Algorithm 1022: Efficient Algorithms for Computing a Rank-Revealing UTV Factorization on Parallel Computing Architectures

N. HEAVNER, University of Colorado at Boulder, USA
F. D. IGUAL, Universidad Complutense de Madrid, Spain
G. QUINTANA-ORTÍ, Universitat Jaume I de Castellón, Spain
P. G. MARTINSSON, University of Texas at Austin, USA

Randomized singular value decomposition (RSVD) is by now a well-established technique for efficiently computing an approximate singular value decomposition of a matrix. Building on the ideas that underpin RSVD, the recently proposed algorithm "randUTV" computes a *full* factorization of a given matrix that provides low-rank approximations with near-optimal error. Because the bulk of randUTV is cast in terms of communication-efficient operations such as matrix-matrix multiplication and unpivoted QR factorizations, it is faster than competing rank-revealing factorization methods such as column-pivoted QR in most high-performance computational settings. In this article, optimized randUTV implementations are presented for both shared-memory and distributed-memory computing environments. For shared memory, randUTV is redesigned in terms of an *algorithm-by-blocks* that, together with a runtime task scheduler, eliminates bottlenecks from data synchronization points to achieve acceleration over the standard *blocked algorithm* based on a purely fork-join approach. The distributed-memory implementation is based on the ScaLAPACK library. The performance of our new codes compares favorably with competing factorizations available on both shared-memory and distributed-memory architectures.

CCS Concepts: • **Mathematics of computing → Computations on matrices;**

N. Heavner and P. G. Martinsson collaborated in the ideas and the writing of the manuscript.
F. D. Igual and G. Quintana-Ortí collaborated in the ideas, the design of the algorithms, the implementation and assessing of the codes, the discussion of the experimental results, and the writing of the manuscript.
F. D. Igual was supported by the EU (FEDER) and Spanish MINECO (GA No. RTI2018-093684-B-I00), and by Spanish CM (GA No. S2018/TCS-4423). This work has been supported by the Madrid Government (Comunidad de Madrid- Spain) under the Multiannual Agreement with Complutense University in the line Program to Stimulate Research for Young Doctors in the context of the V PRICIT (Regional Programme of Research and Technological Innovation) under grant no. PR65/19-22445. P. G. Martinsson acknowledges support from the Office of Naval Research (grant no. N00014-18-1-2354), from the National Science Foundation (grant nos. DMS-1620472 and DMS-1952735) and from the Texas Advanced Computing Center. G. Quintana-Ortí was supported by the Spanish Ministry of Science, Innovation and Universities under grant no. RTI2018-098156-B-C54 co-financed with FEDER funds.
Authors' addresses: N. Heavner, Department of Applied Mathematics, University of Colorado Boulder, 11 Engineering Dr, Boulder, CO 80309, United States; email: Nathan.Heavner@colorado.edu; F. D. Igual, Deptartamento de Arquitectura de Computadores y Automática, Universidad Complutense de Madrid, Av. Complutense s/n, 28040 Madrid, Spain, 28040; email: figual@ucm.es; G. Quintana-Ortí, Departamento de Ingeniería y Ciencia de los Computadores, Universitat Jaume I de Castellón, Av. Sos Baynat s/n, 12071 Castellón, Spain; email: gquintan@uji.es; P. G. Martinsson, Department of Mathematics, The University of Texas at Austin, 2515 Speedway, PMA 8.100, Austin, TX 78712; email: pgm@oden.utexas.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
0098-3500/2022/05-ART21 $15.00
https://doi.org/10.1145/3507466

Additional Key Words and Phrases: Numerical linear algebra, rank-revealing matrix factorization, singular value decomposition, high performance, randomized methods, block algorithm, algorithm-by-blocks, randomized SVD

ACM Reference format:
N. Heavner, F. D. Igual, G. Quintana-Ortí, and P. G. Martinsson. 2022. Algorithm 1022: Efficient Algorithms for Computing a Rank-Revealing UTV Factorization on Parallel Computing Architectures. *ACM Trans. Math. Softw.* 48, 2, Article 21 (May 2022), 42 pages.
https://doi.org/10.1145/3507466

1 INTRODUCTION

Computational linear algebra faces significant challenges as high-performance computing moves further away from the serial into the parallel. Classical algorithms were designed to minimize the number of floating point operations and do not always lead to optimal performance on modern communication-bound architectures. The obstacle is particularly apparent in the area of rank-revealing matrix factorizations. Traditional techniques based on column-pivoted QR factorizations or Krylov methods tend to make efficient parallelization challenging, as they are most naturally viewed as a sequence of matrix-vector operations.

In this article, we describe techniques for efficient implementations of a randomized algorithm for computing a so-called rank-revealing UTV decomposition [24] (randUTV from now on) on shared- and distributed-memory architectures. Given an input matrix A of size $m \times n$, the goal of randUTV is to compute a factorization

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

where the middle factor T is upper triangular (or upper trapezoidal in the case of $m < n$) and the left and right factors U, V are orthogonal. The factorization is rank-revealing in the sense that

$$\|A - U(:, 1:k)T(1:k, :)V^*\| \approx \inf\{\|A - B\| : B \text{ has rank } k\}. \tag{2}$$

In a factorization resulting from randUTV, the middle matrix T often has elements above the diagonal that are very small in modulus, which means that the diagonal entries of T become excellent approximations to the singular values of A. This type of factorization becomes useful for solving tasks such as low-rank approximation, determining bases for approximations to the fundamental subspaces of A, solving ill-conditioned or over-/under-determined linear systems in a least-square sense, and estimating the singular values of A.

The randomized algorithm randUTV that we describe and implement features the following characteristics that in many environments make it preferable to classical rank-revealing factorizations such as column-pivoted QR (CPQR) and the singular value decomposition (SVD).

- The use of randomization in the algorithm is risk free in the sense that the computed matrices U, T, and V necessarily satisfy (1), up to machine precision. The randomness in the algorithm impacts only the question of how close to optimal (1) is at revealing the numerical rank. That it to high probability reveals the rank almost as well as the SVD, and far better than traditional CPQR, is well supported by both theoretical analysis and extensive numerical experiments; we refer the interested reader to [24], and in particular to Figures 6–10.
- It casts most of its operations in terms of matrix-matrix multiplications, which are highly efficient in parallel computing environments. It was demonstrated in [24] that a straightforward blocked implementation of randUTV executes faster than even highly optimized implementations of CPQR in symmetric multiprocessing (SMP) systems. In this manuscript,

we present an implementation that improves on the performances in [24] for SMP and obtains similar findings for distributed-memory architectures.
- The factorization is built incrementally and can be halted once a specified tolerance has been met. To be precise, if the execution halts once the top k rows of T have been constructed, only $O(mnk)$ flops will have been expended.

In this article, we present two efficient implementations for computing the RANDUTV factorization: the first is for shared-memory machines and the second is for distributed-memory machines. Regarding shared-memory architectures, a previous work [24] proposed a blocked algorithm in which parallelism was extracted on a per-task basis, relying on parallel BLAS implementations and, hence, following a fork-join parallel execution model. Here, we propose a novel algorithm-by-blocks [31], in which sequential tasks are dynamically added to a Directed Acyclic Graph (DAG) and executed by means of a runtime task scheduler (`libflame`'s SuperMatrix [7]). This approach enhances performance by mitigating the effects of the inherent synchronization points in fork-join models and has shown its potential in other linear algebra implementations [8]. In addition, given the recent improvements in terms of performance of modern SVD implementations (e.g., in Intel MKL), we show how runtime-based implementations of RANDUTV are still on par with them in terms of performance. Regarding our second proposal, it is the first time a distributed-memory version of RANDUTV is presented in the literature. Performance results reveal excellent scalability compared with state-of-the-art distributed-memory implementations.

The main contributions of this article compared with the state-of-the-art are as follows.

(1) We propose a novel algorithm-by-blocks for computing the RANDUTV factorization that maximizes performance at no programmability cost.
(2) We have integrated our solution with an existing task-based software infrastructure (`libflame` SuperMatrix), providing an out-of-the-box implementation based on tasks for RANDUTV.
(3) On shared-memory architectures, we provide a detailed study of the optimal block sizes compared with a parallel-BLAS-based solution and report qualitative and quantitative differences between them that can be of interest for the community. Similarly, we have carried out a detailed performance and scalability study on two highly parallel shared-memory machines.
(4) Performance results reveal the benefits of the *algorithm-by-blocks* compared with the *blocked algorithm* on our target testbed, yielding performance improvements between 1.73× and 2.54× for the largest tested matrices. Accelerations compared with proprietary MKL SVD implementations also reveal substantial performance gains, with improvements up to 3.65× for selected cases, and, in general, in all cases that involve relatively large matrices ($n > 4{,}000$)
(5) On distributed-memory architectures, the comparison in terms of execution time with ScaLAPACK SVD, ScaLAPACK CPQR, and PLiC CPQR reveal consistent performance gains ranging from 2.8× to 6.6×, and an excellent scalability on the tested platforms.
(6) On distributed-memory architectures, we provide a detailed performance study regarding block sizes, grid sizes, threads per process, and more, on several nodes.

The article is structured as follows: Section 2 introduces our notation and briefly reviews relevant concepts. In Section 3, we familiarize the reader with the RANDUTV algorithm that was recently described in [24]. Sections 4 and 5 describe the shared- and distributed-memory implementations that form the main contribution of this article. In Section 6, we present numerical results that compare our implementations to highly optimized implementations of competing factorizations. Section 7 summarizes the key findings and outlines some possibilities for further improvements and extensions.

2 PRELIMINARIES

We use the notation $A \in \mathbb{R}^{m\times n}$ to specify that A is an $m \times n$ matrix with real entries. An *orthogonal* matrix is a square matrix whose column vectors each have a unit norm and are pairwise orthogonal. $\sigma_i(A)$ represents the i-th singular value of A, and $\inf(A) = \min_i\{\sigma_i(A)\}$. The default norm $\|\cdot\|$ is the spectral norm. We also use the standard matrix indexing notation $A(c : d, e : f)$ to denote the submatrix of A consisting of the entries in the c-th through d-th rows of the e-th through f-th columns.

2.1 The Singular Value Decomposition (SVD)

Let $A \in \mathbb{R}^{m\times n}$ and $p = \min(m, n)$. It is well known [17, 34, 37] that any matrix A admits an SVD of the form

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

where U and V are orthogonal and Σ is diagonal. We may also speak of the *economic* SVD of A, given by

$$\underset{m\times n}{A} = \underset{m\times p}{U}\ \underset{p\times p}{\Sigma}\ \underset{p\times n}{V^*},$$

in which case U and V are not necessarily orthogonal (because they are not square), but their columns remain orthonormal. The diagonal elements of Σ are the *singular values* $\{\sigma_i\}_{i=1}^{p}$ of A. These are ordered so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{p-1} \geq \sigma_p \geq 0$. The columns u_i and v_i of U and V are called the *left* and *right singular vectors*, respectively, of A.

A key fact about the SVD is that it provides theoretically optimal rank-k approximations to A. Specifically, the Eckart-Young-Mirsky Theorem [14, 28] states that given the SVD of a matrix A as described earlier and a fixed $1 \leq k \leq p$, we have that

$$\|A - U(:, 1 : k)\Sigma(1 : k, 1 : k)(V(:, 1 : k))^*\| = \inf\{\|A - B\| : B \text{ has rank } k\}.$$

A corollary of this result is that the subspaces spanned by the leading k left and right singular vectors of A provide the optimal rank-k approximations to the column and row spaces, respectively, of A. For instance, if P is the orthogonal projection onto the subspace spanned by the left singular vectors of A, then $PA = U(:, 1 : k)\Sigma(1 : k, 1 : k)(V(:, 1 : k))^*$; thus, $\|A - PA\| = \inf\{\|A - B\| : B \text{ has rank } k\}$.

2.2 The QR Decomposition

Given a matrix $A \in \mathbb{R}^{m\times n}$, let $p = \min(m, n)$. A QR decomposition of A is given by

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

where Q is orthogonal and R is upper triangular. If $m > n$, then any QR can be reduced to the "economic" QR

$$\underset{m\times n}{A} = \underset{m\times n}{Q}\ \underset{n\times n}{R}.$$

The standard algorithm for computing a QR factorization relies on Householder reflectors (HQR). A full discussion of the HQR algorithm can be found in [17, 34, 37]. To compute the factorization of the matrix A, p Householder transformations must be computed and applied, where each Householder reflector has the following structure: $H = I - \tau vv^*$. The outputs of HQR are an upper triangular matrix R of the QR factorization, a unit lower triangular matrix $V \in \mathbb{R}^{m\times p}$, and a vector $t \in \mathbb{R}^p$. The matrix V stores the "Householder vectors" v_i and the vector t stores the τ_i values. The

Householder vectors v_i and the scalar values τ_i can be used to cheaply build or apply Q or Q^* (see Section 2.3). In this article, we make critical use of the fact that for $m > n$, the leading p columns of Q form an orthonormal basis for the column space of A.

2.3 Compact WY Representation of Collections of Householder Reflectors

Let b, $0 < b \leq p$ be an integer which we will call the block size, let $A \in \mathbb{R}^{m\times n}$, and let H_i, $i = 1, \ldots, b$ be Householder transformations. As a Householder transformation has the structure $H_i = I - \tau_i v_i v_i^*$, an efficient application of it to a matrix A can be done with just a matrix-vector product and a rank-1 update instead of a matrix-matrix product. Obviously, if the b Householder transformations H_i are applied one after another, the computation requires $O(bmn)$ flops overall. However, both operations (the product and the rank-1 update) are matrix-vector based; therefore, they do not yield high performance on modern architectures.

If b Householder transformations must be applied, the product $H = H_b H_{b-1} \cdots H_2 H_1$ may be expressed in the form

$$H = I - WSW^*,$$

where $W \in \mathbb{R}^{m\times b}$ is lower trapezoidal and $S \in \mathbb{R}^{b\times b}$ is upper triangular. This reformulation of the product of Householder matrices is called the compact WY representation [33]. If the Householder transformations H_i are known, matrices W and S of the compact WY are inexpensive to compute. Matrix W just comprises the Householder vectors v_i, and matrix S can be cheaply computed from the Householder vectors v_i (or W) and the τ_i values.

This expression can be used to build the product HA:

$$HA = A - WSW^*A.$$

The computational cost of this formula is about the same. However, in this case, only matrix-matrix operations are employed (once S is built). Since on modern architectures matrix-matrix operations are usually much more efficient than matrix-vector operations, this approach will yield higher performance. Recall that, due to the existence of cache memories and a much larger flop-to-memory access ratio, the average cost in time of one flop (floating-point operation) in a matrix-matrix operation is usually much smaller (several times) than the average cost of a flop in a matrix-vector operation.

3 THE UTV FACTORIZATION

In this section, we discuss the rank-revealing UTV matrix factorization, establishing its usefulness in computational linear algebra and reviewing efficient algorithms for its computation. In Section 3.1, we review the classical UTV matrix decomposition, summarizing its benefits over other standard decompositions, such as column-pivoted QR and SVD. In Section 3.2, we summarize a recent work [24] that proposes a randomized blocked algorithm for computing this factorization.

3.1 The Classical UTV Factorization

Let $A \in \mathbb{R}^{m\times n}$ and set $p = \min(m, n)$. A UTV decomposition of A is any factorization of the form

$$\underset{m\times n}{A} = \underset{m\times m}{U} \quad \underset{m\times n}{T} \quad \underset{n\times n}{V^*}, \tag{3}$$

where T is triangular and U and V are both orthogonal. In this article, we take T to be *upper* triangular, which is typically the more convenient choice when $m \geq n$. It is often desirable to compute a *rank-revealing* UTV (RRUTV) decomposition. For any $1 \leq k \leq p$, consider the partitioning of T

$$T \rightarrow \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right), \tag{4}$$

where T_{11} is $k \times k$. We informally say that a UTV factorization is rank-revealing if

(1) $\inf(T_{11}) \approx \sigma_k(A)$,
(2) $\|T_{22}\| \approx \sigma_{k+1}(A)$.

The flexibility of the factors in a UTV decomposition renders certain advantages over other canonical forms, such as CPQR and SVD (note that each of these are special cases of UTV factorizations). Since the right factor in CPQR is restricted to a permutation matrix, UTV has more freedom to provide better low-rank and subspace approximations. Also, since UTV does not have SVD's restriction of diagonality on the middle factor, the UTV is less expensive to compute and has more efficient methods for updating and downdating (e.g., see [3, 15, 29, 35, 36]).

3.2 The randUTV Algorithm

In [24], a new algorithm called randUTV was proposed for computing an RRUTV factorization. randUTV is designed to parallelize well, enhancing the RRUTVs viability as a competitor to both CPQR and SVD for a wide class of problem types. It yields low-rank approximations that are comparable to SVD in terms of optimality, while its computational speed is similar to that of CPQR. In fact, as we will demonstrate, our new implementations of randUTV are often significantly *faster* than state-of-the-art algorithms for computing the CPQR. Unlike classical methods for building SVDs and RRUTVs, randUTV processes the input matrix by blocks of b contiguous columns. randUTV shares the advantage of CPQR in that the factorization is computed incrementally and may be stopped early to incur an overall cost of $O(mnk)$, where k is the rank of the computed factorization.

The driving idea behind the structure of the randUTV algorithm is to build the middle factor T with a right-looking approach, that is, in each iteration multiple columns of T (a block column) are obtained simultaneously and only the right part of T is accessed. To illustrate, consider an input matrix $A \in \mathbb{R}^{m \times n}$ and let $p = \min(m, n)$. A block size parameter b with $1 \leq b \leq p$ must be chosen before randUTV begins. For simplicity, assume that b divides p evenly. The algorithm begins by initializing $T^{(0)} := A$. Then, the bulk of the work is done in a loop requiring p/b steps. In the i-th step (starting at 0), a new matrix $T^{(i+1)}$ is computed with

$$T^{(i+1)} := (U^{(i)})^* T^{(i)} V^{(i)},$$

for some orthogonal matrices $U^{(i)}$ and $V^{(i)}$. $U^{(i)}$ and $V^{(i)}$ are chosen such that:

- the leading $(i+1)b$ columns of $T^{(i+1)}$ are upper triangular with $i+1$ diagonal blocks of dimension $b \times b$ on the main diagonal.
- using the partitioning in Equation (4) to define $T_{11}^{(i+1)}$ and $T_{22}^{(i+1)}$, we have that $\inf(T_{11}^{(i+1)}) \approx \sigma_k(A)$ and $\|T_{22}^{(i+1)}\| \approx \sigma_{k+1}(A)$ for $1 \leq k \leq (i+1)b$.
- $T_{11}^{(i+1)}(k,k) \approx \sigma_k(A)$ for $1 \leq k \leq (i+1)b$.

