Designing LU-QR hybrid solvers for performance and stability

Mathieu Faverge[1], Julien Herrmann[2], Julien Langou[3*],
Bradley R. Lowery[3*], Yves Robert[2,4†] and Jack Dongarra[4]
1. *Laboratoire LaBRI, IPB ENSEIRB-MatMeca, Bordeaux, France*
2. *Laboratoire LIP, École Normale Supérieure de Lyon, France*
3. *University Colorado Denver, USA*
4. *University of Tennessee Knoxville, USA*

***Abstract*—This paper introduces hybrid LU-QR algorithms for solving dense linear systems of the form $Ax = b$. Throughout a matrix factorization, these algorithms dynamically alternate LU with local pivoting and QR elimination steps, based upon some robustness criterion. LU elimination steps can be very efficiently parallelized, and are twice as cheap in terms of operations, as QR steps. However, LU steps are not necessarily stable, while QR steps are always stable. The hybrid algorithms execute a QR step when a robustness criterion detects some risk for instability, and they execute an LU step otherwise. Ideally, the choice between LU and QR steps must have a small computational overhead and must provide a satisfactory level of stability with as few QR steps as possible. In this paper, we introduce several robustness criteria and we establish upper bounds on the growth factor of the norm of the updated matrix incurred by each of these criteria. In addition, we describe the implementation of the hybrid algorithms through an extension of the PaRSEC software to allow for dynamic choices during execution. Finally, we analyze both stability and performance results compared to state-of-the-art linear solvers on parallel distributed multicore platforms.**

I. Introduction

Consider a dense linear system $Ax = b$ to solve, where A is a square tiled-matrix, with n tiles per row or column. Each tile is a block of n_b-by-n_b elements, so that the actual size of A is $N = n \times n_b$. Here, n_b is a parameter tuned to squeeze the most out of arithmetic units and memory hierarchy. To solve the linear system $Ax = b$, with A a general matrix, one usually applies a series of transformations, pre-multiplying A by several elementary matrices. There are two main approaches: *LU factorization*, where one uses lower unit triangular matrices, and *QR factorization*, where one uses orthogonal Householder matrices. To the best of our knowledge, this paper is the first study to propose a mix of both approaches during a single factorization. The LU factorization update is based upon matrix-matrix multiplications, a kernel that can be very efficiently parallelized, and whose library implementations typically achieve close to peak CPU performance. Unfortunately, the efficiency of LU factorization is hindered by the need to perform partial pivoting at each step of the algorithm, to ensure numerical stability. On the contrary, the QR factorization is always stable, but requires twice as many operations, and a more complicated update step that is not as parallel as a matrix-matrix product. Tiled QR factorizations [1], [2] greatly improve the parallelism of the update step since they involve no pivoting but are based upon more complicated kernels whose library implementations requires twice as many operations as LU.

The main objective of this paper is to explore the design of *hybrid* algorithms that would combine the low cost and high CPU efficiency of the LU factorization, while retaining the numerical stability of the QR approach. The main idea is the following: at each step of the elimination, we perform a *robustness* test to know if the diagonal tile can be stably used to eliminate the tiles beneath it using an LU step. If the test succeeds, then go for an elimination step based upon LU kernels, without any further pivoting involving sub-diagonal tiles in the panel. Otherwise, if the test fails, then go for a step with QR kernels. On the one extreme, if all tests succeed throughout the algorithm, we implement an LU factorization without pivoting.On the other extreme, if all tests fail, we implement a QR factorization. On the average, only a fraction of the tests will fail. If this fraction remains small enough, we will reach a CPU performance close to that of LU without pivoting. Of course the challenge is to design a test that is accurate enough (and not too costly) so that LU kernels are applied only when it is numerically safe to do so.

