Distributed data analytics (2016-2017)
Information Systems and Machine Learning Lab
University of Hildesheim

Exercise Sheet 3

Mario Rodríguez Ruiz

May 4, 2017

Contents

1 Explain your system **3**

2 Point to Point Communication **3**

2.1 Implement the sendAll routine using the naive way i.e using a single for loop 3

2.2 Implement the sendAll routine using the efficient way as explained above . . 4

2.3 Compare performance of the two implementations by recording the time to nish each tas 5

3 Collective Communication **8**

3.1 Pick an image with a high resolution i.e. resolution > 2048x2048 8

3.2 Data division strategy i.e. how you divide your data among processes 8

3.3 How you assign tasks to different processes? 8

3.4 How you combine results from all the processes? 9

3.5 Did you implement for RGB or gray scale histogram? 10

3.6 Provide runtime analysis on varying number of processes 10

3.6.1 Gray scale histogram 10

List of Figures

2.1 Recoursive doubling 4

2.2 Compare performance(16 processes) 7

2.3 Compare performance(32 processes) 7

3.1 Image to processing 8

3.2 Combine results from all the processes 9

3.3 Combine results from all the processes 10

List of Tables

1.1 My system 3

2.1 Compare naive and efficient 6

3.1 Results for processes-time 10

1 Explain your system

	Home System
Machine	Asus Notebook ROG G60Jx
Operating System	Windows 10 Pro 64-bit
CPU	Intel Core i7 720QM @1.60GHz
Number of cores	4
Number of threads	8
RAM	16GB @665MHz (9-9-9-24)
Programming language version Python	v3.6.1:69c0db5 64 bit
Programming language version Java	v1.8

Table 1.1: My system

2 Point to Point Communication

2.1 Implement the sendAll routine using the naive way i.e using a single for loop

A naive way to send this array is using a for loop at Process 0 and sequentially send it to all other processes, it will take P-1 (16-1 or 32-1) steps.

```
1  import numpy as np
2  from mpi4py import MPI
3
4  PROCESS_INIT = 1
5  TAM_ARRAY = 100000000
6
7  comm = MPI.COMM_WORLD
8  rank = comm.Get_rank()
9  size = comm.Get_size()
10
11 # ----------- The naive way --------------
12 comm.Barrier()
13 start = MPI.Wtime()
14 if rank == 0:
15   data = np.arange(TAM_ARRAY, dtype='i')
16   for i in range(PROCESS_INIT, size):
17     comm.Send([data, MPI.INT], dest=i, tag=11)
18     #print ("Array sent from {} to {}".format(rank, i))
19 else:
20   dataRecv = np.empty(TAM_ARRAY, dtype='i')
21   comm.Recv([dataRecv, MPI.INT], source=0, tag=11)
22   #print ("Array received in {} from {}".format(rank, 0))
23   #print(dataRecv)
24
25 comm.Barrier()
26 end = MPI.Wtime()
```

```
27
28 tim = end-start
29 totalTime = comm.allreduce(tim, op = MPI.SUM)
30
31 if rank == 0:
32   print("Time naive: ", (totalTime/size))
```

2.2 Implement the sendAll routine using the efficient way as explained above

The first processor first sends the data to only two other processors. Then each of these processors send the data to two other processors, and so on (Figure 2.1). At each stage, the number of processors sending and receiving the same array.

