DDA_Exercise_2

Farjad Ahmed - 1747371
Semester: 1st Semester

Table of Contents

Question 1.1

In this question, Nsendall works by sending data from process 0 to all other nodes. If rank is 0, we get the data and each rank calls the EsendAll function.
At the end of this routine, comm.barrier is called, all ranks stop at this point unless all reach the barrier, this ensures that 0 has sent the data and all others receive after it is sent from 0.

```
def NsendAll(data):
    if data is not None:
        for i in range(1, P):
            comm.send( data, dest = i)
        # comm.barrier()
    else:
        get_data = comm.recv( source = root)
        # print('data received is:', get_data, 'and tag is', rank)
    comm.barrier()

if rank == 0:
    sdata = generate_vec(n)
else:
    sdata = None

NsendAll(sdata)
# EsendAll(sdata)
```

Question 1.2

In this part an efficient way of the earlier function was implemented. As per the description, when data is generated from rank 0. Each rank calls this function of EsendAll. Here, first rank comes and checks whether destA and destB exist or not, if they exist, it sends data to them.
All other ranks calling this function, go into the else condition and receive from address ‘recvProv’. After that they calculate the destinations and check if they exist, they send the data. Now, note that comm.barrier exists at the bottom of this function. Process 0 waits there, and all other ranks also reach this barrier. When all reach the barrier, the program proceeds.

```
def EsendAll(data):
    if data is not None:
        # print('i am here and rank is', rank)
        destA = 2 * rank + 1
        destB = 2 * rank + 2
        if destA < P:
            comm.send(data, destA)
        if destB < P:
            comm.send(data, destB)
    else:
        recvProc = int((rank - 1)/2)
        gdata = comm.recv(source = recvProc)
        destA = 2 * rank + 1
        destB = 2 * rank + 2
        if destA < P:
            comm.send(gdata, destA)
        if destB < P:
            comm.send(gdata, destB)
    comm.barrier()
```

```
if rank == 0:
    sdata = generate_vec(n)
else:
    sdata = None

# NsendAll(sdata)
EsendAll(sdata)
```

Question 1.3

Performance comparison:
The table below compares the processing time significantly reduced in the EsendAll routine as compared to NsendAll. For instance consider **NsendAll for 10**7 with 32 processors** shows time values in the range **10 seconds** while with **EsendAll for 10**7 with 32** processors, the processing time is in the range of **4 seconds**.

Function	Size	Cores	Time
NsendAll	10**3	16	NsendAll called running time: 0.159356 running time: 0.353664 running time: 0.234435 running time: 0.154435 running time: 0.005936 running time: 0.212556 running time: 0.311285 running time: 0.403474 running time: 0.357406 running time: 0.253844 running time: 0.312179 running time: 0.457626 running time: 0.324058 running time: 0.354276 running time: 0.17623 running time: 0.168365
NsendAll	10**5	16	NsendAll called running time: 0.34168 running time: 0.178134 running time: 0.283465 running time: 0.213065 running time: 0.231925 running time: 0.329728 running time: 0.274118 running time: 0.181243 running time: 0.271286 running time: 0.203372 running time: 0.220857 running time: 0.239515 running time: 0.326387 running time: 0.280601 running time: 0.133197 running time: 0.13159

NsendAll	10**7	16	NsendAll called running time: 11.579294 running time: 11.444745 running time: 11.654366 running time: 11.835539 running time: 11.544588 running time: 11.650189 running time: 11.57214 running time: 11.622536 running time: 11.768636 running time: 11.606119 running time: 11.636913 running time: 11.604832 running time: 11.881089 running time: 11.651413 running time: 11.856159 running time: 11.766235
NsendAll	10**3	32	NsendAll called running time: 0.172537 running time: 0.148881 running time: 0.219665 running time: 0.069618 running time: 0.200686 running time: 0.141653 running time: 0.055134 running time: 0.20399 running time: 0.178661 running time: 0.086361 running time: 0.077404 running time: 0.133247 running time: 0.144534 running time: 0.175987 running time: 0.095734 running time: 0.094168 running time: 0.171055 running time: 0.018971 running time: 0.16796 running time: 0.115057 running time: 0.143057 running time: 0.095651 running time: 0.272442 running time: 0.114388 running time: 0.168419 running time: 0.165381 running time: 0.17322 running time: 0.065185 running time: 0.1928 running time: 0.378791 running time: 0.104804 running time: 0.149381

