Received: 5 March 2020 | Revised: 1 August 2022 | Accepted: 30 March 2023

DOI: 10.1002/cpe.7726

RESEARCH ARTICLE

WILEY

Computing rank-revealing factorizations of matrices stored out-of-core

N. Heavner[1] | P. G. Martinsson[2] | G. Quintana-Ortí[3]

[1]Department of Applied Mathematics, University of Colorado at Boulder, Boulder, Colorado, USA

[2]Department of Mathematics, University of Texas at Austin, Austin, Texas, USA

[3]Departamento de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Castellón, Spain

Correspondence
G. Quintana-Ortí, Departamento de Ingeniería y Ciencia de Computadores, Universitat Jaume I, Avda. Sos Baynat s/n, 12.071 Castellón, Spain.
Email: gquintan@uji.es

Funding information
National Science Foundation, Grant/Award Numbers: DMS-1620472, DMS-1952735, DMS-2012606; Office of Naval Research, Grant/Award Number: N00014-18-1-2354; Spanish Ministry of Science, Innovation and Universities, Grant/Award Numbers: RTI2018-098156-B-C54, PID2021-123627OB-C55; Department of Energy ASCR, Grant/Award Number: DE-SC0022251

Summary

This paper describes efficient algorithms for computing rank-revealing factorizations of matrices that are too large to fit in main memory (RAM), and must instead be stored on slow external memory devices such as disks (out-of-core or out-of-memory). Traditional algorithms for computing rank-revealing factorizations (such as the column pivoted QR factorization and the singular value decomposition) are very communication intensive as they require many vector-vector and matrix-vector operations, which become prohibitively expensive when data is not in RAM. Randomization allows to reformulate new methods so that large contiguous blocks of the matrix are processed in bulk. The paper describes two distinct methods. The first is a blocked version of column pivoted Householder QR, organized as a "left-looking" method to minimize the number of the expensive write operations. The second method results employs a UTV factorization. It is organized as an algorithm-by-blocks to overlap computations and I/O operations. As it incorporates power iterations, it is much better at revealing the numerical rank. Numerical experiments on several computers demonstrate that the new algorithms are almost as fast when processing data stored on slow memory devices as traditional algorithms are for data stored in RAM.

KEYWORDS

blocked matrix computations, householder QR factorization, HQRRP factorization, out-of-core computation, partial rank-revealing factorization, randomized numerical linear algebra, randUTV factorization, rank-revealing factorization, shared-memory multicore processors, shared-memory multiprocessors

1 | INTRODUCTION

1.1 | Problem formulation

We consider the task of computing a rank-revealing factorization of a matrix that is so large that it must be stored on an external memory device such as a spinning or solid-state disk drive. In this case, the matrix is said to be stored *out-of-core*. Specifically, given an $m \times n$ matrix A, we seek a factorization of the form

$$\underset{m\times n}{A} = \underset{m\times m}{U}\ \underset{m\times n}{R}\ \underset{n\times n}{V^*}, \tag{1}$$

where U and V are unitary matrices, and where R is upper triangular. We use the term "rank-revealing" in an informal way, as meaning simply that for any k with $1 \leq k \leq \min(m, n)$, the truncation of (1) to the k dominant terms provides a rank-k approximation to A that is almost as good as the theoretically optimal one. To be precise, with $A_k = U(:, 1 : k)R(1 : k, :)V^*$, we ask that

$$\|A - A_k\| \approx \inf\{\|A - B\| \ : \ \text{where } B \text{ has rank } k\}.$$

Problems which make use of such a factorization include solving ill-conditioned linear systems, rank-deficient and total least squares problems,[1-3] subset selection,[4] and matrix approximation,[5,6] among others.

1.2 | Prior work

There are several well-established options for computing a rank-revealing factorization. If a full factorization is required, the singular value decomposition (SVD) and the QR decomposition with column pivoting (CPQR) are two popular options. The SVD provides theoretically optimal low rank approximations for many choices of matrix norms, but can be prohibitively expensive to compute. The most practical methods[7-9] are mainly based on Householder transformations. The CPQR usually reveals rank reasonably well (see, e.g., Reference 10 for a notable exception) and requires much less computational work than the SVD. The original method was developed by Golub and Kahan, and was later optimized and refined for current computers.[11-13] Neither factorization is easily implemented when the matrix is stored out-of-core, however. The traditional algorithms for computing both factorizations require many slow matrix-vector operations, which in turn necessitate many read operations (data brought from an external device to main memory) and write operations (data sent from main memory to an external device).

Some software efforts in computing factorizations of dense matrices stored in external devices include POOCLAPACK,[14-16] the SOLAR[17] library, a runtime to execute algorithms-by-blocks when the matrix is stored in disk,[18] as well as an out-of-core extension[19] to ScaLAPACK[20] that does not appear in the current version of the library. Even for these libraries, factorization routines are usually limited to Cholesky, unpivoted QR, and LU decompositions. A truncated SVD algorithm that is effective for out-of-core matrices was introduced by Halko et al.[21] This technique, however, only yields a *partial* factorization, and it requires that an upper bound for the target rank k is known in advance. Recent efforts have been able to compute the SVD of large dense matrices stored in an external device. Demchik et al.[22] employed randomization to compute the SVD, but no performance results for dense matrices were shown. Kabir et al.[23] used a two-stage approach to compute the SVD that reduces the dense matrix to band form in a first stage, which allowed to factorize matrices of up to dimension 100×100 k. To our knowledge, there are no currently available and widely used software for computing a full rank-revealing factorization out-of-core.

1.3 | Contributions of present work

This article describes two different algorithms for computing rank-revealing factorizations of matrices stored out-of-core. The first, `HQRRP_left`, is based on the `HQRRP` algorithm,[24] which uses randomization techniques to build a fully blocked column pivoted QR factorization. `HQRRP` relies largely on matrix-matrix operations, reducing the number of reads and writes that must occur between the external device and main memory (RAM). `HQRRP_left` further reduces the number of write operations required by redesigning `HQRRP` as a left-looking algorithm. Numerical experiments reveal that this reduction in the writing time is critical to the algorithm's performance, particularly when the data is stored on a spinning disk. (The `HQRRP` method is closely related to the technique in Reference 25.)

The second contribution of this article is an out-of-core implementation of the `randUTV` algorithm,[26] which uses randomization to build a rank-revealing UTV factorization (see, e.g., References 27 and 28 for a good introduction to this factorization). The out-of-core implementation, `randUTV_AB`, modifies `randUTV` in such a way as to achieve overlap of communication and floating point operations. The result, demonstrated with numerical experiments, is that `randUTV_AB` suffers only a small extra cost by storing the matrix in an external device with respect to the same factorization of a matrix stored in RAM.

The new techniques presented enable processing of very large matrices at modest cost. To illustrate, in July, 2022 a personal desktop with a 2TB high-performance SSD (like the one in the machine "ut" in Section 4.3.1) can be acquired for a few thousand dollars, whereas a workstation with an equivalent amount of RAM would be an order of magnitude more expensive. Since solid state storage technology is rapidly getting both cheaper and faster, we expect to see demand for out-of-core algorithms to continue to increase.

1.4 | Assessing the quality of a rank-revealing factorization

The term "rank-revealing factorization" has been used with slightly different meanings in the literature;[28-31] our requirement that the approximation error $\|A - A_k\|$ is close to optimal is invariably a part of the requirement, but in more theoretical papers the statement is often made precise by bounding how rapidly the discrepancy is allowed to increase as the matrix dimensions tend to infinity.

The focus of the current paper is how to efficiently implement algorithms that have already been published to make them work on matrices so large that they do not fit in the main memory of the computer. The precision at which they reveal rank has already been carefully studied: The randomized CPQR was analyzed in Reference 25, with additional numerical results presented in Reference 24. The conclusion of these papers was that the randomized pivoting reveals the numerical rank to roughly the same precision as classical pivoting. The randomized UTV factorization was studied in Reference 26, with additional results reported in Reference 32. The precision of `randUTV` depends on a tuning parameter q that controls the number of power iterations that are taken at each step. Higher q results in better precision, but a slower execution time. When no powering is done ($q = 0$), `randUTV` is about as good at revealing the rank as CPQR, with `randUTV` often having a slight edge. As q is increased to one or two, the precision very rapidly improves, and often gets to within striking distance of the optimal precision of the SVD. Moreover, the diagonal entries of the middle factor T often provide excellent approximations to the true singular values of the matrix.

1.5 | Outline of paper

In Section 2, we discuss the first out-of-core factorization, a rank-revealing QR factorization `HQRRP` which can be stopped early to yield a partial decomposition. Section 3 explores an out-of-core implementation of `randUTV`, an efficient algorithm for obtaining a rank-revealing orthogonal matrix decomposition. In Section 4, we present numerical experiments which demonstrate the performance of both algorithms. Finally, Section 5 contains the main conclusions of our work.

2 | PARTIAL RANK-REVEALING QR FACTORIZATION

In this section, we introduce an out-of-core implementation of a rank-revealing column pivoted QR factorization. In Subsections 2.1 and 2.2, we review a fully blocked algorithm for computing a column pivoted QR factorization, `HQRRP`. In Subsection 2.3, we discuss modifications of `HQRRP` which enhance its efficiency when the matrix is stored out-of-core.

2.1 | Overview of `HQRRP`

Consider an input matrix $A \in \mathbb{R}^{m \times n}$ with $m \geq n$. `HQRRP`[24] is a *blocked* algorithm for computing a column-pivoted QR factorization

$$\underset{m \times n}{A} = \underset{m \times m}{Q} \underset{m \times n}{R} \underset{n \times n}{P^*},$$

where Q is orthogonal, R is upper trapezoidal, and P is a permutation matrix.

The bulk of the algorithm's work is executed in a loop with $\lceil n/b \rceil$ iterations, where $1 \leq b \leq n$ is the *block size* parameter. For notational simplicity, for the remaining discussion it is assumed that n is a multiple of b so that $n = bp$ for some $p \in \mathbb{N}$. At the start of the `HQRRP` algorithm are the following initializations:

$$R^{(0)} = A, \quad Q^{(0)} = I, \quad P^{(0)} = I.$$

During the i-th iteration, matrices $Q^{(i)}$ and $P^{(i)}$ are constructed such that

$$R^{(i)} = (Q^{(i)})^* R^{(i-1)} P^{(i)},$$

where $R^{(i)}(:, 1 : ib)$ is upper trapezoidal. After p steps, $R^{(p)}$ is upper trapezoidal, and the final factorization can be written as

$$\begin{aligned} R &= R^{(p)} \\ P &= P^{(1)} P^{(2)} \cdots P^{(p)} \\ Q &= Q^{(1)} Q^{(2)} \cdots Q^{(p)}. \end{aligned}$$

Remark 1. In the case where $m > n$, it is often desirable to build an "economy-size" factorization $A = QRP^*$ involving a thin matrix Q of size $m \times n$. `HQRRP` can easily accommodate this, as it constructs Q as a sequence of Householder reflectors, which can be transformed to a thin orthonormal matrix using standard software. We also remark that in the case where $m \gg n$, it is generally advantageous to

follow standard practice of first computing an unpivoted QR factorization, and then applying the procedure described here to the resulting small square upper triangular factor.

