EPFL

HPC for numerical methods and data analysis

Fall Semester 2023

Prof. Laura Grigori

Assistant: Mariana Martinez

Session 4 – October 10, 2023

QR Factorization

Exercise 0 Matrix-vector multiplication

If you were not able to finish last week's exercise for parallelized matrix-vector multiplication you can continue doing this exercise (specially if you have questions).

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 exercises, 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 1 Reminder of QR

If we recall what a QR factorization is, given a matrix $W \in \mathbb{R}^{m\times n}$, with $m \geq n$, its QR factorization is

$$W = QR = \begin{bmatrix}\tilde{Q} & \bar{Q}\end{bmatrix}\begin{bmatrix}\tilde{R}\\ 0\end{bmatrix} = \tilde{Q}\tilde{R},$$

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

a) Using this factorization, state an orthonormal basis for the span of W and one for the nullspace of W.
An orthonormal basis is given by the columns of $\tilde{Q}$ *while an orthonormal basis is given by the columns of* $\bar{Q}$.

b) Consider the code below, it computes $\tilde{Q}$ 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 $\|I - \tilde{Q}\tilde{Q}^\top\|$, $\kappa(W)$, and $\kappa(\tilde{Q})$. 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 paralell 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 itarating 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)
```

What could be improved when building the projector P is not using the whole Q matrix on each iteration. Notice that before reaching the last iteration, the last columns of this matrix are just zeros. We could just use the non zero columns when computing the projector. Python's built-in matrix-matrix/vector multiply will make our code faster since it uses optimized rutines.

Exercise 1 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:

$$P_{j-1} = I - \tilde{Q}_{j-1}\tilde{Q}_{j-1}^{\top}.$$

Notice that because of this, every time we want to project a column of W, W_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 $\|I - \tilde{Q}\tilde{Q}^\top\|$, $\kappa(W)$, and $\kappa(\tilde{Q})$. State the time it takes for this code to run. Compute the speedup and compare the computation time with `numpy`'s QR function.

Following the script a solution would be:

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

# CSG

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

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

m = 50*size
n = 20*size
local_size = int(m/size)

# Define
W = None
Q = None
R = None
Qkreceived = 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')
    Qkreceived = np.zeros((m, 1), dtype = 'd')
    R = np.zeros((n,n), dtype = 'd')
    P = np.eye( m, m, dtype = 'd')

# In here: we first build Q and then we build R
W_local = np.zeros((local_size, n), dtype = 'd')
q_local = np.zeros((local_size, 1), dtype = 'd')
QT_local = np.zeros((local_size, m), dtype = 'd')
P_local = np.zeros((local_size, m), dtype = 'd')
W_local = comm.bcast(W, root = 0)
comm.Scatterv(P, P_local, root = 0)

# For the first column
q_local = P_local@W_local[:, 0]

# Normalize
comm.Gather(q_local, Qkreceived, root = 0)
if( rank == 0):
    col = Qkreceived[:, 0]/norm(Qkreceived)
    Q[:, 0] = col
    QT[0, :] = col
comm.Barrier()
```

```
comm.Scatterv(QT, QT_local, root = 0) # We have COLUMNS of Q (or rows of Qt)
# Start interating 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 = 1/size*np.eye(m,m) - np.transpose(QT_local)@QT_local
    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]
    # Normalize
    comm.Gather(q_local, Qkreceived, root = 0)
    if(rank == 0):
        col = Qkreceived[:, 0]/norm(Qkreceived)
        Q[:, k] = col
        QT[k, :] = col
    comm.Barrier()
    comm.Scatterv(QT, QT_local, root = 0)

# Compute R as R = QtA
W_rows = np.zeros((local_size, n), dtype = 'd')
Q_local = np.zeros((local_size, n), dtype = 'd')
comm.Scatterv(W, W_rows, root = 0)
comm.Scatterv(Q, Q_local, root = 0)
localMult_R = np.transpose(Q_local)@W_rows
comm.Reduce(localMult_R, R, op = MPI.SUM, root = 0)


# Print in rank = 0
if( rank == 0):
    wt = MPI.Wtime() - wt
    #print("Q: \n", Q)
    #print("R: \n", R)
    print("Time taken: ", wt)
```

This is still slower than numpy's QR decomposition.