DDA Lab Ex - 2

Sreyashi Saha Matriculation No: 1747271

Exercise 1:

In this exercise I wrote a parallel program using point-to-point communication routines. There is a worker with rank 0 and I have initialised an integer array and it sends to all P – 1 workers in COMM WORLD, and I have named this routine **sendAll**.

```
def Nsendall(array):
    for i in range(1,number_of_processes):
        print(i)
        comm.send(array, dest=i,tag=i)
        print('Process {} sent data:'.format(rank),array)
    comm.Barrier()
```

The 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.

In rank 0 after initializing I have called my function and passed my array as input for the function. Inside the function I called a loop which runs till the number of processes of my system and sends each process the array. I have then used the barrier command so that there is a synchronized operation among the processes belonging to the given communicator. That is, all the processes from a given communicator will wait within the MPI_Barrier until all of them are inside, and at that point, they will leave the operation.

In the else part that is when my rank is not equal to zero I receive the data from each of the processes and print from which process I'm receiving the data.

I have also calculated the time required for each rank to send and receive the data.

Processors	10^3	10^5	10^7
16	Rank: 11 Time: 0.229996 Rank: 15 Time: 0.38070400000000004 Rank: 6 Time: 0.38712899999999995 Rank: 13 Time: 0.47388399999999997 Rank: 3 Time: 0.290516 Rank: 9 Time: 0.35608399999999996 Rank: 10 Time: 0.39372300000000005 Rank: 12 Time: 0.374957 Rank: 7 Time: 0.291203 Rank: 5 Time: 0.289149 Rank: 2 Time: 0.27864900000000004 Rank: 1 Time: 0.230849 Rank: 14 Time: 0.283606 Rank: 8 Time: 0.335947 Rank: 0 Time: 0.302223 Rank: 4 Time: 0.161813	Rank: 13 Time: 0.199402 Rank: 14 Time: 0.34101099999999995 Rank: 15 Time: 0.34160999999999997 Rank: 7 Time: 0.294102 Rank: 11 Time: 0.34177900000000005 Rank: 3 Time: 0.289737 Rank: 6 Time: 0.42524399999999996 Rank: 12 Time: 0.42429399999999995 Rank: 5 Time: 0.325203 Rank: 9 Time: 0.29789600000000005 Rank: 10 Time: 0.43483099999999997 Rank: 2 Time: 0.43753200000000003 Rank: 4 Time: 0.36534 Rank: 8 Time: 0.251094 Rank: 1 Time: 0.262588 Rank: 0 Time: 0.38041100000000005	Rank: 14 Time: 1.870312 Rank: 9 Time: 1.882478 Rank: 11 Time: 1.863094 Rank: 12 Time: 1.881907 Rank: 15 Time: 1.915812 Rank: 7 Time: 1.711377 Rank: 13 Time: 1.792682 Rank: 4 Time: 1.930523 Rank: 5 Time: 1.800806 Rank: 6 Time: 1.875012 Rank: 10 Time: 1.970938 Rank: 0 Time: 1.866477 Rank: 1 Time: 1.916077 Rank: 2 Time: 1.752128 Rank: 3 Time: 1.792017 Rank: 8 Time: 1.8186849999999999

