EPFL

HPC for numerical methods and data analysis

Fall Semester 2024

Prof. Laura Grigori

Assistant: Mariana Martínez Aguilar

Session 5 – October 8, 2024

TSQR Factorization

For this week you can choose what to do: either these exercises or continue working on your project (or maybe even both).

Exercise 1 CGS and MGS

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 $W : \mathbb{R}^n \to \mathbb{R}^m$. Recall that computing the QR decomposition using CGS is numerically unstable. An alternative algorithm, which is mathematically equivalent is the Modified Gram-Schmidt algorithm (MGS). Define Q_{j-1} as the matrix we get at the j-th step, $Q_{j-1} = \begin{bmatrix}q_1 & q_2 & \dots & q_{j-1}\end{bmatrix}$ and P_j as the projector onto the subspace orthogonal to the column space $\text{col}(Q_{j-1})$. Then we can write this as:

$$P_j = I - Q_{j-1}Q_{j-1}^* = (I - q_{j-1}q_{j-1}^*)\dots(I - q_1q_1^*).$$

a) How many synchronizations do we need for each vector w_j in this case? Why?
Solution: We need j-1 synchronizations since we need these previous vectors.

b) Why is this more stable than CSG? *Solution:* Intuitively, CSG is more unstable because you subtract off the projections of the (k+1)th vector onto de first k vectors. You ensure that this new vector is orthogonal to the input vector in question but fail to ensure that the vectors you get at the end of the process are orthogonal to each other. So MGS is more stable because you do ensure that the vectors you get at the end of the process are orthogonal to each other.

c) Make the necessary modifications to your last week's code to implement MGS. Compare both codes by computing $\|I - \tilde{Q}\tilde{Q}^\top\|$.

Exercise 2 TSQR

Remember that communication refers to messages between processors. In the recent years we've seen trends causing floating point to become faster than communication. This is why it's important to minimize communication when dealing with parallelism. The TSQR, "Tall Skinny QR" algorithm is a communication avoiding algorithm for matrices with many more rows than columns. In this exercise we are going to assume we're using $P = 4$ processors and the matrix we want to factorize is $A \in \mathbb{R}^{m\times n}$ with $m \geq n$. The computation can be expressed as a product of intermediate orthonormal factors in a binary tree like structure. We scatter row wise our matrix A along 4 processors $A_1, A_2, A_3, A_4 \in \mathbb{R}^{m/4\times n}$. At the leaves of the binary tree, we perform in parallel 4 local QR factorizations to get $A_1 = \hat{Q}_1^{(2)}\hat{R}_1^{(2)}, A_2 = \hat{Q}_2^{(2)}\hat{R}_2^{(2)}, A_3 = \hat{Q}_3^{(2)}\hat{R}_3^{(2)}, A_4 = \hat{Q}_4^{(2)}\hat{R}_4^{(2)}$. Here $\hat{Q}_l^{(2)} \in \mathbb{R}^{m/4\times m/4}$ and $\hat{R}_l^{(2)} \in \mathbb{R}^{m/4\times n}$. In block structure we get

$$\begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix} = \begin{bmatrix} \hat{Q}_1^{(2)} & & & \\ & \hat{Q}_2^{(2)} & & \\ & & \hat{Q}_3^{(2)} & \\ & & & \hat{Q}_4^{(2)} \end{bmatrix} \begin{bmatrix} \hat{R}_1^{(2)} \\ \hat{R}_2^{(2)} \\ \hat{R}_3^{(2)} \\ \hat{R}_4^{(2)} \end{bmatrix}$$

