Parallel Implementation of Image Processing Algorithms

REVIEW REPORT

Submitted by

Malavika Jayakumar(19BCE2458)
Vishnu Anil Kumar Nair(19BCE2467)
Harsh Nair(19BCE2497)

Prepared for

CSE4001 – Parallel and Distributed Computing
Project Component

Submitted to

Dr. M.Narayanamoorthi

School of Computer Science and Engineering

CONTENTS

1. ABSTRACT:

Image segmentation is a branch of digital image processing which focuses on partitioning an image into different parts according to their features and properties. The primary goal of image segmentation is to simplify the image for easier analysis. In image segmentation, you divide an image into various parts that have similar attributes.This helps in a study of images for further analysis. Our project revolves around performing comparative analysis of serial and parallel implementations of Gaussian Blurring, Otsu thresholding and Sobel edge detection algorithm in Python (using PyMP).

2. INTRODUCTION:

Image segmentation is the process of dividing an image into multiple parts. It is typically used to identify objects or other relevant information in digital images. There are many ways to perform image segmentation including thresholding methods, colour-based segmentation, transform methods among many others. Alternately edge detection can be used for image segmentation and data extraction in areas such as image processing, computer vision, and machine vision. Image thresholding is a simple, yet effective, way of partitioning an image into a foreground and background. This image analysis technique is a type of image segmentation that isolates objects by converting grayscale images into binary images. *Image thresholding* is most effective in images with high levels of contrast.

2.1. Gaussian Blurring

In image processing, a Gaussian blur (also known as Gaussian smoothing) is the result of blurring an image by a Gaussian function. This is typically generated by convolving an image with an FIR(Finite impulse response) kernel of Gaussian values.
Each pixel's new value is set to a weighted average of that pixel's neighborhood. The original pixel's value receives the heaviest weight (having the highest Gaussian value) and neighboring pixels receive smaller weights as their distance to the original pixel increases. This results in a blur that preserves boundaries and edges better than other, more uniform blurring filters. In theory, the Gaussian function at every point on the image will be non-zero, meaning that the entire image would need to be included in the calculations for each pixel. Gaussian blurring is commonly used when reducing the size of an image. When downsampling an image, it is common to apply a low-pass filter to the image prior to resampling. This is to ensure that spurious high-frequency information does not appear in the downsampled image.
It its mathematically expressed as indicated below:

$$R = \sum_{i=-1}^{1} \sum_{j=-1}^{1} P_{x+i,y+j} \, S_{1+i,1+j}$$

P is the image matrix, S is the applied mask or kernel, and R denotes the resultant matrix.
Figure depicts the typical kernel used for Gaussian Blurring.

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

2.2. Otsu Thresholding

Converting a greyscale image to monochrome is a common image processing task. Otsu's method, named after its inventor Nobuyuki Otsu, is one of many binarization algorithms.This method involves iterating through all the possible threshold values and calculating a measure of spread for the pixel levels each side of the threshold, i.e. the pixels that either fall in foreground or background.

Here's understand the idea behind Otsu's approach- The method processes image histogram, segmenting the objects by minimization of the variance on each of the classes. Usually, this technique produces the appropriate results for bimodal images. The histogram of such image contains two clearly expressed peaks, which represent different ranges of intensity values.
The core idea is separating the image histogram into two clusters with a threshold defined as a result of minimization the weighted variance of these classes denoted by $\sigma_w^2(t)$.The core idea is separating the image histogram into two clusters with a threshold defined as a result of minimization the weighted variance of these classes denoted by $\sigma_w^2(t)$.

The whole computation equation can be described as:
$\sigma_w^2(t) = w_1(t)\sigma_1^2(t) + w_2(t)\sigma_2^2(t)$, where $w_1(t), w_2(t)$ are the probabilities of the two classes divided by a threshold t, which value is within the range from 0 to 255 inclusively.

There are actually two options to find the threshold. The first is to minimize the within-class variance defined above $\sigma_w^2(t)$, the second is to maximize the between-class variance using the expression below:

$\sigma_b^2(t) = w_1(t)w_2(t)[\mu_1(t) - \mu_2(t)]^2$, where μ_i is a mean of class i.

