# HPC for numerical methods and data analysis

*Fall Semester 2023*

*Prof. Laura Grigori*

*Assistant: Tommaso Vanzan*

*Prepared by: Mariana Martinez Aguilar*

**Session 2 – September 26, 2023**

# Communication in Python and MPI

### Exercise I Reminder of a simple MPI code in Python

Given two vectors, $b, c$ we want to compute $d = 2b + c$. Execute the following simple code on 2 processors several times.

```
from mpi4py import MPI
import numpy as np

b = np.array([1, 2, 3, 4])
c = np.array([5, 6, 7, 8])
a = np.zeros_like(b)
d = np.zeros_like(b)

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

if rank == 0:
    for i in range(4):
        a[i] = b[i] + c[i]
    comm.Send(a, dest = 1, tag = 77)
else:
    comm.Recv(a, source = 0, tag = 77)
    for i in range(4):
        d[i] = a[i] + b[i]

print("I am rank = ", rank )
print("d: ", d)
```

Observe the order in which the prints take place and the value of $d$ at the end.

**Exercise II Point to point communication - blocking and non-blocking communication**

a) Provide a brief definition of MPI. What is a communicator?

b) Execute the following simple code on 4 processors.

```
from mpi4py import MPI
import numpy as np

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    print("From process: ", rank, "\n data sent:", data, "\n")
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
    print("From process: ", rank, "\n data received:", data, "\n")
elif rank == 2:
    data = np.array([1, 1, 1, 1, 1])
    print("From process: ", rank, "\n data sent:", data, "\n")
    comm.send(data, dest=3, tag = 66)
else:
    data = comm.recv(source = 2, tag = 66)
    print("From process: ", rank, "\n data received:", data, "\n")
```

In this case, why do we need to be careful when specifying the `dest` and `tag` parameters on both `comm.send` and `comm.recv`?

c) Describe the difference between blocking communication and non-blocking communication in MPI. Modify the code above such that it uses `comm.isend` instead of `comm.send` and `comm.irecv` instead of `comm.recv` while ensuring the messages are passed correctly.

**Exercise III Collective communication - scattering and broadcasting**

a) Run the following script on 4 processors:

```
from mpi4py import MPI
import numpy as np

# Initialize MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

```
size = comm.Get_size()

# Define the vector
if rank == 0:
    vector = np.array([16, 62, 97, 25])
else:
    vector = None

data1 = comm.bcast(vector, root = 0)
data2 = comm.scatter(vector, root = 0)

print("rank: ", rank, " data1: ", data1, " data2: ", data2)
```

What is the difference in MPI between scattering and broadcasting?

b) Consider the multiplication of a matrix $A \in \mathbb{R}^{m\times n}$ with a vector $v \in \mathbb{R}^n$. Write a Python file containing a script that:

- Creates a matrix of dimension $m \times n$
- Creates a vector of dimension $n$
- Makes sure that the dimensions of the matrix and the vector agree in such way that we can compute $Av$
- Computes $Av$ using MPI's scattering, make sure you execute your code on the right amount of processors (*Hints: you'll need to use* `comm.gather`. *What are the entries of $Av$?*)

**Exercise IV Collective communication - all-to-all and reduce**

- Run the following code on 4 processors:

```
from mpi4py import MPI
import numpy as np

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

senddata = rank*np.ones(size, dtype = int)

recvdata = comm.alltoall(senddata)

print(" process ", rank, " sending ", senddata, " receiving ", recvdata )
```

What is `comm.alltoall` doing? Compare it to `comm.scatter`.

- In this exercise we are going to use reduction operations on MPI. Run the following code on 4 processors:

```
from mpi4py import MPI
import numpy as np

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

senddata = rank*np.ones(size, dtype = int)

global_result1 = comm.reduce(senddata, op = MPI.SUM, root = 0)
global_result2 = comm.reduce(rank, op = MPI.MAX, root = 0)

#Print
print(" process ", rank, " sending ", senddata)

#Print the result on the root process
if rank == 0:
    print(" Reduction operation1: ", global_result1,
          "\n Reduction operation2: ", global_result2)
```

  What is a reduction operation? What is the difference between this and `comm.gather`?

- In the previous code, change `comm.reduce` to `comm.allreduce`. What is the difference between the two? (Note, `comm.allreduce` doesn't use the argument `root`).

**Exercise V Deciding what to use - Mid point rule**

Numerical integration describes a family of algorithms for calculating the value of definite integrals. One of the simplest algorithms to do so is called the Mid Point Rule. Assume that $f(x)$ is continous on $[a, b]$. Let $n$ be a positive integer and $h = (b-a)/n$. If $[a, b]$ is divided into $n$ subintervals, $\{x_0, x_1, ..., x_{n-1}\}$, then if $m_i = (x_i + x_{i+1})/2$ is the midpoint of the i-th subinterval, set:

$$M_n = \sum_{i=1}^{n} f(m_i)h.$$

Then:

$$\lim_{n\to\infty} M_n = \int_a^b f(x)dx.$$

Thus, for a fixed $n$, we can approximate this integral as:

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} f(m_i)h$$

Set $n = s * 500$, $f(x) = \cos(x)$, $a = 0$, $b = \pi/2$. Write a Python script such that:

- Defines a function that given $x_i, h, n$ first calculates 500 mid points on a subinterval $[x_i, x_{i+1}]$ and returns the approximation of the integral on this subinterval.
- Using MPI approximates the integral of $f$ on $[a, b]$
- Run your script on $s$ processors

to approximate the integral of $f$. (*Hints: there are many ways of doing this, one approach is using* `comm.bcast` *and* `comm.reduce`).