NsendAll	10**5	32	NsendAll called running time: 0.206056 running time: 0.187944 running time: 0.255549 running time: 0.15103 running time: 0.134534 running time: 0.109323 running time: 0.142849 running time: 0.190408 running time: 0.329566 running time: 0.081076 running time: 0.178881 running time: 0.09134 running time: 0.128055 running time: 0.229687 running time: 0.172743 running time: 0.166574 running time: 0.102137 running time: 0.202634 running time: 0.118898 running time: 0.384548 running time: 0.162391 running time: 0.334592 running time: 0.183692 running time: 0.1559 running time: 0.156642 running time: 0.135706 running time: 0.233059 running time: 0.20998 running time: 0.177637 running time: 0.183657 running time: 0.224207 running time: 0.248054

NsendAll	10**7	32	NsendAll called running time: 10.248772 running time: 10.234881 running time: 10.149042 running time: 10.149038 running time: 10.092397 running time: 10.162036 running time: 10.145616 running time: 10.339876 running time: 10.168317 running time: 10.098901 running time: 10.220161 running time: 10.16308 running time: 10.160073 running time: 10.137831 running time: 10.184047 running time: 10.141286 running time: 10.304131 running time: 10.232828 running time: 10.26918 running time: 10.228204 running time: 10.215394 running time: 10.114933 running time: 10.172168 running time: 10.143166 running time: 10.216119 running time: 10.186161 running time: 10.119373 running time: 10.16234 running time: 10.152138 running time: 10.197926 running time: 10.287628 running time: 10.26453
EsendAll	10**3	16	EsendAll called running time: 0.064325 running time: 0.100437 running time: 0.007095 running time: 0.062726 running time: 0.180749 running time: 0.162566 running time: 0.016412 running time: 0.3188 running time: 0.183995 running time: 0.114449 running time: 0.098032 running time: 0.03059 running time: 0.219537 running time: 0.070231 running time: 0.216548 running time: 0.113163

EsendAll	10**5	16	EsendAll called running time: 0.024578 running time: 0.115125 running time: 0.063936 running time: 0.113291 running time: 0.184165 running time: 0.098175 running time: 0.121582 running time: 0.121919 running time: 0.109555 running time: 0.071311 running time: 0.184248 running time: 0.175277 running time: 0.207258 running time: 0.06772 running time: 0.129519 running time: 0.03002
EsendAll	10**7	16	EsendAll called running time: 1.582073 running time: 1.690314 running time: 1.581374 running time: 1.610968 running time: 1.523522 running time: 1.680243 running time: 1.604436 running time: 1.604463 running time: 1.59285 running time: 1.728755 running time: 1.601939 running time: 1.654255 running time: 1.645149 running time: 1.635159 running time: 1.553728 running time: 1.669127

EsendAll	10**3	32	EsendAll called running time: 0.022392 running time: 0.011731 running time: 0.080654 running time: 0.047669 running time: 0.047713 running time: 0.025652 running time: 0.127719 running time: 0.025106 running time: 0.071482 running time: 0.292801 running time: 0.153347 running time: 0.026444 running time: 0.282698 running time: 0.04726 running time: 0.050528 running time: 0.106073 running time: 0.071371 running time: 0.150871 running time: 0.052811 running time: 0.12903 running time: 0.039883 running time: 0.10857 running time: 0.053736 running time: 0.2079 running time: 0.041045 running time: 0.035639 running time: 0.098258 running time: 0.088738 running time: 0.109811 running time: 0.121605 running time: 0.102647 running time: 0.126395

