Reproducible Tall-Skinny QR

James Demmel and Hong Diep Nguyen

*University of California, Berkeley, Berkeley, USA

demmel@cs.berkeley.edu, hdnguyen@eecs.berkeley.edu

Abstract*—Reproducibility is the ability to obtain bitwise identical results from different runs of the same program on the same input data, regardless of the available computing resources, or how they are scheduled. Recently, techniques have been proposed to attain reproducibility for BLAS operations [1], [2], [5], all of which rely on reproducibly computing the floating-point sum and dot product. Nonetheless, a reproducible BLAS library does not automatically translate into a reproducible higher-level linear algebra library, especially when communication is optimized. For instance, for the QR factorization, conventional algorithms such as Householder transformation or Gram-Schmidt process can be used to reproducibly factorize a floating-point matrix by fixing the high-level order of computation, for example column-by-column from left to right, and by using reproducible versions of level-1 BLAS operations such as dot product and 2-norm. In a massively parallel environment, those algorithms have high communication cost due to the need for synchronization after each step. The Tall-Skinny QR algorithm obtains much better performance in massively parallel environments by reducing the number of messages by a factor of n to $\mathcal{O}(\log(P))$ where P is the processor count, by reducing the number of reduction operations to $\mathcal{O}(1)$. Those reduction operations however are highly dependent on the network topology, in particular the number of computing nodes, and therefore are difficult to implement reproducibly and with reasonable performance. In this paper we present a new technique to reproducibly compute a QR factorization for a tall skinny matrix, which is based on the Cholesky QR algorithm to attain reproducibility as well as to improve communication cost, and the iterative refinement technique to guarantee the accuracy of the computed results. Our technique exhibits strong scalability in massively parallel environments, and at the same time can provide results of almost the same accuracy as the conventional Householder QR algorithm unless the matrix is extremely badly conditioned, in which case a warning can be given. Initial experimental results in Matlab show that for not too ill-conditioned matrices whose condition number is smaller than sqrt(1/e) where e is the machine epsilon, our technique runs less than 4 times slower than the built-in Matlab `qr()` function, and always computes numerically stable results in terms of column-wise relative error.

I. Introduction

A QR factorization based on the conventional Householder transformation (Householder QR for short) can be made reproducible by fixing the order of computation and by using the reproducible versions of BLAS routines [10] such as the reproducible dot product and 2-norm [11]. However in a highly parallel environment, the column-by-column computation requires many synchronizations and therefore can be bad for strong scaling. An alternative is to use the Cholesky QR algorithm which is much more efficient in terms of communication. However, Cholesky QR is known to be less stable and cannot guarantee the orthogonality of the computed Q matrix.

Communication-avoiding algorithms such as Tall-Skinny QR and Communication-avoiding QR [6] reduce the amount of communication at a cost of a small number of extra FLOPs. Also by making use of level-3 BLAS Routines, those communication-avoiding algorithms obtain much better performance than the conventional Householder QR in highly parallel environments. The TSQR factorization for a matrix of size $n \times b$ requires $2b$x fewer messages, and exhibits a speedup of 6.7x on 16 processors of a Pentium III cluster, up to 4x on 32 processors of an IBM BG/L [6], and up to 13x on GPU [14]. Communication-avoiding algorithms change the order of communication intentionally and usually depending on the available computing resources such as number of processors, available memory size, etc. Therefore it is more difficult to attain reproducibility using the TSQR algorithm without any prior information about the platform. Another issue with the TSQR algorithm is that it stores the Q matrix implicitly using a hierarchical tree of intermediate matrices of Householder vectors (which we call Y matrices), so it also needs to record and reuse the reduction tree used by TSQR in order to apply Q. A recent paper [3] introduced a new technique to reconstruct Householder vectors from either the computed Q matrix or R matrix. In both cases, in order to obtain a reproducible QR factorization, a reproducible R matrix must be computed first.

In this paper we introduce a new algorithm to reproducibly compute a QR factorization of a tall skinny matrix, which usually requires much less communication than the conventional Householder QR and is almost as stable as the Householder QR provided that the input matrix is not too ill-conditioned (which the algorithm will confirm). The new algorithm is based on the Cholesky QR algorithm, which is fast in computation for both sequential and parallel environments but is unstable when the input matrix is ill-conditioned. It uses the new technique presented in [3] to reconstruct Householder vectors. In addition, a refinement technique is used to improve the numerical quality.

The paper is organized as follows: Section II briefly summarizes the currently available algorithms for QR factorization of a tall skinny matrix. Section III describes our new technique to reproducibly compute the QR factorization. Section IV shows some experimental results in Matlab. Finally, Section V contains some conclusions as well as

some discussion of possible future work.

A. Notation

Throughout this paper, we use Matlab-like notations to describe algorithms, for example `chol` and `lu` stand for the Cholesky and LU factorization without pivoting operations respectively. The division $x = A\backslash b$ stands for the solution of a system of linear equations $Ax = b$, or $x = A^{-1}b$, in contrast to $A/B = AB^{-1}$.

$A(r_1 : r_2, c_1 : c_2)$ denotes a submatrix of A which spans from row r_1 to row r_2 and from column c_1 to column c_2. $A(i, :)$ and $A(:, j)$ denote the ith row and the jth column of A respectively. $I_{m,n} \in \mathbb{R}^{m\times n}$ denotes an identity matrix with ones on the diagonal and zeros elsewhere. $0_{m,n} \in \mathbb{R}^{m\times n}$ denotes a zero matrix. $[A; B]$ denotes a vertical concatenation operation which forms a new matrix by stacking A on top of B where A and B have the same number of columns.

