
HPC for numerical methods and data analysis

Fall Semester 2024

Prof. Laura Grigori

Assistant: Mariana Martínez Aguilar

Session 4 – October 1st, 2024

Splitting communicators and QR Factorization

Exercise I 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.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_comm0`, `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 == 0:
    color0 = 0
else:
```

```

    color0 = 1
if rank%2 == 0:
    color1 = 0
else:
    color1 = 1
if int(rank/2) == 0:
    color2 = 0
else:
    color2 = 1

new_comm0 = comm.Split(color = color0)
new_rank0 = new_comm0.Get_rank()
new_size0 = new_comm0.Get_size()

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,
      " color0: ", color0,
      " new rank0: ", new_rank0,
      " color1: ", color1,
      " new rank1: ", new_rank1,
      " color2: ", color2,
      " new rank2: ", new_rank2)

```

Exercise II Getting submatrices

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

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

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.

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. Is there another way of doing this?

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)

Exercise IV Reminder of QR

If we recall what a QR factorization is, given a matrix $A \in \mathbb{R}^{m \times n}$, with $\text{rank}(A) = n$, its QR factorization is

$$A = \hat{Q}\hat{R} = [Q \quad \tilde{Q}] \begin{bmatrix} R \\ 0 \end{bmatrix} = QR,$$

where $\hat{Q} \in \mathbb{R}^{m \times n}$ orthogonal and $\hat{R} \in \mathbb{R}^{m \times n}$ upper triangular. Note that A can be seen as a map $W : \mathbb{R}^n \rightarrow \mathbb{R}^m$.

- Using this factorization, state an orthonormal basis for the span of W and one for the nullspace of W .
- Consider the code below, it computes Q without using MPI. Try running the code with the two matrices defined. Do you notice any problems with CGS here? What could be improved when building the projector P ? Compute $\|I - QQ^T\|$, $\kappa(A)$, and $\kappa(Q)$. State the time it takes for this code to run. Compare this implementation with `numpy`'s QR function. What would happen if we just use Python's built in matrix-matrix/vector multiply `@` instead of the user-defined `matrixVectorMultiply` and `matrixMatrixMultiply`?

```
import numpy as np
from numpy.linalg import norm
import time
```

```

# Non parallel implementation of QR algorithm (just get Q)

def matrixVectorMultiply(A, x):
    '''
    Serial implementation of matrix vector multiply
    '''
    m = A.shape[0]
    y = np.zeros((m,), dtype = 'd')
    for i in range(m):
        y[i] = A[i, :]*x
    return y

def matrixMatrixMultiply(A, B):
    '''
    Computes the product C = A*B with outer
    product summation
    '''
    m = A.shape[0]
    n = A.shape[1]
    p = B.shape[1]
    C = np.zeros((m, p), dtype = 'd')
    for i in range(n):
        C += A[:, i]*B[i, :]
    return C

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

# Define the matrix
## TEST1: MATRIX1
size = 4
m = 50*size
n = 20*size
W = np.arange(1, m*n + 1, 1, dtype = 'd')
W = np.reshape(W, (m, n))
W = W + np.eye(m, n) # Make this full rank
# ## TEST2: MATRIX2
# m = 4
# n = 3
# ep = 1e-12
# W = np.array([[1, 1, 1], [ep, 0, 0], [0, ep, 0], [0, 0, ep]])

I = np.eye(m, m, dtype = 'd')
Q = np.zeros((m,n), dtype = 'd')

# First column
qk = W[:, 0]
qk = qk/norm(qk)
Q[:, 0] = qk

# Start iterating through the columns of W
for k in range(1, n):
    ## Build the projector
    # Is there a better way of defining this projector?
    P = I - matrixMatrixMultiply(Q, np.transpose(Q))
    qk = matrixVectorMultiply(P, W[:, k]) # project
    qk = qk/norm(qk) # Normalize
    Q[:, k] = qk

wt = time.time() - wt

```

```

# print(Q)
print("Time taken: ", wt)

wt = time.time()
Q, R = np.linalg.qr(W)
wt = time.time() - wt
print("Time with numpy's QR: ", wt)

```

Exercise V CGS and MPI

Consider the script given above. Which parts could benefit from using MPI? Which information do you need to scatter/broadcast? In this section we are going to implement CGS, this means that for every q_k we need to define the following projector:

$$P_{j-1} = I - Q_{j-1}Q_{j-1}^\top.$$

Notice that because of this, every time we want to project a column of A , A_k we need one synchronization. Take this into consideration for your code. There are different ways of implementing this, below is a rough sketch you could use to guide yourself. Using different values for m and n , compute $\|I - QQ^\top\|$, $\kappa(A)$, and $\kappa(Q)$. State the time it takes for this code to run. Compute the speedup and compare the computation time with `numpy`'s QR function.

```

from mpi4py import MPI
import numpy as np
from numpy.linalg import norm

# CSG (first attempt, just calculate Q)

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

m = 3*size
n = 2*size
local_size = int(m/size) # Dividing by rows

# Define
W = None
Q = None
QT = None
P = None
if rank == 0:
    W = np.arange(1, m*n + 1, 1, dtype = 'd')
    W = np.reshape(W, (m, n))
    W = W + np.eye(m, n) # Make this full rank
    Q = np.zeros((m,n), dtype = 'd')
    QT = np.zeros((n,m), dtype = 'd')
    P = np.eye(m, m, dtype = 'd') # first projector is just I

# In here: we first build Q and then we build R
# Decide what needs to be scattered/broadcast
W_local =
q_local =

```

```

QT_local =
P_local =
W_local =
comm.Scatterv(P, P_local, root = 0)

# For the first column
q_local = P_local@W_local[:, 0]
# Normalize, put this column in Q (and row in QT)

# Start iterating in the columns
for k in range(1, n):
    # We've already built column 0 so we move to column 1
    # First: we must build the projector P, using SUMMA
    localMult = # What needs to go here so that we can do a reduce?
    comm.Reduce(localMult, P, op = MPI.SUM, root = 0) # Projector
    comm.Scatterv(P, P_local, root = 0) # scatter rows of projector
    q_local = P_local@W_local[:, k] # project the k-th column of W
    # Normalize, put this column in Q (and row in QT)
    # Update the part of Q and QT that is in each processor
    comm.Scatterv(QT, QT_local, root = 0)

# Print in rank = 0
if( rank == 0):
    print("Q: \n", Q)

```