2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00040

PAQR: Pivoting Avoiding QR factorization

Wissam Sid-Lakhdar*, Sebastien Cayrols*, Daniel Bielich*, Ahmad Abdelfattah*, Piotr Luszczek*, Mark Gates*, Stanimire Tomov*, Hans Johansen‡, David Williams-Young‡, Timothy Davis†, Jack Dongarra*§ Hartwig Anzt*

* University of Tennessee, † Texas A&M University, ‡ Lawrence Berkeley National Laboratory, § Oak Ridge National Laboratory

***Abstract*—The solution of linear least-squares problems is at the heart of many scientific and engineering applications. While any method able to minimize the backward error of such problems is considered numerically stable, the theory states that the forward error depends on the condition number of the matrix in the system of equations. On the one hand, the QR factorization is an efficient method to solve such problems, but the solutions it produces may have large forward errors when the matrix is rank deficient. On the other hand, rank-revealing QR (RRQR) is able to produce smaller forward errors on rank deficient matrices, but its cost is prohibitive compared to QR due to memory-inefficient operations. The aim of this paper is to propose PAQR for the solution of rank-deficient linear least-squares problems as an alternative solution method. It has the same (or smaller) cost as QR and is as accurate as QR with column pivoting in many practical cases. In addition to presenting the algorithm and its implementations on different hardware architectures, we compare its accuracy and performance results on a variety of application-derived problems.**

***Index Terms*—Linear least-squares, QR factorization, QR decomposition, deficient matrix, rank-deficient, low-rank**

I. Introduction

a) Context: The solution of linear least-squares problems is at the heart of many scientific and engineering fields [1]. Formally, the problem is defined as:

$$\min_x ||Ax - b||_2 \tag{1}$$

where A is a (large) rectangular m-by-n matrix, b is the *right-hand side*, and x is the solution of the system.

While several methods exist [1]–[3] [4, ch.5] to solve such problems, the *QR factorization* plays a key role. It corresponds to the factorization of the matrix A into:

$$A = QR, \tag{2}$$

where Q is an m-by-n orthonormal matrix, and R an upper triangular n-by-n matrix. Given such a decomposition, the solution x is obtained by multiplying the inverse of Q to b:

$$y = Q^T b \tag{3}$$

then solving the triangular system of equations:

$$x = R^{-1} y \tag{4}$$

b) Challenge: The matrices arising in practical scientific and engineering applications like Quantum Chemistry and Weighted Least-squares (as studied in Section V-A1) present multiple challenges. First, the increase in computational power of modern computers has led to the increase in the size of the targeted matrices. This challenge has been addressed through the development of numerical libraries that take advantage of hardware accelerators, with both shared- and distributed-memory parallelism [5]–[7].

Second, in practice, matrices are often *low-rank* [8]. In essence, some columns of the matrix are redundant, or more precisely, they can be expressed as linear combinations of other columns [9]. Moreover, the characterization of rank deficiency can be blurry because of numerical round-off errors. This is due to the limited precision of the floating-point representation of real numbers' arithmetic. The implication is that, not a single, but an infinite number of potential solutions to the least-squares problem may exist. In such cases, although the QR factorization is a numerically stable operation [10, p. 384], the calculated solutions can be arbitrarily far from the true solution. This challenge has been addressed through the development of more robust methods, such as *QR with column pivoting* (QRCP) variants, such as *Rank-Revealing QR* (RRQR). Unfortunately, due to the communication and data movement induced by pivoting, these algorithms are very expensive in large-scale settings.

c) Contribution: The two challenges mentioned above lead to rethinking the traditional methods to solve large-scale rank deficient linear least-squares problems. We propose a new variant of the QRCP factorization family that we call *Pivoting Avoiding QR factorization* (PAQR). The guiding idea behind PAQR is that linearly dependant columns of the matrix can be detected and removed on the fly during the factorization process. This new variant yields accurate solutions without the expensive cost of column pivoting.

The initial goal of PAQR was to be numerically stable, more so than QR on challenging problems while remaining as fast as QR. However, PAQR turns out to be at least as fast as QR on full-rank problems, but faster than QR on rank-deficient problems. Moreover, it can be empirically as accurate as QRCP variants on practical rank-deficient least-squares problems.

d) Overview: This paper is organized as follows: Section II describes the QR factorization and QRCP variants such as RRQR, as well as the recent randomized variants. Section III describes the PAQR algorithm together with its limitations. Section IV describes the sequential, batched GPU, and distributed-memory implementations of PAQR, focusing on data layout, computations, and the communication volume.

Section V compares the numerical accuracy of PAQR with that of QR and QRCP variant, together with the performance of the methods in different computational settings. Section VI summarizes the work and discusses extensions of this work that are under consideration.

II. Background

a) QR: Algorithm 1 describes the overall flow of a QR factorization. At each iteration, the algorithm constructs an orthogonal transformation (Line 3), such as Gram-Schmidt projection, Givens rotation, or Householder reflection, then applies it to the trailing matrix. *Orthogonalization*$(A_{i:j,k})$ call takes the k'th column for orthogonalization, and produces its orthogonal transformation $V_{1:m,i}$, the diagonal element $R_{i,i}$ and the scaling factor τ_i. *Update*$(A_{i:m,i+1:n}, V_{1:m,i}, \tau_i)$ call takes the trailing matrix $A_{i:m,i+1:n}$, $V_{1:m,i}$, and τ_i, and outputs the updated trailing matrix $A_{i:m,i+1:n}$.

Algorithm 1 QR factorization

```
Input $A \in \mathbb{R}^{m\times n}$
Output $V \in \mathbb{R}^{m\times n}, R, \tau$
1: $V, R \leftarrow [0]$
2: for $i \leftarrow 1 \dots n$ do
3:     $V_{i:m,i}, R_{i,i}, \tau_i \leftarrow$ Orthogonalization$(A_{i:m,i})$
4:     $R_{1:i-1,i} \leftarrow A_{1:i-1,i}$
5:     $A_{i:m,i+1:n} \leftarrow$ Update$(A_{i:m,i+1:n}, V_{i:m,i}, \tau_i)$
6: end for
```

This algorithm is a non-blocked variant, as it updates the trailing matrix with a single column at a time. On the other hand, a blocked variant updates the trailing matrix with multiple columns at a time. The latter variant is preferred in practice, as it takes advantage of Level 3 BLAS routines for improved performance.

b) QRCP: Other variants of QR factorization exist that provide different numerical properties. *QR with Column Pivoting* (QRCP) is a family of such variants. It consists of selecting a column as pivot at every iteration that is likely different than what is usually selected by QR. Unlike the QR factorization, some QRCP variants are robust and able to reveal the true rank of matrices at the cost of communication-bound operations rendering it much slower than the QR factorization, which is compute-bound. An important variant of QRCP is *Rank Revealing QR* (RRQR) [11]. It selects the column with the largest norm and permutes it to the leading position. It decomposes a column-pivoted matrix $A\Pi \in \mathbb{R}^{m\times n}$ with rank k as

$$A\Pi = QR = Q\begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \tag{5}$$

where $Q \in \mathbb{R}^{m\times m}$ is orthonormal, $R_{11} \in \mathbb{R}^{k\times k}$ is upper triangular, $R_{12} \in \mathbb{R}^{k\times(n-k)}$, and $R_{22} \in \mathbb{R}^{(m-k)\times(n-k)}$. This factorization is said to be *rank revealing* [12], [13] if it satisfies the following two conditions:

$$\sigma_{min}(R_{11}) \geq \frac{\sigma_k(A)}{p(k,n)},\ \ \sigma_{max}(R_{22}) \leq \sigma_{k+1}(A)p(k,n) \tag{6}$$

where $p(k,n)$ is a low degree polynomial in k and n and σ_i are singular values. The *Strong Rank Revealing QR* [14] has the additional property of limiting the magnitude of the elements of $R_{11}^{-1}R_{12}$ to a given tolerance.