Processors	10^3	10^5	10^7
32	R:25 Time:0.7005140000000001 R:13 Time:0.82471 R:5 Time:0.749557 R:17 Time:0.556735 R:19 Time:0.582302 R:11 Time:0.534199 R:7 Time:0.53506 R:9 Time:0.505099 R:3 Time:0.683189 R:30 Time:0.287793 R:27 Time:0.391681 R:29 Time:0.273852 R:15 Time:0.284363 R:21 Time:0.344463 R:26 Time:0.259708 R:10 Time:0.38739 R:14 Time:0.48394 R:22 Time:0.467735 R:31 Time:0.683013 R:18 Time:0.439177 R:23 Time:0.508266 R:1 Time:0.540564 R:28 Time:0.48497 R:6 Time:0.44951 R:2 Time:0.7762910000000001 R:12 Time:0.6574599999999999 R:24 Time:0.578008 R:16 Time:0.296922 R:20 Time:0.362099 R:8 Time:0.674732 R:0 Time:0.28394 R:4 Time:0.381776	R:7 Time:0.405533 R:15 Time:0.623022 R:23 Time:0.359059 R:11 Time:0.337992 R:25 Time:0.431048 R:14 Time:0.457527 R:19 Time:0.57141 R:22 Time:0.51609 R:13 Time:0.476232 R:6 Time:0.399417 R:9 Time:0.616324 R:18 Time:0.652146 R:21 Time:0.586141 R:26 Time:0.61401 R:27 Time:0.487203 R:10 Time:0.367185 R:2 Time:0.567417 R:5 Time:0.471123 R:17 Time:0.471608 R:28 Time:0.426764 R:12 Time:0.919779 R:30 Time:0.43206 R:1 Time:0.703522 R:20 Time:0.609 R:4 Time:0.569429 R:24 Time:0.448678 R:31 Time:0.361245 R:8 Time:0.431855 R:29 Time:0.343515 R:16 Time:0.7028449999999999 R:0 Time:0.571263 R:3 Time:0.364292	R:27 Time:6.076389 R:31 Time:6.050466 R:22 Time:6.232684 R:25 Time:6.274221 R:29 Time:6.025327 R:30 Time:6.031 R:11 Time:6.100799 R:23 Time:6.290098 R:26 Time:6.307545 R:28 Time:5.983463 R:2 Time:6.337359 R:6 Time:6.114413 R:7 Time:6.322382 R:13 Time:6.299178 R:14 Time:6.183791 R:18 Time:6.361982 R:19 Time:6.442932 R:20 Time:6.181699 R:15 Time:6.158728 R:1 Time:6.001611 R:9 Time:6.12905 R:8 Time:6.225403 R:12 Time:6.256637 R:4 Time:6.201655 R:10 Time:6.247921 R:16 Time:6.134071 R:24 Time:6.230276 R:3 Time:6.25654 R:17 Time:6.29788 R:21 Time:6.00558 R:5 Time:6.218579 R:0 Time:6.430632

We can see that for Nsendall, as we increase the number of processes the time taken to send and receive also increases.

Short description of an efficient way: EsendAll

Another possible ways is to use a recursive doubling algorithm, which requires log(P)

steps. Here I have P workers, where $P \geq 2^d$ i.e. if P = 33 than d = 5 and rank is the current worker Rank. The root worker with rank 0 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, I make sure destA and destB exist.

Here my approach is almost the same as the sandal function. Here I have just initialised the array in the root node. In the else part I just receive the data using recv.comm() and print the array. Here also I have calculated the time required for each of the processes to carry out the operation.

```
def Esendall(data):
    destA = 2*rank + 1
    destB = 2*rank + 2
    if destA<=size:
        comm.send(data,dest =destA)
        # print('Process {} sent data:'.format(rank),data)
    if destB<=size:
        comm.send(data,dest = destB)
        # print('Process {} sent data:'.format(rank),data)
```

Processors	10^3	10^5	10^7
16	R:6 Time:0.277842 R:12 Time:0.085373 R:14 Time:0.145202 R:7 Time:0.196362 R:2 Time:0.267193 R:11 Time:0.12776 R:4 Time:0.143862 R:8 Time:0.284263 R:15 Time:0.126985 R:10 Time:0.095042 R:5 Time:0.097443 R:1 Time:0.286563 R:3 Time:0.208203 R:0 Time:0.138751 R:9 Time:0.138417 R:13 Time:0.106972	R:6 Time:0.28798500000000005 R:14 Time:0.293393 R:7 Time:0.38803699999999997 R:13 Time:0.30035999999999996 R:2 Time:0.30809 R:3 Time:0.25895500000000005 R:9 Time:0.31048600000000004 R:15 Time:0.311897 R:11 Time:0.32282299999999997 R:5 Time:0.39920500000000003 R:4 Time:0.249606 R:12 Time:0.35701700000000003 R:1 Time:0.40895899999999996 R:10 Time:0.36235399999999995 R:0 Time:0.334954 R:8 Time:0.45115700000000003	R:14 Time:1.088759 R:4 Time:1.015261 R:6 Time:1.191222 R:10 Time:1.172788 R:11 Time:1.110385 R:12 Time:1.200357 R:15 Time:1.0048 R:2 Time:1.134806 R:5 Time:1.136205 R:7 Time:1.079733 R:8 Time:1.068741 R:9 Time:1.029787 R:13 Time:1.203281 R:1 Time:1.142285 R:3 Time:1.193943 R:0 Time:1.084795

