Lab Course: Distributed Data Analytics Exercise Sheet 2

Excercise 1: Point to Point communication:

Naive way:

Short description of a naive way NsendAll: A naive way to send this array is using a for loop at worker 0 and sequentially send it to all other processes i.e. it will take P – 1 steps. Hint: Make sure all the workers exit the sendAll routine at the same time i.e its useful to use MPI Barrier at the end of this function.

Solution:

An array is to be sent as a whole to P-1 workers, where P is the number of processes.

As described in the question, a single loop that iterates over 1 to P-1 is executed, with in which the array is sent from process 0 to all other processes.

MPI_Barrier is used at the end of the function to make sure that every process has finished execution.

```
def NsendAll(array,time):

    if rank==0:

        for i in range(1,size):

            comm.Send([array,MPI.FLOAT],dest=i)

    else:
        data_buffer = np.empty((1,n),dtype=float)
        comm.Recv([data_buffer,MPI.FLOAT],source=0)

    comm.Barrier() #MAKES ALL TIME2 STRICTLY GREATER THAN ALL TIME1'S

comm.Barrier()
NsendAll(array,MPI.Wtime())
```

Steps:

1)In the function NsendAll(), the data is sent from process 0 to all the remaining processes.

2) A buffer is created at the receiving process and the receiving process uses this buffer to get the array.

3) Barrier() function is used at the end of the routine to make sure that all the process has completed execution before leaving the function.

Result:

Iterative send:

Processes/Input size	10**3	10**5	10**7
16	2.416 s	2.440 s	3.795 s
32	5.004 s	6.543 s	8.230 s

Inference:

For n = 16,

1) As the size of the array increases, the time taken to distribute the array also increases.

For n=32,

2) When the number of processes to which the root process has to distribute data increases (changing n from 16 to 32), the time taken to send data also increases with increasing array size (10**3, 10**5, 10**7) as follows: 5.004s, 6.543s, 8.230 s

Efficient way: (EsendAll)

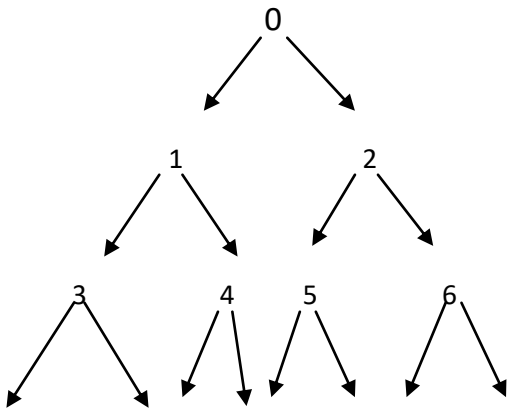
Short description of an efficient way: EsendAll Another possible ways is to use a recursive doubling algorithm, which will require log(P) steps. Suppose you have P workers, where P ≥ 2 d i.e. if P = 33 than d = 5 and rank is the current worker ID. Lets say the root worker has rank 0. The root worker sends to worker with Rank 1 and worker with Rank 2 only. All other workers first receive a message from recvProc, i.e. recvP roc = int((rank − 1)/2) and sends to two more processes destA = 2 × rank + 1 and destB = 2 × rank + 2. But before sending, make sure destA and destB exist.

Solution:

In this algorithm, a process receives data from int((rank − 1)/2) and sends data to

destA = 2 × rank + 1 and destB = 2 × rank + 2

The sending and receiving operation happens in the following fashion:



.....

.....

So this algorithm takes log(P) steps to distribute data to P processes.

Code Snippet:

```
def EsendAll(array):
    if rank==0:
        if(1<size):
            comm.Send([array,MPI.FLOAT],dest=1)
        if(2<size):
            comm.Send([array,MPI.FLOAT],dest=2)
    else:
        data_buffer = np.empty((1,n),dtype=float)
        comm.Recv([data_buffer,MPI.FLOAT],source=int((rank-1)/2))
        if(2*rank+1 < size):
            comm.Send([array,MPI.FLOAT],dest=(2*rank+1))
        if(2*rank+2 < size):
            comm.Send([array,MPI.FLOAT],dest=(2*rank+2))
    comm.Barrier() #MAKES ALL TIME2 STRICTLY GREATER THAN ALL TIME1'S
```

```
comm.Barrier()
EsendAll(array)
```

Explanation:

1) If the rank of the process is 0, it sends the array to processes 1 and 2 if they exist.
2) Process 1 and 2 sends data to 3 and 4 if they exists and the chain continues by process i sending data to (2*i)+1 and (2*i)+2 if they exist.
3) Each process receives data from int(rank-1/2)

Result:

Efficient send:

Processes/Input size	10**3	10**5	10**7
16	2.408 s	2.423 s	3.529 s
32	4.740 s	4.817 s	43.493 s

Inference:

For n = 16,

1) As the size of the array increases, the time taken to distribute the array also increases.