An example of the sparsity patterns for T as the algorithm advances is shown in Figure 1. As can be seen, after each iteration, most of the "mass" or "norm" of the rest of the matrix (the bottom-right part) is moved to the current diagonal block: a large part of the "mass" of the elements in the bottom-right part of the matrix is moved to the current block column, the elements below the diagonal of the current block column are nullified, and the current diagonal block is diagonalized.

Once $T^{(i)}$ is upper triangular, U and V can then be built in the following way:

$$V := V^{(0)} V^{(1)} \cdots V^{(p/b-1)}, \quad U := U^{(0)} U^{(1)} \cdots U^{(p/b-1)}.$$

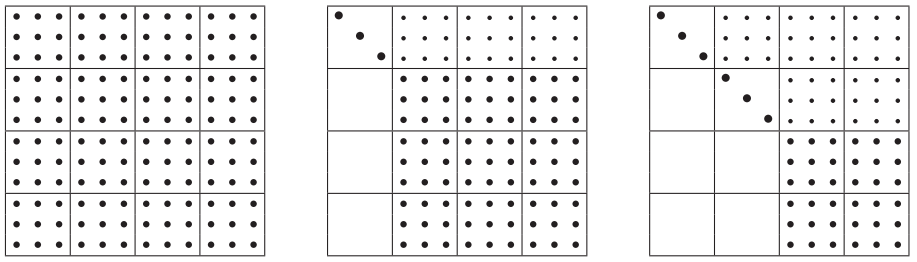


Fig. 1. An illustration of the sparsity pattern after the first three $T^{(i)}$ have been computed for randUTV if $n = 12, b = 3$. randUTV continues until the entirety of $T^{(i)}$ is upper triangular. A blank represents a nullified element and the size of the dot represents an approximation of the absolute value of the element.

In practice, both $V^{(i)}$ and $U^{(i)}$ are constructed in two separate stages and applied to $T^{(i)}$ at different points in the algorithm. We will henceforth refer to these matrices as $V_\alpha^{(i)}$, $U_\alpha^{(i)}$ and $V_\beta^{(i)}$, $U_\beta^{(i)}$ for the first and second stages, respectively. First, $V_\alpha^{(i)}$ is employed to move most of the "mass" of the rest of the matrix (the bottom-right part) to the current block column. Then, $U_\alpha^{(i)}$ is employed to nullify all elements below the main diagonal (moving all of the "mass" of the current block column below the main diagonal to the diagonal block), and then $V_\beta^{(i)}$ and $U_\beta^{(i)}$ are employed to diagonalize the current diagonal block. Also, just one T matrix is stored, whose contents are overwritten with the new $T^{(i+1)}$ at each step. Similarly, in case the matrices U and V are required to be formed, only one matrix U and one matrix V would be stored. Therefore, the outline for randUTV is the following:

(1) Initialize $T := A$, $V := I$, $U := I$.
(2) for $i = 0, \ldots, b/p - 1$:
  (i) Build $V_\alpha^{(i)}$.
  (ii) Update T and V: $T \leftarrow TV_\alpha^{(i)}$, $V \leftarrow VV_\alpha^{(i)}$.
  (iii) Build $U_\alpha^{(i)}$.
  (iv) Update T and U: $T \leftarrow (U_\alpha^{(i)})^* T$, $U \leftarrow UU_\alpha^{(i)}$.
  (v) Build $V_\beta^{(i)}$ and $U_\beta^{(i)}$ simultaneously.
  (vi) Update T, V, and U: $T \leftarrow (U_\beta^{(i)})^* TV_\beta^{(i)}$, $V \leftarrow VV_\beta^{(i)}$, $U \leftarrow UU_\beta^{(i)}$.

The complete algorithm written with the FLAME methodology/notation is presented in Figure 2. Given an $m \times n$ matrix A, this algorithm computes its UTV factorization $A = UTV^*$. The two input parameters *build_U* and *build_V* are Boolean variables that indicate whether the orthogonal matrices U and V must be built. The other input parameters q and b are the number of steps of the power iteration process and the block size, respectively. In this algorithm, the unpivoted_QR function computes the unpivoted QR factorization of the input parameter and returns the following three results: the upper triangular factor R, the matrix with the Householder vectors, and the upper triangular factor S of the compact WY representation. The SVD function computes the SVD factorization of the input parameter and returns the following three results: the singular values in the diagonal of the first output and the two orthogonal matrices U and V.

Next, the computation of the four transformations $V_\alpha^{(i)}$, $U_\alpha^{(i)}$, $V_\beta^{(i)}$, and $U_\beta^{(i)}$ previously mentioned is described in detail. In each case, we indicate the correspondence between each transformation and the part of the above algorithm in which it is computed.

Algorithm: $[U, T, V] := \text{RANDUTV}(A, \text{build\_U}, \text{build\_V}, q, b)$

$T := A$

if (build_U) then $U := \text{EYE}(\, m(A),\ m(A)\,)$

if (build_V) then $V := \text{EYE}(\, n(A),\ n(A)\,)$

Partition $T \rightarrow \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline T_{BL} & T_{BR} \end{array}\right)$, if (build_U) then $U \rightarrow \left(\begin{array}{c|c} U_L & U_R \end{array}\right)$, if (build_V) then $V \rightarrow \left(\begin{array}{c|c} V_L & V_R \end{array}\right)$

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

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

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

Repartition

$$\left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline T_{BL} & T_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} T_{00} & T_{01} & T_{02} \\ \hline T_{10} & T_{11} & T_{12} \\ \hline T_{20} & T_{21} & T_{22} \end{array}\right),$$

if (build_U) then $\left(\begin{array}{c|c} U_L & U_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_0 & U_1 & U_2 \end{array}\right)$, if (build_V) then $\left(\begin{array}{c|c} V_L & V_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} V_0 & V_1 & V_2 \end{array}\right)$

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

// Step 1. Reduction of the mass of T_{12} and T_{22}

1. $G := \text{GENERATE\_IID\_STDNORM\_MATRIX}(m(A) - m(T_{00}), b)$
2. $Y := \left(\left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right)^* \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right)\right)^q \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right)^* G$
3. $[Y, W_V, S_V] := \text{UNPIVOTED\_QR}(Y)$
4. $\left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right) := \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right) - \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right) W_V S_V W_V^*$
5. if (build_V) then $\left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) := \left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) - \left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) W_V S_V W_V^*$

// Step 2. Annihilation of the strictly lower part of T_{11} and all of T_{21}

6. $[\left(\begin{array}{c} T_{11} \\ \hline T_{21} \end{array}\right), W_U, S_U] := \text{UNPIVOTED\_QR}\left(\left(\begin{array}{c} T_{11} \\ \hline T_{21} \end{array}\right)\right)$
7. $\left(\begin{array}{c} T_{12} \\ \hline T_{22} \end{array}\right) := \left(\begin{array}{c} T_{12} \\ \hline T_{22} \end{array}\right) - W_U S_U^* W_U^* \left(\begin{array}{c} T_{12} \\ \hline T_{22} \end{array}\right)$
8. if (build_U) then $\left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) := \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) - \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) W_U S_U W_U^*$

// Step 3. Diagonalization of T_{11}

9. $[T_{11}, U_S, V_S] := \text{SVD}(T_{11})$
10. $T_{01} := T_{01}\ V_S$
11. $T_{12} := U_S^*\ T_{12}$
12. if (build_U) then $U_1 := U_1\ U_S$
13. if (build_V) then $V_1 := V_1\ V_S$

Continue with

$$\left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline T_{BL} & T_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} T_{00} & T_{01} & T_{02} \\ \hline T_{10} & T_{11} & T_{12} \\ \hline T_{20} & T_{21} & T_{22} \end{array}\right),$$

if (build_U) then $\left(\begin{array}{c|c} U_L & U_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_0 & U_1 & U_2 \end{array}\right)$, if (build_V) then $\left(\begin{array}{c|c} V_L & V_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} V_0 & V_1 & V_2 \end{array}\right)$

endwhile

Fig. 2. The randUTV algorithm written with the FLAME methodology/notation. In this algorithm, the UNPIVOTED_QR function returns three results: the upper triangular factor R, the matrix with the Householder vectors, and the upper triangular factor T of the compact WY representation. The SVD function also returns three results: the diagonal and the orthogonal matrices U and V.

3.2.1 Building $V_\alpha^{(i)}$. This transformation is constructed and applied to maximize the rank-revealing properties of the final factorization. Specifically, consider the following partitioning at step i of both matrices T and $V_\alpha^{(i)}$:

$$T \to \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline T_{BL} & T_{BR} \end{array}\right), \quad V_\alpha^{(i)} \to \left(\begin{array}{c|c} I & 0 \\ \hline 0 & (V_\alpha^{(i)})_{BR} \end{array}\right),$$

where the top left (TL) block of each partition is $(i+1)b \times (i+1)b$ and, hence, the top left part of T represents the part already computed. Then, $V_\alpha^{(i)}$ is constructed such that the leading b columns of $(V_\alpha^{(i)})_{BR}$ form an orthonormal approximate basis for the leading b right singular vectors of T_{BR}. An efficient method for such a construction has been developed recently (e.g., see [20, 25–27, 32]) using ideas in random matrix theory. $V_\alpha^{(i)}$ is built as follows:

(1) Draw a thin Gaussian random matrix $G^{(i)} \in \mathbb{R}^{(m-ib)\times b}$.
(2) Compute $Y^{(i)} := ((T_{BR})^* T_{BR})^q (T_{BR})^* G^{(i)}$ for some small integer q.
(3) Perform the unpivoted QR factorization of $Y^{(i)}$ to obtain an orthogonal $Q^{(i)}$ and upper triangular $R^{(i)}$ such that $Y^{(i)} = Q^{(i)} R^{(i)}$.
(4) Set $(V_\alpha^{(i)})_{BR} := Q^{(i)}$.

The parameter q, often called the "power iteration" parameter, determines the accuracy of the approximate basis found in $(V_\alpha^{(i)})_{BR}$. Thus, raising q improves the rank-revealing properties of the resulting factorization but also increases the computational cost. Since raising q too high can also cause numerical overflow, in practice it is chosen to be small ($q = 0, 1, 2$ are typical). For more details, see [20].

The computation and application of the transformation $V_\alpha^{(i)}$ is performed in Step 1 of the randUTV algorithm presented in Figure 2. This step comprises five sentences or lines labeled from 1 to 5. Sentences 1 and 2 perform the "power iteration" process. Sentence 3 computes the orthogonal matrix $V_\alpha^{(i)}$. Sentences 4 and 5 apply this transformation to the output matrices T and V, respectively. For reasons of efficiency in the application of the transformation, $V_\alpha^{(i)}$ is not explicitly computed. Instead, the unpivoted_QR returns factors W_V and S_V such that $V_\alpha^{(i)} = I - W_V S_V W_V^*$ since these two factors allow application of $V_\alpha^{(i)}$ very efficiently.

3.2.2 Building $U_\alpha^{(i)}$. This transformation is constructed and applied to satisfy both the rank-revealing and upper triangular requirements of the RRUTV. Consider the following partitioning at step i of both matrices T and $U_\alpha^{(i)}$:

$$T \to \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline T_{BL} & T_{BR} \end{array}\right), \quad U_\alpha^{(i)} \to \left(\begin{array}{c|c} I & 0 \\ \hline 0 & (U_\alpha^{(i)})_{BR} \end{array}\right), \quad T_{BR} \to \left(\begin{array}{c|c} T_1 & T_2 \end{array}\right),$$

where the top left block (blocks with suffix TL) of each partition is $ib \times ib$. As can be seen, the first two expressions are analogous to those of the previous step. The last expression is a refining of the first one so that T_1 has b columns.

To obtain $(U_\alpha^{(i)})_{BR}$ such that $((U_\alpha^{(i)})_{BR})^* T_1$ is upper triangular, we may compute the unpivoted QR factorization of T_1 to obtain $W^{(i)}, S^{(i)}$ such that $T_1 = W^{(i)} S^{(i)}$. Next, observe that when the building of $U_\alpha^{(i)}$ occurs, the range of the leading b columns of T_{BR} is approximately the same as that of the leading b left singular vectors of T_{BR}. Therefore, the $W^{(i)}$ from the unpivoted QR factorization also forms an orthonormal approximate basis for the leading b left singular vectors

of T_{BR}. Thus, $W^{(i)}$ is an approximately optimal choice of matrix from a rank-revealing perspective. Therefore, we let $(U_\alpha^{(i)})_{BR} := W^{(i)}$.

This computation is performed in Step 2 of the randUTV algorithm presented in Figure 2. This step comprises three sentences labeled from 6 to 8. Sentence 6 computes the unpivoted QR of T_1. Sentences 7 and 8 apply this transformation to the output matrices T and U, respectively. As before, for reasons of efficiency in the application of the transformation $U_\alpha^{(i)}$, it is not explicitly computed. The unpivoted_QR returns factors W_U and S_U such that $U_\alpha^{(i)} = I - W_U S_U W_U^*$.

3.2.3 Building $V_\beta^{(i)}$ and $U_\beta^{(i)}$. These two transformations introduce more sparsity into T at low computational cost, pushing it closer to diagonality and, thus, decreasing $|T(k,k) - \sigma_k(A)|$ for $k = 1, \ldots, (i+1)b$. They are computed simultaneously by calculating the SVD of the top left block of T_{BR} of dimension $b \times b$ (called T_{11}) to obtain $U_S^{(i)}, V_S^{(i)}, D_S^{(i)}$ such that $T_{11} = U_S^{(i)} D_S^{(i)} (V_S^{(i)})^*$. Then, we set

$$V_\beta^{(i)} := \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & V_S^{(i)} & 0 \\ \hline 0 & 0 & I \end{array}\right), \; U_\beta^{(i)} := \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & U_S^{(i)} & 0 \\ \hline 0 & 0 & I \end{array}\right).$$

Following the update step, $T \to (U_\beta^{(i)})^* T V_\beta^{(i)}$, T_{11} is diagonal.

This computation is performed in Step 3 of the randUTV algorithm presented in Figure 2. This step comprises five sentences labeled from 9 to 13. Sentence 9 computes the SVD of T_{11}. Sentences 10 and 11 apply these transformations to the output matrix T. Sentences 12 and 13 apply these transformations to the output matrices U and V, respectively.

3.2.4 Computational Cost. When no orthonormal matrices are generated and q steps of power iteration are applied, the computational cost in flops of the randUTV factorization of an $m \times n$ matrix by using the algorithm presented in Figure 2 is the following (see [24]):

$$(5 + 2q)mn^2 - (3 + 2q)\frac{n^3}{3}.$$

Since the computational cost of the QR factorization of an $m \times n$ matrix is $2mn^2 - 2n^3/3$, the computational cost of the randUTV algorithm of square matrices for $q = 0$, $q = 1$, and $q = 2$ is 3, 4, and 5 times as large as the computational cost of the QR factorization, respectively.

3.2.5 A Numerical Example with a Precision Comparison. Let us show a short numerical example of how the randUTV algorithm works. The matrix to be processed is a 6×6 matrix built as a random permutation of the first mn positive numbers. For this example, we have used the block size $b = 2$ and $q = 2$. In order to reduce the amount of data shown, the matrices in the example are shown with only two decimal digits. Consider an input matrix initialized as

$$A = \begin{pmatrix} 13 & 33 & 5 & 15 & 30 & 32 \\ 2 & 26 & 7 & 24 & 23 & 6 \\ 18 & 28 & 9 & 19 & 36 & 29 \\ 22 & 16 & 25 & 35 & 21 & 14 \\ 8 & 10 & 3 & 31 & 4 & 20 \\ 1 & 17 & 27 & 11 & 34 & 12 \end{pmatrix}$$

After applying Step 1 of the first iteration (right transformations), matrix T becomes

$$T = \left(\begin{array}{cc|cccc} -55.68 & -15.37 & -4.32 & -1.24 & -6.99 & 5.17 \\ -40.00 & -0.84 & 0.66 & 13.01 & 1.26 & -9.89 \\ \hline -59.40 & -9.50 & -5.50 & -5.83 & -0.21 & 1.97 \\ -52.04 & 21.87 & 1.34 & -4.74 & -0.18 & -3.99 \\ -31.61 & 14.34 & -9.94 & 9.29 & -6.95 & 10.60 \\ -43.36 & -3.34 & 18.42 & -3.67 & 13.62 & -3.27 \end{array}\right)$$

Observe how a large part of the "mass" of the matrix has been moved to the first block column (the first two columns), especially the first one. After applying Step 2 of the first iteration (the left triangularization), matrix T becomes

$$T = \left(\begin{array}{cc|cccc} 117.54 & 0.06 & -0.12 & 0.06 & -0.09 & 0.05 \\ 0.00 & -31.97 & 1.77 & -3.30 & 1.27 & 0.45 \\ \hline 0.00 & 0.00 & -4.07 & -3.36 & 2.35 & -1.19 \\ 0.00 & 0.00 & 2.71 & -17.03 & 0.68 & 3.27 \\ 0.00 & 0.00 & -9.10 & 1.32 & -6.47 & 15.36 \\ 0.00 & 0.00 & 19.48 & -3.59 & 15.33 & -4.38 \end{array}\right)$$

Again, note how all of the "mass" in the first block column has been moved to the upper triangular part of the diagonal $b \times b$ block and an important part of the "mass" of the top right part has been moved to the bottom right part of the matrix. After applying Step 3 of the first iteration (the SVD of the diagonal block), matrix T becomes

$$T = \left(\begin{array}{cc|cccc} 117.54 & 0.00 & -0.12 & 0.06 & -0.09 & 0.05 \\ 0.00 & 31.97 & 1.77 & -3.30 & 1.27 & 0.45 \\ \hline 0.00 & 0.00 & -4.07 & -3.36 & 2.35 & -1.19 \\ 0.00 & 0.00 & 2.71 & -17.03 & 0.68 & 3.27 \\ 0.00 & 0.00 & -9.10 & 1.32 & -6.47 & 15.36 \\ 0.00 & 0.00 & 19.48 & -3.59 & 15.33 & -4.38 \end{array}\right)$$

The diagonal block has been diagonalized. The above three steps make up the first iteration. This same process must be repeated for two more iterations to compute the randUTV factorization of the original matrix A.

