**PARALLEL AND DISTRIBUTED PROGRAMMING**

**~ FINAL PROJECT ~**

**Requirement:** Compute the Hough transform of an image. Each project will have 2 implementations: one with "regular" threads or tasks/futures, and one distributed (possibly, but not required, using MPI).

**What is the Hough transform?** A feature extraction method that is used to detect lines in an image.

**CHAPTER 1:** THE “REGULAR” IMPLEMENTATION

**Overview**

This program performs line detection on an image using the Hough Transform. It utilizes multithreading to divide the computational workload across multiple threads, allowing for faster processing on multi-core systems. The detected lines are drawn on the original image and saved to disk.

This document details the algorithms, synchronization techniques, and estimated performance metrics used in the program.

**1. Algorithms**

**1.1 Image Preprocessing**

* **Input**: The program accepts a single image file path as a command-line argument.
* **Steps**:
  1. The image is loaded in color using cv::imread.
  2. The image is converted to greyscale using cv::cvtColor. Grayscale images are necessary for edge detection and for Hough Transform to reduce computational complexity.

**1.2 Edge Detection (Canny Algorithm)**

* The **Canny Edge Detector** is used to identify the edges in the image. It works in the following steps:
  1. **Noise Reduction**: Applies a Gaussian filter to smooth the image and reduce noise.
  2. **Gradient Calculation**: Computes intensity gradients using the Sobel operator to highlight regions with high spatial intensity changes.
  3. **Non-Maximum Suppression**: Ensures that only local maxima along the gradient direction are considered edges.
  4. **Double Thresholding**: Identifies strong edges (gradient > 200) and weak edges (gradient between 100 and 200). Weak edges are retained only if connected to strong edges.

**1.3 Line Detection (Hough Transform)**

* The **Hough Transform** detects straight lines in the edge-detected image by converting points from Cartesian coordinates to polar coordinates:
  + Accumulators are used to count occurrences of, and peaks in the accumulator space correspond to lines.

**1.4 Multithreading for Parallel Processing**

* The grayscale image is divided into horizontal chunks, with each thread processing one chunk:
  1. Each thread applies the **Canny Edge Detector** to its chunk.
  2. Each thread uses the **Hough Transform** to detect lines in its chunk.
  3. The detected lines from each thread are merged into a shared data structure (std::vector<cv::Vec2f>).

**2. Synchronization in Parallelized Implementation**

**2.1 Shared Data Structure**

* The vector sharedLines is used to store lines detected by all threads. Threads add their detected lines to this shared vector.

**2.2 Mutex for Synchronization**

* A std::mutex (linesMutex) ensures that threads access sharedLines safely:
  + Each thread locks the mutex using std::lock\_guard<std::mutex> before writing to the shared vector.
  + This prevents race conditions and ensures data integrity.

**2.3 Thread Creation and Joining**

* Threads are created using std::thread and passed:
  + Thread-specific parameters (start and end rows).
  + References to the grayscale image and shared data structure.
* The main thread waits for all threads to complete using thread.join().

**3. Performance Measurement**

|  |  |
| --- | --- |
| **Number of threads** | **Time taken (in ms)** |
| 1 thread | 72 |
| 2 threads | 65 |
| 4 threads | 67 |
| 6 threads | 60 |

**4. Code Walkthrough**

**Main Function**

1. **Image Loading**:
   * Loads and converts the image to grayscale.
2. **Chunk Division**:
   * Divides the image into chunks for threads.
3. **Thread Execution**:
   * Processes each chunk in a separate thread.
4. **Line Drawing**:
   * Draws detected lines on the original image and saves it to disk.

**processChunk Function**

1. **Edge Detection**:
   * Detects edges in the assigned chunk using cv::Canny.
2. **Hough Transform**:
   * Detects lines in the edge-detected image using cv::HoughLines.
3. **Synchronization**:
   * Uses std::mutex to safely add detected lines to the shared vector.

**5. Conclusion**

This program demonstrates efficient parallel processing for line detection using OpenCV. While multithreading significantly speeds up computation, synchronization overhead limits scalability.

**CHAPTER 2:** THE DISTRIBUTED IMPLEMENTATION

**Overview**

The program divides an input grayscale image among multiple processes to compute line detection using the Hough Transform algorithm.

The system uses the MPI library for inter-process communication and OpenCV for image processing tasks. The goal is to improve performance by parallelizing the work across multiple processes, ensuring scalability on distributed systems.

