# Parallel Programming: Practice 5

22 - 26 April 2024

The answers must be submitted in a single document (pdf format). This document must include the full name of all members of the group. If the exercise is an implementation exercise, the source code can be submitted as separate .py files in a zipped archive.

## Installing mpi4py

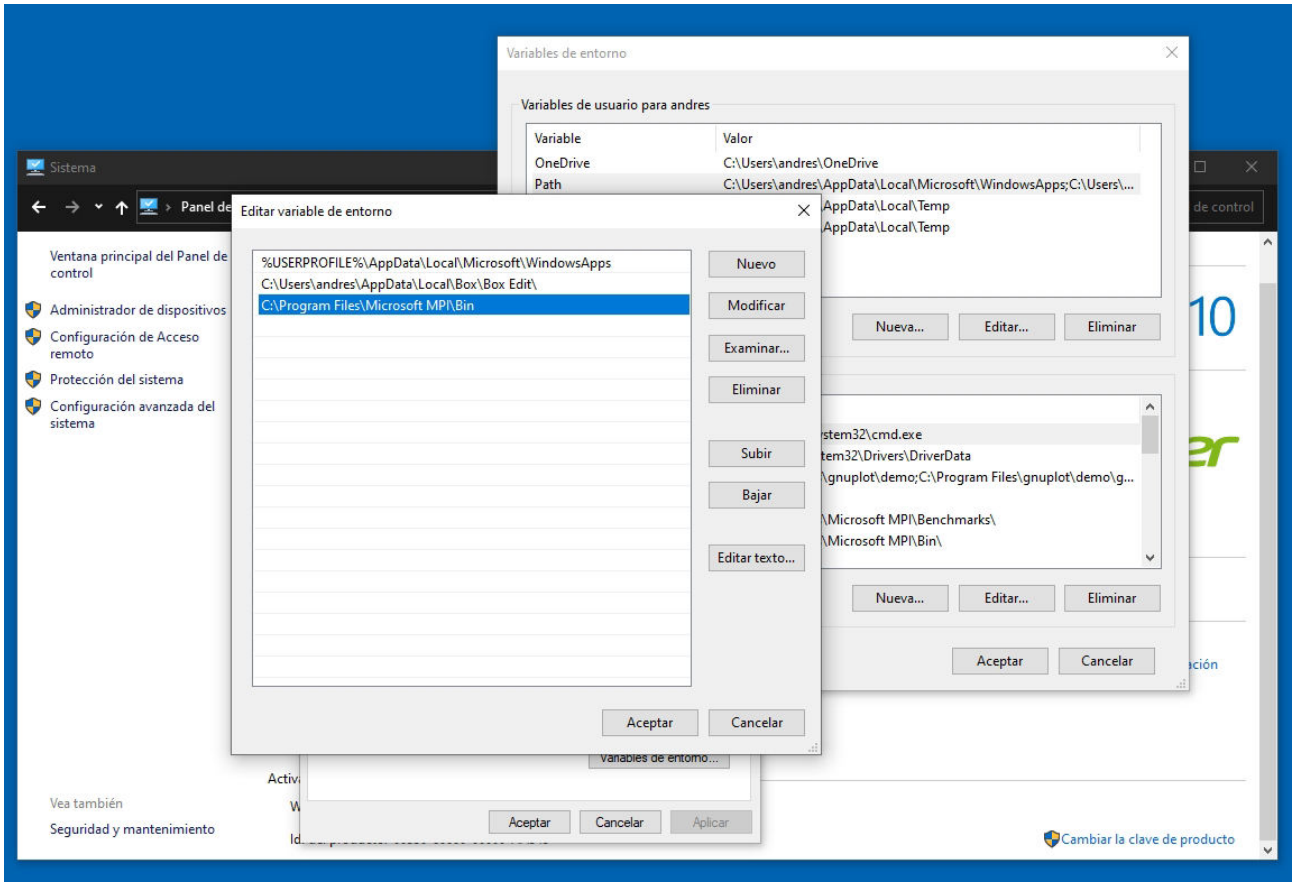
In this practice we are going to use the `mpi4py` interface with the MPI standard for inter-process message switching. First we need to install an MPI implementation on the system and then the `mpi4py` Python library.

## Installation on Windows

Microsoft provides an implementation of MPI that can be downloaded from `https://www.microsoft.com/en-us/download/details.aspx?id=100593`. Attention: you must install the package with the `.exe` extension.

The MPI programmes must be added to the system path, to do this go to:

1. Control panel
2. System icon
3. Advanced system configuration
4. Environment variables
5. Select the `PATH` variable and press the edit button.
6. Click on browse and choose the `bin` directory within the MPI installation (usually `C:\Program Files\Microsoft MPI Bin`).




To install `mpi4py` in Anaconda, as there is no precompiled package, you have to use `pip` from the Anaconda console:

```
pip install mpi4py
```

Depending on whether Anaconda was installed with administrator privileges, the above command may require the same privileges. To run the console as administrator, right-click and choose the option from the drop-down menu.

## Installation on Linux and Mac OS X

In Linux distributions there is usually a `python-mpi4py` package that can be installed with `apt-get` or the operating system's package manager. If Python is installed with Anaconda, just install the package:

```
conda install -c anaconda mpi4py
```

The same command should work on Mac OS X systems. If it doesn't work you can try

```
pip install mpi4py
```

## Launch of MPI programmes

MPI programs must be launched with the `mpiexec` command from the system console (on Windows it may be necessary to use the Anaconda console). Example:

```
mpiexec -np 4 python axis1.py
```

The -np parameter indicates the number of processes to start, depending on the MPI version you may need to add the --oversubscribe parameter to be able to run more processes than the processor has cores. Also depending on your installation you may need to specify the Python version using the python3 command instead of python.

