Exercise 1 – Basic Parallel Vector Operations with MPI

a) Add two vectors and store results in a third vector.

```
from mpi4py import MPI
import numpy as np
```

```
comm=MPI.COMM_WORLD
worker_id= comm.Get_rank()
p=comm.Get_size()
```

```
N=10**8
group=round(N/p)

def add(x,y): #summation of arrays for groups
    L = len(x)
    z=np.zeros(L)
    for i in range(L):
        z[i]=x[i]+y[i]
    return z

t0=MPI.Wtime()

if worker_id !=0:
    x=comm.recv(source=0, tag=1)
    y=comm.recv(source=0, tag=2)          #created arrays are received

    x=x[(group*worker_id):(group*(1+worker_id))]
    y=y[(group*worker_id):(group*(1+worker_id))]       #arrays are split for each worker
    z=add(x,y)

    comm.send(z,dest=0)                # total of each worker sent Worker 0
else:
    x=np.random.randint(1,10,N)
    y=np.random.randint(1,10,N)
    print("N:", N)
    print("x=",x)
    print("y=",y)

    for i in range(1,p):
        comm.send(x, dest=i, tag=1)
        comm.send(y, dest=i, tag=2)         #created arrays are sent to workers

    x=x[(group*worker_id):(group*(1+worker_id))]
    y=y[(group*worker_id):(group*(1+worker_id))]
    z=add(x,y)

    z_2=np.zeros(group)
    for i in range(1,p):
        z_2=comm.recv(source=i)
        z=np.append(z,z_2)                  #append to have result

    print("z=", z)

    t1=MPI.Wtime()-t0
    print("Timing:", t1)
```

Explanations:

Master node is Worker 0. Inside Worker 0, x and y arrays are created (size=N and random integers between 1 and 9) and then sent to other workers by using comm.send function. Workers assigned to arrays with each has size N/p (size of vector/ number of workers). For example, if there are 2 workers and array size is 100 then worker 0 takes array part of 0-49 and worker 1 takes 50-99. Add function used for array summation which workers sum their x and y parts then return to z. In else statement, Worker 0 sums first part of arrays and rest of the workers used comm.send to send their summed results to else part. Worker 0 receives results from other workers by comm.recv function. At the end, each worker results are appended to single array.

Result example and timing table with explanations:

N=10**8

```
(base) C:\Users\user>mpiexec -n 1 python part_1a.py
N: 100000000
x= [8 6 6 ... 1 1 8]
y= [5 6 4 ... 1 5 6]
z= [13. 12. 10. ...  2.  6. 14.]
Timing: 32.261695499997586

(base) C:\Users\user>mpiexec -n 2 python part_1a.py
N: 100000000
x= [5 9 9 ... 7 1 8]
y= [6 5 6 ... 5 9 5]
z= [11. 14. 15. ... 12. 10. 13.]
Timing: 21.741025600000285

(base) C:\Users\user>mpiexec -n 3 python part_1a.py
N: 100000000
x= [7 2 5 ... 4 3 2]
y= [5 9 8 ... 3 8 9]
z= [12. 11. 13. ...  4.  7. 11.]
Timing: 17.80743180005811

(base) C:\Users\user>mpiexec -n 4 python part_1a.py
N: 100000000
x= [6 4 9 ... 9 4 9]
y= [3 5 1 ... 2 4 2]
z= [ 9.  9. 10. ... 11.  8. 11.]
Timing: 17.20675129990559

(base) C:\Users\user>mpiexec -n 5 python part_1a.py
N: 100000000
x= [6 5 4 ... 1 1 1]
y= [7 6 2 ... 4 6 7]
z= [13. 11.  6. ...  5.  7.  8.]
Timing: 17.589539999957196
```

Tables are in seconds.

Size/Workers	1	2	3	4	5
10**6	0.3377	0.2088	0.1863	0.1598	0.1933
10**7	3.3856	2.0318	1.6361	1.6280	1.6737
10**8	32.2617	21.7410	17.8074	17.2068	17.5895

Starting from N=10**10 system gives memory error. Also, trying 10**9 computer become so slow and not ending in long time, so I couldn't wait it to finish. In this exercise, N=10**(6-7-8) were tried.