2.2 | Choosing the orthogonal matrices

At the i-th step of `HQRRP`, consider the partitioning of $R^{(i-1)}$

$$R^{(i-1)} \rightarrow \begin{pmatrix} R_{11}^{(i-1)} & R_{12}^{(i-1)} \\ R_{21}^{(i-1)} & R_{22}^{(i-1)} \end{pmatrix},$$

where $R_{11}^{(i-1)}$ is of size $(i-1)b \times (i-1)b$. The permutation matrix $P^{(i)}$ is chosen to find b columns of $R_{22}^{(i-1)}$ that form a good choice for the next set of pivots. (Exactly what this means mathematically is a slightly intricate question, but a commonly used objective is that the volume of the parallelepiped spanned by chosen columns should be close to the volume spanned by the best possible choice of b columns. See References 33 and 34 for further details.) We can find such a selection by projecting the columns of $R_{22}^{(i-1)}$ into a lower dimensional space and cheaply finding b good pivot columns there. The steps for computing $P^{(i)}$ are thus as follows:

1. Draw a random matrix $G^{(i)} \in \mathbb{R}^{b \times (m-(i-1)b)}$, with i.i.d. entries drawn from the standard normal distribution.
2. Compute $Y^{(i)} = G^{(i)} R_{22}^{(i-1)} \in \mathbb{R}^{b \times (n-(i-1)b)}$.
3. Perform b steps of the traditional Golub-Businger column pivoted QR[11] on the matrix $Y^{(i)}$ to obtain $Y^{(i)} P_{22}^{(i)} = W_{\text{trash}} S_{\text{trash}}$, where $P_{22}^{(i)}$ is of size $n-(i-1)b \times n-(i-1)b$. (We use the subscript "trash" to indicate that a quantity is not actually used.)
4. Set

$$P^{(i)} = \begin{pmatrix} I & 0 \\ 0 & P_{22}^{(i)} \end{pmatrix}.$$

This method for selecting multiple pivot columns has shown itself to be effective and reliable, consistently producing factorizations that reveal the rank as well as traditional CPQR.[24]

Remark 2. There is an alternate "downdating" method for computing $Y^{(i)}$ during each step that reduces the asymptotic flop count of the `HQRRP` algorithm.[25] With this technique, `HQRRP` has the same asymptotic flop count as the unpivoted QR factorization and the CPQR; the reader may see Reference 24 for details. However, this downdating method will not be used in this article's primary implementation as the communication restrictions imposed by this downdating method make the basic scheme discussed in this section more practical.

Once $P^{(i)}$ has been computed, $Q^{(i)}$ is built with well-established steps:

1. Perform unpivoted QR factorization on $A_{22}^{(i-1)} P_{22}^{(i)}(:, 1:b)$ to obtain

$$A_{22}^{(i-1)} P_{22}^{(i)}(:, 1:b) = Q_{22}^{(i)} A_{22}^{(i)}(:, 1:b).$$

2. Set

$$Q^{(i)} = \begin{pmatrix} I & 0 \\ 0 & Q_{22}^{(i)} \end{pmatrix}.$$

2.3 | Executing `HQRRP` out-of-core

When the matrix to be factorized is large enough that it must be stored out-of-core, communication (I/O operations) costs become a major concern. While it is desirable to minimize all forms of communication, writing to an external device is typically more expensive than reading from it. In linear algebra, in every row, column, or block iteration, right-looking algorithms factorize the current row, column, or block and then update the rest of the matrix, which usually requires an overall cost of $\mathcal{O}(n^3)$ writes. In contrast, left-looking algorithms apply all the previous transformations to the

current row, column, or block and then factorize it, which usually requires an overall cost of only $\mathcal{O}(n^2)$ writes. When working on main memory, performances are only slightly different, but when working with matrices stored in an external device with slow writes, left-looking algorithms deliver higher performances.

The original HQRRP was a classical right-looking algorithm, and therefore it performed many write operations. We have designed a left-looking variation of the original HQRRP algorithm, so that the number of write operations are much smaller.

2.3.1 | Left-looking algorithms

Several standard matrix factorizations have been re-ordered as left-looking algorithms for out-of-core computations, largely with the goal of reducing certain I/O operations.[15,17,35,36]

As our new algorithm employs the QR factorization, a high level description of the left-looking algorithm for computing the QR factorization follows. Let $A \in \mathbb{R}^{n\times n}$ and b be a block size $1 \leq b \leq n$; for simplicity let $n = bp$, where $b, p \in \mathbb{N}$. Then for $i = 0, 1, \ldots, p-1$, the algorithm for computing the QR factorization is described by the following steps (indices start at zero):

1. $A_{\text{col}} \leftarrow A(:, ib : (i+1)b - 1)$.
2. $A_{\text{col}} \leftarrow Q^* A_{\text{col}}$.
3. Compute the unpivoted QR factorization of $A_{\text{col}}(ib : n-1, :)$ yielding Q_i, R_i.
4. $A(0 : ib-1, ib : (i+1)b-1) \leftarrow A_{\text{col}}(0 : ib-1, :)$.
5. $A(ib : n-1, ib : (i+1)b-1) \leftarrow R_i$.
6. $Q \leftarrow Q \begin{pmatrix} I & 0 \\ 0 & Q_i \end{pmatrix}$.

In this algorithm, Q is initialized as the identity matrix I, and after the algorithm is finished, A is overwritten with the upper triangular matrix R. In practice, Q is never formed, but instead the Householder vectors are stored in the lower triangular portion of R as it is computed. This overview omits other computational details as well, but clearly highlights the fact that only the column block of $A(:, ib : (i+1)b-1)$ is updated in every iteration. In contrast, the traditional right-looking algorithm updates the bottom-right block $A(ib : n-1, ib : n-1)$ during each iteration of the loop, thus requiring much more writing to disk.

2.3.2 | HQRRP as a left-looking algorithm

HQRRP as a left-looking algorithm follows the pattern of the unpivoted factorization discussed in Section 2.3.1, with the added complication of choosing and applying the permutation matrix P. We select the permutations using precisely the same procedure as in Section 2.2. Observe that the projection procedure used in the selection of P requires that the right-most columns of A be updated according to the elementary orthogonal matrices encoded in Q during each iteration. We must therefore choose whether to write out the updated columns of A to disk or repeat some of the arithmetic operations during each iteration. Since the main idea of a left-looking algorithm is to avoid writing to the right-most columns of A every iteration, we must accept the repetition of operations as part of HQRRP_left with the current technologies. Numerical experiments indicated that the I/O-saving benefits of HQRRP_left easily outweigh the costs of the extra flops as compared to the original right-looking HQRRP.

For full clarity, we explicitly specify the entire algorithm. Let $A \in \mathbb{R}^{n\times n}$ and b be a block size $1 \leq b \leq n$; for simplicity let $n = bp$, where $b, p \in \mathbb{N}$. Initialize $Q = P = I \in \mathbb{R}^{n\times n}$. Then for $i = 0, 1, \ldots, p-1$, the algorithm for computing the QR factorization is described by the following steps (indices start at zero):

1. $A_{\text{cur}} \leftarrow AP(:, ib : n-1)$.
2. $A_{\text{cur}} \leftarrow QA_{\text{cur}}$.
3. $A_{\text{col}} \leftarrow A_{\text{cur}}(ib : n-1, :)$.
4. Draw a random matrix $G^{(i)} \in \mathbb{R}^{b\times(n-(i-1)b)}$, with i.i.d. entries drawn from the standard normal distribution.
5. $Y \leftarrow G^{(i)} A_{\text{col}}$.
6. Perform b steps of the traditional Golub-Businger column pivoted QR[11] on Y to obtain $P_i, W_{\text{trash}}, S_{\text{trash}}$ such that $YP_i = W_{\text{trash}} S_{\text{trash}}$.
7. Compute the unpivoted QR factorization of $A_{\text{col}} P_i(:, 0 : b-1)$, yielding Q_i, R_i such that $A_{\text{col}} P_i(:, 0 : b-1) = Q_i R_i$.
8. $AP(0 : ib-1, ib : (i+1)b-1) \leftarrow A_{\text{cur}} P_i(0 : ib-1, 0 : b-1)$.
9. $AP(ib : n-1, ib : (i+1)b-1) \leftarrow R_i$.

10. $Q \leftarrow Q \begin{pmatrix} I & 0 \\ 0 & Q_i \end{pmatrix}$.
11. $P \leftarrow P \begin{pmatrix} I & 0 \\ 0 & P_i \end{pmatrix}$.

3 | FULL RANK-REVEALING ORTHOGONAL FACTORIZATION

In this section, we introduce an efficient implementation of a rank-revealing orthogonal decomposition for a matrix stored out-of-core. In Subsection 3.1, we review an efficient algorithm, `randUTV`, for building such a decomposition when the matrices are stored in main memory. In Subsection 3.2, we discuss some modifications of `randUTV` that optimize its efficiency in the out-of-core setting.

3.1 | Overview of `randUTV`

Let $A \in \mathbb{R}^{m \times n}$ with $m \geq n$. The `randUTV` algorithm[26] builds a rank-revealing UTV factorization of A, that is, a decomposition

$$\underset{m \times n}{A} = \underset{m \times m}{U} \underset{m \times n}{T} \underset{n \times n}{V^*}$$

such that U and V are orthogonal and T is upper triangular. The `randUTV` algorithm is blocked, so it proceeds by choosing first a block size b with $1 \leq b \leq n$ and performing its work inside a loop with $\lceil n/b \rceil$ iterations. For ease of notation, it is assumed that $n = bp$ for $b, p \in \mathbb{N}$. At the beginning, we initialize

$$T^{(0)} = A, \quad U^{(0)} = I, \quad V^{(0)} = I.$$

For the matrix $T^{(i)}$ (and analogously for matrices U and V), the following partitioning will be employed:

$$T^{(i)} \rightarrow \begin{pmatrix} T_{11}^{(i)} & T_{12}^{(i)} \\ T_{21}^{(i)} & T_{22}^{(i)} \end{pmatrix},$$

where $T_{11}^{(i)}$ is $ib \times ib$. At the i-th iteration, for $i = 1, \ldots, p$, we form matrices $T^{(i)}$, $U^{(i)}$ and $V^{(i)}$ as follows:

1. Construct an orthogonal matrix $\hat{V}_{22}^{(i)}$ such that its leading b columns span approximately the same subspace as the leading b right singular vectors of $T_{22}^{(i-1)}$.
2. Compute the unpivoted QR factorization of $T_{22}^{(i-1)} \hat{V}_{22}^{(i)}(:, 1:b)$ to obtain

$$T_{22}^{(i-1)} \hat{V}_{22}^{(i)}(:, 1:b) = \hat{U}_{22}^{(i)} R.$$

3. Compute the SVD of $\left(\hat{U}_{22}^{(i)}(:, 1:b)\right)^* T_{22}^{(i-1)} \hat{V}_{22}^{(i)}(:, 1:b)$, yielding

$$\left(\hat{U}_{22}^{(i)}(:, 1:b)\right)^* T_{22}^{(i-1)} \hat{V}_{22}^{(i)}(:, 1:b) = U_{\text{SVD}} D V_{\text{SVD}}^*.$$

4. Calculate $V^{(i)}$ with

$$V^{(i)} = \begin{pmatrix} I & 0 \\ 0 & \hat{V}_{22}^{(i)} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & V_{\text{SVD}} & 0 \\ 0 & 0 & I \end{pmatrix}.$$

5. Calculate $U^{(i)}$ with

$$U^{(i)} = \begin{pmatrix} I & 0 \\ 0 & \hat{U}_{22}^{(i)} \end{pmatrix} \begin{pmatrix} I & 0 & 0 \\ 0 & U_{\text{SVD}} & 0 \\ 0 & 0 & I \end{pmatrix}.$$

6. Calculate $T^{(i)}$ with

$$T^{(i)} = (U^{(i)})^* T^{(i-1)} V^{(i)}.$$

Once all $\lceil n/b \rceil$ iterations have completed, we may compute the final factors with

$$\begin{aligned} T &= T^{(p)}, \\ U &= U^{(1)} U^{(2)} \cdots U^{(p)}, \\ V &= V^{(1)} V^{(2)} \cdots V^{(p)}. \end{aligned}$$

This leaves only the question of how the matrix $\hat{V}_{22}^{(i)}$ is formed in step 1 above. The method, inspired from work in randomized linear algebra including References 37-39, is the following:

1. Draw a random matrix $G^{(i)} \in \mathbb{R}^{(m-ib)\times b}$, with i.i.d. entries drawn from the standard normal distribution.
2. Compute the unpivoted QR factorization of $((T_{22}^{(i-1)})^* T_{22}^{(i-1)})^q (T_{22}^{(i-1)})^* G$, where q is some small nonnegative integer, typically less than three. The result is

$$((T_{22}^{(i-1)})^* (T_{22}^{(i-1)})^* G = \hat{V}_{22}^{(i)} R_{\text{trash}}.$$

This simple algorithm has been demonstrated to consistently provide high-quality subspace approximations to the space spanned by the leading b right singular vectors of $T_{22}^{(i-1)}$, which explains why `randUTV` reveals rank almost as well as the SVD. The importance of power iteration depends on the decay in the singular spectrum. When the singular values decay rapidly, power iteration is not necessary, and one may set $q = 0$. Conversely, when the singular values decay slowly, one or two steps of power iteration ($q \in \{1, 2\}$) greatly improves the alignment of the computed orthonormal basis with the space spanned by the dominant singular vectors we are trying to capture. See [Sec. 4.5][38] and [Sec. 11][40] for additional details on the role of power iteration, and Reference 26 for a more through discussion of `randUTV`.