For n=32,

2) When the number of processes to which the root process has to distribute data increases (changing n from 16 to 32), the time taken to send data also increases with increasing array size (10**3, 10**5, 10**7) as follows: 4.740 s, 4.817 s, 43.493 s

Comparison between NsendAll, EsendAll:

Comparison 1:

Processes/Input size	Nsend-10**3	Esend - 10**3
16	2.416 s	2.408 s
32	5.004 s	4.740 s

Comparison 2:

Processes/Input size	Nsend-10**5	Esend - 10**5
16	2.440 s	2.423 s
32	6.543 s	4.817 s

Comparison 3:

Processes/Input size	Nsend-10**7	Esend - 10**7
16	3.795 s	3.529 s
32	8.230 s	43.493 s

Comparison 1 table compares naive send and efficient send for input of size 10**3.

It is observed that the time to run for Esend decreases compared to Nsend.

Comparison 2 table compares naive send and efficient send for input of size 10**5.

It is observed that the time to run for Esend decreases compared to Nsend.

Comparison 3 table compares naive send and efficient send for input of size 10**7.

It is observed that the time to run for Esend decreases compared to Nsend for 16 processes.

For 32 processes, time to run for Esend increases compared to Nsend for 16 processes.

This implies that Efficient send works comparatively well unless the overhead of the computation increases that the performance (in time) drastically increases.

Excercise 2: Collective communication

Image histogram:

The solution below explains calculating the image histogram for gray scale images.

1)

The image used is

The size in pixels of the above image is 4723 x 3134 pixels.

The image is read as a gray scale image using imread() function by the cv2 library.

2)

The number of processes is identified by the Get_size() function.
Numpy library's array_split() function is used to split the image array into the p partitions, where p is the number of processes.
The splitted array is sent to all the processes **using the collective communication routine scatter()**

3) Once each process receives its part of the image array, frequency of occurrence of each gray scale value is calculated and stored in a dictionary
    a. A dictionary is created with the key signifying the gray scale value and the value signifying the frequency of occurrence of the gray scale value

    b. When a gray scale value is not found in the dictionary from a., a new key value pair is added to the dictionary with the key being the gray scale value and the value being the frequency of occurrence of the gray scale value
    c. When a gray scale value is already found in the dictionary, then the value is incremented.
4) The frequency values computed in each process is gathered using **gather routine** and sent to process 0.
   Process 0 now combines frequency dictionaries from all the slave processes and combines a resultant dictionary that represents the frequency of occurrence of each gray scale value in the given image.

5) I have implemented for gray scale histogram.

6) **Run time analysis:**

Processes	Time to execute
1	20.040 s
2	11.838 s
3	9.794 s
4	8.691 s

Note: The above timings are for frequency calculations only.

Inference:

The time to execute the frequency calculation and displaying the image histogram seems to reduce with the increase in the number of processes . The **parallel execution of the frequency calulation** is the reason for the **speedup** that is observed when the number of processes increase.

Code snippets:

Reading the image and splitting the image array:

```
if(rank==0):

    img = cv2.imread('imggray.jpg',0)

    splitted_array = np.array_split(img, size)



else:
    splitted_array = None
```

Scattering the splitted array:

```
data = comm.scatter(splitted_array,root=0)
```

Frequency calculation and gathering results from each process:

```
if(rank!=0 or rank==0):


    for i in range(len(data)):

        for j in range(len(data[i])):
            if data[i][j] in frequency_dict:
                frequency_dict[data[i][j]]+=1
            else:
                frequency_dict[data[i][j]]=1



data_frequency = comm.gather(frequency_dict,root=0)
```

Combining frequencies from different processes.

```
if(rank==0):


    result_dict = data_frequency[0]
    for i in range(1,len(data_frequency)):
        for key in data_frequency[i]:
            if key in result_dict:
                result_dict[key] += data_frequency[i][key]
            else:
                result_dict[key] = data_frequency[i][key]
    print('final frequency', result_dict,'sda')
```