II. Previous Work

Let $Q \in \mathbb{R}^{n\times b}$, $R \in \mathbb{R}^{b\times b}$ be a QR factorization of $A \in \mathbb{R}^{n\times b}$, i.e. $A = QR$ where R is upper triangular and Q is orthogonal. Therefore $A^TA = R^TQ^TQR = R^TR$. It means that R is a Cholesky factor of A^TA. Note that, unlike R from Cholesky, R from QR factorization can have negative diagonals. That leads to Algorithm 1 to compute the QR factorization of A based on Cholesky factorization.

Algorithm 1 Cholesky QR $[Q, R] = \texttt{cholQR}(A)$

Require: A is $n \times b$ matrix, $\texttt{cond}(A) < \varepsilon^{-1/2}$
1: $Z = A^TA$
2: $R = \texttt{chol}(Z)$
3: $Q = A/R$

`cholQR` only uses three kernels:

- the matrix multiplication A^TA which costs $2nb^2$ if BLAS `gemm` routine is used but only costs nb^2 FLOPs if the symmetry of Z is taken into account and BLAS `syrk` is used,
- the Cholesky factorization of a $b\times b$ matrix which costs $b^3/3$ FLOPs using LAPACK's `potrf` routine,
- the solution Q of a triangular linear system $QR = A$ which costs nb^2 FLOPs using BLAS `trsm` routine.

In total `cholQR` costs $2nb^2+\mathcal{O}(b^3)$ FLOPs. In cases where only the R matrix is needed, the last line of Algorithm 1 can be omitted, which leads to a modified algorithm named `cholR` that costs $nb^2 + \mathcal{O}(b^3)$ FLOPs instead of $2nb^2 + \mathcal{O}(b^3)$ FLOPs.

With Q being computed from the triangular solve, `cholQR` will likely obtain backward stability in the sense that $\| A - QR \| = \mathcal{O}(\varepsilon) \| A \|$ where ε is machine epsilon. However, the orthogonality of Q cannot be guaranteed. In a recent paper [3], a new technique building on a method introduced by Yamamoto [4] was proposed to reconstruct the Householder vectors from A and R matrices, which guarantees the orthogonality of Q. In brief, that algorithm can be explained as following: Let Y be a lower triangular unit diagonal matrix containing the Householder vectors corresponding to Q in YTY^T compact format [13], where T is a $b\times b$ upper triangular matrix so that $Q = I_{n,b} - YTY_1^T$, where Y_1 is the top $b \times b$ block of Y. Therefore $A = QR = [R; 0_{n-b,b}] - YTY_1^TR$, which can be rewritten as : $A - [R; 0_{n-b,b}] = -YTY_1^TR$.

Let $V = -TY_1^TR$. Since T, Y_1^T, and R are upper triangular, V is also an upper triangular matrix. This means that (Y, V) is an LU factorization of $A-R$ without pivoting. Note that (Y, V) is unique since Y is a unit lower triangular matrix. Similarly to the Householder transformation, in order to avoid cancellation of the diagonal elements of $A - R$, a sign flipping diagonal matrix S whose diagonal elements are either 1 or -1 is used [3]: $(L, U) = \texttt{lu}(A - SR)$. Algorithm 2 (`modLU`) is a simplified version of [3, Algorithm 11].

Algorithm 2 Modified LU factorization $[L, U, S] = \texttt{modLU}(A, R)$

Require: A is $n \times b$ matrix, R is $b \times b$ upper triangular matrix
1: $S = I_{b,b}$
2: **for** $i = 1$ to b **do**
3: **if** $\texttt{sgn}(A(i,i)) = \texttt{sgn}(R(i,i))$ **then**
4: $S(i,i) = -1$
5: **end if**
6: $A(i + 1 : n, i) = A(i + 1 : n, i)/(A(i,i) + S(i,i)R(i,i))$
7: $z = A(i, i + 1 : b) - S(i,i)R(i, i + 1 : b)$
8: $A(i + 1 : n, i + 1 : b) = A(i + 1 : n, i + 1 : b) - A(i + 1 : n, i)z$
9: **end for**
10: $U = \texttt{triu}(A)$
11: $L = \texttt{tril}(A)$

`modLU` returns a tuple (L, U, S) corresponding to the lower triangular, upper triangular and sign flipping matrices. `modLU` has the same cost as a normal LU factorization without pivoting.

Algorithm 3 reconstructs the Householder vectors of the QR factorization of A from a computed R matrix. It returns the result in LAPACK format [12] where the strictly lower triangular part of Y contains the Householder vectors, and the upper triangular part of Y contains the R matrix. It costs $nb^2+\mathcal{O}(b^3)$ FLOPs, in which the most expensive part is the triangular matrix solve in line 6. Algorithm 3 also computes a T in the YTY^T representation of Householder transformation, based on the fact that $V = TY_1^TR$, which means $T = V/(Y_1^TR)$.