Likewise the SVD factorization, when factorizing tall and skinny matrices ($m \times n$ matrices with $m \gg n$), it would be much more efficient to first compute a non-pivoted QR factorization and then the `randUTV` factorization of the resulting R factor, since in that case all the transformations applied from the right in the `randUTV` factorization could be applied to only the top n rows.

3.2 | Executing `randUTV` out-of-core: An algorithm-by-blocks

For matrices so large they do not fit in RAM, `randUTV` requires significant management of I/O tasks. If the orthogonal matrices U and V are required, then these must be stored out-of-core as well. To implement `randUTV` efficiently under these constraints, it is helpful to reorganize the algorithm as an *algorithm-by-blocks*. Like blocked algorithms, algorithms-by-blocks seek to cast most of the flops in a factorization in terms of the highly efficient Level 3 BLAS (Basic Linear Algebra Subprograms). Unlike blocked algorithms, the algorithms-by-blocks take maximum advantage of the full main memory by making all the data blocks being transferred of the same size, which, besides, makes easier to overlap communication and computation. All these advantages make an algorithm-by-blocks more efficient. In the following sections, we present the core technologies behind the design and implementation of `randUTV` as an algorithm-by-blocks.

3.2.1 | Algorithms-by-blocks: An overview

When working with matrices stored in RAM, *blocked* algorithms can improve performances by processing multiple columns (or rows) of the input matrix A in each iteration of its main loop. For instance, some classical factorizations drive several columns to upper triangular form in each iteration of the main loop. This design allows most of the operations to be cast in terms of the Level 3 BLAS (matrix-matrix operations), and more specifically in `xgemm` operations (matrix-matrix products). As vendor-provided and open-source multithreaded implementations of the Level 3 BLAS are highly efficient (with performances close to the peak speed), blocked algorithms usually offer high performances. Processing one column (or one row) at a time would require the employment of the slower matrix-vector operations (Level 2 BLAS) and much more communication. Thus, a blocked implementation of `randUTV` relying largely on standard calls to parallel BLAS was found to be faster than the highly optimized MKL CPQR implementation for a shared memory system, *despite* `randUTV` *having a much higher flop count than the CPQR algorithm.*[26]

On the other side, in blocked algorithms the amount of data being processed by every iteration varies extremely. Usually, as the factorization advances, every call to parallel BLAS must handle an increasing (or a decreasing) amount of data. For instance, to factorize an $n \times n$ matrix with block size b, the first iteration of right-looking algorithms usually requires the processing of the full matrix, whereas the last iteration requires just to process a very small amount of data (in some cases a $b \times b$ block). When the data is stored in an slow external device, this extremely high variation in the data being transferred harms performances since external devices work optimally only for certain given transfer sizes. Moreover, these high variation in the data being transferred and processed makes an optimal scheduling of I/O operations and computational operations much more difficult since the cost of the operations varies even much more (the I/O cost is usually $\mathcal{O}(n^2)$, whereas the computational cost is usually $\mathcal{O}(n^3)$). In addition, this high variation also makes a poor use of main memory either at the beginning or at the end of the factorizations.

We are therefore led to seek a technique other than blocking to obtain higher performances, although we will not abandon the strategy of casting most operations in terms of the Level 3 BLAS. The key lies in changing the method with which we aggregate multiple lower-level BLAS flops into a single Level 3 BLAS operation. Blocked algorithms do this by raising the granularity of the algorithm's main loop. The alternative approach, called algorithms-by-blocks, is to instead raise the granularity of the *data*. With this method, the algorithm may be designed as if only scalar elements of the input are dealt with at one time. Then, the algorithm is transformed into Level 3 BLAS by conceiving of each scalar as a supmatrix or block of size $b \times b$. Each scalar operation turns into a matrix-matrix operation, and operations in the algorithm will, at the finest level of detail, operate on usually a few (between one and four, but usually two or three) $b \times b$ blocks. Each operation on a few blocks is called a task. This arrangement removes the problems of blocked algorithms since every task will work with a similar amount of data, the memory can be employed as a cache to store blocks of memory, and an overlapping of computation and communication is much easier and more efficient. The performance benefits obtained by algorithm-by-blocks with respect to blocked algorithms for linear algebra problems on shared-memory architectures with data stored in RAM are usually significant[41] when more than a few cores are employed, because they remove the thread synchronization points that blocked algorithms insert after every call to the parallel BLAS. On the other side, when the data is stored in an external device, a runtime[18] to execute algorithms-by-blocks was described, but it was only applied to the following basic factorizations: Cholesky, LU with incremental pivoting, and unpivoted incremental QR. In our work we have built an algorithm-by-blocks for computing the much more complex `randUTV` factorization on large matrices stored in an external device by extending the mentioned approach and runtime. Despite the original `randUTV` factorization being much more difficult and complex than those basic factorizations, our approach to reveal the rank of matrices stored in an external device has obtained good performances when compared to high-performance algorithms revealing the rank of matrices stored in RAM, thus making the revealing of the rank of very large matrices feasible.

An algorithm-by-blocks for computing the `randUTV` requires that the QR factorization performed inside also works on $b \times b$ blocks. In order to design this internal QR factorization process such that each unit of work requires only $b \times b$ submatrices, the algoritm-by-blocks for computing the QR factorization must employ an algorithm based on updating an existing QR factorization. We shall refer to this algorithm as `QR_AB`. We consider only the part of `QR_AB` that makes the first column of blocks upper triangular, since that is all what is required for `randUTV_AB`.

Figure 1 shows this process for a 9×9 matrix with block size 3. In this figure, the continuous lines show the 3×3 blocks of the matrix involved in the current task, the '$\bullet$' symbol represents a non-modified element by the current task, the '$\star$' symbol represents a modified element by the current task, and the '$\cdot$' symbol represents an element nullified by the current task. The nullified elements are shown because, as usual in linear algebra factorizations, they store information about the Householder transformations that will be used later to apply these transformations. The first task, called *Compute_QR*, computes the QR factorization of the leading dense block A_{00}, thus nullifying all the elements below the main diagonal and modifying all the elements on or above the main diagonal. The second task, called *Apply_left_Qt_of_dense_QR*, applies the Householder transformations obtained in the previous task (and stored in A_{00}) to block A_{01}. The third task performs the same operation onto A_{02}. The fourth task annihilates block A_{10}, which is called *Compute_QR_of_td_QR* (where 'td' means triangular-dense since the upper block A_{00} is triangular and the lower block A_{10} is dense). The fifth task, called *Apply_left_Qt_of_td_QR*, applies the transformations of the previous task to blocks A_{01} and A_{11}. The sixth task performs the same operation onto A_{02} and A_{12}. Analogously, the seventh, eighth, and ninth tasks update the first and third row of blocks by performing the same work as the fourth, fifth, and sixth tasks. By taking advantage of the zeros present in the factorizations for each iteration, a well-implemented `QR_AB` costs essentially no more flops than the traditional blocked unpivoted QR. The algorithm is described in greater detail in References 42 and 43.

3.2.2 | Algorithms-by-blocks for `randUTV`

An algorithm-by-blocks for `randUTV`, which we will call `randUTV_AB`, performs mostly the same operations as the original, but they are rearranged as many more tasks working on usually square blocks (except maybe the right-most and bottom-most blocks). We will discuss in some detail how this plays out in the first step of the algorithm. First, choose a block size b. (When working on matrices stored on main memory, small block sizes such as $b = 128$ or 256 usually work well. In contrast, when working on very large matrices stored on an external device, much larger block sizes such as $b = 10{,}240$ must be employed.) For simplicity, assume b divides both m and n evenly. Recall that at the beginning of `randUTV`, T is initialized with $T = A$. Consider the following partitioning of the matrix T:

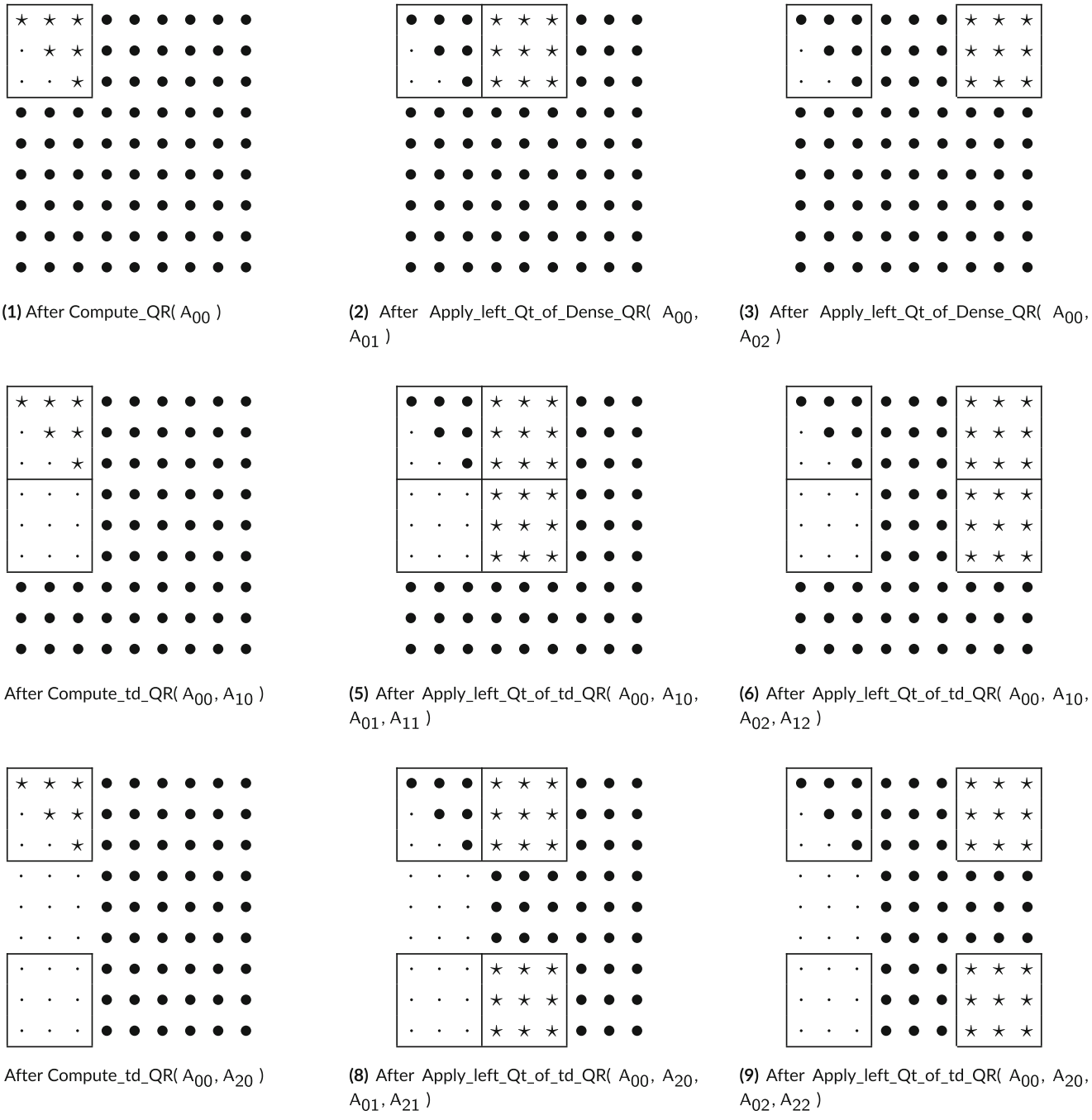

FIGURE 1 An illustration of the first tasks performed by an algorithm-by-blocks for computing the QR factorization. The '•' symbol represents a non-modified element by the current task, '⋆' represents a modified element by the current task, and '·' represents a nullified element by the current task (they are shown because they store information about the Householder transformations that will be later used to apply them). The continuous lines surround the blocks involved in the current task.