EsendAll	10**5	32	EsendAll called running time: 0.021878 running time: 0.034672 running time: 0.15223 running time: 0.037712 running time: 0.059036 running time: 0.239659 running time: 0.112673 running time: 0.028945 running time: 0.028586 running time: 0.342289 running time: 0.366047 running time: 0.080619 running time: 0.074687 running time: 0.263982 running time: 0.021887 running time: 0.028099 running time: 0.271427 running time: 0.064754 running time: 0.033864 running time: 0.226815 running time: 0.272878 running time: 0.125013 running time: 0.025677 running time: 0.030372 running time: 0.280932 running time: 0.052482 running time: 0.024224 running time: 0.062875 running time: 0.110605 running time: 0.315311 running time: 0.150367 running time: 0.119758

EsendAll	10**7	32	EsendAll called running time: 4.100928 running time: 4.279331 running time: 4.1626 running time: 4.125122 running time: 4.229934 running time: 4.250827 running time: 4.177577 running time: 4.335814 running time: 4.182839 running time: 4.191158 running time: 4.263427 running time: 4.260635 running time: 4.220966 running time: 4.15334 running time: 4.146074 running time: 4.172264 running time: 4.201901 running time: 4.205717 running time: 4.128226 running time: 4.25607 running time: 4.271031 running time: 4.267099 running time: 4.255811 running time: 4.180725 running time: 4.217977 running time: 4.215956 running time: 4.237181 running time: 4.238527 running time: 4.20636 running time: 4.153557 running time: 4.151884 running time: 4.245647

Question 2

The image considered for this question is shown below, the resolution is 5874 × 3916.

Algorithm and Calculation for Gray-scale Image

This section contains the histogram implementation for GRAYSCALE and **RGB** images.

The algorithm flow is explained in steps below, followed by the results and the code.

1. The initializations are as follows, the most important one here is the '**bucket**' that is an array of **256** elements, that store the frequency of the **ith** intensity value at the **ith** pixel.

```
bucket = np.zeros(256)
imageName = 'building.jpg'
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
root = 0
```

2. Rank 0 reads and splits the image using np.array_split, in as many parts as total ranks. Moreover, it plots the original histogram from opencv to check with the actual result.

```
if rank == 0:
    original_histogram(imageName)
    sdata = cv2.imread(imageName, cv2.IMREAD_GRAYSCALE)
    print('shape of original image is', sdata.shape)
    sdata = np.array_split(sdata, size)
    # print(type(sdata))
    # print(sdata)

else:
    sdata = None
```

3. If rank is not root, the image data that is '**sdata**' is 'None'.
4. From root rank, the split chunks of the image are scattered using comm.scatter.
5. Now this scattered data received at each rank is a **list of np.arrays.**
6. For ease of calculations and simplicity, each split of data received at each rank is converted into a list of values by first converting the list of np.arrays to list of lists then to a flat_list that consists of all intensity values in a list.
7. Once we have a list of intensities, a custom implemented method mycounter() is called that returns an int class dictionary containing counts for each intensity value in the flat_list. It counts and stores values of counts in a dictionary. However, due to nested loops this consumes tremendous times in computing high resolution images but it is working. I have tested it with low resolution images of size 40x50 and that works fine for this counter function.

```
def mycounter(lst):
    mydict = {}
    for i1 in lst:
        mydict[i1] = 0
        for i2 in lst:
            if i1 == i2:
                mydict[i1] += 1
    return mydict

data = comm.scatter(sdata, root=0)
data = [l.tolist() for l in data]
# print('i am rank {} and i have {}'.format(rank, data))
flat_list = [item for sublist in data for item in sublist]
mycounts = Counter(flat_list)
# mycounts = mycounter(flat_list)
print(mycounts)
# print(mycounts)

for i in range(len(bucket)):
    if i in mycounts:
        bucket[i] = mycounts[i]
# print(bucket)
```

8. Now, each rank computes this count dictionary and stores them in **mycounts.** We need to find a way to combine all these intensity counts and sum their occurrences.
9. For this we need to put these in a format that could be used with comm.reduce to sum the data components coming from each rank. So we utilize an np.array called '**bucket'** of length 256. Each index in this bucket corresponds to the count of that **ith** intensity value. For example, bucket[23] would return how many times the intensity value '23' occurs in the image chunk being processed by the rank running it.

```
for i in range(len(bucket)):
    bucket[i] = mycounts[i]
# print(bucket)
```

10. Now, that we have frequency counts of each intensity value in a format that could be summed across each rank, comm.reduce is called here.