Algorithm 3 Reconstruct Householder vectors from R matrix $[Y,T] = \texttt{r2y}(A,R)$

Require: A is $n \times b$ matrix, R is $b \times b$ upper triangular matrix

1: $A1 = A(1:b, 1:b)$ ▷ Upper part of A
2: $A2 = A(b+1:end, 1:b)$ ▷ Lower part of A
3: $[Y1, U, S] = \texttt{modLU}(A1, R)$
4: $R = SR$
5: $Y1(1:b+1:b^2) = 0$ ▷ Set diagonal of $Y1$ to 0
6: $Y2 = A2/U$
7: $Y = [Y1 + R; Y2]$
8: $T = -U/(Y1^T R)$

Using Algorithms 1 and 3, we can compute a QR factorization of A using level-3 BLAS routines as described in Algorithm 4.

Algorithm 4 Reproducible QR factorziation $[Y,T] = \texttt{repQR}(A)$

Require: A is $n \times b$ matrix, $\texttt{cond}(A) < \varepsilon^{-1/2}$

1: $Z = A^T A$
2: $R = \texttt{chol}(Z)$
3: $[Y,T] = \texttt{r2y}(A,R)$

Recall that since the Q matrix computed by Algorithm 4 is stored in Householder vector format, it is always orthogonal. Note that Q can be reconstructed by applying the Householder transformations to an identity matrix or by using formula $I - YTY^T$ where T is a $b \times b$ upper triangular matrix satisfying $T^{-1} + T^{-T} = Y^T Y$ [3]. If the input matrix is very well conditioned, i.e. $\texttt{cond}(A) \approx 1$, one can use T also computed from Algorithm 4 to save computational cost. However, Algorithm 4 is unstable since both `cholQR` and `r2y` are sensitive to the condition number of A, especially `cholQR` since $\texttt{cond}(Z) \approx \texttt{cond}(A)^2$. It means that if $\texttt{cond}(A) > \varepsilon^{-1/2}$ then $\texttt{cond}(Z) > \varepsilon^{-1}$, so we might even fail to compute a Cholesky decomposition of Z.

The following section introduces techniques to improve the numerical quality of both the `r2y` and `cholQR` functions.

III. Algorithm

In this section we will discuss new techniques to improve the numerical quality of Algorithm 4. Since Q is stored in Householder vector format, Q is always guaranteed to be orthogonal. Therefore we pay attention to the backward stability, i.e. obtaining a small residual $A - QR$. More specifically, we aim to obtain almost the same accuracy as the Householder QR, which means that we need to guarantee a relatively small column-wise relative error:

$$error_{col} = \max_{i=1}^{b} \frac{\| A(:,i) - QR(:,i) \|}{\| A(:,i) \|}, \qquad (1)$$

where $A(:,i)$ denotes the i-th column of A.

A. Recursive Cholesky QR

It is shown that when the input matrix A is ill-conditioned, i.e. $\texttt{cond}(A) > \varepsilon^{-1/2}$, `cholQR` can fail to compute the Cholesky factorization of $Z = A^T A$. It means that `repQR` algorithm cannot proceed. In practice, in case of failure, i.e. the algorithm would need to divide by the square root of a nonpositive pivot, both Matlab's `chol()` function and LAPACK's `dpotrf` routine, return an upper triangular matrix R and a positive integer $p < b$ where R is of size $p \times p$ and $R^T R = Z(1:p, 1:p)$. $p = 0$ means Cholesky factorization runs to completion. In our case, R is actually an R factor of the first p columns of A since $Z(1:p, 1:p) = A^T(:, 1:p)A(:, 1:p)$. Therefore, we can use Algorithm 3 to compute the QR factorization of the first p columns of A, which can then be used to update the trailing matrix much like the Householder QR. Algorithm 5 depicts the process in detail.

Algorithm 5 Recursive Cholesky QR for ill-conditioned matrix $Y = \texttt{repQR2}(A)$

Require: A is $n \times b$ non-singular matrix matrix

1: $Z = A^T A$
2: $[R,p] = \texttt{chol}(Z)$
3: **if** $p = 0$ **then** ▷ Cholesky factorization runs to completion
4: $[Y,T] = \texttt{r2y}(A,R)$ **return**
5: **end if**
6: $A1 = A(1:n, 1:p)$
7: $A2 = A(1:n, p+1:b)$
8: $[Y1, T] = \texttt{r2y}(A1, R)$ ▷ QR factorization of first half
9: $A2 = \texttt{apply\_yty}(A2, Y1, T)$ ▷ Update second half
10: $Y2 = \texttt{repQR2}(A2(p+1:n, 1:b-p))$ ▷ Recursively factorize second half
11: $A2(p+1:n, 1:b-p) = Y2$
12: $Y = [Y1, A2]$ ▷ Horizontally concatenate 2 matrices

The updating of trailing matrix `apply_yty` can be done by applying the Householder reflectors in $Y1$ one-by-one to $A2$, which is costly in terms of communication. Alternatively, the update can be done using the T matrix of YTY^T representation [13]: $A2 = (I - Y_1 T^T Y_1^T)A2 = A2 - Y_1 T^T (Y_1^T A2)$, T is also computed by function `r2y`. This requires one reduction to compute $Y_1^T A2$ and one broadcast of the $p \times p$ matrix $T^T(Y_1 A2)$ to all processors.

At the cost of an additional reduction and broadcast to compute the max-norms of the columns of A and scaling each column by the nearest power of 2 to have (close to) unit norm, we can maintain reproducibility, avoid over/underflow problems, and identify and skip columns that are exactly zero. Alternatively, the reproducible dot product used to compute $A^T A$ could use enough extra exponent range for the same purpose.