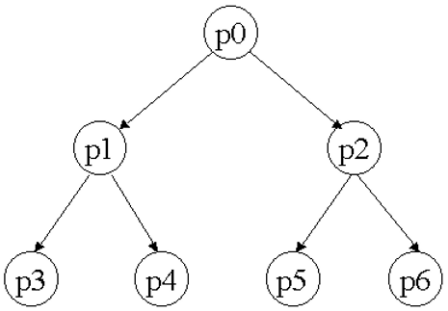

Figure 2.1: Recoursive doubling

```
1  import numpy as np
2  from mpi4py import MPI
3
4  PROCESS_INIT = 1
5  TAM_ARRAY = 100000000
6
7  comm = MPI.COMM_WORLD
8  rank = comm.Get_rank()
9  size = comm.Get_size()
10
11 # ----------- The efficient way --------------
12 comm.Barrier()
13 start = MPI.Wtime()
14 if rank == 0:
15   data = np.arange(TAM_ARRAY, dtype='i')
16   for i in range(PROCESS_INIT, 3):
17     comm.Send([data, MPI.INT], dest=i, tag=11)
18     #print ("Array sent from {} to {}".format(rank, i))
19 else:
20   dataRecv = np.empty(TAM_ARRAY, dtype='i')
21   recvProc = int((rank-1)/2)
```

```
22    comm.Recv([dataRecv, MPI.INT], source=recvProc, tag=11)
23    #print ("Array received in {} from {}".format(rank, recvProc
          ))
24
25    destA = 2*rank + 1
26    if destA < size:
27      comm.Send([dataRecv, MPI.INT], dest=destA, tag=11)
28      #print ("Array sent from {} to {}".format(rank, destA))
29
30    destB = 2*rank + 2
31    if destB < size:
32      comm.Send([dataRecv, MPI.INT], dest=destB, tag=11)
33      #          print ("Array sent from {} to {}".format(rank,
            destB))
34  comm.Barrier()
35  end = MPI.Wtime()
```

2.3 Compare performance of the two implementations by recording the time to nish each tas

Measurements have been made using the following script, which automatically changes the number of processes and the size of the vector and, finally, prints the time of both the total program and the processes.

```
1  import subprocess
2  import time
3
4  tamIni = 1000
5  tamFin = 100000000
6  procsIni = 16
7  procsFin = 32
8  pathNaive = ' python C:\\Users\\Mario\\Documents\\
       EclipseProjects\\Exercise3\\pointToPoint\\naive.py '
9  pathEfficient = ' python C:\\Users\\Mario\\Documents\\
       EclipseProjects\\Exercise3\\pointToPoint\\efficient.py '
10 mpiexec = 'mpiexec -n '
11
12 print("Measure kind\tProcesses\tTam. Array\tTime")
13 while tamIni <= tamFin:
14   while procsIni <= procsFin:
15     fullNaive = mpiexec + str(procsIni) + pathNaive + str(
           tamIni)
16     fullEficcient = mpiexec + str(procsIni) + pathEfficient +
           str (tamIni)
17
18     start = time.time()
19     subprocess.call(fullNaive)
```

```
20      end = time.time()
21      print("Full program time naive:\t{}\t{}\t{}".format(
            procsIni, tamIni, (end-start)))
22
23      start = time.time()
24      subprocess.call(fullEficcient)
25      end = time.time()
26      print("Full program time efficient:\t{}\t{}\t{}".format(
            procsIni, tamIni, (end-start)))
27
28      procsIni+=procsIni
29    print('----')
30    procsIni=16
31    tamIni*=100
```

The following table (Table 2.1) shows the breakdown of results with respect to time. The time is classified according to the size of the array and the processes that have executed the program.
It should be noted that the time measurements obtained have been the result of the average time taken by each process to perform its task independently. The total time of program execution has not been taken into account because this could not give a correct estimate of performance.

Sum processes time	Processes	Tam. Array	Time
naive	16	1E+03	0,00292
efficient	16	1E+03	0,00156
naive	32	1E+03	0,00543
efficient	32	1E+03	0,00374
—-			
naive	16	1E+05	0,01027
efficient	16	1E+05	0,00541
naive	32	1E+05	0,02915
efficient	32	1E+05	0,01017
—-			
naive	16	1E+07	0,35217
efficient	16	1E+07	0,20902
naive	32	1E+07	0,69521
efficient	32	1E+07	0,36756
—-			
naive	16	1E+08	3,17952
efficient	16	1E+08	1,90448
naive	32	1E+08	6,53734
efficient	32	1E+08	3,24019

Table 2.1: Compare naive and efficient

Below are the graphs of the results found in the tables above. The first of these (Figure

2.2) encompasses the performance obtained under 16 different processes. On the other hand, Figure 2.3 classifies the results obtained with 32 different processes.

