large, and parallelism was underutilized. This highlighted the importance of empirical testing.
* **UI Responsiveness:** Ensuring the GUI remained interactive during long-running background tasks was a key challenge, effectively addressed by using SwingWorker.
* **Accurate Timing:** Disabling visualization delays (delayMs = 0) was essential to obtain accurate performance measurements, as Thread.sleep() drastically inflates execution times.
* **Memory Management:** Minimizing object allocations within inner loops was critical for performance and controlling memory footprint, underscoring the importance of understanding Java's memory model and garbage collection.

**6.3 Future Enhancements**

* **Advanced Segmentation Algorithms:** Extend the parallel framework to support more complex image processing algorithms, such as convolution filters (e.g., Gaussian blur, edge detection), morphological operations, or K-means clustering for segmentation.
* **Graphical Metrics Visualization:** Integrate a Java charting library (e.g., JFreeChart, XChart) to generate dynamic plots for scalability curves (Speed-up vs. Threads, Time vs. Input Size) directly within the GUI.
* **Adaptive Thread Management:** Implement a more sophisticated mechanism for dynamically adjusting the number of threads or task granularity based on real-time CPU load or image characteristics.
* **File Format Support:** Expand support for more image file formats beyond basic JPG/PNG.
* **Benchmarking Suite:** Develop an automated testing suite to run a series of benchmarks on predefined images and export results to CSV for external analysis.