Using this recursive scheme, we can always factorize a non-singular matrix even with high condition number. In the worst case, where every 2 consecutive columns during the computation are almost linearly dependent then each

time we can only compute the Cholesky QR factorization of the first column, the computation process will look exactly like the Householder QR factorization. In that case, the algorithm will cost $\mathcal{O}(nb^3)$ FLOPs instead of $\mathcal{O}(nb^2)$ since we might need to perform Cholesky QR factorization of each submatrix $A(i:n, i:b), 1 \leq i \leq b$.

B. Recursive Reconstruction of Householder vectors from R matrix

In this section we suppose that a matrix R has been successfully computed by applying `cholQR` algorithm to an input matrix A. It implies that A is not too ill-conditioned. Roughly speaking, A needs to satisfy $\texttt{cond}(A) < \varepsilon^{-1/2}$.

It is known that Cholesky factorization has perturbation bounds that are strongly dependent on the condition number [7, Section 10.2].

Theorem III.1. *Let $A \in \mathbb{R}^{n\times n}$ be symmetric positive definite with the Cholesky factorization $A = R^T R$ and let ΔA be a symmetric matrix satisfying $\| A^{-1}\Delta A \|_2 < 1$. Then $A + \Delta A$ has the Cholesky factorization $A + \Delta A = (R + \Delta R)^T (R + \Delta R)$, where*

$$\frac{\| \Delta R \|_F}{\| R \|_p} \leq 2^{-1/2} \frac{\kappa_2(A)\epsilon}{1 - \kappa_2(A)\epsilon} \tag{2}$$

with $\epsilon = \frac{\|\Delta A\|_F}{\|A\|_p}, \quad p = 2, F, \quad \kappa_2(A) = \| A^{-1} \|_2 \| A \|_2$

In our case of computing the Cholesky factorization of $Z = A^T A$, in finite precision arithmetic according to [7, Section 3.5] we have $\| Z(i,i) - (A^T A)(i,i) \| \leq \gamma_n \| A^T \| \| A \|$. Let $A^T A = Z + \Delta Z$, and ignoring the lower order error term, we have $\| \Delta Z \| \lesssim n\varepsilon \| Z \|$, then $A^T A$ has the Cholesky factorization $A^T A = \hat{R}^T \hat{R}, \hat{R} = R + \Delta R$, where

$$\frac{\| \Delta R \|_F}{\| R \|_p} \leq 2^{-1/2} \frac{n\kappa_2(Z)\varepsilon}{1 - \kappa_2(Z)n\varepsilon} \approx 2^{-1/2} \frac{n\kappa_2^2(A)\varepsilon}{1 - n\kappa_2^2(A)\varepsilon}. \tag{3}$$

Since `r2y` computes the LU factorization of $A - SR$ where S is the diagonal sign flipping matrix, it has the same perturbation behavior as the LU factorization [7, Section 9.11]. In addition, since the diagonal elements of R are non-zero, using the sign flipping technique, diagonal elements during Gaussian elimination of $A - SR$ are also non-zero. However, diagonal elements of $A - SR$ can still be very small which might lead to a large growth factor since we do not use pivoting.

In order to improve the numerical quality of Algorithm 5, we propose to use iterative refinement steps to improve the numerical quality of both `cholQR` and `r2y`. Iterative refinement is based on the fact that in exact arithmetic $A/R = Q$ is an orthogonal matrix. In this case R is the computed Cholesky factor of the computed Z, if R is a good enough approximation of the R matrix in the QR factorization of A than $B = A/R$ should be close to an orthogonal matrix, or $\texttt{cond}(B)$ should be close to 1. In that case, we can apply Algorithm 1 to compute a good approximation of the R factor of B. Algorithm 6 depicts the iterative refinement steps.

Algorithm 6 Recursively Reconstruct Householder vector from R $[Y, T, R] = \texttt{rec\_r2y}(A, R)$

Require: A is $n \times b$ matrix, R is $b \times b$ upper triangular matrix. $\texttt{cond}(A), \texttt{cond}(R) < \varepsilon^{-1/2}$

1: **repeat**
2: $B = A/R$ ▷ Precondition A by inverse of R
3: $R1 = \texttt{cholR}(B)$
4: $R = R1 * R$
5: $cnd = \texttt{cond}(R1)$
6: **until** cnd is small
7: $[Y, T] = \texttt{r2y}(B, R1)$
8: $Y = \texttt{tril}(Y, -1) + \texttt{triu}(Y)R$

Stopping Criteria: The question remains of when to stop the refinement process. The main goal of the refinement process is to reduce the condition number of input matrix for the Cholesky QR algorithm `cholR`. Ideally, the refinement stops when we obtain $B = A/R$ with $\texttt{cond}(B) \approx 1$. However, in practice, due to rounding error we might never obtain $\texttt{cond}(B) = 1$ even if R is exactly computed. Also, computing a too accurate B would require an excessive running time, since each iterative step to refine B is twice as costly as Cholesky QR factorization.

First, to prevent the refinement from running forever, we stop the refinement process at a maximum number of iterations regardless of the quality of computed R matrix. In practice, in most cases we only need 1 or 2 iterations to obtain a good accuracy. Therefore, in our implementation we set the maximum number of iterations to 4.

