Code Explanation
DDA Exercise 03

Farjad Ahmed - 1747371
First Semester - Group 1

This report contains detailed explanation and graphs as required in the assignment. Followed by the code and references at the end.

1. First required libraries are imported and variables are initialized. The number of clusters ‘k’ is set to 3, and filename is provided on the filepath.

```python
from array import array
from pkgutil import get_data
from threading import local
from unittest.mock import NonCallableMagicMock
from mpi4py import MPI
import numpy as np
from numpy import genfromtxt
import math
import time
comm = MPI.COMM_WORLD # create a communicator
size = comm.size # get the size of the cluster
rank = comm.rank # get the rank of the process
root = 0 # root process

st = MPI.Wtime()
np.random.seed(0)
#Farjad Ahmed_DDA_Exercise_3_KMeans
k = 3 # number of clusters
filename = 'Absenteeism_at_work_AAA/Absenteeism_at_work.csv'
```

2. Next, we define the necessary functions.

```python
def get_euclidean(a,b):
    sum_squared = 0 # variable to store the sum of squared differences
    for i in range(len(a)):  # iterate over each element of the array
        sum_squared += math.pow((a[i] - b[i]), 2) # add the squared difference to the sum
    return math.sqrt(sum_squared) # take the square root of the sum to get the euclidean distance

def read_file(file):
    my_data = genfromtxt(filename, delimiter=';', skip_header = 1) # read the data from the file
    return my_data

def get_centroids(data, k): # get the centroids
    centroids = np.random.randint(data.shape[0], size=k) # get random centroids
    cents = [] # create a list to store the centroids
    for x in centroids:
        cents.append(np.asanyarray(data[x], dtype = float)) # append the centroids to the list
    return cents # return the list of centroids

def distance_calculation(data_mat, centroids): # calculate the distance between the data and the centroids
    dist_mat = np.zeros((data_mat.shape[0], k)) # create a matrix to store the distances
    for rows in range(data_mat.shape[0]): # iterate over each row of the data matrix
        for p in range(len(centroids)): # iterate over each centroid
            dist_mat[rows][p] = get_euclidean(data_mat[rows], centroids[p]) # calculate the distance between the data and the centroid
    return dist_mat # return the distance matrix
```

The function read_file() uses the method provided by numpy ‘genfromtxt’ to read the csv file. The delimiter is provided in the arguments and header is skipped to avoid the header being added in the data matrix.
The function distance_calculation() takes two parameters,

1. The data chunk that is distributed to a rank
2. Centroid coordinates for each of the k clusters

This function iterates over each row of the data chunk and calculates the distance of each point (row) with the given centroid. Following this, the function get_euclidean() is called, this is used to calculate the euclidean distance between two points. The formula that is implemented in this algorithm is as follows:

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (y_i - x_i)^2}$$

reference[1]

3. Rank 0 takes the responsibility of reading the data and splitting it to be further distributed in the later steps. Moreover, it also calls the get_centroids() function and generates the ‘k’ random centroid points from the dataset.

```
if rank == 0:
    filedata = read_file(filename)  # read the file
    cent_loc = get_centroids(filedata, k) # get the centroids
    sdata = np.array_split(filedata, size, axis=0) # split the data
else:
    sdata = None
    cent_loc = None
```

4. Once, the data is split, which is stored in sdata. This can be scattered to all the ranks and since each rank has to compute the memberships/associations with each cluster, the centroids need to be sent to all rank, this is achieved by broadcasting them. Loopflag is initialized, loop count variable is initialized and old_global variable is initialized that will be used to compare the old centroids with the newly calculated ones in the while loop as explained later in this report.

```
bdata = comm.bcast(cent_loc, root = root) # broadcast the centroids to all the processes
getdata = comm.scatter(sdata , root = 0) # scatter the data to all the processes
old_global = None # create a variable to store the old centroids
loopflag = True  # create a flag to check if the loop should continue
loop = 0 # create a variable to count the number of iterations
```

5. Now, each rank has received the centroids and their respective data chunks. As per the algorithm shared in the question, certain steps will be carried forward until the algorithm converges. Hence, the loop is started at this point.

```
while loopflag: # loop until the flag is false
```