Implementing such a hybrid algorithm on a state-of-the-art distributed-memory platform, whose nodes are themselves equipped with multiple cores, is a programming challenge. Within a node, the architecture is a shared-memory machine, running many parallel threads on the cores. But the global architecture is a distributed-memory machine, and requires MPI communication primitives for inter-node communications. We rely on the PaRSEC software [3], so that we can concentrate on the algorithm and forget about MPI and threads. Once we have specified the algorithm at a task level, the PaRSEC software will recognize which operations are local to a node (and hence correspond to shared-memory accesses), and which are not (and hence must be converted

*The research of this author was fully supported by the National Science Foundation grant # NSF CCF 1054864.

†The research of this author was supported in part by the French ANR *Rescue* project and by the Department of Energy # DOE DE-SC0010682.

into MPI communications). Previous experiments show that this approach is very powerful, and that the use of a higher-level framework does not prevent our algorithms from achieving the same performance as state-of-the-art library releases [4].

However, implementing a hybrid algorithm requires the programmer to implement a *dynamic* task graph of the computation. Indeed, the task graph of the hybrid factorization algorithm is no longer known statically (contrarily to a standard LU or QR factorization). At each step of the elimination, we use either LU-based or QR-based tasks, but not both. This requires the algorithm to dynamically fork upon the outcome of the robustness test, in order to apply the selected kernels. The solution is to prepare a graph that includes both types of tasks, namely LU and QR kernels, to select the adequate tasks on the fly, and to discard the useless ones. We have to join both potential execution flows at the end of each step, symmetrically. Most of this mechanism is transparent to the user. We discuss this extension of PaRSEC in more detail in Section IV.

The major contributions of this paper are the following: (i) the introduction of new LU-QR hybrid algorithms; (ii) the design of several robustness criteria, with bounds on the induced growth factor; (iii) a comprehensive experimental evaluation of the best trade-offs between performance and numerical stability; and (iv) he extension of PaRSEC to deal with dynamic task graphs. The rest of the paper is organized as follows. First we explain the main principles of LU-QR hybrid algorithms in Section II. Then we describe robustness criteria in Section III. Next we detail the implementation within the PaRSEC framework in Section IV. We report experimental results in Section V. We discuss related work in Section VI. Finally, we provide concluding remarks and future directions in Section VII.

II. Hybrid LU-QR algorithms

In this section, we describe hybrid algorithms to solve a dense linear system $Ax = b$, where $A = (A_{ij})_{(i,j)\in[\![1..n]\!]^2}$ is a square tiled-matrix, with n tiles per row or column. Each tile is a block of n_b-by-n_b elements, so that A is of order $N = n \times n_b$.

The common goal of a classical one-sided factorization (LU or QR) is to *triangularize* the matrix A through a succession of elementary transformations. Consider the first step of such an algorithm. We partition A by block such that $A = \begin{pmatrix} A_{11} & C \\ B & D \end{pmatrix}$. In terms of tile, A_{11} is 1-by-1, B is $(n-1)$-by-1, C is 1-by-$(n-1)$, and D is $(n-1)$-by-$(n-1)$. The first block-column $\begin{pmatrix} A_{11} \\ B \end{pmatrix}$ is the *panel* of the current step.

Traditional algorithms (LU or QR) perform the same type of transformation at each step. The key observation of this paper is that any type of transformation (LU or QR) can be used for a given step independently of what was used for the previous steps. The common framework of a step is the following:

$$\begin{pmatrix} A_{11} & C \\ B & D \end{pmatrix} \Leftrightarrow \begin{pmatrix} factor & apply \\ eliminate & update \end{pmatrix} \Leftrightarrow \begin{pmatrix} U_{11} & C' \\ 0 & D' \end{pmatrix}. \quad (1)$$

First, A_{11} is *factored* and transformed in the upper triangular matrix U_{11}. Then, the transformation of the factorization of A_{11} is *applied* to C. Then A_{11} is used to *eliminate* B. Finally D is accordingly *updated*. Recursively factoring D' with the same framework will complete the factorization to an upper triangular matrix. For each step, we have a choice for an LU step or a QR step. The operation count for each kernel is given in Table I.