Second, in order to determine a good enough condition number for $R1$, we need to look at the numerical behavior of the algorithm. As mentioned in the previous section, the Cholesky factorization of $A^T A$ has the perturbation bounds:

$$\frac{\| \Delta R \|_F}{\| R \|_p} \leq 2^{-1/2} \frac{n\kappa_2^2(A)\varepsilon}{1 - n\kappa_2^2(A)\varepsilon}$$

Therefore, in order to obtain a good Cholesky factorization of $A^T A$, input matrix A must at least satisfy: $n\kappa_2^2(A)\varepsilon < 1$, or $\kappa_2(R) \approx \kappa_2(A) < (n\varepsilon)^{-1/2}$.

Since $\texttt{cond}(R) \approx \texttt{cond}(A) < \varepsilon^{-1/2}$, the solution of the triangular system $B = A/R$ has a small forward error bound. Let $\hat{B}$ be the exact solution of A/R, we have [7, Section 8.2]:

$$\frac{\| B(:,i) - \hat{B}(:,i) \|}{\| \hat{B}(:,i) \|} \leq \frac{\texttt{cond}(R)\gamma_b}{1 - \texttt{cond}(R)\gamma_b}, \gamma_b = \frac{cb\varepsilon}{1 - cb\varepsilon} \tag{4}$$

$$\Rightarrow \quad \| B - \hat{B} \| \leq \| \hat{B} \| \frac{cb\varepsilon^{1/2}}{(1 - \texttt{cond}(R)\gamma_b)(1 - cb\varepsilon)} \tag{5}$$

for a small constant c.

Moreover, let $\hat{Q}$ be the orthogonal matrix so that $A = \hat{Q}(R + \Delta R)$, we have:

$$\hat{B} = AR^{-1} = \hat{Q}(R + \Delta R)R^{-1}. \tag{6}$$

Therefore

$$\begin{aligned}\| \hat{B} - \hat{Q} \|_p &= \| \hat{Q}\Delta R R^{-1} \|_p = \| \Delta R R^{-1} \|_p, \quad p = 2, F \\ &\leq \| \Delta R \|_p \| R^{-1} \|_p \\ &\leq \| R^{-1} \|_p \| R \|_p \, 2^{-1/2} \frac{n\kappa_2^2(A)\varepsilon}{1 - n\kappa_2^2(A)\varepsilon} \\ &\leq 2^{-1/2} \frac{n\kappa_p(R)\kappa_2^2(A)\varepsilon}{1 - n\kappa_2^2(A)\varepsilon} \approx 2^{-1/2} \frac{n\kappa_2^3(R)\varepsilon}{1 - n\kappa_2^2(R)\varepsilon}. \end{aligned} \tag{7}$$

In order to compute a good approximation of Q, we would want $\| B - Q \| < 1$. Therefore we need to have $\texttt{cond}(R) < (n\varepsilon)^{-1/3}$. This condition is used as the stopping criteria for the iterative refinement process in our Matlab implementation.

With this recursive algorithm for reconstructing Householder vectors, we obtain the final version of our algorithm, which is described in Algorithm 7. Like Householder transformation, Algorithm 7 makes no assumption about the condition number of input matrix beside its non-singularity.

Algorithm 7 Recursive Cholesky QR for ill-conditioned matrix $Y = \texttt{rec\_repQR}(A)$

Require: A is $n \times b$ non-singular matrix
1: $Z = A^T A$
2: $[R, p] = \texttt{chol}(Z)$
3: **if** $p = 0$ **then** ▷ Cholesky factorization runs to completion
4: $[Y, T, R] = \texttt{rec\_r2y}(A, R)$
5: **return**
6: **end if**
7: $A1 = A(1 : n, 1 : p)$
8: $A2 = A(1 : n, p + 1 : b)$
9: $[Y1, T] = \texttt{rec\_r2y}(A1, R)$ ▷ QR factorization of first half
10: $A2 = \texttt{apply\_yty}(A2, Y1, T)$ ▷ Update second half
11: $Y2 = \texttt{rec\_repQR}(A2(p + 1 : n, 1 : b - p))$ ▷ Recursively factorize second half
12: $A2(p + 1 : n, 1 : b - p) = Y2$
13: $Y = [Y1, A2]$ ▷ Horizontally concat 2 matrices

C. Reproducibility

In parallel environments, in all the algorithms presented in this paper, the only operation that requires reduction operations is the matrix multiplication $A^T A$ for Cholesky QR factorization. The other operations either require only broadcast operations or can be performed independently locally. All the conditional branches are also performed locally on a single node.

In the context of this paper, we make an assumption that all local computation can be made reproducible by fixing the order of computation or by using some library which supports reproducibility on a single node, such as the latest version of Intel MKL [8] library with the Conditional Numerical Reproducibility (CNR) feature. Therefore in order to attain reproducibility, we only need to reproducibly compute the matrix multiplication. That can be done using using new techniques proposed in [1], [2], or less efficiently, by enforcing a deterministic order of summation, or using exact arithmetic.