6. In the while loop, the goal is to calculate the associations of each record with its closest cluster, since clusters are started randomly, associations are calculated and as per these associations new centroids are calculated i.e global centroids, until the point that the centroids stop moving and do not change. That reflected that each data point is associated to its actual global mean as per the data.

```
    old_global = bdata # store the old centroids

    distances = distance_calculation(getdata, bdata) # calculate the distance between the data and the centroids

    associations = np.array([]) # create a variable to store the associations
    for d in range(len(distances)): # iterate over each row of the distance matrix
        index_min = np.argmin(distances[d]) # get the index of the minimum distance
        associations = np.append(associations, index_min) # append the index to the list of associations
```

7. In this step, first of all old_global is assigned bdata, which is the randomly selected centroid for the first iteration of this loop. Later on, this is updated with the newly calculated centroids in the previous iteration. Next, distances receives a matrix of rows of size of the data rows and k columns, this contains distances of each row in the data chunk received by a rank with respect to each cluster point.
8. This matrix is used to now associate each row of the data chunk to the closest cluster point. For this, an array called associations is initialized. Next, a loop goes over each of the row of the data and np.argmin is utilized to find the index of the minimum element from that row. Since each row contains 3 columns, referring to the

0th, 1st, 2nd …. Kth cluster, the index thus returned is the corresponding cluster to which that row is associated. This is then appended to the associations array.

9. Since, for each rank the data has been categorized as per the nearest centroid, next task is to organize them as per each cluster. That is to say, to separate data points belonging to each of the kth clusters.

```
    indices = [] # create a list to store the indices of the data
    for ith in range(k): # iterate over each centroid
        find = np.where(associations==ith) # get the indices of the data that belong to the centroid
        indices.append((find[0])) # append the indices to the list

    local_centroids = [] # create a list to store the local centroids
    for ks in range(k): # iterate over each centroid
        local_centroids.append(np.zeros(getdata[0].shape)) # append a zero array to the list
    local_centroids = np.array(local_centroids) # convert the list to an array
    print('local centroids are: ', local_centroids)
```

10. A list indices is created that is of the length k. It iterates over the associations computed and picks the indices of each of the data records corresponding to each of the clusters, being appended to the list ‘indices’. This is done using np.where, which returns the index of the specified value in the arguments.

```
    local_centroids_lengths = np.zeros(len(indices)) # create a list to store the lengths of the local centroids
    for a in range(len(indices)): # iterate over each centroid
        temp1 = [] # create a list to store the data that belongs to the centroid
        for i in indices[a]: # iterate over each index
            temp1.append(np.asarray(getdata[i]))  # append the data to the list
        local_centroids[a] = np.sum(temp1, axis = 0) # calculate the sum of the data and append it to the local centroid
        local_centroids_lengths[a] = len(temp1) # calculate the length of the data and append it to the local centroid

    reduced_points = comm.reduce(local_centroids, root = root) # reduce the local centroids to the root process
    reduced_lenghts = comm.reduce(local_centroids_lengths, root = root) # reduce the local centroids lengths to the root process
```

11. When, all these points are added to the local_centroid list. Nested loops run over this, to sum the data points for each of the cluster along with the count of how many data points were summed to get the total sum for the data chunk at each rank. This information is stored in temp1, and temp1 appends to the local_centroids list and the respective count goes into local_centroids_lenghts, and so on the master loop runs k times to do so for each of the cluster and data points for which the nested loop runs over each row.
12. Once this information is accumulated by each rank, this needs to be combined and added, in order to sum the data points with respect to each centroid and take the mean to find the global centroid. Since, sum and lengths both are calculated, this can be reduced respectively and sent to the rank root i.e 0 in this case.
13. Next, once these sums are reduced at rank 0, a loop runs over the k length that also corresponds to the arrays just computed. And calculate the mean by dividing the ith element of reduced_points by the ith reduced_lengths, this is stored in the variable ‘avg’ and then appended to the list call global_centroids.

```
bdata = comm.bcast(global_centroids, root = root) # broadcast the global centroids to all the processes
a_bdata = np.concatenate(bdata) # concatenate the global centroids
b_old_global = np.concatenate(old_global) # concatenate the old global centroids

if np.all(np.equal(a_bdata, b_old_global)): # if the global centroids are equal to the old global centroids
    loopflag = False # set the flag to false
    getasso = comm.gather(associations, root = root) # gather the associations to the root process
    if getasso is not None: # if the process is not the root process
        getasso = np.concatenate(getasso) # concatenate the associations
        # print('all associations are \n', getasso) # print the associations
else:
    loop += 1 # if the global centroids are not equal to the old global centroids, increment the loop counter
```

14. The global_centroids thus calculated are then broadcasted to all ranks to continue the while loops, calculating the associations and so on. This, global_centroids is also compared to the old_global centroids since this algorithm reiterates until this condition is met, hence for the stopping condition we compared the old and the new centroids using np.all and np.equal after concatenating the data, just to make it simple to debug and track.

