Lab Course: Distributed Data Analytics
Exercise Sheet 1
Group 2 - Monday

Submitted by: Sruthy Annie Santhosh, 312213

Topic: Distributed Computing with Message Passing Interface (MPI)

- Exercise 0: Explain your system

  Hardware and Software SetUp:

Processor	M1 chip
Operating System	MacOS Monterey
Number of Cores	8
RAM	8 GB
Python version	3.8.8

  The code editor used in this sheet : VSCode
  The output images attached are screenshots from the terminal.

- Exercise 1: Basic Parallel Vector Operations with MPI

  <u>Part A) Add two vectors and store results in a third vector.</u>

  In this question, we have to do vector addition in a parallel manner, using MPI. Point to point communication is done. We mainly use two MPI commands : send and recv for sending and receiving data between processes (taken from the MPI documentation in learnweb).

  Here the master worker split the vectors to different sizes and send them to the other workers. Then the other workers find the partial sums in a parallel manner and send them back to the master worker. Finally the

master worker appends all the results to obtain the output vector. The worker having rank = 0 acts as the master worker.

Input: 2 vectors of same size
Output : Vector of the same size as that of the input.

In this program, I have done vector addition for 3 different vector sizes { 10^3, 10^4,10^5}. The program is also run for different numbers of workers like {2,3,4}, this value is taken from the terminal. For verifying whether the addition process is happening correctly, addition is done on vectors of size 3 also. The major steps involved are:

Rank is 0 (master) :
1) The timer starts(MPI.Wtime()). Initializes 2 random numpy vectors of the given size. The vectors are split into different sub-vectors corresponding to the number of other workers available. This splitting is done using array_split. Array-split ensure that unequal splits can also occur.

```
##Splitting the vector so as to send to different workers
V1_split = np.array_split(V1, size-1)
V2_split = np.array_split(V2, size-1)
```

2) For loop is used to send the sub-vectors to various other workers using send command. The sub-vectors and the destination worker rank is specified.

```
#for loop to send vector splits to other workers
for i in range(size-1):
    vector = [V1_split[i-1],V2_split[i-1]]
    comm.send(vector,i+1)
```

3) Another for loop is used to receive the data which will be send back by the other workers to the master. These data will be arrays containing the sum of elts of their corresponding sub-vectors.

```
#for loop to recieve the sum of vectors from other workers
for i in range(size-1):
    final = comm.recv(source=MPI.ANY_SOURCE)
    V3 =np.append(V3,final) #Appending all the partial sums
```

4) These partial sums are appended by master to form the final output.
5) The timer ends.

For other workers (rank!=0):
1)The vector splits are received from the root using comm.recv
2) Addition of the sub-vectors are carried out.
3) The sum array is send back to master worker using comm.send

```
#Vector splits are recieved and partial sums are found
vector = comm.recv(source=0)
print("\nVector 1 has reached rank ",rank)
print("\nVector 2 has reached rank ",rank)
sum = vector[0] + vector[1]

#The partial sum is send back to master worker
comm.send(sum,0)
```

The output for n=3 (for verification)

```
The length of the vectors -  3

The first vector-  [0.5730024  0.08034225 0.62542778]

The second vector-  [0.36963733 0.13616636 0.57873096]

The number of splits- 3

Sum of the vectors of size 3 by workers 4 - [0.94263972 0.21650861 1.20415873]

Time taken by 4 workers for vector of size 3 is 0.00037166599940974265
```

Size of the vector	Number of workers	Time taken(s)
10^3	2	0.014627
10^3	3	0.015638
10^3	4	0.025003
10^4	2	0.001482
10^4	3	0.002209
10^4	4	0.000879
10^5	2	0.007773
10^5	3	0.004591
10^5	4	0.006282

From the above table we can observe that, as the number of workers increase, the time is not decreasing. The time is actually increasing for size 10^3 and 10^5. For 10^4, we see a slight decrease. Also, as the vector size increases, the time decreases. Since multiple processes are created to parallely run on a single physical computer, more communication overhead would be there. That maybe the reason that time increases even when we increase the workers.

Part B) Find an average of numbers in a vector.

In this part, we have to find the average of the numbers in a vector using MPI point to point communication. In this program also we are using send and recv commands of MPI. Here the master worker splits the array into sub arrays using array_split. Then the sub arrays are send to various workers using comm.send
The other workers receive the subarrays using comm,receive and then calculate the mean which will be send back to the master. The master

worker calculates the mean of all the partial means obtained and hence finds the average of the input vector.