According to [1], [2], a reproducible summation algorithm in double precision uses 8 times more FLOPs than a conventional summation algorithm. It means that a reproducible matrix multiplication of size $n \times n$ will require $8n^3$ floating-point additions, and n^3 floating-point multiplications. In our case of tall-skinny matrix multiplication, we can improve the performance by using a blocking technique. Suppose that the input matrix is partitioned into submatrices of fixed block size $NB \times NB$ which will be distributed among computing nodes. With the assumption that the data layout of each block is fixed, a performance-optimized library can be used to reproducibly compute the products of corresponding blocks, which costs $2NB^3$ FLOPs. The reduction operations can then be performed using reproducible summation algorithm which costs $\mathcal{O}(8NB^2)$ FLOPs per reduction to produce the final reproducible result at a modest extra cost. It means that the computation overhead in this case is only a factor of $1 + 4/NB$ higher in terms of FLOP count. In practice, the overhead can be larger by a constant factor only since BLAS level-3 routines are better optimized than BLAS level-1 routines.

D. Communication Cost

For the cost analysis of presented algorithms, we use the (γ, α, β) model [3]:

- γ is the computational cost of 1 FLOP,
- α is the latency cost of sending each message,
- β is the bandwidth cost of sending each word.

With this model, the cost of sending a block of w elements is $\alpha + \beta w$. The cost of a broadcasting of w-word blocks to all p processors is $\alpha \log p + \beta(w \log p)$. And the cost of a reduction operation (`MPI_Reduce` or `MPI_AllReduce` [9]) is $\gamma(w \log p) + \alpha \log p + \beta(w \log p)$. When $w \leq p$, pipelined algorithm can be used to reduce the bandwidth cost of broadcast and reduction operation to βw by subdividing the message into many small pieces. For simplicity, our cost analyses do not make any assumption about the relation between w and p, therefore we use $\beta(w \log p)$ as the bandwidth cost for broadcast and reduction.

Since we are interested in tall skinny matrices, in the context of this paper we assume a 1-D network layout where n rows of the input matrix are distributed equally among p processors with $p \ll n$, i.e. each processor holds n/p consecutive rows of A, Y and B. For simplicity, we assume load balancing for parallel computation of the algorithms, including the matrix multiplication, triangular matrix solution, and LU factorization.

The costs of Householder QR are given in [3]: $\gamma(2nb^2/p - 2b^3/3) + \alpha(2b \log p) + \beta((b^2/2) \log p)$.

`cholQR` requires 2 reduction operations: one for the computation of $A^T A$ (line 1 of Algorithm 1), and one for broadcasting R to all processors (line 3). If symme-

try is taken into account, the total cost of cholQR is $\gamma(nb^2/p + b^3/3 + (b^2/2)\log p) + \alpha(2\log p) + \beta(b^2 \log p)$.

r2y uses only 1 reduction operation to broadcast U to all processor (line 6 of Algorithm 3) which costs $\alpha \log p + \beta((b^2/2)\log p)$. Since modLU has the similar computational cost as a normal LU factorization, the total cost of r2y is $\gamma(nb^2/p) + \alpha(\log p) + \beta((b^2/2)\log p)$.

Each iteration of Algorithm 6 involves 2 reduction operations in line 2 for broadcasting R and line 3 for computing B^TB. Thus, the cost of each iteration is $\gamma(2nb^2/p + 5b^3/6 + (b^2/2)\log p) + \alpha(2\log p) + \beta(b^2 \log p)$. Let k be the number of iterations, then the total cost of rec_r2y is $\gamma((2k+1)nb^2/p + 5kb^3/6 + k(b^2/2)\log p) + \alpha((2k+1)\log p) + \beta((2k+1)(b^2/2)\log p)$.

The communication cost of Algorithm 7 is more complicated and depends on input matrices. In the case of well conditioned input matrices, i.e. $\mathtt{cond}(A) < \varepsilon^{-1/2}$, the algorithm exits at line 5, the total cost of rec_repQR is $\gamma((2k+2)nb^2/p + 5kb^3/6 + b^3/3 + (k+1)(b^2/2)) + \alpha((2k+2)\log p) + \beta((2k+2)b^2 \log p)$ where k is the number of iterations of rec_r2y. Therefore the computational cost and the bandwidth cost of rec_repQR are $(k+1)$ times larger than those of Householder QR. However, the latency cost of rec_repQR is $b/(k+1)$ times smaller than that of Householder QR. In practice, the iterative refinement in Algorithm 6 usually stops at 1 or 2 iterations. Thus, for well conditioned matrices, the latency cost of rec_repQR is $b/2$ or $b/3$ times smaller than the latency cost of Householder QR, which gives rec_repQR an advantage over Householder QR on massively parallel environments.

IV. Experimental Results

In this section we present some experimental results in Matlab version R2012a on an Intel® Core™ i7-2720QM processor with 4 cores of 2.2 GHz speed and 8GB of main memory. Note that since there is no reproducible library for level-3 BLAS routines available yet, we could not carry out real tests of the reproducible algorithms presented in this paper. As discussed in the previous section, our algorithms can attain reproducibility once a reproducible matrix multiplication implementation is available. According [2], the reproducible sum has almost the same accuracy as the normal sum. This can also be applied to the case of matrix multiplication. Since the only operation that needs to be performed in a reproducible manner is the matrix multiplication, the accuracy observation in the following Matlab tests can also be applied to future tests using reproducible matrix multiplication operation. Also, since most of the running time is spent in large dense matrix operations, the additional overhead of Matlab is limited.