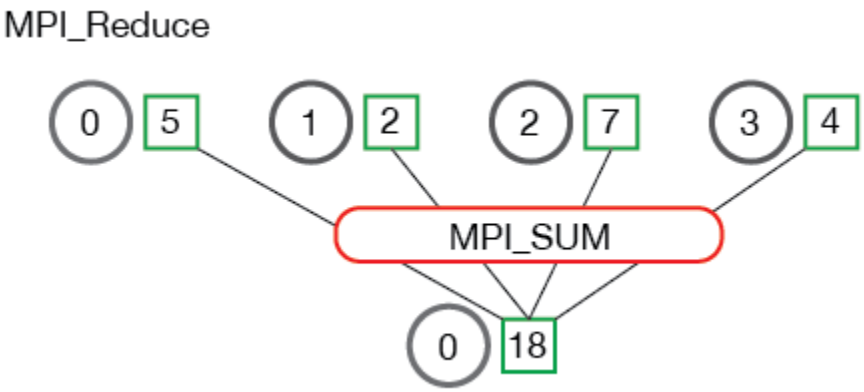
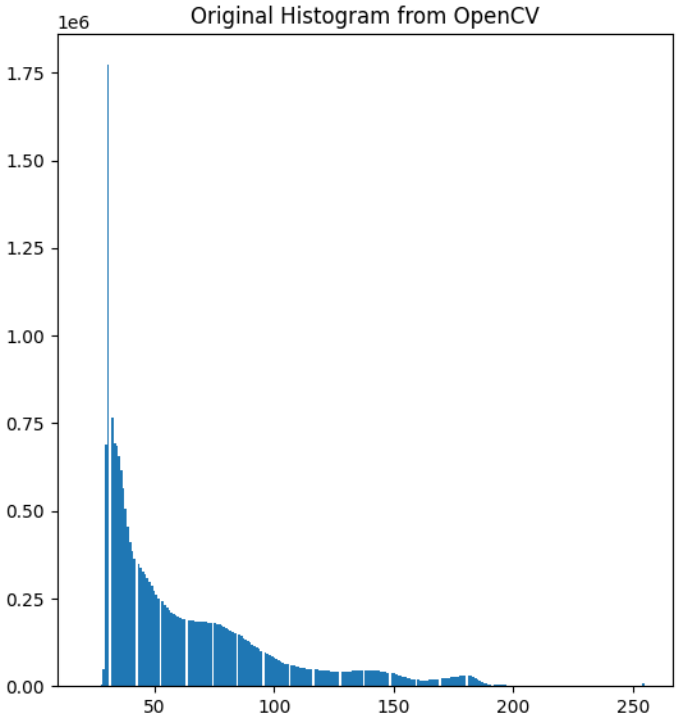


