Project 1: Orthogonalization Techniques for a Set of Vectors

HPC for numerical methods and data analysis

November 5th, 2024

Mathilde Simoni
SCIPER: 371423

0. Introduction

The goal of this project is to implement and compare, using Python and MPI, different algorithms which compute the thin **QR** factorization of a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, written as follows:

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q} \in \mathbb{R}^{m \times n}$ is an orthogonal matrix and $\mathbf{R} \in \mathbb{R}^{n \times n}$ is upper triangular. The number of columns is usually between 50 and a few hundreds and m can be much larger (**A** is called a "tall-and-skinny" matrix).

The thin **QR** factorization is a way to orthogonalize n vectors stored as the columns of **A**. As explained in [1], this is very useful for large-scale Krylov methods, to compute the low rank approximation of a matrix, or to solve least squares problems. For example, a linear least square problem consists in solving:

$$\arg\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|,$$

where $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the coefficient matrix and $\mathbf{b} \in \mathbb{R}^{n}$ a vector. If $\mathbf{A} = \mathbf{QR}$, then:

$$\begin{aligned}\arg\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\| &= \arg\min_{\mathbf{x}} \|\mathbf{QRx} - \mathbf{b}\| \\ &= \arg\min_{\mathbf{x}} \|\mathbf{Q}^T\mathbf{QRx} - \mathbf{Q}^T\mathbf{b}\| \\ &= \arg\min_{\mathbf{x}} \|\mathbf{Rx} - \mathbf{Q}^T\mathbf{b}\|,\end{aligned}$$

where we used the fact that orthogonal matrices maintain norms and $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$. Hence, since **R** is triangular, the system is now easy to solve with, for instance, backward substitution.

1. The Algorithms

The algorithms studied in this project are Classical Gram-Schmidt, Cholesky-QR and TSQR. Details were extracted from the lecture notes [1] and summarized in the following section. Note the use a Matlab-like 1-indexed notation (for an $m \times n$ matrix, rows are indexed from 1 to m and columns from 1 to n).

1.1 Classical Gram Schmidt (CGS)

Classical Gram-Schmidt produces the orthogonal matrix **Q** by a sequence of projections. It processes column by column, left to right, where each successive column vector $\mathbf{A}(:, j)$ is orthogonalized against the previous ones by using a projector P_{j-1} on $\text{span}(\mathbf{Q}(:, 1 : j-1))^{\perp}$. For CGS, it is computed as follows:

$$P_{j-1} = I - \mathbf{Q}(:, 1 : j-1))\mathbf{Q}(:, 1 : j-1))^T.$$

The column j of **Q** can then be computed:

$$\begin{aligned}\mathbf{Q}(:, j) &= P_{j-1}\mathbf{A}(:, j) \\ &= (\mathbf{I} - \mathbf{Q}(:, 1 : j-1)\mathbf{Q}(:, 1 : j-1)^T)\mathbf{A}(:, j) \\ &= \mathbf{A}(:, j) - \mathbf{Q}(:, 1 : j-1)\mathbf{Q}(:, 1 : j-1)^T\mathbf{A}(:, j)\end{aligned} \qquad (1)$$

Finally, $\mathbf{Q}(:,j)$ is normalized to make it a unit vector.

1.2 Cholesky-QR (CholQR)

This method, in contrast to CGS, is not iterative nor projection-based. Recall the cholesky decomposition of a symmetric positive definite matrix $\mathbf{A} = \mathbf{R}^T\mathbf{R}$ where $\mathbf{R}$ is an upper triangular matrix. This method exploits the following fact:

$$\mathbf{A} = \mathbf{QR} \Rightarrow \mathbf{A}^T\mathbf{A} = \mathbf{R}^T\mathbf{Q}^T\mathbf{QR} = \mathbf{R}^T\mathbf{R},$$

since $\mathbf{Q}$ is an orthogonal matrix ($\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}^{-1}\mathbf{Q} = \mathbf{I}$).

Hence, the Cholesky factor of $\mathbf{A}^T\mathbf{A}$ is the triangular factor of the $\mathbf{QR}$ decomposition of $\mathbf{A}$. After having obtained $\mathbf{R}$ this way, it computes the orthogonal factor as $\mathbf{Q} = \mathbf{AR}^{-1}$. Note that this method requires the matrix $\mathbf{A}^T\mathbf{A}$ to be symmetric positive definite and well conditioned for the method to work.

1.3 TSQR

TSQR, if implemented as “sequential”, is the Householder-QR factorization. The parallel version, including the reduction tree and communication routines will be detailed in the next section. Householder-QR can be thought as an orthogonal triangularization as each step uses an orthogonal transformation which annihilates entries in the matrix $\mathbf{A}$ until it becomes the triangular factor $\mathbf{R}$.[1]

Specifically, for each column $\mathbf{A}(:,j)$, it constructs a Householder reflector to zero out all elements below the current diagonal element. It does it using a Householder matrix $\mathbf{H}_j$ and updating $\mathbf{A}(:,j)$ as follows:

$$\begin{aligned}\mathbf{A}(:,j) &= \mathbf{H}_j\mathbf{A}(:,j)\\ &= \left(\mathbf{I}_m - \tau_j y_j y_j^T\right)\mathbf{A}(:,j)\\ &= \begin{bmatrix}\mathbf{A}(1:j-1,j)\\ \pm\left\|\mathbf{A}(j:m,j)\right\|\\ \mathbf{0}_{m-j}\end{bmatrix} \qquad\qquad y_j = \begin{bmatrix}\mathbf{0}_{j-1}\\ \mathbf{A}(1,j)+\operatorname{sgn}(\mathbf{A}(1,j))\ \left\|\mathbf{A}(j:m,j)\right\|_2\\ \mathbf{A}(j+1:m,j)\end{bmatrix}\end{aligned}$$

The matrix $\mathbf{Q}$ is then obtained by multiplying all householder matrices used in the iterative process.