As I expected timing decreased when number of workers increase to my processor number-4 and started to increase after that. So, calculation can be speed up to a point with increasing worker numbers because of their communication with each other takes time too. Therefore, in this problem optimal number of workers are 4 for me. Shortest time is reached when size of array is smaller (N) along with optimal number of workers. Also, calculation is checked and working fine.

b) Find an average of numbers in a vector.

```
import numpy as np
from mpi4py import MPI
```

```
comm=MPI.COMM_WORLD
worker_id=comm.Get_rank()
p=comm.Get_size()
```

```
N=10**7
group=round(N/p)

def add2(v):          #array summation
    sum_all=sum(v)
    return sum_all

t0=MPI.Wtime()

if worker_id!=0:
    v=comm.recv(source=0, tag=1)
    v=v[(group*worker_id):(group*(1+worker_id))]       #array is split for each worker
    sum_all =add2(v)
    comm.send(sum_all, dest=0)
else:
    v=np.random.randint(1,10,N)
    for i in range(1,p):
        comm.send(v, dest=i, tag=1)         # total of each worker sent Worker 0

    v=v[(group*worker_id):(group*(1+worker_id))]
    sum_all=add2(v)
    sum_all2=np.zeros(group)
    for i in range(1, p):
        sum_all2=comm.recv(source=i)
        sum_all+=sum_all2
    ave=sum_all/N                  #average to have result
    print("N=",N)
    print("v=",v)
    print("Average of numbers in vector v=", ave)
    t1=MPI.Wtime()-t0
    print("Timing:",t1)
```

Explanation:

Similar to the part a. Only difference is this time there is one vector v which needs a summation of its arrays (using add2 function) by “p” number of workers and then divide the summation to “N” to get average numbers in vector v. Again, MPI communication commands (comm.send and .recv) are used between workers to transfer data.

Result example and timing table with explanations:

N=10**8

```
(base) C:\Users\user>mpiexec -n 1 python part_1b.py
N= 100000000
Average of numbers in vector v= 5.00015516
Timing: 10.12362840003334

(base) C:\Users\user>mpiexec -n 2 python part_1b.py
N= 100000000
Average of numbers in vector v= 5.00014735
Timing: 6.827175999991596

(base) C:\Users\user>mpiexec -n 3 python part_1b.py
N= 100000000
Average of numbers in vector v= 4.99995011
Timing: 6.239050600095652

(base) C:\Users\user>mpiexec -n 4 python part_1b.py
N= 100000000
Average of numbers in vector v= 5.00005424
Timing: 6.216576399980113

(base) C:\Users\user>mpiexec -n 5 python part_1b.py
N= 100000000
Average of numbers in vector v= 4.99946601
Timing: 6.945105299935676
```

Size/Workers	1	2	3	4	5
10**6	0.1014	0.0746	0.0596	0.0652	0.0655
10**7	0.9480	0.6337	0.5948	0.5851	0.6083
10**8	10.1236	6.8271	6.2390	6.2166	6.9451

Again, memory problem with 10**10 and seems to take so long with 10**9. So, 10**(6-7-8) are tried.

As size of array increases, optimal number of workers become 4. However, when it is 10**6 it is fastest with 3 workers. Other than that, results are similar with previous part (a). It seems like when size(N) of the problem increases it becomes more efficient to use number of processors you have.

Exercise 2 – Parallel Matrix Vector Multiplication using MPI

```
import numpy as np
from mpi4py import MPI
```

```
comm=MPI.COMM_WORLD
worker_id=comm.Get_rank()
p=comm.Get_size()
```

```
N=10**2
group=round(N/p)

t0=MPI.Wtime()

if worker_id!=0:
    A=comm.recv(source=0,tag=1)
    b=comm.recv(source=0,tag=2)              #created matrix and vector are received

    A=A[(group*worker_id):(group*(1+worker_id)),:]      #arrays are split for each worker

    c=np.dot(A,b)
    comm.send(c,dest=0)             # total of each worker sent Worker 0
else:
    A=np.random.randint(1,10,(N,N))
    b=np.random.randint(1,10,(N,1))
    print("N=",N)
    print("A matrix is",np.shape(A))
    print("b vector is",np.shape(b))

    for i in range(1, p):
        comm.send(A,dest=i, tag=1)
        comm.send(b,dest=i, tag=2)        #created matrix and vector are sent to workers

    A=A[(group*worker_id):(group*(1+worker_id))]
    c=np.dot(A,b)

    c_2=np.zeros(group)
    for i in range(1, p):          #matrix multiplication part result are received from workers
        c_2=comm.recv(source=i)
        c=np.append(c,c_2)
    print("c.T prewiev=",c.T)
    print("c vector is", np.shape(c))
    t1=MPI.Wtime()-t0
    print("Timing:", t1)
```

Master node is Worker 0. Inside Worker 0, matrix A and vector b are created and then sent to other workers by comm.send function. To multiple matrix A with vector b with dot product each worker takes N/p number of matrix A. Data transfer made by comm.send and comm.recv functions in MPI. Result from each worker appended and how much time this process takes is showed.