Table 1 compares the precision of different factorizations for a small example (the 6×6 matrix seen above). Each column in the table shows the diagonal of the obtained factors after the factorization has been computed: the diagonal of Σ (the singular values) for the SVD, the diagonal of R for QR and CPQR, and the diagonal of T for randUTV ($b = 2$ in this case). The QR factorization is not a rank-revealing tool; it is included simply as a reference. The CPQR shows a diagonal with values similar in absolute value to the SVD (but special matrices are known on which it can fail). On the other hand, the diagonals of randUTV are closer to the singular values.

4 EFFICIENT SHARED-MEMORY RANDUTV IMPLEMENTATION

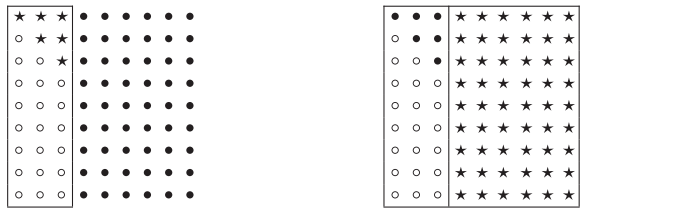
Since the shared-memory multicore/multiprocessor architecture is ubiquitous in modern computing, it is a prime candidate for an efficiently designed implementation of the randUTV algorithm presented in Section 3.2.

Efficient implementations for modern architectures (ranging from unicore processors to SMP to GPUs) must be strongly based on matrix-matrix operations since the ratio of flops to memory accesses in matrix-matrix operations is $O(n)$ ($O(n^3)$ flops to $O(n^2)$ memory accesses), much higher than those of vector-vector and matrix-vector operations. This increased ratio provides

Table 1. Diagonals of the Obtained Factors for Several Factorizations

	SVD	QR	CPQR	randUTV $q = 0$	randUTV $q = 1$	randUTV $q = 2$
1	117.54	−32.34	−65.86	110.07	117.51	117.54
2	32.76	−35.04	−38.79	22.81	29.97	31.97
3	29.41	27.38	−23.15	30.83	29.41	29.92
4	17.74	−26.43	−17.36	11.18	14.09	17.18
5	10.85	−7.62	13.42	24.68	14.93	11.28
6	4.47	−15.59	−7.07	4.56	4.47	4.47

For SVD the diagonal of Σ (the singular values) is shown, for QR and CPQR the diagonal of R is shown, and for randUTV the diagonal of T is shown for several values of q (in this case $b = 2$).



(1) After Compute_dense_QR(A_0) (2) After Apply_left_Qt_of_Dense_QR(A_0, A_1)

Fig. 3. An illustration of the first tasks performed by a blocked algorithm for computing 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 (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.

much higher performances on modern computers since current processors and GPUs can perform many flops per memory access.

There are usually two options to reformulate an algorithm to employ matrix-matrix operations: *blocked algorithms* and *algorithms-by-blocks*. Blocked algorithms execute several iterations of the scalar algorithm at the same time, thus, computing multiple columns (or rows) of the desired result in each iteration of its main loop. This design decision allows most of the computations to be cast in terms of matrix-matrix operations (Level 3 BLAS). See Figure 3 for a graphical representation of a blocked factorization (the text corresponds to a QR factorization, but an LU factorization would be analogous). On the other hand, algorithms-by-blocks apply the scalar algorithm on square blocks instead of scalars. By working on square blocks, matrix-matrix operations (Level 3 BLAS) can be heavily employed. See Figure 4 for a graphical representation of an algorithm-by-blocks for computing a factorization (the text corresponds to a QR, but an LU would be similar).

The main advantage of the randUTV algorithm described earlier is its efficiency in parallel computing environments mainly because it can be *blocked* easily. That is, it drives multiple columns of the input matrix A to upper triangular form in each iteration of its 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 and close to the peak speed, randUTV should yield high performance. Martinsson et al. [24] provided an efficient blocked implementation of the randUTV algorithm that was faster than competing rank-revealing

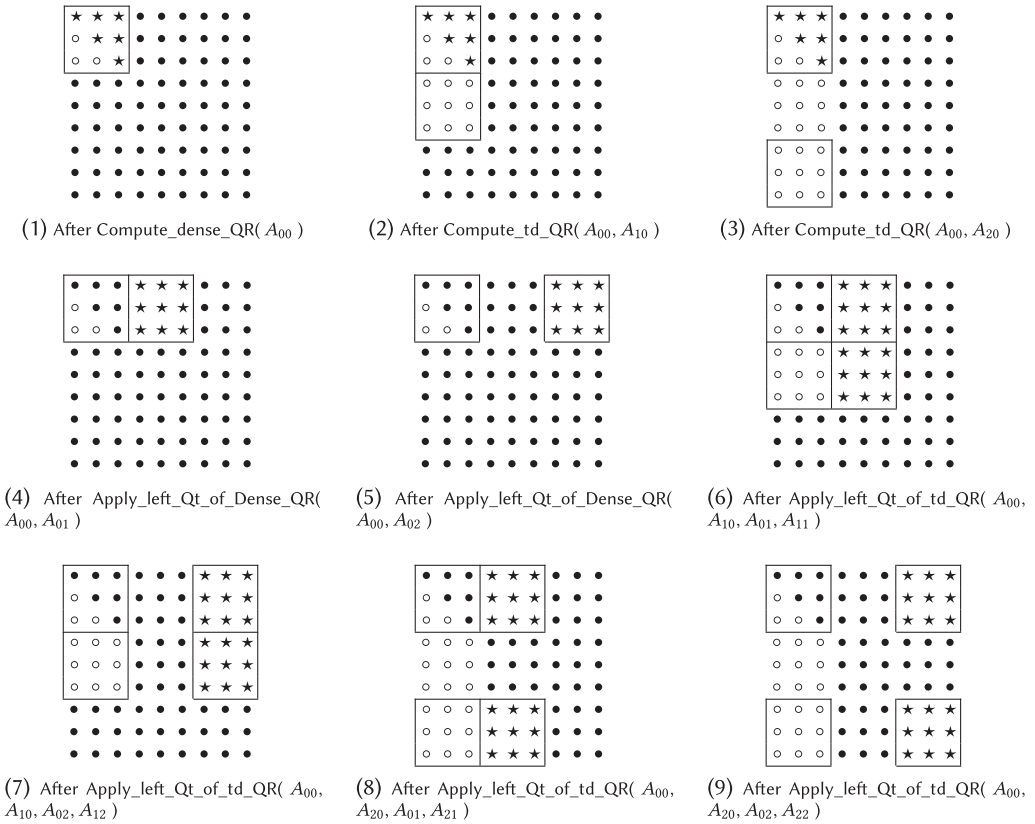


Fig. 4. An illustration of the first tasks performed by an algorithm-by-blocks for computing the QR factorization. The • symbol represents a non-modified element by the current task, ⋆ represents a modified element by the current task, and ○ represents a nullified element by the current task (they are shown because they store information about the Householder transformations that will be used later to apply them). The continuous lines surround the blocks involved in the current task.

factorizations, such as SVD and CPQR, *despite* RANDUTV *having a much higher flop count than both of them.* As usual in many linear algebra codes, this blocked implementation of RANDUTV kept all of the parallelism inside the BLAS library.

In Section 4.1, we discuss a scheme called algorithms-by-blocks for designing more efficient algorithms on architectures with multiple or many cores. Section 4.2 explores the application of algorithms-by-blocks to RANDUTV. Sections 4.3 and 4.4 familiarize the reader with the software used to implement algorithms-by-blocks and a runtime system to schedule the various matrix operations, respectively.

4.1 Algorithms-by-blocks: An Overview

Pushing all parallelism into multithreaded implementations of the BLAS library can make programming much easier, but its performance is limited for more than just a few cores [31], which can become a serious limitation. Most high-performance blocked algorithms for computing factorizations (such as Cholesky, LU, and QR) involve at least one task in each iteration that works on very few data (therefore, non-parallelized) and whose result is required in the following operations

of the current iteration. These serial tasks usually involve the processing of blocks with at least one small dimension b, where b is typically chosen to be 32 or 64, usually much smaller than the matrix dimensions. For instance, in the blocked Cholesky factorization, this performance-limited task is the computation of the Cholesky factorization of the diagonal block. In the blocked QR and LU factorizations, this performance-limited part is the computation of the factorization of a part of the current block column (see Step 1 in Figure 3). Thus, since these serial tasks form a synchronization point, all but one core in the system are left idle during these computations. For only four or five total cores, time lost is minimal. As the number of available cores increases, though, a significant waste in efficiency builds up. The randUTV factorization is also affected by this problem, since each iteration contains three tasks of this type: the QR factorization of matrix Y, the QR factorization of the current block column of T, and the SVD of the diagonal block of T.

This led us to seek a technique other than blocking to obtain higher performance, 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. In randUTV, for instance, multiple columns of the input are typically processed in one iteration of the main loop. Processing one column at a time would require matrix-vector operations (Level-2 BLAS) in each iteration, but processing multiple columns at a time aggregates these into much more efficient matrix-matrix operations (Level-3 BLAS).

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 submatrix 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 allows more flexibility than blocking in ordering the operations, eliminating the bottleneck caused by the synchronization points in the blocking method. The performance benefits obtained by the algorithm-by-blocks approach with respect to the approach based on blocked algorithms for linear algebra problems on shared-memory architectures are usually significant [7, 31].

4.2 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. Therefore, its structure is the same as the one described earlier (see Figure 2) and is not shown. The key difference is that this new philosophy of defining smaller tasks (than those of blocked algorithms) that operate on square blocks can allow greater flexibility in the order of completion.

As said before, the algorithms-by-blocks must raise the granularity of the data. Thus, a block size b must be chosen (in practice, block sizes around $b = 256$ work well). For simplicity, assume that 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:

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

where each submatrix or block B_{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 partitioned accordingly. In this partitioning, we have not employed

the letter T for the above blocks to avoid confusion with the blocks T_{ij} of the randUTV algorithm. In that algorithm, T_{11} is the $b \times b$ diagonal block being processed, T_{22} is the right-bottom part of the matrix to be processed, T_{00} is the left-top part of the matrix already processed, and so on. Thus, their dimensions (except for T_{11}) depend on the current iteration of the algorithm. In contrast, all of the B_{ij} blocks are of dimension $b \times b$.

Once the blocks have been defined, we must redefine all operations of the algorithm in terms of these blocks. The blocks or submatrices B_{ij} (and those of the rest of the matrices) are treated as the fundamental unit of data in the algorithm so that each operation is expressed only in these terms. Therefore, all of the computations that involve large parts of the matrices in the randUTV algorithm (see Figure 2) must be broken into small tasks operating on these square blocks.

For instance, the first instruction of Step 1 of the randUTV algorithm is the generation of a matrix G with random normal entries. As this operation must be expressed only in terms of square blocks, for the first iteration this operation will comprise the following tasks: `Generate_random(` G_1 `)`, `Generate_random(` G_2 `)`, ... `Generate_random(` G_N `)`, where the routine `Generate_random` fills its argument (one square block of dimension $b \times b$) with random normal entries.

The two most computationally expensive parts in the algorithm are the following. The first is the computation of matrix Y (the power iteration loop), performed in Step 1. The second is the factorization of a column block and the updating of the corresponding matrices, which is performed in Step 1 (factorization of Y) and in Step 2 (factorization of $\binom{T_{11}}{T_{21}}$). We comment on them next.

4.2.1 Computation of Matrix Y. The computation of matrix Y in the power iteration loop is as follows:

$$Y := ((T_{BR})^* T_{BR})^q (T_{BR})^* G = \left(\left(\left.\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right.^* \left.\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right. \right)^q \left.\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right.^* G.$$

Although it is computationally very expensive, it comprises just a series of highly efficient matrix-matrix products.

This operation will be broken into several tasks so that each calculates the product of two square blocks of T and the accumulation into a block of Y. In the simplified case, where $q = 0$, we have $M \times N$ products of two blocks. For instance, in the first iteration, the list of tasks would be the following (recall that matrix T is partitioned into square blocks B_{ij}):

$$\begin{array}{rcl} Y_1 & = & Y_1 + B_{11}^* G_1, \\ Y_1 & = & Y_1 + B_{21}^* G_2, \\ Y_1 & = & Y_1 + B_{31}^* G_3, \\ & \vdots & \\ Y_2 & = & Y_2 + B_{12}^* G_1, \\ Y_2 & = & Y_2 + B_{22}^* G_2, \\ Y_2 & = & Y_2 + B_{32}^* G_3, \\ & \vdots & \end{array}$$

In this case, the maximum number of tasks that can be executed in parallel depends on the number of block rows of Y since two accumulations on the same block row of Y cannot be performed at the same time. In the first iteration of the algorithm, the number of block rows is M when the product involves T_{BR} and N when the product involves T_{BR}^*. As the algorithm advances, this parallelism decreases in one unit for each iteration, which might be a limiting factor for the parallelism in this operation, especially on small matrices and the last iterations.

4.2.2 Incremental QR Factorization. The second most computationally expensive part in the randUTV algorithm is the factorization of a block column and the subsequent updating of the corresponding matrices. As can be seen, Step 1 of randUTV requires the factorization of Y, whereas Step 2 requires the factorization of $\begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix}$.

The traditional QR factorization of these block columns does not decompose this operation into small tasks that operate on square blocks. Since an algorithm-by-blocks for computing the randUTV requires that the QR factorization performed inside works also on $b \times b$ blocks, we have employed an alternative algorithm, usually known as the incremental QR factorization, based on the idea of updating a QR factorization when more rows are added to the input matrix. See, for example, [19, 30, 31] for details on this approach to the 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 that is required for randUTV_AB. In this way, this operation can be conceptualized as occurring in an iteration with a high number of tasks, each one operating on a few square blocks.

Figure 4 shows this process for a 9×9 matrix with block size $b = 3$. In this figure, the continuous lines show the 3×3 block or blocks involved in the current task, $\bullet$ represents a non-modified element by the current task, $\star$ represents a modified element by the current task, and $\circ$ represents a nullified element by the current task. The nullified elements are shown because, as usual, they store information about the Householder transformations that will be later used to apply these transformations. The first task, called *Compute_dense_QR*, annihilates the leading dense block A_{00} by employing a QR factorization. The second task annihilates block A_{10}. This task is called *Compute_td_QR*, where *td* stands for triangular-dense because of the shape of the two arguments: A_{00} is triangular, whereas A_{10} is dense. Analogously, the third task annihilates block A_{20}. After these three tasks have been successfully executed, all of the elements below the main diagonal have been annihilated and the rest of the matrix must be correspondingly updated.

The fourth task, called *Apply_left_Qt_of_dense_QR*, applies the Householder transformations obtained in the first task (and stored in A_{00}) to block A_{01}. The fifth task performs the same operation onto A_{02}. The sixth task, *Apply_left_Qt_of_td_QR*, applies the transformations obtained in the second task to blocks A_{01} and A_{11}. The seventh task performs the same operation onto A_{02} and A_{12}. Analogously, the eighth and ninth tasks apply the same processing as the two previous ones on the first and third row of blocks. By taking advantage of the zeros present in the factorizations for each iteration, a well-implemented `QR_AB` cost essentially no more flops than the traditional blocked unpivoted QR. The algorithm is described in greater detail in [7, 30, 31].

To obtain high performance when executing the previous incremental QR factorization in a parallel environment, runtime systems should execute as many tasks as early and simultaneously as possible. Note that in this small example of dimension 9×9, the fourth and fifth tasks can be executed just after the first task since they need to access (read) only the strictly lower triangular part of A_{00}. However, since the upper part of A_{00} is being updated by the second and third tasks, runtime systems will not allow the early execution of the fourth and fifth tasks (at the same time as the second and third tasks are executed). Therefore, A_{00} becomes a false bottleneck or data dependency. One way to fix this bottleneck and to increase the parallelism is to employ an auxiliary square block to make a copy of A_{00} just after the first task has been executed. In fact, this copy is an additional task (but much faster). In this case, the original fourth and fifth tasks could be executed just after the copy while at the same time the second and third tasks are being computed (and they are updating A_{00}). In general, by employing two auxiliary blocks of dimension $b \times b$ (one for the factorization of Y and another one for the factorization of the current block column of A), the annihilation of the block column and the updating of the first block row could be executed at the same time.

The maximum number of tasks that can be executed in parallel depends on the number of block rows to update when applying transformations from the right and the number of column blocks to update when applying transformations from the left. For instance, in Figure 4, the maximum number of tasks that can be executed in parallel is two: after executing task 1, tasks 2 and 3 can be executed in parallel, and so on. In the first iteration of the algorithm, the number of block rows is M when updating from the right or N when updating from the left. As the algorithm advances, this parallelism decreases in one unit for each iteration, which might be a limiting factor, especially on small matrices, the last iterations, and when no orthonormal matrices are built. In the latter case, the maximum number of tasks that can be executed in parallel is larger since blocks of matrix T could be updated at the same time as blocks of U and/or V.

4.2.3 Variants of the randUTV_AB *Algorithm.* Next, we describe some improvements to the previous algorithm. Considering the earlier comments, we have implemented and assessed the following variants of the randUTV_AB algorithm.