	LU step, var A1		QR step	
factor A	$2/3$	GETRF	$4/3$	GEQRT
eliminate B	$(n-1)$	TRSM	$2(n-1)$	TSQRT
apply C	$(n-1)$	TRSM	$2(n-1)$	TSMQR
update D	$2(n-1)^2$	GEMM	$4(n-1)^2$	UNMQR

Table I: Computational cost of each kernel. The unit is n_b^3 operations.

Generally speaking, QR transformations are twice as costly as their LU counterparts. The bulk of the computations take place in the update of the trailing matrix D. This obviously favors LU *update* kernels. In addition, the LU *update* kernels are fully parallel and can be applied independently on the $(n-1)^2$ trailing tiles. Unfortunately, LU updates (using GEMM) are stable only when $\|A_{11}^{-1}\|^{-1}$ is larger than $\|B\|$ (see Section III). If this is not the case, we have to resort to QR kernels. Not only are these are twice as costly, but they also suffer from enforcing more dependencies: all columns can still be processed (*apply* and *update* kernels) independently, but inside a column, the kernels must be applied in sequence.

The hybrid *LU-QR Algorithm* uses the standard 2D block-cyclic distribution of tiles along a virtual p-by-q cluster grid. The 2D block-cyclic distribution nicely balances the load across resources for both LU and QR steps. Thus at step k of the factorization, the panel is split into p *domains* of approximately $\frac{n-k+1}{p}$ tile rows. Domains will be associated with physical memory regions, typically a domain per node in a distributed memory platform. Thus an important design goal is to minimize the number of communications across domains, because these correspond to nonlocal communications between nodes. At each step k of the factorization, the domain of the node owning the diagonal tile A_{kk} is called the *diagonal domain*.

The hybrid *LU-QR Algorithm* applies LU kernels when it is numerically safe to do so, and QR kernels otherwise. Coming back to the first elimination step, the sequence of operations is described in Algorithm 1.

Several variants (QR factorization of the diagonal, block

Algorithm 1: Hybrid LU-QR algorithm

for $k = 1$ *to* n **do**
  Factor: Compute a factorization of the diagonal tile: either with LU partial pivoting or QR;
  Check: Compute some robustness criteria (see Section III) involving only tiles $A_{i,k}$, where $k \leq i \leq n$, in the elimination panel;
  Apply, Eliminate, Update:
  if *the criterion succeeds* **then**
    Perform an LU step;
  else
    Perform a QR step;

Algorithm 2: Step k of an LU step - var (A1)

Factor: $A_{kk} \leftarrow GETRF(A_{kk})$;
for $i = k + 1$ *to* n **do**
  Eliminate: $A_{ik} \leftarrow TRSM(A_{kk}, A_{ik})$;
for $j = k + 1$ *to* n **do**
  Apply: $A_{kj} \leftarrow SWPTRSM(A_{kk}, A_{kj})$;
for $i = k + 1$ *to* n **do**
  for $j = k + 1$ *to* n **do**
    Update: $A_{ij} \leftarrow GEMM(A_{ik}, A_{kj}, A_{ij})$;

LU factorization) are described in the companion technical report [5].

A. LU step

We assume that the criterion validates an LU step (see Section III). We describe the variant (A1) of an LU step given in Algorithm 2. The kernels for the LU step are the following:

- *Factor:* $A_{kk} \leftarrow GETRF(A_{kk})$ is an LU factorization with partial pivoting: $P_{kk}A_{kk} = L_{kk}U_{kk}$, the output matrices L_{kk} and U_{kk} are stored in place of the input A_{kk}.
- *Eliminate:* $A_{ik} \leftarrow TRSM(A_{kk}, A_{ik})$ solves in-place, the upper triangular system $A_{ik} \leftarrow A_{ik}U_{kk}^{-1}$, with U_{kk} stored in the upper part of A_{kk}.
- *Apply:* $A_{kj} \leftarrow SWPTRSM(A_{kk}, A_{ik})$ solves the unit lower triangular system $A_{kj} \leftarrow L_{kk}^{-1}P_{kk}A_{kj}$, with L_{kk} stored in the (strictly) lower part of A_{kk}.
- *Update:* $A_{ij} \leftarrow GEMM(A_{ik}, A_{kj}, A_{ij})$ is a general matrix product $A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$.