**1. Algorithms**

**Master Process Workflow**

The master process coordinates the entire computation by:

* **Loading and Preprocessing:**
  + Reads the input image.
  + Converts the image to grayscale.
  + Applies Gaussian blur to reduce noise.
* **Dividing Work:**
  + Divides the rows of the image into equal chunks for each worker process.
  + Sends the required data (start row, end row, image properties, and pixel data) to each worker process.
* **Collecting Results:**
  + Receives the detected lines from each worker process.
  + Aggregates the results and overlays the lines on the original image.
* **Saving the Output:**
  + Saves the final image with detected lines.

**Worker Process Workflow**

Each worker process performs the following:

* **Receiving Data:**
  + Receives its assigned chunk of the image from the master.
* **Edge Detection:**
  + Applies the Canny edge detection algorithm to its chunk.
* **Hough Line Detection:**
  + Computes the Hough Transform on the edges to detect lines.
* **Sending Results:**
  + Sends the detected lines back to the master process.

**2. Parallelization and Synchronization**

**MPI Communication**

The program employs MPI for communication between processes:

* **Master to Worker Communication:**
  + Sends image chunk details and pixel data using MPI\_Send.
* **Worker to Master Communication:**
  + Sends the detected lines and their count back to the master using MPI\_Send.

**Data Distribution and Aggregation**

* The master divides the image into chunks, ensuring each worker receives approximately equal rows.
* The master collects results from all workers and aggregates them into a single set of lines.

**Synchronization Mechanisms**

1. **Barriers and Consistency:**
   * The program relies on point-to-point communication (e.g., MPI\_Send and MPI\_Recv) to ensure synchronization.
2. **Avoiding Deadlocks:**
   * Each communication pair has a clear sequence to prevent deadlocks (e.g., master sends data first, and workers respond).

**3. Performance Measurement**

|  |  |
| --- | --- |
| **Number of threads** | **Time taken (in ms)** |
| 2 threads | 126 |
| 4 threads | 120 |
| 6 threads | 115 |

**4. Code Walkthrough**

**Main Function**

1. **MPI Initialization**:
   * Initializes the MPI environment.
   * Retrieves the rank (ID) of the current process and the total number of processes in the communicator.
2. **Argument Validation**:
   * Ensures that the program is run with exactly one additional argument (the image path).
   * If the validation fails, process 0 (master) prints a usage message, and all processes finalize the MPI environment.
3. **Role Delegation**:
   * Process 0 (master) executes the masterProcess function to manage image chunk distribution and collect results.
   * All other processes (workers) execute the workerProcess function to process assigned chunks of the image.
4. **MPI Finalization**:
   * Ensures all processes clean up and exit the MPI environment properly.

**Master Process (masterProcess)**

1. **Image Loading**:
   * Reads the input image and converts it to grayscale.
   * Applies a Gaussian blur to smooth the image and reduce noise for better edge detection.
2. **Chunk Division**:
   * Divides the image rows among the workers (processes other than the master).
   * Calculates rowsPerProcess and accounts for any remainingRows to ensure all rows are processed.
3. **Chunk Distribution**:
   * Sends the following data to each worker:
     + Start and end row indices for their chunk.
     + Image metadata (imgCols, imgType, etc.).
     + Pixel data for their chunk using MPI\_Send.
4. **Result Collection**:
   * Receives the number of detected lines and the corresponding data from each worker using MPI\_Recv.
   * Aggregates all detected lines into a single allLines vector.
5. **Line Drawing**:
   * Iterates through the aggregated lines and draws them on the original image.
   * Saves the output image as output\_image.jpg.

**Worker Process (workerProcess)**

1. **Data Reception**:
   * Receives its assigned chunk's metadata and pixel data from the master.
   * Constructs the cv::Mat image chunk using the received data.
2. **Edge Detection**:
   * Applies the Canny edge detection algorithm on the chunk.
3. **Hough Transform**:
   * Detects lines in the edge-detected chunk using the Hough Line Transform.
4. **Result Transmission**:
   * Sends the number of detected lines and the corresponding data back to the master using MPI\_Send.

**Conclusion**

This implementation demonstrates the effectiveness of MPI in parallelizing the Hough line detection algorithm. The division of work among processes and efficient communication ensures scalability, making it suitable for large-scale image processing tasks.