For the remainder of this paper, unless explicitly stated otherwise, QRCP refers to as the whole family of column pivoting variants.

c) Approximate blocked RRQR: Another variant [15] is both blocked and approximately rank-revealing. The main idea is to apply QRCP only on panels instead of the entire matrix. The loss of rank-revealing property stems from limiting the choice of pivots to columns only within a panel. Thus it trades the robustness for increased efficiency coming from Level 3 BLAS routines in the updates of the trailing matrix. Inside a panel, the *rejected* columns are identified as a linear combination of the previous columns and are pivoted to the end of the matrix. After computing an initial R_{11} factor this way, RRQR is then applied to the set of rejected columns in order to obtain the final R_{11}. The remaining columns are factored using QR for the construction of R_{22}.

d) Communication-avoiding algorithms: The classic QR factorization is well-known to be communication sub-optimal in distributed-memory environments. To remedy this, *Communication-Avoiding* (*CA*) variant of the QR factorization was developed [16] called CAQR. The idea was to split the panel into blocks of rows, compute a QR factorization on each block (local to a given process), then use a reduction tree (global to all processes) to compute the final R of the panel.

The *CA* approach was later extended to the RRQR algorithm to create the CARRQR algorithm [17]. The problem of RRQR is its sequential nature as at each step: the column with the largest norm of the trailing matrix has to be moved to the leading position. The proposed solution is to use a tournament pivoting strategy to find the best k pivots at once. The tournament pivoting is a reduction-tree operation, where at each tree node, a matrix of $2k$ columns is factorized using RRQR and only the largest k pivots are passed to the parent. Once the pivots are moved to the leading position, the classical iteration of the blocked QR factorization is used.

The communication-avoiding research is independent to the work in this paper, as CAQR could benefit to all QR factorization variants, including PAQR proposed in this paper.

e) Randomized algorithms: In recent years, the line of research using randomized methods in linear algebra has gained momentum as a scalable alternative to the classical approaches [8], [18]–[20]. The general idea is to project the matrix of interest on a random sub-space sized according to the rank of the input matrix, and then apply the expensive algorithms such as QRCP on the projection, which is a much smaller matrix. Such approaches benefit from strong theoretical guarantees such as high probability of capturing the true rank of a matrix. The performance is comparable to that of QR, and even better in the case of truncated randomized QRCP.

Randomized algorithms are complimentary to the work of this paper as they require the rank of the matrix as an input while the work in this paper relies on a threshold tolerance to be provided by the user instead.

III. Proposed algorithm

Two main metrics exist for evaluating the numerical accuracy of algorithms that deal with linear systems of equations and linear least-squares problems: *forward* and *backward error*. The forward error is defined as [21]:

$$e_{\text{forward}} = \frac{\|x - \hat{x}\|_p}{\|\hat{x}\|_p} \tag{7}$$

where $\hat{x}$ is the true solution and x is the computed solution, for a given p-norm. The backward error is defined as [22]:

$$e_{\text{backward}} = \frac{\|Ax - b\|_p}{\|A\|_p \|x\|_p + \|b\|_p}. \tag{8}$$

Even though the residual from the numerically stable algorithms is bound by the backward error, the forward error remains bound by the backward error magnified by the 2-norm *condition number* κ_2 of the input matrix [9]:

$$\kappa_2(A) = \|A\|_2 \|A^\dagger\|_2 = \frac{\sigma_{max}}{\sigma_{min}} \tag{9}$$

where $\|\cdot\|_2$ represents the 2-norm, $A^\dagger$ is the pseudoinverse, and σ_{max} and σ_{min} are the largest and smallest singular values of A, respectively.

In finite-precision arithmetic, machine precision ϵ indicates the attainable accuracy and unit round-off error [10]. Matrix A is considered numerically rank-deficient if:

$$\kappa_p(A) > \epsilon^{-1} \tag{10}$$

for a p-norm condition number.

The most accurate (but expensive) way to solve a rank-deficient least-squares problem is to compute the SVD of A and truncate the smallest singular values, so as to avoid the conditioning issue in Equation (10). However, a more common way to solve such problems is to use RRQR. Given the existing error bounds [12] linking the singular values of a matrix with the R matrix returned by RRQR, an early stopping criterion exists allowing to terminate the factorization prematurely once the remaining singular values of the trailing matrix are known to degrade the conditioning of R. However, the cost of column pivoting induced by this method makes it impractical at scale.

The intuition behind PAQR comes from the observation that a reduction of the condition number of a rank-deficient matrix can be achieved by detecting and removing columns on the fly if they contribute to the numerical deficiency of the matrix. It is done during the standard QR factorization without any pivoting. These skipped columns are linearly dependent to the already processed columns on the left. They do not contribute to the linearly independent columns whose count represents the matrix rank. This technique was applied in the development of a sparse rank-revealing QR by Davis in [23]. In PAQR, we follow the convention to define a column as *rejected* when it is identified as a linear combination of the previous columns [15]. As the decision for skipping the columns is made on-the-fly, PAQR is responsive to the numerical properties of matrix elements that change over the course of the algorithm's steps, which is superior to the passive approach of the classic QR factorization.

PAQR is a member of the QRCP family that follows the proposed *rejection pivoting* strategy, which differs from the local [15] and pairwise [24] pivoting strategies. The main difference between PAQR and the other existing QRCP variants (such as RRQR) is that PAQR completely avoids any kind of pivoting, and thus, does not incur any additional data movement. While QRCP variants focus on choosing columns as pivots, PAQR instead focuses on flagging columns as rejected. Note that, in its current development, PAQR does not guarantee the same properties given by RRQR's Equation (6). Indeed, post-processing (such as in [15]) may be needed on the R matrix in order to reveal the true rank of the matrix. This study however is outside the scope of this paper.

A. Algorithm details

PAQR is given in Algorithm 2. While its building blocks are the same as those of QR, it differs in three ways. First, at each iteration, PAQR computes a *deficiency criterion* (Line 5) (cf. Section III-B) to decide whether to flag the current column of A as rejected (Line 6). In which case, the algorithm proceeds immediately to the next iteration. Second, once columns are flagged as rejected, subsequent operations need to account for them. This is done by updating the indices of the trailing matrix (Lines 9 and 10) using the index k instead of the original loop index i. k is increased only when a column is estimated to be linearly independent from previous ones (Line 11). Consequently, the size of the output matrices V and R will be smaller (Line 14–15).