- Variant `v00`: This is a naïve implementation of the basic randUTV_AB algorithm described earlier.
- Variant `v01`: Two auxiliary blocks of dimension $b \times b$ have been employed to remove data dependencies and bottlenecks, and increase parallelism when computing the incremental QR factorization. This improvement has also been applied to the next variants.
- Variant `v02`: In each iteration, the computation of the SVD of the diagonal block has been moved upwards to be able to unlock as many tasks as soon as possible to increase the parallelism. Once the current block column, $\binom{T_{11}}{T_{21}}$, has been upper-triangularized and before $\binom{T_{12}}{T_{22}}$ is updated (and maybe matrices U and V), the computation of the SVD of the T_{11} can be computed to minimize the *fork-join* effect and activate as many tasks as possible.
- Variant `v03`: This implementation increases the parallelism of the computation of the matrix Y in each iteration by rewriting each product by using an auxiliary matrix of the same size as Y (and another one of the same size as G). Recall that this power iteration consists of a series of matrix-matrix products, where the second matrix is a block column. The shape of this second matrix clearly limits the parallelism since several tasks (block-block products) must be accumulated onto the same block of Y.

  The technique employed is to rewrite each matrix-matrix product into the same number of block-block products, but accumulating on different left-hand sides by using an auxiliary matrix. For instance, the product $Y := TG$ is rewritten as:

$$Y := TG = (T_E, T_O) \begin{pmatrix} G_E \\ G_O \end{pmatrix} = T_E G_E + T_O G_O,$$

  where T_E contains the even block columns of T, T_O contains the odd block columns of T, G_E contains the even block rows of G, and G_O contains the odd block rows of G.

  This expression is computed as

$$\begin{aligned} Y &:= T_E\, G_E \\ Z &:= T_O\, G_O \\ Y &:= Z + Y. \end{aligned}$$

  In this sequence, in the first iteration of the algorithm, the first line allows execution of up to M tasks simultaneously, whereas the second line allows execution of up to M more tasks simultaneously. In this way, the number of tasks that can be concurrently executed is twice

the original number, with the only drawback being up to M additional tasks of accumulation ($Y := Z + Y$), which are very simple and fast.

- Variant `v04`: This implementation increases the parallelism of the QR updates from both the left side and the right side.

  When computing a dense QR factorization of a block or a triangular-dense QR factorization of two blocks (see Figure 4), the maximum number of tasks that can be executed in parallel depends on the number of blocks to be updated with that block factorization. The larger the number of blocks to be updated, the larger the parallelism.

  A way to increase this parallelism is to apply the incremental QR factorization described earlier to the even block rows of Y, whereas at the same time the incremental QR factorization is applied to the odd block rows of Y. Hence, the maximum number of tasks is twice the original number, with the only drawback being that at the end, the second block row must be nullified (with additional tasks). Although the second block row is upper triangular, in order to simplify the programming, the usual triangular-dense QR factorization is employed. For instance, at the beginning of the first iteration, once Y has been computed, Y_0 and Y_1 are triangularized at the same time. Then, the first block row of A can be updated with the transformations of Y_0 at the same time as the second block row of A is updated with the transformations of Y_1, thus doubling the number of available tasks. Then, Y_2 and Y_3 must be nullified and the corresponding blocks updated at the same time, and so on. Finally, block Y_1 should be nullified based on Y_0 with additional tasks that were not in the original algorithm. This method is a modification of one of the communication-avoiding QR factorizations presented in [16, 22], adapted to work with Householder transformations and well suited for square matrices.
- Variant `v05`: This implementation can improve performance by using a hierarchical storage in main memory. Instead of storing the data in column-major order, each matrix is stored in a block column-major order with block size b (the same as the algorithmic block size).

4.3 The FLAME Abstraction for Implementing Algorithm-by-blocks

A nontrivial obstacle to implementing an algorithm-by-blocks is the issue of programmability. Using the traditional approach of calls to LAPACK [1] and BLAS [23] libraries for the computational steps, keeping track of indexing quickly becomes complicated and error prone. The FLAME (Formal Linear Algebra Methods Environment) project [18, 21] is one solution to this problem. FLAME is a framework for designing linear algebraic algorithms that departs from the traditional index-based-loop methodology. Instead, the input matrix is interacted with as a collection of submatrices, basing its loops on re-partitionings of the input. The FLAME API [5] for the C language codifies these ideas, enabling a user of the API to code high-performance implementations of linear algebra algorithms at a high level of abstraction. Furthermore, the methodology of the FLAME framework and its implementation in terms of the `libflame` library [38] makes it a natural fit for use with an algorithm-by-blocks. Thus, the actual code for the implementation of randUTV_AB looks practically the same as the written version of the algorithm given in Figure 2 since all of the changes are performed inside each operation.

4.4 Executing the Algorithm-by-blocks

The runtime system called SuperMatrix [7] is an integral part of the `libflame` distribution and has been leveraged to expose and exploit task-level parallelism in randUTV_AB. The execution of any program proceeds in two phases: the analysis stage and the execution stage.

	Operation	Operands		
		In		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
	Gemm_tn_oo: $C = C + A^*B$	A_{11}	G_1	Y_1
	Comp_dense_QR			Y_0, S_0
	Copy	Y_0		E_0
	Comp_td_QR			Y_0, Y_1, S_1
	Apply_right_Q_of_dense_QR	E_0	S_0	A_{00}
	Apply_right_Q_of_dense_QR	E_0	S_0	A_{10}
	Apply_right_Q_td_QR	Y_1	S_1	A_{00}, A_{01}
	Apply_right_Q_td_QR	Y_1	S_1	A_{10}, A_{11}
	Comp_dense_QR			A_{00}, X_0
	Copy	A_{00}		D_0
	Comp_td_QR			A_{00}, A_{10}, X_1
	Apply_left_Qt_of_dense_QR	D_0	X_0	A_{01}
	Apply_left_Qt_of_td_QR	A_{10}	X_1	A_{01}, A_{11}
	Keep_upper_triang			A_{00}
	Set_to_zero			A_{10}
	Svd_of_block			A_{00}, P_0, Q_0
	Gemm_abta: $A = B^*A$	P_0		A_{01}
	Svd_of_block			A_{11}, P_0, Q_0
	Gemm_aabt: $A = AB^*$	Q_0		A_{01}

Fig. 5. A list of the operations queued up by the runtime during the analyzer stage of the RANDUTV algorithm in the simplified case that the block size b is $n/2$. The In column specifies pieces of required input data. The In/Out column specifies required pieces of data that will be altered upon completion of the operation. The execution of the RANDUTV algorithm-by-blocks comprises two phases. In the first stage (the analysis), the runtime builds the list of tasks recording the dependencies. In the second stage (the scheduling/dispatching stage), the runtime schedules and executes the tasks in the queue dynamically. The tasks may be completed in any order that does not violate the data dependencies encoded in the table.

(1) In the analysis stage, instead of executing the code sequentially, the runtime builds a list of tasks recording the dependency information associated with each operation and placing it in a queue. An example of the queue built up by the runtime for RANDUTV_AB for the case that $A \in \mathbb{R}^{n \times n}$ and the block size is $b = n/2$ is given in Figure 5.
(2) In the execution stage, the tasks in the queue are dynamically scheduled and executed. Each task is executed as soon as its input data become available and a core is free to complete the work but keeps a sequentially consistent order among tasks.

Figure 6 shows an actual DAG that illustrates the data dependencies between tasks for the complete execution. From the code perspective, the main FLAME formulation (see Figure 2) remains unchanged, replacing the actual calls to BLAS/LAPACK codes by the task creation — including input/output per-task data — and its addition to the DAG. From that point on, the scheduling/dispatching stage is transparent for the developer.

5 EFFICIENT DISTRIBUTED-MEMORY RANDUTV IMPLEMENTATION

Distributed-memory computing architectures are commonly used for solving large problems as they extend both memory and processing power over single systems. In this section, we discuss

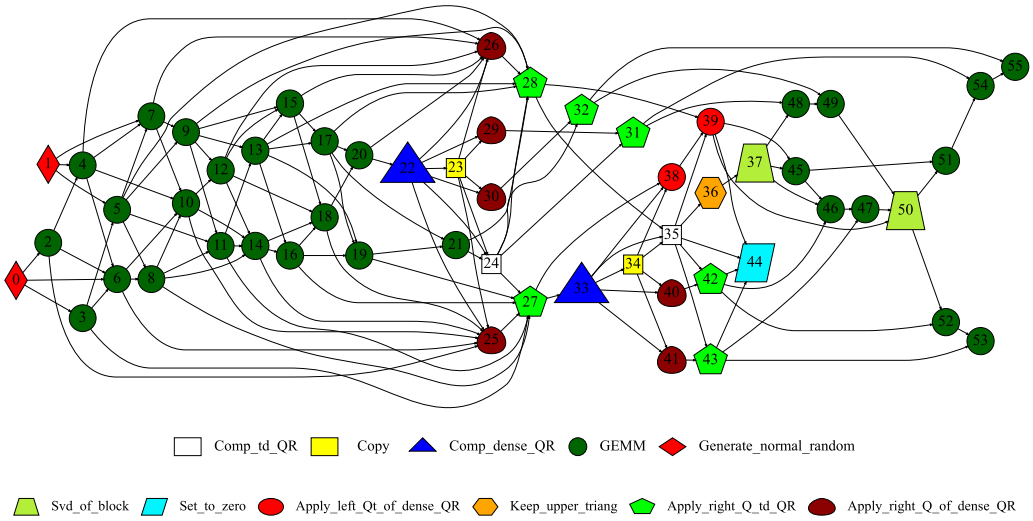


Fig. 6. Complete Directed Acyclic Graph exposed to the runtime task scheduler during the dispatching stage in the simplified case that the block size is $b = n/2$. The colors used for each task follow the convention employed in Figure 5.

an efficient implementation of randUTV for distributed-memory architectures. In Section 5.1, we discuss the infrastructure employed to implement and assess our new codes. In Section 5.2, we describe the implementation of randUTV in distributed-memory architectures.

5.1 ScaLAPACK Overview

The ScaLAPACK (Scalable LAPACK) software library [6, 9, 11] was used in the presented implementations since it provides a wide functionality to implement linear algebra applications on distributed-memory architectures. This library provides much of the functionality of LAPACK for distributed-memory environments. It hides most of the communication details from the developer with an object-based API, where each matrix's object information is passed to library routines. This design choice enhances the programmability of the library, enabling codes to be written similarly to a standard LAPACK implementation. However, as it is implemented in Fortran-77, its object orientation is not perfect and the programming effort is larger.

In addition to a wide functionality to implement new linear algebra codes, ScaLAPACK provides an efficient SVD and a Level-2 BLAS CPQR. A Level-3 BLAS CPQR from the PLiC library [4], based also on ScaLAPACK, was employed. The availability of these factorizations is key to being able to assess and compare our new distributed-memory randUTV code. Another advantage of ScaLAPACK is that it is usually included in MKL, which increases its portability and simplifies its download and installation.

ScaLAPACK was designed to be portable to a variety of computing distributed-memory architectures and relies on only three external libraries. The first is the sequential LAPACK [1]. The second is the sequential BLAS (Basic Linear Algebra Subroutines) [12, 13, 23], providing specifications for the most common operations involving vectors and matrices. The third is the BLACS (Basic Linear Algebra Communication Subroutines), which, as the name suggests, is a specification for common matrix and vector communication tasks [2]. The PBLAS library is a key module inside ScaLAPACK. It comprises most BLAS routines rewritten for use in distributed-memory

environments. This library is written using a combination of the sequential BLAS library and the BLACS library.

All ScaLAPACK routines assume the so-called "block-cyclic distribution" scheme [10]. The block-cyclic distribution scheme involves four parameters. The first two, m_b and b, define the block size, that is, the dimensions of the submatrices used as the fundamental unit for communication among processes. Despite this flexibility, nearly all of the linear algebra main routines usually employ $m_b = n_b$ for the purpose of simplicity. The last two parameters, typically called P and Q, determine the shape of the logical process grid.

5.2 Implementation of randUTV in Distributed-Memory Architectures

In this section, we describe our implementations of the randUTV algorithm on distributed-memory architectures.

Since Algorithm 2 is very rich in BLAS-3 operations and this type of operation can also be very efficient in distributed-memory architectures, we have employed this same algorithm for the distributed-memory implementation.

The distributed-memory implementation of randUTV uses the standard blocked algorithm of [24] rather than the algorithm-by-blocks approach (as discussed in Section 4.2) to assess a fair comparison since the only available SVD and CPQR factorizations employed the blocked algorithms approach as well. The use of the ScaLAPACK library to implement the randUTV algorithm allows for a more fair comparison since the underlying basic building blocks (matrix-matrix products, Householder transformations, etc.) of all of the assessed routines are basically the same.

Unlike the algorithm-by-blocks approach on shared-memory architectures, the algorithm-by-blocks approach has some issues on distributed-memory architectures. The DAG allows execution of any task as soon as its operands are ready, but the higher efficiency of collective communications needs a more coordinated methodology. Although this has been avoided in some high-performance distributed-memory libraries by using special languages, they do not provide for an SVD and CPQR factorizations to compare with our randUTV.

Like in some other factorizations (QR, SVD, etc.), when applying randUTV to a matrix with $m \gg n$, it is best to perform an unpivoted QR factorization first and then perform the randUTV factorization on the resulting square triangular factor.

The two most computationally expensive parts in the randUTV algorithm are the following: the computation of matrix Y (in Step 1), and the updating of matrices T, U, and V with the Householder transformations of QR factorizations (in Steps 1 and 2).

5.2.1 Computation of Matrix Y. Even for $q = 0$, the computation of matrix Y is one of the most expensive parts in this algorithm. As this operation is basically a series of matrix-matrix products on distributed data, this operation is usually very efficient, and its speed is determined by the ScaLAPACK library being employed. To build matrix Y, a series of matrix-matrix products (routine `pdgemm` of ScaLAPACK) must be employed.

5.2.2 QR Factorization and Updating. The other most expensive part in the randUTV algorithm is the updating of the corresponding matrices after the QR factorization of some block columns. Matrix Y and block $\begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix}$ must be factorized in Step 1 and Step 2, respectively.

Since matrix Y is a block column, to factorize it and to update the corresponding parts of T and V, routines `pdgeqr2` (to factorize Y), `pdlarft` (to compute factor S_V), and `pdlarfb` (to update matrices T and V) have been employed. Obviously, the most expensive part is the updating with the `pdlarfb` routine of matrices T and V from the right after the factorization of Y. Since this updating requires a product such as $A(I - W_V S_V W_V^*) = A - AW_V S_V W_V^*$, efficient general (`pdgemm`)

and triangular (pdtrmm) matrix-matrix products are usually employed inside the pdlarfb routine, which guarantees high performance.

The factorization of $\begin{pmatrix} T_{11} \\ T_{21} \end{pmatrix}$ and the corresponding updates are analogous. In this case, T is updated from the left and U is updated from the right. As before, these updates require only efficient general (pdgemm) and triangular (pdtrmm) matrix-matrix products.

On the other side, on distributed-memory machines, we implemented and assessed another approach for computing QR factorization and updating a matrix by using the incremental QR factorization. This incremental QR factorization is similar to the method employed in task scheduling for shared-memory machines (but without task scheduling since ScaLAPACK does not allow it). In our approach to the incremental QR factorization and updating, each block iteration has two stages. The first stage is to compute the dense QR of the diagonal block and then apply the updating of the rest of the corresponding matrices. The second stage is to annihilate all square blocks below the diagonal block by moving the "mass" to the diagonal block with the corresponding update. This second stage is the so-called triangular-dense QR factorization since the top block is already triangular (the diagonal block) and the bottom block is dense (the block to be annihilated).

5.2.3 Other Operations. Two computationally cheaper operations in each iteration of the randUTV algorithm are the SVD of the diagonal block and its corresponding updates (in Step 3) and the generation of matrix G (in Step 1). The computation of the SVD of the diagonal block in each iteration, T_{11}, is very fast since no communication is required. To compute it, routine dgesvd from LAPACK is employed. Then, the updating of matrices T, U, and V requires employment of the output matrices U_S and V_S of this SVD. These updates require products with replacement (one operand is also the result), such as $A := AB$. Therefore, a small auxiliary space and an additional copy have been employed in addition to the pdgemm routine. However, the overall cost of this part is much smaller since only small parts of T, U, and V (some block rows and block columns) must be updated.

The other operation discussed earlier, the generation of matrix G, was performed in parallel, avoiding any performance bottleneck. Unlike uniform random number generators, LAPACK and ScaLAPACK do not include a normal random number generator, which is key in each iteration to forming an orthonormal approximate basis for the leading b right singular vectors of T_{22} (in Step 1). Thus, the Box-Mueller transformation method was employed to convert uniform variates to normal variates.

To conclude, it is obvious that general matrix-matrix multiplications (pdgemm) form the dominant cost within the implementation of randUTV. Since this operation belongs to Level-3 BLAS and can be very efficiently implemented on this type of architecture, the final implementation of randUTV can be expected to be very efficient.

6 PERFORMANCE ANALYSIS

In this section, we investigate the speed of our new implementations of the algorithm for computing the randUTV factorization and compare it to the speeds of highly optimized methods for computing the SVD and the CPQR factorization. In all of the experiments, double-precision real matrices were processed.

To fairly compare the different implementations being assessed, the flop count or the usual flop rate could not be employed since the computation of the SVD, CPQR, and 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 since they vary greatly because of the large range of matrix dimensions assessed in the experiments. Therefore, scaled computational times (absolute computational times divided by n^3) are usually employed. Hence, the lower the

scaled computational times, the better the performance. Since all of 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 (as in time-to-solution). Those scaled times are multiplied by a constant (usually 10^{10}) to make the figures in the vertical axis more readable.

On the other hand, a few plots show speedups. The speedup is usually computed as the quotient of the time obtained by the serial (one core) implementation and the time obtained by the parallel implementation (on many cores). Thus, this concept communicates how many times the parallel implementation is as fast as the serial one. Hence, the higher the speedups, the better the performance of the parallel implementation. This measure is usually very useful in checking the scalability of an implementation. Note that in this type of plot, every implementation compares against itself on one core.

Some additional plots show scaled velocities. In those plots, one of the methods is chosen as a reference with a velocity of 1, and the other methods are scaled accordingly. Thus, this concept communicates how many times faster some method is compared to the chosen one as reference 1. Hence, the higher the velocity, the better the performance.