Recall that $\hat{R}_l^{(2)} \in \mathbb{R}^{m/4\times n}$ are tall and skinny upper triangular matrices, hence they can be written as

$$\hat{R}_l^{(2)} = \begin{bmatrix} R_l^{(2)} \\ 0 \end{bmatrix},$$

where $R_l^{(2)} \in \mathbb{R}^{n\times n}$. In the second level of the binary tree we combine the upper triangular matrices $R_1^{(2)}$ with $R_2^{(2)}$ and $R_3^{(2)}$ with $R_4^{(2)}$ to get block structured matrices. We perform the QR factorization in parallel to get the following structured matrices

$$\begin{bmatrix} R_1^{(2)} \\ R_{(2)}^{(2)} \end{bmatrix} = \begin{bmatrix} \hat{Q}_{11}^{(1)} & \hat{Q}_{12}^{(1)} \\ \hat{Q}_{21}^{(1)} & \hat{Q}_{22}^{(1)} \end{bmatrix} \begin{bmatrix} R_1^{(1)} \\ 0 \end{bmatrix} \qquad \begin{bmatrix} R_3^{(2)} \\ R_{(4)}^{(2)} \end{bmatrix} = \begin{bmatrix} \hat{Q}_{33}^{(1)} & \hat{Q}_{34}^{(1)} \\ \hat{Q}_{43}^{(1)} & \hat{Q}_{44}^{(1)} \end{bmatrix} \begin{bmatrix} R_3^{(1)} \\ 0 \end{bmatrix}.$$

This can be written as

$$\begin{bmatrix} \hat{R}_1^{(2)} \\ \hat{R}_2^{(2)} \\ \hat{R}_3^{(2)} \\ \hat{R}_4^{(2)} \end{bmatrix} = \begin{bmatrix} \hat{Q}_{11}^{(1)} & & \hat{Q}_{12}^{(1)} & & & & & \\ & I & & & & & & \\ \hat{Q}_{21}^{(1)} & & \hat{Q}_{22}^{(1)} & & & & & \\ & & & I & & & & \\ & & & & \hat{Q}_{33}^{(1)} & & \hat{Q}_{34}^{(1)} & \\ & & & & & I & & \\ & & & & \hat{Q}_{43}^{(1)} & & \hat{Q}_{44}^{(1)} & \\ & & & & & & & I \end{bmatrix} \begin{bmatrix} R_1^{(1)} \\ 0 \\ R_3^{(1)} \\ 0 \end{bmatrix}.$$

Finally at the root of the tree we compute the last QR factorization

$$\begin{bmatrix} R_1^{(1)} \\ R_3^{(1)} \end{bmatrix} = \begin{bmatrix} \hat{Q}_{11}^{(0)} & \hat{Q}_{13}^{(0)} \\ \hat{Q}_{31}^{(0)} & \hat{Q}_{33}^{(0)} \end{bmatrix} \begin{bmatrix} R_1^0 \\ 0 \end{bmatrix}$$

Which can be written as

$$\begin{bmatrix} R_1^{(1)} \\ 0 \\ R_{(3)}^{(1)} \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{Q}_{11}^{(0)} & & \hat{Q}_{13}^{(0)} & \\ & I & & \\ \hat{Q}_{31}^{(0)} & & \hat{Q}_{33}^{(0)} & \\ & & & I \end{bmatrix} \begin{bmatrix} R_1^{(0)} \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

$\hat{Q}$ is represented implicitly as the product of the intermediate orthogonal matrices.

a) Let $Q \in \mathbb{C}^{a\times b}$ and $S \in \mathbb{C}^{b\times c}$ be matrices such that their columns form an orthonormal set. Show that the columns of QS also form an orthonormal set.
Solution: Notice that Q is not a square matrix and hence we can't talk about its inverse. What we do know is the following:

$$q_i^\top q_j = \delta_{ij},$$

where q_i is the i-th column of Q. We can rewrite this as:

$$Q^\top Q = I.$$

Then for QS:

$$(QS)^\top (QS) = S^\top Q^\top QS = I.$$

b) What are the dimensions of the intermediate orthogonal matrices $\hat{Q}_{ij}^d$? *Solution:* In the first step, the matrices $\hat{Q}_i^2 \in \mathbb{R}^{m/4\times n}$, for the second step we get that $\hat{Q}_{ij}^1 \in \mathbb{R}^{n\times n}$, similarly on the third step $\hat{Q}_{ij}^0 \in \mathbb{R}^{n\times n}$.