Processors	10^3	10^5	10^7
32	R:26 Time:0.0998609 R:12 Time:0.417682 R:14 Time:0.409845 R:2 Time:0.3650640 R:11 Time:0.4074649 R:10 Time:0.402884 R:4 Time:0.322505 R:3 Time:0.271869 R:6 Time:0.185357 R:30 Time:0.10345099 R:9 Time:0.2401360 R:27 Time:0.1635760 R:19 Time:0.19532400 R:25 Time:0.205426 R:15 Time:0.68924000 R:8 Time:0.505356 R:18 Time:0.465789 R:0 Time:0.1782850 R:20 Time:0.4013210 R:7 Time:0.273204 R:22 Time:0.373853 R:13 Time:0.2689209 R:23 Time:0.2384849 R:1 Time:0.335296 R:16 Time:0.2022850 R:17 Time:0.202646 R:5 Time:0.40628 R:21 Time:0.238761 R:24 Time:0.294508 R:31 Time:0.182867 R:28 Time:0.1223579 R:29 Time:0.10910299	R:30 Time:2.683199 R:22 Time:2.812229 R:26 Time:2.671246 R:14 Time:3.057892 R:28 Time:2.812983 R:31 Time:2.720835 R:10 Time:2.676718 R:6 Time:2.824172 R:8 Time:3.008462 R:11 Time:3.158399 R:18 Time:2.589385 R:23 Time:2.810754 R:24 Time:2.638809 R:3 Time:3.181581 R:1 Time:2.800816 R:12 Time:2.936476 R:15 Time:2.762935 R:16 Time:2.900605 R:19 Time:3.048075 R:25 Time:2.90753 R:2 Time:2.87471 R:20 Time:2.875426 R:21 Time:2.949144 R:27 Time:2.754124 R:7 Time:2.745033 R:9 Time:2.780462 R:17 Time:2.897632 R:13 Time:2.6477399999999998 R:29 Time:2.768659 R:0 Time:3.012805 R:5 Time:2.814671 R:4 Time:2.715172	R:30 Time:2.683199 R:22 Time:2.812229 R:26 Time:2.671246 R:14 Time:3.057892 R:28 Time:2.812983 R:31 Time:2.720835 R:10 Time:2.676718 R:6 Time:2.824172 R:8 Time:3.008462 R:11 Time:3.158399 R:18 Time:2.589385 R:23 Time:2.810754 R:24 Time:2.638809 R:3 Time:3.181581 R:1 Time:2.800816 R:12 Time:2.936476 R:15 Time:2.762935 R:16 Time:2.900605 R:19 Time:3.048075 R:25 Time:2.90753 R:2 Time:2.87471 R:20 Time:2.875426 R:21 Time:2.949144 R:27 Time:2.754124 R:7 Time:2.745033 R:9 Time:2.780462 R:17 Time:2.897632 R:13 Time:2.647739999999998 R:29 Time:2.768659 R:0 Time:3.012805 R:5 Time:2.814671 R:4 Time:2.715172

We can see that for Esendall, as we increase the number of processes the time taken to send and receive also increases. However, if I compare it to Nsendall, it takes much lesser time to complete the process.

Exercise 2:

In this exercise I have to find an Image histogram. Images generally have RGB or gray scale values. I'll plot a histogram by just calculating the frequency of occurrence of each

gray scale or RGB value. I have used parallel implementation using collective communication.

I have considered the following grayscale image for my plotting a histogram:

I have used OpenCV to read my image and check if the histogram is being plotted or not. Then I have read the image in the root node itself and split it to send to the rest of my processes. I have used comm.scatter() function which sends my chunks of matrices obtained from the image and sent it the remaining processes. I have then appended all the values obtained from scattering the chunks of matrices to a list named **elements** for frequency calculation. Next to this I have calculated the frequencies of each value(i.e. usually from 0 to 255 for a grayscale image) by using a dictionary which stores the frequencies corresponding to the key value. The code for the same is given below.

```
freq = {}
for item in elements:
    if (item in freq):
        freq[item] += 1
    else:
        freq[item] = 1
```