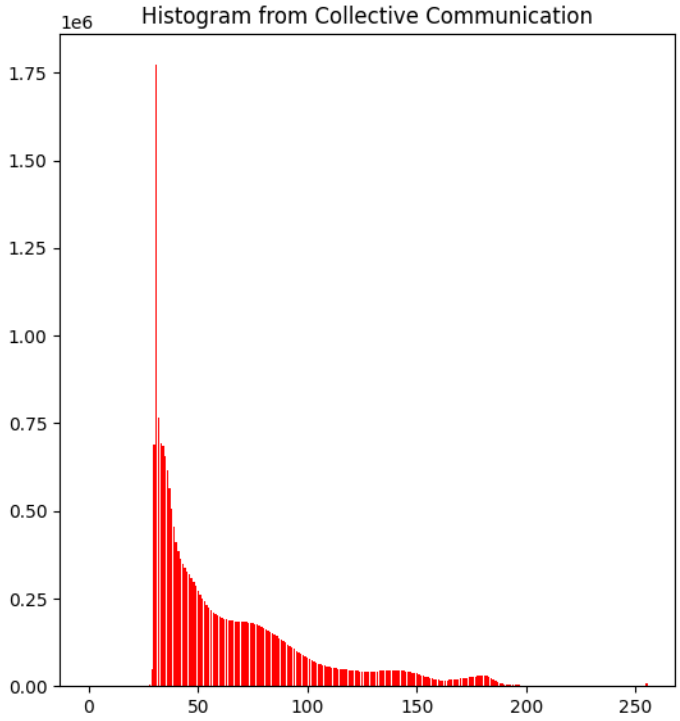
Image reference[1]

```
recv = comm.reduce(bucket)
# print('recv is ', recv)

if recv is not None:
    # print(len(recv))
    x =np.arange(256)
    # print('i am rank {} and the recvb is {}'.format(rank, recv))
    plt.bar(x, recv)
    plt.title('Histogram generated from Collective Communication Methods')
    plt.show()
```

11. Reduce returns values in recv, if it is not None, we have the total frequency counts for each intensity value from all ranks, cumulatively in recv. This can now be plotted using a bar chart which is the histogram of the image given. ‘recv’ are the values for y axis, for x axis, we create an array from 0 to 255 using np.arange.

Results


Original Histogram from OpenCV
1e6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
50
100
150
200
250


Histogram from Collective Communication
1e6
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
50
100
150
200
250

The Code

```
from collections import defaultdict
from operator import invert
from turtle import color
from typing import Counter
from mpi4py import MPI
import numpy as np
import cv2
from itertools import groupby
import matplotlib.pyplot as plt
st = MPI.Wtime()
bucket = np.zeros(256)
imageName = 'bigpic.png'
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
root = 0

def mycounter(lst): # This is a function to count the number of times each intensity appears in the image
    mydict = {} # This is a dictionary to store the number of times each intensity appears
    for i1 in lst: # This is a loop to iterate through the intensities
        mydict[i1] = 0 # This is to initialize the dictionary with 0
        for i2 in lst: # This is a loop to iterate through the intensities
            if i1 == i2: # This is to check if the intensity is the same
                mydict[i1] += 1 # This is to add 1 to the number of times the intensity appears
    return mydict # This is to return the dictionary


def original_histogram(imgName): # This is a function to create the histogram of the original image
    img = cv2.imread(imageName, cv2.IMREAD_GRAYSCALE) # This is to read the image
    dst = cv2.calcHist(img, [0], None, [256], [0,256])  # This is to create the histogram
    # #Grayscale histogram
    plt.figure(figsize=(10,6)) # This is to set the size of the figure
    plt.subplot(1, 2, 1) # This is to set the subplot
    get = plt.hist(img.ravel(), bins = 256) # This is to create the histogram
    plt.title('Original Histogram from OpenCV') # This is to set the title
    return img
```

```
if rank == 0:
    sdata = original_histogram(imageName) # This is to run the function to create the histogram of the ori
    print('shape of original image is', sdata.shape) # This is to print the shape of the original image
    sdata = np.array_split(sdata, size) # This is to split the image into chunks

else:
    sdata = None

data = comm.scatter(sdata, root=0) # This is to scatter the image chunks to the other processes
data = [l.tolist() for l in data] # This is to convert the image chunks to a list
# print('i am rank {} and i have {}'.format(rank, data))
flat_list = [item for sublist in data for item in sublist] # This is to flatten the list
# mycounts = Counter(flat_list) # This is to count the number of times each intensity appears in the image
mycounts = mycounter(flat_list) # This is to count the number of times each intensity appears in the image
# print(mycounts)


for i in range(len(bucket)): # This is to iterate through the intensities
    if i in mycounts: # This is to check if the intensity appears in the image
        bucket[i] = mycounts[i] # This is to add the number of times the intensity appears to the bucket
# print(bucket)

recv = comm.reduce(bucket) # This is to reduce the bucket to the root process
# print('recv is ', recv)

if recv is not None:
    # print(len(recv))
    x =np.arange(256) # This is to create the x-axis
    # print('i am rank {} and the recvb is {}'.format(rank, recv))
    plt.subplot(1, 2, 2) # This is to set the subplot
    plt.bar(x, recv, color = 'r')  # This is to create the histogram
    plt.title('Histogram from Collective Communication') # This is to set the title
    et = MPI.Wtime() # This is to get the end time
    print('running time is {} for size {}'.format(round(et-st, 5), size)) # This is to print the running t
    plt.show() # This is to show the histogram
```

