# Parallel Image Processing

## Introduction

Parallel image processing refers to the technique of simultaneously performing multiple image processing tasks or operations on different portions of an image or on multiple images concurrently. It leverages parallel computing architectures and algorithms to achieve faster and more efficient image processing.

What is the difference between Traditional sequential image processing Parallel image processing?

The main difference between traditional sequential image processing and parallel image processing lies in how the image processing tasks are executed.

## Sequential Image Processing:

In traditional sequential image processing, the image is processed one after another in a sequential manner. Each pixel or region of the image is processed individually, and the output of one processing step becomes the input for the next step. The processing steps are dependent on each other, meaning that the output of one step determines the input for the next step. This sequential execution limits the performance and efficiency of image processing algorithms, especially when dealing with large images.

## Parallel Image Processing:

Parallel image processing, on the other hand, involves dividing the image or the image processing tasks into smaller subtasks that can be processed concurrently. These subtasks can be processed independently and simultaneously on multiple processing units or threads. Parallel processing leverages the power of parallel computing architectures, such as multi-core processors or GPUs, to perform computations in parallel and achieve faster processing times.

## Tools:

### MPI (Message Passing Interface):

MPI is a communication protocol and library used for parallel computing. It enables communication and coordination between multiple computing nodes in a distributed computing environment. MPI is commonly used in high-performance computing (HPC) to develop parallel programs and algorithms that can be executed across multiple processors or nodes.

Open CV (Open Source Computer Vision Library):

Open CV is a widely used open-source library for computer vision and image processing tasks Open CV provides a wide range of tools and functions to manipulate and analyze images and videos, perform various transformations, apply filters, detect and track objects, and much more.

### chrono (C++ Chrono Library):

chrono is a C++ library that provides facilities for measuring and manipulating time durations and points in time.

## Implementation:

Read the image .

The root splits the image and distributes it to processors based on the number of processors. Each processor takes the part and applies the filter to it. The root assembles the parts of the image after processing and prints the final output.

## Filters

Gaussian Filter: A Gaussian filter is a type of linear filter commonly used for image smoothing or blurring.

Input

Code:

```
// 2-Create a separate image for Gaussian Blur

void applyGaussianBlur(cv::Mat& image) {

    double start_time = cv::getTickCount(); // Start

    cv::Mat blurredImg;

    cv::GaussianBlur(image, blurredImg, cv::Size(5, 5), 3);

    blurredImg.copyTo(image);

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\nGaussian operation completed successfully : " << execution_time << " seconds" << std::endl;

}
```

Output:

## Edge Detection Filter:

Edge detection is not typically referred to as a filter, but rather a technique used to identify boundaries or sharp transitions in an image.

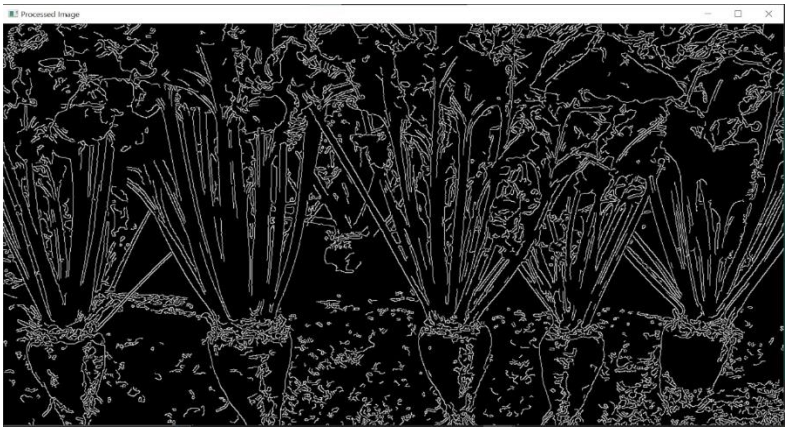
### Input

### Code:

```
// 3-Create a separate image for edge detection
cv::Mat edgeDetectionImage(cv::Mat& image) {
    cv::Mat edge_image;
    Canny(image, edge_image, 100, 200);
    image = edge_image.clone();
    return edge_image;
}
void edgeDetection(cv::Mat& local_image) {
    double start_time = cv::getTickCount(); // Start
    local_image = edgeDetectionImage(local_image);
    double end_time = cv::getTickCount(); // End
    double execution_time = (end_time - start_time) / cv::getTickFrequency();
    std::cout << "\n\nEdge Detection operation completed successfully : " << execution_time << "
seconds" << std::endl;
}
// Create a separate image for rotateImage
cv::Mat rotateImage(const cv::Mat& image, double angle) {
    cv::Mat rotatedImg;
    cv::Point2f center(image.cols / 2.0f, image.rows / 2.0f);
    cv::Mat rotationMatrix = cv::getRotationMatrix2D(center, angle, 1.0);
```

```
    cv::warpAffine(image, rotatedImg, rotationMatrix, image.size());
    return rotatedImg;
}
```

## Output



## Histogram Equalization:

Histogram equalization is a technique used to enhance the contrast of an image. It redistributes the pixel intensities in an image to maximize the contrast by stretching the histogram across the entire range of possible values.