2. Parallelization

In order to parallelize the above algorithms, we distribute the matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ among processors using a block row distribution. If we have P processors, the matrix is decomposed as follows:

$$\mathbf{A} = \begin{bmatrix}\mathbf{A}_1\\ \mathbf{A}_2\\ \dots\\ \mathbf{A}_{P-1}\\ \mathbf{A}_P\end{bmatrix} \tag{2}$$

Note that we use a 1-indexed notation: the P processors are indexed from 1 to P. The same “row distribution” is used for $\mathbf{Q}$ where $\mathbf{Q}_i$ is the local part of $\mathbf{Q}$ on processor i. In the next sections, we follow, we sometimes follow the simpler notation $\mathbf{Q}_i^j = \mathbf{Q}_i(:,j)$ which is the j^{th}column of the portion of $\mathbf{Q}$ stored on processor i.

Assumptions for this project

1) The number of processors P divides the number of rows m. This is to make sure that all the $\mathbf{A}_i$ all have the same number of rows ($\frac{m}{P}$) (as discussed in the lecture, this is allowed).

2) $\frac{m}{n} \geq P$ so that $\mathbf{A}_i \in \mathbb{R}^{\frac{m}{P}\times n}$, the local matrix at processor i, has more rows than columns.

3) The number of processors P is a power of 2: $P = 2, 4, 8, 16, 32, 64....$ This assumption is for the TSQR method to work (check the pseudo-code below).

2.1 Classical Gram-Schmidt (CGS)

The pseudo-codes for Classical Gram-Schmidt are found in Figure 1 (extracted from [1]).

```
Require: $\mathbf{A}$ is an $m \times n$ matrix with $m \geq n$
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular
 1: function $[\mathbf{Q}, \mathbf{R}] = \mathrm{CGS}(\mathbf{A})$
 2:     $\mathbf{R} = \mathbf{0}$
 3:     $\mathbf{R}(1,1) = \|\mathbf{A}(:,1)\|_2$
 4:     $\mathbf{Q}(:,1) = \mathbf{A}(:,1) \;/\; \mathbf{R}(1,1)$
 5:     for $j = 2$ to $n$ do
 6:         $\mathbf{R}(1:j{-}1, j) = \mathbf{Q}(:, 1:j{-}1)^T \cdot \mathbf{A}(:,j)$
 7:         $\mathbf{Q}(:,j) = \mathbf{A}(:,j) - \mathbf{Q}(:,1:j{-}1) \cdot \mathbf{R}(1:j{-}1,j)$
 8:         $\mathbf{R}(j,j) = \|\mathbf{Q}(:,j)\|_2$
 9:         $\mathbf{Q}(:,j) = \mathbf{Q}(:,j) \;/\; \mathbf{R}(j,j)$
10:     end for
11: end function
```

a. Sequential implementation

```
Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over $P$ processors
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ is $m \times n$
Ensure: $\mathbf{R}$ is stored redundantly on all processors and $\mathbf{Q}$'s distribution matches $\mathbf{A}$
 1: function $[\mathbf{Q}, \mathbf{R}] = \text{1D-CGS}(\mathbf{A})$
 2:     $I = \textsc{MyProcID}()$
 3:     $\mathbf{R} = \mathbf{0}$
 4:     $\overline{\beta} = \|\mathbf{A}_I(:,\underline{1})\|_2^2$
 5:     All-reduce $\overline{\beta}$ over all processors, take square root, and store in $\mathbf{R}(1,1)$
 6:     $\mathbf{Q}_I(:,1) = \mathbf{A}_I(:,1) \;/\; \mathbf{R}(1,1)$
 7:     for $j = 2$ to $n$ do
 8:         $\overline{\mathbf{r}} = \mathbf{Q}_I(:,1:j{-}1)^T \cdot \mathbf{A}_I(:,j)$
 9:         All-reduce $\overline{\mathbf{r}}$ over all processors, store in $\mathbf{R}(1:j{-}1,j)$
10:         $\mathbf{Q}_I(:,j) = \mathbf{A}_I(:,j) - \mathbf{Q}_I(:,1:j{-}1) \cdot \mathbf{R}(1:j{-}1,j)$
11:         $\overline{\beta} = \|\mathbf{Q}_I(:,\underline{j})\|_2^2$
12:         All-reduce $\overline{\beta}$ over all processors, take square root, and store in $\mathbf{R}(j,j)$
13:         $\mathbf{Q}_I(:,j) = \mathbf{Q}_I(:,j) \;/\; \mathbf{R}(j,j)$
14:     end for
15: end function
```

b. Parallel implementation

Figure 1: Pseudo-code of parallel Classical Gram-Schmidt

First, note that while $\mathbf{A}$ is row-wise distributed, $\mathbf{R}$ is redundantly stored on all processors. This is not a big issue on performance since $\mathbf{A}$ is "tall-and-skinny" with $n \ll m$ (and $\mathbf{R} \in \mathbb{R}^{n \times n}$).