Algorithm 2 PAQR factorization

```
Input $A \in \mathbb{R}^{m \times n}$
Output $V, R, \tau, \delta$
 1: $V, R \leftarrow [0]$
 2: $k \leftarrow 1$
 3: for $i = 1 \ldots \min\{m, n\}$ do
 4:     $V_{k:m,k}, R_{k,k}, \tau_k \leftarrow$ generate_reflector($A_{k:m,i}$)
 5:     if $i^{th}$ column of $A$ is rejected then
 6:         $\delta_i \leftarrow$ TRUE(1)          ▷ skip the current column
 7:     else
 8:         $\delta_i \leftarrow$ FALSE(0)      ▷ include the current column
 9:         $R_{1:k-1,k} \leftarrow A_{1:k-1,i}$
10:         $A_{k:m,i+1:n} \leftarrow$ apply_reflector($A_{k:m,i+1:n}, V_{k:m,k}, \tau_k$)
11:         $k \leftarrow k + 1$
12:     end if
13: end for
14: $V \leftarrow V_{1:m,1:k-1}$
15: $R \leftarrow R_{1:k-1,1:k-1}$
```

Figure 1 shows an example of the result of the execution of PAQR with the second and fourth columns skipped due to the failure of the rank deficiency criterion.

The left matrix in Figure 1 represents the V and R matrices returned by PAQR based on the original columns of A. In most modern implementations, the QR factorization is done *in-place*, which means that the output V and R overwrite the original input A. The array of zeros and ones represent the δ vector returned by PAQR. This vector stores the flagged (rejected) columns of A.

The right matrix in Figure 1 represents a compressed (whether explicitly or implicitly) representation of V and R, where only the linearly independent columns are kept.

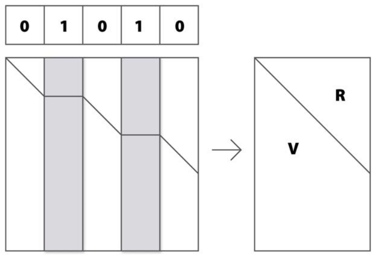

Fig. 1. Example of execution of PAQR factorization. Columns in grey are flagged as rejected.

B. Deficiency criteria

The decision of flagging a column of A to be ignored is a critical component of PAQR. The most criteria from the literature are costly, as they relate the singular values of the matrix. Hence, we present three alternatives. The first criterion [1, ch. 2 page 89] is the base one while the second and third ones are novel in that they are column-oriented. The first criterion needs a potentially costly *a-priori* estimate of the largest singular value of the matrix while the other two do not. They rely on the observation that the QR factorization applies on the trailing matrix a series of well-chosen orthogonal projections, relative to the previously factorized columns of A. Thus, when we construct the associated Householder reflector of a sub-column, its norm represents the projection of the corresponding original column of A on the space orthogonal to the sub-space spanning the previously linearly independent columns of A. If a vector is a linear combination of a set of other vectors, it should be included in the space spanned by these vectors. In practice, however, this condition can, and should, be relaxed, due to the existence of round-off errors with the limited precision arithmetic representation of the matrix elements.

The first deficiency criterion, the most costly one, requires the computation of the 2-norm of A:

$$|R_{k,k}| < \alpha * \|A\|_2. \tag{11}$$

Here $|R_{k,k}|$ is the norm of the Householder vector constructed at iteration k (and stored on the k^{th} diagonal of R), $\|A\|_2$ is the 2-norm of A, equivalent to the largest singular value of A (σ_1). The critical parameter α is chosen *a priori* as an input to the algorithm. Its value can be adjusted according to the numerical arithmetic precision (e.g. $\alpha \equiv \epsilon$). Given that the computation of this quantity can be expensive, a possible alternative to Equation (11) is the following:

$$|R_{k,k}| < \alpha \times \max_{1\leq j\leq n} \|A_{:,j}\|. \tag{12}$$

As the column with the largest 2-norm of a matrix A is in general a good approximation of the largest singular value σ_1 [15].

We propose the second deficiency criterion that is simpler:

$$|R_{k,k}| < \alpha \times \|A_{:,k}\| \tag{13}$$

where $A_{:,k}$ is the k^{th} column of A. The cost induced by this criterion corresponds to computing the norm of each column of A which is performed only once (at the beginning of the factorization). This is not the case for QRCP in which the norm is computed at each iteration to guarantee the best column is pivoted to the leading position. The idea for this deficiency criterion is that, once the previous Householder reflections have been applied, if the norm of the new constructed Householder reflection is small relative to its initial value, it is a linear combination of the previous $k-1$ reflections and is thus rejected.

The third deficiency criterion we propose takes into account how the QR factorization operates: at iteration k, the k^{th} column of A was the target of updates from only the columns on the left. Computing the largest singular values spanning the initial k columns of A is infeasible, so we propose the following criterion only in conjunction with the approximation mentioned above:

$$|R_{k,k}| < \alpha \times \max_{1\leq j\leq k} \|A_{:,j}\|. \tag{14}$$

C. Known Limitations

Although in practical applications PAQR is able to reduce the rank-deficiency of a matrix enough to drastically improve the forward error of a rank-deficient least-squares system, it can be sensitive to the input column order. Because PAQR does not pivot this can affect the number of columns kept in R_{11}. However, the columns of R_{11} still contain the span of A, and it remains correct to reject the columns in R_{22} as they are flagged as linear combinations of the factorized columns in R_{11}.

We now present in Equation (15) a synthetic corner case on which PAQR does not detect and remove any columns but for which the forward error still grows beyond control of our proposed criteria. We call these *Cliff* matrices, due to the pattern of their singular value spectrum: mostly constant singular values drop sharply off a numerical "cliff" for the smallest ones. We define them formally as:

$$\text{Cliff}(m,n,\alpha)_{i,j} = \begin{cases} \sqrt{\frac{1-(\max(m,n)\times\alpha)^2}{j-1}} & \text{if } i < j \\ \max(m,n)\times\alpha & \text{if } i = j \\ 0 & \text{if } i > j \end{cases}. \tag{15}$$

A practical example of this matrix corresponds to the matrix Gks (see Section V for application matrices).

Given the uniqueness of the QR decomposition of a matrix (up to the signs of the diagonal elements of R), the product of any orthonormal matrix by this Cliff matrix will generate a QR factorization whose R factors are the Cliff matrices. By construction, the norm of each column of a Cliff matrix is 1. Moreover, deficiency criteria (13) and (11) are both violated at every iteration of PAQR. This means that no column will be flagged as rejected, and that the PAQR factorization will be equivalent to the QR factorization. Unfortunately, from the experimental point of view, the forward error grows with n (the number of columns of a Cliff matrix) relative to the least-squares solution. The accumulation of errors in the computed solution by either QR or PAQR produces `NaNs` (Not-a-Number). Excluding such edge cases, PAQR delivers stable results with high numerical accuracy in practical application cases, as we show in Section V.

IV. Implementations

We propose several implementations of PAQR, each of which targeting a different computer architecture: sequential implementation, designed for LAPACK [24] (Section IV-A), batched GPU implementation, designed for MAGMA (Section IV-B), and distributed-memory implementation, designed for ScaLAPACK [6] (Section IV-C). Note that in the remaining part of the paper, we use Householder orthogonalization as our method of choice for constructing reflectors.

A. LAPACK Implementation (sequential)

A current prerequisite of PAQR (compared to QR) is the computation of the column norms of A. Our future work can benefit from *randomized SVD* or iterative methods to quickly approximate the 2-norm of A costing $O(n^2)$ operations. As of now however, we consider the computation of the 2-norm of each column of A regardless of the selected α.