### Input:

### Code:

```
// 7-Function to perform histogram equalization on the local chunk of the image

void equalizeHistogram(cv::Mat& image) {

    double start_time = cv::getTickCount(); // Start

    cv::Mat equalizedImg;

    cv::cvtColor(image, image, cv::COLOR_BGR2GRAY); // Convert image to grayscale

    cv::equalizeHist(image, equalizedImg);

    equalizedImg.copyTo(image);

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\nequalize Histogram operation completed successfully : " << execution_time << " seconds" << std::endl;

}
```

### Output:

## Local Thresholding Filter:

Local thresholding is a technique where a threshold is applied to small local regions of an image instead of the entire image.

Input:

## Code:

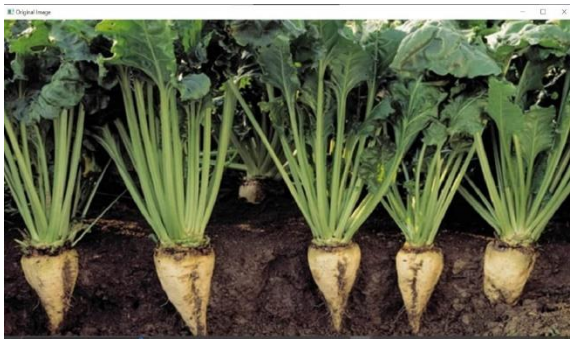
```
// 10-Function to perform local thresholding

void localThresholding(cv::Mat& image, int blockSize, double C) {

    cv::Mat grayImage;

    cv::cvtColor(image, grayImage, cv::COLOR_BGR2GRAY);

    cv::Mat thresholdedImage;

    cv::adaptiveThreshold(grayImage, thresholdedImage, 255, cv::ADAPTIVE_THRESH_GAUSSIAN_C, cv::THRESH_BINARY_INV, blockSize, C);

    thresholdedImage.copyTo(image);

}
```

## Output:

## Global Thresholding Filter:

Global thresholding applies a single threshold value to the entire image.

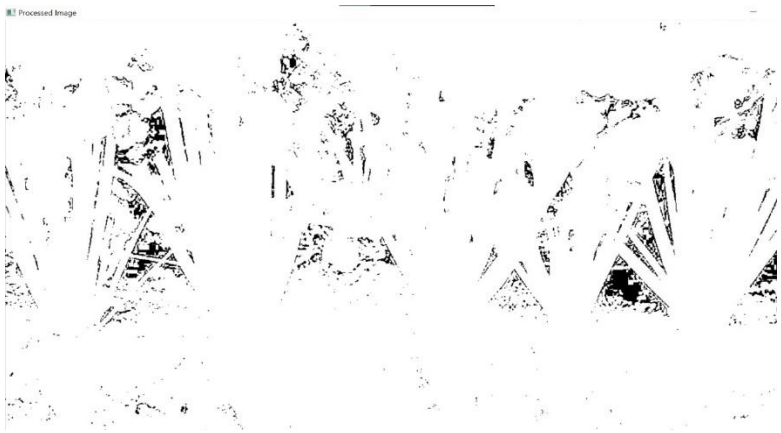
## Input:



## Code:

```
// 9-Function to perform global thresholding
void globalThresholding(cv::Mat& image, int threshold) {
    cv::Mat grayImage;
    cv::cvtColor(image, grayImage, cv::COLOR_BGR2GRAY);
    cv::Mat thresholdedImage;
    cv::threshold(grayImage, thresholdedImage, threshold, 255, cv::THRESH_BINARY);
    thresholdedImage.copyTo(image);
}
```

## Output:



## Median Filter:

A median filter is a nonlinear filter used to reduce noise in an image.

## Input:

## Code:

```
// 11-Function to apply median filter to the local chunk of the image
void applyMedianFilter(cv::Mat& image, int kernelSize) {
    double start_time = cv::getTickCount(); // Start
    if (kernelSize % 2 == 0) { // Check if the kernel size is even
        std::cout << "Kernel size must be an odd integer. Setting kernel size to 3." << std::endl;
        kernelSize = 3; // Set the kernel size to 3 if it's even
    }
    cv::Mat filteredImage;
    cv::medianBlur(image, filteredImage, kernelSize);
    filteredImage.copyTo(image);
    double end_time = cv::getTickCount(); // End
    double execution_time = (end_time - start_time) / cv::getTickFrequency();
    std::cout << "\n\nMedian filter operation completed successfully : " << execution_time << "
seconds" << std::endl;
}
```

## Output:

## Convert To Grayscale:

A gray-scale filter is not a standard term in image processing. However, if you are referring to converting a color image to a gray-scale image.

## Input:

## Code:

```
// 1-Create a separate image for grayScal

void convertToGrayscale(cv::Mat& image) {

    double start_time = cv::getTickCount(); // Start

    if (image.channels() == 3) {

        cv::cvtColor(image, image, cv::COLOR_BGR2GRAY);

    }

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\nGray operation completed successfully : " << execution_time << " seconds" << std::endl;

}
```

## Output:

## Image  Rotation:

The process of rotating the image changes the orientation of the image around a specific axis, which allows changing the angle and improving the orientation of the image without affecting its content.