Input: Vector of size {10^3, 10^4,10^5}
Output: Average value

Here the number of workers used are {2,3,4}

1) Splitting of vectors according to the size of the workers

```
##Splitting the vector so as to send to different workers
V1_split = np.array_split(V1, size-1)
```

2) Master worker (rank == 0)

```
#for loop to send vector splits to other workers
for i in range(size-1):
    vector = V1_split[i-1]
    comm.send(vector,i+1)
```

```
#for loop to recieve the avg of vectors from other workers
for i in range(size-1):
    final = comm.recv(source=MPI.ANY_SOURCE)
    V2 =np.append(V2,final) #Appending all the partial averages
```

The mean of V2 is calculated which is the output.

3) Other workers (rank!=0)

```
## Other workers
else:
    #Vector splits are recieved and partial means are found
    vector = comm.recv(source=0)
    print("\nVector 1 has reached rank ",rank)

    avg = np.mean(vector)

    #The partial sum is send back to master worker
    comm.send(avg,0)
```

Size of the vector	Number of Workers	Time taken (s)
10^3	2	0.006224
10^3	3	0.006532
10^3	4	0.006181
10^4	2	0.001007
10^4	3	0.000619
10^4	4	0.000443
10^5	2	0.002424
10^5	3	0.002995
10^5	4	0.003343

We observe a similar trend as observed in the previous question. For size 10^5
The t**ime increases as processes increase**. For size 10^3, the time is fluctuating, it increases first and then decreases while for 10^4 it is decreasing. Overall as s**ize increases the time taken is less**.

Hence I conclude that parallelization is more efficient for larger size data. When using for less size data, the **communication overhead** will not make it beneficial.

Sample output

```
The length of the vectors -  3

The input vector: [0.03740207 0.20068485 0.46453265]

The number of splits- 1

Average of the vector of size 3 by workers 2 - 0.234206524238593 45
```

- Exercise 2: Parallel Matrix Vector multiplication using MPI

In this question also we perform point to point communication using MPI. We use comm.send and comm.recv to send and receive data between workers. Here we need to do multiplication of a matrix with a vector.

Input: Vector(N) and a Matrix(NxN) of sizes {10^2,10^3,10^4}
Output: Vector of same size as that of the input vector

I have done the program on 2, 3 and 4 workers and have timed them.

We know that for matrix multiplication, we multiply the rows of the first matrix with the columns of the second matrix and sum it element wise. Here the second matrix is a vector, hence it doesnot need to be split. We need to split the matrix horizontally (row wise) to get sub matrices.

The general formula for a matrix-vector product is

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

This splitting is done by the master worker. Then it sends these sub matrices and the vector to other workers. The other workers will perform the multiplication and then send back the partial results. These results are received by the master which appends them to a vector to form the final product.

Rank 0 (master worker):

1)Splitting of matrix row wise using array_split

```
#Splitting the matrix horizontally
matrix_split = np.array_split(matrix, size-1)

print("\nThe number of splits-",len(matrix_split))
```

2)Sending and receiving data

```
#Splitting the matrix horizontally
matrix_split = np.array_split(matrix, size-1)

print("\nThe number of splits-",len(matrix_split))
```

```
#for loop to recieve the product of vector and matrix from other workers
for i in range(size-1):
    final = comm.recv(source=MPI.ANY_SOURCE)
    C =np.append(C,final) #Appending all the partial products
```

Other workers (rank !=0)

```
#Vector and Matrix splits are recieved and products are found
data = comm.recv(source=0)
print("\nVector  has reached rank ",rank)
print("\nMatrix split has reached rank ",rank)

#Matrix multiplication is carried out(element wise multiplication of row and vector)
pdt = np.matmul(data[0] , data[1])

#The partial product is send back to master worker
comm.send(pdt,0)
```

Size of the vector	Number of Workers	Time taken (s)
10^2	2	0.002486
10^2	3	0.003406
10^2	4	0.004433
10^3	2	0.023400
10^3	3	0.024598
10^3	4	0.026436
10^4	2	2.689539
10^4	3	2.235485
10^4	4	2.047402

From the table we observe that, for less size ($<10^4$), the time increases with increase in workers. But for size 10^4, the time decreases when we increase workers. Also overall the time taken increases when size increases. **Hence we can say that for matrix of larger size, parallel computation(more workers) takes lesser time** while for lesser size, as workers increase due to communication overhead, time slightly increases.