c) Show that the columns of the resulting $\hat{Q}$ are orthogonal. How is this matrix computed? *Solution:* Each of these sub blocks is an orthogonal matrix since it was obtained from a QR decomposition. Notice that a diagonal block matrix of orthogonal matrices is orthogonal. Because of the first exercise we have that the product of these matrices is orthogonal as well. The resulting Q is obtained by going through the binary tree in a reverse order. This is part of the project!

d) Suppose that you are given the implicit representation of Q as a product of orthogonal matrices. This representation is a tree of sets of Householder vectors, $\{\hat{Q}_{r,k}^d\}$. Here, k indicates the processor number (both where it is computed and where it is stored), and d indicates the level in the tree. How would you get $\hat{Q}$ explicitly? We only need the "thin" $\hat{Q}$, meaning only its first n columns. Note that we can do this by applying the intermediate factors $\{\hat{Q}_{r,k}^d\}$ to the first n columns of the $m \times m$ identity matrix. Write a Python script using MPI that does this (assume you are using 4 processors). (*Hint: looking at the graphical representation of parallel TSQR might help.*)

Solution: The code below gives an idea of how to build the final representation of Q.

```
from mpi4py import MPI
import numpy as np
from numpy.linalg import norm, qr
from math import log, ceil
from scipy.sparse import block_diag
```

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

# Assuming we have 4 processors and we load the local
# Qks matrices

localQks = None

if rank == 0:
    Q00 = np.array([[-0.04020151,  0.05148748, -0.48794801, -0.50571448],
       [-0.10050378, -0.89895473,  0.31509767, -0.00195979],
       [-0.18090681, -0.28214307, -0.76422519,  0.50179491],
       [-0.26130983, -0.17916811, -0.1463844 , -0.68430465],
       [-0.34171286, -0.07619316, -0.05978922, -0.03687965],
       [-0.42211588,  0.02678179,  0.02680597,  0.01877834],
       [-0.50251891,  0.12975674,  0.11340116,  0.07443633],
       [-0.58292193,  0.2327317 ,  0.19999634,  0.13009432]])
    Q01 = np.array([[-3.44792388e-01, -5.30143546e-01, -3.18019926e-01,
        -1.75412956e-01],
       [ 0.00000000e+00, -7.95057345e-01,  2.76598659e-01,
         1.52190162e-01],
       [ 0.00000000e+00,  0.00000000e+00, -8.89957752e-01,
         1.28586166e-01],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
        -9.59425146e-01],
       [-9.38678970e-01,  1.94730536e-01,  1.16814005e-01,
         6.44320942e-02],
       [ 0.00000000e+00, -2.21159801e-01, -1.29175059e-01,
        -6.98995180e-02],
       [ 0.00000000e+00,  0.00000000e+00, -3.18403475e-14,
        -6.66256004e-15],
       [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00,
         4.27020805e-15]])
    Q02 = np.array([[-3.49273257e-01, -5.53256366e-01, -2.34955891e-01,
        -1.17622003e-01],
       [-0.00000000e+00, -7.70590564e-01,  2.23490598e-01,
         1.11731916e-01],
       [-0.00000000e+00, -0.00000000e+00, -9.36499697e-01,
         6.56458092e-02],
       [-0.00000000e+00, -0.00000000e+00,  0.00000000e+00,
        -9.82321345e-01],
       [-9.37020913e-01,  2.06225550e-01,  8.75794853e-02,
         4.38434399e-02],
       [-0.00000000e+00, -2.39934571e-01, -1.00726527e-01,
        -4.99419515e-02],
       [-0.00000000e+00, -0.00000000e+00,  2.87692119e-14,
        -5.48659862e-15],
       [-0.00000000e+00, -0.00000000e+00, -0.00000000e+00,
         5.21093862e-14]])
    localQks = [Q00, Q01, Q02]
elif rank == 1:
    Q10 = np.array([[-0.24365028, -0.59774677,  0.74934607, -0.04802035],
       [-0.27318365, -0.44629611, -0.40471424,  0.7018256 ],
       [-0.30271702, -0.29484544, -0.35703775, -0.59017753],
       [-0.33225039, -0.14339478, -0.10355031, -0.17789222],
       [-0.36178375,  0.00805589, -0.16647782,  0.05840092],
```

