3116 – Lab Distributed Data Analytics – Group 2

Exercise Sheet 2

Yuvaraj Prem Kumar

303384, premyu@uni-hildsheim.de

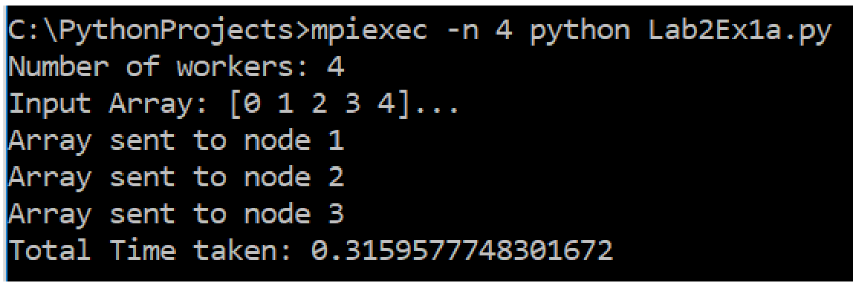
Exercise 1: Point to Point Communication

Part(a)

This is a naïve method of sending the data array. The same coding algorithm from Exercise 1 is used again here. The basic steps are as follows:

1. Function "*NSendAll*" is called to execute the data sending via a single For loop.
2. The data is sent from root node 0, to the (P-1) worker nodes in a sequential manner
3. Each worker will exit the function, due to the use of *MPI_Barrier()* function.

The below screenshot shows the result for sample data array of [10^7] and P = {4}. The input data is just a numpy array of [0,1,2,3..] etc. The full results are in Table 1.



```
C:\PythonProjects>mpiexec -n 4 python Lab2Ex1a.py
Number of workers: 4
Input Array: [0 1 2 3 4]...
Array sent to node 1
Array sent to node 2
Array sent to node 3
Total Time taken: 0.3159577748301672
```

Part(b)

In this part, the efficient solution is implemented as a binary tree traversal. Root node 0 will send the data to Nodes 1 and 2, and Node 1 will send to Node 3 and 4. Similarly for Node 2 sending to Nodes 5 and 6.

The diagram below shows the traversal layout of the P worker nodes.

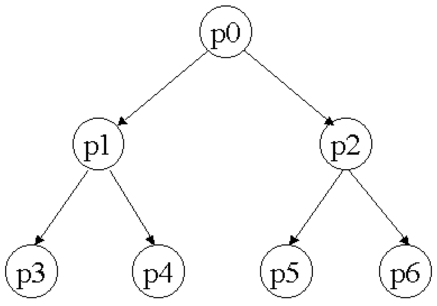
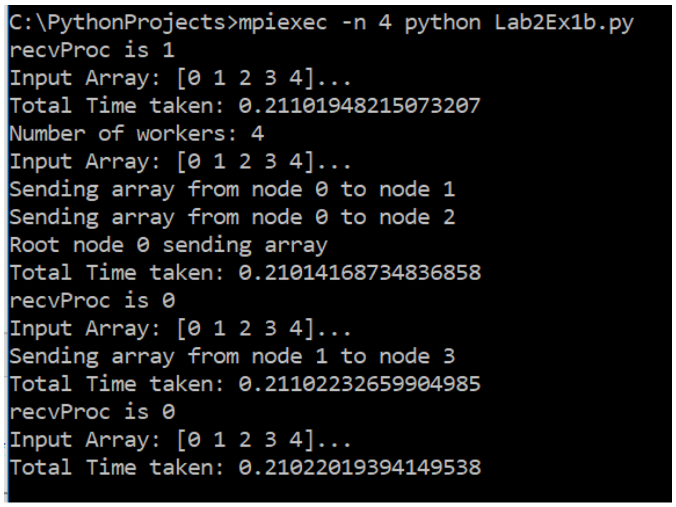


Figure 1: Binary tree message passing[1]

The basic steps are outlined as follows:

1. Function *ESendAll* is defined, based on the log(P) steps. For each node (including root node), there exists two possible destinations:
   a. *DestA* = $2 \times Rank + 1$
   b. *DestB* = $2 \times Rank + 2$
2. If *DestA* or *DestB* exists, data will be sent to either one of these two destinations. A regular "if" statement is used here instead of a "try…except", Although this is against the pythonic principle of EAFP, the overhead cost is lower for the "if" statement, since we can reliably assume the condition instead of waiting to catch an exception.
3. For nodes (other than root); they will receive data from a node defined as: $int(\frac{Rank-1}{2})$