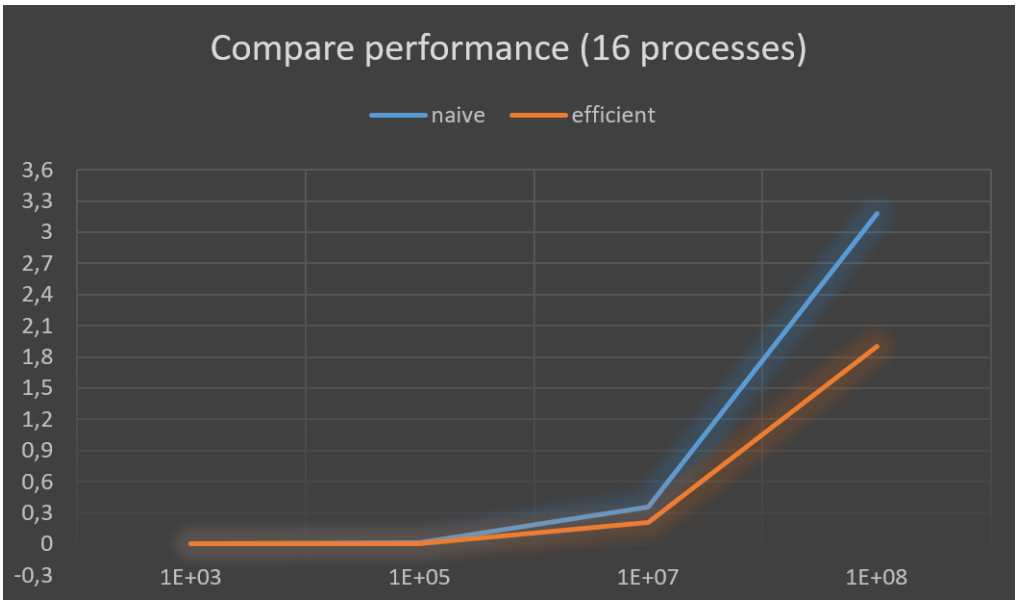

Figure 2.2: Compare performance(16 processes)

Figure 2.2 and Figure 2.3 show that the theoretical calculations prior to the execution of the algorithm are certain. In almost all results, the time obtained in **efficient** is close to half that obtained with **naive**.