Implementation of RGB Histogram

The idea here is similar to grayscale but the splitting of the image and tracking data for each channel is the interesting part here.

1. Initializations and imports. The bucket_arrays is an array of arrays, this will store the frequency of each intensity of each of the channels to be sent to comm.reduce to be added with that from the other processors.

```
from operator import invert
import re
from typing import Counter
from mpi4py import MPI
import numpy as np
import cv2
from itertools import groupby
import matplotlib.pyplot as plt
st = MPI.Wtime()
bucket = np.zeros(256)
imageName = 'bigpic.png'
# imageName = 'smallrgb.png'
comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()
root = 0
channels = 3 # This is to set the number of channels
# This is to create an array of arrays of zeros
bucket_arrays = np.array([np.zeros(256), np.zeros(256), np.zeros(256)])
num_of_intensities = 256 * channels
```

2. Defining mycounter() function and a function to read, plot and return the original image and its histogram to verify with my results.

```
def mycounter(lst): # This is a function to count the number of times each intensity appears in the image
    mydict = {} # This is a dictionary to store the number of times each intensity appears
    for i1 in lst: # This is a loop to iterate through the intensities
        mydict[i1] = 0 # This is to initialize the dictionary with 0
        for i2 in lst: # This is a loop to iterate through the intensities
            if i1 == i2: # This is to check if the intensity is the same
                mydict[i1] += 1 # This is to add 1 to the number of times the intensity appears
    return mydict # This is to return the dictionary

def original_histogram(imgName): # This is a function to create the histogram of the original image
    img = cv2.imread(imgName) # This is to read the image
    img = img[:3900, :5870, :] # This is to crop the image
    print('Restructured dimensions of original image are', img.shape) # This is to print the shape of the original image
    color = ('b','g','r') # This is to set the color channels
    plt.figure(figsize=(10,6)) # This is to set the size of the figure
    plt.subplot(1, 2, 1) # This is to set the subplot
    plt.title('Original Image Histogram') # This is to set the title
    for i,col in enumerate(color): # This is to iterate through the color channels
        histr = cv2.calcHist([img],[i],None,[256],[0,256]) # This is to create the histogram
        plt.plot(histr,color = col) # This is to plot the histogram
        plt.xlim([0,256]) # This is to set the x-axis limits
    # plt.show()
    return img
```

3. Now, the image is read and reshaped due to a limitation of this algorithm, explained later in this point. The image is split by rank 0 into each channel. Each channel of the image is extracted and stacked on top of eachother using vstack. To better understand, consider an image of 40x50. We stack each of the three channels and get an array of shape 120x50. Next, using hsplit, this ndarray is divided by columns in as many parts as the number of processors. This could raise errors if the number of columns are not divisible by the number of processors, which is why the image was cropped in the initial steps.

```
if rank == 0:
    img = original_histogram(imageName)
    img = img[:3900, :5870, :] # Restructuring the image to make it easier to split
    imgb = img[:,:,0] # This is to set the blue channel
    imgg = img[:,:,1] # This is to set the green channel
    imgr = img[:,:,2] # This is to set the red channel
    stacked = np.vstack((imgb, imgg, imgr)) # Stacking the image channels
    # splitCal = int(img.shape[1]/size)
    # print('splitCal is', splitCal)
    # since we are doing hsplit, 5 works for an image with 50 columns
    sdata = np.hsplit(stacked, size) # Splitting the image into chunks
    # splits = np.split(stacked, size)
    # comments for an image of shape 40x50x3
    # print('splits are of size', splits[0].shape)
    # rgb channels stacked, so 40 x 50 per each channel gives a stacked matrix of 120x50
    # after split by columns we get 5 matrices of 120x10 length, here 120 is contains 3 chunks each of size 40 for each channel
else:
    sdata = None
```