We performed our tests on multiple sets of test matrices of size 10000×32, with condition number varying from 2^{10} to 2^{53}:

1) matrices generated using the gallery function from Matlab with matrix type randsvd: $A = U\Sigma V^T$, where $U \in \mathbb{R}^{n\times n}$ and $V \in \mathbb{R}^{b\times b}$ are random orthogonal matrices and $\Sigma = diag(\sigma_i)$ is a given matrix of singular values.
2) matrices generated by $A = QR$ where Q, R are formed by taking the QR factorization of a random matrix and by setting the center element $R(b/2, b/2)$ to a small value.
3) matrices generated by $A = QR$ where Q is a random orthogonal matrix, and R is a random upper triangular matrix so that $\mathtt{cond}(R(2i+1:2i+2, 2i+1:2i+2)) = \kappa, 0 \le i < b/2$, for a given condition number κ, and $R(i,j) \approx \varepsilon$ elsewhere.

For each test set, we collect the following data:

- Running time of the 4 algorithms r2y, repQR, repQR2, and rec_repQR. Note that for r2y, we use Matlab's built-in qr() function to compute the R matrix. The running time of Matlab's qr() is included in the running time of r2y in the figures. For repQR, a very small running time which is close to 0 means that the algorithm fails to compute a QR factorization of the input matrix.
- Column-wise relative error of all those 4 algorithms which is computed by (1).

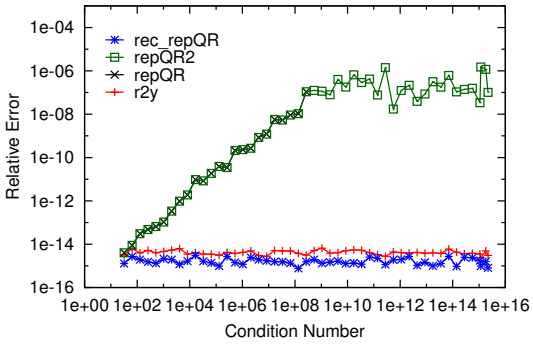
Figure 1 summarizes experimental results of the first set of test matrices. As explained in Section II, due to the strong dependence of Cholesky algorithm's numerical quality on the condition number of A, the naive repQR stops working when the condition number of A reaches about $\epsilon^{-1/2}$. Using a recursive technique, repQR2 still succeeds when the input matrix has an even higher condition number. However the column-wise relative error of repQR2 when $\mathtt{cond}(A) > \epsilon^{-1/2}$ is limited by $\epsilon^{-1/2}$. Meanwhile both r2y and rec_repQR obtain very good column-wise relative errors, since rec_repQR employs the refinement steps for the QR factorization of sub-panel. In terms of running time, rec_repQR is usually around 4 times slower than Matlab's qr function. Meanwhile r2y is less than 3 times slower than the Matlab qr function.

In the second test set (Figure 2), repQR and repQR2 exhibit even a stronger dependence on the condition number of input matrices. Differently from the first test, in this test r2y is also sensitive to the condition number of input number. This phenomenon might be explained by the fact that the very small value in the middle of the diagonal of R makes the trailing matrix more sensitive to the perturbation by rounding errors. In this test, rec_repQR still provides good accuracy regardless of the input matrices' condition number.

Test matrices in the third test are generated in such a way that Cholesky QR factorization will fail more often. When the condition number of input matrices is smaller than $\varepsilon^{-1/2}$, the Cholesky QR factorization can proceed successfully on the whole input matrix. Therefore all the 4 algorithms behave exactly the same as in the second test set. However, when the condition number of the input matrix gets bigger than $\varepsilon^{-1/2}$, rec_repQR and repQR2 see a surge in running time since both those two algorithms might need to restart the Cholesky QR factorization as often as

Figure 1: Test 1

(a) Accuracy



(b) Performance

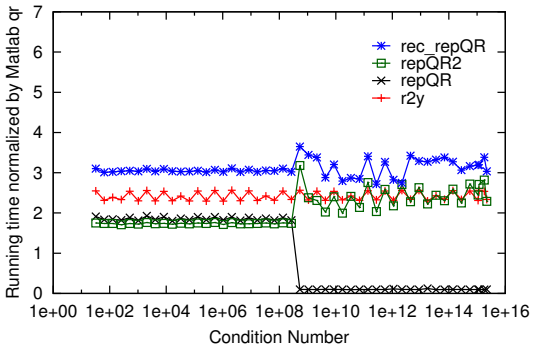
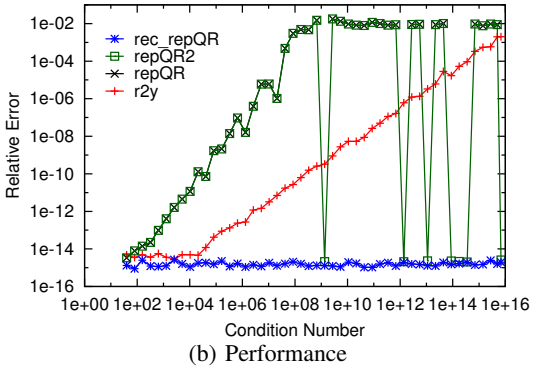