In terms of parallelism, the factorization of the diagonal tile is followed by the $TRSM$ kernels that can be processed in parallel, then every $GEMM$ kernel can be processed concurrently. These highly parallelizable updates constitute one of the two main advantages of the LU step over the QR step. The second main advantage is halving the number of operations.

During the *factor* step, one variant is to factor the whole diagonal domain instead of only factoring the diagonal tile.

Algorithm 3: Step k of the HQR factorization

for $i = k + 1$ *to* n **do**
  $elim(i, eliminator(i, k), k)$;

Algorithm 4: Elimination $elim(i, eliminator(i, k), k)$

(a) With TS kernels
$A_{eliminator(i,k),k} \leftarrow GEQRT(A_{eliminator(i,k),k})$;
$A_{i,k}, A_{eliminator(i,k),k} \leftarrow$
$TSQRT(A_{i,k}, A_{eliminator(i,k),k})$;
for $j = k + 1$ *to* $n - 1$ **do**
  $A_{eliminator(i,k),j} \leftarrow$
  $UNMQR(A_{eliminator(i,k),j}, A_{eliminator(i,k),k}$;
  $A_{i,j}, A_{eliminator(i,k),j} \leftarrow$
  $TSMQR(A_{i,j}, A_{eliminator(i,k),j}, A_{i,k})$;
(b) With TT kernels
$A_{eliminator(i,k),k} \leftarrow GEQRT(A_{eliminator(i,k),k})$;
$A_{i,k} \leftarrow GEQRT(A_{i,k})$;
for $j = k + 1$ *to* $n - 1$ **do**
  $A_{eliminator(i,k),j} \leftarrow$
  $UNMQR(A_{eliminator(i,k),j}, A_{eliminator(i,k),k}$;
  $A_{i,j} \leftarrow UNMQR(A_{i,j}, A_{i,k}$;
$A_{i,k}, A_{eliminator(i,k),k} \leftarrow$
$TTQRT(A_{i,k}, A_{eliminator(i,k),k})$;
for $j = k + 1$ *to* $n - 1$ **do**
  $A_{i,j}, A_{eliminator(i,k),j} \leftarrow$
  $TTMQR(A_{i,j}, A_{eliminator(i,k),j}, A_{i,k})$;

Considering Algorithm 2, the difference lies in the first line: rather than calling $GETRF(A_{kk})$, thereby searching for pivots only within the diagonal tile A_{kk}, we implemented a variant where we extend the search for pivots across the *diagonal domain* (the *Apply* step is modified accordingly). Working on the diagonal domain instead of the diagonal tile increases the smallest singular value of the factored region and therefore increases the likelihood of an LU step. Since all tiles in the diagonal domain are local to a single node, extending the search to the diagonal domain is done without any inter-domain communication. The stability analysis of Section III applies to both scenarios, the one where A_{kk} is factored in isolation, and the one where it is factored with the help of the diagonal domain. In the experimental section, we will use the variant which factors the diagonal domain.

B. QR step

If the decision to process a QR step is taken by the criterion, the LU decomposition of the diagonal domain is dropped, and the factorization of the panel starts over. This step of the factorization is then processed using orthogonal transformations. Every tile below the diagonal (matrix B in Equation (1)) is zeroed out using a triangular tile, or eliminator tile. In a QR step, the diagonal tile is factored (with a GEQRF kernel) and used to eliminate all the other tiles of the panel (with a TSQRT kernel) The trailing sub-matrix is updated, respectively, with UNMQR and TSMQR

kernels. To further increase the degree of parallelism of the algorithm, it is possible to use several eliminator tiles inside a panel, typically one (or more) per domain. The only condition is that concurrent elimination operations must involve disjoint tile pairs (the unique eliminator of tile A_{ik} will be referred to as $A_{eliminator(i,k),k}$). Of course, in the end, there must remain only one non-zero tile on the panel diagonal, so that all eliminators except the diagonal tile must be eliminated later on (with a TTQRT kernel on the panel and TTMQR updates on the trailing submatrix), using a reduction tree of arbitrary shape. This reduction tree will involve inter-domain communications. In our hybrid LU-QR algorithm, the QR step is processed following an instance of the generic hierarchical QR factorization HQR [4] described in Algorithms 3 and 4.