Suppose we are at iteration j. There are 2 communication routines used:

- Compute the norms to make the columns of $\mathbf{Q}$ unit vectors (lines 11-12, b.). It is done by computing the local $\mathbf{Q}_i(:,j)^T\mathbf{Q}_i(:,j)$ inner products and subsequently calling an *ALL-REDUCE* and square root operation to update $\mathbf{R}$ on each processor.
- Compute the projector and update $\mathbf{Q}_i(:,j)$ with Equation 1 (lines 6-7, a.). An explanation of the parallel computation is given for an example of 4 processors. Equation 1 would be obtained the following way:

$$
\begin{aligned}
\begin{bmatrix} \mathbf{Q}_1^j \\ \mathbf{Q}_2^j \\ \mathbf{Q}_3^j \\ \mathbf{Q}_4^j \end{bmatrix}
&= \begin{bmatrix} \mathbf{A}_1^j \\ \mathbf{A}_2^j \\ \mathbf{A}_3^j \\ \mathbf{A}_4^j \end{bmatrix}
- \begin{bmatrix} \mathbf{Q}_1^{1:j-1} \\ \mathbf{Q}_2^{1:j-1} \\ \mathbf{Q}_3^{1:j-1} \\ \mathbf{Q}_4^{1:j-1} \end{bmatrix}
\begin{bmatrix} \mathbf{Q}_1^{1:j-1} & \mathbf{Q}_2^{1:j-1} & \mathbf{Q}_3^{1:j-1} & \mathbf{Q}_4^{1:j-1} \end{bmatrix}
\begin{bmatrix} \mathbf{A}_1^j \\ \mathbf{A}_2^j \\ \mathbf{A}_3^j \\ \mathbf{A}_4^j \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{A}_1^j \\ \mathbf{A}_2^j \\ \mathbf{A}_3^j \\ \mathbf{A}_4^j \end{bmatrix}
- \begin{bmatrix} \mathbf{Q}_1^{1:j-1} \\ \mathbf{Q}_2^{1:j-1} \\ \mathbf{Q}_3^{1:j-1} \\ \mathbf{Q}_4^{1:j-1} \end{bmatrix}
\begin{bmatrix} \mathbf{Q}_1^{1:j-1}\mathbf{A}_1^j + \mathbf{Q}_2^{1:j-1}\mathbf{A}_2^j + \mathbf{Q}_3^{1:j-1}\mathbf{A}_3^j + \mathbf{Q}_4^{1:j-1}\mathbf{A}_4^j \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{A}_1^j \\ \mathbf{A}_2^j \\ \mathbf{A}_3^j \\ \mathbf{A}_4^j \end{bmatrix}
- \begin{bmatrix} \mathbf{Q}_1^{1:j-1} \\ \mathbf{Q}_2^{1:j-1} \\ \mathbf{Q}_3^{1:j-1} \\ \mathbf{Q}_4^{1:j-1} \end{bmatrix} \mathbf{R}^j
= \begin{bmatrix} \mathbf{A}_1^j\mathbf{Q}_1^{1:j-1}\mathbf{R}^j \\ \mathbf{A}_2^j\mathbf{Q}_2^{1:j-1}\mathbf{R}^j \\ \mathbf{A}_3^j\mathbf{Q}_3^{1:j-1}\mathbf{R}^j \\ \mathbf{A}_4^j\mathbf{Q}_4^{1:j-1}\mathbf{R}^j \end{bmatrix}
\end{aligned}
$$

Thus, only one *ALL-REDUCE* operation is needed to compute $\mathbf{R}^j$, summing all local matrix-vector products $\mathbf{Q}_i^{1:j-1}\mathbf{A}_i^j$ ($\mathbf{A}_i^j$ is a vector) . After that, the update of $\mathbf{Q}_i^j$ can be performed without further communication. Note that the loop runs for n iterations, updating each column of $\mathbf{Q}_i$ iteratively.

2.2 Cholesky-QR (CholQR)

The pseudo-code for this algorithm is found in Figure 2 and was extracted from [1].

```
Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over $P$ processors
Ensure: $\mathbf{QR} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\mathbf{Q}$ is $m \times n$
Ensure: $\mathbf{R}$ is stored redundantly on all processors and $\mathbf{Q}$'s distribution matches $\mathbf{A}$
1: function $[\mathbf{Q}, \mathbf{R}] = \text{ParCholQR}(\mathbf{A})$
2:     $I = \text{MyProcID}()$
3:     $\overline{\mathbf{G}} = \mathbf{A}_I^T \mathbf{A}_I$                    ▷ Local symmetric rank-$k$ update
4:     All-reduce $\overline{\mathbf{G}}$ over all processors, store in $\mathbf{G}$
5:     $\mathbf{R} = \text{Cholesky}(\mathbf{G})$                ▷ Performed redundantly on all processors
6:     $\mathbf{Q}_I = \mathbf{A}_I \mathbf{R}^{-1}$                 ▷ Local triangular solve with multiple RHS
7: end function
```

Figure 2: Pseudo-code of parallel Cholesky-QR

Here, parallelization is exploited to compute the matrix-matricx product $\mathbf{A}^T\mathbf{A}$. For an example of 4 processors:

$$\mathbf{A}^T\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 & \mathbf{A}_3 & \mathbf{A}_4 \end{bmatrix} \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \mathbf{A}_4 \end{bmatrix} = \mathbf{A}_1^T\mathbf{A}_1 + \mathbf{A}_2^T\mathbf{A}_2 + \mathbf{A}_3^T\mathbf{A}_3 + \mathbf{A}_4^T\mathbf{A}_4 \tag{3}$$

Hence, We can perform the local matrix-matrix multiplications $\mathbf{A}_i^T\mathbf{A}_i$ and then summing them with an *ALL-REDUCE* operation. This allows to store the result $\mathbf{A}^T\mathbf{A}$ on all processors (which is of size $(n \times n)$ so it does not take too much memory), find $\mathbf{R}$ with Cholesky, and finally obtain the local $\mathbf{Q}_i$ with $\mathbf{Q}_i = \mathbf{A}_i\mathbf{R}^{-1}$. We observe that this method has a reduced communication cost compared to CGS: there is not main loop ran n times and there is only 1 *ALL-REDUCE* operation.