15. When the stopping condition is achieved, the loopflag is set to false and associations are gathered to obtain a list of all the associations of each row for the whole data set, through concatenating the association chunk coming from each rank. If the condition is not met, the loop count is increased.
16. Lastly, to calculate the performance of the algorithm, im recording time for rank 0 since it is doing a few more tasks than the other ranks such as loading, splitting the data, generating centroids and also handling reduce and gather processes, hence for simplicity and ease of observation a single rank is used to analyze the performance.

```
if rank == root: # if the process is the root process
    et = MPI.Wtime() # get the end time
    serialTime = 0.24908564300000002  # set the serial time
    diff = et-st # calculate the difference between the start and end time
    print('time taken is', diff) # print the difference
    Sp = serialTime/diff # calculate the speedup
    print('Sp is: ', Sp) # print the speedup
    print('loops count: ', loop) # print the number of iterations
```

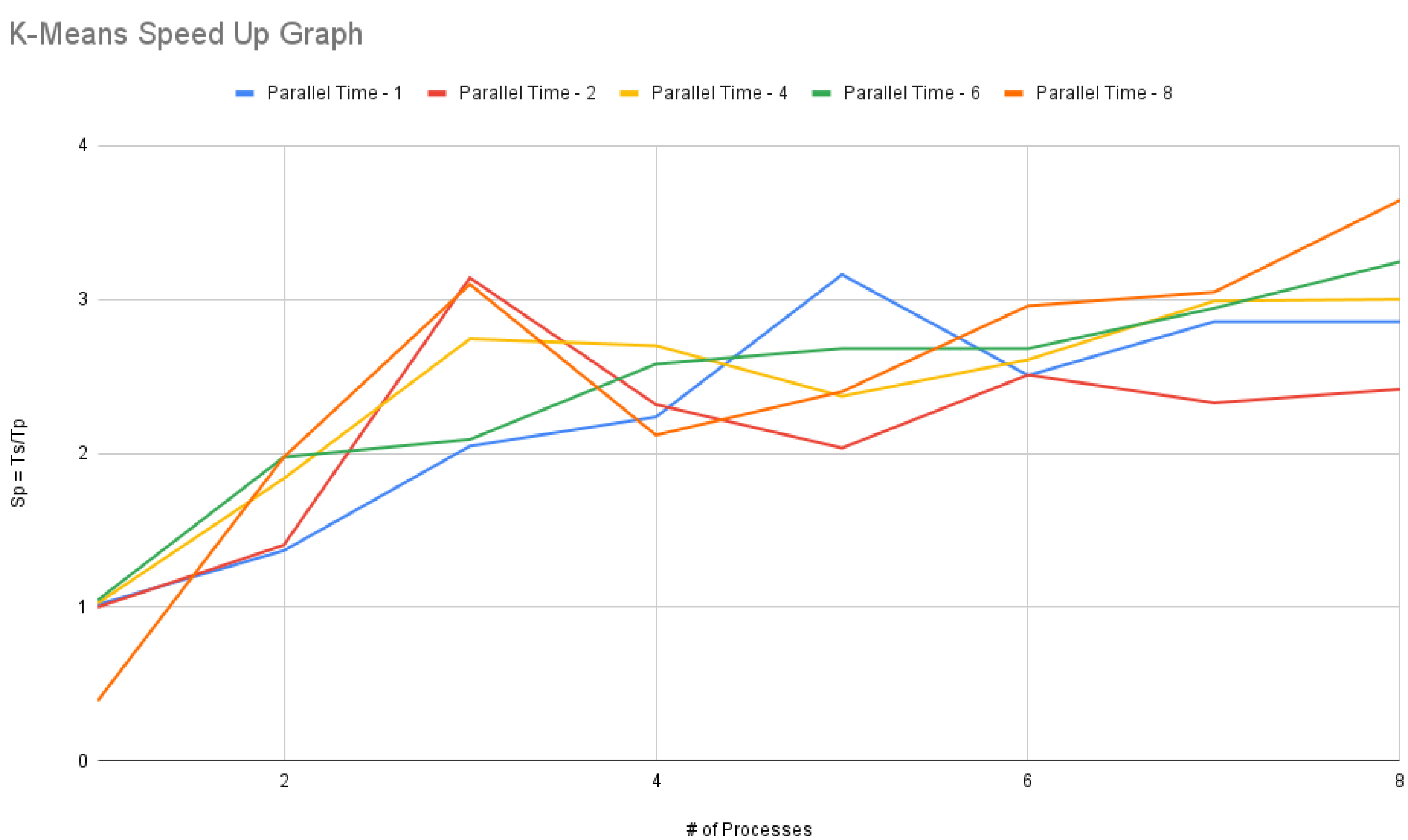
17. The end time is recorded here and the difference is taken from the start time. Sp is calculated as per the mentioned slides in the exercise question. The serial time is adjusted by running it on a single rank and then the records for multiple processes are noted. Loop count is also printed at the end.