Each elimination $elim(i, eliminator(i,k), k)$ consists of two sub-steps: first in column k, tile (i,k) is zeroed out (or killed) by tile $(eliminator(i,k), k)$; and in each following column $j > k$, tiles (i,j) and $(eliminator(i,k), j)$ are updated; all these updates are independent and can be triggered as soon as the elimination is completed. The orthogonal transformation $elim(i, eliminator(i,k), k)$ uses either a TTQRT kernel or a TSQRT kernel depending upon whether the tile to eliminate is either triangular or square. In the *LU-QR Algorithm*, any combination of reduction trees of the HQR algorithm described in [4] is available. It is then possible to use an intra-domain reduction tree to locally eliminate many tiles without inter-domain communication. A unique triangular tile is left on each node and then the reductions across domains are performed following a second level of reduction tree.

III. Robustness criteria

The decision to process an LU or a QR step is done dynamically during the factorization, and constitutes the heart of the algorithm. Indeed, the decision criteria has to be able to detect a potentially "large" stability deterioration (according to a threshold) due to an LU step before its actual computation, in order to preventively switch to a QR step. As explained in Section II, in our hybrid LU-QR algorithm, the diagonal tile is factored using an LU decomposition with partial pivoting. At the same time, some data (like the norm of non local tiles belonging to other domains) are collected and exchanged (using an all-reduce algorithm) between all nodes hosting at least one tile of the panel. Based upon this information, all nodes make the decision to continue the LU factorization step or to drop the LU decomposition of the diagonal tile and process a full QR factorization step. The decision is broadcast to the other nodes not involved in the panel factorization within the next data communication. The decision process cost will depend on the choice of the criterion and must not imply a large computational overhead compared to the factorization cost. A good criterion will detect only the "worst" steps and will provide a good stability result with as few QR steps as possible. In this section, we present three criteria, going from the most elaborate (but also most costly) to the simplest ones.

The stability of a step is determined by the growth of the norm of the updated matrix. If a criterion determines the potential for an unacceptable growth due to an LU step, then a QR step is used. A QR step is stable as there is no growth in the norm (2-norm) since it is a unitary transformation. Each criterion depends on a threshold α that allows us to tighten or loosen the stability requirement, and thus influence the amount of LU steps that we can afford during the factorization. The optimal choice of α is not known. In Section V-C, we experiment with different choices of α for each criterion.

A. Max criterion

LU factorization with partial pivoting chooses the largest element of a column as the pivot element. Partial pivoting is accepted as being numerically stable. However, pivoting across nodes is expensive. To avoid this pivoting, we generalize the criteria to tiles and determine if the diagonal tile is an acceptable pivot. A step is an LU step if

$$\alpha \times \|(A_{kk}^{(k)})^{-1}\|_1^{-1} \geq \max_{i>k} \|A_{i,k}^{(k)}\|_1. \tag{2}$$

For the analysis we do not make an assumption as to how the diagonal tile is factored. We only assume that the diagonal tile is factored in a stable way (LU with partial pivoting or QR are acceptable). Note that, for the variant using pivoting in the diagonal domain (see Section II-A), which is the variant we experiment with in Section V, $A_{kk}^{(k)}$ represents the diagonal tile after pivoting among tiles in the diagonal domain.