The probability P is calculated for each pixel value in two separated clusters C_1, C_2 using the cluster probability functions expressed as:

$$w_1(t) = \sum_{i=1}^{t} P(i),$$
$$w_2(t) = \sum_{i=t+1}^{I} P(i)$$

Despite the fact that the method was announced in 1979, it still forms the basis of some complex solutions. Let's take as an example an urgent task of robotic mapping, concluding in accurate spatial representation of any environment covered by a robot. This information is key for a properly robot autonomous functioning.

2.3. Sobel Edge Detection

Sobel Edge Detection or sobel operator is a type of image processing and computer vision algorithm that emphasizes edges. It is particularly useful in edge identification algorithms.

Sobel edge operator is a derivative mask and is used for edge detection. Sobel operator is used to detect two kinds of edges in an image:

a. Vertical direction
b. Horizontal direction

When applied to an image, the Sobel operator basically employs the convolution method to obtain the edges.
The horizontal mask is used to identify the horizontal edges and the vertical mask for vertical direction.

Horizontal mask (sobel - Y):

1	2	1
0	0	0
-1	-2	-1

$$\underset{\text{y - Derivative}}{\begin{vmatrix} 1 \\ 0 \\ -1 \end{vmatrix}} * \underset{\text{1D Gaussian Filter}}{\overline{\underline{\begin{matrix} 1 & 2 & 1 \end{matrix}}}} = \underset{\text{Sobel - Y}}{\begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix}}$$

Vertical mask:

-1	0	1
-2	0	2
-1	0	1

$$\underset{\text{1D Gaussian Filter}}{\begin{vmatrix} 1 \\ 2 \\ 1 \end{vmatrix}} * \underset{\text{x - Derivative}}{\overline{\underline{\begin{matrix} 1 & 0 & -1 \end{matrix}}}} = \underset{\text{Sobel - X}}{\begin{vmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{vmatrix}}$$

Here the coefficients of masks are not fixed and they can be adjusted according to our requirement unless they do not violate any property of derivative masks.

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \qquad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

***A -> Image matrix**

The following equation is used to the Gx and Gy matrices to generate the final image that incorporates both horizontal and vertical images.

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

3. LITERATURE SURVEY

3.1.1. ***<u>Nazma Nausheen, Ayan Seal, Pritee Khanna, and Santanu Halder. 2018. A FPGA based implementation of Sobel edge detection. Microprocess. Microsyst. 56, C (February 2018), 84–91</u>***.

In this paper the authors discuss how they have implemented the image/video processing on the Zynq platform in a Hardware/Software (HW/SW) co-design approach and various hardware related issues. The use of this demonstration shows a possible framework for the hardware acceleration of image/video processing applications. In the future, using OpenCV the optimized FPGA implementation can be achieved, which is relatively faster than the hardware/software Co-Design.

They have detected all of the edges in the picture after locating all of the large variances in brightness. Sobel Edge detection is a popular edge detection algorithm in image processing. Sobel, along with Canny edge detection and Prewitt edge detection, is one of the most widely used edge detection algorithms in today's technology.

The Sobel Edge Detection Algorithm is a gradient-based edge detection approach that discovers edges using horizontal mask (HM) and vertical mask (VM) .

3.1.2. ***Priyanka, Rishabh Shukla, Laxmi Shrivastava.2020 .Image Restoration of Image with Gaussian Filter.International Research Journal of Engineering and Technology (IRJET) .e-ISSN: 2395-0056.Volume 07. Issue 12 Dec 2020.***

This paper focuses on image restoration, which often includes image deblurring and filtering.

Image restoration is concerned with the reconstruction of blur parameters of a clean image from a noisy and distorted one.

Image restoration aims to reduce distortion by post-processing the image. In image processing, Gaussian smoothing (also known as Gaussian blur) results in distorting an image by Gaussian operate. The identical convolving of the image by a Gaussian operates is relatively related to a Gaussian blur toward the image.

This is also known as a two-dimensional pathological function. In contrast, a lot of exactly replicate the under exposure result would be a circle by the convolving (i.e., circular blur box).

While the Gaussian function is mapped to another Gaussian function with a different width under the Fourier 11 transform, Gaussian blur so low pass filter applying Gaussian blur has the outcome of removing the image's high-frequency constituents.

The Gaussian blur technique is especially useful for filtering photos with a lot of noise, because the filtering results revealed a relative independence on the noise characteristics, as well as a robust dependence on the variance worth of the Gaussian kernel.