$$T \rightarrow \left(\begin{array}{c|c|c|c} T_{11} & T_{12} & \cdots & T_{1N} \\ \hline T_{21} & T_{22} & \cdots & T_{2N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline T_{M1} & T_{M2} & \cdots & T_{MN} \end{array}\right),$$

where each supmatrix or block T_{ij} is $b \times b$, $N = n/b$, and $M = m/b$. Note that the rest of matrices (G, Y, U, and V) must also be accordingly partitioned. The submatrices T_{ij} (and those of the rest of matrices) are treated as the fundamental unit of data in the algorithm,

so that each operation is expressed only in these terms. For the first step of the algorithm when no orthonormal matrices are built, for instance:

1. **Constructing** $V^{(0)}$: The first step, $Y^{(0)} = (T^*T)^q T^* G^{(0)}$, is broken into several tasks, each one of which computes the product of two blocks. In the simplified case where $q = 0$, we have $M \times N$ products of two blocks. The second step, the QR factorization of $Y^{(0)}$, uses the `QR_AB` algorithm previously described. Thus, the decomposition of each $Y_i^{(0)}$ is computed separately, and the resulting upper triangular factor $R^{(0)}$ (stored in $Y_1^{(0)}$) is updated after each step. Then, matrix T is updated with the previous transformations obtained in the factorization of $Y^{(0)}$, that is, $T \leftarrow TV^{(0)}$. See, for example, References 14, 18, and 42 for details on an approach to a full unpivoted QR factorization.
2. **Constructing** $U^{(0)}$: This step requires an unpivoted QR factorization of the first column block of T. A similar algorithm to the previous one, `QR_AB`, has been employed, the main difference being that the transformations are applied from the left int this case. Then, the rest of the matrix T must be updated with the previous transformations, that is, $T \leftarrow (U^{(0)})^* T$.
3. **Computing the SVD of** T_{11}: This step is the same one as in `randUTV` and `randUTV_AB`. In both cases, T_{11} is interacted with as a single unit. Then, matrix T must be properly updated with the transformations obtained in this step.

3.2.3 | The FLAME abstraction for implementing algorithm-by-blocks

A nontrivial obstacle to implementing an algorithm-by-blocks is the issue of programmability. The FLAME (Formal Linear Algebra Methods Environment) project[44,45] helps to solve this issue. FLAME is a framework for designing linear-algebra algorithms. In this approach the input matrix is viewed as a collection of submatrices, basing its loops on re-partitionings of the input data. The FLAME API[46] for the C programming language enables a user to code high-performance implementations of linear-algebra algorithms at a high level of abstraction. Besides, this methodology makes it a natural fit for use with an algorithm-by-blocks. Thus, the actual code for the implementation of `randUTV_AB` looks very similar to the written version of the algorithm given in Figure 2.

3.2.4 | Scheduling and dispatching the operations for an algorithm-by-blocks

The runtime described in Reference 18 allows to execute algorithms-by-blocks for single factorizations that work on data stored in an external device. We have employed and extended that runtime to compute the `randUTV` factorization. This algorithm is called the `randUTV_AB`. To understand how this OOC runtime works, consider the problem of factorizing a matrix of 2×2 blocks

$$A \leftarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array} \right),$$

where each block is of size $b \times b$. We will consider the case where the power iteration parameter $q = 0$ for simplicity. The execution of the program proceeds in two phases: the analysis stage and the execution stage.

In the first stage (analysis), instead of executing the code sequentially, the runtime builds a list or queue of tasks recording the operands associated with each task or operation. Figure 3 shows an example of the list built up by the runtime for `randUTV_AB` for the case $A \in \mathbb{R}^{n \times n}$, $b = n/2$, and $q = 0$. The S factors obtained in the QR factorization of the Y blocks are stored in blocks called B, whereas the S factors obtained in the factorization of the blocks of the current column block of A are stored in blocks called C.

In the second stage (execution), the runtime executes (or dispatches) all the tasks in the list. Several method can be employed to execute these tasks.

1. **Traditional method.** This is an straightforward implementation. For each task, all the input operands are read from the external device, then the task is executed with its operands stored in RAM, and finally all the output operands are written to the external device. The main advantage is its simplicity, but it obviously performs many I/O operations.

   Figure 4 illustrates the execution of the first six tasks of `randUTV_AB` for a matrix $A \in \mathbb{R}^{n \times n}$ with block size $b = n/2$.
2. **Method with cache.** Since the work to do has been decomposed into many "small" tasks, a fast method to accelerate performances is to store as many blocks as possible in main memory. By efficiently using a part of the main memory as a cache of the blocks stored in the external device, the number of I/O operations can be reduced. Since most blocks are of the same size, the management of the cache is very efficient. The only blocks with different sizes (smaller) are those that store the S factors of the unpivoted QR factorizations, but all blocks will be treated in the same way to simplify the programming and accelerate the execution. When the cache is full with blocks and a new block must be loaded, the Least-Recently

Algorithm: $[U, T, V] = \text{randUTV\_AB}(A, q, n_b)$

$V = \text{eye}(n(A), n(A)), \quad U = \text{eye}(m(A), m(A))$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), V \rightarrow \left(\begin{array}{c|c} V_L & V_R \end{array}\right), U \rightarrow \left(\begin{array}{c|c} U_L & U_R \end{array}\right)$

where A_{TL} is 0×0, V_L has 0 columns, U_L has 0 columns

while $m(A_{TL}) < m(A)$ **do**

Determine block size $b = \min(n_b, n(A_{BR}))$

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} V_L & V_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} V_0 & V_1 & V_2 \end{array}\right), \left(\begin{array}{c|c} U_L & U_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_0 & U_1 & U_2 \end{array}\right)$$

where A_{11} is $b \times b$, V_1 has b rows, U_1 has b rows

$$\begin{aligned}
&\% \text{ Right transform V} \\
G &= \text{generate_iid_stdnorm_matrix}(m(A) - m(A_{00}), n_b) \\
Y &= \left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)^* \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\right)^q \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)^* G \\
[Y, T_V] &= \text{unpivoted_QR}(Y) \\
\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) &= \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) - \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) W_V T_V W_V^* \\
\left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) &= \left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) - \left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) W_V T_V W_V^* \\
&\% \text{ Left transform U} \\
[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), T_U] &= \text{unpivoted_QR}\left(\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)\right) \\
\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) &= \left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) - W_U^* T_U^* W_U \left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) \\
\left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) &= \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) - \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) W_U T_U W_U^* \\
&\% \text{ Small SVD} \\
[A_{11}, U_{SVD}, V_{SVD}] &= \text{SVD}(A_{11}) \\
A_{01} &= A_{01} V_{SVD} \\
A_{12} &= U_{SVD}^* A_{12} \\
V_1 &= V_1 V_{SVD} \\
U_1 &= U_1 U_{SVD}
\end{aligned}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c|c} V_L & V_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} V_0 & V_1 & V_2 \end{array}\right), \left(\begin{array}{c|c} U_L & U_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_0 & U_1 & U_2 \end{array}\right)$$

endwhile

FIGURE 2 The `randUTV` algorithm adapted for algorithms-by-blocks written with the FLAME methodology/notation. In this algorithm, W_V and W_U are the unit lower trapezoidal matrices stored below the diagonal of Y and $\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$, respectively.

Operation	Operands	
	In	Out
Generate_normal_random		G_0
Generate_normal_random		G_1
Gemm_tn_oz: $C = A^*B$	A_{00}, G_0	Y_0
Gemm_tn_oz: $C = A^*B$	A_{01}, G_0	Y_1
Gemm_tn_oo: $C = C + A^*B$	A_{10}, G_1, Y_0	Y_0
Gemm_tn_oo: $C = C + A^*B$	A_{11}, G_1, Y_1	Y_1
Comp_dense_QR	Y_0	Y_0, B_0
Comp_td_QR	Y_0, Y_1	Y_0, Y_1, B_1
Apply_right_Q_of_dense_QR	Y_0, B_0, A_{00}	A_{00}
Apply_right_Q_of_dense_QR	Y_0, B_0, A_{10}	A_{10}
Apply_right_Q_td_QR	Y_1, B_1, A_{00}, A_{01}	A_{00}, A_{01}
Apply_right_Q_td_QR	Y_1, B_1, A_{10}, A_{11}	A_{10}, A_{11}
Comp_dense_QR	A_{00}	A_{00}, C_0
Comp_td_QR	A_{00}, A_{10}	A_{00}, A_{10}, C_1
Apply_left_Qt_of_dense_QR	A_{00}, C_0, A_{01}	A_{01}
Apply_left_Qt_of_td_QR	A_{10}, C_1, A_{01}, A_{11}	A_{01}, A_{11}
Keep_upper_triang	A_{00}	A_{00}
Set_to_zero		A_{10}
Svd_of_block	A_{00}	A_{00}, P_0, Q_0
Gemm_abta: $A = B^*A$	P_0, A_{01}	A_{01}
Svd_of_block	A_{11}	A_{11}, P_0, Q_0
Gemm_aabt: $A = AB^*$	A_{01}, Q_0	A_{01}

FIGURE 3 A list of the tasks or operations queued up by the runtime during the analyzer stage in the simplified case that the block size is $b = n/2$. The "In" column specifies pieces of required input data. The "Out" column specifies required pieces of data that will be altered upon completion of the operation.

Used (LRU) block is selected. If the block has been modified while staying in RAM, it must be written to the external device. Otherwise, it can plainly be discarded and overwritten with the new block.

Figure 5 illustrates the execution of the first six tasks of `randUTV_AB` for a matrix $A \in \mathbb{R}^{n \times n}$ with block size $b = n/2$ and a cache with 7 blocks. To reduce the length of the example but still show the benefits of this approach, the matrix is small and the cache of blocks is rather large. Note that when the cache is full, a block must be replaced. For instance, to load A_{11} block, the least-recently used block (A_{00}) is selected. As it has not been modified, it need not be written to disk.

3. **Method with cache + overlapping of computation and communication.** Though the use of a cache allows the reuse of blocks currently stored in main memory, when a block is not already in main memory, it must be read from the slow external device. An additional problem happens if the cache is full and the block selected to be replaced has been modified, because it must be written to the external device before loading the new block. During all this time, the cores cannot compute and therefore overall performances drop. One way to avoid this is to use one core to perform all the I/O operations (communications) while the other cores compute. The disadvantage of this method is that the computations will be slower than the two previous methods since they employ one fewer core. However, the communications can be made transparent (or at least be reduced) since I/O operations are performed at the same time as the computation.

