Assignment 2 - Collective communications

Start Assignment

- Due No due date
- Points 20
- Submitting a text entry box

Exercise 1: Parallel Monte Carlo for PI

- Implement a parallel version of Monte Carlo using the function above:
- Ensure your program works correctly if N is not an exact multiple of the number of processes P

```
def compute_points():

    random.seed(42)

    circle_points= 0

    # Total Random numbers generated= possible x
    # values* possible y values
    for i in range(INTERVAL**2):

        # Randomly generated x and y values from a
        # uniform distribution
        # Range of x and y values is -1 to 1

         rand_x= random.uniform(-1, 1)
        rand_y= random.uniform(-1, 1)

        # Distance between (x, y) from the origin
        origin_dist= rand_x**2 + rand_y**2

        # Checking if (x, y) lies inside the circle
        if origin_dist<= 1:
            circle_points+= 1
```

Exercise 2 Parallel Stochastic Gradient Descent for Linear Regression

- Implement a parallel version of Stochastic Gradient Descent (SGD) for linear regression using MPI. The goal is to distribute the dataset across multiple MPI processes, perform local updates (gradient descent steps) on each process, and periodically synchronize the model parameters across all processes.

- Ensure your program works correctly if N is not an exact multiple of the number of processes P

Dataset: Use a synthetic dataset for simplicity. You can generate a dataset with a single feature and a linear relationship with the target variable. For example, you can model the relationship y = 2x + noise, where noise is some Gaussian noise.

```
def compute_gradient(data, labels, weight):
    predictions = data * weight
    errors = predictions - labels
    gradient = 2 * np.dot(data, errors) / len(data)
  return gradient
```

Exercise 3: Matrix vector product

1. Use the code above to implement the MPI version of matrix-vector multiplication.
2. Ensure your program works correctly if N is not an exact multiple of the number of processes P
3. Process 0 compares the result with the `dot` product.
4. `Plot the scalability of your implementation.`

```
import numpy as np
from scipy.sparse import lil_matrix
from numpy.random import rand, seed
from numba import njit
from mpi4py import MPI

''' This program compute parallel csc matrix vector multiplication using mpi '''

COMM = MPI.COMM_WORLD
nbOfproc = COMM.Get_size()
RANK = COMM.Get_rank()

seed(42)

def matrixVectorMult(A, b, x):

    row, col = A.shape
    for i in range(row):
        a = A[i]
        for j in range(col):
            x[i] += a[j] * b[j]

    return 0

########################initialize matrix A and vector b ######################
```

```
#matrix sizes
SIZE = 1000
#Local_size =

# counts = block of each proc
#counts =

if RANK == 0:
    A = lil_matrix((SIZE, SIZE))
    A[0, :100] = rand(100)
    A[1, 100:200] = A[0, :100]
    A.setdiag(rand(SIZE))
    A = A.toarray()
    b = rand(SIZE)
else :
    A = None
    b = None

#########Send b to all procs and scatter A (each proc has its own local matrix#####
#LocalMatrix =
# Scatter the matrix A

######################Compute A*b locally#########################################
#LocalX =

start = MPI.Wtime()
matrixVectorMult(LocalMatrix, b, LocalX)
stop = MPI.Wtime()
if RANK == 0:
    print("CPU time of parallel multiplication is ", (stop - start)*1000)

###################Gather te results ############################################
# sendcouns = local size of result
#sendcounts =
# if RANK == 0:
#     X = ...
# else :
#     X = ..

# Gather the result into X
###################Print the results ############################################
if RANK == 0 :
```

```
    X_ = A.dot(b)
    print("The result of A*b using dot is :", np.max(X_ - X))
  # print("The result of A*b using parallel version is :", X)
```