Performance Analysis

The performance tables and chart is attached below. It represents the speed up performance of the K-Means parallel algorithm. The speed up graph shows the improvement in running speed as per the graph. However, I did notice some anomalies when running repeatedly on different numbers of processes which could be due to some background processes I believe but overall the performance is as expected.

k = 1			k = 2			k = 4		
Serial Time	# of Processors	Parallel Time - 1	Serial Time	# of Processors	Parallel Time - 2	Serial Time	# of Processors	Parallel Time - 4
0.066362857	1	1.017743149	0.110212461	1	1.001044911	0.441177817	1	1.02640785
	2	1.367709896		2	1.403148944		2	1.83613511
	3	2.046456088		3	3.13929565		3	2.74241891
	4	2.235682274		4	2.316254826		4	2.698850235
	5	3.159726637		5	2.03453998		5	2.370463402
	6	2.505609893		6	2.508746409		6	2.607687863
	7	2.854517904		7	2.327027373		7	2.988261786
	8	2.854517904		8	2.416378603		8	2.999882998

k = 6			k = 8		
Serial Time	# of Processors	Parallel Time - 6	Serial Time	# of Processors	Parallel Time - 8
0.632796196	1	1.046286751	0.634335094	1	0.3926720204
	2	1.974873466		2	1.974750161
	3	2.089798185		3	3.097050248
	4	2.579661614		4	2.118501885
	5	2.679897105		5	2.400037086
	6	2.679897105		6	2.957405139
	7	2.942087546		7	3.046781001
	8	3.243974243		8	3.643460281



The Code

```
from array import array
from pkgutil import get_data
from threading import local
from unittest.mock import NonCallableMagicMock
from mpi4py import MPI
import numpy as np
from numpy import genfromtxt
import math
import time
comm = MPI.COMM_WORLD # create a communicator
size = comm.size # get the size of the cluster
rank = comm.rank # get the rank of the process
root = 0 # root process

st = MPI.Wtime()
np.random.seed(0)
#Farjad Ahmed_DDA_Exercise_3_KMeans
k = 3 # number of clusters
filename = 'Absenteeism_at_work_AAA/Absenteeism_at_work.csv' # file name

def get_euclidean(a,b):
    sum_squared = 0 # variable to store the sum of squared differences
    for i in range(len(a)):  # iterate over each element of the array
        sum_squared += math.pow((a[i] - b[i]), 2) # add the squared difference to the sum
    return math.sqrt(sum_squared) # take the square root of the sum to get the euclidean distance

def read_file(file):
    my_data = genfromtxt(filename, delimiter=';', skip_header = 1) # read the data from the file
    return my_data

def get_centroids(data, k): # get the centroids
    centroids = np.random.randint(data.shape[0], size=k) # get random centroids
    cents = [] # create a list to store the centroids
    for x in centroids:
        cents.append(np.asanyarray(data[x], dtype = float)) # append the centroids to the list
    return cents # return the list of centroids