   The I/O thread and the rest of the threads (computational threads) are completely decoupled. The I/O thread works by bringing data into the cache of blocks in main memory in advance, and sometimes it must remove data that is currently in cache to make room for newer data. This decoupling makes the programming more difficult, but it accelerates the execution since the speed of cores and the speed of the external devices can be very different between computers (or even between different external devices in the same computer).

   Figure 6 illustrates the execution with overlapping of computation and communication of the first six tasks of `randUTV_AB` for a matrix $A \in \mathbb{R}^{n \times n}$ with block size $b = n/2$, and a cache with seven blocks. In this case the execution of tasks is not clearly marked since the execution of different tasks can overlap. For instance reading or writing of operands of one task can be executed at the same time as the computation of another task. For ease of notation, the figure shows that each I/O operation takes exactly the same as a computational task. In practice, the I/O operations and the computational tasks are decoupled.

Task
$G_0 \leftarrow$ Generate_normal_random()
Write_block(G_0)
$G_1 \leftarrow$ Generate_normal_random()
Write_block(G_1)
$A_{00} \leftarrow$ Read_block()
$G_0 \leftarrow$ Read_block()
$Y_0 \leftarrow$ Gemm_tn_oz(A_{00}, G_0)
Write_block(Y_0)
$A_{01} \leftarrow$ Read_block()
$G_0 \leftarrow$ Read_block()
$Y_1 \leftarrow$ Gemm_tn_oz(A_{01}, G_0)
Write_block(Y_1)
$A_{10} \leftarrow$ Read_block()
$G_1 \leftarrow$ Read_block()
$Y_0 \leftarrow$ Read_block()
$Y_0 \leftarrow$ Gemm_tn_oo(Y_0, A_{10}, G_1)
Write_block(Y_0)
$A_{11} \leftarrow$ Read_block()
$G_1 \leftarrow$ Read_block()
$Y_1 \leftarrow$ Read_block()
$Y_1 \leftarrow$ Gemm_tn_oo(Y_1, A_{11}, G_1)
Write_block(Y_1)
⋮

FIGURE 4 Execution of the first tasks by using the traditional approach. The execution of the six first tasks requires 16 I/O operations (6 writes and 10 reads). Double horizontal lines mark the boundaries between consecutive tasks.

Task	Cache after Task						
$G_0 \leftarrow$ Generate_normal_random()	G_0	–	–	–	–	–	–
$G_1 \leftarrow$ Generate_normal_random()	G_0	G_1	–	–	–	–	–
$A_{00} \leftarrow$ Read_block()	G_0	G_1	A_{00}	–	–	–	–
$Y_0 \leftarrow$ Gemm_tn_oz(A_{00}, G_0)	G_0	G_1	A_{00}	Y_0	–	–	–
$A_{01} \leftarrow$ Read_block()	G_0	G_1	A_{00}	Y_0	A_{01}	–	–
$Y_1 \leftarrow$ Gemm_tn_oz(A_{01}, G_0)	G_0	G_1	A_{00}	Y_0	A_{01}	Y_1	–
$A_{10} \leftarrow$ Read_block()	G_0	G_1	A_{00}	Y_0	A_{01}	Y_1	A_{10}
$Y_0 \leftarrow$ Gemm_tn_oo(Y_0, A_{10}, G_1)	G_0	G_1	A_{00}	Y_0	A_{01}	Y_1	A_{10}
$A_{11} \leftarrow$ Read_block()	G_0	G_1	A_{11}	Y_0	A_{01}	Y_1	A_{10}
$Y_1 \leftarrow$ Gemm_tn_oo(Y_1, A_{11}, G_1)	G_0	G_1	A_{11}	Y_0	A_{01}	Y_1	A_{10}
⋮							

FIGURE 5 Execution of the first tasks by using a cache with 7 blocks. The execution of the six first tasks requires only 4 I/O operations (all of them reads). Double horizontal lines mark the boundaries between consecutive tasks.

Computational Task	I/O Task	Cache after Task						
$G_0 \leftarrow$ Generate_normal_random()	$A_{00} \leftarrow$ Read_block()	G_0	A_{00}	–	–	–	–	–
$G_1 \leftarrow$ Generate_normal_random()	$A_{01} \leftarrow$ Read_block()	G_0	A_{00}	G_1	A_{01}	–	–	–
$Y_0 \leftarrow$ Gemm_tn_oz(A_{00}, G_0)	$A_{10} \leftarrow$ Read_block()	G_0	A_{00}	G_1	A_{01}	Y_0	A_{10}	–
$Y_1 \leftarrow$ Gemm_tn_oz(A_{01}, G_0)	$A_{11} \leftarrow$ Read_block()	G_0	G_1	A_{10}	Y_0	A_{01}	A_{10}	Y_1
$Y_0 \leftarrow$ Gemm_tn_oo(Y_0, A_{10}, G_1)		G_0	G_1	A_{10}	Y_0	A_{01}	A_{10}	Y_1
$Y_1 \leftarrow$ Gemm_tn_oo(Y_1, A_{11}, G_1)		G_0	G_1	A_{10}	Y_0	A_{01}	A_{10}	Y_1
⋮								

FIGURE 6 Execution of the first tasks by using a cache with seven blocks and overlapping of computation and communication. The execution of the six first tasks requires only four I/O operations (all of them reads), and all of them are performed at the same time as the computation. No double horizontal lines are employed to mark the boundaries between consecutive tasks since I/O operations and computations of different tasks can be executed simultaneously.

4 | NUMERICAL RESULTS

In this section, we present the experiments demonstrating the accuracy, scalability, and computational costs of implementations of `HQRRP` and `randUTV_AB` for matrices stored out-of-core. In Subsection 4.2, we compare several implementations of `HQRRP` with different strategies for handling the I/O. In Subsection 4.3, we examine computational cost of an implementation of `randUTV`.

4.1 | Accuracy

The codes for computing the `randUTV_AB` factorization on matrices stored out-of-core were thoroughly tested in the platforms described below to analyze their accuracy. The following residuals of the `randUTV_AB` factorizations of many matrices stored out-of-core with different matrix dimensions and block sizes were computed: $\|A - UTV^*\|_F$, $\|I - U^*U\|_F$, $\|I - V^*V\|_F$, and $\|\text{sv}(A) - \text{sv}(T)\|_F$, where sv($X$) is a vector containing the sorted singular values of matrix X. All the above residuals were similar to those obtained by other factorizations such as the SVD, QR, and CPQR. Moreover, since the out-of-core randUTV factorization works in the same way as the in-core tiled randUTV factorization (also called randUTV algorithm-by-blocks), many cases were analyzed in detail to check that the results of our out-of-core codes matched the in-core codes. In all analyzed codes, the results were identical or very close to machine precision.

4.2 | Partial CPQR factorization with `HQRRP`

The machine used for these experiments had four DIMM memory chips with 16 GiB of DDR4 memory each. The CPU was an Intel®Core ™i7-6700K (4.00 GHz) with four cores. Experiments were run on two different hard drives. One was a Toshiba P300 HDD with 3 TB of memory and 7200 RPM; the other was a Samsung V-NAND SSD 950 Pro with 512 GiB of memory. The code was compiled with `gcc` (version 5.4.0) and linked with the Intel®MKL library (Version 2017.0.3).

The computational cost of three different implementations of `HQRRP` for matrices stored out-of-core were assessed.

1. **Algorithm 1 – In place:** This implementation of the `HQRRP` did not carry out physical permutations on the columns of A but instead applied the permutation information during each I/O task. In other words, the routine returns RP^* rather than R. As a result, no more data than necessary is transferred to and from the hard drive, but many reads occur from noncontiguous locations in the drive.
2. **Algorithm 2 – Physical pivoting:** Permutations *are* physically performed during the computation, so that upon completion R is stored in the hard drive. This implementation reads and writes more data than the "in place" version, but the data transfer mostly occurs in contiguous portions of memory.
3. **Algorithm 3 – Left-looking:** An implementation of a left-looking version of `HQRRP`, outlined in Section 2.3. This version requires only $\mathcal{O}(n^2)$ write operations, but the total number of operations (including reading, writing, and flops) is asymptotically higher than the "in place" and "physical pivoting" implementations.

Figure 7 shows the times scaled by the square of the matrix dimension of several factorizations. Two key observations can be made of the results depicted in this figure. The first observation is that the number of writes required by the implementation has a dramatic effect on its performance,

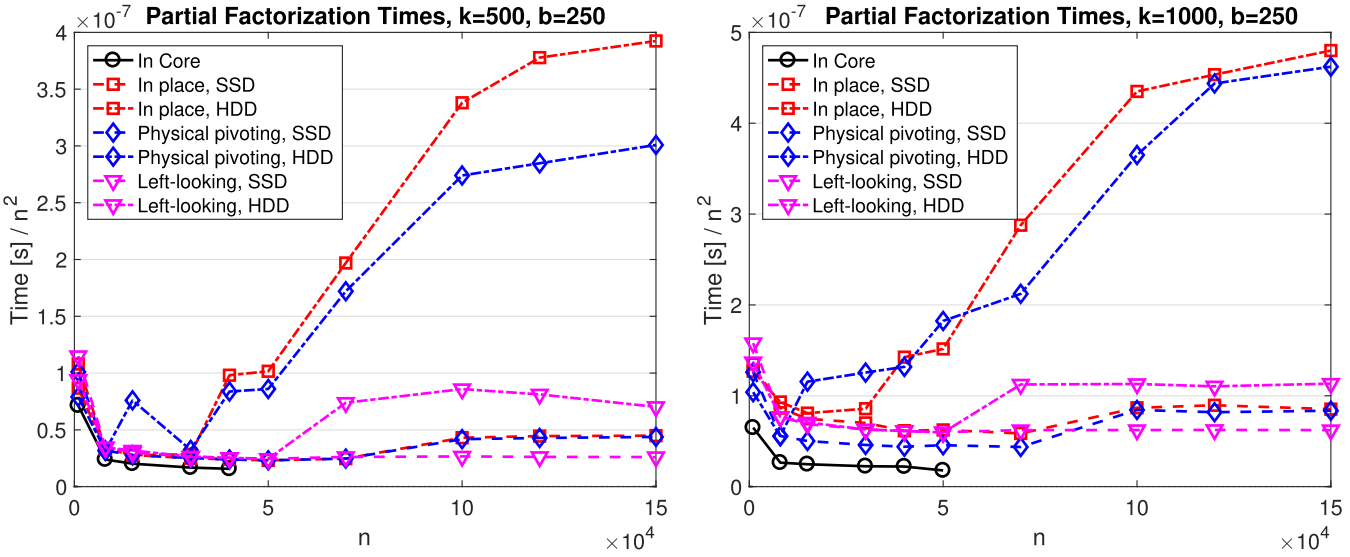

FIGURE 7 A comparison of the computational cost for three different algorithms for computing a partial CPQR when the matrix may be too large to fit in RAM. In the left figure, 1000 columns of the input matrix were processed. In the right figure, 500 columns were processed. The block size b in each case was 250.

especially when the matrix is stored on a hard disk drive. Algorithms 2 and 3, while performing reasonably well on the SSD, did not even scale correctly on the HDD, or at the very least did not reach asymptotic behavior as early as the SSD experiments. Algorithm 3, which requires $\mathcal{O}(n^2)$ write operations rather than $\mathcal{O}(n^3)$, outperforms the right-looking alternatives on both hard drives for large matrices. The second observation is that the best-performing algorithm takes roughly three times longer than the predicted performance for an in-core factorization of a matrix of the same size on the SSD. On the HDD, it takes about 5.4 times longer.

4.3 | Full factorization with `randUTV`

In this subsection, we assess the performances of our new implementations for computing the `randUTV` factorization when the data are stored in a disk drive, and compare it to the performances of highly optimized methods for computing the SVD, the column pivoted QR (CPQR) factorization, and the `randUTV` factorization when the data are stored in RAM.