## Input:

## Code:

```
// 4-Rotate the local chunk of the image

void rotateLocalImage(cv::Mat& local_image, double angle) {

    double start_time = cv::getTickCount(); // Start

    local_image = rotateImage(local_image, angle);

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\nRotate Image  operation completed successfully: " << execution_time << " seconds"
<< std::endl;

}
```

## Output:

## Image scaling

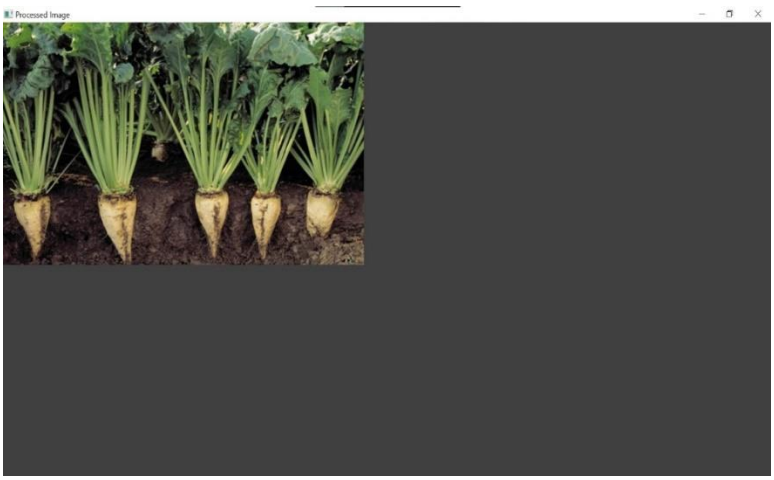
The image resizing process adjusts the size of the image to suit specific needs, allowing it to be enlarged or reduced without distorting the quality

## Input:

## Code:

```
// 5-Function to scale the local chunk of the image

void scaleImage(cv::Mat& image, double scale_factor) {    double start_time = cv::getTickCount(); // Start

    cv::Mat scaledImg;

    cv::resize(image, scaledImg, cv::Size(), scale_factor, scale_factor);

    scaledImg.copyTo(image);

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\nScale Image  operation completed successfully: " << execution_time << " seconds"
<< std::endl;

}
```

## Output



## Color Space Conversion:

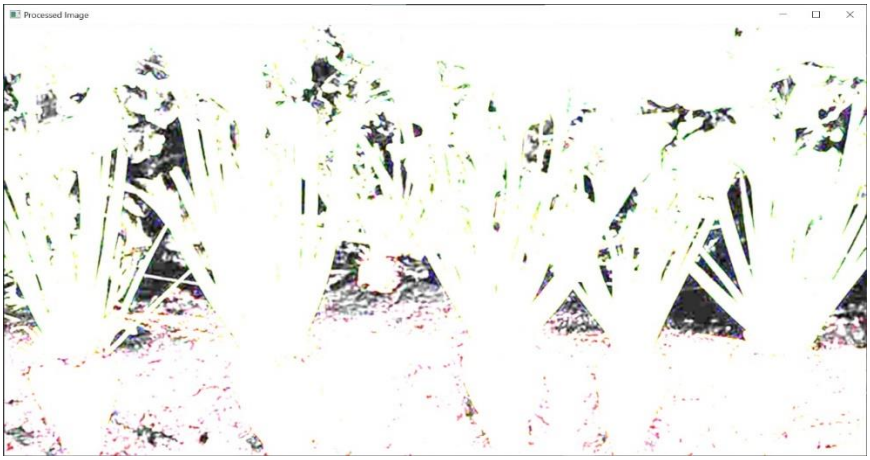
The process of color space conversion converts the representation of colors in an image from one color system to another, which helps in improving visualization and processing of images and adapting them to the requirements of different applications.

## Input:

## Code:

```
// 8-Function to convert the local chunk of the image to a different color space

void convertColorSpace(cv::Mat& image, int conversion_code) {

    double start_time = cv::getTickCount(); // Start

    cv::cvtColor(image, image, conversion_code);

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\nColor space operation completed successfully : " << execution_time << " seconds"
<< std::endl;

}
```

## Output:



## Color space operation:

The process of applying a color space operation modifies the color representation of the image without altering its content. By transforming the image's color model, such as converting from RGB to CMYK or HSV, adjustments to hue, saturation, brightness, or other color attributes can be made. This alteration enables fine-tuning of the image's visual appearance while preserving its underlying content.

## Input:

## Code:

```
// 6-Function to apply a custom filter kernel to the local chunk of the image

void applyCustomFilter(cv::Mat& image, cv::Mat& kernel) {

    double start_time = cv::getTickCount(); // Start

    cv::Mat filteredImg;

    cv::filter2D(image, filteredImg, -1, kernel);

    filteredImg.copyTo(image);

    double end_time = cv::getTickCount(); // End

    double execution_time = (end_time - start_time) / cv::getTickFrequency();

    std::cout << "\n\napply Custom Filter operation completed successfully: " << execution_time << " seconds" << std::endl;

}
```

## Output: