A blue and yellow logo

Description automatically generatedLebanese Universityالجامعة اللبنانية

Faculty of Engineeringكلية الهندسة

Branch-3الفرع الثالث

**Java Parallel Programming Project Proposal**

***Prepared by:***

* Omar Ismail 6304
* Roudy Barakat 6305

**Presented for**:

Dr. Mohammed Aoude

**Date**: 11 / 6 / 2025

**Title:** Parallel Color Image Segmentation with GUI for Performance Analysis (RGB Thresholding)

**Prepared by:**

* Omar Ismail (ULFG 3 – Telecommunications Engineering)
* Roudy Barakat (ULFG 3 – Telecommunications Engineering)

**Presented for:** Dr. Mohammed Aoude

**1. Team Roster**

* **Student 1:** Omar Ismail (ID: 6304, ULFG 3 – Telecommunications Engineering)
* **Student 2:** Roudy Barakat (ID: 6305, ULFG 3 – Telecommunications Engineering)

**2. Domain & Problem Statement**

**Domain:** Image Processing

**Problem:** Color image segmentation, particularly RGB thresholding, is a computationally intensive task. When applied sequentially to high-resolution images, it leads to prolonged execution times and inefficient CPU utilization (high CPU idling). This project addresses this bottleneck by implementing a parallel segmentation solution. The goal is to significantly accelerate the segmentation process and ensure near-full utilization of available CPU cores by distributing the workload.

**3. Sequential Baseline**

The sequential baseline version of the image segmentation process will:

* Load an RGB image (e.g., .png or .jpg format).
* Iterate through each pixel, applying thresholding independently to the Red, Green, and Blue (RGB) color components based on a user-defined threshold and segmentation type (e.g., grayscale conversion, red channel highlight, green channel highlight, or a custom RGB condition).
* Output a new BufferedImage containing the segmented result.
* Include an optional delay mechanism for step-by-step visualization, demonstrating the inherent sequential processing.

**4. Parallelization Strategy**

We will implement a ForkJoinPool-based parallel image segmentation. The strategy employs a divide-and-conquer approach:

* The input image will be logically split into horizontal blocks (strips of rows).
* Each block will be assigned to a RecursiveAction task within the ForkJoinPool.
* These tasks will concurrently apply the same RGB thresholding logic to their respective image blocks.
* The LEAF\_TASK\_THRESHOLD (the minimum number of rows a task processes before further splitting) will be dynamically calculated to optimize for the number of available threads and image height, balancing overhead and parallelism.
* The shared BufferedImage for the output will be updated directly by concurrent tasks, leveraging the JPanel.repaint() mechanism for live visualization of the parallel process.

**5. User Interface (UI) - Enhancing Usability and Demonstrability**

To enhance the project's usability, visualization, and demonstrability, a comprehensive Swing-based Graphical User Interface (SegmentationGUI.java) has been developed. The UI serves as a central dashboard for:

* **Image Upload:** A button to easily select and load source images for segmentation. A preview area displays the selected image.
* **Parameter Configuration:** Intuitive input fields for:
  + **Threshold:** (0-255) for segmentation logic.
  + **Segmentation Type:** Dropdown for grayscale, red, green, custom modes.
  + **Number of Threads:** To control the parallelism of the ForkJoinPool.
  + **Delay (ms/update):** An optional parameter to introduce a delay in milliseconds per image update, crucial for visualizing the step-by-step progression of both sequential and parallel algorithms.
* **Live Progress Displays:** Two dedicated LiveImageDisplay panels (seqLivePanel and parLivePanel) will show the real-time transformation of the image as the sequential and parallel algorithms process it, providing immediate visual feedback on their execution patterns.
* **Action Buttons:**
  + **"Start Segmentation"**: Initiates a single run of both sequential and parallel algorithms, displaying their respective times and calculated metrics.
  + **"Run Scalability Test"**: Executes the parallel algorithm multiple times with varying thread counts (1, 2, 4, 8, and system's max cores) to gather data for scalability analysis.
* **Performance Metrics Display:** A large JTextArea (metricsTextArea) will dynamically update with detailed performance statistics, including:
  + Execution times (Sequential and Parallel).
  + Calculated Speed-up (Sp​=Ts​/Tp​).
  + Approximate Memory Footprint and Memory Overhead Ratio.
  + Scalability test results for various thread counts.
* **Combined Output Display:** Upon completion of a standard segmentation run, a separate window will pop up showing the final sequential output and parallel output side-by-side for direct visual comparison.

The UI is designed to be user-friendly, allowing for easy experimentation with different parameters and direct observation of the performance gains and parallel execution behavior.

**6. Performance Metrics**

We will rigorously measure and report on the following performance metrics:

* **Speed-up (**Sp​=Ts​/Tp​**):**
  + **Target:** ≥3× on an 8-core CPU (aim for 70% of ideal speed-up).
  + **Notes:** Backed by data from multiple runs. Measured using System.nanoTime() for precise timing.
* **Scalability:**
  + **Target:** Performance should improve with more cores (explain observed plateau).
  + **Notes:** Demonstrated through a "thread-sweep" test (running with 1, 2, 4, 8, and N threads, where N is the number of available CPU cores). The GUI provides data points for this, which can be used to construct a scalability curve.
* **CPU Utilization:**
  + **Target:** ≥85 during the compute phase.
  + **Notes:** While precise in-JVM measurement is challenging, the efficient task distribution of ForkJoinPool aims to keep cores busy. We will discuss observed utilization patterns.
* **Memory Overhead:**
  + **Target:** ≤2× sequential footprint.
  + **Notes:** No reckless allocation. Measured by approximating heap memory usage before and after computations, with System.gc() calls to aid consistency.

**7. Risk Analysis**

* **Memory Bottlenecks:** RGB thresholding is inherently more memory-intensive than grayscale. Careful management of BufferedImage copies and pixel array reuse (as implemented) is crucial to avoid excessive memory allocation and GC overhead.
* **Fork/Join Granularity Tuning:** The LEAF\_TASK\_THRESHOLD is critical. If too fine-grained, task creation/scheduling overhead can negate parallelism gains; if too coarse, cores may idle. Dynamic adjustment and empirical tuning will be applied.
* **File I/O Exclusion:** Performance timings will strictly exclude file I/O operations (image loading/saving) to isolate and measure only the computational performance of the segmentation algorithms.
* **Swing Event Dispatch Thread (EDT) Responsiveness:** Long-running computations can block the EDT, freezing the UI. The use of SwingWorker ensures that heavy processing runs on a background thread, keeping the GUI responsive and allowing live progress updates.