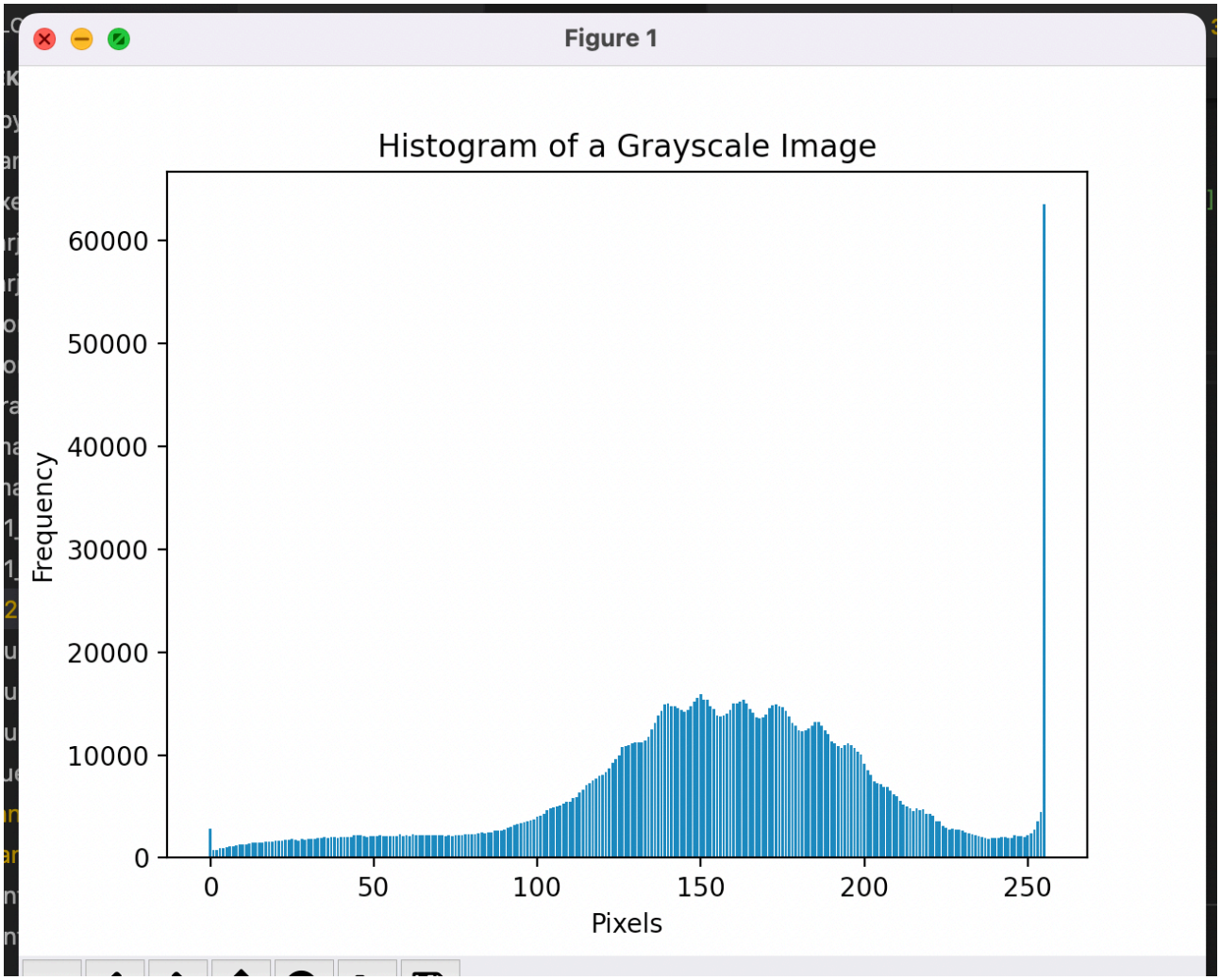
Next I have gathered all the dictionaries used to calculate frequencies from my other processes. Then in the root node I have summed the values with same keys in the dictionaries obtained by gather function. I found the code for the same in GitHub

```
if rank == 0:
```

```
    # sum the values with same keys
    res = dict(functools.reduce(operator.add,
                                map(collections.Counter,data)))
```

Using the above function I actually obtained a single dictionary with all the frequencies summed up for same key values. After that, I have plotted my histogram by considering the key values of the dictionary as points of my x-axis and the frequencies corresponding to each key value as points on my y-axis.

The Histogram that I obtained after plotting:



I have also calculated the time to see how long it takes to calculate the frequencies and obtain a histogram for an image. I have calculated the same for processes = 1,2,3,4 and I have observed that as I increase my number of processes I can see that the work is divided among the processes and it takes comparatively lesser time. The output of the same can be seen below:

```
sreyashisaha@Sreyashis-MacBook-Air Week3 % mpiexec -n 1 python3 Q2.py
 Rank: 0 Time taken to calculate the histogram: 0.51954
sreyashisaha@Sreyashis-MacBook-Air Week3 % mpiexec -n 2 python3 Q2.py
 Rank: 0 Time taken to calculate the histogram: 0.259151
sreyashisaha@Sreyashis-MacBook-Air Week3 % mpiexec -n 3 python3 Q2.py
 Rank: 0 Time taken to calculate the histogram: 0.216502
sreyashisaha@Sreyashis-MacBook-Air Week3 % mpiexec -n 4 python3 Q2.py
 Rank: 0 Time taken to calculate the histogram: 0.224377
sreyashisaha@Sreyashis-MacBook-Air Week3 % mpiexec -n 5 python3 Q2.py
 Rank: 0 Time taken to calculate the histogram: 0.199154
```

RGB :

For the RGB image I have implemented the same procedure and got a similar graph. But the frequencies are very small for some.