3.1.3. ***Elhanan Elboher, Michael Werman(2011) .Efficient and Accurate Gaussian Image Filtering Using Running Sums .arXiv:1107.4958v1.25 Jul 2011***

This paper presents a simple and efficient method to convolve an image with a Gaussian kernel. The computation is performed in a constant number of operations per pixel using running sums along the image rows and columns. They investigate the error function used for kernel approximation and its relation to the properties of the input signal. Based on natural image statistics a quadratic form kernel error function has been proposed so that the output image l2 error is minimized. They've applied the proposed approach to approximate the Gaussian kernel by linear combination of constant functions. This results in a very efficient Gaussian filtering method. Our experiments show that the proposed technique is faster than state of the art methods while preserving a similar accuracy.

Image filtering is a commonly used image processing tool that requires fast and efficient computation. When the kernel size increases, direct computation of the kernel response needs more operations and the process gets slower.

In this paper, the authors present an efficient filtering algorithm for separable non uniform kernels and use it for highly fast and accurate Gaussian filtering.

The method is based on one-dimensional running sums (integral images) along single rows and columns of the image. The proposed algorithm is extremely basic and may be built in only a few lines of code.

Complexity analysis, as well as experimental data, reveal that it is faster than state-of-the-art approaches for Gaussian convolution while maintaining similar approximation accuracy.

3.1.4. *Goh, T.Y., Basah, S.N., Yazid, H., Safar, M.J., & Saad, F.S. (2018). Performance analysis of image thresholding: Otsu technique. Measurement, 114, 298-307.*

Image thresholding is commonly used as the first step in many algorithms for image analysis, object representation, and visualization. Although several image thresholding approaches have been suggested in the literature and their application is well understood, their performance analyses are quite limited.

In this paper, the authors have critically analyzed the viability of successful image thresholding under a variety of scene parameters. The emphasis is on the Otsu method image thresholding approach, which is widely utilized in many computer vision applications. They've also established a set of requirements to ensure successful image segmentation. Experiment using real-image data to validate the viability of those settings. The outcome demonstrates the capability of the proposed conditions to accurately forecast the outcome of image thresholding using the Otsu approach.

In practice, the success of image thresholding may be anticipated ahead of time using obtainable scene parameters.

3.1.5. *Mohd. Aquib Ansari , Diksha Kurchaniya and Manish Dixit (2017).A Comprehensive Analysis of Image Edge Detection Techniques.International Journal of Multimedia and Ubiquitous Engineering Vol.12, No.11 (2017), pp.1-12*

One of the most essential goals of image processing is to efficiently evaluate the image's content and extract meaningful and valuable information from it.Various image interpretation researchers have raised a lot of notice about it.One of the most difficult tasks in picture interpretation is appropriately mining the image's edges information.Edges are key characteristics of an image and can be produced from the object's outlines.In image analysis and processing, edge detection is commonly utilized. To detect the edges, various algorithms are available.In this paper, the comprehensive analysis is done on the several edge detection techniques such as Prewitt, Sobel, Canny, Roberts and Laplacian of Gaussian. It is experimentally observed that the Canny edge detector is working better than others. This work is implemented on Matlab R2015a.

4. WORKING

4.1. Parallelization

Using threads, the sequential algorithms described in sections 2.1, 2.2, and 2.3 have been parallelized. A thread of execution is the smallest set of preprogrammed instructions that may be managed independently by a scheduler, which is often a component of the operating system.A process can have several threads, but only one is referred to be the master thread. All of these threads can run at the same time while sharing all or some of the resources indicated in the code. The way resource sharing is implemented differs from one platform to the next.

The algorithms outlined in the preceding sections are surprisingly parallel because they are naturally parallel. The computation of one pixel's intensity is unrelated to the computation of another pixel's intensity. Theoretically, parallelizing the sequential algorithms will result in a speedup.
The Flynn's classification's Single Instruction Multiple Data (SIMD) architecture has been employed since the same computation must be done on all pixels. Each thread will carry out the same command on various pixels in the image.

