HPC for numerical methods and data analysis

Fall Semester 2023

Prof. Laura Grigori

Assistant: Mariana Martinez

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?
MPI stands for "Message Passing Interface", it is a standard released in 1994 for message passing library for parallel programs. Communicators are objects that provide the appropriate scope for all communication operations. (Source: https://research.computing.yale.edu/sites/default/files/files

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`?
In this case we need to be careful when specifying `dest` because we have 4 processes. We are sending data from process 0 to process 1 and from process 2 to process 3. We don't want to make sure this is done accordingly. The parameter `tag` acts as a filter and ensures that even if we send two messages they are received correctly.

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.
In blocking communication, the sender or receiver is not able to perform any other actions until the corresponding message has been sent or received (technically, until the buffer is safe to use). This is done with `comm.send` and `comm.recv`. In non-blocking communication, the program is allowed to continue execution while the message is being sent or received. This is achieved with `comm.isend` and `comm.irecv`. These two methods return an instance of the class `Request`. The completion can then be managed using the Test, Wait, and Cancel methods of

this class. As seen below, the method `wait` immediately following the non-blocking methods blocks the process until the corresponding send and receives have completed.

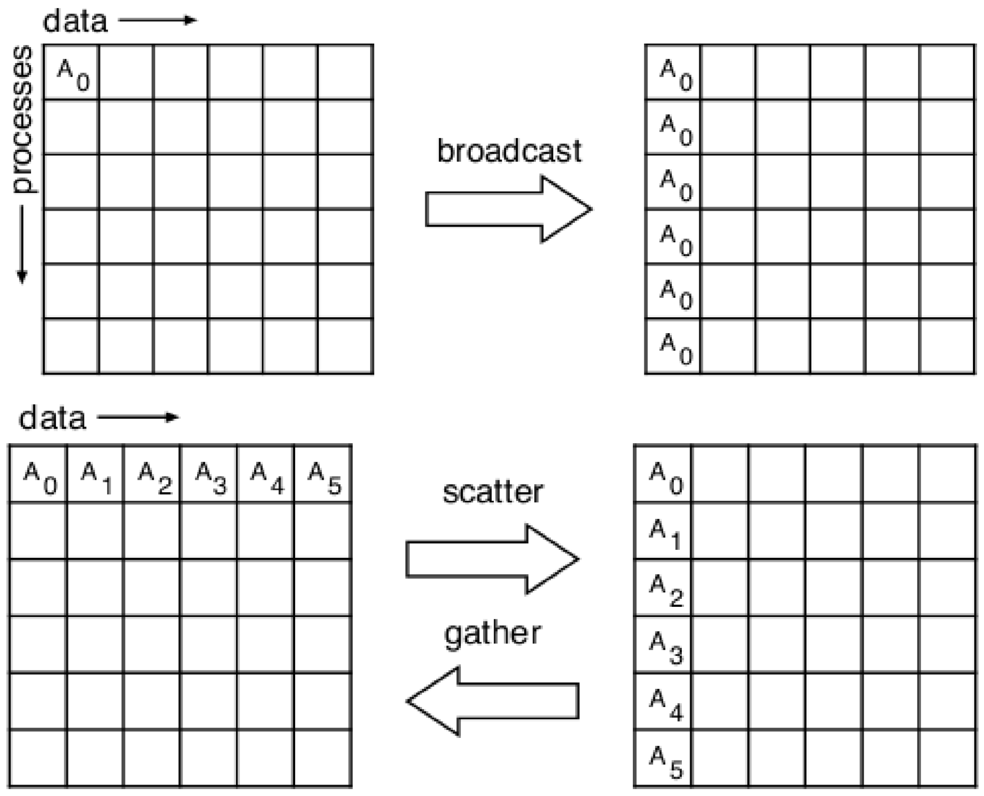
```
# Exercise 2, sending messages to processes
# sources:
#https://mpi4py.readthedocs.io/en/stable/tutorial.html#point-to-point-communication
#https://nyu-cds.github.io/python-mpi/03-nonblocking/

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}
    req = comm.isend(data, dest=1, tag=11)
    req.wait()
    print("From process: ", rank, "\n data sent:", data, "\n")
elif rank == 1:
    req = comm.irecv(source=0, tag=11)
    data = req.wait()
    print("From process: ", rank, "\n data received:", data, "\n")
elif rank == 2:
    data = np.array([1, 1, 1, 1, 1])
    req = comm.isend(data, dest=3, tag = 66)
    print("From process: ", rank, "\n data sent:", data, "\n")
else:
    req = comm.irecv(source = 2, tag = 66)
    data = req.wait()
    print("From process: ", rank, "\n data received:", data, "\n")
```

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?
A broadcast and scatter are two of the standard collective communication techniques. During a broadcast, one process sends the same data to all processes in a communicator. One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes. In comparison, scatter involves a designated root process sending data to all processes in a communicator. Broadcast sends the same piece of data to all processes while scatter sends chunks of an array to different processes. Gather is the inverse of scatter. Instead of spreading elements from one process to many processes, the gather operation takes elements from many processes and gathers them to one single process. Below are some diagrams illustrating these communication techniques.