2.3 TSQR

```
Require: $\mathbf{A}$ is an $m \times n$ matrix 1D-row-distributed over power-of-two $P$ processors
Ensure: $\hat{\mathbf{Q}}\mathbf{R} = \mathbf{A}$ where $\mathbf{R}$ is upper triangular and $\hat{\mathbf{Q}} = \hat{\mathbf{Q}}^{(\log P)} \cdots \hat{\mathbf{Q}}^{(0)}$
Ensure: $\mathbf{R}$ is stored on processor 1 and each $\mathbf{Y}^{(k)}$ is distributed across $2^k$ processors
1: function $\left[\left\{\mathbf{Y}_I^{(k)}\right\}, \mathbf{R}\right] = \text{ParTSQR}(\mathbf{A})$
2:     $I = \text{MyProcID}()$
3:     $\left[\mathbf{Y}_I^{(\log P)}, \mathbf{R}_I^{(\log P)}\right] = \text{HouseholderQR}(\mathbf{A}_I)$  ▷ Eliminate lower triangle of local
   block
4:     for $k = \log P - 1$ down to 0 do
5:         Break if $I$ doesn't have a partner proc
6:         Determine $J$, partner proc ID
7:         if $I > J$ then
8:             Send $\mathbf{R}_I^{(k+1)}$ to processor $J$
9:         else
10:            Receive $\mathbf{R}_J^{(k+1)}$ from processor $J$
11:            $\left[\mathbf{Y}_I^{(k)}, \mathbf{R}_I^{(k)}\right] = \text{HouseholderQR}\left(\begin{bmatrix} \mathbf{R}_I^{(k+1)} \\ \mathbf{R}_J^{(k+1)} \end{bmatrix}\right)$  ▷ Eliminate $J$th triangle
12:        end if
13:    end for
14:    if $I = 1$ then
15:        $\mathbf{R} = \mathbf{R}_1^{(0)}$
16:    end if
17: end function
```

Figure 3: Pseudo code of Parallel TSQR Method

Parallelizing the Householder-QR algorithm is costly as it requires computing norms at each iteration. TSQR offers a communication-efficient alternative, reducing messages from $O(n \log P)$ to $O(\log P)$. The algorithm given in Figure 3 and based on [1] minimizes communication by using the flexibility in the Householder method's order of annihilation.

First, each processor computes a local **QR** factorization of its local matrix $\mathbf{A}_i$. The P local triangular factors are then reduced to a single upper-triangular factor $\mathbf{R}$ through a binomial reduction tree, resulting in only $O(\log P)$ messages.

We illustrate the process using $P = 4$ as an example. The binomial tree is given in Figure 4.

$$\begin{tikzcd}
\mathbf{A}_1 \arrow[r] & \mathbf{R}_1^{(2)} \arrow[dr] & & \\
& & \mathbf{R}_1^{(1)} \arrow[ddr] & \\
\mathbf{A}_2 \arrow[r] & \mathbf{R}_2^{(2)} \arrow[ur] & & \\
& & & \mathbf{R}_1^{(0)} \\
\mathbf{A}_3 \arrow[r] & \mathbf{R}_3^{(2)} \arrow[dr] & & \\
& & \mathbf{R}_3^{(1)} \arrow[uur] & \\
\mathbf{A}_4 \arrow[r] & \mathbf{R}_4^{(2)} \arrow[ur] & &
\end{tikzcd}$$

Figure 4: Binomial reduction tree for$P = 4$

At each iteration, processor I identifies its partner processor J as defined in Figure 4. The processor with the higher ID in the pair sends its local triangular factor, which the partner stacks onto its own to compute a new Householder-QR factorization. Hence, half of the processors drop out at each iteration, which enables this approach to require only a constant number of collective exchanges ($O(\log P)$).

Each local $\mathbf{Q}_i$ at level k factor is stored in a list during the initial traversal of the tree. After reaching the root, the algorithm traverses the tree in reverse to compute the full $\mathbf{Q}_i$ factors by applying the stored local orthogonal factors to the first columns of the identity matrix. Further algebraic details explaining how this is done efficiently (without the need to store the full $\mathbf{Q}$ matrix) are provided in the Appendix.

3. Numerical Stability

Numerical stability can be measured by computing the loss of orthogonality of $\mathbf{Q}$ defined as $\|\mathbf{I} - \mathbf{Q}^T\mathbf{Q}\|_2$ and its condition number, $\kappa(\mathbf{Q})$. Since the goal is to have $\mathbf{Q}^T\mathbf{Q} = \mathbf{I}$ (def of orthogonality), $\|\mathbf{I} - \mathbf{Q}^T\mathbf{Q}\|_2$ should be as small as possible and $\kappa(\mathbf{Q})$ close to 1.Therefore, these quantities measures how close the matrix $\mathbf{Q}$ is from being orthogonal. This bound is important as the loss of orthogonality could affect the behavior of downstream applications using the **QR** decomposition, such as Krylov kubspace methods.