To fairly compare the different implementations included in this study, the usual flop rate or the usual flop count cannot be employed since the computation of the SVD, the CPQR, and the `randUTV` factorizations require a very different number of flops (the dominant n^3-term in the asymptotic flop count is very different). Absolute computational times are not shown either as they vary greatly because of the huge range of matrix dimensions employed in the experiments. Therefore, scaled computational times (absolute computational times divided by n^3) are employed instead. The lower the scaled computational times, the better the performances are. Since all the implementations being assessed have asymptotic complexity $O(n^3)$ when applied to an $n \times n$ matrix, these graphs better reveal the computational speed. The scaled times are multiplied by a constant to make the figures in the vertical axis more readable. The value of this constant is shown in the vertical axis, and it is usually 10^{10}.

For the implementations of `randUTV`, both in-core and out-of-core, results are shown for 0, 1, and 2 iterations ($q = 0, q = 1$, and $q = 2$, respectively) in the power iteration process. Recall that the higher q, the higher the approximation to the singular values obtained in the main diagonal of matrix T.

The following out-of-core implementations were assessed in the experiments of this subsection:

1. OOC-RANDUTV-T: This is the traditional implementation for computing the `randUTV` factorization of a matrix stored in the disk by using an algorithm-by-blocks.
2. OOC-RANDUTV-V: This is the implementation for computing the `randUTV` factorization of a matrix stored in the disk by using an algorithm-by-blocks with a cache of blocks and overlapping of computation and I/O. Unlike the previous implementation, this one uses all the cores but one for computation and one core for I/O. Not all of the RAM in the computer is employed in the cache for matrix blocks, and a large amount of memory is left available for the operating system, since some initial experiments supported this approach.

3. OOC-QR-T: This is the traditional right-looking implementation for computing the QR factorization of a matrix stored in the disk by using an algorithm-by-blocks.
4. OOC-QR-V: This is the right-looking implementation for computing the QR factorization of a matrix stored in the disk by using an algorithm-by-blocks with a cache of blocks and overlapping of computation and I/O. One core is employed for I/O and the rest, for computations. Not all of the RAM in the computer is employed in the cache of matrix blocks.

Although the two methods for computing the QR factorization included in the experiments do not reveal the rank, they were included as a performance reference for the others.

In order to be included in this study, we asked some authors to send us their out-of-core codes to reveal the rank (e.g., SVD). Unfortunately, no codes were made available to us.

Our aim is to factorize very large matrices that do not usually fit in RAM unless a very expensive main memory is available. However, as a performance reference for our out-of-core implementations, we have included the following in-core implementations:

1. MKL SVD: The routine called `dgesvd` from MKL's LAPACK was used to compute the Singular Value Decomposition of matrices stored in RAM.
2. MKL CPQR: The routine called `dgeqp3` from MKL's LAPACK was used to compute the column-pivoting QR factorization of matrices stored in RAM.
3. `RANDUTV` PBLAS (`randUTV` with parallel BLAS): This is the traditional implementation for computing the `randUTV` factorization of matrices stored in RAM that relies on the parallel BLAS to take advantage of all the cores in the system. The parallel BLAS library from MKL was employed with these codes for the purpose of a fair comparison.
4. `RANDUTV` AB (`randUTV` with Algorithm-by-Blocks): This is the new implementation for computing the `randUTV` factorization by scheduling all the tasks to be computed in parallel, and then executing them with serial BLAS. The serial BLAS library from MKL was employed with these new codes for the purpose of a fair comparison.
5. MKL QR: The routine called `dgeqrf` from MKL's LAPACK was used to compute the QR factorization of matrices stored in RAM. Although this routine does not reveal the rank, it was included in some experiments as a performance reference for the others.

For most of the experiments, two plots are shown. The left plot shows the performances when no orthonormal matrices are computed. In this case, just the upper triangular factor R is computed for the CPQR and QR, just the upper triangular factor T is computed for the `randUTV`, and just the singular values are computed for the SVD. In contrast, the right plot shows the performances when all orthonormal matrices are explicitly formed in addition to the previously mentioned factors. In this case, matrix Q is computed for the CPQR and QR, and matrices U and V are computed for the `randUTV` and for the SVD. The right plot slightly favors the CPQR and QR since only one orthonormal matrix is formed.

4.3.1 | Experimental setup

The experiments reported in this section were performed on three computers. Next they are briefly described.

1. `ua`: This HP computer contained two Intel Xeon® CPU X5560 processors at 2.8 GHz, with 12 cores and 48 GiB of RAM in total. Its OS was GNU/Linux (Version 3.10.0-514.21.1.el7.x86_64). Intel's `icc` compiler (Version 12.0.0 20101006) was employed. LAPACK and BLAS routines were taken from the Intel(R) Math Kernel Library (MKL) Version 10.3.0 Product Build 20100927 for Intel(R) 64 architecture, since this library usually delivers much higher performances than LAPACK and BLAS from the Netlib repository. The Hard Disk Drive (HDD) employed in the experiments to store all the data in the out-of-core implementations was an HP MM0500EANCR (Firmware Revision HPG3). Though its capacity was about 500 GiB, only about 400 GiB were available for users, which was about 8.3 times as large as the main memory. According to the Linux operating system `hdparm` tool, the read speed of this drive was 91.13 MB/s.
2. `ut`: This Dell computer contained two Intel Xeon® Gold 6254 processors at 3.10 GHz, with 36 cores and 754 GiB of RAM in total. Its OS was GNU/Linux (Version 5.0.0-37-generic). Intel's `icc` compiler (Version 19.0.5.281 20190815) was employed. LAPACK and BLAS routines were taken from the Intel(R) Math Kernel Library (MKL) Version 2019.0.5 Product Build 20190808 for Intel(R) 64 architecture for the same reason as before. The disk drive (SDD) employed in the experiments to store all the data in the out-of-core implementations was a Toshiba KXG50PNV2T04 NVMe (Firmware Revision AFDA4105) with a capacity of 2 TiB. According to the Linux operating system `hdparm` tool, the read speed of this disk was 9744.21 MB/s on cached reads and the read speed of this disk was 2410.66 MB/s on buffered disk reads.
3. `ucm`: This Supermicro computer contained two Intel Xeon® processors E5-2695 v3 at 2.30 GHz, with 28 cores and 125 GiB of RAM in total. In this computer the so-called *Turbo Boost* mode of the CPU was turned off in our experiments. Its OS was GNU/Linux (Version 2.6.32-504.el6.x86_64).

Intel's `icc` compiler (Version 18.0.1 20171018) was employed. LAPACK and BLAS routines were taken from the Intel(R) Math Kernel Library (MKL) Version 2018.0.1 Product Build 20171007 for Intel(R) 64 architecture for the same reason as before. The disk drive (SDD) employed in the experiments to store all the data in the out-of-core implementations was a Samsung SSD 850 EVO (Firmware Revision EMT02B6Q) with a capacity of 1 TB. According to the Linux operating system `hdparm` tool, the read speed of this disk was 10,144.81 MB/s on cached reads and the read speed of this disk was 441.48 MB/s on buffered disk reads.

In our out-of-core implementations the block cache employed 16 GiB (of the 48 GiB), 256 GiB (of the 754 GiB), and 32 GiB (of the 128 GiB) in `ua`, `ut`, and `ucm`, respectively. The rest was left for the operating system kernel and buffers, and the application's code.

In contrast, unless explicitly stated otherwise, all the experiments employed all the cores in the computer.

Notice that the `ua` computer has a very slow spinning disk as well as a low computational power. Notice also that both `ut` and `ucm` have SSD disks, but their performances are different. The `ucm` computer has an SSD with a SATA interface, which is limited to 600 MiB/s, whereas the `ut` computer has an SSD with an M.2 interface, which does not have that limitation.

In all the experiments double-precision real matrices were processed. All the matrices used in these experiments were randomly generated since generation is fast.

4.3.2 | Effect of block sizes

Figure 8 shows the scaled computational times obtained by our implementations for computing both the QR factorization and the `randUTV` factorization versus several block sizes when matrices of dimension 81,920 × 81,920 are processed in `ua`. The aim of these two plots is to determine the optimal block sizes. As can be seen, performances of the QR factorization do not strongly depend on the block size. In contrast, for the `randUTV` factorization performances do depend on the block size, and block sizes between 7680 and 11,264 usually offer optimal results.

From now on, in `ua` the block size 10,240 will be employed for both the out-of-core QR and the out-of-core `randUTV` factorizations since it returns near-optimal results in most cases. As `ut` and `ucm` have larger central memories than `ua`, larger matrices will be assessed in those two computers. Since in linear algebra the larger the matrix sizes being tested, the larger the optimal block sizes usually are, in `ut` and `ucm` the block size 20,480 will be employed.

4.3.3 | Comparison of out-of-core variants

The plots in all the next figures include a vertical dashed gray line showing the largest theoretical matrix size that can be stored in the RAM of the computer when the `randUTV` factorization is computed. For instance, in `ua` this size is about 80,000 when no orthonormal matrices are built, and therefore matrices U and V do not need to be stored. In contrast, this size is about 46,000 when orthonormal matrices are built, and therefore matrices U, and V must be stored. In practice, the actual threshold must be slightly smaller than those in the pictures since main memory must be also employed for the operating system kernel, operating system's disk cache and buffers, application's code, application's other data, etc.

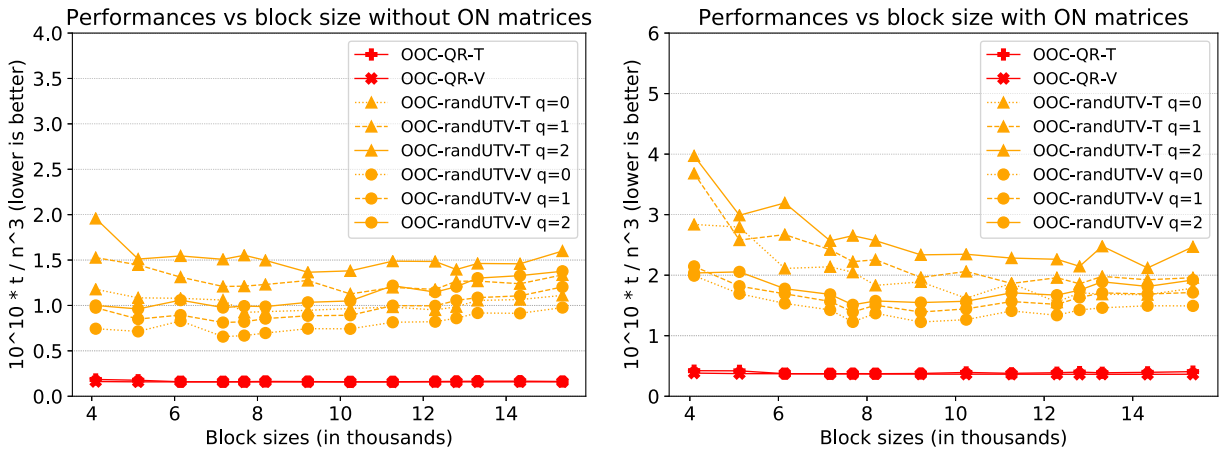

FIGURE 8 Performances of QR and `randUTV` implementations versus block sizes on matrices of dimension 81,920 × 81,920 in `ua`.