Modern linear algebra libraries rely on *blocking* strategies to optimize their performance. While memory-bound Level 1 BLAS (scalar-based) and Level 2 BLAS (vector-based) operations occur within limited size blocks (e.g., panels), the majority of the operations is described through and carried out by highly efficient computation-bound Level 3 BLAS operations which are matrix-based.

Inside a panel, special attention must be paid to the computation of Householder vectors, as the LAPACK implementation applies a post-processing in case the computed norm of the Householder vector is smaller than a machine precision threshold. For this reason, the PAQR deficiency criterion is checked before the potential application of this post-processing, as the computation of the Householder norm may be artificially inflated by an appropriate adaptive scaling factor.

Moreover, in the presence of previously rejected columns, the number of effective Householder vectors computed and stored is smaller than the current number of columns of A in the panel. Hence, every new Householder vector would need to be stored at a different location in memory than its origin. Traditionally, the computation of the Householder vectors requires the loading of a sub-column (of A) in registers, scaling, then storing at the original location in memory (representing V). In our implementation, we avoid an unnecessary copy of this vector to its final destination by storing the result of its scaling at its final correct location right away. This is achieved by merging the `xSCAL` and `xCOPY` operations into a `xSCALCOPY` routine, all Level 1 BLAS.

Outside of a panel, the Householder vectors are stored contiguously in memory, and hence the efficient `xLARFT` and `xLARFB` routines can still be used to build the blocking factor, the T matrix, and update the trailing matrix, respectively. The potentially smaller number of Householder vectors to be applied should be the major source of speedup of PAQR over QR.

While the V matrix can be packed on the fly using the above-mentioned optimization, the R matrix is sparse. Indeed, the rejected columns are flagged but remain present inside the R matrix. Hence, two strategies may be applied to use this sparse R matrix for the least-squares solution phase. The first one is to compact R, either during the factorization, or as a post-treatment. The drawback, however, is the extra memory traffic of potentially the whole R matrix, even in the presence of a single rejected column. An extreme example is when the second column of A is rejected. The second strategy is to keep R sparse, but develop a tailored `xTRSM` routine that can accommodate this sparsity pattern.

Figure 1 shows a representation of R resulting from the PAQR factorization using the first (right) and the second strategy (left).

B. MAGMA Implementation (batched GPU mode)

For GPU accelerators, we developed a kernel to operate on a large set of relatively small matrices in parallel, which is often called *a batch setting*. The kernel takes as input an array of pointers that belong to independent matrices of the same size. Each matrix $A \in \mathbb{R}^{m\times n}$ is assumed to satisfy $m \geq n$. The corresponding output is $RV_{m\times\hat{n}}$ such that $\hat{n} \leq n$. Each RV matrix is condensed such that the $\hat{n}$ columns are adjacent to each other and aligned to the left of the matrix. The input matrices must be of the same size, but can have different degrees of rank deficiency. An additional output is δ, the array of flags of each matrix that point to the column indices that have been rejected during the factorization.

The batch PAQR implementation uses one kernel to perform the whole factorization. Each matrix is assigned to one thread-block. The main advantage of this approach is the optimal memory traffic, since each matrix is read and written exactly once. The implementation applies to small matrices that fit in the GPU's shared memory. While the register file provides a faster data access, the shared memory is more flexible, and is our choice for the kernel design. Since detecting rank deficiency occurs at run time, it is non-trivial to maintain constant compile-time indexing of the register file, which is necessary to avoid register spilling. The kernel implements Algorithm 2 in an unblocked manner, which means that the application of the Householder reflectors are performed one column at a time, thus eliminating the need for constructing the T factor. The kernel works with any number of threads in the range $[n{:}m]$.

There are two main bottlenecks of the design. The first one is computing the norm of the current column. The second is the matrix-vector multiply (v^TA) during the application of the elementary Householder reflector $(I - \tau vv^T)A$, which requires a reduction across the columns of A. At each iteration, the norm of the current column is computed using a standard tree reduction in the shared memory of the GPU. If the computed norm is less than a given threshold (defined by the user through the kernel interface), the whole iteration is skipped and the corresponding flag is set. Otherwise, the kernel proceeds with the application of the Householder reflector to the trailing submatrix. The update step begins with the computation of the v^TA product, for which threads re-organize themselves evenly across the remaining columns of A, and an equivalent number of independent tree reductions are executed all in parallel in shared memory. The output of the product is scaled with τ and stored in a shared memory vector Y, which is used to compute the remaining rank-1 update ($A = A - v \times Y$). As an example, if 64 threads are used to factorize a 128×8 matrix, the first iteration begins with a tree reduction using all 64 threads to compute the norm of 128-element vector. The 64 threads are then reorganized into 7 groups of 9 threads, with one thread remaining idle, to compute the v^TA product. In this case, 7 independent tree reductions are performed, each one using 9 threads to reduce a 128-element vector. Out of each group, one thread writes the corresponding element of Y. The rank-1 update is performed using one thread per row. If there are more rows than threads, a round-robin scheme is used. The kernel interface exposes two important tuning parameters to the user. The first is α, the parameter used in the deficiency criteria, which controls the numerical behavior, and also affects the performance, of the batch PAQR kernel. The second is the number of threads used in the factorization, which controls the occupancy and the performance of the kernel as well.

C. ScaLAPACK Implementation (distributed-memory mode)

ScaLAPACK has a similar structure to LAPACK, in the sense that the high-level routines resemble a sequential implementation, while internally they rely on lower-level libraries to seamlessly handle inter-process communication.

A first major difference of this distributed-memory implementation, compared to the sequential or shared-memory one, is that A is distributed over the MPI processes following a *2D-block-cyclic* scheme as shown in Figure 2. Given the characteristics of PAQR (some Householder vectors may contain more rows than would have otherwise been the case in QR), the communication pattern of the factorization might differ. Indeed, some processes may be involved in the communication and computation relative to a panel while these processes would not have been involved otherwise. However, this difference does not increase the overall volume of communication and/or computation.

A second major difference is that the block of Householder vectors computed within each panel is broadcast from the set of panel processes to the set of processes updating the trailing matrix. While the number of vectors to be communicated is deterministic in QR (corresponding to the size of the panel) this number is dynamic in PAQR, as it depends on the number of rejected columns, and should be communicated together with the Householder vectors. While the reduction in computation is the major source of speedup of PAQR over QR in a shared-memory environment, the reduction in communication volume of the Householder vectors is an important additional source of speedup in the distributed-memory setting.

Figure 2 depicts a representation of R and V at the end of a PAQR factorization.

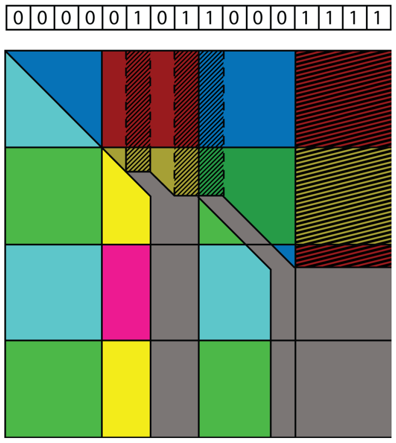

Fig. 2. PAQR on a 2D-block cyclic matrix in distributed-memory environment. The dark colors correspond to R. The light colors correspond to V. Every color (blue, red, green, yellow) correspond to a different process in the grid. The dashed area corresponds to rejected columns in R. The grey areas correspond to unused space freed-up by PAQR.

V. Experiments

This section presents the experiments comparing PAQR with QR and QRCP.