A sample program is run with the same parameters as in part(a):



```
C:\PythonProjects>mpiexec -n 4 python Lab2Ex1b.py
recvProc is 1
Input Array: [0 1 2 3 4]...
Total Time taken: 0.21101948215073207
Number of workers: 4
Input Array: [0 1 2 3 4]...
Sending array from node 0 to node 1
Sending array from node 0 to node 2
Root node 0 sending array
Total Time taken: 0.21014168734836858
recvProc is 0
Input Array: [0 1 2 3 4]...
Sending array from node 1 to node 3
Total Time taken: 0.21102232659904985
recvProc is 0
Input Array: [0 1 2 3 4]...
Total Time taken: 0.21022019394149538
```

The full experiment is conducted with the following parameters. Table 1 shows the result set from part(a) and part(b).

$$P = \{4,16,32\}$$

$$N = [10^3, 10^5, 10^7]$$

Method	Array size N	# of Workers	Avg. processing time (s)
NSendAll	10^3	4	0.00176128
	10^3	16	0.09423466
	10^3	32	0.16299677
	10^5	4	0.00428999
	10^5	16	0.12119485
	10^5	32	0.25187307
	10^7	4	0.43489995
	10^7	16	1.07308566
	10^7	32	2.24445817
ESendAll	10^3	4	0.00220842
	10^3	16	0.15720264
	10^3	32	0.45048168
	10^5	4	0.00531512
	10^5	16	0.15507215
	10^5	32	0.33720803
	10^7	4	0.18228379
	10^7	16	0.66567566
	10^7	32	1.39620460

Table 1: Results from the two functions

We can see that *"ESendAll"* method performs better compared to the sequential *"NSendAll"* method. The difference for a smaller 10^3 array is not so obvious, but the somewhat larger difference and be seen for larger arrays 10^5 and 10^7. There is a limitation to my machine's performance. However, in a proper distributed parallel execution environment, the result should show a greater difference and result in a faster execution time for the *"ESendAll"* method. Both part(a) and (b) employ the use of comm.Barrier(). Barrier is a collective operation used to synchronize a group of nodes. It guarantees that by the end of the operation, the remaining nodes have at least entered the barrier.

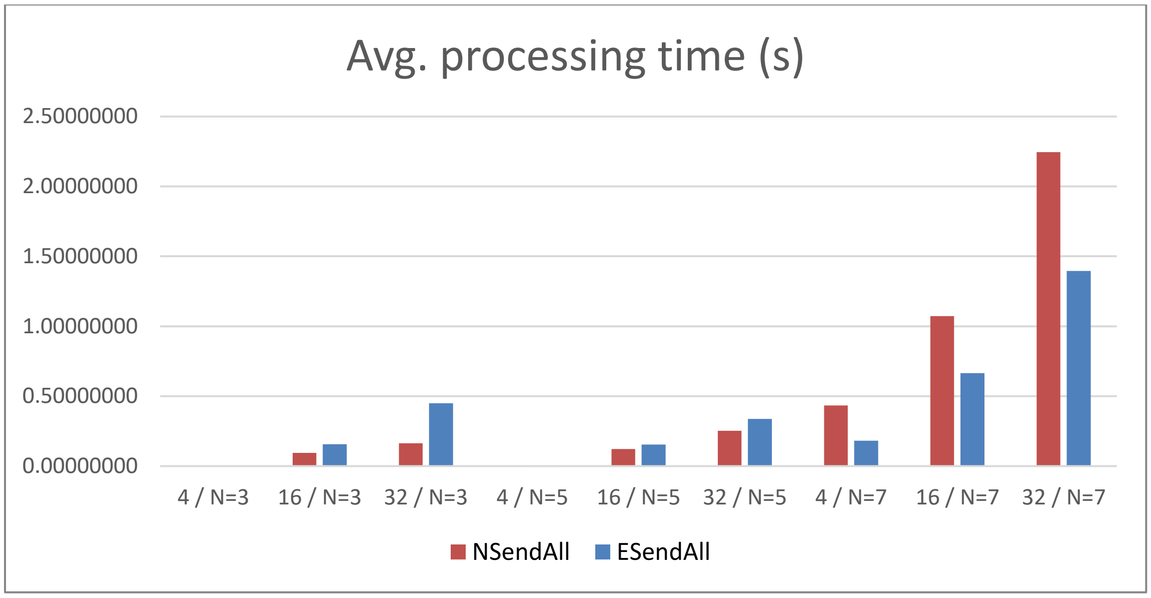


Figure 2: Histogram of average processing times

A plot of the histogram shows a clearer view of the trend. For low values of N=3, the EsendAll function is actually slower compared to NSendAll, due to the overhead computation cost involved. However, it performs far better for a large array of size N=7 as expected. This represents an almost linear speedup, in terms of the number of workers against computation time.