4.2. Algorithms

4.2.1. Gaussian Blurring

Sequential Implementation:

```
For each pixel in image
        a. Compute the convolved value of the pixel using the
        horizontal and vertical Gaussian kernel.
        End for
 For each pixel in image
        a. Compute the resultant gradient approximation
   end for.
```

Parallel Implementation:

```
with pymp.Parallel(2) as p1:
with pymp.Parallel(2) as p2:

for i in p1.range(1,l+1):
for j in p2.range(1,b+1):
```

4.2.2. Otsu Thresholding

Sequential Implementation:

```
1. Compute Histogram and probabilities of each intensity
   level
2. Set up initial ωi(0) and μi(0)
3. Step through all possible thresholds t=1,... maximum
   intensity
      a. Update ωi and μi
      b. Compute σ2 (t)
4. Desired threshold corresponds to the maximum σ2 (t)
```

Parallel Implementation:

```
#/* binarization output into image2 */
x_size2 = x_size1
y_size2 = y_size1
with pymp.Parallel(2) as p1:
with pymp.Parallel(2) as p2:
for y in p1.range(0,y_size2):
     for x in p2.range(0,x_size2):
          if (image1[y][x] > threshold):
               image2[y][x] = MAX_BRIGHTNESS
          else:
               image2[y][x] = 0
```

4.2.3. Sobel Edge Detection

Sequential Implementation:

```
1. For each pixel in image 19
      a. Compute the convolved value of the pixel using the
      horizontal Sobel kernel
    end for.
2. For each pixel in image
      a. Compute the convolved value of the pixel
      using the vertical Sobel kernel
    end for
3. For each pixel in image
      a. Compute the resultant gradient approximation
   end for
```

Parallel Implementation:

```
with pymp.Parallel(2) as p1:
with pymp.Parallel(2) as p2:

for i in p1.range(1,l+1):
     for j in p2.range(1,b+1):
          Making sections for computing Gx and Gy
          simultaneously
```

4.3. Code Snippets

4.3.1. Gaussian Blur(Serial)

```
convx = array([[1/16, 2/16, 1/16],
      [2/16, 4/16, 2/16],
      [1/16, 2/16, 1/16]])
l=face.shape[0]
b=face.shape[1]
padded = np.zeros((l+2,b+2))
for i in range(0,l):
    for j in range(0,b):
        padded[i+1][j+1]=face[i][j]


res = np.zeros((l,b), dtype='uint8')
i=None
j=None
```

```
for i in range(1,l+1):
    for j in range(1,b+1):
        res[i-1][j-1] = (convx[0][0]*padded[i-1][j-1] +
convx[0][1]*padded[i-1][j]+convx[0][2]*padded[i-1][j+1]+
                convx[1][0]*padded[i][j-
1]+convx[1][1]*padded[i][j] + convx[1][2]*padded[i][j+1]+
                convx[2][0]*padded[i+1][j-1]+
convx[2][1]*padded[i+1][j] +convx[2][2]*padded[i+1][j+1])


img = Image.fromarray(res)
```

4.3.2. Gaussian Blur(Parallel)

```
convx = array([[1/16, 2/16, 1/16],
      [2/16, 4/16, 2/16],
      [1/16, 2/16, 1/16]])
l=face.shape[0]
b=face.shape[1]
#padded = np.zeros((l+2,b+2))
padded = pymp.shared.array((l+2,b+2), dtype='uint8')
i=None
j=None
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(0,l):
            for j in p2.range(0,b):
                padded[i+1][j+1]=face[i][j]


res = pymp.shared.array((l,b), dtype='uint8')
i=None
j=None
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(1,l+1):
            for j in p2.range(1,b+1):
                res[i-1][j-1] = (convx[0][0]*padded[i-1][j-1] +
convx[0][1]*padded[i-1][j]+convx[0][2]*padded[i-1][j+1]+
                        convx[1][0]*padded[i][j-
1]+convx[1][1]*padded[i][j] + convx[1][2]*padded[i][j+1]+
                        convx[2][0]*padded[i+1][j-1]+
convx[2][1]*padded[i+1][j] +convx[2][2]*padded[i+1][j+1])


img = Image.fromarray(res)
```