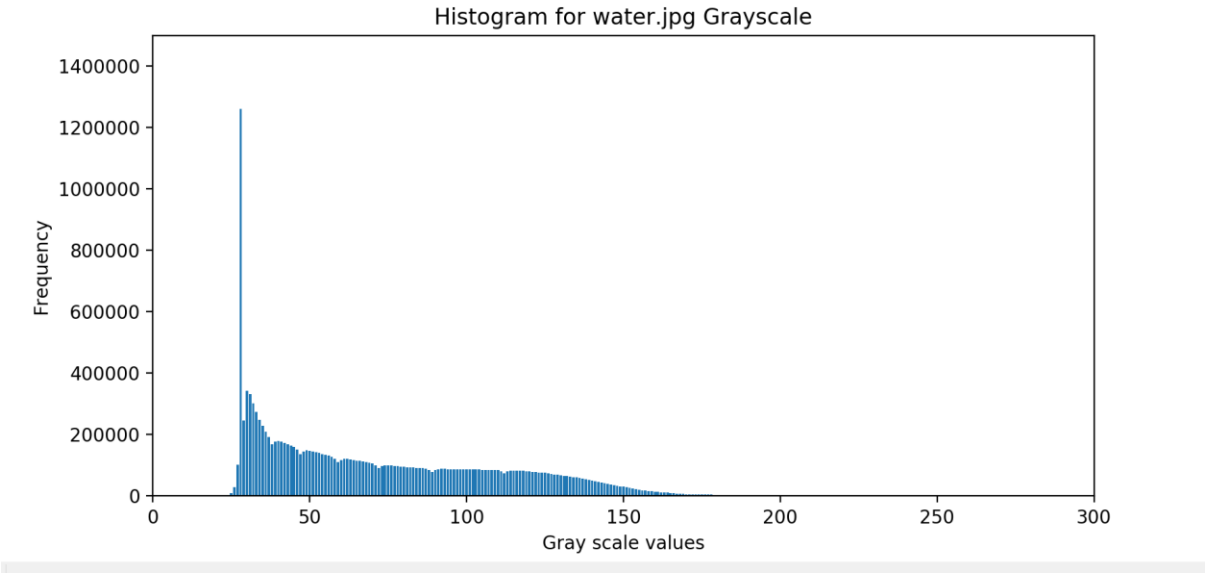
Displaying the image histogram:

```
plt.axis([0, 300, 0, 1500000])
plt.bar(result_dict.keys(), result_dict.values())
plt.(result_dict,256,[0,256])
plt.title('Histogram for water.jpg Grayscale')
plt.xlabel('Gray scale values')
plt.ylabel('Frequency')
plt.show()
```