The tests are performed on 2 different matrices:

Matrix C The first matrix named **C** is generated by uniformly discretizing the parametric function f, also used in [2].

$$\mathbf{C} \in \mathbb{R}^{m \times n}, \quad \mathbf{C}(i,j) = f\left(\frac{i-1}{m-1}, \frac{j-1}{n-1}\right), \quad i \in 1, ..., m, \quad j \in 1, ..., n,$$

$$f(x, \mu) = \frac{\sin(10(\mu + x))}{\cos(100(\mu - x)) + 1.1}.$$

We use $m = 50000$ and $n = 600$ (with more than 50000 rows, the sequential algorithm could not run hence comparisons of runtimes could not be performed). This matrix is unfortunately very badly conditioned, with $\kappa(\mathbf{C}) = 6.05e15$. Therefore, CholQR method crashes when computing its **QR** factorization.

Matrix D We obtain the second matrix from the SuiteSparse Matrix Collection, in which it is called *net125*. We load it as a numpy array using the *scipy* library. The matrix originally has 36720 columns and rows. In order to make it "tall-and'-skinny", we only keep 150 columns. In addition, we keep the first 36672 rows to satisfy the condition "$P \mid m$" for $P = 64, 32, 16, 8$. This matrix **D** was chosen so that it is big enough to see the effect of parallelization while $\mathbf{D}^T\mathbf{D}$ is well conditioned and symmetric positive definite, which makes it possible to run Cholesky-QR. Indeed, we have $\kappa(\mathbf{D}^T\mathbf{D}) = 18.60^2 \simeq 346$.

3.1 Theoretical Bounds

The theoretical bounds for loss of orthogonality are stated in Table 1.

	$\| \mathbf{I} - \mathbf{Q}^T\mathbf{Q} \|_2$
CGS	$O(\varepsilon)\kappa^2(\mathbf{A})$
CholQR	$O(\varepsilon)\kappa^2(\mathbf{A})$
TSQR	$O(\varepsilon)$

Table 1: Theoretical bounds for loss of orthogonality

For CGS and CholQR, it is required that $O(\varepsilon)\kappa^2(\mathbf{A}) < 1$. $\kappa(\mathbf{A})$ is the condition number of $\mathbf{A}$ and ε the machine precision.

3.2 Results

For CGS, we compute the loss of orthogonality and condition number of input vectors at each iteration, since the method orthogonalizes the columns of the initial matrix $\mathbf{A}$ one by one (plots are in the Appendix, they were put in log scale for y when the change in value is too big to visualize correctly with a normal scale). Regarding Cholesky-QR and TSQR, we only compute the loss of orthogonality and condition number of $\mathbf{Q}$ at the end. We use $P = 8$ processors for all the tests below.

	CGS	**CholQR**	**TSQR**
$\| \mathbf{I} - \mathbf{Q}^T\mathbf{Q} \|_2$	381.17	/	1.42e-14
$\kappa(\mathbf{Q})$	7.28e16	/	1 + 6.9e-15

Table 2: Numerical stability ($\mathbf{C}$)

	CGS	**CholQR**	**TSQR**
$\| \mathbf{I} - \mathbf{Q}^T\mathbf{Q} \|_2$	3.62e-13	4.56e-14	1.48e-14
$\kappa(\mathbf{Q})$	1 + 5.66e-14	1 + 5.6e-15	1 + 1.6e-15

Table 3: Numerical stability ($\mathbf{D}$)

Observations Regarding $\mathbf{C}$, since $\kappa(\mathbf{C}) = 6.05e15$, then $\kappa^2(\mathbf{C}) = O(10^{30})$ which is above machine precision. Therefore it does not make sense to analyze if the stability satisfies the theoretical bounds for CGS as the requirement $O(\varepsilon)\kappa^2(\mathbf{A}) < 1$ is not satisfied. However, we observe that TSQR obtains very good results even though $\mathbf{C}$ is badly-conditioned, confirming the theoritical bound of $O(\varepsilon)$.

Since $\mathbf{D}$ is well-conditioned, Cholesky-QR does not crash. We observe that the loos of orthogonality is a little bit smaller for TSQR, as it does not depend on the condition number of $\kappa(\mathbf{D})$. However, the difference is small (check plots in the Appendix).

Therefore, in order to see the difference better, we test stability on a third matrix $\mathbf{E}$, with 52288 rows, 500 columns and $\kappa(\mathbf{E}) = 2915.87$. On Table 4, we observe a bigger difference between TSQR and the 2 other methods, which stability depends on $\kappa(\mathbf{E})$. TSQR value on the loss of orthogonality has the same order for $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{E}$, confirming that it does not depend on the condition number of the initial matrices.

	CGS	**CholQR**	**TSQR**
$\| \mathbf{I} - \mathbf{Q}^T\mathbf{Q} \|_2$	4.46e-13	1.52e-12	1.47e-14
$\kappa(\mathbf{Q})$	1 + 2.3e-13	1 + 6.5e-13	1 + 8.2e-15

Table 4: Numerical stability ($\mathbf{E}$)

Summary The poorer stability of CGS and Cholesky-QR aligns with theoretical predictions. For CGS, instability likely arises from how the projector is computed; it can be improved by projecting iteratively on each vector, a method known as Modified Gram-Schmidt, though it is less efficient. For Cholesky-QR, instability rises from explicitly forming $\mathbf{A}^T\mathbf{A}$.