The important question of the algorithm's accuracy and ability to reveal the rank of the target matrix is addressed in [27]. To summarize, the rank-revealing properties far exceed those of the standard QR factorization and, in certain cases, they can approach that of the theoretically optimal SVD. We refer the reader to [27] for a thorough discussion.

6.1 Accuracy

We computed the following residuals for all of our new shared- and distributed-memory implementations for computing the RANDUTV factorization: $\|A - UTV^*\|_F$, $\|I - U^*U\|_F$, $\|I - V^*V\|_F$, and $\|\text{svd}(A) - \text{svd}(T)\|_F$, where $\text{svd}(X)$ is a vector with the singular values of X ordered. In all of the experiments performed on many matrix dimensions and block sizes, the maximum residuals of the new implementations of RANDUTV were similar to the analogous residuals that we computed in other factorizations, such as SVD, CPQR, and QR.

6.2 Computational Speed on Shared-Memory Architectures

6.2.1 Experimental Setup. We employed the following computers in the experiments with shared-memory architectures:

- `marbore`: It featured two Intel Xeon CPUs E5-2695 v3 (2.30 GHz), with 28 cores and 128 GB of RAM in total. In this computer, the so-called *Turbo Boost* mode of the two CPUs was turned off in our experiments.

  Its OS was GNU/Linux (kernel version 2.6.32-504.el6.x86_64). GCC compiler (version 6.3.0 20170516) was used. Intel Math Kernel Library (MKL) version 2018.0.1 Product Build 20171007 for Intel(R) 64 architecture was employed since LAPACK routines from this library usually deliver much higher performance than LAPACK routines from the Netlib repository.
- `mimir`: It featured two Intel Xeon CPUs Gold 6254 (3.10 GHz), with 36 cores and 791 GB of RAM in total. The *Max Turbo Frequency* of the CPUs was 4.00 GHz.

  Its OS was GNU/Linux (kernel version 5.0.0-32-generic). Intel C compiler (version 19.0.5.281 20190815) was used. Intel Math Kernel Library (MKL) version 2019.0.5 Product Build 20190808 for Intel(R) 64 architecture was employed because of the same reason given earlier.
- `makalu`: It featured two Intel Xeon CPUs Gold 6138 (2.00 GHz), with 40 cores and 92 GB of RAM in total. The *Max Turbo Frequency* of the CPUs was 3.70 GHz.

Its OS was GNU/Linux (kernel version 4.19.0-13-amd64). Intel C compiler (version 18.0.1 20171018) was used. Intel Math Kernel Library (MKL) version 2018.0.1 Product Build 20171007 for Intel(R) 64 architecture was employed because of the same reason given earlier.

Unless explicitly stated otherwise, all of the experiments employed all cores in the computer and were extracted from a single run. When using routines of MKL's LAPACK, optimal block sizes determined by that software were employed. In a few experiments, in addition to MKL's LAPACK routines, we also assessed Netlib's LAPACK 3.4.0 routines. In this case, the NETLIB term is used. When using routines of Netlib's LAPACK, several block sizes were employed and best results were reported. For the purpose of a fair comparison, these routines from Netlib were linked to the BLAS library from MKL.

All matrices used in the experiments were randomly generated. Similar results for RANDUTV were obtained on other types of matrices, since one of the main advantages of the RANDUTV algorithm is that its performance does not depend on the matrix being factorized.

The following implementations were considered in the experiments:

- MKL SVD: The routine `dgesvd` from MKL LAPACK implementation was used to compute the Singular Value Decomposition.
- NETLIB SVD: Same as the previous one, but the code for computing the SVD from the reference Netlib LAPACK was employed.
- MKL SDD: The routine `dgesdd` from MKL was used to compute the Singular Value Decomposition. Unlike the previous SVD, this one uses the divide-and-conquer approach. This code is usually faster, but it requires a much larger auxiliary workspace when the orthonormal matrices are built.
- NETLIB SDD: Same as the previous one, but the code for computing the SVD with the divide-and-conquer approach from Netlibs LAPACK was employed.
- MKL CPQR: The routine `dgeqp3` from MKL's LAPACK was used to compute the column-pivoting QR factorization.
- RANDUTV_PBLAS (RANDUTV with parallel BLAS): This is the traditional implementation for computing the RANDUTV factorization that relies on the parallel BLAS to take advantage of all of the cores in the system. The parallel BLAS implementation from MKL was employed with these codes for the sake of a fair comparison. Our implementations were coded with `libflame` [38, 39] (release 11104).
- RANDUTV_AB (RANDUTV with algorithm-by-blocks): These are our new implementations for computing the RANDUTV factorization by scheduling all of the tasks to be computed in parallel and then executing them with serial MKL BLAS. Our implementations were also coded via `libflame`.
- MKL QR: The routine `dgeqrf` from MKL LAPACK implementation was used to compute the QR factorization. Although this routine does not reveal the rank, it was included in some experiments as a performance reference for the previous implementations.

For each experiment, two plots are shown:

- The left plot shows the performances when no orthonormal matrices are computed. In this case, just the singular values are computed for the SVD, just the upper triangular factor R is computed for the CPQR and the QR, and just the upper triangular factor T is computed for the RANDUTV. In the latter case, the algorithm of Figure 2 is invoked with arguments `build_U` and `build_V` set to false.
- The right plot shows the performance when all orthonormal matrices are explicitly formed in addition to the singular values (SVD), the upper triangular matrix R (CPQR), or the upper

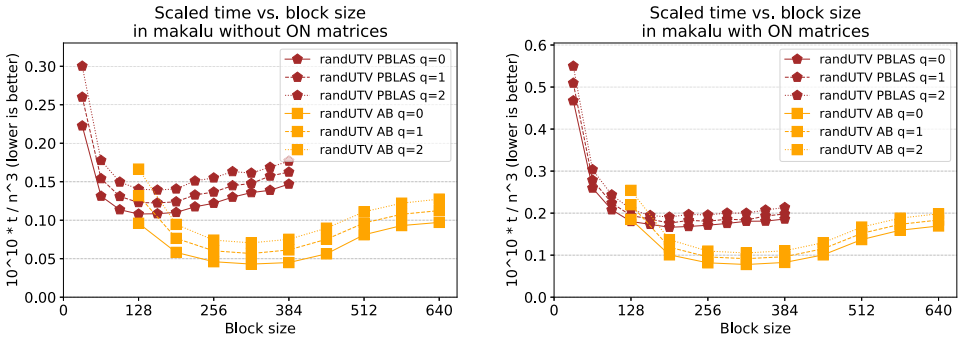


Fig. 7. Scaled time of randUTV implementations versus block size on matrices of dimension 20,000 × 20,000 in makalu.

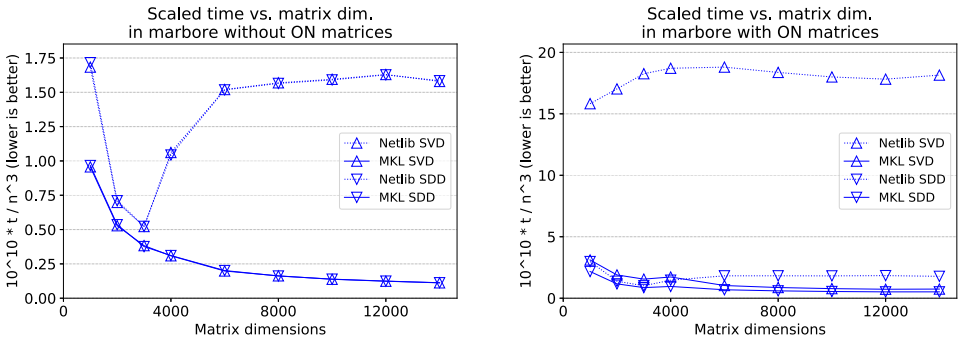


Fig. 8. Scaled time versus matrix dimension for SVD implementations for both Netlib and MKL libraries.

triangular matrix T (randUTV). In this case, matrices U and V are computed for the SVD, matrix Q is computed for the CPQR and the QR, and matrices U and V are computed for the randUTV. In the latter case, the algorithm of Figure 2 is called with both arguments `build_U` and `build_V` set to true. The right plot slightly favors CPQR and QR since only one orthonormal matrix is formed.

6.2.2 Block Size Impact in randUTV. Figure 7 shows the scaled computational times obtained by two implementations for computing the randUTV factorization (randUTV_PBLAS and the basic randUTV_AB `v01`) on several block sizes when processing matrices of dimension 20,000 × 20,000. The aim of these two plots is to determine the optimal block sizes and to illustrate the impact of a proper block size selection on the overall attained performance. The other factorizations (SVD and CPQR) are not shown since in those cases the optimal block sizes were determined by Intel's software. Optimal block sizes are about 160 and 320 for randUTV_PBLAS and randUTV_AB `v01`, respectively. This is a common observation on many task-parallel implementations, as small block sizes favor task-parallelism, whereas large block sizes usually reveal better per-task performance. As usual, those optimal sizes slightly depended on matrix dimensions: the larger the matrix dimensions, the slightly larger the block sizes.

6.2.3 Performance of State-of-the-Art SVD Factorizations. Figure 8 compares the scaled times of four implementations for computing the SVD factorization: MKL SVD, MKL SDD, Netlib SVD, and Netlib SDD. Performance is shown with respect to matrix dimensions. Block sizes similar to those

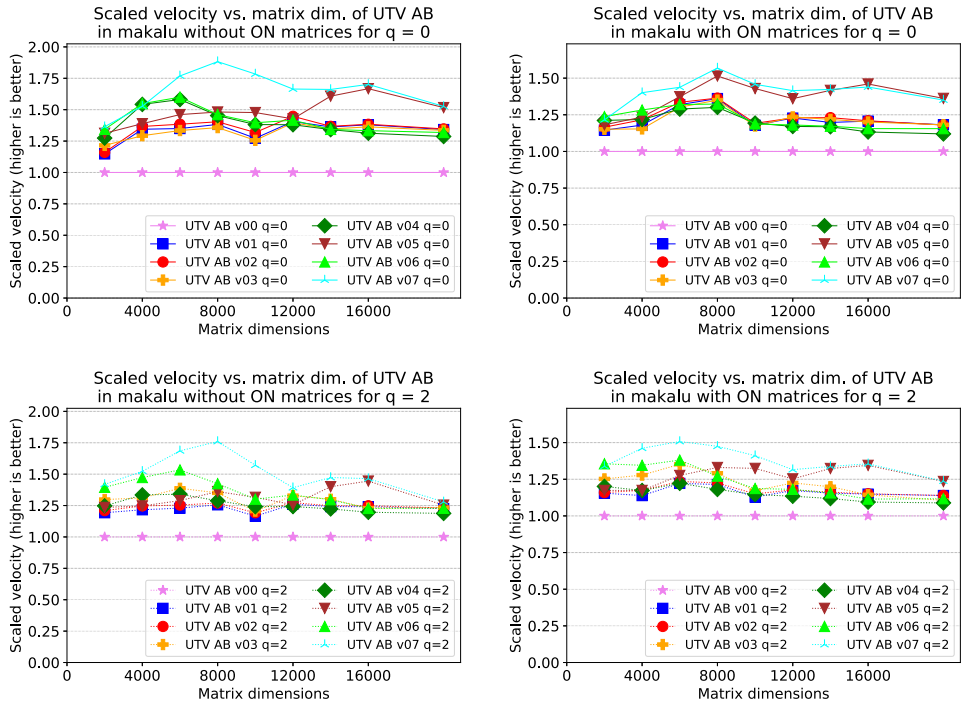


Fig. 9. Scaled velocity versus matrix dimension for the randUTV_AB implementations in `makalu`. In all plots, the performance of the `v00` variant is 1 for each matrix size. The first row shows results for $q = 0$, whereas the second row shows results for $q = 2$.

in the previous figure were used for Netlib's routines and the best results were reported. When no orthonormal matrices are computed, both the traditional SVD and the divide-and-conquer SVD render similar performance for this matrix type. In this case, MKL routines are up to about 14 times as fast as Netlib's routines. In addition, the performance of both Netlib's routines degrades drastically for matrices of dimension larger than $n = 3{,}000$. This is caused because the usual algorithm for computing the SVD (the one employed by Netlib) contains a higher ratio of BLAS-1 and BLAS-2 operations, typically memory bound, making the overall algorithm more sensitive to memory bandwidth. A matrix of dimension $n = 3{,}000$ requires about 69 MB, whereas the L3 cache size of `marbore` is 70 MB (2×35 MB per socket). A matrix of dimension $n = 4{,}000$ (when the performance drops) requires 122 MB, which is much larger than the cache of `marbore`. This same degradation happens in Netlib's CPQR and MKL CPQR (see next plots) since the underlying algorithms also perform only half of the flops in efficient BLAS-3 operations. When orthonormal matrices are computed, the traditional SVD is much slower than the divide-and-conquer SVD. In this case, the MKL SVD routine is up to 24.4 times as fast as the Netlib SVD, and the MKL SDD routine is up to 3.4 times as fast as the Netlib SDD. Note how MKL codes for computing the SVD are up to more than one order of magnitude faster than reference Netlib codes, showing the great performance achieved by the tuned Intel implementation.

6.2.4 Performance of randUTV Variants. Figure 9 compares the scaled performance of the proposed implementations of randUTV_AB in Section 4.2.3 with respect to matrix dimensions for $q = 0$ (top plots) and $q = 2$ (bottom plots). As usual, a range of block sizes was tested, and performance for the best block size was reported. The performance of the basic variant (`v00`) was chosen

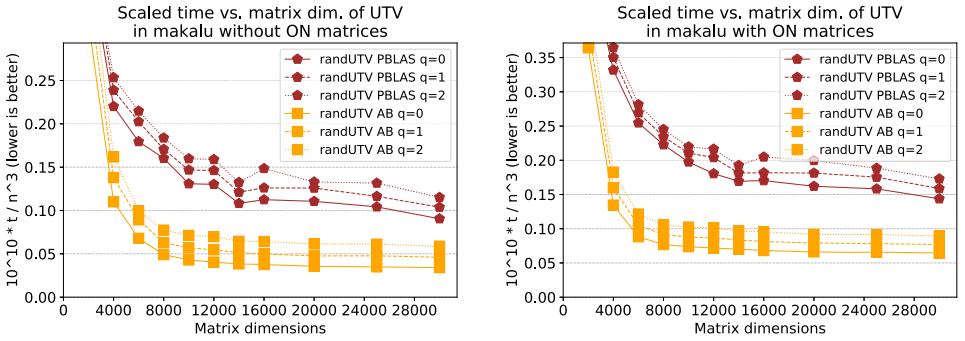


Fig. 10. Scaled time versus matrix dimension for randUTV implementations in makalu.

as a reference for each matrix size. In addition to the variants described in Section 4.2.3, two more variants are reported. The v06 variant simultaneously combines the methods in the v01, v02, v03, and v04 variants; the v07 variant simultaneously combines the methods in the v01, v02, v03, v04, and v05 variants. To avoid the overhead of the increased parallelism (the additional tasks) when there are already many tasks to process, both variants increase parallelism only when there are fewer row or column blocks to process than cores. As 40 cores were employed and the usual block size is about 320, this threshold was about 13,000.[1] Observe how the v01 variant clearly outperforms the basic v00 variant. In contrast, the v02 variant does not introduce significant performance improvements with respect to the v01 variant. On the other side, the v03 and v04 variants yield performance improvements of around 20% for small and medium matrices (when the dimension is smaller than about 12,000 or 14,000) with respect to the v01 variant. This improvement is smaller when orthonormal matrices are built, since there is more potential task parallelism under those circumstances. When the matrix dimension is more than 40 (the number of cores in the tested machine) times the optimal block size, there is no gain and there is a performance penalty from increasing the parallelism since there is plenty of parallelism, and the overhead to increase parallelism reduces performance. As can be seen, the performance improvement of the v06 and v07 variants is much larger when no orthonormal matrices are built. In those cases, it is usually between 30% and 50% faster; in some cases, it is up to 80% faster. Unless explicitly stated otherwise, the next experiments show results for the v07 variant.

Figure 10 compares the scaled times of both implementations of randUTV (randUTV_PBLAS and randUTV_AB) with respect to matrix dimensions. Again, only performance results for optimal block sizes are reported. When no orthonormal matrices are built, randUTV_AB is between 2.7 ($q = 0$) and 3.4 ($q = 2$) times as fast as randUTV_PBLAS for the largest matrix size; when orthonormal matrices are built, randUTV_AB is between 2.2 ($q = 0$) and 2.7 ($q = 2$) times as fast as randUTV_PBLAS for the largest matrix size.

6.2.5 Performance of randUTV versus State-of-the-Art SVD Factorizations. Figure 11 shows the scaled times of the best implementations with respect to matrix dimensions. In this case, randUTV_AB is usually faster when no orthonormal matrices are built, and this performance improvement is more evident when orthonormal matrices are built. Actually, the performance of randUTV_AB, MKL SVD, and MKL SDD are similar or even faster than MKL CPQR, a factorization that requires much fewer floating point operations.

[1]Note that the optimal block size also depends on the matrix size. Therefore, as the matrix size increases or decreases, the threshold could also be affected by a larger or smaller optimal block size.