4.3.3. Otsu Thresholding(Serial)

```
def otsu_th():
    print("Otsu's binarization process starts now.\n")
  #/* Histogram generation */
    for y in range(0,y_size1):
        for x in range(0,x_size1):
            hist[image1[y][x]] += 1

  #/* calculation of probability density */
    for i in range(0,GRAYLEVEL):
        prob[i] = float(hist[i]) / (x_size1 * y_size1)
    for i in range(0, 256):
        print("Serial: " + str(prob[i]))
  #/* omega & myu generation */
    omega[0] = prob[0]
    myu[0] = 0.0      #/* 0.0 times prob[0] equals zero */
    for i in range(1,GRAYLEVEL):
        omega[i] = omega[i-1] + prob[i]
        myu[i] = myu[i-1] + i*prob[i]

    '''/* sigma maximization sigma stands for inter-class variance and
determines optimal threshold value */'''
    threshold = 0
    max_sigma = 0.0
    for i in range(0,GRAYLEVEL-1):
        if (omega[i] != 0.0 and omega[i] != 1.0):
            sigma[i] = ((myu[GRAYLEVEL-1]*omega[i] - myu[i])**2) /
(omega[i]*(1.0 - omega[i]))
        else:
            sigma[i] = 0.0
        if (sigma[i] > max_sigma):
            max_sigma = sigma[i]
            threshold = i

    print("\nthreshold value = "+ str(threshold))

  #/* binarization output into image2 */
    x_size2 = x_size1
    y_size2 = y_size1
    for y in range(0,y_size2):
        for x in range(0,x_size2):
            if (image1[y][x] > threshold):
                image2[y][x] = MAX_BRIGHTNESS
            else:
                image2[y][x] = 0
```

4.3.3. Otsu Thresholding(Parallel)

```
def otsu_th():
    print("Otsu's binarization process starts now.\n")
    #/* Histogram generation */
    for y in range(0,y_size1):
        for x in range(0,x_size1):
            hist[image1[y][x]] += 1

  #/* calculation of probability density */
    for i in range(0,GRAYLEVEL):
        prob[i] = float(hist[i]) / (x_size1 * y_size1)
    for i in range(0, 256):
        print("Parallel: " + str(prob[i]))
      #/* omega & myu generation */
    omega[0] = prob[0]
    myu[0] = 0.0       #/* 0.0 times prob[0] equals zero */
    for i in range(1,GRAYLEVEL):
        omega[i] = omega[i-1] + prob[i]
        myu[i] = myu[i-1] + i*prob[i]
    '''/* sigma maximization
    sigma stands for inter-class variance and determines optimal
threshold value */'''
    threshold = 0
    max_sigma = 0.0
    for i in range(0,GRAYLEVEL-1):
        if (omega[i] != 0.0 and omega[i] != 1.0):
            sigma[i] = ((myu[GRAYLEVEL-1]*omega[i] - myu[i])**2) /
(omega[i]*(1.0 - omega[i]))
        else:
            sigma[i] = 0.0
        if (sigma[i] > max_sigma):
            max_sigma = sigma[i]
            threshold = i
   print("\nthreshold value = "+ str(threshold))
```

```
    #/* binarization output into image2 */
    x_size2 = x_size1
    y_size2 = y_size1
    with pymp.Parallel(2) as p1:
        with pymp.Parallel(2) as p2:
            for y in p1.range(0,y_size2):
                for x in p2.range(0,x_size2):
                    if (image1[y][x] > threshold):
                        image2[y][x] = MAX_BRIGHTNESS
                    else:
                        image2[y][x] = 0
```

4.3.5. Sobel Edge Detection(Serial)

```
convx = array([[-1, 0, 1],
      [-2, 0, 2],
      [-1, 0, 1]])
l=face.shape[0]
b=face.shape[1]
#padded = np.zeros((l+2,b+2))
padded = pymp.shared.array((l+2,b+2), dtype='uint8')
i=None
j=None

for i in range(0,l):
    for j in range(0,b):
        padded[i+1][j+1]=face[i][j]
```

```
res = pymp.shared.array((l,b), dtype='uint8')
i=None
j=None
```

```
for i in range(1,l+1):
    for j in range(1,b+1):
        res[i-1][j-1] = (convx[0][0]*padded[i-1][j-1] +
convx[0][1]*padded[i-1][j]+convx[0][2]*padded[i-1][j+1]+
                convx[1][0]*padded[i][j-1]+convx[1][1]*padded[i][j] +
convx[1][2]*padded[i][j+1]+
                convx[2][0]*padded[i+1][j-1]+ convx[2][1]*padded[i+1][j]
+convx[2][2]*padded[i+1][j+1])

        res[i-1][j-1]= (res[i-1][j-1]**2)
```

```
resy = pymp.shared.array((l,b), dtype='uint8')
i=None
j=None
convy = [[1, 2, 1],
    [0, 0, 0],
    [-1, -2, -1]
    ]
```

```
for i in range(1,l+1):
    for j in range(1,b+1):
        resy[i-1][j-1] = (convy[0][0]*padded[i-1][j-1] +
convy[0][1]*padded[i-1][j]+convy[0][2]*padded[i-1][j+1]+
                convy[1][0]*padded[i][j-1]+convy[1][1]*padded[i][j] +
convy[1][2]*padded[i][j+1]+
                convy[2][0]*padded[i+1][j-1]+ convy[2][1]*padded[i+1][j]
+convy[2][2]*padded[i+1][j+1])

        resy[i-1][j-1]= (resy[i-1][j-1]**2)
res2 = pymp.shared.array((l,b), dtype='uint8')

for i in range(0,l):
    for j in range(0,b):
        res2[i][j] = int((res[i-1][j-1]+resy[i-1][j-1]))
        if res2[i][j] > 15:
            res2[i][j]=255
```