4. Now the split image data is sent to comm.scatter which distributes all chunks of the ndarray to all the processors. This data is received in the variable vsplit, which takes the data and splits it vertically by the number of channels. So, lets say 5 processors were running this program, so each rank had a data matrix of 120x10 and after vsplit, we get 3 ndarrays of shape 40x10, corresponding to each channel of the color image.

```
data = comm.scatter(sdata, root=0) # Scatter the data to all the processes
vsplit = np.vsplit(data,channels) # Splitting the data into channels
```

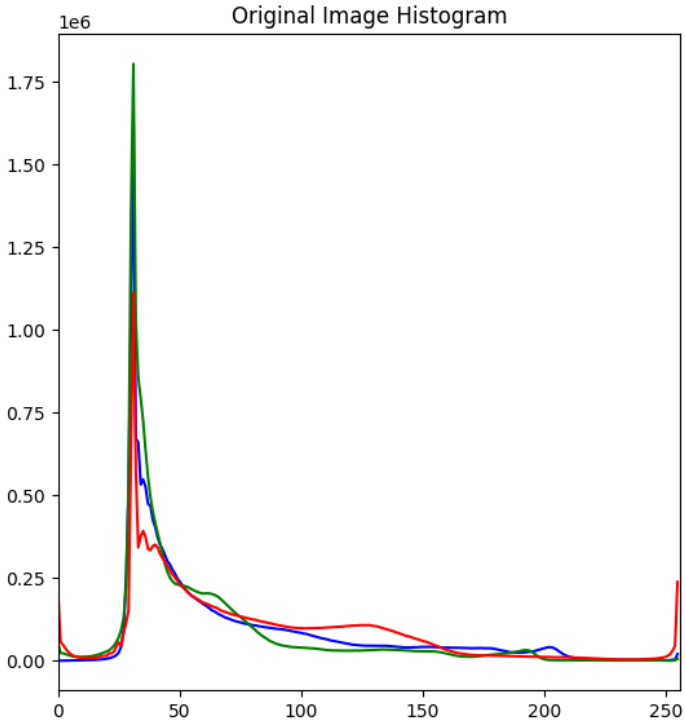
5. Moving further, we have information of each channel at each rank in the shape of a 40x10 ndarray per channel. A loop goes over each channel's array one by one, flattening the ndarray to get a flat list of intensity values in **flat_s2**. The **custom implemented function mycounter()** is used to count the number of occurrences of each intensity value in the **flat_s2** list then stored in a dictionary and returned to the variable **mycounts**.
6. Then in the nested loop, **bucket_arrays**'s **ith array** loops over each of the 256 elements, and takes values from mycounts dictionary. Hence, now **bucket_arrays** have the frequency count of each of the 256 intensity values at the **ith index** of the array. After the nested loops, the bucket_arrays is sent to comm.reduce which compiles bucket_arrays from all processes and adds the corresponding channel's frequency buckets. Eventually returning an array of length 3 containing 3 arrays of 256 elements corresponding to the count of each intensity value.