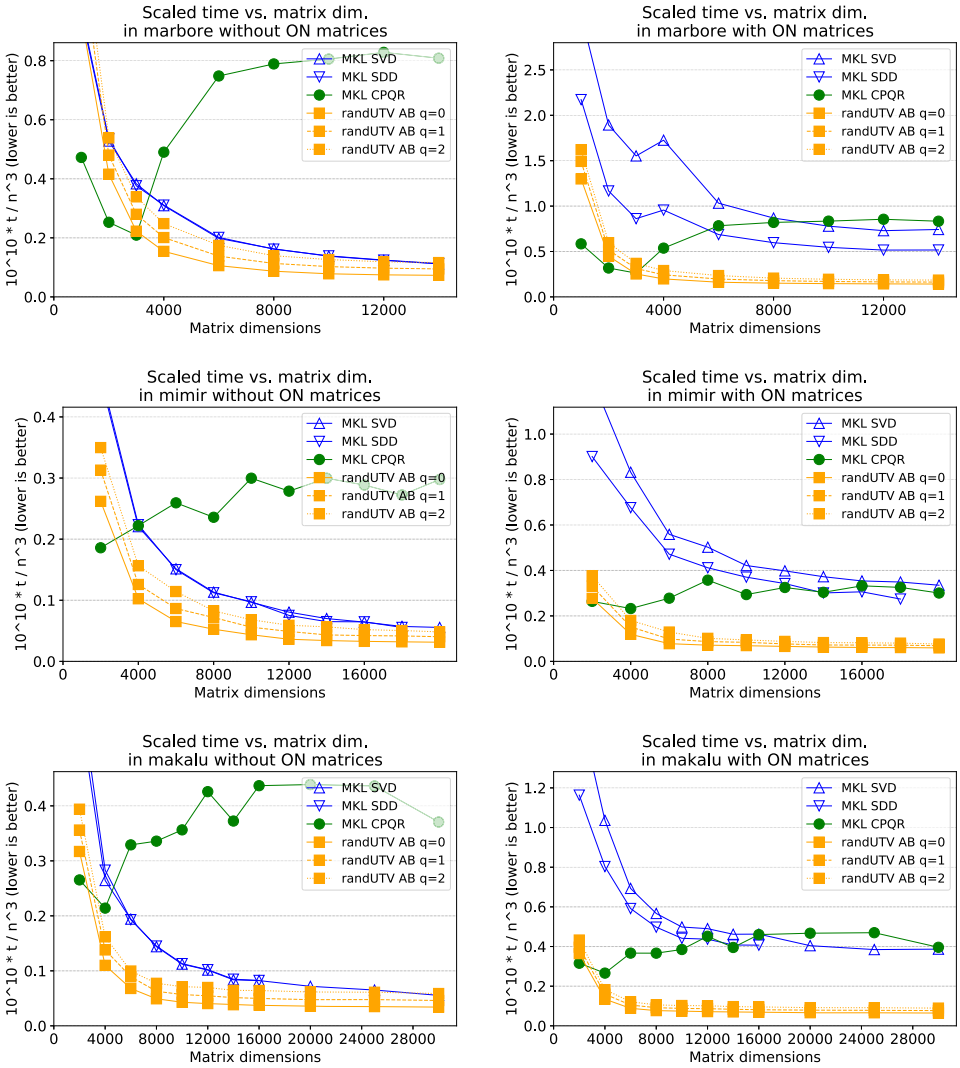


Fig. 11. Scaled time versus matrix dimension for the best implementations. The top row shows results for `marbore` with 28 cores, the middle row shows results for `mimir` with 36 cores, and the bottom row shows results for `makalu` with 40 cores.

Table 2 summarizes the ratios of the speed of RANDUTV_AB and the other rank-revealing factorizations normalized to its execution time on the largest matrix dimension with data shown in Figure 11. The performance improvement is significantly larger when orthonormal matrices are built and $q = 0$, but gains are still present when no orthonormal matrices are built and $q = 2$.

6.2.6 Scalability and Efficiency Analysis. Figure 12 shows the speedups obtained by the best implementations on the three machines. The top row shows results of `marbore` on matrices of dimension 14,000 × 14,000, the middle row shows results of `mimir` on matrices of dimension 18,000 × 18,000 (the largest dimension in which all the best implementations could be run), and the bottom row shows results of `makalu` on matrices of dimension 20,000 × 20,000. Recall that in this figure every implementation compares against itself on one core.

Table 2. Performance Ratios of randUTV_AB versus That of the Other Rank-Revealing Factorizations on the Largest Matrices in the Three Machines

		No orthonormal matrices			Orthonormal matrices		
		UTV vs.	UTV vs.	UTV vs.	UTV vs.	UTV vs.	UTV vs.
q	Machine	SDD	SVD	CPQR	SDD	SVD	CPQR
0	marbore	1.54	1.53	11.07	3.65	5.24	5.89
0	mimir	1.74	1.77	9.50	4.51	5.65	5.08
0	makalu	2.19	1.63	10.87	5.97	5.99	6.14
1	marbore	1.19	1.18	8.53	3.17	4.55	5.11
1	mimir	1.33	1.38	7.38	3.85	4.86	4.36
1	makalu	1.66	1.20	8.04	5.01	5.03	5.15
2	marbore	0.97	0.96	6.95	2.79	4.01	4.50
2	mimir	1.10	1.15	6.16	3.42	4.39	3.94
2	makalu	1.29	0.94	6.31	4.26	4.32	4.43

A value of 2.00 in a cell means randUTV_AB is twice as fast.

The scalability of randUTV_AB is always similar or even better than the scalability of the highly efficient unpivoted QR factorization, and it does not depend on whether the orthonormal matrices are built. Note that it always grows whenever more cores are employed. In contrast, the SVD factorizations perform very well in one case (even with a slight superspeedup): when orthonormal matrices are built using 18 cores or fewer in mimir. In all other cases, the speedups are usually modest when no orthonormal matrices are built, and the scalability even drops (the speedups do not grow much) when going from half the number of cores to the full number of cores. All in all, the scalability of randUTV_AB is similar (or even better) to that of the QR factorization and much higher than the rest of the implementations.

Another goal of this study was to measure the efficiency of the implementations. There are multiple definitions of efficiency in parallel environments. One of the most popular definitions is the speedup divided by the number of cores. According to this definition, the maximum efficiency is 1, which represents the fraction of the system that is doing useful work. Figure 13 shows the efficiency obtained by the best implementations on the three machines. The top row shows results of marbore on matrices of dimension $14{,}000 \times 14{,}000$, the middle row shows results of mimir on matrices of dimension $18{,}000 \times 18{,}000$ (the largest dimension in which all of the best implementations could be run), and the bottom row shows results of makalu on matrices of dimension $20{,}000 \times 20{,}000$. Recall that, also in this figure, every implementation compares against itself on one core. As can be seen, in all cases the efficiency of randUTV_AB is either the highest one or the second highest one. randUTV_AB is the only factorization that achieves an efficiency similar to or higher than 50% (the speedups are higher than half the number of cores) when employing the maximum number of cores in both architectures, whereas the efficiency of all of the other factorizations are usually lower. In contrast, the efficiency of SVD factorizations is usually lower except for mimir when orthonormal matrices are built. The efficiency of CPQR factorizations is even lower.

Another way of assessing the efficiency of the new randUTV_AB codes on shared-memory architectures is to compare their times with those of the QR factorization. Recall that the computational cost in flops of the randUTV algorithm for $q = 0$, $q = 1$, and $q = 2$ is 3, 4, and 5 times as large as the computational cost of the QR factorization, respectively, when no orthonormal matrices are built. When factorizing matrices of dimension 30,000 on makalu and no orthonormal matrices are built, the quotient of the times of the randUTV_AB for $q = 0$, $q = 1$, and $q = 2$ and the time of the QR factorization is 3.14, 4.24, 5.40, respectively. As can be seen, those figures are very close

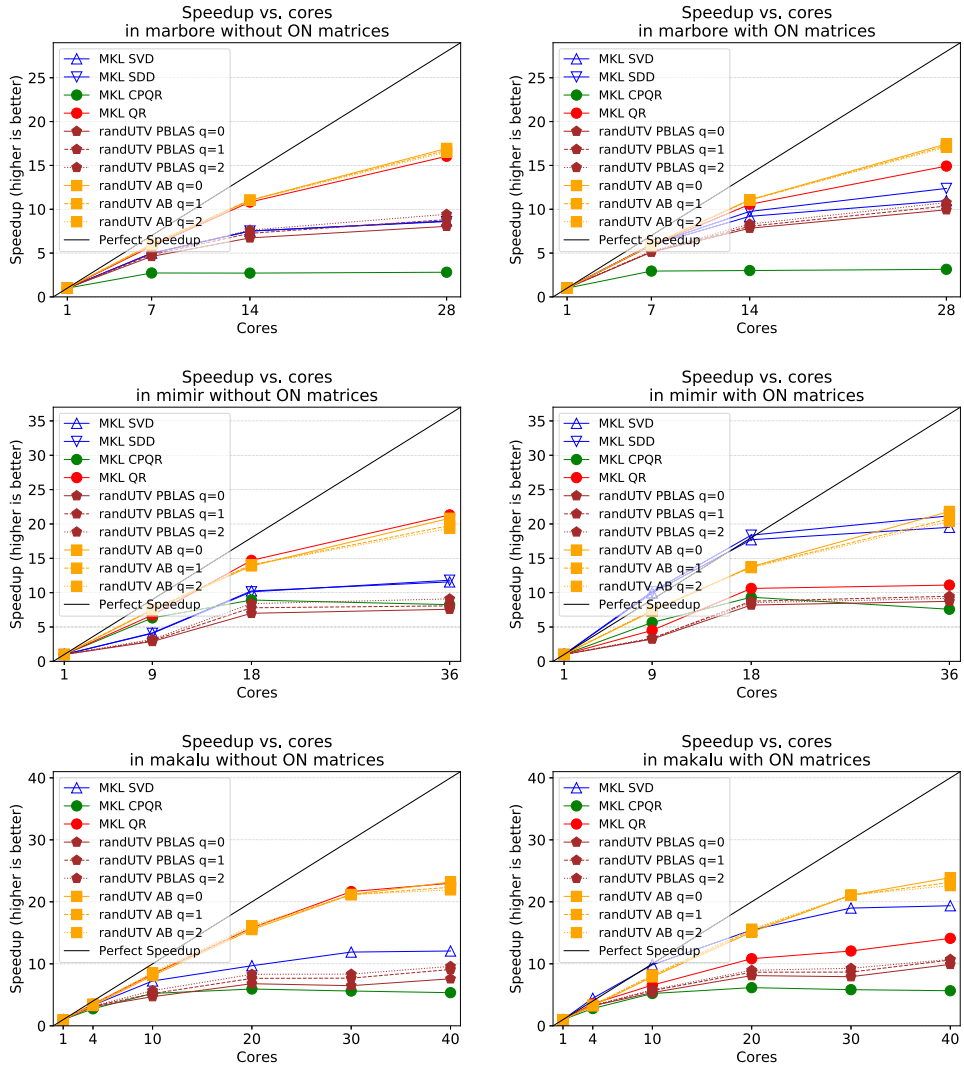


Fig. 12. Speedups versus number of cores for the best implementations. The top row shows results for `marbore` with up to 28 cores and $m = n = 14{,}000$, the middle row shows results for `mimir` with up to 36 cores and $m = n = 18{,}000$, and the bottom row shows results for `makalu` with up to 40 cores and $m = n = 20{,}000$.

to the theoretical quotients of 3, 4, and 5. Hence, for $q = 2$, the RANDUTV is only about 8% less efficient (5.40 vs. 5) than the QR factorization from the MKL library, which is well known to be very efficient.

6.2.7 Detailed Performance Analysis. Figure 14 reports task execution traces (obtained through code instrumentation by means of the `Extrae/Paraver` framework[2]) on `makalu`, for $n = 8{,}192$ and $b = 320$ running on 40 cores. Note that, for the purpose of our experiment, the maximum potential task parallelism in this case is around ($8{,}192/320 \approx 26$). This will ultimately

[2]https://tools.bsc.es/paraver.

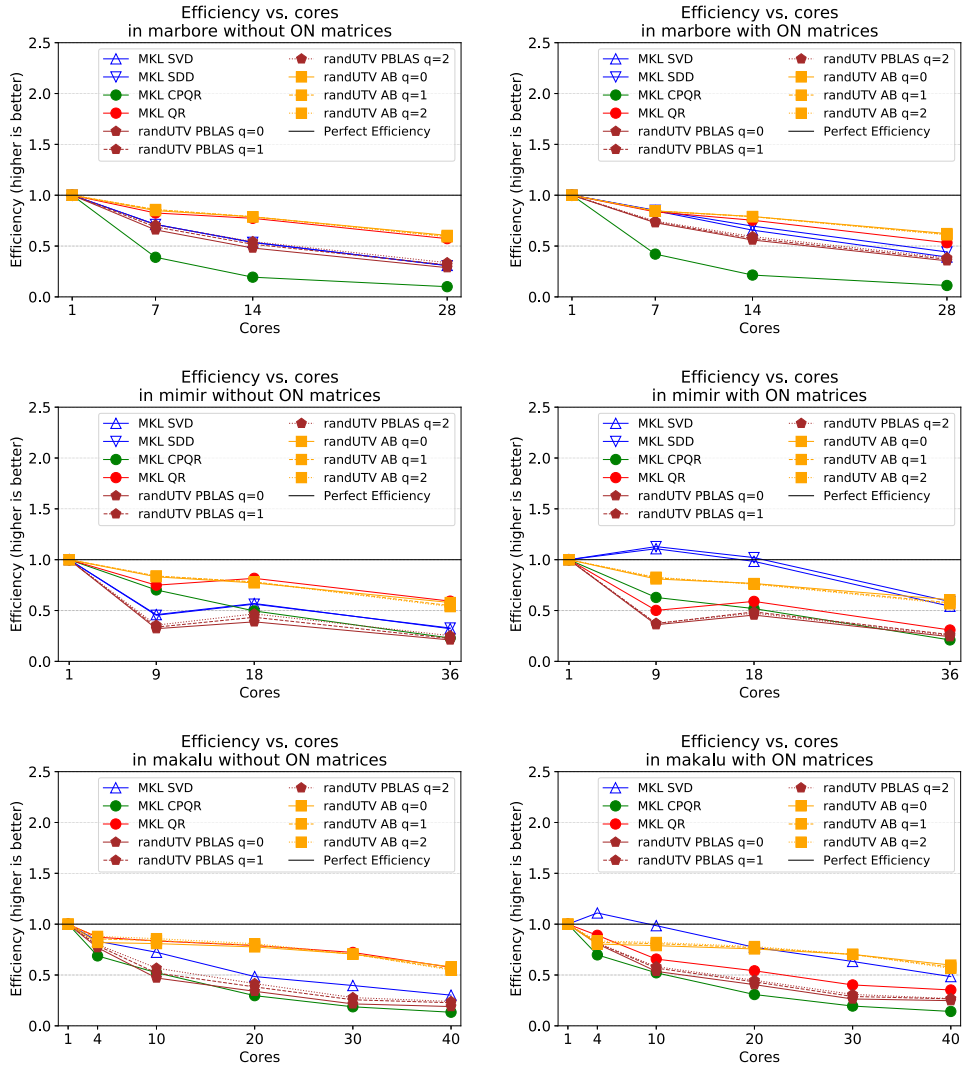


Fig. 13. Efficiency versus number of cores for the best implementations. The top row shows results for `marbore` with up to 28 cores and $m = n = 14{,}000$, the middle row shows results for `mimir` with up to 36 cores and $m = n = 18{,}000$, and the bottom row shows results for `makalu` with up to 40 cores and $m = n = 20{,}000$.

limit the performance of those variants that do not automatically increase the degree of task parallelism.

The experiment comprises four successive executions —`v05` and `v07` without and with construction of orthonormal matrices. The top traces illustrate the execution of tasks following the color convention in Figure 5; the plots in blue represent instantaneous core occupation. A number of caveats can be extracted from the observation of the traces:

- First, observe the considerable reduction in execution time from the `v07` implementations compared with their `v05` counterparts.

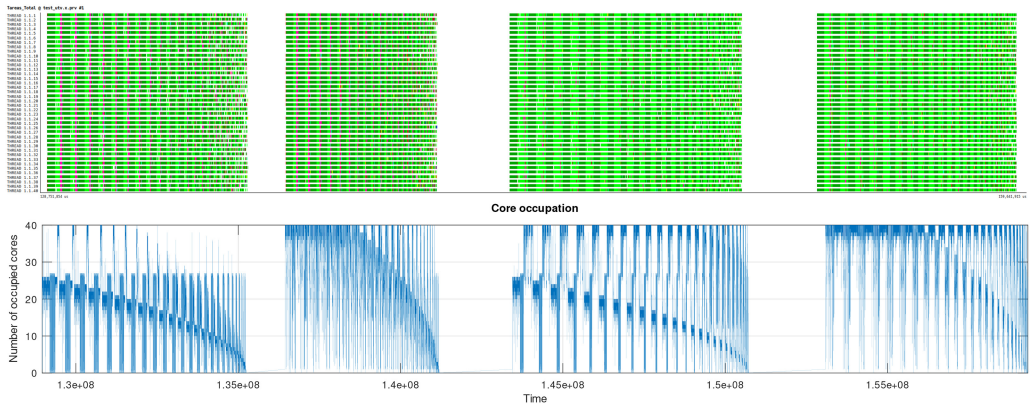


Fig. 14. Detailed execution traces for two different variants of the algorithm-by-blocks for randUTV with $n = 8{,}192$ and $b = 320$ on makalu, using 40 threads. From left to right: v05 without orthonormal matrices, v07 without orthonormal matrices, v05 with orthonormal matrices, and v07 with orthonormal matrices. The plot on the top shows the dataflow task execution, using the same color convention as that used in Figure 5; the plot on the bottom reports the instantaneous core occupation for each execution.

Table 3. Detailed Time Percentage Distribution Per Task Type for Variants v05 and v07 for randUTV, with $n = 8{,}192$ and $b = 320$ on makalu, using 40 Threads

v05 (no ON)	0.16	2.66	0.70	19.57	1.33	29.76	0.01	0.62	44.80	0.01	0.06	0.66
v07 (no ON)	0.29	2.52	1.27	17.70	2.25	26.63	0.02	0.56	48.43	0.01	0.04	0.65
v05 (ON)	0.13	1.79	0.45	12.11	2.45	53.23	0.02	0.39	29.22	0.00	0.04	0.51
v07 (ON)	0.17	1.58	0.79	11.22	4.22	49.42	0.01	0.36	32.01	0.01	0.03	0.43

Colors follow the convention in Figure 5.

- This improvement is more evident when orthonormal matrices are not computed, as parallelism in those cases is more scarce; hence, the improvements of our techniques to expose extra task parallelism are more evident.
- All in all, the primary source of the observed performance improvement, as suggested in previous sections, is the increase of task parallelism (and, hence, core occupation). This phenomenon can be observed in the occupation plots, which clearly report extra occupation throughout the whole computation in v07 compared with their v05 counterparts.