Sample output:

```
Shape of the matrix-  (10000, 10000)

Matrix- [[0.09442942 0.07027412 0.84975255 ... 0.78934442 0.44876766 0.85499288]
 [0.41564085 0.12304089 0.97481762 ... 0.95227903 0.46253072 0.91919755]
 [0.82474565 0.71439049 0.05313126 ... 0.06771111 0.62024514 0.10248585]
 ...
 [0.77361077 0.3470131  0.35679518 ... 0.71023926 0.13733949 0.92032439]
 [0.55202157 0.6427726  0.17608989 ... 0.15079272 0.70316767 0.43084691]
 [0.19712891 0.25097487 0.39959931 ... 0.12853656 0.18888203 0.8199904 ]]

Vector- [0.5428212  0.25155461 0.01471028 ... 0.92667192 0.73862289 0.72365145]

The number of splits- 3

Product of the vector and matrix of size 10000 by workers 4 - [2517.81767818 2475.45707683
 2531.08456078]

 Shape of final product- (10000,)

Time taken by 4 workers for vector of size 10000 is 2.186921041997266
```

- Exercise 2: Parallel Matrix Operation using MPI

  In this question we have to perform matrix multiplication on 2 matrices (NxN) to get an output matrix (NxN) using collective communication in MPI. Random initialization is done as in the previous questions.

  Input: matrix A and B of sizes $\{10, 10^2, 10^3\}$ (10^4 - system was hanging)
  Output : matrix C of same size as that of A and B.
  Here I used 3 workers {2,3,4} and timed the execution as well.

If $\mathbf{A}$ is an $m \times n$ matrix and $\mathbf{B}$ is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

the *matrix product* $\mathbf{C} = \mathbf{AB}$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matri

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix}$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, ..., m$ and $j = 1, ..., p$.

In matrix multiplication, to get the new elt at position(i,j) we need to find the sum of the element wise product of the row i and col j. This is similar to the above question if we consider the second matrix as a stack of column vectors.

In this question for each element, I am splitting the required row and column into sub arrays for doing element wise multiplication. These required row and column splits are done by the master worker and then send to all workers through scatter method. **Comm.scatter**, allows to send different sub arrays to different workers simultaneously. These workers will do sub-matrix multiplication and then these results will be send back through gather method. **Comm.gather**, collects the values from all the processes simultaneously and stores them. Then the master worker, sums up all the product values to get the required element.

This is done for all the elements using two for loops. This process is timed.

1)Master worker(rank = 0), initialize the matrices.
2)Iteration occurs through each row and column using 2 for loops.

3) For each element, split the required row and column using array split across all the processes which is done by master worker

```
#master worker
if(rank == 0):
    #splitting the matrices according to corresponding row (i)
    # and column(j) for matrix multiplication
    row_split = np.array_split(matrix_A[i,:],size)
    col_split = np.array_split(matrix_B[:,j],size)
```

4)The sub arrays are sent to all the processes using comm.scatter

```
#sending the different sub splits of required row and column to
# all the other workers
row_elts = comm.scatter(row_split, root = 0)
col_elts = comm.scatter(col_split, root = 0)
```

5)The results of element wise multiplication of sub arrays are returned back using comm.gather

```
#Doing the element wise multiplication of the row and col splits
# and returning them all back to master
pdt = comm.gather(np.matmul(row_elts,col_elts),root = 0)
```

6)All the values gathered are summed up by master worker and stored at the correct position of the new matrix.

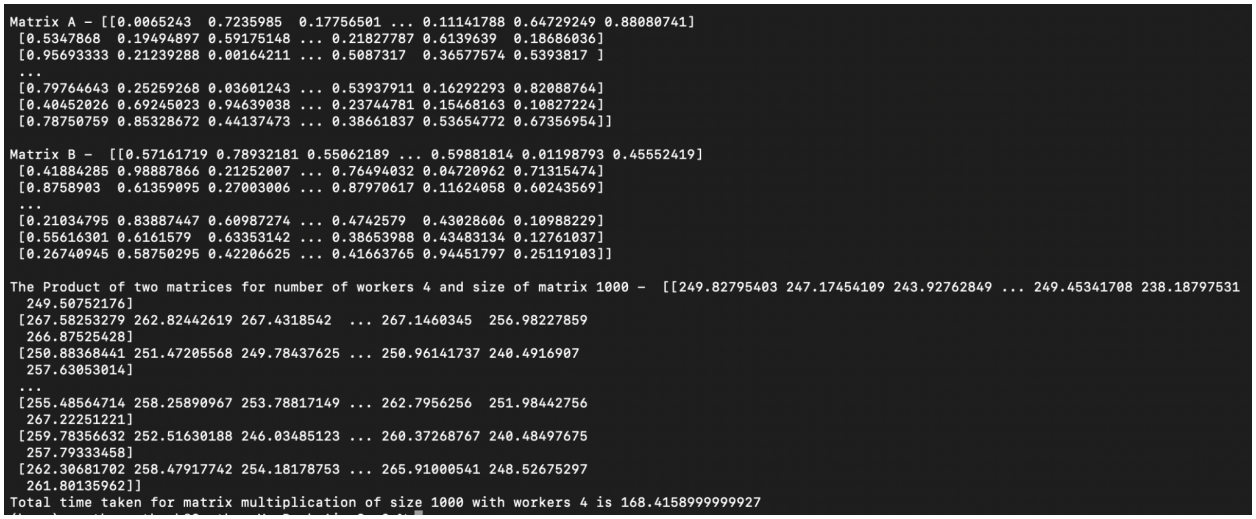
```
#master worker
if(rank == 0):
    #sums the element wise products to get the new elt at position
    # i ,j ( ith row and j th col)
    matrix_C[i][j]=  np.sum(pdt)
```