4. Sequential Performance

In this section, the algorithms are ran on 1 core. The following parameters are used:

- **Type of computer**: Helvetios cluster with account “math-505”
- **Python version**: Python 3.9.10 (already installed on the cluster)
- **Used libraries**:
  - **Scipy** (1.13.1): import a matrix of type *.mtx*
  - **Numpy** (2.0.2): to manipulate matrices efficiently

The results for **C** and **D** are presented in Table 5, they are the average times of 4 runs. Note that the matrix **C** now has 49984 rows instead of 50000. This is to compare the sequential runtimes with the parallel runtimes in the next section, based on our initial assumption that the number of rows must divide the number of processors (32 and 64 did not divide 50000).

	CGS	**CholQR**	**TSQR**
C	12.29	/	4.50
D	0.82	0.31	0.65

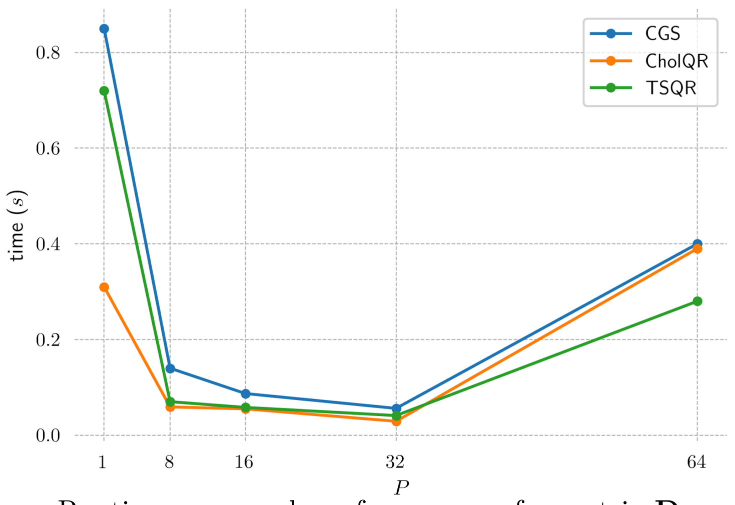
Table 5: Sequential runtimes (s)

Observations First, we observe that the times for **D** are smaller than for **C** which are probably due to **D**’s smaller size (36720 rows vs 49984, 150 columns instead of 600). Indeed, for example, the **R** factors stored redundantly on each processor for CGS and CholQR have sizes $150^2 = 22500$ or $600^2 = 360000$, which is different by a factor of 16.

The faster time of TSQR and Cholesky method make sense are they are implemented using optimized `numpy` functions (TSQR is simply a `np.linalg.qr()` call and Cholesky-QR uses `np.linalg.cholesly()`). Generally, we expect Cholesky-QR to be faster than TSQR, since it uses the fact that $\mathbf{A}^T\mathbf{A}$ is symmetric positive definite, which require fewer operations than the multiples matrix-vector multiplications needed to iteratively zero out elements below the diagonal in Householder-QR (reminder: sequential TSQR is simply Householder-QR)

5. Parallel Performance

The parallel runtimes on helvetios cluster for the 3 algorithms are gathered in Figure 5. The number of processors used is 8 (1 node, 8 tasks, serial mode), 16 (1 node, 16 tasks, serial mode) 32 (1 node, 32 tasks, serial mode) and 64 (2 nodes, 32 tasks, parallel mode). Similarly to the previous section, the values are the average of 4 runs. The parameters (computer, libraries and versions) are similar to the previous section.



Runtimes vs. number of processors for matrix **D**

		CGS	**CholQR**	**TSQR**
C	$P = 8$	2.08	/	0.93
	$P = 16$	1.56	/	0.74
	$P = 32$	1.12	/	0.72
	$P = 64$	0.97	/	1.040
D	$P = 8$	0.14	0.059	0.070
	$P = 16$	0.87	0.055	0.058
	$P = 32$	0.056	0.029	0.041
	$P = 64$	0.40	0.39	0.28

Runtimes for matrices **C** and **D** (s)

Figure 5: Runtimes data

Observations Based on theoretical algorithmic costs (presented in Table 6), we should observe that Cholesky is faster than TSQR and substantially faster than CGS, due to the lower number of needed "words" and "messages" parts communication cost. This is indeed what we observe in Table 6, with CGS taking approximately twice the time to run compared to Cholesky, and TSQR sitting between the 2, close to Cholesky-QR.

Another notable result is the performance drop with $P = 64$ in almost all cases (except CGS with matrix $\mathbf{C}$). This may be due to the fact that 64 processors add excessive communication overhead relative to the matrix size. For bigger matrices, this "plateau" will be reached for a higher P. However, it is still better than the sequential times, except for the cholesky-QR method. This performance drop highlights the importance of carefully selecting the processor count to avoid both under- and over-utilization.

Algorithm	**# flops**	**# words**	**# messages**
CGS	$\frac{2mn^2}{P}$	$O(n^2 + n \log P)$	$O(n \log P)$
Cholesky-QR	$\frac{2mn^2}{P} + \frac{n^3}{3}$	$O(n^2)$	$O(\log P)$
TSQR	$\frac{2mn^2}{P} + \frac{2n^3}{3} \log P$	$O(n^2 \log P)$	$O(\log P)$

Table 6: Algorithmic costs [1]

Note that these runtimes include the time to split the initial $\mathbf{A}$ among processors (since this is a necessary process) but not the time to gather back the $\mathbf{Q}_i$ into $\mathbf{Q}$ (since we discussed in the lecture that is this not necessary).

6. Conclusion

TSQR stands out as an optimal choice, combining numerical stability with low computational cost. Unlike Cholesky-QR, which requires a well-conditioned matrix, TSQR can also handle singular matrices. While the condition number of $\mathbf{A}$ influences that of $\mathbf{R}$, $\mathbf{Q}$ remains nearly perfectly orthogonal.