Figure 15 contains a histogram of core occupations for the same experiments as those reported in Figure 14. In the figure, low percentages mean that cores are idle most of the time. Note the qualitative and quantitative differences between v05 and v07: whereas the first present a peak core occupation around 26 (recall the potential task parallelism calculated earlier), variant v07 increases this potential parallelism and exhibits a large percentage of time (around 42%) using 40 cores in the case of computing orthonormal matrices (this percentage is also significantly larger when orthonormal matrices are not computed).

Finally, Table 3 lists the percentage of time devoted for each task type for the same round of experiments. Note how all implementations are heavily xgemm based, and how extra xgemm tasks are generated in all cases in variants v07 compared with their v05 counterpart, yielding higher performance.

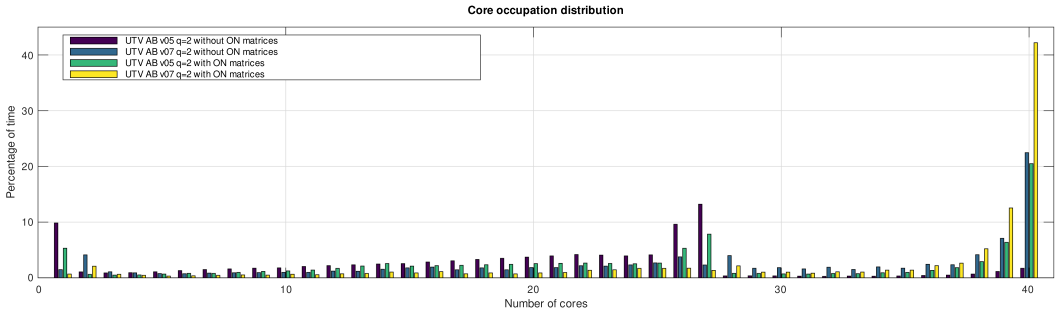


Fig. 15. Histogram of core occupation for two different variants of the algorithm-by-blocks for randUTV, with $n = 8{,}192$ and $b = 320$ on makalu, using 40 threads.. Low percentages mean that cores are idle most of the time.

6.2.8 Concluding Remarks for Shared-memory Architectures. In conclusion, randUTV_AB is the clear winner over competing factorization methods in terms of raw speed when orthonormal matrices are required and the matrix is not too small ($n \gtrsim 4{,}000$). In terms of scalability, randUTV_AB outperforms the competition as well. Also, the algorithm-by-blocks implementation gives noticeable speedup over the blocked PBLAS version. Although the accuracy of SVD is much higher, the fact that randUTV_AB can compete with MKL SVD at all in terms of speed is remarkable given the large effort usually invested by Intel on its software. This is evidenced by the fact that the MKL CPQR is left in the dust by both MKL SVD and randUTV_AB, each of which costs far more flops than MKL CPQR. The scalability results of randUTV_AB and its excellent timing evince its potential as a high-performance tool in shared-memory computing.

6.3 Computational Speed on Distributed-Memory Architectures

The experiments on distributed-memory architectures reported in this subsection were performed on two computers:

- ua: A cluster of HP computers. Each node of the cluster contained two Intel Xeon CPU X5560 processors at 2.8 GHz, with 12 cores and 48 GB of RAM in total. The nodes were connected with an Infiniband 4X QDR network. This network is capable of supporting 40 Gb/s signaling rate, with a peak data rate of 32 Gb/s in each direction.

  Its OS was GNU/Linux (version 3.10.0-514.21.1.el7.x86_64). Intel's `ifort` compiler (version 12.0.0 20101006) was employed. LAPACK and ScaLAPACK routines were taken from the Intel Math Kernel Library (MKL) version 10.3.0 Product Build 20100927 for Intel(R) 64 architecture, since this library usually delivers much higher performance than LAPACK and ScaLAPACK codes from the Netlib repository.
- skx: Nodes with Skylake architecture of the stampede2 supercomputer. Each node of the cluster contained two Intel Xeon CPU Platinum 8160 processors at 2.1 GHz, with 48 cores and 192 GB of RAM in total. The nodes were connected with a 100 Gb/s Intel Omni-Path (OPA) network with a fat tree topology.

  Its OS was GNU/Linux (version 3.10.0-957.5.1.el7.x86_64). Intel's `ifort` compiler (version 18.0.2) was employed. LAPACK and ScaLAPACK routines were also taken from the Intel Math Kernel Library (MKL) version 2018.0.2 Product Build 20180127 for Intel 64 architecture.

Most of the experiments were run in the first computer since in the second one the number of available hours was limited. Unless explicitly stated otherwise, the following plots show results from the first computer.

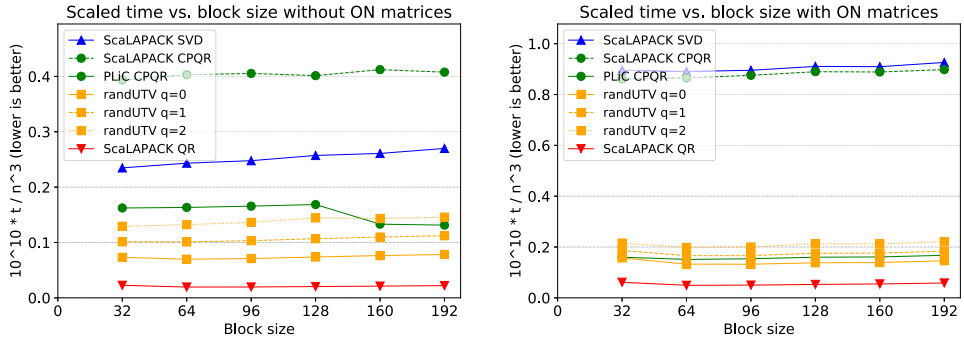


Fig. 16. Scaled time versus block size on 96 cores arranged as a 6×16 mesh.

All matrices used in these experiments were randomly generated since they can be generated much faster and the cluster was being heavily loaded by other users.

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

- ScaLAPACK SVD: The routine called `pdgesvd` from MKL's ScaLAPACK is used to compute the SVD.
- ScaLAPACK CPQR: The routine called `pdgeqpf` from MKL's ScaLAPACK is used to compute the CPQR factorization.
- PLiC CPQR: The routine called `pdgeqp3` from the PLiC library (Parallel Library for Control) [4] is used to compute the CPQR factorization by using BLAS-3. This source code was linked to the ScaLAPACK library from MKL for the purpose of a fair comparison.
- randUTV: A new implementation for computing the randUTV factorization based on the ScaLAPACK infrastructure and library. This source code was linked to the ScaLAPACK library from MKL for the purpose of a fair comparison.
- ScaLAPACK QR: The routine called `dgeqrf` from MKL's ScaLAPACK is used to compute the QR factorization. Although this routine does not reveal the rank, it was included in some experiments as a reference for the others.

Like in the previous subsection on shared-memory architectures, two plots are shown for every experiment. The left plot shows the performance when no orthonormal matrices are computed (the codes compute just the singular values for the SVD, the upper triangular matrix R for the CPQR and the QR factorizations, and the upper triangular matrix T for the randUTV factorization). The right plot shows the performance when, in addition to those, all orthonormal matrices are explicitly formed (matrices U and V for SVD and randUTV and matrix Q for QR and CPQR). Recall that the right plot slightly favors CPQR and QR since only one orthonormal matrix is built.

The first task was to assess the QR factorization and our implementation of the incremental QR factorization on distributed-memory machines. We assessed several nodes, several mesh configurations, block sizes, and matrix sizes. In all cases, the QR factorization in ScaLAPACK clearly outperformed our incremental QR factorization. For instance, when computing the QR factorization of matrices of dimension $25{,}600 \times 25{,}600$, the speed of the QR factorization in ScaLAPACK (`pdgeqrf`) was 370 GFlops/s, whereas the speed of the incremental QR factorization was 250 GFlops/s. Because of this performance gap, we did not implement a full randUTV based on the incremental QR factorization. The following plots report results and performance when employing the usual QR factorization.

Figure 16 shows the performance of all implementations described earlier on several block sizes when using 96 cores arranged as a 6×16 mesh on matrices of dimension $25{,}600 \times 25{,}600$. As can

Table 4. Ratios of the Speed of Distributed-Memory randUTV versus That of the Other Rank-Revealing Factorizations on the Largest Matrices in Some Configurations of the Two Machines

			No ON matrices		ON matrices	
			UTV vs.	UTV vs.	UTV vs.	UTV vs.
q	Machine	Mesh	SVD	CPQR	SVD	CPQR
0	ua	4×12	5.42	2.66	6.62	1.72
0	ua	6×16	3.37	1.88	6.73	1.14
0	skx	24×32	5.43	3.04	21.40	1.95
1	ua	4×12	3.69	1.81	5.23	1.36
1	ua	6×16	2.32	1.30	5.36	0.91
1	skx	24×32	4.36	2.44	18.41	1.68
2	ua	4×12	2.79	1.37	4.33	1.12
2	ua	6×16	1.82	1.02	4.49	0.76
2	skx	24×32	3.57	2.00	16.17	1.47

A value of 2.00 in a cell means randUTV is twice as fast. Only PLiC CPQR factorization is included since it is usually much faster than ScaLAPACK CPQR.

be seen, most implementations perform slightly better on small block sizes, such as 32 and 64, the only exception being PLiC CPQR, which performs a bit better on large block sizes when no orthonormal matrices are built.

Figure 17 shows the performance of all of the implementations for many topologies on matrices of dimension 20,480 × 20,480. The top row shows the results on one node (12 cores), the second row shows the results on two nodes (24 cores), the third row shows the results on four nodes (48 cores), and the fourth row shows the results on eight nodes (96 cores). As can be seen, the best topologies are usually $p \times q$, with p slightly smaller than q.

Figure 18 shows the performance versus matrix dimensions on two different numbers of cores: 48 cores arranged as 4×12 (top row) and 96 cores arranged as 6×16 (bottom row). As can be easily seen, randUTV is usually much faster than the other rank-revealing factorizations. As expected, randUTV is slower than the QR factorization; a non-rank-revealing factorization is included as a reference. On medium and large matrices, the performance of ScaLAPACK CPQR is much lower than that of randUTV, whereas the performance of PLiC CPQR is more similar to that of randUTV. Nevertheless, recall that the precision of CPQR is usually much smaller than that of randUTV.

Table 4 summarizes the ratios of the speed of distributed-memory randUTV and the other two best rank-revealing factorizations (the time of the other factorizations divided by the time of the randUTV) on the largest matrix dimension with data shown in Figure 18 and Figure 21. In this table, a value of 2.00 in a cell means that randUTV_AB is twice as fast. The performance gap is much larger when orthonormal matrices are built and $q = 0$; the gap is not so large when orthonormal matrices are built and $q = 2$.

In distributed-memory applications, the traditional approach creates one process per core. However, creating fewer processes and then a corresponding number of threads per process can improve performance in some cases. Obviously, the product of the number of processes and the number of threads per process must be equal to the total number of cores. The advantage of this approach is that the creation of fewer processes reduces the communication cost, which is usually the main bottleneck in distributed-memory applications. In the case of linear algebra

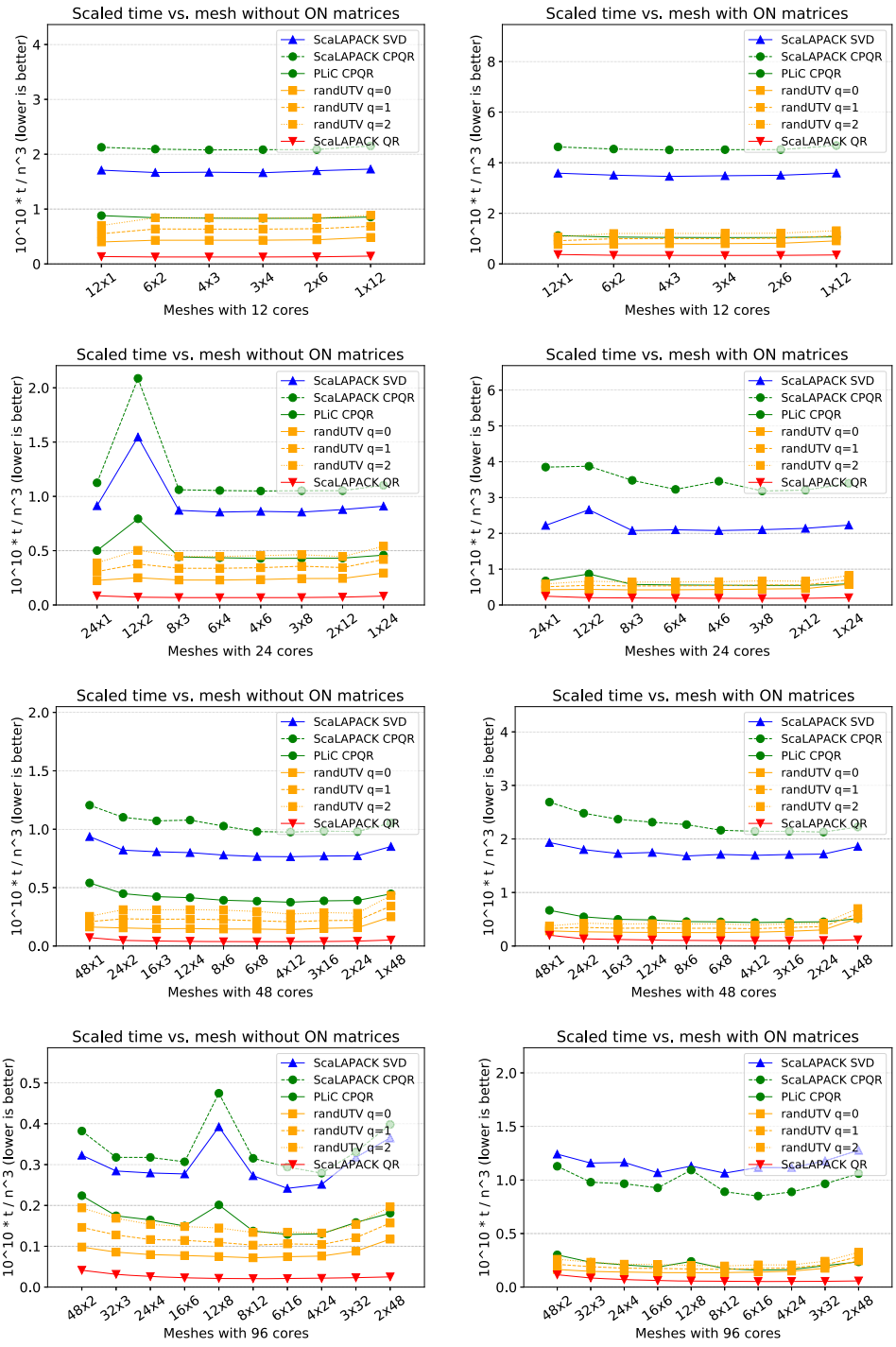


Fig. 17. Scaled times on several topologies on matrices of dimension 20,480 × 20,480.

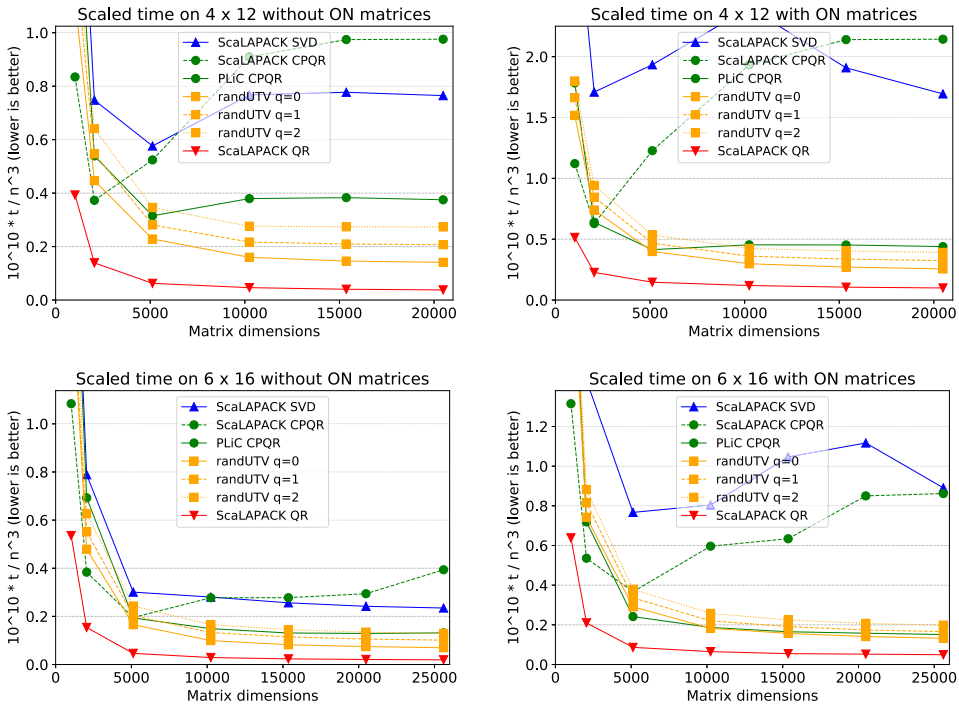


Fig. 18. Scaled time versus matrix dimension on two different numbers of cores. The top row shows results on 48 cores arranged as 4×12; the bottom row shows results on 96 cores arranged as 6×16.

applications, creating and using several threads per process can be easily achieved by employing shared-memory parallel LAPACK and BLAS libraries. Nevertheless, great care must be taken to ensure a proper pinning of processes to cores, since performance drops markedly otherwise. This was achieved by using the `-genv I_MPI_PIN_DOMAIN socket` flag when executing the `mpirun`/`mpiexec` command in the machine used in the experiments.

Figure 19 shows the scaled timings of the factorizations of matrices of dimension $25{,}600 \times 25{,}600$ on 96 cores when using several configurations with different numbers of threads per process. These plots include the results on a complete set of topologies to isolate the effect of the increased number of threads. As usual, the left three plots show performance when no orthonormal matrices are built and the right three plots show performance when orthonormal matrices are built. The top row shows performance when one process per core (96 processes) and then one thread per process are created ($96 \times 1 = 96$). The second row shows performance when one process per two cores (48 processes) and then two threads per process are created ($48 \times 2 = 96$). The third row shows performance when one process per three cores (32 processes) and then three threads per process are created ($32 \times 3 = 96$). As can be seen, the SVD increases performance only when orthonormal matrices are created, whereas RANDUTV increases performance in both cases (both with and without orthonormal matrices).