Figure 2: Test 2

(a) Accuracy



(b) Performance

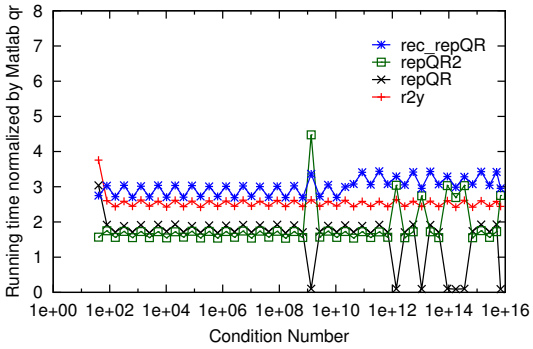
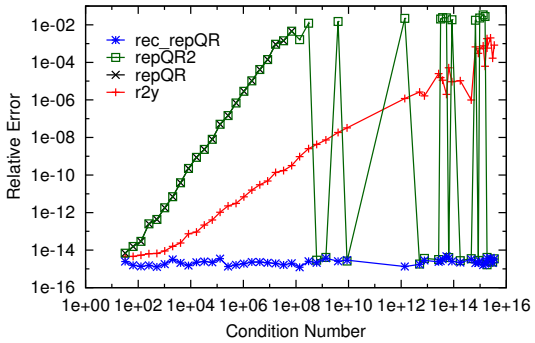
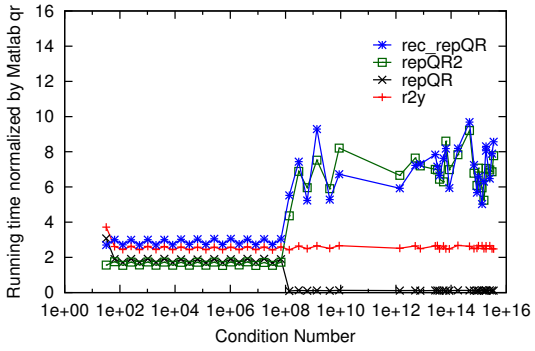


Figure 3: Test 3

(a) Accuracy



(b) Performance



every two columns of the input matrix. This type of input matrix is however not so common in practice.

V. Conclusion and Future work

In this paper, we introduced a technique to reproducibly compute a QR factorization of a non-singular matrix and return the result in Householder vector format, which guarantees the orthogonality of the computed Q factor. In most cases, our proposed technique obtains very good accuracy as measured by the norms of the columns of the residual $A - QR$, compared to the norms of the corresponding columns of A. When the matrix is well-conditioned, i.e. whose condition number is smaller than $\varepsilon^{-1/2}$, our technique runs 4 times slower than the Matlab's built-in `qr()` function for test matrices of size 10000×32, but is much more efficient in terms of communication. When the condition number of the input matrix gets higher, our technique still provides a good accuracy at the cost of increased running time since refinement steps need to be performed.

This work still needs to be completed by some formal proof of numerical quality in finite precision arithmetic. We will also need to collect more experimental data once a library for reproducible level-3 BLAS routines is available, such as the ReproBLAS [11] which is still work in progress. We also want to investigate alternative approaches to reduce the number of iterative refinement steps when the condition number of input matrix is high, for example Cholesky factorization with pivoting, or extra-precise Cholesky QR factorization which requires both the multiplication $A^T A$

and the Cholesky factorization to be computed in higher precision.

VI. Acknowledgments

This research is supported in part by NSF grant NSF ACI-1339676, DOE grants DOE DE-SC0010200, DOE DE-SC0008699, DOE DE-SC0008700, DOE AC02-05CH11231, and DARPA grant HR0011-12-2-0016, ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

References

[1] James Demmel and Hong Diep Nguyen, *Fast Reproducible Floating-Point Summation*. 21st IEEE Symposium on Computer Arithmetic (ARITH), vol., no., pp.163,172, 7-10 April 2013.

[2] James Demmel and Hong Diep Nguyen, *Parallel Reproducible Summation*. IEEE Transactions on Computers, DOI: 10.1109/TC.2014.2345391.

[3] Grey Ballard, James Demmel, Laura Grigori, Mathias Jacquelin, Hong Diep Nguyen, and Edgar Solomonik, *Reconstructing Householder Vectors from Tall-Skinny QR*. Tech report, UCB/EECS-2013-175, October 2013.

[4] Yusaku Yamamoto, *Aggregation of the compact WY representations generated by the TSQR algorithm*. Conference talk presented at SIAM Applied Linear Algebra, 2012.

[5] Sylvain Collange, Devid Defour, Stef Graillat, and Roman Iakymchuk, *A Reproducible Accurate Summation Algorithm for High-Performance Computing*. In Proceedings of the SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14) held as part of the 2014 SIAM Annual Meeting. Chicago, Il, USA, July 6–11, 2014.

[6] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou, *Communication-optimal parallel and sequential QR and LU factorizations*. SIAM Journal on Scientific Computing, Vol. 34, No 1, 2012.

[7] Nicholas J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM, ISBN: 978-0-89871-521-7.

[8] Intel© Math Kernal Library, `https://software.intel.com/en-us/intel-mkl`.

[9] Message Passing Interface, `http://www.mpi-forum.org`.

[10] Basic Linear Algebra Subroutines, `http://www.netlib.org/blas/`

[11] Reproducible Basic Linear Algebra Subroutines, `http://bebop.cs.berkeley.edu/reproblas`.

[12] Linear Algebra PACKage, `http://www.netlib.org/lapack/`.

[13] Robert Schreiber and Charles Van Loan, *A Storage-Efficient WY Representation for Products of Householder Transformations*. SIAM Journal on Scientific and Statistical Computing, Vol. 10, No 1, pages 53–57, 1989

[14] Michael Anderson, Grey Ballard, James Demmel and Kurt Keutzer, *Communication-Avoiding QR Decomposition for GPUs*, in Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium, pages 48–58, 2011,