EPFL

HPC for numerical methods and data analysis

Fall Semester 2023

Prof. Laura Grigori

Assistant: Mariana Martinez

Session 3 – October 3, 2023

Dense linear algebra and MPI

Exercise I 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?
It distributes A's 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.

```
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)
num_cols_block = int(cols/size)

matrix = None
matrix_to_send = 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]])
  # There are several ways of sending the matrix in the
  # order you want. This is very important since Python
  # is row major!
  arrs = np.split(matrix, size, axis=1)
  raveled = [np.ravel(arr) for arr in arrs]
  matrix_to_send = np.concatenate(raveled)
  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_cols_block, rows), dtype = 'int')
local_vector = np.empty((num_cols_block, 1), 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_to_send, submatrix, root=0)
comm.Scatterv(vector, local_vector, root=0)
submatrix = np.transpose(submatrix)
print("rank: ", rank, "submatrix: ", submatrix, " local vector: ", local_vector)

# Compute local multiplication
local_result = matrix_vector_multiplication(submatrix, local_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.Reduce(local_result, global_result, op=MPI.SUM, root = 0)

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

Exercise II Splitting communicators

In the previous exercise we splat the matrix in either columns or rows. But we can split such matrix into blocks as well using the `comm`. The `Split` function on MPI. It splits the communicator by color and key.

Every process gets a `color` (a parameter) depending on which communicator they will be. Same color process will end up on the same communicator. In other words, `color` controls the subset assignment, processes with the same color belong to the same new communicator.

The `key` parameter is an indication of the rank each process will get on the new communicator. The process with the lowest key value will get rank 0, the process with the second lowest will get rank 1, and so on. By default, if you don't care about the order of the processes, you can simply pass their rank in the original communicator as key, this way, the processes will retain the same order. In other words, `key` controls the rank assignment.

Run the following script on 4 processors, how is the communicator being split? What is the difference between `new_comm1` and `new_comm2`? In this case, what is `key` doing?

```
from mpi4py import MPI
import numpy as np
```

```
# Testing what comm.Split() does

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

# Defining the subset assignment
if rank%2 == 0:
    color1 = 0
else:
    color1 = 1

if int(rank/2) == 0:
    color2 = 0
else:
    color2 = 1

new_comm1 = comm.Split(color = color1, key = rank)
new_rank1 = new_comm1.Get_rank()
new_size1 = new_comm1.Get_size()

new_comm2 = comm.Split(color = color2, key = rank)
new_rank2 = new_comm2.Get_rank()
new_size2 = new_comm2.Get_size()

print("Original rank: ", rank,
      " color1: ", color1,
      " new rank1: ", new_rank1,
      " color2: ", color2,
      " new rank2: ", new_rank2)
```

The difference is that `new_comm1` *is splitting the communicator according to the ranks' parity. Then* `new_comm2` *is splitting the communicator if the rank is greater than 2 or not. In this case* `key` *is just resetting the ranks in the new sub communicators (going from 0 to the size of the sub communicator).* //

Suppose that you are given a matrix $A \in \mathbb{R}^{2n\times 2n}$, where $n \in \mathbb{N}$:

$$A = \begin{bmatrix} A_0 & A_1 \\ A_2 & A_3 \end{bmatrix},$$

where $A_k \in \mathbb{R}^{n\times n}$. Write a Python script using MPI such that:

- In the root process it defines a matrix $A \in \mathbb{R}^{2n\times 2n}$, with n the number of processors.
- Using `comm.Split` and `comm.Scatter` distributes the matrix into 4 square sub-blocks by first splitting the matrix into columns and then splitting those columns into rows.
- Prints the sub-blocks in the correct sub-communicator.

```
from mpi4py import MPI
import numpy as np

# Testing what comm.Split() does

# Initialize MPI (world)
```

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

# Define the matrix
n_blocks = 2
n = size
npr = 2

matrix = None
matrix_transpose = None

if rank%npr == 0:
  matrix = np.arange(1, n*n + 1, 1, dtype=int)
  matrix = np.reshape(matrix, (n,n))
  arrs = np.split(matrix, n, axis=1)
  raveled = [np.ravel(arr) for arr in arrs]
  matrix_transpose = np.concatenate(raveled)
  print(matrix)


comm_cols = comm.Split(color = rank/npr, key = rank%npr)
comm_rows = comm.Split(color = rank%npr, key = rank/npr)

# Get ranks of subcommunicator
rank_cols = comm_cols.Get_rank()
rank_rows = comm_rows.Get_rank()

# Select columns
submatrix = np.empty((n_blocks, n), dtype = 'int')

# Then we scatter the columns and put them in the right order
receiveMat = np.empty((n_blocks*n), dtype = 'int')
comm_cols.Scatterv(matrix_transpose, receiveMat, root = 0)
subArrs = np.split(receiveMat, n_blocks)
raveled = [np.ravel(arr, order='F') for arr in subArrs]
submatrix = np.ravel(raveled, order = 'F')

# Then we scatter the rows
blockMatrix = np.empty((n_blocks, n_blocks), dtype = 'int')
comm_rows.Scatterv(submatrix, blockMatrix, root = 0)


print("Rank in original communicator: ", rank,
      " color: ", rank, " rank in new subcommunicator: ", rank_cols,
      "submatrix: ", submatrix,
      "block matrix: ", blockMatrix, "\n\n")
```

