Parallel Programming: Practice 6

May 6 – 10, 2024

Answers must be delivered in a single document (pdf format). This document must include the full name of all group members. If the exercise is an implementation exercise, the source code can be delivered in separate .py files within a compressed archive.

Exercise 1

The primitives of mpi4py have a version with the first letter capitalized that is optimized for data of type *buffer* of Python. A typical example of this type of data are vectors and matrices of numpy.

These routines are much more efficient because they transmit data without converting or serializing as is the case with general Python types. The drawback is that the variable that receives the message must be initialized according to the message, with the same type and number of elements. This is the way other languages used for calculations such as C++ and Fortran work.

In the case of point-to-point communication, the syntax of the uppercase versions differs slightly from the lowercase versions. Below you have the two versions to compare them.

Python Objects

```
comm.send(obj, dest=0, tag=0)
obj = comm.recv(src=0, tag=0)
```

Buffered data (e.g. numpy vectors)

```
comm.Send(sendarray, dest=0, tag=0)
comm.Recv(recvarray, src=0, tag=0)
```

Taking this information into account, we are going to re-implement exercise 2 from the last practice, but in this case the list that we will send will be a numpy vector. The program will have to do the following.

1. Initialize MPI and get the rank of the current process and the total number of processes.
2. If the rank of the current process is 0, it should generate a numpy vector of random integers between 1 and 100 of length n and send it to the process with rank 1.
3. The process with rank 1 must receive the vector of numbers from process 0 and calculate the sum of all the numbers.
4. The process with rank 1 must send this sum back to process 0.
5. Process 0 must receive the sum back and display it on the screen.
6. Processes with rank different from 0 and 1 must wait without doing anything.

- Make two versions, in the first we modify the sending and receiving of the data list so that it is a numpy vector, but we do not modify the sending of the calculated sum. In the second version we will also modify the sending of the calculated sum (in this case we will have to send it as a vector with only 1 element)

Exercise 2

In the same way as happens with point-to-point operations (send and recv), the collective distribution operation "broadcast" also has 2 variants, uppercase and lowercase, depending on the type of data to be transferred.

As in the previous case, the uppercase function is optimized for buffer data (such as Numpy vectors).

Python Objects

datarecv = comm.bcast(datasend, root=0)

Buffered data (e.g. numpy arrays)

comm.Bcast(array, root=0)

To check this improvement we are going to work with a small program with the following characteristics:

1. Initialize MPI and get the rank of the current process and the total number of processes.
2. If the range of the current process is 0, the process must generate a list of random integers between 1 and 100 of length n.
3. Process 0 must broadcast this list to all other processes using the MPI broadcast function.
4. The execution time it takes to perform the MPI broadcast operation must be measured
5. Process 0 will display the time consumed in the operation on the screen.

Remember that to measure time you must use the MPI.Wtime function

- Perform a first version of the exercise using the bcast function (that is, broadcasting a Python list)
- Perform a second version of the exercise using the Bcast function (that is, broadcasting a Numpy vector)
- Run the 2 versions changing the value of n (100,1000,10000,100000,1000000,10000000...) and compare how the times evolve. do you see? how do you explain it? Remember that when working with times you have to make several measurements.

Exercise 3

A very useful primitive when working with vectors in MPI is scatter where the elements of a vector are distributed from one process to all (including the origin).

In the following example, process 0 generates a list of 5 elements and distributes it among the processes.

```
from mpi4py import MPI import
random

comm  =  MPI.COMM_WORLD
rank   =  comm.Get_rank()
size    =  comm.Get_size()

if rank == 0:
        # Data to          to distribute
        n=5
        data       =      [random.randint(1,                  100) for _ in
range(n)]
        print(f"Process {rank} distributes: {data}") else:

        data = None

#Scatter the data data=comm.scatter(data,
root=0)

# Each process prints its part of the data print(f"Process {rank}
receives: {data}")
```

If you run this code you can see that it only works correctly if we launch it with 5 processes. This is because when using the scatter function there must be a correspondence between the number of data to be distributed and the number of processes.

As with the rest of the primitives we have seen so far, scatter also has an uppercase variant that is optimized for buffer data. The syntax of this variant is:

```
comm.Scatter(data, recbuff, root=0)
```

- Modify the previous example so that the list to be distributed among the processes is a Numpy vector and use the Scatter function.
- Modify the size of n (10 instead of 5) and check what happens when you run the example and your version of the exercise. Try different values of n (list size) and different number of processes and indicate in which cases each option works (scatter and Scatter)

Exercise 4

To avoid the problems of the Scatter function, MPI provides the Scatterv function which, in its simplest form, receives a vector with the number of elements in each process:

```
comm.Scatterv([array_global, tamanyos], array_local, root = 0)
```

The size vector must be defined in all processes and indicates the number of elements of the vector that each process receives.

Using Scatterv, a different number of elements can be distributed to each process, which solves the limitation of the previous exercise. The trivial approach, splitting the elements among p − 1 processes and assigning the rest to process p, does not always work well. For example, if we have 101 elements and we distribute them among 11, we will have 10 elements left in all the processes except the last one, which will have only one. This process will end very soon and will hardly participate in the execution of the program, an effect known as unbalanced load.

The best option is to do the integer division by the number of elements n and processes p, and then distribute the rest of the division between the processes. The following expression obtains precisely this:

```
size_local = (n + rank) // p
```

- Modify the previous exercise so that it uses the Scatterv function and can work with different values of n and p. Processor 0 will have to calculate how many elements are going to be distributed to each processor and send it to the rest of the processors.

Exercise 5

The inverse of scatter is the gather operation that combines a vector (or an element) from each process into a single vector in a specific process.

The gather function (like the rest of mpi's functions) has two variants: one in uppercase and one in lowercase. The syntax is as follows:

Python Objects

```
datarecv = comm.gather(datasend, root=0)
```

Buffered data (e.g. numpy arrays)

```
comm.Gather(datasend, datareceived, root=0)
```

- To check the operation of the gather function we are going to work with a small program with the following characteristics:

1. Each MPI process generates a random integer (between 1 and 100).

2. Each MPI process calculates the square of the generated random number.

3. All MPI processes send the squares of the generated numbers to the process with rank 0.

4. The process with rank 0 adds all the squares received to obtain the total sum of the squares of the numbers generated by all processes.

5. The process with rank 0 prints the total sum of the squares.

Exercise 6

The objective of the previous exercise is to reduce the data that has been generated in the different processes to a single value that will be in process 0.

- Optimize the code from the previous exercise taking into account the final objective of the exercise.

Exercise 7

As we have mentioned previously, with gather each process can send a single piece of data to the destination process or more than one piece of data.

In the following code each process sends a list with 5 elements.

```
from mpi4py import MPI import
                random
comm  =  MPI.COMM_WORLD
rank  =  comm.Get_rank()
size  =  comm.Get_size()


# Number of elements that each process will generate local_size = 5



# Generate random data in each process local_data =[random.randint(1, 100) for _
in range(local_size)]

print(f"Process {rank} sends: {local_data}")


# Gather data from all processes in the root process

global_data=comm.gather(local_data, root=0)


# Print the results in the root process
```

```
if rank == 0:
        print("Global data gathered desde there processes:")
        print(global_data)
```

- Modify the previous exercise so that each process sends a numpy array and therefore the Gather function (with capital letters) has to be used. Remember that in these versions you have to initialize the variable that receives the message (in this case global_data)

- Compare the results obtained with both versions. What differences do you observe?