4.3.6. Sobel Edge Detection(Parallel)

```
convx = array([[-1, 0, 1],
      [-2, 0, 2],
      [-1, 0, 1]])
l=face.shape[0]
b=face.shape[1]
#padded = np.zeros((l+2,b+2))
padded = pymp.shared.array((l+2,b+2), dtype='uint8')
i=None
j=None
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(0,l):
            for j in p2.range(0,b):
                padded[i+1][j+1]=face[i][j]


res = pymp.shared.array((l,b), dtype='uint8')
i=None
j=None
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(1,l+1):
            for j in p2.range(1,b+1):
                res[i-1][j-1] = (convx[0][0]*padded[i-1][j-1] +
convx[0][1]*padded[i-1][j]+convx[0][2]*padded[i-1][j+1]+
                        convx[1][0]*padded[i][j-
1]+convx[1][1]*padded[i][j] + convx[1][2]*padded[i][j+1]+
                        convx[2][0]*padded[i+1][j-1]+
convx[2][1]*padded[i+1][j] +convx[2][2]*padded[i+1][j+1])

                res[i-1][j-1]= (res[i-1][j-1]**2)
```

```
resy = pymp.shared.array((l,b), dtype='uint8')
i=None
j=None
convy = [[1, 2, 1],
    [0, 0, 0],
    [-1, -2, -1]
    ]
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(1,l+1):
            for j in p2.range(1,b+1):
                resy[i-1][j-1] = (convy[0][0]*padded[i-1][j-1] +
convy[0][1]*padded[i-1][j]+convy[0][2]*padded[i-1][j+1]+
                        convy[1][0]*padded[i][j-
1]+convy[1][1]*padded[i][j] + convy[1][2]*padded[i][j+1]+
                        convy[2][0]*padded[i+1][j-1]+
convy[2][1]*padded[i+1][j] +convy[2][2]*padded[i+1][j+1])

                resy[i-1][j-1]= (resy[i-1][j-1]**2)
```

```
res2 = pymp.shared.array((l,b), dtype='uint8')
with pymp.Parallel(2) as p1:
    with pymp.Parallel(2) as p2:
        for i in p1.range(0,l):
            for j in p2.range(0,b):
                res2[i][j] = int((res[i-1][j-1]+resy[i-1][j-1]))
                if res2[i][j] > 15:
                    res2[i][j]=255
```

5. REQUIREMENTS

5.1 Software Requirements

5.1.1 Jupyter Notebook

The Jupyter Notebook is an open-source web application that allows data scientists to create and share documents that integrate live code, equations, computational output, visualizations, and other multimedia resources, along with explanatory text in a single document.