## Exercise 1

All `mpi4py` programmes have a similar start:

```
from mpi4py import MPI

comm = MPI.COMM_WORLD

rank = comm.Get_rank()

size = comm.Get_size()

print("Hello world, I am the process ", rank, " of ",
size)
```

Communications in MPI are done in groups and `MPI.COMM_WORLD` is the default group with all processes. In Python, each group is an object and operations on a group are methods of that object. The `Get_rank` and `Get_size` methods return the process number within the group (starting at zero) and the size of the group.

- Launch the above program several times, varying the number of processes.
- Modify the program so that only the first and last processes display the message.

## Exercise 2

Inter-process messaging is done with `send` and `recv`. By default, sending is asynchronous but receiving is blocking. In the following program process 0 sends a message to the rest of the processes

```
from mpi4py import MPI


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

```
if rank == 0:
   message_sent = "Message from 0" else:
   sent_message = ""

if rank == 0:
   for i in range(1,size): print("Sending
       to", i) comm.send(sent_message,
       dest=i)
else:
   message_received = comm.recv()
   print("Received by", rank, ":", message_received)
```

Run the above code to familiarise yourself with the send and recv primitives. Once you are clear on how they work, do the following exercise.

- Write a Python program using MPI that performs the following task:
  1. Initialise 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 must generate a list 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 list of numbers from process 0 and calculate the sum of all 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 a rank other than 0 and 1 must wait and do nothing.

Note: You can use the Python random library to generate random numbers.

## Exercise 3

The MPI.Wtime() function is a timing function provided by MPI (Message Passing Interface) to measure time in seconds. This function is commonly used in MPI programs to record the execution time of sections of code, operations of

communication, o any other task that is desired to be measured. It would work as follows:

```
start_time = MPI.Wtime()

# Section of code to measure

end_time = MPI.Wtime()

elapsed_time = end_time - start_time

print("Elapsed Elapsed time:", elapsed_time,
"seconds")
```

- Modify the previous exercise so that all processors other than 0 receive the list of random numbers. In addition, we are going to measure the time it takes to carry out the communication operations, in order to be able to compare it later, for this we will use the MPI.Wtime function. The communication time will be measured and shown on the screen by processor 0 (which is the one that carries out all the sending). Specifically, the exercise will perform the following tasks:
  1. Initialise MPI and get the rank of the current process and the total number of processes.
  2. If the current process range is 0, the process must generate a list of random integers between 1 and 100 of length n (e.g. 1000).
  3. Process 0 must measure the execution time to send this list to all other processes using point-to-point communications (send/recv) and display it on the screen.
  4. Processes with a rank other than 0 must receive the list generated by process 0.
  5. Each process must calculate the sum of the numbers received and display it on the screen.

## Exercise 4

In the previous exercises we have sent a message from process 0 to all the others, a broadcast operation. This operation is very common and there is a primitive to perform it efficiently. In general, broadcasting is implemented in a fairly sophisticated way (using a message tree) and taking advantage of the characteristics of the communication network. Broadcasting is very easy with MPI, you just have to

specify the origin with the `root` parameter.

```
received_message = comm.bcast(sent_message, root=0)
```

Note that unlike `send` and `recv`, here all processes have to execute the same call to `bcast`.

- Modify the above program to use `bcast()` instead of `send()` and `recv()`. Remember to measure the communication time with the MPI.Wtime() primitive.
- Compare the times of the previous exercise and this one, which is faster? Note: to be able to compare, the length of the list to be sent must be the same in both cases. Try different sizes of n and different number of processes.

## Exercise 5

Another useful collective operation is reduction, a kind of inverse of diffusion. While in diffusion we send an identical piece of data to all processes, in reduction each process sends a different piece of data and only one process receives it. This received data is a combination of the sent data, such as the sum.

The reduction is done in MPI with the `reduce` primitive similar to `bcast`. In this case the root parameter indicates the destination instead of the source.

The following code shows an example of using the reduce operation, in which all processors generate a random number and send it to processor 0 making the sum of all of them.

```
from mpi4py import MPI
import random

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

number = random.random()
print("Processor", rank, ":", number)

value = comm.reduce(number, root=0)

print("Value ", value)
```

The operation with which messages are combined in the reduction is specified with the parameter `op`. The default operation is addition (so if the op parameter is not specified an addition will be made) and in addition the following operations are available:

| | |
|---|---|
| MPI.MAX | Returns the maximum |
| MPI.MIN | Returns the minimum |
| MPI.SUM | Add up the elements |
| MPI.PROD | Multiply the elements |
| MPI.LAND | And logical |
| MPI.LOR | O logical |
| MPI.BAND | And bit by bit |
| MPI.BOR | O bit by bit |
| MPI.MAXLOC | Returns the maximum and an associated number (index). |
| MPI.MINLOC | Returns the minimum and an associated number (index). |

Run the above code to familiarise yourself with the reduce primitive. Once you are clear on how it works, do the following exercise (you can use the previous exercises as a basis)

- Write a Python program using MPI that performs the following task:
  1. Initialise MPI and get the rank of the current process and the total number of processes.
  2. If the current process range is 0, the process shall generate a list of random integers between 1 and 100 of length n.
  3. Process 0 shall transmit this list to all other processes using the MPI broadcast function.
  4. All processes must calculate the sum of the numbers received.
  5. Process 0 must compare execution time to receive the partial sums calculated by the other processors using point-to-point communications (send/recv) and using the MPI reduction function.
  6. Process 0 must show the sums calculated with both point-to-point communications and the MPI reduction function, together with the execution time of both communication operations.

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

• Compare the times of receiving the sums with point-to-point communication and with collective communication, which is faster? Try different n-sizes and different number of processes.