Exercise 2: Collective Communication

The approach used here is the same as in Lab 1, whereby a matrix array of size *NxN* is split into slices by *P* workers. Except in this exercise, collective communication is used.

1. The chosen image is that of the Malaysian capital of Kuala Lumpur. The image dimensions are 3840 (width) × 2160 (height). The image is read as greyscale only – my personal choice due to time constraints in implementing an RGB solution.

2. The image is imported using *OpenCV*, and split into arrays using *numpy.array_split* based on the number of workers.
3. The split arrays are distributed evenly to worker nodes using *MPI_Scatter*. *MPI_Scatter* takes an array of elements and distributes the elements in the order of process rank.

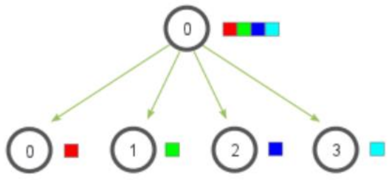


Figure 4: MPI_Scatter workflow

4. The worker nodes receive the split arrays, and calculate the sum of each attribute values in the matrix according to the following algorithm:

```
$for\ i\ in\ range(0, len(array))$:
    $for\ j\ in\ range(0, len(array[i]))$:
        $pixels[array[i][j]] += 1$
```

5. The computed data is passed back to the root node via *MPI_Reduce*. This function is passed the parameter *"MPI_SUM"* as the main point here is to sum up the results from each worker node to get the final intensity matrix.

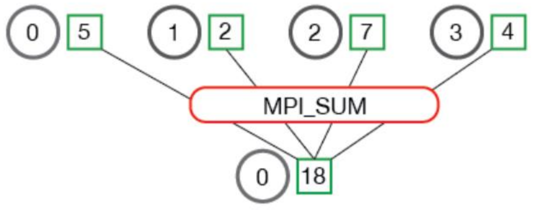


Figure 5: MPI_Reduce

6. A histogram of the greyscale intensity matrix is plotted.

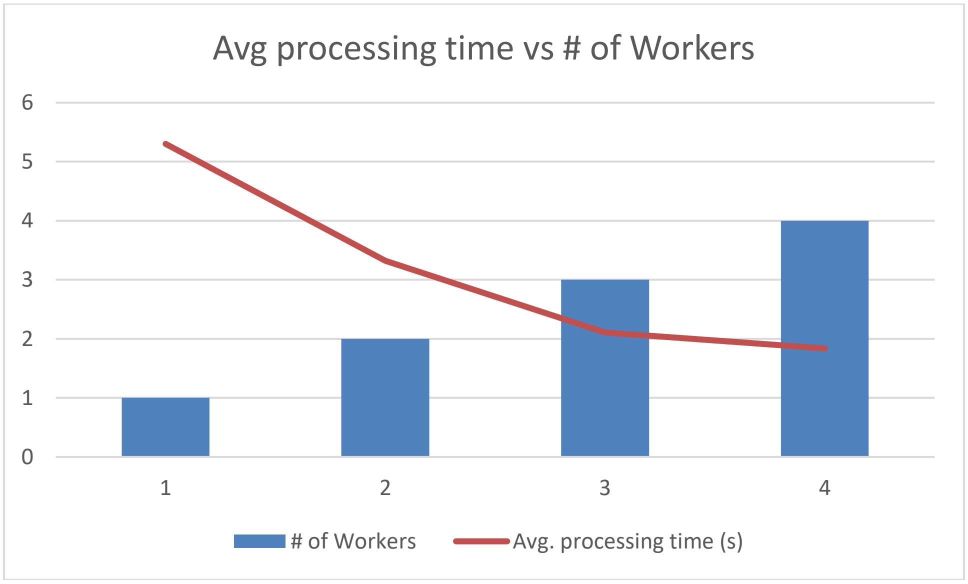
Beside the above, the histogram is calculated using the in-built Python libraries of OpenCV and Matplotlib Hist() function. The idea here is to compare the execution time of the manual frequency calculation versus the state-of-the-art libraries. The plots of the histograms are also compared to demonstrate the accuracy of the manual calculation.