Table 5 shows the best timing (in seconds) for several topologies with 96 cores so that a finer detail comparison can be achieved. Matrices being factorized are $25{,}600 \times 25{,}600$. As can be seen, SVD increases performance 13% when orthonormal matrices are built, whereas RANDUTV with $q = 2$ improves performance 20% in both cases. Performance usually increases when using two threads per process but remains similar or drops when using more than two threads per process.

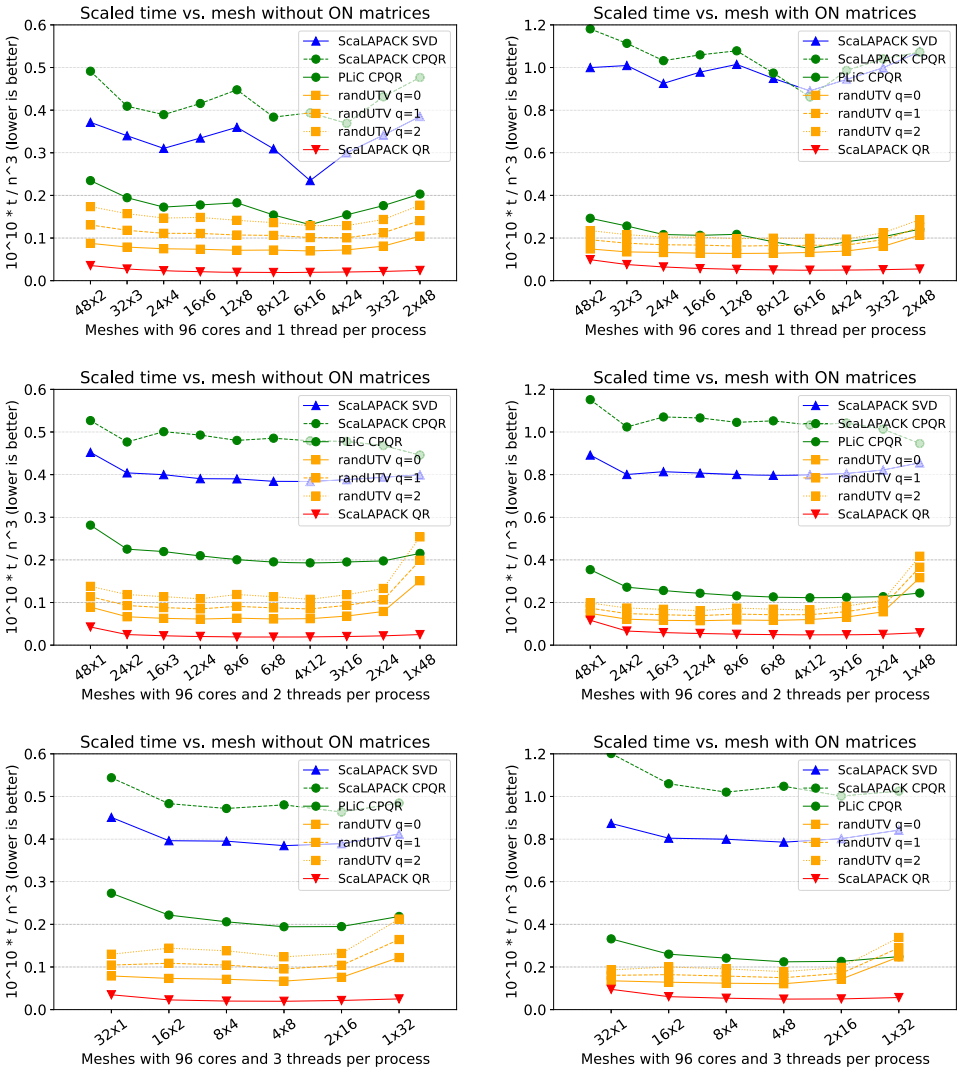


Fig. 19. Scaled times on several topologies on matrices of dimension 25,600 × 25,600.

Table 5. Best Timing in Seconds of Several Topologies with 96 Cores on Matrices of Dimension 25,600 × 25,600 Considering Several Numbers of Threads Per Process

	No ON matrices Threads per process			ON matrices Threads per process		
Factorization	1	2	3	1	2	3
SVD	393.9	644.0	645.3	1494.1	1336.4	1318.0
RANDUTV $q = 0$	117.0	102.7	112.2	214.2	192.3	203.7
RANDUTV $q = 1$	168.6	142.5	160.1	272.1	232.3	251.4
RANDUTV $q = 2$	216.7	180.4	207.8	327.5	271.7	298.7

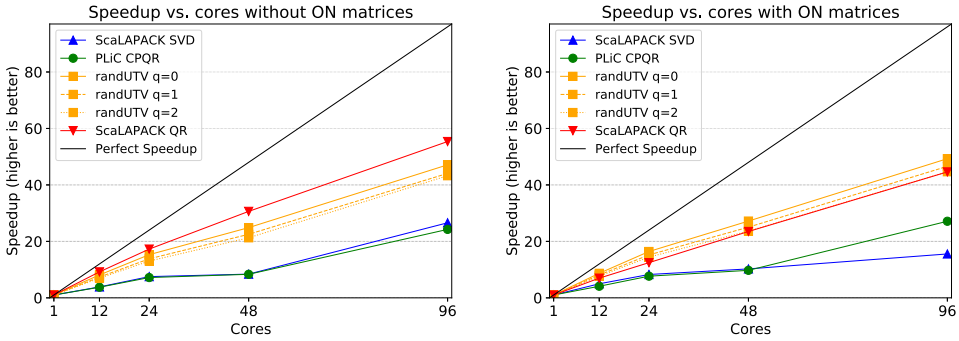


Fig. 20. Speedups versus number of cores for all of the implementations on matrices of dimension 20,480 × 20,480.

Figure 20 shows the speedups obtained by all of the implementations on matrices of dimension 20,480 × 20,480. Recall that in this plot, every implementation compares against itself on one core. The best topologies have been selected for the following number of cores: 3×4 for 12 cores, 6×4 for 24 cores, 4×12 for 48 cores, and 6×16 for 96 cores. When no orthonormal matrices are built, speedups of randUTV on the largest number of cores (96) are between 47.1 ($q = 0$) and 43.3 ($q = 2$). When orthonormal matrices are built, speedups of randUTV on the largest number of cores (96) are between 49.3 ($q = 0$) and 44.7 ($q = 2$). In both cases, the efficiency is close to 50%. When no orthonormal matrices are built, speedups of randUTV are a bit lower than those of QR factorization; when orthonormal matrices are built, speedups of randUTV are a bit higher than those of QR factorization. In both cases, the speedups of randUTV are much higher than those obtained by the SVD and the CPQR factorization, showing the great scalability potential of this factorization.

As previously done on shared-memory architectures, an additional way of assessing the efficiency of the new randUTV codes on distributed-memory architectures is to compare their times with the times of the QR factorization. As said before, when no orthonormal matrices are built, the computational cost in flops of the randUTV algorithm for $q = 0$, $q = 1$, and $q = 2$ is 3, 4, and 5 times as large as the computational cost of the QR factorization, respectively. When factorizing matrices of dimension 25,600 on 96 cores of ua organized as a 6×16 mesh and no orthonormal matrices are built, the quotient of the times of the randUTV for $q = 0$, $q = 1$, and $q = 2$ and the time of the QR factorization is 3.58, 5.18, and 6.62, respectively. On the other hand, when factorizing matrices of dimension 40,960 on 768 cores of skx organized as a 24×32 mesh and no orthonormal matrices are built, the quotient of the times of the randUTV for $q = 0$, $q = 1$, and $q = 2$ and the time of the QR factorization is 3.65, 4.55, and 5.56, respectively. As can be seen, those quotients are very close (closer in the second case) to the theoretical quotients of 3, 4, and 5. Hence, on skx for $q = 2$, the randUTV is only 11.8% less efficient than the QR factorization from the MKL ScaLAPACK library.

Figure 21 shows the performance versus matrix dimensions on 16 nodes of skx (768 cores in total arranged as a 24×32 logical mesh). The goal of this experiment was to assess a much larger number of cores than previous experiments as well as a more performant platform with more modern hardware and software. As can be seen, the performances of randUTV are much higher than those of ScaLAPACK codes. When orthonormal matrices are built, PLiC CPQR is much closer to randUTV, and the ScaLAPACK SVD shows a performance drop between matrix sizes 20,480 and 25,600.

In summary, randUTV is significantly faster than the available distributed-memory implementations of SVD. It also matches the best CPQR implementation tested. randUTV is known to reveal rank far better than CPQR [24]. It also furnishes orthonormal bases for the row space and for the

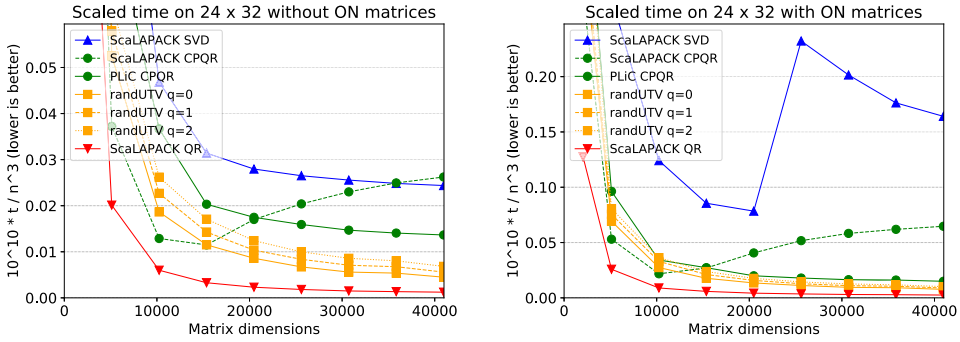


Fig. 21. Scaled times versus matrix dimensions on 768 cores (arranged as 24×32) of skx.

(numerical) null space of the matrix. This means that just matching the speed of CPQR represents a major gain in information at no additional computational cost. Furthermore, RANDUTV is faster than even CPQR in the case that orthonormal matrices are required. We finally observe that the potential for scalability of RANDUTV is a clear step above competing implementations for rank-revealing factorizations in distributed memory.

7 CONCLUSIONS

We have described two new implementations of the RANDUTV algorithm for computing a rank revealing factorization matrix, targeting shared-memory and distributed-memory architectures, respectively.

Regarding shared memory, the new implementation proposes an *algorithm-by-blocks* that, built on top of a runtime task scheduler (`libflame`'s SuperMatrix) implements a dataflow execution model. Based on a DAG, this model reduces the amount of synchronization points, increasing performance on massively parallel architectures. Actually, performance results on up to 40 cores reveal excellent performance and scalability results compared with state-of-the-art proprietary libraries.

We have also proposed a distributed-memory algorithm for RANDUTV. This proposal leverages the classic blocked algorithm rather than the algorithm-by-blocks, and makes heavy use of ScaLAPACK. Performance results show competitive performance and excellent scalability compared with alternative state-of-the-art implementations.

In this article, we focused exclusively on the case of multicore CPUs with shared-memory and homogeneous distributed-memory architectures. We expect that the relative advantages of RANDUTV will be even more pronounced in more severely communication-constrained environments, such as GPU-based architectures (composed of one or many nodes). Variations of the method modified for such environments is proposed as future work.

ACKNOWLEDGMENTS

The authors would like to thank Javier Navarrete (Universitat d'Alacant) for granting access to the distributed-memory server ua.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide (3rd ed.).* SIAM, Philadelphia, PA.

[2] Ed Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, Bernard Tourancheau, and Robert van de Geijn. 1991. Basic linear algebra communication subprograms. In *Proceedings of 6th Distributed Memory Computing Conference.* IEEE, 287–290.

[3] Jesse L. Barlow. 2002. Modification and maintenance of ULV decompositions. In *Applied Mathematics and Scientific Computing*. Springer, 31–62.

[4] Peter Benner, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2008. Solving linear-quadratic optimal control problems on parallel computers. *Optimization Methods and Software* 23, 6 (2008), 879–909. https://doi.org/10.1080/10556780802058721

[5] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. Geijn. 2005. Representing linear algebra algorithms in code: The FLAME application program interfaces. *ACM Transactions on Mathematical Software* 31, 1 (2005), 27–59.

[6] L. Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. 1997. *ScaLAPACK Users' Guide*. SIAM.

[7] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert Van De Geijn. 2007. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 116–125.

[8] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. 2008. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*. ACM, New York, NY, 123–132. https://doi.org/10.1145/1345206.1345227

[9] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R. Clinton Whaley. 1996. ScaLAPACK: A portable linear algebra library for distributed memory computers? Design issues and performance. *Computer Physics Communications* 97, 1–2 (1996), 1–15.

[10] Jaeyoung Choi, Jack J. Dongarra, L. Susan Ostrouchov, Antoine P. Petitet, David W. Walker, and R. Clint Whaley. 1996. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming* 5, 3 (1996), 173–184.

[11] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David W. Walker. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *4th Symposium on the Frontiers of Massively Parallel Computation*. IEEE, 120–127.

[12] Jack J. Dongarra, Jermey Du Cruz, Sven Hammarling, and Iain S. Duff. 1990. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software* 16, 1 (1990), 18–28.

[13] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. 1988. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software* 14, 1 (1988), 18–32.

[14] Carl Eckart and Gale Young. 1936. The approximation of one matrix by another of lower rank. *Psychometrika* 1, 3 (1936), 211–218.

[15] Hasan Erbay, Jesse L. Barlow, and Zhenyue Zhang. 2002. A modified Gram–Schmidt-based downdating technique for ULV decompositions with applications to recursive TLS problems. *Computational Statistics & Data Analysis* 41, 1 (2002), 195–209.

[16] A. M. Vidal and G. Quintana. 1992. Parallel algorithms for QR factorization on shared memory multiprocessors. In *Proceedings of the European Workshop on Parallel Computing'92, Parallel Computing: From Theory to Sound Practice*, E. Milgrom W. Joosen (Ed.). IOS Press, 72–75.

[17] Gene H. Golub and Charles F. Van Loan. 1996. *Matrix Computations* (3rd ed.). Johns Hopkins University Press, Baltimore, MD.

[18] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. Van De Geijn. 2001. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software* 27, 4 (2001), 422–455.

[19] Brian C. Gunter and Robert A. Van De Geijn. 2005. Parallel out-of-core computation and updating of the QR factorization. *ACM Transactions on Mathematical Software* 31, 1 (2005), 60–78.

[20] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. 2011. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review* 53, 2 (2011), 217–288.

[21] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert A. Van De Geijn, and Field G. Van Zee. 2012. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing* 72, 9 (2012), 1134–1143.

[22] A. M. Vidal, J. M. Badía, and G. Quintana. 1994. Efficient parallel QR decomposition on a network of transputers. In *Proceedings of Transputers'94, Advanced Research and Industrial Applications*. IOS Press, 247–265.

[23] Chuck L. Lawson, Richard J. Hanson, David R. Kincaid, and Fred T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software* 5, 3 (1979), 308–323.

[24] P. G. Martinsson, G. Quintana-Ortí, and N. Heavner. 2019. RandUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization. *ACM Transactions on Mathematical Software* 45, 1, Article Article 4 (March 2019), 26 pages. https://doi.org/10.1145/3242670

[25] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. 2006. *A Randomized Algorithm for the Approximation of Matrices*. Technical Report. Yale CS Research Report YALEU/DCS/RR-1361. Yale University, Computer Science Department, New Haven, CT.

[26] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. 2011. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis* 30, 1 (2011), 47–68. https://doi.org/10.1016/j.acha.2010.02.003

[27] Per-Gunnar Martinsson and Joel Tropp. 2020. Randomized numerical linear algebra: foundations & algorithms. (2020). arXiv:math.NA/2002.01387.

[28] Leon Mirsky. 1960. Symmetric gauge functions and unitarily invariant norms. *The Quarterly Journal of Mathematics* 11, 1 (1960), 50–59.

[29] Haesun Park and Lars Eldén. 1995. Downdating the rank-revealing URV decomposition. *SIAM SIAM Journal on Matrix Analysis and Applications* 16, 1 (1995), 138–155.

[30] Gregorio Quintana-Ortí, Francisco D. Igual, Mercedes Marqués, Enrique S. Quintana-Ortí, and Robert A. Van de Geijn. 2012. A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures. *ACM Transactions on Mathematical Software* 38, 4 (2012), 25.

[31] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. 2009. Programming matrix algorithms-by-blocks for thread-level parallelism. *Transactions on Mathematical Software* 36, 3, Article 14 (July 2009), 26 pages. https://doi.org/10.1145/1527286.1527288

[32] Vladimir Rokhlin, Arthur Szlam, and Mark Tygert. 2009. A randomized algorithm for principal component analysis. *SIAM Journal on Matrix Analysis and Applications* 31, 3 (2009), 1100–1124.

[33] Robert Schreiber and Charles Van Loan. 1989. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing* 10, 1 (1989), 53–57.

[34] G. W. Stewart. 1998. *Matrix Algorithms Volume 1: Basic Decompositions*. SIAM, Philadelphia, PA.

[35] G. W. Stewart. 1992. An updating algorithm for subspace tracking. *IEEE Transactions on Signal Processing* 40, 6 (Jun 1992), 1535–1541. https://doi.org/10.1109/78.139256

[36] Gilbert W. Stewart. 1993. Updating a rank-revealing ULV decomposition. *SIAM Journal on Matrix Analysis and Applications* 14, 2 (1993), 494–499.

[37] Lloyd N. Trefethen and David Bau III. 1997. *Numerical Linear Algebra*, Volume 50. SIAM, Philadelphia, PA.

[38] Field G. Van Zee. 2012. `libflame: The Complete Reference`. `www.lulu.com`. Retrieved April 6, 2022 from http://www.cs.utexas.edu/users/flame/web/FLAMEPublications.html.

[39] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. 2009. The libflame library for dense matrix computations. *IEEE Computation in Science & Engineering* 11, 6 (2009), 56–62.

Received April 2020; revised December 2021; accepted December 2021