In essence, TSQR achieves the low communication cost of Cholesky-QR (we have observed that the runtimes for these 2 methods are very close) along with the numerical stability of Householder-QR (the loss of orthonality is $O(\varepsilon)$, not depending on the condition number of $\mathbf{A}$), making it the best choice among the 3 options for parallel applications.

Notes:

1) ChatGPT [3] was used to correct grammar errors and rewrite a few paragraphs to make them more understandable in english (improve formulation), as well as to find suitable matrices on the Suite Space Matrix collection (for example, $\mathbf{D}$ was found as a matrix with a low condition number).

2) The submitted zip file along this report contains 3 python files:
- *"functions.py": contains the implementation of 6 algorithms: CGS, CGS_sequential, cholQR, cholQR_sequential, TSQR, TSQR_sequential*
- *"QR.py": file to run one of the parallel methods (CGS, cholQR or TSQR). It takes as input the method, and the matrix ($\mathbf{C}$, $\mathbf{D}$, or $\mathbf{E}$)*
- *"QR_sequential.py": file to run one of the sequential methods (CGS_sequential, cholQR_sequential or TSQR_sequential). Similarly to "QR.py", it takes as input the method and the matrix.*

The submitted zip file also contains examples of batch files used to run the programs on the helvetios cluster and a Jupyter notebook to produce the plots and download matrices from the suiteSparse collection.