A. Experimental Setting

1) Matrices: Three sets of matrices were used: (a) Test matrices; (b) Weighted least-squares (WLS) matrices; and (c) Quantum many-body matrices.

a) Test matrices: The first set of matrices summarized in Table I were used [17] for the validation of *Communication Avoiding Rank Revealing QR* (CARRQR). The *Rand* and *Vandermonde* matrices were added, while the *Gks*, *H-C*, *Scale* and *Kahan* matrices were omitted.

b) Weighted least-squares (WLS) matrices: The second set of matrices consists of Vandermonde-like matrices that can be used for interpolating 3D polynomials from scattered data using an application of the *weighted least-squares* (WLS) algorithm [31]. These WLS interpolation matrices are derived from finite-volume discretizations on irregular meshes, where m cells, each with n geometric moments, are used to calculate "stencils" X that determine polynomial coefficients:

$$W\,A\,X \approx W\,I \tag{16}$$

TABLE I
TEST MATRICES USED IN SECTION V

No.	Matrix	Description
1	Rand	rand function in MATLAB generating random matrices.
2	Vandermonde	vander function in MATLAB generating Vandermonde matrices.
3	Baart	Discretization of the 1^{st} kind Fredholm integral equation [25].
4	Break-1	Break 1 distribution, matrix with prescribed singular values [26].
5	Break-9	Break 9 distribution, matrix with prescribed singular values [26].
6	Deriv2	Computation of second derivative [25].
7	Devil	The devil's stairs, a matrix with gaps in its singular values [27].
8	Exponential	Exponential Distribution, $\sigma_1 = 1$, $\sigma_i = \sigma_i - 1$ $(i = 2, ..., n), \alpha = 10 - 1/11$ [26].
9	Foxgood	Severely ill-posed test problem [25].
10	Gks	An upper-triangular matrix whose j-th diagonal element is $1/\sqrt{j}$ and whose i, j element is $-1/\sqrt{j}$, for $j > i$ [28], [29].
11	Gravity	1D gravity surveying problem [25].
12	H-C	Matrix with prescribed singular values, see description in [30].
13	Heat	Inverse heat equation [25].
14	Phillips	Phillips' famous test problem [25].
15	Random	Random matrix $A = 2 \times \text{rand}(n) - 1$ [29].
16	Scale	Scaled random matrix, a random matrix whose i-th row is scaled by the factor $\theta i/\theta$ [29]. We choose $\theta = 10 \cdot$.
17	Shaw	1D image restoration model [25].
18	Spikes	Test problem with a "spiky" solution [25].
19	Stewart	Matrix $A = U\Sigma V^T + 0.1\sigma 50 \times rand(n)$ [27].
20	Ursell	Integral equation with no square integrable solution [25].
21	Wing	Test problem with a discontinuous solution [25].
22	Kahan	Kahan matrix.

for I being the identity matrix. Note that this is a least-squares system with multiple solutions $X \in \mathbb{R}^{n\times n}$ for multiple right-hand sides with $W \in \mathbb{R}^{m\times m}$ being a diagonal weight matrix. The weight matrix is designed to decay rapidly to emphasize closer values over distant ones in the interpolation, which can create very small row scaling. This results in least-squares systems that can have very poor conditioning, due to small weights and cells being arbitrarily small or close together, which can create matrix entries beyond the limits of floating point precision: $O(\gamma^m)$, for any $\gamma > 0$. γ are "m-th order" volume moments, meaning if the smallest dimension is γ, there could be two rows that differ by as little as γ^m. If there exist insufficient or co-linear cells, this may also prevent determination of some of the coefficients, making the interpolation matrix A rank-deficient. Finally, to maintain uniform matrix size for the entire batch, missing interpolation data are replaced with zero-padded rows, which may occur in any part of the matrix.

c) Quantum many-body matrices: The third set of matrices comes from quantum many-body problems. Fundamentally, solution of these problems for molecular systems requires manipulation of a high-dimensional tensor which describes the interactions between electrons: the Coulomb tensor, $g_{pq,rs}$, $p, q, r, s \in [0, n)$. Despite its large dimension, which grows as $O(N_A^2)$ with system size (N_A), the Coulomb tensor is inherently low-rank, and can be straightforwardly shown that it admits a matrix rank which grows as $O(N_A)$ with system size when expressed in an atom-centered basis. This low-rank character has sparked research efforts dedicated to the construction and manipulation of data-sparse representations of the Coulomb tensor. Projection, grid, and local orbital methods have the benefit of exhibiting a relatively low communication overhead but do not produce the most compact representations. Matrix factorizations generally produce much more compact representations, but are accompanied by a much higher computational complexity and communication requirements which often complicate their usage on massively parallel architectures. Here, we examine the application of PAQR to produce low-rank representations of a representative set of Coulomb tensors generated from a range of molecular systems.

The Coulomb tensor has a natural matrization in $\mathbb{R}^{n^2\times n^2}$ by combining adjacent indices $g_{pq,rs} \to g_{i,j}$, $i, j \in [0, N)$ with $N = n^2$. We have applied our methodology to three molecular test cases, all calculations were carried out within the NWChemEx program using either the 6-31G [32], [33] or 6-31G(d) [34] atom-centered Gaussian basis sets. These cases are: a Uracil trimer (36 atoms, $N = 57,600$ within 6-31G), 5-mer (60 atoms, $N = 160,000$ within 6-31G), and the Beta Carotene molecule (96 atoms, $N = 506,944$ within 6-31G(d)).

B. Experimental results

1) Accuracy: Table II summarizes the numerical accuracy tests of PAQR compared to that of QR and QRCP. These results are obtained with our MATLAB implementation of PAQR using the set of Test matrices Table I. The QR and QRCP implementations are the MATLAB's own version of the corresponding algorithms and were invoked as [q,r] = qr(A) and [q,r,p] = qr(A), respectively. All matrices are of size 1000×1000. For each matrix A, random solution vectors $\hat{x}$ are generated. The corresponding right-hand sides b are computed as $b = Ax$. This ensures the existence of a valid solution to the system of equations $Ax = b$. The triangular solve routine ($\mathtt{xTRSM}$) is used on the R factors returned by the three variants. While QR returns a full R factor, PAQR and QRCP return a truncated R factor, that is supposed to only capture the (approximate) rank.

The forward and backward errors are already defined in Equation (7) and Equation (8), respectively. The *orthogonality error* corresponds to:

$$\frac{||A^T(Ax - b)||_2}{||A||_2^2}. \tag{17}$$

This metric is more suited to least-squares problems compared to the backward error which is more appropriate for systems of linear equations.

Among the 22 test matrices, seven are full-rank (*Rand, Break-1, Break-9, Deriv2, Phillips, Random, Stewart*), while the others exhibit varying degrees of rank deficiency.

From our preliminary tests, we notice that all the deficiency criteria presented in Section III-B give similar numerical results except for the GKS matrix. Therefore, in the remaining of the paper we use the deficiency criterion from Equation (13) and, unless otherwise mentioned, we set $\alpha = m \times \epsilon$, where m is the row-space dimension of the matrix.

All of the following PAQR results use the first deficiency criteria (Equation (14)) for its robustness, as the other criteria are only approximations.