Results:

```
C:\Users\admin\Anaconda3\Scripts>ptime mpiexec -n 4 python imagepy.py

ptime 1.0 for Win32, Freeware - http://www.pc-tools.net/
Copyright(C) 2002, Jem Berkes <jberkes@pc-tools.net>

===  mpiexec -n 4 python imagepy.py ===
final frequency {28: 1261696, 30: 342252, 29: 245300, 27: 101546, 31: 330855, 32: 300861, 33: 274154,
163919, 38: 168045, 40: 179835, 26: 29541, 25: 8919, 39: 177458, 43: 168362, 50: 146744, 54: 136684, 6
72656, 23: 1126, 24: 3078, 73: 96928, 84: 91281, 90: 83945, 89: 78015, 87: 88336, 80: 94737, 70: 10523
 98447, 68: 109818, 78: 97257, 96: 87137, 101: 86770, 102: 85746, 92: 87899, 66: 113810, 81: 93641, 11
69: 108518, 67: 112163, 79: 96387, 74: 100470, 86: 90021, 65: 115200, 106: 84869, 111: 80200, 105: 852
5493, 108: 84690, 109: 84708, 95: 87793, 104: 86101, 113: 79906, 115: 82785, 116: 83017, 114: 82803, 1
45, 127: 71987, 21: 186, 130: 67614, 124: 75784, 126: 73371, 134: 61041, 132: 64604, 128: 70314, 131:
39: 52210, 143: 43455, 145: 39305, 142: 45939, 146: 37205, 147: 35033, 144: 41256, 150: 29788, 152: 26
 149: 31585, 161: 12962, 159: 15227, 19: 29, 157: 17414, 160: 13785, 163: 11382, 165: 9291, 168: 7160,
0: 3551, 177: 4476, 173: 5161, 174: 4839, 178: 4082, 175: 4633, 182: 3767, 183: 3409, 176: 4394, 17: 1
93: 528, 194: 424, 191: 1940, 201: 240, 202: 251, 190: 2306, 189: 2417, 192: 772, 198: 311, 197: 326,
212: 101, 211: 140, 214: 47, 213: 75, 216: 32, 215: 45, 218: 9, 217: 16, 219: 2, 220: 2, 16: 1} sda

Execution time: 8.691 s
```

```
C:\Users\admin\Anaconda3\Scripts>ptime mpiexec -n 4 python imagepy.py

ptime 1.0 for Win32, Freeware - http://www.pc-tools.net/
Copyright(C) 2002, Jem Berkes <jberkes@pc-tools.net>

===  mpiexec -n 4 python imagepy.py ===
final frequency {28: 1261696, 30: 342252, 29: 245300, 27: 101546, 31: 330855, 32: 300861, 33: 274154, 34: 249000, 35: 227986, 36: 209814, 37: 192009, 41: 176149, 45: 160230, 48: 145166, 46: 150283, 44:
163919, 38: 168045, 40: 179835, 26: 29541, 25: 8919, 39: 177458, 43: 168362, 50: 146744, 54: 136684, 60: 116972, 59: 109946, 58: 121761, 55: 134090, 56: 131848, 51: 144897, 49: 148904, 47: 135727, 42: 1
72656, 23: 1126, 24: 3078, 73: 96928, 84: 91281, 90: 83945, 89: 78015, 87: 88336, 80: 94737, 70: 105234, 61: 120865, 64: 117252, 62: 120118, 57: 127990, 52: 141682, 53: 140044, 72: 91534, 75: 99372, 77:
 98447, 68: 109818, 78: 97257, 96: 87137, 101: 86770, 102: 85746, 92: 87899, 66: 113810, 81: 93641, 117: 81823, 103: 86554, 99: 87150, 82: 93008, 63: 118647, 71: 99299, 91: 87032, 85: 90210, 83: 92264,
69: 108518, 67: 112163, 79: 96387, 74: 100470, 86: 90021, 65: 115200, 106: 84869, 111: 80200, 105: 85291, 76: 98583, 97: 87431, 93: 88018, 88: 84026, 100: 87074, 94: 87736, 98: 87537, 118: 81859, 107: 8
5493, 108: 84690, 109: 84708, 95: 87793, 104: 86101, 113: 79906, 115: 82785, 116: 83017, 114: 82803, 121: 78956, 123: 76758, 120: 79683, 122: 78152, 22: 382, 119: 80714, 125: 74985, 110: 83739, 112: 747
45, 127: 71987, 21: 186, 130: 67614, 124: 75784, 126: 73371, 134: 61041, 132: 64604, 128: 70314, 131: 65731, 135: 59909, 133: 62885, 129: 68990, 20: 62, 138: 53905, 137: 56558, 140: 50062, 136: 57960, 1
39: 52210, 143: 43455, 145: 39305, 142: 45939, 146: 37205, 147: 35033, 144: 41256, 150: 29788, 152: 26360, 141: 47773, 148: 33423, 154: 22641, 155: 20962, 153: 24555, 151: 28216, 156: 18971, 158: 16135,
 149: 31585, 161: 12962, 159: 15227, 19: 29, 157: 17414, 160: 13785, 163: 11382, 165: 9291, 168: 7160, 166: 8555, 162: 12254, 164: 10507, 18: 8, 170: 6090, 169: 6527, 167: 7805, 172: 5357, 171: 5730, 18
0: 3551, 177: 4476, 173: 5161, 174: 4839, 178: 4082, 175: 4633, 182: 3767, 183: 3409, 176: 4394, 17: 1, 184: 3493, 179: 3865, 181: 3621, 187: 2598, 186: 2564, 188: 2718, 185: 3406, 195: 345, 196: 330, 1
93: 528, 194: 424, 191: 1940, 201: 240, 202: 251, 190: 2306, 189: 2417, 192: 772, 198: 311, 197: 326, 199: 275, 200: 279, 203: 245, 204: 252, 206: 255, 205: 237, 208: 204, 210: 172, 209: 174, 207: 242,
212: 101, 211: 140, 214: 47, 213: 75, 216: 32, 215: 45, 218: 9, 217: 16, 219: 2, 220: 2, 16: 1} sda

Execution time: 10.205 s
```



RGB scale,