7) The process repeats for all the elements of the matrix.

Size of the matrix	Number of Workers	Time taken (s)
10	2	0.013928
10	3	0.016375
10	4	0.029531
10	8	0.051469
10^2	2	0.856426
10^2	3	1.124669
10^2	4	1.681387
10^2	8	3.369952
10^3	2	96.07822
10^3	3	121.3685
10^3	4	168.4158
10^3	8	401.7848

From the table we can observe that the t**ime increases when workers increase and when size of the matrix increases**. Thus the parallelization may not be effective as even when workers increase time does not decrease.This may be due to huge communication overhead. Also time increases with the size as expected.

Sample output:



```
Matrix A -  [[0.0065243  0.7235985  0.17756501 ... 0.11141788 0.64729249 0.88080741]
 [0.5347868  0.19494897 0.59175148 ... 0.21827787 0.6139639  0.18686036]
 [0.95693333 0.21239288 0.00164211 ... 0.5087317  0.36577574 0.5393817 ]
 ...
 [0.79764643 0.25259268 0.03601243 ... 0.53937911 0.16292293 0.82088764]
 [0.40452026 0.69245023 0.94639038 ... 0.23744781 0.15468163 0.10827224]
 [0.78750759 0.85328672 0.44137473 ... 0.38661837 0.53654772 0.67356954]]

Matrix B -  [[0.57161719 0.78932181 0.55062189 ... 0.59881814 0.01198793 0.45552419]
 [0.41884285 0.98887866 0.21252007 ... 0.76494032 0.04720962 0.71315474]
 [0.8758903  0.61359095 0.27003006 ... 0.87970617 0.11624058 0.60243569]
 ...
 [0.21034795 0.83887447 0.60987274 ... 0.4742579  0.43028606 0.10988229]
 [0.55616301 0.6161579  0.63353142 ... 0.38653988 0.43483134 0.12761037]
 [0.26740945 0.58750295 0.42206625 ... 0.41663765 0.94451797 0.25119103]]

The Product of two matrices for number of workers 4 and size of matrix 1000 -  [[249.82795403 247.17454109 243.92762849 ... 249.45341708 238.18797531
  249.50752176]
 [267.58253279 262.82442619 267.4318542  ... 267.1460345  256.98227859
  266.87525428]
 [250.88368441 251.47205568 249.78437625 ... 250.96141737 240.4916907
  257.63053014]
 ...
 [255.48564714 258.25890967 253.78817149 ... 262.7956256  251.98442756
  267.22251221]
 [259.78356632 252.51630188 246.03485123 ... 260.37268767 240.48497675
  257.79333458]
 [262.30681702 258.47917742 254.18178753 ... 265.91000541 248.52675297
  261.80135962]]
Total time taken for matrix multiplication of size 1000 with workers 4 is 168.4158999999927
```

References:

- https://mpi4py.readthedocs.io/en/stable/reference/mpi4py.MPI.Comm.html#mpi4py.MPI.Comm.send
- https://numpy.org/doc/stable/reference/generated/numpy.array_split.html
- https://rantahar.github.io/introduction-to-mpi/aio/index.html
- https://stackoverflow.com/questions/64311489/siice-a-numpy-array-based-on-an-interval
- https://stackoverflow.com/questions/40336601/python-appending-array-to-an-array
- https://stackoverflow.com/questions/41575243/matrix-multiplication-using-mpi-scatter-and-mpi-gather
- https://www.christianbaun.de/CGC1718/Skript/CloudPresentation_Shamima_Akhter.pdf