First, all methods on all matrices have a backward error of the order of (or smaller than) the machine precision ϵ. This

TABLE II

ACCURACY RESULTS COMPARING PAQR WITH QR AND RRQR BASED ON FORWARD, BACKWARD, AND ORTHOGONALITY ERRORS ON THE SET OF TEST MATRICES. FOR DOUBLE PRECISION ARITHMETIC, ERROR SHOULD BE AROUND 10^{-16}.

Matrix	$\kappa_2(A)$	Forward error			Backward error			Orthogonality error			$Rncol$	rank(R)	
		QR	PAQR	QRCP	QR	PAQR	QRCP	QR	PAQR	QRCP	PAQR	PAQR	SVD
Random*	10^{+03}	10^{-14}	10^{-13}	10^{-14}	10^{-16}	10^{-15}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	1000	1000	1000
Rand*	10^{+04}	10^{-13}	10^{-12}	10^{-12}	10^{-16}	10^{-15}	10^{-16}	10^{-15}	10^{-16}	10^{-15}	1000	1000	1000
Deriv2	10^{+06}	10^{-09}	10^{-08}	10^{-10}	10^{-15}	10^{-14}	10^{-16}	10^{-15}	10^{-15}	10^{-14}	1000	1000	1000
Stewart*	10^{+06}	10^{-11}	10^{-10}	10^{-11}	10^{-16}	10^{-16}	10^{-16}	10^{-15}	10^{-15}	10^{-15}	1000	1000	1000
Phillips	10^{+10}	10^{-07}	10^{-06}	10^{-06}	10^{-16}	10^{-15}	10^{-16}	10^{-15}	10^{-15}	10^{-14}	1000	1000	1000
Break-1*	10^{+11}	10^{-05}	10^{-04}	10^{-05}	10^{-16}	10^{-15}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	1000	1000	1000
Break-9*	10^{+11}	10^{-05}	10^{-04}	10^{-04}	10^{-16}	10^{-15}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	1000	1000	1000
Ursell	10^{+13}	10^{-03}	10^{-03}	10^{-01}	10^{-16}	10^{-15}	10^{-15}	10^{-15}	10^{-14}	10^{-15}	1000	999	999
H-C*	10^{+13}	10^{-04}	10^{-03}	10^{-01}	10^{-16}	10^{-15}	10^{-14}	10^{-15}	10^{-16}	10^{-14}	1000	999	999
Scale*	10^{+17}	10^{-12}	10^{+00}	10^{+00}	10^{-16}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	914	794	802
Kahan	10^{+17}	10^{+01}	10^{-02}	10^{-02}	10^{-16}	10^{-14}	10^{-16}	10^{-14}	10^{-14}	10^{-15}	999	999	999
Baart	10^{+19}	10^{+02}	10^{+01}	10^{+01}	10^{-17}	10^{-15}	10^{-14}	10^{-14}	10^{-15}	10^{-14}	163	16	13
Devil*	10^{+19}	10^{+01}	10^{+00}	10^{+00}	10^{-16}	10^{-15}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	469	421	440
Exponential*	10^{+19}	10^{+02}	10^{+00}	10^{+00}	10^{-17}	10^{-15}	10^{-14}	10^{-14}	10^{-15}	10^{-14}	152	136	140
Foxgood	10^{+21}	10^{+03}	10^{+00}	10^{+00}	10^{-17}	10^{-16}	10^{-14}	10^{-13}	10^{-14}	10^{-11}	71	31	30
Gks	10^{+21}	10^{+280}	10^{+280}	10^{-02}	10^{-19}	10^{-19}	10^{-15}	10^{+262}	10^{+262}	10^{-15}	1000	999	999
Gravity	10^{+20}	10^{+03}	10^{+00}	10^{+00}	10^{-18}	10^{-16}	10^{-15}	10^{-14}	10^{-14}	10^{-12}	152	44	45
Shaw	10^{+20}	10^{+03}	10^{+00}	10^{+00}	10^{-18}	10^{-17}	10^{-16}	10^{-13}	10^{-15}	10^{-12}	77	19	20
Spikes	10^{+20}	10^{+03}	10^{+02}	10^{+01}	10^{-18}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	56	31	31
Wing	10^{+21}	10^{+05}	10^{+01}	10^{+01}	10^{-20}	10^{-16}	10^{-15}	10^{-13}	10^{-14}	10^{-12}	32	8	8
Vandermonde	10^{+22}	10^{+70}	10^{+00}	10^{+00}	10^{-18}	10^{-15}	10^{-15}	10^{+54}	10^{-15}	10^{-11}	103	42	43
Heat	10^{+232}	10^{+215}	10^{+00}	10^{+00}	10^{-230}	10^{-15}	10^{-14}	10^{-15}	10^{-15}	10^{-13}	987	588	588

means that all methods correctly minimize the least-squares error of the system.

Second, on the set of full-rank matrices, all methods have similar forward errors. We note a slight discrepancy of no more than an order of magnitude worse for PAQR. This is attributed to the fact that our MATLAB PAQR implementation differs from the MATLAB QR and QRCP implementations.

Third, the key result of this paper, on the set of rank-deficient matrices, PAQR exhibits a much more stable behaviour than QR and similar behavior to QRCP in terms of forward error. For instance, we can observe stark differences between PAQR and QR on the *Vandermonde* and *heat* matrices.

Fourth, as expected, PAQR does not remove any column from R when A is full-rank. More importantly, when A is rank-deficient, the number of retained (non-flagged) columns remains greater than the rank of A. For instance, in the case of the *heat* matrix, PAQR only flags 13 columns, while the true rank of the matrix is 588. Nevertheless, the removal of these few columns is still enough to lead to an accurate solution (forward error of 1.01 compared to $1.40 \times 10^{+215}$ with QR).

Among these matrices, we want to discuss two that are of interest: Gks and Scale. The Gks matrix is an example of the pathological case presented in Section III-C. The rank of this matrix is 999 and its spectrum reveals that the first $n-1$ singular values range from 26.1 and 0.041 while the smallest singular value is equal to 6.6×10^{-20}. In that case, as expected PAQR is not able to detect the single column that QRCP permutes at the end and then is removed before the call to `TRSM`. Note that when using the criterion one (more strict), PAQR gives similar results to QRCP. The Scale matrix is a perfect illustration that in some cases QRCP can fail to reveal the rank. Here, the spectrum does not have a gap and so the numerical rank 802 is more sensitive to small roundoff errors and approximations. Therefore, the truncation based on the diagonal elements of R causes both PAQR and QRCP to fail, whereas the classical QR factorization does not.

2) PAQR's Efficiency:

a) LAPACK Implementation: We now compare the performance of PAQR with QR and QRCP in the LAPACK implementation. Given that this implementation is sequential, we want to highlight the importance of the location of the rejected columns in the matrix. To this end, we generate random matrices of size $10,000 \times 10,000$ with the following characteristics: A_{full}, full-rank; A_{beg}, where the first 5000 columns are set to zero; A_{mid}, where the middle 5000 columns are set to zero; A_{end}, where the last 5000 columns are set to zero.