The table below shows the execution times:

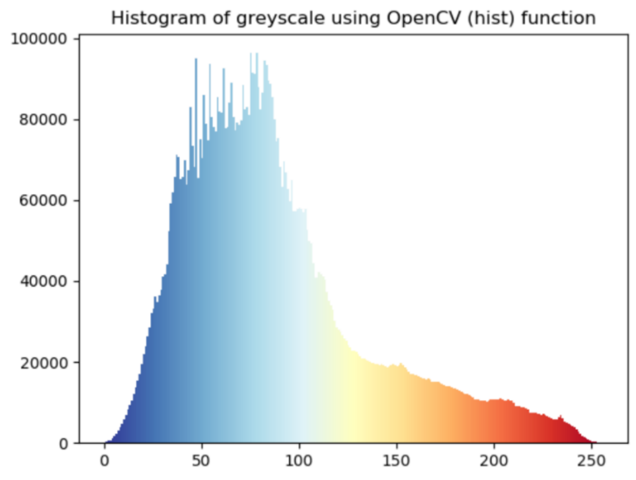
Method	# of Workers	Avg. processing time (s)
MPI	1	5.30255909
	2	3.32299757
	3	2.10746087
	4	1.83829329
State-of-the-art	N/A	Almost instantaneous

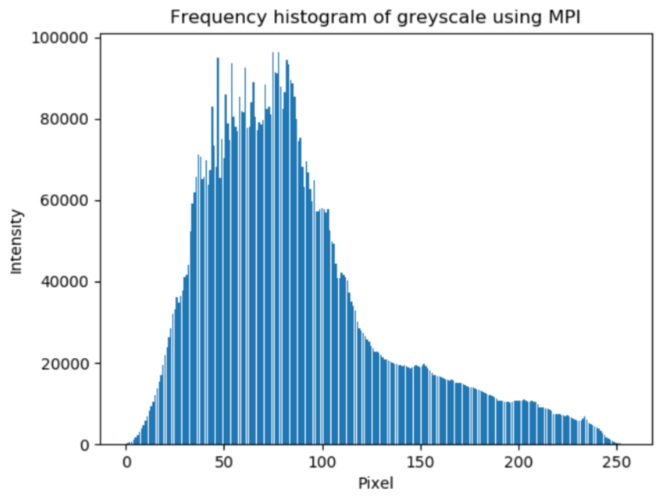
Table 2: Histogram calculation time

As expected, the state-of-the-art implementation far outperforms any manual calculation. Python is a scripting language, instead of an optimized programming language unlike Java/C++. Computing multiple ‘for’ loops occurs large resource overhead in python. Hence its main usefulness is the wide variety of libraries such OpenCV and Numpy. For the MPI implementation, as expected the average processing time decreases almost linearly when the number of processes increases. The graph below shows the trend more clearly:



However merely adding more processes does not guarantee an increase in efficiency, as there will be latency due to the switching from threads to processes. Having said that, we should see a much better performance on a real distributed computing environment





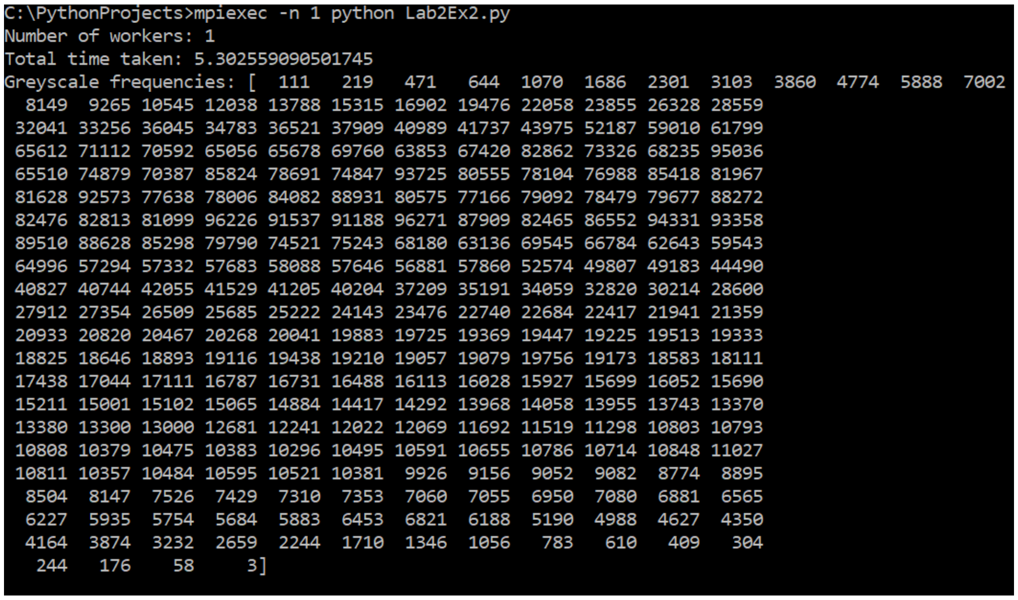
The above histograms merely show the similar output via manual frequency calculation and built-in OpenCV histogram function.