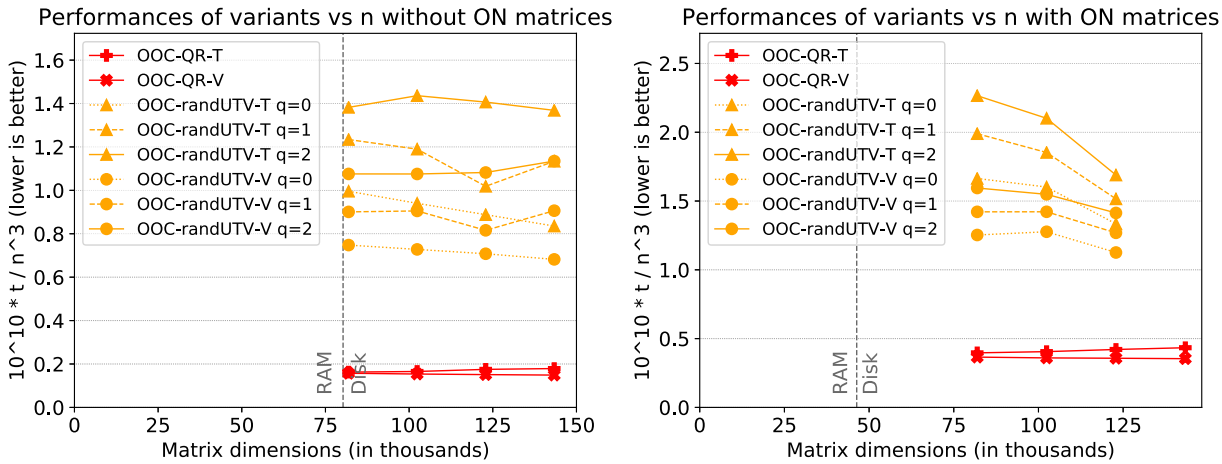

FIGURE 9 Performances versus matrix dimensions for the different implementations of the QR and `randUTV` factorizations in `ua`.

Figure 9 shows the performances for the different implementations of the QR and `randUTV` factorizations on several large matrix sizes. As can be seen, the overlapping of computation and I/O for the QR factorization increases performances slightly: The speedup is about between 3% and 20% when no orthonormal matrices are built, and it is about between 8% and 22% when orthonormal matrices are built. In contrast, the overlapping of computation and I/O for the `randUTV` clearly improve performances: The speedup is about between 25% and 37% when no orthonormal matrices are built, and it is about between 18% and 42% when orthonormal matrices are built.

To check the benefits of the overlapping of computation and I/O in the other platforms, Table 1 reports the total and decomposed times of the computation of the `randUTV` factorization of large matrices that do not fit in main memory in both `ut` and `ucm` when no orthornormal matrices are built and $q = 0$ (the power iteration process). To build this table, each operation in the factorization (both computational and I/O) was recorded. As can be seen, in `ut` the performances are very encouraging because the I/O time is much smaller than the computational time (about 18%), thus making the application computation-bound. However, in `ucm` the results are different since the disk of `ut` is 5.4 times as fast as the disk of `ucm`. The slower disk of `ucm` makes that the overall computational time and the overall I/O time are similar, the latter being slightly larger. On the other side, in `ut` the overlapping of computation and I/O is perfect since the real time is only 0.12% larger than the computational time. In `ucm` the overlapping of computation and I/O is almost perfect since the real time is only 27% larger than the computational time, and 13% larger than the I/O time (in this case larger than the computational time).

Figure 10 shows a real example of the scheduling of the different tasks during some time of the `randUTV` factorizations of a 348,160 × 348,160 matrix in `ut` when no orthonormal matrices are built and $q = 0$. The left plot shows the scheduling during one hour of the experiment; the right plot is a zoom of the first ten minutes of the left plot. The top part of the two plots shows the I/O tasks performed by the application: A red rectangle is a write operation, whereas a green rectangle is a read operation. The bottom part of the two plots shows the computation performed by the application. The names of the tasks are only shown when there is enough room. As can be seen, the overlapping of computation and I/O is almost perfect: The computer performs I/O operations and computation at the same time. Another interesting remark obtained from this plot is the fact that the I/O operations are usually smaller than the computational operations despite the high processing power of the computer (36 cores), which

TABLE 1 Decomposed times (in seconds) of the `randUTV` factorization of large matrices of dimension $n \times n$ on the `ut` and `ucm` platforms.

	`ut` $n = 348{,}160$	`ucm` $n = 143{,}360$
Computational time	229,627.6	34,165.5
I/O time	41,312.9	38,253.5
Computational time + I/O time	270,940.5	72,419.0
Real time	229,910.0	43,326.0
Ratio Real time / Computational time	1.001	1.268
Ratio Computational time / I/O time	5.558	0.893

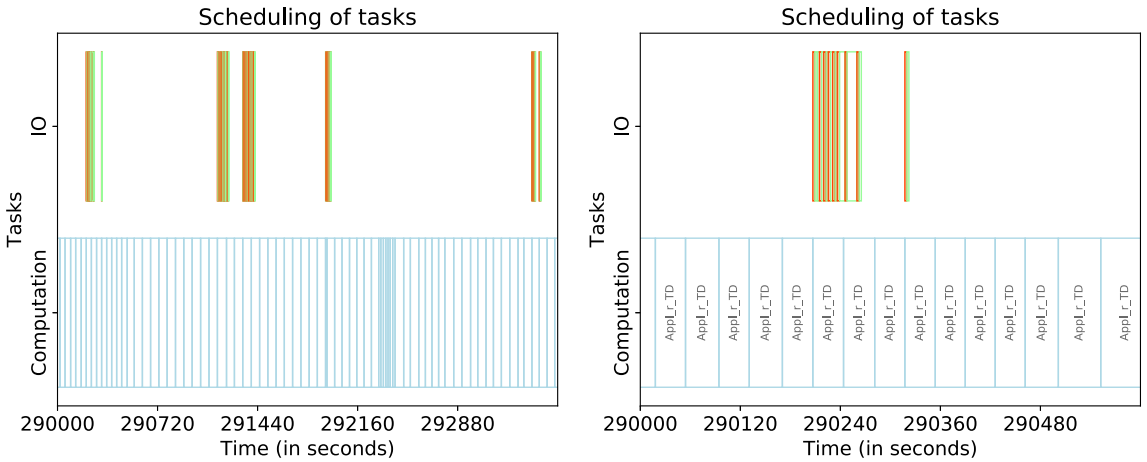

FIGURE 10 Example of the scheduling of the tasks in the execution of the `randUTV` factorization of a 348,160 × 348,160 matrix in `ut` in two different periods. The left plot shows the scheduling during 1 h; the right plot is a zoom of the first 10 min of the left plot.

makes the application still computation-bound in this case (`ut`). This means that higher computational power could further reduce the processing time of the factorizations.

4.3.4 | Performances of out-of-core codes versus in-core codes

Figure 11 shows the performances of the best out-of-core implementations as a function of the matrix dimensions. This plot also shows performances of in-core factorizations so that the speed of both types of implementations can be compared. Notice that when no orthonormal matrices are built (left plots) the in-core MKL SVD is much faster on `ut` and `ucm` than on `ua`. In those cases, the MKL SVD is even much faster than the MKL CPQR despite having a much higher computational cost. These high performances are caused by the employment of a new implementation for computing the SVD in MKL that replaces the traditional implementation for parallel architectures on matrices with dimensions larger than 4000. We think that some of the techniques employed by Intel for this new implementation of the SVD in MKL could also be applied to our randUTV to make it faster when no orthornormal matrices are built.

The top row of plots of Figure 11 shows the results obtained in `ua`. In these results the best out-of-core QR factorization is about 28% slower than the in-core QR factorization when no orthonormal matrices are built, and about 29% slower than the in-core QR factorization when orthonormal matrices are built. On the other side, the out-of-core `randUTV` is about 51% slower than the in-core `randUTV` factorization when no orthonormal matrices are built, and between 41% ($q = 0$) and 33% ($q = 2$) slower than the in-core `randUTV` factorization when orthonormal matrices are built. In all these cases the comparison has been performed considering the best performance of the out-of-core implementations against the best performance of the in-core factorizations, which is usually obtained on very large matrices (but smaller than those factorized by the out-of-core implementations).

The center row of plots of Figure 11 shows the results obtained in `ut`. In these results the out-of-core QR factorization is about 2.58 times as slow as the in-core QR factorization when no orthonormal matrices are built, and about 2.10 times as slow as the in-core QR factorization when orthonormal matrices are built. On the other side, the out-of-core `randUTV` is between 1.89 ($q = 0$) and 1.48 ($q = 2$) times as slow as the in-core `randUTV` factorization when no orthonormal matrices are built, and between 2.07 ($q = 0$) and 1.80 ($q = 2$) times as slow as the in-core `randUTV` factorization when orthonormal matrices are built.

The bottom row of plots of Figure 11 shows the results obtained in `ucm`. In these results the out-of-core QR factorization is about 1.76 times as slow as the in-core QR factorization when no orthonormal matrices are built, and about 2.00 times as slow as the in-core QR factorization when orthonormal matrices are built. On the other side, the out-of-core `randUTV` is between 2.15 ($q = 0$) and 2.36 ($q = 2$) times as slow as the in-core `randUTV` factorization when no orthonormal matrices are built, and between 2.10 ($q = 0$) and 2.03 ($q = 2$) times as slow as the in-core `randUTV` factorization when orthonormal matrices are built.

Therefore, as can be seen, in the worst cases the speed of revealing the rank of matrices stored in the slow external devices is only about two times as slow as the speed of revealing the rank of matrices already stored in the much faster main memory, which shows the good performances attained by the new implementations presented here.

In conclusion, our out-of-core implementations of the `randUTV` factorization are able to efficiently process very large data that do not fit in RAM and must be stored in the disk drive. Despite the slow speed of the hard drive with respect to the main memory, these methods are able to process matrices with only a minor loss of performances.

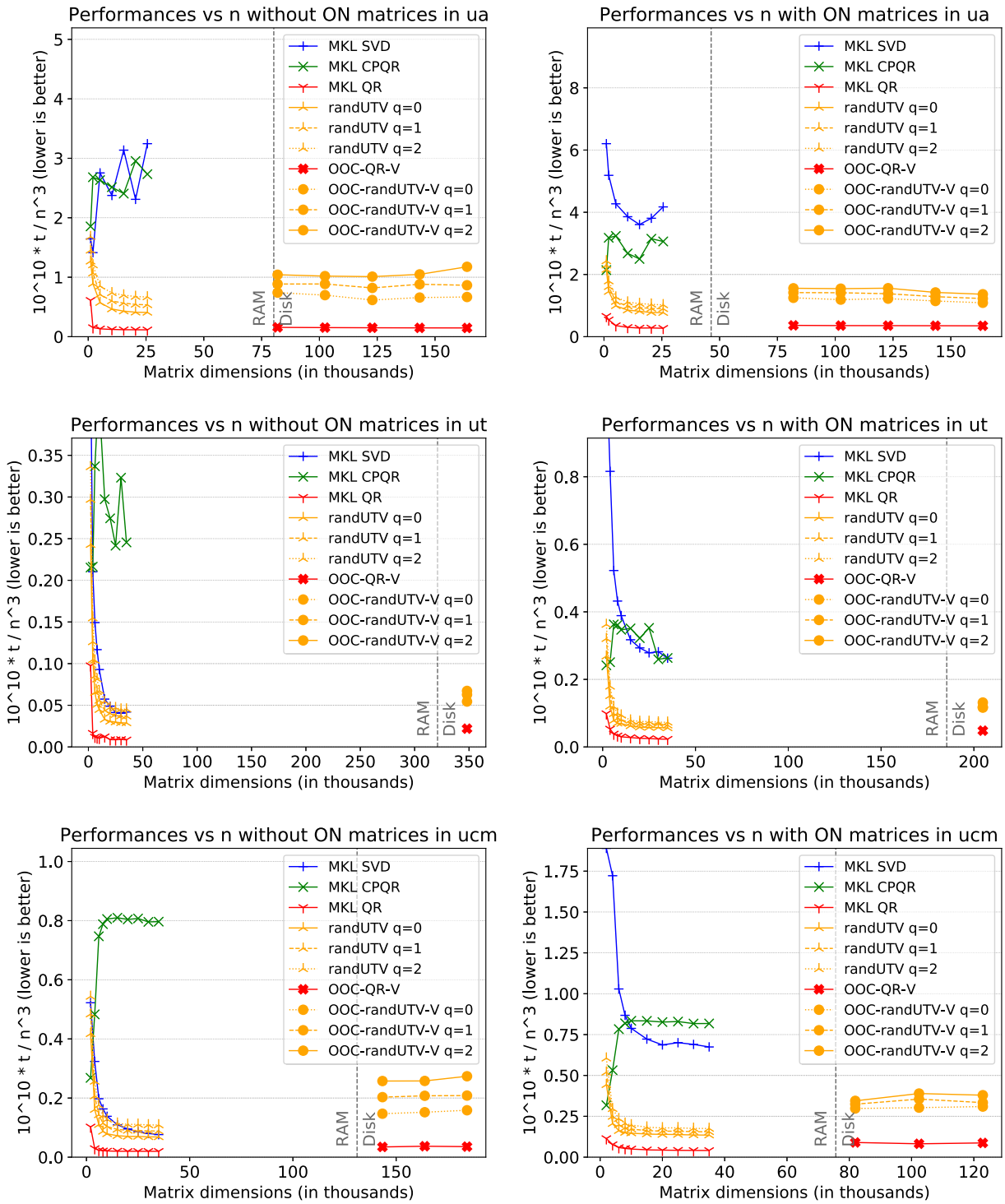

FIGURE 11 Performances versus matrix dimensions for the best implementations of the QR and `randUTV` factorizations. In-core implementations are included as a reference. Recall that the QR factorization does not reveal the rank, and it is included as a reference. The top row shows results in `ua`, the center row shows results in `ut`, and the bottom row shows results in `ucm`.