Hint: if we want to distribute a matrix $A \in \mathbb{R}^{m\times n}$ first by columns and then by rows, we would need to split the communicator twice.

Exercise III 2D distribution for matrix vector multiplication

Consider a matrix $A \in \mathbb{R}^{n\times n}$. We can write this matrix as blocks:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,p} \\ A_{2,1} & A_{2,2} & \dots & A_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p,1} & A_{p,2} & \dots & A_{p,p} \end{bmatrix},$$

where $p \leq n$. With this notation, not all blocks necessarily have the same dimensions. Then we can write the block version of the matrix-vector multiplication:

$$y = Ax = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,p} \\ A_{2,1} & A_{2,2} & \dots & A_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ A_{p,1} & A_{p,2} & \dots & A_{p,p} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{p} A_{1,k}x_k \\ \sum_{k=1}^{p} A_{2,k}x_k \\ \vdots \\ \sum_{k=1}^{p} A_{p,k}x_k \end{bmatrix}.$$

First let $p = 2n$ where n is the number of processors being used. With your answer from the previous exercise, write a Python script such that:

- In the root process defines the matrix A and the vector x
- Using `comm.Split` distributes the blocks of both the matrix and the vector accordingly, the matrix should be split first by columns and then into rows (like on the previous exercise)
- Computes the matrix-vector multiplication using `broadcast`, `scatter`, and/or `reduction`, both on a subset of processors (this is a 2D blocked layout for matrix-vector multiplication)

```python
from mpi4py import MPI
import numpy as np

# 2D distribution for matrix-vector multiplication

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

# Define the matrix
n_blocks = 4
n = size*2
npr = 2
matrix = None
matrix_transpose = None
x = None
y = None
sol = None

if rank%npr == 0:
  matrix = np.arange(1, n*n + 1, 1, dtype=int)
  matrix = np.reshape(matrix, (n,n))
  arrs = np.split(matrix, n, axis=1)
  raveled = [np.ravel(arr) for arr in arrs]
  matrix_transpose = np.concatenate(raveled)
  x = np.arange(1, n+1, 1, dtype = int)
  x = np.reshape(x, (n,1) )
```

```
    sol = np.empty((n,1), dtype = int)

comm_cols = comm.Split(color = rank/npr, key = rank%npr)
comm_rows = comm.Split(color = rank%npr, key = rank/npr)

# Get ranks of subcommunicator
rank_cols = comm_cols.Get_rank()
rank_rows = comm_rows.Get_rank()

### DISTRIBUTE THE MATRIX
# Select columns
submatrix = np.empty((n_blocks, n), dtype = 'int')

# Then we scatter the columns and put them in the right order
receiveMat = np.empty((n_blocks*n), dtype = 'int')
comm_cols.Scatterv(matrix_transpose, receiveMat, root = 0)
subArrs = np.split(receiveMat, n_blocks)
raveled = [np.ravel(arr, order='F') for arr in subArrs]
submatrix = np.ravel(raveled, order = 'F')

# Then we scatter the rows
blockMatrix = np.empty((n_blocks, n_blocks), dtype = 'int')
comm_rows.Scatterv(submatrix, blockMatrix, root = 0)

### DISTRIBUTE X USING COLUMNS
x_block = np.empty((n_blocks, 1), dtype = 'int')
comm_cols.Scatterv(x, x_block, root = 0)

# Multiply in place each block matrix with each x_block
local_mult = blockMatrix@x_block

# Now sum those local multiplications along rows
rowmult = np.empty((n_blocks, 1), dtype = 'int')
comm_cols.Reduce(local_mult, rowmult, op = MPI.SUM, root = 0)

# # Now we gather all of this on the root process of the original comm
if rank_cols == 0:
    comm_rows.Gather(rowmult, sol, root = 0)

# Print in the root process
if(rank == 0):
    print("Solution with MPI: ", np.transpose(sol),
          "Solution with Python: ", np.transpose(matrix@x))
```