EPFL

HPC for numerical methods and data analysis

Fall Semester 2024

Prof. Laura Grigori

Assistant: Mariana Martínez Aguilar

Session 3 – October 3, 2023

Dense linear algebra and MPI

Exercise I 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 II Reminder of matrix vector multiplication in Python

Suppose that we want to compute the matrix-vector multiplication $y = Ax$, where $A \in \mathbb{R}^{n\times n}$ and $x, y \in \mathbb{R}^n$. If $A^i \in \mathbb{R}^{1\times n}$ is the i-th row of A then the entries of y can be written as inner products:

$$y = Ax = \begin{bmatrix} A^1 \\ A^2 \\ \vdots \\ A^n \end{bmatrix} x = \begin{bmatrix} A^1 x \\ A^2 x \\ \vdots \\ A^n x \end{bmatrix}.$$

If A_i denotes the i-th column of A then the matrix multiplication can be written as the weighted sum of A's columns:

$$y = Ax = \begin{bmatrix} A_1 & A_2 & \dots & A_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \sum_{i=1}^{n} A_i x_i.$$

Note that $A_i \in \mathbb{R}^{n\times 1}$ and $x_i \in \mathbb{R}$, thus $A_i x_i \in \mathbb{R}^{n\times 1}$.

If we have p processors then we can distribute the columns/rows of A in such way that each processor has n/p columns/rows. We call this one dimensional distribution. This is done with `comm.Scatterv`. Then we use `comm.Gatherv` to gather data to one process from all other processes in a group providing different amount of data and displacements at the receiving sides.
Consider the code below (also note that we are printing the time it takes for the code to execute):

```
from mpi4py import MPI
import numpy as np

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

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

```
wt = MPI.Wtime() # We are going to time this

# Define the matrix and vector
cols = 4
rows = 8
num_rows_block = int(rows/size)

matrix = None
vector = None
# Try changing dtype below to see what happens!
global_result = np.empty((rows, 1), dtype = 'int')

if rank == 0:
  matrix = np.array([[1, 2, 3, 4],
                     [5, 6, 7, 8],
                     [9, 10, 11, 12],
                     [13, 14, 15, 16],
                     [17, 18, 19, 20],
                     [21, 22, 23, 24],
                     [25, 26, 27, 28],
                     [29, 30, 31, 32]])
  vector = np.array([7, 8, 9, 10])

# Define the buffer where we are going to receive the block of the matrix
submatrix = np.empty((num_rows_block, cols), dtype='int')
# Scatterv: Scatter Vector, scatter data from one process to all other
# processes in a group providing different amount of data and displacements
# at the sending side
comm.Scatterv(matrix, submatrix, root=0)
vector = comm.bcast(vector, root = 0)

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

# Gather results on the root process
# Gatherv: Gather Vector, gather data to one process from all
# other processes in a group providing different amount of
# data and displacements at the receiving sides
comm.Gatherv(local_result, global_result, root = 0)

# Print the result on the root process
if rank == 0:
    wt = MPI.Wtime() - wt
    print("Matrix:")
    print(matrix)
    print("Vector:")
    print(vector)
    print("Result:")
    print(global_result)
    print("Time taken: ")
    print( wt )
```

a) Is this script distributing A's columns or rows?

b) If your previous answer was "rows" then write a Python script to compute the matrix multiplication Ax but distributing A's columns on different processors. If your previous answer was "columns" then write a Python script to compute the matrix multiplication Ax but distributing A's rows on different processors.

Exercise III 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.