(Sources: `https://nyu-cds.github.io/python-mpi/05-collectives/`
`https://research.computing.yale.edu/sites/default/files/files/mpi4py.pdf`)

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?*)

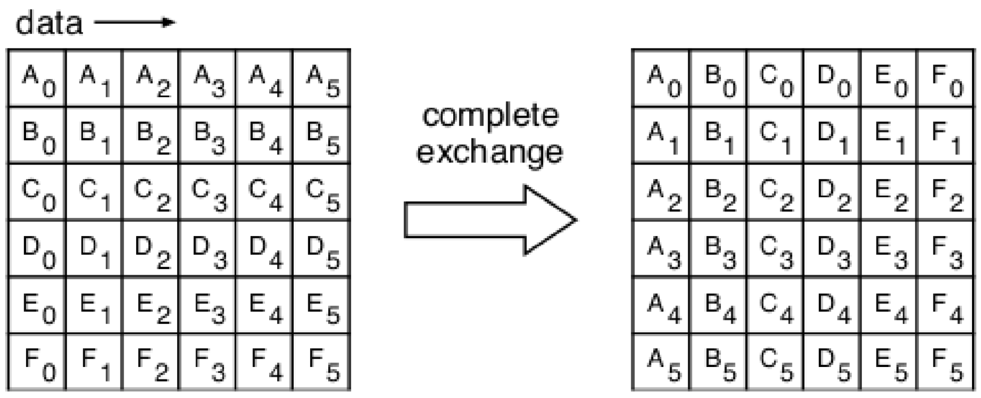
```python
# Generated with ChatGPT (modified)
from mpi4py import MPI
import numpy as np

# Function to perform matrix-vector multiplication
def matrix_vector_multiplication(matrix, vector):
    result = np.dot(matrix, vector)
    return result

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

# Define the matrix and vector
matrix = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
vector = np.array([7, 8, 9, 10])

# Check if the matrix and vector dimensions are compatible
if matrix.shape[1] != len(vector):
    if rank == 0:
        print("Matrix and vector dimensions are not compatible for multiplication.")
elif size != matrix.shape[1]:
    print("Number of processors used doesn't match with number of rows in matrix.")
else:
    # Split the work among processes
    print(" rank: ", rank)
    local_row = comm.scatter(matrix, root=0)

    # Compute local multiplication
    local_result = matrix_vector_multiplication(local_row, vector)

    # Gather results on the root process
    global_result = comm.gather(local_result, root=0)

    # Print the result on the root process
    if rank == 0:
        print("Matrix:")
        print(matrix)
        print("Vector:")
        print(vector)
        print("Result:")
        print(global_result)
        print("Result with numpy: ")
        print( matrix@vector)
```

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`.
  `alltoall` combines the scatter and gather functionality. It is better explained in the diagram below.


data
complete exchange
A_0 A_1 A_2 A_3 A_4 A_5
B_0 B_1 B_2 B_3 B_4 B_5
C_0 C_1 C_2 C_3 C_4 C_5
D_0 D_1 D_2 D_3 D_4 D_5
E_0 E_1 E_2 E_3 E_4 E_5
F_0 F_1 F_2 F_3 F_4 F_5
A_0 B_0 C_0 D_0 E_0 F_0
A_1 B_1 C_1 D_1 E_1 F_1
A_2 B_2 C_2 D_2 E_2 F_2
A_3 B_3 C_3 D_3 E_3 F_3
A_4 B_4 C_4 D_4 E_4 F_4
A_5 B_5 C_5 D_5 E_5 F_5

- 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`?
Data reduction involves reducing a set of numbers into a smaller set of numbers via a function. The `comm.reduce` method takes an array of input elements and returns an array of output elements to the root process. The output elements contain the reduced result. MPI contains a set of common reduction operations that can be used, although custom reduction operations can also be defined. (Source: `https://nyu-cds.github.io/python-mpi/05-collectives/`)

- 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`).
  In `comm.reduce` the output array of the reduction is saved just in the root process but with `comm.allreduce` such output is saved in all the processes.

```
# Exericise 4: All-to-all
# Source: https://subscription.packtpub.com/book/programming/9781785289583/3/ch03lvl1sec49/colle

from mpi4py import MPI
import numpy as np

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

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

global_result1 = comm.reduce(senddata, op = MPI.SUM, root = 0)
global_result2 = comm.reduce(rank, op = MPI.MAX, root = 0)
global_result3 = comm.allreduce(senddata, op = MPI.SUM)
global_result4 = comm.allreduce(rank, op = MPI.MAX)

#Print
print("Process: ", rank,
      " operation1: ", global_result1,
      " operation2: ", global_result2,
      " operation3: ", global_result3,
      " operation4: ", global_result4)
```

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`).

```
# Source: https://nyu-cds.github.io/python-mpi/05-collectives/
# (modified)
import numpy
from math import acos, cos, pi
from mpi4py import MPI


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

def integral(x_i, h, n):
    integ = 0.0
    for j in range(n):
        x_ij = x_i + (j + 0.5) * h
        integ += cos(x_ij) * h
    return integ

a = 0.0
b = pi / 2.0
my_int = 0
integral_sum = numpy.zeros(1)

# Initialize value of n only if this is rank 0
if rank == 0:
    n0 = 500 # default value n = 500
```

```
else:
    n0 = 0 # if not n = 0

# Broadcast n to all processes
n = comm.bcast(n0, root=0)

# Compute partition
h = (b - a) / (n * size) # calculate h *after* we receive n
x_i = a + rank * h * n
my_int = integral(x_i, h, n)

# Send partition back to root process, computing sum across all partitions
print("Process ", rank, " has the partial integral ", my_int)
integral_sum = comm.reduce(my_int, MPI.SUM, root = 0)

# Only print the result in process 0
if rank == 0:
    print('The Integral Sum =', integral_sum)
```