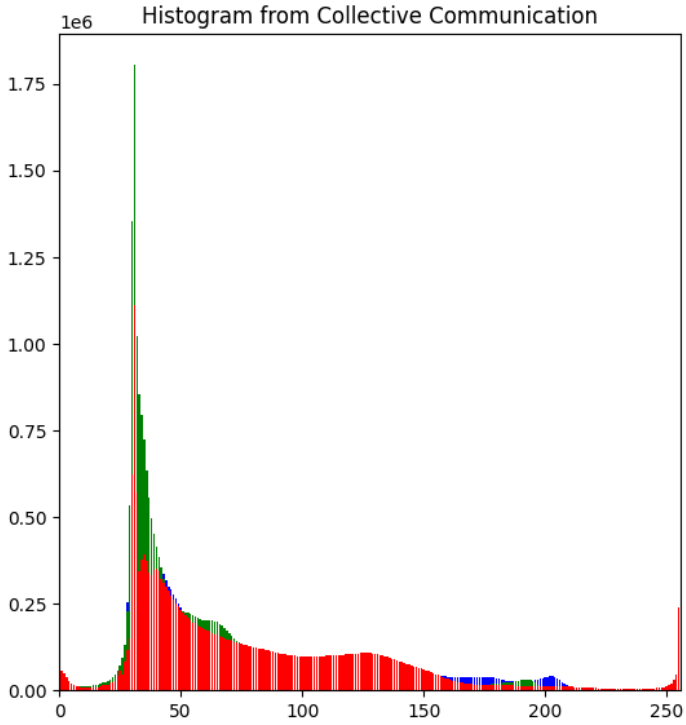
```
for chans in range(channels):  # Looping through the channels
    v = vsplit[chans] # Getting the channel
    s1 = [l.tolist() for l in v] #list of no arrays to list of lists
    flat_s2 = [item for sublist in s1 for item in sublist] #list of lists to list
    # mycounts = Counter(flat_s2) # Counting the number of times each intensity occurs
    mycounts = mycounter(flat_s2) # Counting the number of times each intensity occurs
    for i in range(len(bucket)): # Looping through the intensities
        if i in mycounts: # If the intensity is present in the list
            bucket_arrays[chans][i] = mycounts[i] # Assigning the count to the bucket

recv = comm.reduce(bucket_arrays) # Reduce the buckets to the root
```

7. In this step, the loop goes over each array for each channel and plots them and records the time.

```python
if recv is not None:
    x =np.arange(256)  # x axis
    colorlst = ['b','g','r'] # color list
    plt.subplot(1, 2, 2) # subplot
    plt.title('Histogram generated from Collective Communication Methods') # title
    for freq_arrays in range(channels): # Looping through the channels
        plt.bar(x, recv[freq_arrays], color = colorlst[freq_arrays]) # plotting the histogram
        plt.xlim([0,256]) # x axis limits
    et = MPI.Wtime() # End time
    print('running time is {} for size {}'.format(round(et-st, 5), size)) # print the time taken
    plt.show() # show the plot
```

Results





Question 2.6
Processing Times

Performance is calculated by commenting the **plt.show()** to prevent pop-ups, for the RGB histogram algorithm. Note: The custom implemented function **mycounter()** was taking too long for larger images to be calculated hence these performance measures were computed using **Counter()** function. My function **mycounter()** works fine for low resolution images but not good for high resolution images. This is primarily due to nested for loops, this could be solved using smarted approaches, probably a recursive implementation could be a good solution. I do understand the issue with my function and I will see to such issues in assignments in the future. Overall, the time is reducing as expected with the increase in processors.

Processes	Performance
1	`running time is 7.01903 for size 1`
2	`running time is 4.35078 for size 2`
3	Split error in hsplit as expected in the algorithm
4	Split error in hsplit as expected in the algorithm
5	`running time is 4.80404 for size 5`

Performance using custom implemented counter function mycounter()

For simplicity, I will use a low resolution image of shape 23x15. In this, the function works, and the time reduces from 1 processor to 2, but there is slight increment in 5 which I believe could be due to the overhead time taken by each process running nested for loops.

Image used for this:

Processes	Performance
1	`running time is 0.49152 for size 1`
2	Split error in hsplit as expected in the algorithm
3	`running time is 0.47158 for size 3`
4	Split error in hsplit as expected in the algorithm
5	`running time is 0.57071 for size 5`

References:
[1] https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/