References:

[1] Python Programming Tutorials. (n.d.). Retrieved from https://pythonprogramming.net/bar-chart-histogram-matplotlib-tutorial/
[2] Isai B. Cicourel. (n.d.). Image Manipulation in Python. Retrieved from https://www.codementor.io/isaib.cicourel/image-manipulation-in-python-du1089j1u
[3] Image Processing without OpenCV | Python. (2019, April 3). Retrieved from https://www.geeksforgeeks.org/image-processing-without-opencv-python/
[4] Python Tutorial - Image Histogram - 2018. (n.d.). Retrieved from https://www.bogotobogo.com/python/OpenCV_Python/python_opencv3_image_histogram_calcHist.php
[5] Image manipulation in Python using MPI. (11, 4). Retrieved from https://stackoverflow.com/questions/23533250/image-manipulation-in-python-using-mpi
[6] Python - Calculate histogram of image. (2, 5). Retrieved from https://stackoverflow.com/questions/22159160/python-calculate-histogram-of-image
[7] MPI Scatter, Gather, and Allgather · MPI Tutorial. (n.d.). Retrieved from http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
[8] How can I send part of an array with scatter? (6, 6). Retrieved from https://stackoverflow.com/questions/12812422/how-can-i-send-part-of-an-array-with-scatter
[9] MPI Reduce and Allreduce · MPI Tutorial. (n.d.). Retrieved from http://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Appendix A

Sample output for Question 2



```
C:\PythonProjects>mpiexec -n 1 python Lab2Ex2.py
Number of workers: 1
Total time taken: 5.302559090501745
Greyscale frequencies: [  111   219   471   644  1070  1686  2301  3103  3860  4774  5888  7002
  8149  9265 10545 12038 13788 15315 16902 19476 22058 23855 26328 28559
 32041 33256 36045 34783 36521 37909 40989 41737 43975 52187 59010 61799
 65612 71112 70592 65056 65678 69760 63853 67420 82862 73326 68235 95036
 65510 74879 70387 85824 78691 74847 93725 80555 78104 76988 85418 81967
 81628 92573 77638 78006 84082 88931 80575 77166 79092 78479 79677 88272
 82476 82813 81099 96226 91537 91188 96271 87909 82465 86552 94331 93358
 89510 88628 85298 79790 74521 75243 68180 63136 69545 66784 62643 59543
 64996 57294 57332 57683 58088 57646 56881 57860 52574 49807 49183 44490
 40827 40744 42055 41529 41205 40204 37209 35191 34059 32820 30214 28600
 27912 27354 26509 25685 25222 24143 23476 22740 22684 22417 21941 21359
 20933 20820 20467 20268 20041 19883 19725 19369 19447 19225 19513 19333
 18825 18646 18893 19116 19438 19210 19057 19079 19756 19173 18583 18111
 17438 17044 17111 16787 16731 16488 16113 16028 15927 15699 16052 15690
 15211 15001 15102 15065 14884 14417 14292 13968 14058 13955 13743 13370
 13380 13300 13000 12681 12241 12022 12069 11692 11519 11298 10803 10793
 10808 10379 10475 10383 10296 10495 10591 10655 10786 10714 10848 11027
 10811 10357 10484 10595 10521 10381  9926  9156  9052  9082  8774  8895
  8504  8147  7526  7429  7310  7353  7060  7055  6950  7080  6881  6565
  6227  5935  5754  5684  5883  6453  6821  6188  5190  4988  4627  4350
  4164  3874  3232  2659  2244  1710  1346  1056   783   610   409   304
   244   176    58     3]
```

Appendix B

Code for state-of-the-art frequency histogram

```python
import cv2
from matplotlib import pyplot as plt

img = cv2.imread('img.png', cv2.IMREAD_GRAYSCALE)
cv2.namedWindow('KLCC', cv2.WINDOW_NORMAL)
cv2.imshow('KLCC',img)
hist = cv2.calcHist([img],[0],None,[256],[0,256])

cm = plt.cm.get_cmap('RdYlBu_r')
n,bins,patches=plt.hist(img.ravel(),256,[0,256])
print(patches)
plt.title('Histogram of greyscale using OpenCV (hist) function')
bin_centers = 0.5 * (bins[:-1] + bins[1:])
# scale values to interval [0,1]
col = bin_centers - min(bin_centers)
col /= max(col)
for c, p in zip(col, patches):
    plt.setp(p, 'facecolor', cm(c))
plt.show()
while True:
    k = cv2.waitKey(0) & 0xFF
    if k == 27: break             # ESC key to exit
cv2.destroyAllWindows()
```