```
       [-0.39131712,  0.15950655, -0.12395086, -0.26421446],
       [-0.42085049,  0.31095722,  0.1295881 ,  0.14087149],
       [-0.45038386,  0.46240788,  0.2767968 ,  0.17920654]])
    localQks = [Q10]
elif rank == 2:
    Q20 = np.array([[-0.28896025, -0.57720763,  0.1853745 ,  0.70741944],
       [-0.30674241, -0.42393232, -0.10403036, -0.44320608],
       [-0.32452458, -0.27065702, -0.44733808, -0.12684962],
       [-0.34230675, -0.11738172,  0.06829868, -0.15861903],
       [-0.36008892,  0.03589358,  0.15636688, -0.3914781 ],
       [-0.37787109,  0.18916889,  0.67210808, -0.04747802],
       [-0.39565326,  0.34244419, -0.52326045,  0.24541494],
       [-0.41343543,  0.49571949, -0.00751925,  0.21479646]])
    Q21 = np.array([[-0.58111405, -0.69239065,  0.02927876, -0.29897038],
       [ 0.        , -0.42767003,  0.28401417,  0.52490052],
       [ 0.        ,  0.        , -0.5112658 , -0.12163304],
       [ 0.        ,  0.        , -0.        , -0.4044181 ],
       [-0.81382213,  0.49440525, -0.02090665,  0.2134814 ],
       [ 0.        , -0.30538007, -0.4979796 ,  0.28838292],
       [ 0.        ,  0.        ,  0.63926928, -0.08516055],
       [ 0.        ,  0.        , -0.        , -0.56635344]])
    localQks = [Q20, Q21]
else:
    Q30 = np.array([[-0.30791323, -0.56732363,  0.31356566,  0.67477033],
       [-0.32061069, -0.4135436 , -0.14240074, -0.43464687],
       [-0.33330814, -0.25976357, -0.3541964 , -0.10517748],
       [-0.34600559, -0.10598354,  0.46022571, -0.36666114],
       [-0.35870305,  0.0477965 , -0.43596128,  0.00108402],
       [-0.3714005 ,  0.20157653, -0.30561576,  0.04499375],
       [-0.38409795,  0.35535656,  0.5086393 , -0.21621866],
       [-0.39679541,  0.5091366 , -0.0442565 ,  0.40185605]])
    localQks = [Q30]

def getQexplicitly(localQks, comm):
    rank = comm.Get_rank()
    size = comm.Get_size()
    m = localQks[0].shape[0]*size
    n = localQks[0].shape[1]
    Q = None
    if rank == 0:
        Q = np.eye(n, n)
        Q = localQks[-1]@Q
        localQks.pop()
    # Start iterating through the tree backwards
    for k in range(ceil(log(size))-1, -1, -1):
        print("k: ", k)
        color = rank%(2**k)
        key = rank//(2**k)
        comm_branch = comm.Split(color = color, key = key)
        rank_branch = comm_branch.Get_rank()
        print("Rank: ", rank, " color: ", color, " new rank: ", rank_branch)
        if( color == 0):
            # We scatter the columns of the Q we have
            Qrows = np.empty((n,n), dtype = 'd')
            comm_branch.Scatterv(Q, Qrows, root = 0)
            # Local multiplication
            print("size of Qrows: ", Qrows.shape)
            Qlocal = localQks[-1]@Qrows
            print("size of Qlocal: ", Qlocal.shape)
```

```
            localQks.pop()
            # Gather
            Q = comm_branch.gather(Qlocal, root = 0)
            if rank == 0:
                Q = np.concatenate(Q, axis = 0)
                print(Q.shape)
        comm_branch.Free()

getQexplicitly(localQks, comm)
```