5 | CONCLUSIONS

This paper describes a set of algorithms for computing rank-revealing factorizations of matrices that are stored on external memory devices such as solid-state or spinning disk hard drives. Standard techniques for computing rank-revealing factorizations of matrices perform very poorly in this environment, as they inherently consist of a sequence of matrix-vector operations, which necessitate a large number of read and write operations to the disk. We use randomization to reorganize the computation to operate on large blocks of the matrix, thereby dramatically reducing the amount of communication required.

Numerical experiments demonstrate that the speed of the new randomized methods that operate on data stored on external devices is comparable to traditional methods that operate on data stored in main memory. This enables the processing of very large matrices in a cost efficient way. As the performance of solid state hard drives is rapidly improving in terms of both speed and capacity, the methods described are likely to gain even more of a competitive advantage in coming years.

ACKNOWLEDGMENTS

P. G. Martinsson acknowledges support from the Office of Naval Research (award N00014-18-1-2354), from the National Science Foundation (awards DMS-1952735 and DMS-2012606), from the Department of Energy ASCR (DE-SC0022251) and from Nvidia Corp. G. Quintana-Ortí was supported by the Spanish Ministry of Science and Innovation and the Research State Agency under grant PID2021-123627OB-C55 co-financed by FEDER funds (MCIN/AEI/FEDER/UE). The authors would like to thank Francisco D. Igual (Universidad Complutense de Madrid) and Javier Navarrete (Universitat d'Alacant) for granting access to the `ucm` and the `ua` servers, respectively.

CONFLICT OF INTEREST STATEMENT

The authors declare no potential conflict of interest.

DATA AVAILABILITY STATEMENT

All data employed in the experiments of this document are bidimensional matrices, which have been generated with software written by the authors. The software for generating the data is available from the corresponding author upon reasonable request. Data sharing is not applicable to this article since all the data (matrices) were created randomly by using open-source libraries.

ORCID

P. G. Martinsson https://orcid.org/0000-0002-1048-5270
G. Quintana-Ortí https://orcid.org/0000-0002-7912-7826

REFERENCES

1. Lawson CL, Hanson RJ. *Solving Least Squares Problems*. SIAM; 1995.
2. Golub GH, Van Loan CF. An analysis of the total least squares problem. *SIAM J Numer Anal*. 1980;17(6):883-893.
3. Van Huffel S, Vandewalle J. *The Total Least Squares Problem: Computational Aspects and Analysis*. SIAM; 1991.
4. Miller A. *Subset Selection in Regression*. Chapman and Hall/CRC; 2002.
5. Eckart C, Young G. The approximation of one matrix by another of lower rank. *Psychometrika*. 1936;1(3):211-218.
6. Mahoney MW. Randomized algorithms for matrices and data. *Found Trends® Mach Learn*. 2011;3(2):123-224.
7. Golub G, Kahan W. Calculating the singular values and pseudo-inverse of a matrix. *J Soc Ind Appl Math Ser B Numer Anal*. 1965;2(2):205-224. doi:10.1137/0702016
8. Golub GH, Van Loan CF. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. third ed. Johns Hopkins University Press; 1996.
9. Klema V, Laub A. The singular value decomposition: its computation and some applications. *IEEE Trans Automat Contr*. 1980;25(2):164-176.
10. Kahan W. Numerical linear algebra. *Can Math Bull*. 1966;9(5):757-801.
11. Businger P, Golub GH. Linear least squares solutions by householder transformations. *Numer Math*. 1965;7(3):269-276. doi:10.1007/BF01436084
12. Quintana-Ortí G, Sun X, Bischof CH. A BLAS-3 version of the QR factorization with column pivoting. *SIAM J Sci Comput*. 1998;19(5):1486-1494.
13. Drmač Z, Bujanović Z. On the failure of rank-revealing QR factorization software–A case study. *ACM Trans. Math. Softw.* 2008;35(2):1-28. doi:10.1145/1377612.1377616
14. Gunter BC, Van De Geijn RA. Parallel out-of-core computation and updating of the QR factorization. *ACM Trans Math Softw (TOMS)*. 2005;31(1):60-78.
15. Reiley WC, Van De Geijn RA. *POOCLAPACK: Parallel Out-of-Core Linear Algebra Package*. University of Texas at Austin; 1999.
16. Gunter BC, Reiley WC, Geijn RA. Parallel out-of-core cholesky and QR factorizations with POOCLAPACK. Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001, 1885–1894. 2001.
17. Toledo S, Gustavson FG. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. Proceedings of the Fourth Workshop on I/O in Parallel and Distributed Systems: Part of the Federated Computing Research Conference, 28–40, Citeseer. 1996.
18. Quintana-Ortí G, Igual FD, Marqués M, Quintana-Ortí ES, Geijn RA. A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures. *ACM Trans Math Softw (TOMS)*. 2012;38(4):25.
19. D'Azevedo E, Dongarra J. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. *Concurr: Pract Exp*. 2000;12(15):1481-1493.
20. Blackford LS, Choi J, Cleary A, et al. *ScaLAPACK Users' Guide*. SIAM; 1997.

21. Halko N, Martinsson P-G, Shkolnisky Y, Tygert M. An algorithm for the principal component analysis of large data sets. *SIAM J Sci Comput*. 2011;33(5):2580-2594.
22. Demchik V, Bacák M, Bordag S. Out-of-core singular value decomposition. *CoRR*. 2019; abs/1907.06470. http://arxiv.org/abs/1907.06470
23. Kabir K, Haidar A, Tomov S, Bouteiller A, Dongarra J. A framework for out of memory SVD algorithms. In: Kunkel J, Yokota R, Balaji P, Keyes D, eds. *High Performance Computing. ISC 2017*. Lecture Notes in Computer Science. Vol 10266. Springer; 2017.
24. Martinsson P-G, Quintana OG, Heavner N, Geijn R. Householder QR factorization with randomization for column pivoting (HQRRP). *SIAM J Sci Comput*. 2017;39(2):C96-C115.
25. Duersch JA, Gu M. Randomized QR with column pivoting. *SIAM J Sci Comput*. 2017;39(4):C263-C291. doi:10.1137/15M1044680
26. Martinsson PG, Quintana-Ortí G, Heavner N. randUTV: a blocked randomized algorithm for computing a rank-revealing UTV factorization. *ACM Trans. Math. Softw.* 2019;45(1):4:1-4:26. doi:10.1145/3242670
27. Stewart GW. UTV decompositions. *Pitman Res Notes Math Ser*. 1994;303:225.
28. Stewart GW. An updating algorithm for subspace tracking. *IEEE Trans Signal Process*. 1992;40(6):1535-1541.
29. Chandrasekaran S, Ipsen ICF. On rank-revealing factorisations. *SIAM J Matrix Anal Appl*. 1994;15(2):592-622.
30. Chan TF. Rank revealing QR factorizations. *Linear Algebra Appl*. 1987;88:67-82.
31. Chan TF, Hansen PC. Some applications of the rank revealing QR factorization. *SIAM J Sci Stat Comput*. 1992;13(3):727-741.
32. Heavner N. *Building Rank-Revealing Factorizations with Randomization*. PhD thesis. Applied Mathematics. University of Colorado at Boulder; 2019.
33. Gu M, Eisenstat SC. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM J. Sci. Comput*. 1996;17(4):848-869.
34. Civril A, Magdon-Ismail M. On selecting a maximum volume sub-matrix of a matrix and related problems. *Theor Comput Sci*. 2009;410(47-49):4801-4811.
35. Reiley WC. *Efficient Parallel Out-of-Core Implementation of the Cholesky Factorization*. University of Texas at Austin; 1999.
36. Kurzak J, Ltaief H, Dongarra J, Badia RM. Scheduling dense linear algebra operations on multicore processors. *Concurr Comput: Pract Exp*. 2010;22(1):15-44.
37. Rokhlin V, Szlam A, Tygert M. A randomized algorithm for principal component analysis. *SIAM J Matrix Anal Appl*. 2009;31(3):1100-1124.
38. Halko N, Martinsson P-G, Tropp JA. Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev*. 2011;53(2):217-288.
39. Martinsson P-G, Rokhlin V, Tygert M. *A Randomized Algorithm for the Approximation of Matrices*. Yale CS research report YALEU/DCS/RR-1361. Yale University, Computer Science Department; 2006.
40. Martinsson P-G, Tropp JA. Randomized numerical linear algebra: Foundations and algorithms. *Acta Numer*. 2020;29:403-572.
41. Marqués M, Quintana-Ortí G, Quintana-Ortí ES, Van De Geijn RA. Using desktop computers to solve large-scale dense linear algebra problems. *J Supercomput*. 2011;58(2):145-150.
42. Quintana-Ortí G, Quintana-Ortí ES, Van De Geijn RA, Van Zee FG, Chan E. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.* 2009;36(3):14:1-14:26. doi:10.1145/1527286.1527288
43. Chan E, Van Zee FG, Bientinesi P, Quintana-Orti ES, Quintana-Orti G, Geijn R. SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. Paper presented at: PPoPP'08, ACM, New York, NY, USA. 2008 123–132.
44. Gunnels JA, Gustavson FG, Henry GM, Geijn RA. FLAME: formal linear algebra methods environment. *ACM Trans. Math. Softw.* 2001;27(4):422-455. doi:10.1145/504210.504213
45. Igual FD, Chan E, Quintana-Ortí ES, Quintana-Ortí G, Van De Geijn RA, Van Zee FG. The FLAME approach: from dense linear algebra algorithms to high-performance multi-accelerator implementations. *J. Parallel Distrib. Comput*. 2012;72(9):1134-1143. doi:10.1016/j.jpdc.2011.10.014
46. Bientinesi P, Quintana-Ortí ES, van de Geijn RA. Representing linear algebra algorithms in code: the FLAME application program interfaces. *ACM Trans. Math. Softw.* 2005;31(1):27-59. doi:10.1145/1055531.1055533

How to cite this article: Heavner N, Martinsson PG, Quintana-Ortí G. Computing rank-revealing factorizations of matrices stored out-of-core. *Concurrency Computat Pract Exper*. 2023;e7726. doi: 10.1002/cpe.7726