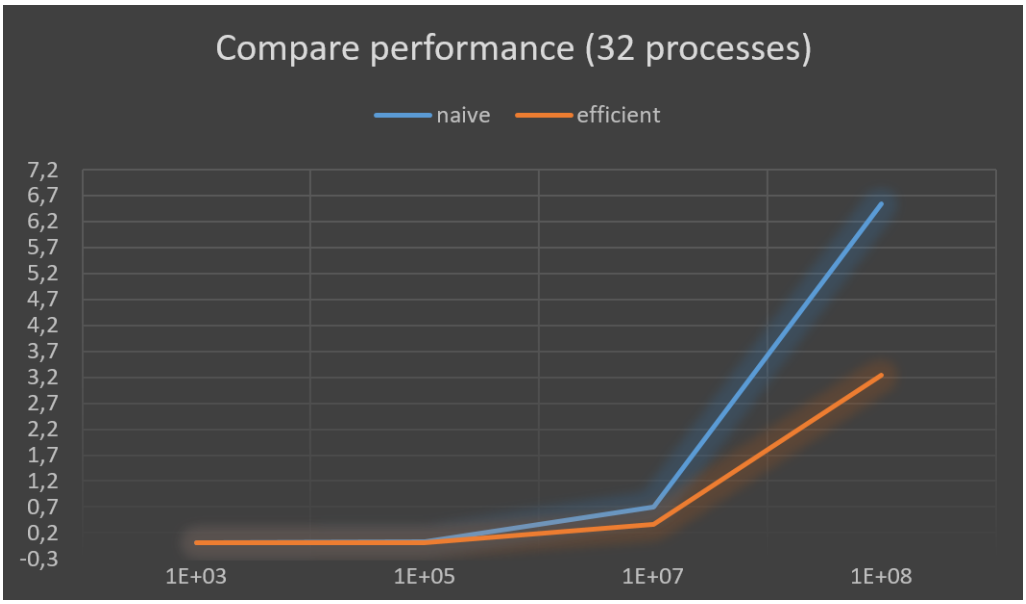

Figure 2.3: Compare performance(32 processes)

3 Collective Communication

3.1 Pick an image with a high resolution i.e. resolution > 2048x2048

The image chosen is the one shown below (Figure 3.1).

Figure 3.1: Image to processing

$height = 3000$
$width = 4500$
https://www.gratistodo.com/wp-content/uploads/2016/08/barco-pirata-the-flying-dutchman-el-
jpg

3.2 Data division strategy i.e. how you divide your data among processes

Dividing the image matrix into as many portions as processes.

3.3 How you assign tasks to different processes?

```
1  # Maximum number of height pixels per process
2  # height of image (3000) / number of processes
3  y_advances = int(gray_img.shape[0]/size)
4
5  # Maximum number of width pixels per process
6  # width of image (4500) / number of processes
7  x_advances = int(gray_img.shape[1]/size)
8
9  # Index that controls the start of pixels height
10 y_ini = int((rank-1)*y_advances)
11
12 # Index that controls the end of pixels height
13 y_fin = y_advances*rank
14
15 # Index that controls the start of pixels width
16 x_ini = (rank-1)*x_advances
17
18 # Index that controls the end of pixels width
```

```
19 x_fin = x_advances*rank
20
21 # For the entire range of height and width corresponding to a
      process
22 # a portion of pixels is extracted and
23 # then the histogram is calculated for this portion.
24 for r in range(y_ini, y_fin):
25   for c in range(x_ini, x_fin):
26     portion = gray_img[r:r+size,c:c+size]
27     hist = np.histogram(portion,bins=grey_levels)
```

3.4 How you combine results from all the processes?

To combine all the results, I've used Gather. All processes send data to a root process that collects the data received.