Table III summarizes the runtimes on one core of a DGX A100 [35] server. We use 1 AMD EPYC 7742 CPU core; MKL 2019.0.3 library as the Level i BLAS mplementation; and GCC 7.3.0 compiler. As expected, the runtime of PAQR is similar to that of QR on the full-rank matrix. Moreover, on the rank-deficient matrices, and on matrices of same size and same amount of rejected columns, the performance of PAQR improves with more rejected columns appearing at the beginning of the matrix.

b) Batch PAQR factorization on GPUs: We tested the GPU-based batch PAQR factorization on two sets of the set of WLS matrices. Since the current kernel design caches the entire matrix in the shared memory of the GPU, we show only two matrix sizes which satisfy this condition. For each set, 1000 matrices of the same size, but different ranks, were tested. Table IV shows the performance of the batched PAQR

TABLE III
LAPACK PERFORMANCE. IMPACT OF THE LOCATION OF REJECTED COLUMNS ON PAQR FACTORIZATION PERFORMANCE.

Method	Time (seconds)			
	A_{full}	A_{beg}	A_{mid}	A_{end}
QR	138			
PAQR	138	129	89	43
QRCP	163	211	209	211

factorization on an NVIDIA Tesla A100 GPU and on the AMD Instinct MI100 GPU. The results correspond to experiments conducted in double precision. We use cuBLAS and hipBLAS as the reference implementations for a standard batch QR factorization. The performance of these two libraries does not take advantage of any rank deficiency in the matrices, so their performance is oblivious to this property. We show the timings of a regular QR kernel of our design qr_gpu for full rank matrices that are randomly generated, and then the paqr_gpu kernel timing on the WLS set of matrices. The paqr_gpu kernel is superior to the reference kernels in every test case.

The following results are obtained for the 27×20 and 125×56 matrices, respectively. On the A100 GPU, the speedups against cuBLAS increase from $2.7\times$ (resp. $2.2\times$) for qr_gpu to $2.9\times$ (resp. $2.3\times$) for paqr_gpu. Note that varying timings for paqr_gpu could be observed based on the pattern of rank deficiency, but it should never be slower than qr_gpu. On the MI100 GPU, the speedups against hipBLAS increase from $7.8\times$ (resp. $1.3\times$) for qr_gpu to $8.7\times$ (resp. $1.4\times$) for paqr_gpu. In order to justify the significant speedups against the vendor libraries for the full rank case, we profiled cuBLAS and hipBLAS for the aforementioned experiments. cuBLAS launches one kernel to perform the whole QR factorization, which is similar to our batch qr_gpu code. However, it uses only 63 thread-blocks for 1000 matrices of size 27×20, and 125 thread-blocks for 1000 matrices of size 125×56. Using fewer thread-blocks indicates that one thread-block processes multiple matrices, which affects the kernel's overall occupancy on the GPU. hipBLAS launches several kernels to perform the factorization, which causes serious overhead in terms of unnecessary global memory traffic. We also observe a significant drop in the performance gain for the set of 125×56 matrices on the MI100 GPU. We believe this is mainly due to the relatively small shared memory on the MI100 (64KB), compared to the A100 (192KB). paqr_gpu requires about 57KB for the larger matrix set, which means that each compute unit in the MI100 GPU can host only one matrix at a time. On the A100, assuming that the CUDA runtime takes the correct scheduling decision, three matrices can be factorized on the same multiprocessor simultaneously.

c) ScaLAPACK Implementation: We now compare the performance of PAQR with QR and QRCP in the ScaLAPACK implementation for the set of Quantum many-body matrices.

Table V shows factorization timings gathered on the Oak Ridge National Laboratory Summit Supercomputer [36]. We use 42 *Power9* CPU cores per node, and do not rely on the GPUs, as ScaLAPACK cannot take advantage of them;

TABLE IV
PERFORMANCE OF THE BATCH PAQR IN DOUBLE PRECISION ON THE A100 GPU USING CUDA-11.6, AND ON THE MI100 GPU USING ROCM-5.0. RESULTS ARE SHOWN FOR TWO DIFFERENT SIZES OF THE WLS SET OF MATRICES. THE *Ref.* ENTRY REFERS TO CUBLAS OR HIPBLAS.

	Size	Ref. (μs)	qr_gpu (μs)	paqr_gpu (μs)
A100	27×20	294.13	109.33 (2.7×)	100.46 (2.9×)
	125×56	6215.2	2852.4 (2.2×)	2692.2 (2.3×)
MI100	27×20	1508.7	194.31 (7.8×)	174.21 (8.7×)
	125×56	11057.85	8561.13 (1.3×)	8039.02 (1.4×)

TABLE V
TIME TO FACTORIZATION RESULTS GATHERED ON SUMMIT FOR QR, QRCP, AND PAQR. BLANK CELLS CORRESPOND TO RUNS THAT DO NOT FIT IN MEMORY. $*$ CORRESPONDS TO IRRELEVANT RUNS. $-$ CORRESPONDS TO SKIPPED RUNS. ϵ CORRESPONDS TO DOUBLE-PRECISION.

#Nodes	Method	Matrix size			
		57 600		160 000	
		Time(s)	#Def cols	Time(s)	#Def cols
1	PAQR ϵ	160	45180		
	PAQR 10^{-8}	117	52073		
	QR	336			
	RRQR	4042			
2	PAQR ϵ	109	45217		
	PAQR 10^{-8}	75	52073		
	QR	243			
	RRQR	2087			
4	PAQR ϵ	54	44792	563	135583
	PAQR 10^{-8}	41	52073	454	150673
	QR	100		1779	
	RRQR	1050		*	
8	PAQR ϵ	38	44300	411	134036
	PAQR 10^{-8}	30	52073	220	150673
	QR	63		919	
	RRQR	556		*	
16	PAQR ϵ	31	43996	191	133930
	PAQR 10^{-8}	25	52073	136	150673
	QR	44		498	
	RRQR	304		*	
32	PAQR ϵ	23	43644	138	133005
	PAQR 10^{-8}	20	52073	96	150673
	QR	32		355	
	RRQR	174		2086	
64	PAQR ϵ	-		78	132636
	PAQR 10^{-8}	-		62	150673
	QR	-		162	
	RRQR	-		1103	

ESSL 6.3.0 library as the Level 3 BLAS implementation; IBM Spectrum 10.4.0.3 as the MPI library; and GCC 9.1.0 compiler. The QR and QRCP implementation used are the ScaLAPACK's PDGEQRF and PDGEQPF routines. PAQR was implemented based on the PDGEQRF routine. Different matrix sizes ($m = n = 57,600$ and $m = n = 160,000$) and varying number of nodes of Summit (1 to 64) were used. Whenever a data point is missing in Table V, this is due to either: (i) the run exceeded the maximum job allocation time (too expensive to compute in a reasonable amount of time); (ii)

or the problem size did not warrant the use of that many nodes. For PAQR, two rejection criteria thresholds were used: $1e^{-8}$ and $\epsilon = 2.22e^{-16}$ (64bit arithmetic machine-precision), under the assumption that, the larger the threshold, the lower the rank of the approximation obtained. Section V-A1 discusses that the true rank of such matrices scales asymptotically as the square root of their size. The column "Def cols" in Table V shows the number of rejected columns detected by PAQR for a given input tolerance.

In every case, PAQR removes a significant number of columns from the factorization. For the problem size of $57,600$, PAQR can achieve an overall speedup greater than 3x compared to traditional QR. When we compare the time to factorization of PAQR to that of QRCP, on one node, PAQR is almost 40x faster. This is a significant speedup where for this problem: PAQR removes $52,073$ rejected columns. For the problem size of $160,000$, we see PAQR can achieve similar results. PAQR on 32 nodes is over 20x faster than QRCP and almost 3.5x faster than QR. For this case, the deficiency criteria is defined as 10^{-8} and PAQR removes $150,673$ columns which corresponds to 94% of the number of columns of the input matrix.