5.1.2. PyMP

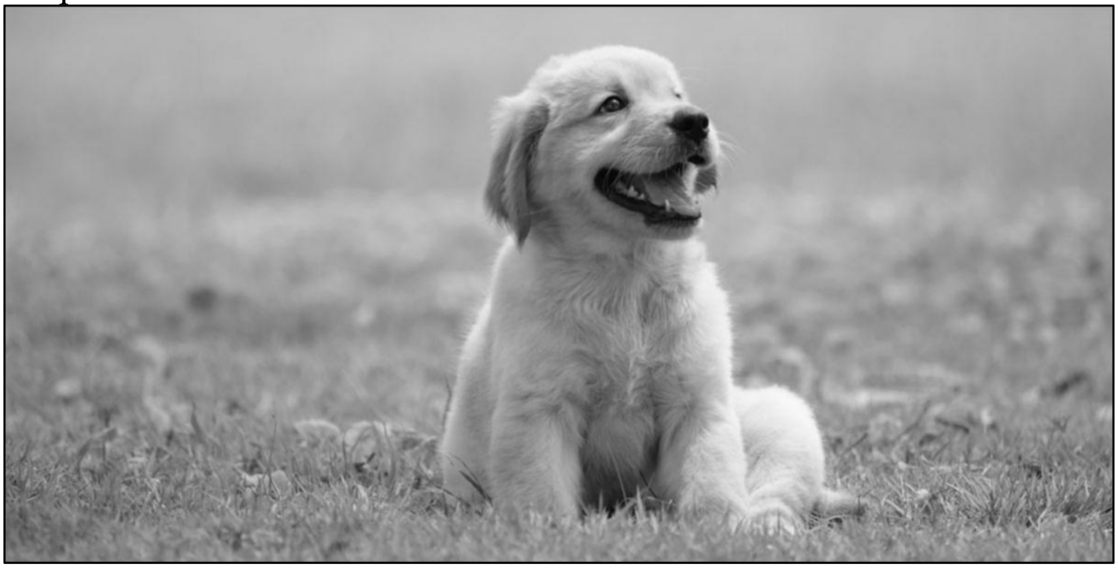
This package brings OpenMP-like functionality to Python. It takes the good qualities of OpenMP such as minimal code changes and high efficiency and combines them with the Python Zen of code clarity and ease-of-use.OpenMP is typically used for loop-level parallelism, but it also supports function-level parallelism.

6. RESULTS AND DISCUSSION

Input:

6.1. Gaussian Blurring -

Output:

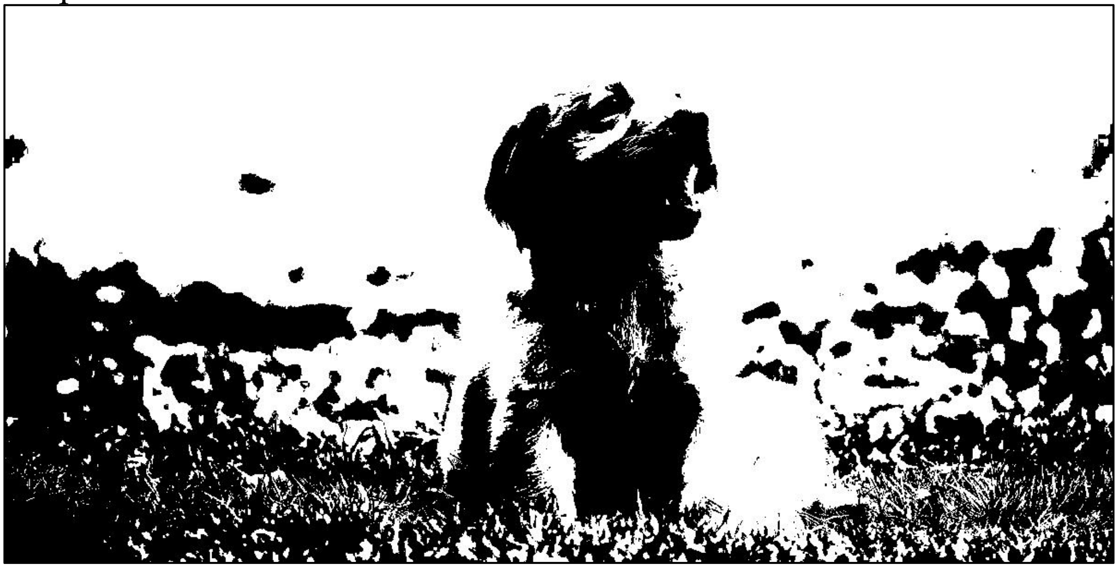


For serial implementation:
Image Resolution-(602, 1200)
Execution Time-0:00:11.408882

For parallel implementation:
Image Resolution-(602, 1200)
Execution Time-0:00:13.980055

6.2. Otsu Thresholding

Output:



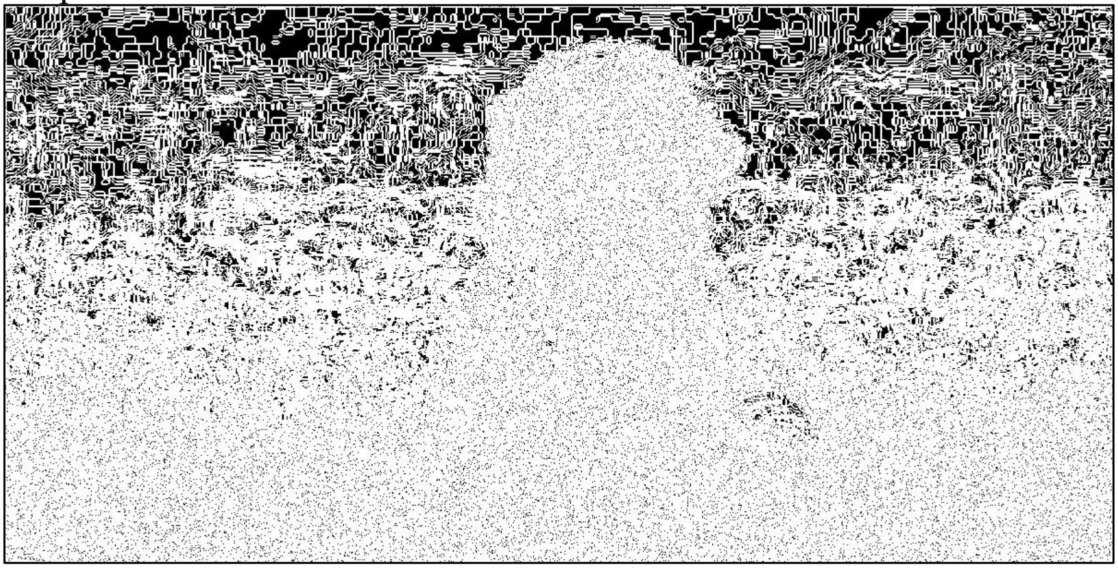
For serial implementation:
Image Resolution-(602, 1200)
Execution Time-0:00:06.644342

For parallel implementation:
Image Resolution-(602, 1200)
Execution Time-0:00:02.885848

Despite the restrictions of PyMP, the parallel code's performance boost over the serial implementation is rather noteworthy. The results above indicate the time it took to complete the serial and the parallel on providing an image as input. One appears to be quite large, while the other appears to be quite small. Python, as an interpreted and high-level abstraction language, is extremely slow in comparison to low-level languages like C, which are extremely fast. In this case, parallelization becomes a more important criterion. We see a significant reduction in the time required for execution.

6.3. Sobel Edge Detection

Output:



<u>For serial implementation:</u>
Image Resolution-(602, 1200)
Execution Time-0:00:50.510607

<u>For parallel implementation:</u>
Image Resolution-(602, 1200)
Execution Time-0:01:01.655974

7. CONCLUSION

Gaussian blurring, Otsu thresholding and Sobel edge detection algorithms were implemented in Python using PyMP. Python being an interpreted language of high level abstraction is very slow in comparison to low level languages like C, which are very fast. Parallelization becomes a more important criterion in this case. We observe that there is a significant reduction in time taken for execution for algorithms when parallelized.

8. REFERENCES

Nausheen, N., Seal, A., Khanna, P., & Halder, S. (2018). A FPGA based implementation of Sobel edge detection. *Microprocessors and Microsystems*, *56*, 84-91.

Priyanka, Rishabh Shukla, Laxmi Shrivastava.2020 .Image Restoration of Image with Gaussian Filter.International Research Journal of Engineering and Technology (IRJET) .e-ISSN: 2395-0056.Volume 07. Issue 12 Dec 2020.

Elboher, E., & Werman, M. (2012, November). Efficient and accurate Gaussian image filtering using running sums. In *2012 12th International Conference on Intelligent Systems Design and Applications (ISDA)* (pp. 897-902). IEEE.

Goh, T. Y., Basah, S. N., Yazid, H., Safar, M. J. A., & Saad, F. S. A. (2018). Performance analysis of image thresholding: Otsu technique. *Measurement*, *114*, 298-307.

Ansari, M. A., Kurchaniya, D., & Dixit, M. (2017). A comprehensive analysis of image edge detection techniques. *International Journal of Multimedia and Ubiquitous Engineering*, *12*(11), 1-12.