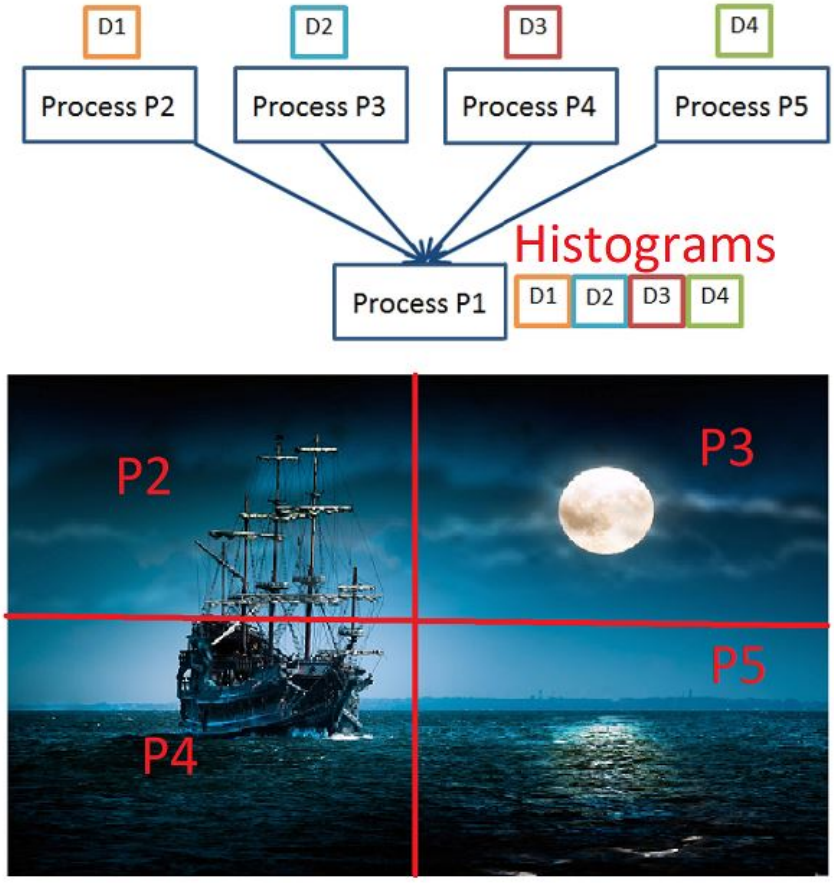

Figure 3.2: Combine results from all the processes

After completing the calculation of the histogram for a process:

```
1 hist = comm.gather(hist, root=0)
```

3.5 Did you implement for RGB or gray scale histogram?

Gray scale histogram

3.6 Provide runtime analysis on varying number of processes

3.6.1 Gray scale histogram

Processes	Time(s)
1	2143,492
2	666,215
3	373,139
4	215,181
8	95,233
16	49,546
32	29,376

Table 3.1: Results for processes-time

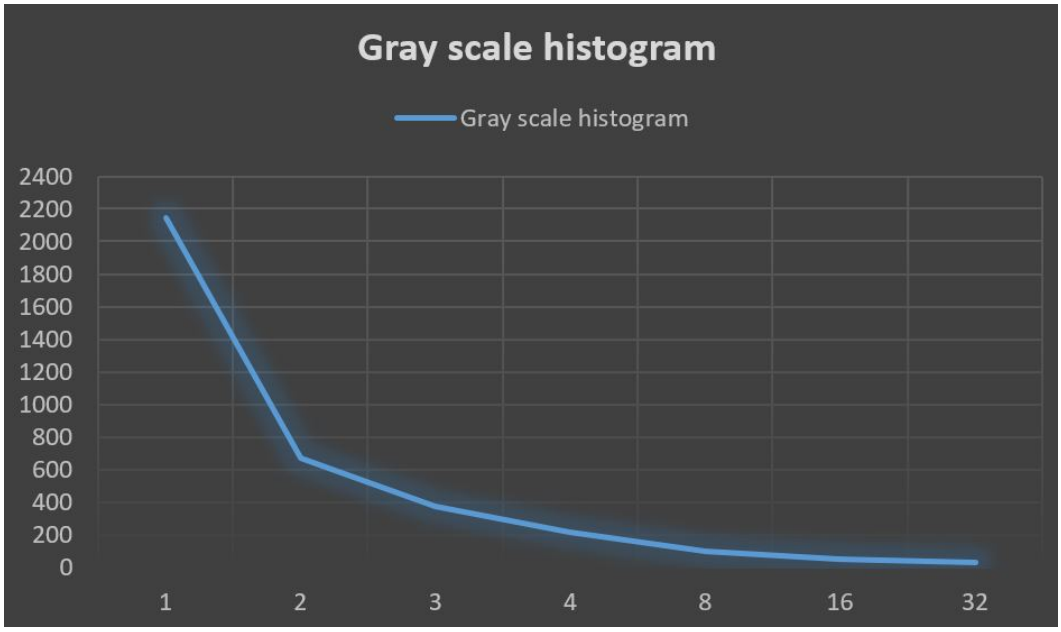

Figure 3.3: Combine results from all the processes

It can be seen in Table X how results improve with respect to time as the number of processes running in parallel increases.