Finally, we performed the PAQR factorization of the third problem from the Quantum many-body matrix set (Beta Carotene) of size $506,944$ on 128 nodes of the Summit supercomputer. The runtime was 1155 seconds. PAQR was able to flag and remove $393,805$ columns. This achievement of PAQR would not be possible to obtain with QRCP in any reasonable amount of time.

VI. Conclusion

This paper presented PAQR, an algorithm for the factorization of rank-deficient matrices arising from linear least-squares problems. We showed our method is well suited for large-scale problems. Indeed, PAQR is faster than QR and appears to be as numerically stable as QRCP. This technique can be implemented using different criteria to flag potentially rejected columns in rank-deficient matrices. These criteria are adaptable as they can be conveniently adjusted according to the type of arithmetic precision used and the type of application originating the matrix elements.

Preliminary tests have revealed the efficiency of the proposed algorithm and its robustness vis-à-vis a variety of practical use cases. Indeed, PAQR was implemented in three modes: sequential, GPU, and distributed-memory. It was applied on a variety of matrices of ranging sizes and arising in various application fields. Further examination of the behaviour of the algorithm will strengthen these results.

References

[1] A. Björck, *Numerical Methods for Least Squares Problems*. SIAM Philadelphia, 1996.

[2] C. Lawson and R. Hanson, *Solving Least Squares Problems*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[3] H. Avron, P. Maymounkov, and S. Toledo, "Blendenpik: Supercharging LAPACK's least-squares solver," *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1217–1236, 2010. [Online]. Available: http://dx.doi.org/10.1137/090767911

[4] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: The Johns Hopkins University Press, 2013.

[5] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer, "The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science," *Journal of Physics: Condensed Matter*, vol. 26, p. 213201, 2014.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997, http://www.netlib.org/scalapack/slug/.

[7] J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Software*, vol. 39, 2013.

[8] J. Xiao, M. Gu, and J. Langou, "Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations," in *24th IEEE International Conference on High Performance Computing, Data, and Analytics (HIPC), Jaipur, India*. IEEE, dec 2017, best Paper Award.

[9] J. Todd, "On condition numbers," *Programmation en Mathématiques Numériques*, vol. 7, no. 165, pp. 141–159, 1966, Éditions Centre Nat. Recherche Sci., Paris, 1968.

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[11] T. F. Chan, "Rank-revealing QR factorizations," *Linear Algebra and its Applications*, vol. 88/89, pp. 67–82, 1987.

[12] S. Chandrasekaran and I. Ipsen, "On rank-revealing QR factorizations," *SIAM Journal on Matrix Analysis and Applications*, vol. 15, 1994.

[13] Y. P. Hong and C.-T. Pan, "Rank-revealing QR factorizations and the singular value decomposition," *Mathematics of Computation*, vol. 58, no. 197, pp. 213–232, 1992. [Online]. Available: http://www.jstor.org/stable/2153029

[14] M. Gu and S. Eisenstat, "Efficient algorithms for computing a strong rank-revealing QR factorization," *SIAMX*, vol. 17, no. 4, pp. pp. 848–869, 1996.

[15] C. Bischof and G. Quintana-Orti, "Computing rank-revealing QR factorizations of dense matrices," *ACM Trans. Math. Soft.*, vol. 24, no. 2, pp. 226–253, 1998.

[16] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012. [Online]. Available: https://doi.org/10.1137/080731992

[17] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang, "Communication avoiding rank revealing QR factorization with column pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 1, pp. 55–89, 2015. [Online]. Available: https://doi.org/10.1137/13092157X

[18] P.-G. Martinsson, G. Quintana OrtÍ, N. Heavner, and R. van de Geijn, "Householder QR factorization with randomization for column pivoting (HQRRP)," *SIAM Journal on Scientific Computing*, vol. 39, no. 2, pp. C96–C115, 2017. [Online]. Available: https://doi.org/10.1137/16M1081270

[19] J. A. Duersch and M. Gu, "Randomized QR with column pivoting," *SIAM Journal on Scientific Computing*, vol. 39, no. 4, pp. C263–C291, 2017. [Online]. Available: https://doi.org/10.1137/15M1044680

[20] ——, "Randomized projection for rank-revealing matrix factorizations and low-rank approximations," *SIAM Review*, vol. 62, no. 3, pp. 661–682, 2020. [Online]. Available: https://doi.org/10.1137/20M1335571

[21] W. Hayes and K. R. Jackson, "A survey of shadowing methods for numerical solutions of ordinary differential equations," *Appl. Numer. Math.*, vol. 53, p. 299–321, 2005.

[22] R. M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods from the Viewpoint of Backward Error Analysis*. Springer, 2013.

[23] T. A. Davis, "Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049670

[24] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. D. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. Philadelphia, Pennsylvania, USA: SIAM, 1999.

[25] P. C. Hansen, “Regularization tools version 4.0 for Matlab 7.3,” *Numerical Algorithms*, vol. 46, no. 2, pp. 189–194, Oct 2007. [Online]. Available: https://doi.org/10.1007/s11075-007-9136-9

[26] C. H. Bischof, “A parallel QR factorization algorithm with controlled local pivoting,” *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 1, pp. 36–57, 1991. [Online]. Available: https://doi.org/10.1137/0912002

[27] G. W. Stewart, “The QLP approximation to the singular value decomposition,” *SIAM Journal on Scientific Computing*, vol. 20, no. 4, pp. 1336–1348, 1999. [Online]. Available: https://doi.org/10.1137/S1064827597319519

[28] G. Golub, V. Klema, and G. W. Stewart, “Rank degeneracy and least squares problems,” Stanford University, Tech. Rep., 1976.

[29] M. Gu and S. C. Eisenstat, “Efficient algorithms for computing a strong rank-revealing QR factorization,” *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 848–869, 1996. [Online]. Available: https://doi.org/10.1137/0917055

[30] D. A. Huckaby and T. F. Chan, “Stewart’s pivoted QLP decomposition for low-rank matrices,” *Numerical Linear Algebra with Applications*, vol. 12, no. 2-3, pp. 153–159, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.404

[31] D. Devendran, D. Graves, H. Johansen, and T. Ligocki, “A fourth-order Cartesian grid embedded boundary method for Poisson’s equation,” *Communications in Applied Mathematics and Computational Science*, vol. 12, no. 1, pp. 51–79, May 2017. [Online]. Available: https://doi.org/10.2140/camcos.2017.12.51

[32] R. Ditchfield, W. J. Hehre, and J. A. Pople, “Self-consistent molecular-orbital methods. IX. An extended Gaussian-type basis for molecular-orbital studies of organic molecules,” *J. Chem. Phys.*, vol. 54, pp. 724–728, 1971.

[33] W. J. Hehre, R. Ditchfield, and J. A. Pople, “Self-consistent molecular orbital methods. XII. further extensions of Gaussian-type basis sets for use in molecular orbital studies of organic molecules,” *J. Chem. Phys.*, vol. 56, pp. 2257–2261, 1972.

[34] P. C. Hariharan and J. A. Pople, “The influence of polarization functions on molecular orbital hydrogenation energies,” *Theor. Chim. Acta*, vol. 28, pp. 213–222, 1973.

[35] “Nvidia gtx a100 datasheet,” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[36] “Summit supercomputer specification,” https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.