Result example and timing table with explanations:

N=10**4

```
(base) C:\Users\user>mpiexec -n 1 python part_2a.py
N= 10000
A matrix is (10000, 10000)
b vector is (10000, 1)
c vector is (10000, 1)
Timing: 1.5885491000954062

(base) C:\Users\user>mpiexec -n 2 python part_2a.py
N= 10000
A matrix is (10000, 10000)
b vector is (10000, 1)
c vector is (10000,)
Timing: 2.309002500027418

(base) C:\Users\user>mpiexec -n 3 python part_2a.py
N= 10000
A matrix is (10000, 10000)
b vector is (10000, 1)
c vector is (9999,)
Timing: 3.0435213999589905

(base) C:\Users\user>mpiexec -n 4 python part_2a.py
N= 10000
A matrix is (10000, 10000)
b vector is (10000, 1)
c vector is (10000,)
Timing: 3.8313712999224663

(base) C:\Users\user>mpiexec -n 5 python part_2a.py
N= 10000
A matrix is (10000, 10000)
b vector is (10000, 1)
c vector is (10000,)
Timing: 4.615488199982792
```

Size/Workers	1	2	3	4	5
10**2	0.0002	0.0009	0.0012	0.0018	0.0021
10**3	0.01597	0.0235	0.0320	0.0411	0.0495
10**4	1.5886	2.3090	3.0435	3.8314	4.6155

N=10**(2-3-4) are tried.

As size of the vector increases, timing also increases. However, looking at the timings, more processors make calculation slower. It is something that I didn't expected. Probably because of

communication between workers when sending and receiving data cause it. So, here it is better to go without additional processors.

Exercise 3 – Parallel Matrix Operation using MPI

```
import numpy as np
from mpi4py import MPI
```

```
comm=MPI.COMM_WORLD
worker_id=comm.Get_rank()
p=comm.Get_size()
```

```
N=10**2
print("N=",N)
group=round(N/p)

A_2= None
B_2= None
result=np.zeros((N,N))

t0=MPI.Wtime()

if worker_id==0:
    A=np.random.rand(N,N)
    B=np.random.rand(N,N)
    print("A:",A,"Dimensions of A:",np.shape(A))
    print("B:",B,"Dimensions of B:",np.shape(B))
    print("Result=",np.dot(A,B))

for i in range(group):
    for j in range(group):
        print("i=",i,"j=",j)
        if worker_id==0:

            A_2=A[(i*p):((i+1)*p),:]
            B_2=B[:,(j*p):((j+1)*p)]
            B_2=B_2.T.copy()
            print("A_2:",A_2)
            print("B_2:",B_2)
        a_2=np.zeros(N)
        b_2=np.zeros(N)

        comm.Scatter(A_2,a_2,root=0)
        comm.Scatter(B_2,b_2,root=0)
        print("Worker=",worker_id," A",a_2)
        print("Worker=",worker_id," B",b_2)

        part_result=np.dot(a_2,b_2)
        result[(p*i)+worker_id,(p*j)+worker_id]=part_result
        print(result)
        t1=MPI.Wtime()-t0
        print("Timing=",t1)
```

Explanation:

Master node is Worker 0. In Worker 0, matrix A and matrix B are created and split to N/p (size of matrix/number of processor). Each worker assigned to rows of instances(i,j). Matrix B is transposed to transfer correct rows. Using comm.Scatter function reached corresponding rows and columns of matrix A and matrix B results from each worker for each for loop. At the end, to get matrix multiplication in result, dot product is calculated for each i and j.

Result example and timing table with explanations:

Size/Workers	1	2	3	4	5
10**1	0.1666	0.0506	0.0272	0.0218	0.0220
10**2	37.6279	13.2692	7.7210	5.6714	4.3587
10**3	Too long	Couldn't even go to next steps			

N=10**(1-2-3) are tried. 10**3 seems to take for too long as this matrix multiplication is a bit more complex than previous problem. So I couldn't wait after some time. Looking at the table, timing increased with size of N as expected. Timing decreased when number of workers are increased. For 10**1 optimal number of processors are 4, whereas for 10**2 optimal number of processors become 5.