7. Appendix

7.1 CGS: loss of orthogonality and Condition Number as Iterations Go

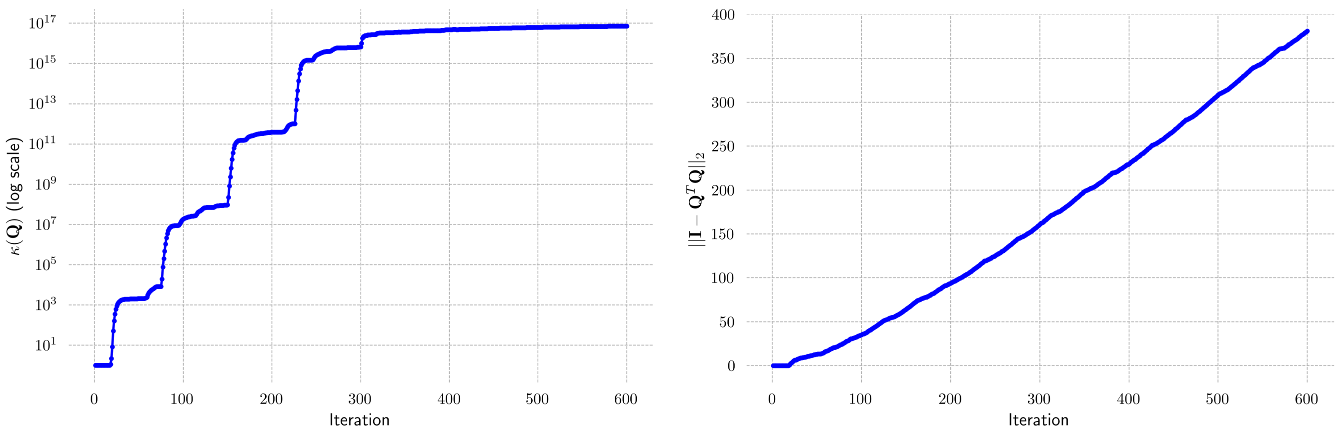


Figure 6: Stability of **C** vs. iteration count

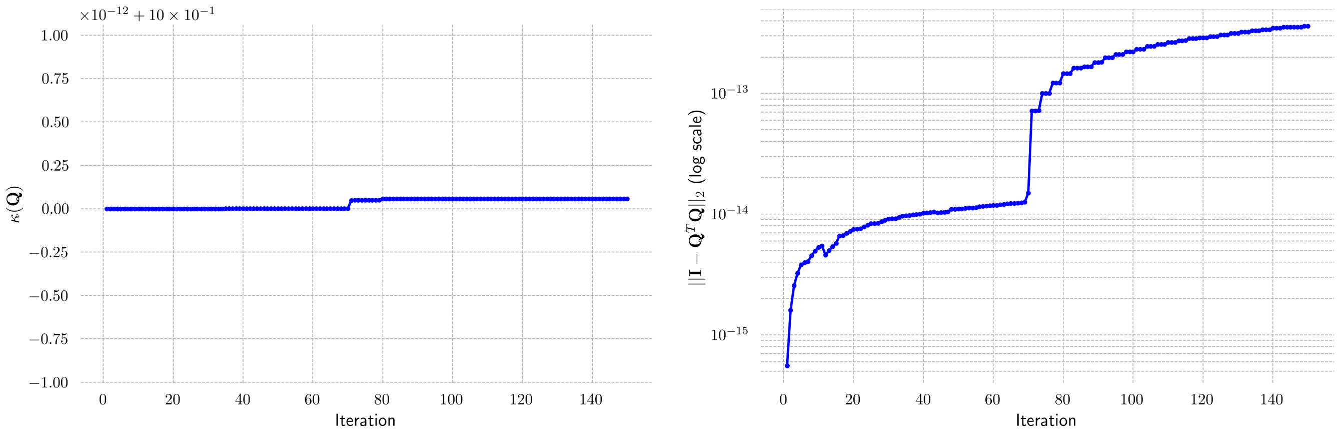


Figure 7: Stability of **D** vs. iteration count

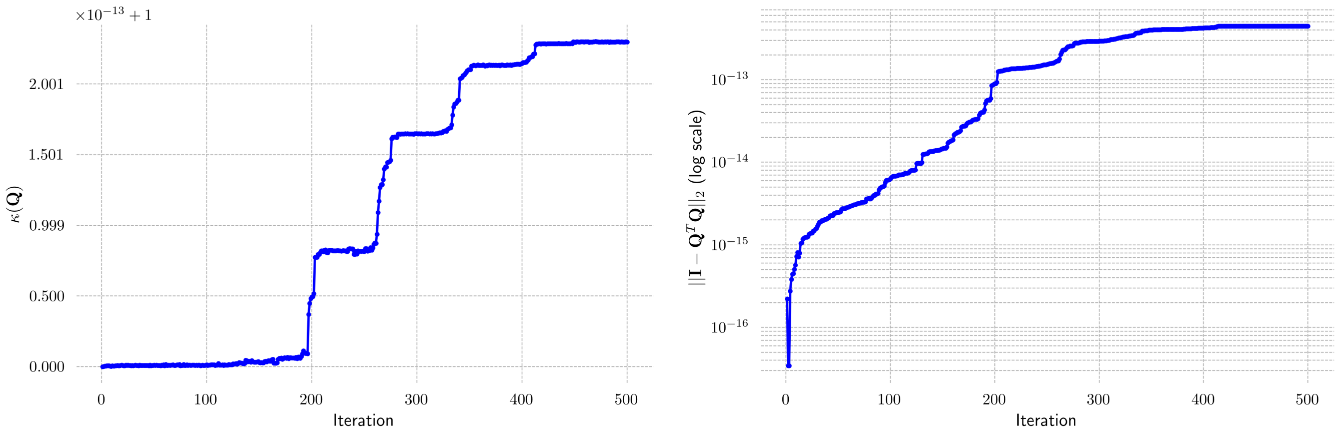


Figure 8: Stability of **E** vs. iteration count

7.2 TSQR: algebra to reconstruct Q

This is an example for $P = 4$ processors. It shows how to reconstruct $\mathbf{Q}$ from the local $\mathbf{Q}_i$ stored at each level of the binomial reduction tree.

Step 0:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \mathbf{A}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_{11}\mathbf{R}_{11} \\ \mathbf{Q}_{21}\mathbf{R}_{21} \\ \mathbf{Q}_{31}\mathbf{R}_{31} \\ \mathbf{Q}_{41}\mathbf{R}_{41} \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_{11} & & & \\ & \mathbf{Q}_{21} & & \\ & & \mathbf{Q}_{31} & \\ & & & \mathbf{Q}_{41} \end{bmatrix} \begin{bmatrix} \mathbf{R}_{11} \\ \mathbf{R}_{21} \\ \mathbf{R}_{31} \\ \mathbf{R}_{41} \end{bmatrix}$$

Step 1:

$$\begin{bmatrix} \mathbf{R}_{11} \\ \mathbf{R}_{21} \end{bmatrix} = \mathbf{Q}_{12}\mathbf{R}_{12}, \qquad \begin{bmatrix} \mathbf{R}_{31} \\ \mathbf{R}_{41} \end{bmatrix} = \mathbf{Q}_{22}\mathbf{R}_{22} \quad \Rightarrow \begin{bmatrix} \mathbf{R}_{11} \\ \mathbf{R}_{21} \\ \mathbf{R}_{31} \\ \mathbf{R}_{41} \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_{12} & \\ & \mathbf{Q}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{R}_{12} \\ \mathbf{R}_{22} \end{bmatrix}$$

Step 2:

$$\begin{bmatrix} \mathbf{R}_{12} \\ \mathbf{R}_{22} \end{bmatrix} = \mathbf{Q}_{13}\mathbf{R}_{13}$$

Assembly:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \mathbf{A}_4 \end{bmatrix} = \begin{bmatrix} \mathbf{Q}_{11} & & & \\ & \mathbf{Q}_{21} & & \\ & & \mathbf{Q}_{31} & \\ & & & \mathbf{Q}_{41} \end{bmatrix} \begin{bmatrix} \mathbf{Q}_{12} & \\ & \mathbf{Q}_{22} \end{bmatrix} \mathbf{Q}_{13}\mathbf{R}_{13}$$

$$\mathbf{R} = \mathbf{R}_{13} \qquad \mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{11} & & & \\ & \mathbf{Q}_{21} & & \\ & & \mathbf{Q}_{31} & \\ & & & \mathbf{Q}_{41} \end{bmatrix} \begin{bmatrix} \mathbf{Q}_{12} & \\ & \mathbf{Q}_{22} \end{bmatrix} \mathbf{Q}_{13}$$

However, it can be observed that the computation of the final $\mathbf{Q}$ does not need to compute the full matrices written above. Indeed, we realize that we can split the intermediate matrices by 2, and traverse the tree in the reverse order with the exact same pairing among processors. This way, we obtain $\mathbf{Q}$ as follows, without having to compute the full $\mathbf{Q}$ explicitely:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_1 \\ \mathbf{Q}_2 \\ \mathbf{Q}_3 \\ \mathbf{Q}_4 \end{bmatrix}$$

Bibliography

[1] L. Grigori, “HPC for Numerical Methods and Data Analysis,” 2024.

[2] O. Balabanov and L. Grigori, “Randomized Gram–Schmidt process with application to GMRES,” *SIAM Journal on Scientific Computing*, vol. 44, no. 3, pp. A1450–A1474, 2022.

[3] OpenAI, “ChatGPT.” [Online]. Available: https://chat.openai.com/