```

```python

def distance_calculation(data_mat, centroids): # calculate the distance between the data and the centroids
    dist_mat = np.zeros((data_mat.shape[0], k)) # create a matrix to store the distances
    for rows in range(data_mat.shape[0]): # iterate over each row of the data matrix
        for p in range(len(centroids)): # iterate over each centroid
            dist_mat[rows][p] = get_euclidean(data_mat[rows], centroids[p]) # calculate the distance between the data and the centroid
    return dist_mat # return the distance matrix

if rank == 0:
    filedata = read_file(filename)  # read the file
    cent_loc = get_centroids(filedata, k) # get the centroids
    sdata = np.array_split(filedata, size, axis=0) # split the data
else:
    sdata = None
    cent_loc = None


bdata = comm.bcast(cent_loc, root = root) # broadcast the centroids to all the processes
getdata = comm.scatter(sdata , root = 0) # scatter the data to all the processes
old_global = None # create a variable to store the old centroids
loopflag = True  # create a flag to check if the loop should continue
loop = 0 # create a variable to count the number of iterations


while loopflag: # loop until the flag is false

    old_global = bdata # store the old centroids

    distances = distance_calculation(getdata, bdata) # calculate the distance between the data and the centroids

    associations = np.array([]) # create a variable to store the associations
    for d in range(len(distances)): # iterate over each row of the distance matrix
        index_min = np.argmin(distances[d]) # get the index of the minimum distance
        associations = np.append(associations, index_min) # append the index to the list of associations


    indices = [] # create a list to store the indices of the data
    for ith in range(k): # iterate over each centroid
        find = np.where(associations==ith) # get the indices of the data that belong to the centroid
        indices.append((find[0])) # append the indices to the list

    local_centroids = [] # create a list to store the local centroids
    for ks in range(k): # iterate over each centroid
        local_centroids.append(np.zeros(getdata[0].shape)) # append a zero array to the list
    local_centroids = np.array(local_centroids) # convert the list to an array

    local_centroids_lengths = np.zeros(len(indices)) # create a list to store the lengths of the local centroids
    for a in range(len(indices)): # iterate over each centroid
        temp1 = [] # create a list to store the data that belongs to the centroid
        for i in indices[a]: # iterate over each index
            temp1.append(np.asarray(getdata[i]))  # append the data to the list
        local_centroids[a] = np.sum(temp1, axis = 0) # calculate the sum of the data and append it to the local centroid
        local_centroids_lengths[a] = len(temp1) # calculate the length of the data and append it to the local centroid

    reduced_points = comm.reduce(local_centroids, root = root) # reduce the local centroids to the root process
    reduced_lenghts = comm.reduce(local_centroids_lengths, root = root) # reduce the local centroids lengths to the root process


    if reduced_points is not None:  # if the process is not the root process
        if reduced_lenghts is not None: # if the process is not the root process
            global_centroids = [] # create a list to store the global centroids
            for x in range(len(reduced_points)): # iterate over each centroid
                avg = reduced_points[x]/reduced_lenghts[x] # calculate the average of the data and append it to the global centroid
                global_centroids.append(avg) # append the average to the list
    else:
        global_centroids = None # if the process is the root process, set the global centroids to None

    bdata = comm.bcast(global_centroids, root = root) # broadcast the global centroids to all the processes
    a_bdata = np.concatenate(bdata) # concatenate the global centroids
    b_old_global = np.concatenate(old_global) # concatenate the old global centroids

```

```python
    if np.all(np.equal(a_bdata, b_old_global)): # if the global centroids are equal to the old global centroids
        loopflag = False # set the flag to false
        getasso = comm.gather(associations, root = root) # gather the associations to the root process
        if getasso is not None: # if the process is not the root process
            getasso = np.concatenate(getasso) # concatenate the associations
            print('all associations are \n', getasso) # print the associations
    else:
        loop += 1 # if the global centroids are not equal to the old global centroids, increment the loop counter

if rank == root: # if the process is the root process
    et = MPI.Wtime() # get the end time
    serialTime = 0.24791485400000002  # set the serial time
    diff = et-st # calculate the difference between the start and end time
    print('time taken is', diff) # print the difference
    Sp = serialTime/diff # calculate the speedup
    print('Sp is: ', Sp) # print the speedup
    print('loops count: ', loop) # print the number of iterations
```

References

[1] https://towardsdatascience.com/optimising-pairwise-euclidean-distance-calculations-using-python-fc020112c984