To assess the growth of the norm of the updated matrix, consider the update of the trailing sub-matrix. For all $i, j > k$ we have:

$$\begin{aligned}
\|A_{i,j}^{(k+1)}\|_1 &= \|A_{i,j}^{(k)} - A_{i,k}^{(k)}(A_{k,k}^{(k)})^{-1}A_{k,j}^{(k)}\|_1 \\
&\leq \|A_{i,j}^{(k)}\|_1 + \|A_{i,k}^{(k)}\|_1\|(A_{k,k}^{(k)})^{-1}\|_1\|A_{k,j}^{(k)}\|_1 \\
&\leq \|A_{i,j}^{(k)}\|_1 + \alpha\|A_{k,j}^{(k)}\|_1 \\
&\leq (1+\alpha)\max\left(\|A_{i,j}^{(k)}\|_1, \|A_{k,j}^{(k)}\|_1\right) \\
&\leq (1+\alpha)\max_{i\geq k}\left(\|A_{i,j}^{(k)}\|_1\right).
\end{aligned}$$

The growth of any tile in the trailing sub-matrix is bounded by $1+\alpha$ times the largest tile in the same column. If every step satisfies (2), then we have the following bound:

$$\frac{\max_{i,j,k}\|A_{ij}^{(k)}\|_1}{\max_{i,j}\|A_{i,j}\|_1} \leq (1+\alpha)^{n-1}.$$

The expression above is a growth factor on the norm of the tiles. For $\alpha = 1$, the growth factor of 2^{n-1} is an analogous result to an LU factorization with partial pivoting (scalar

case) [6]. Finally, note that we can obtain this bound by generalizing the standard example for partial pivoting. The following matrix will match the bound above:

$$A = \begin{pmatrix} \alpha^{-1} & 0 & 0 & 1 \\ -1 & \alpha^{-1} & 0 & 1 \\ -1 & -1 & \alpha^{-1} & 1 \\ -1 & -1 & -1 & 1 \end{pmatrix}.$$

B. Sum criterion

A stricter criteria is to compare the diagonal tile to the sum of the off-diagonal tiles:

$$\alpha \times \|(A_{kk}^{(k)})^{-1}\|_1^{-1} \geq \sum_{i>k} \|A_{i,k}^{(k)}\|_1. \tag{3}$$

Again, for the analysis, we only assume A_{kk}^{-1} factored in a stable way. For $\alpha \geq 1$, this criterion (and the Max criterion) is satisfied at every step if A is block diagonally dominant [6]. That is, a general matrix $A \in \mathbb{R}^{n\times n}$ is block diagonally dominant by columns with respect to a given partitioning $A = (A_{ij})$ and a given norm $\|\cdot\|$ if: $\forall j \in [\![1, n]\!], \|A_{jj}^{-1}\|^{-1} \geq \sum_{i\neq j} \|A_{ij}\|$. Again, we need to evaluate the growth of the norm of the updated trailing submatrix. For all $i, j > k$, we have

$$\begin{aligned} \sum_{i>k} \|A_{i,j}^{(k+1)}\|_1 &= \sum_{i>k} \|A_{i,j}^{(k)} - A_{i,k}^{(k)}(A_{k,k}^{(k)})^{-1}A_{k,j}^{(k)}\|_1 \\ &\leq \sum_{i>k} \|A_{i,j}^{(k)}\|_1 \\ &\quad + \|A_{k,j}^{(k)}\|_1 \|(A_{k,k}^{(k)})^{-1}\|_1 \sum_{i>k} \|A_{i,k}^{(k)}\|_1 \\ &\leq \sum_{i>k} \|A_{i,j}^{(k)}\|_1 + \alpha \|A_{k,j}^{(k)}\|_1. \end{aligned}$$

Hence, the growth of the updated matrix can be bounded in terms of an entire column rather than just an individual tile. The only growth in the sum is due to the norm of a single tile. For $\alpha = 1$, the inequality becomes $\sum_{i>k} \|A_{i,j}^{(k+1)}\|_1 \leq \sum_{i\geq k} \|A_{i,j}^{(k)}\|_1$. If every step of the algorithm satisfies (3) (with $\alpha = 1$), then by induction we have: $\sum_{i>k} \|A_{i,j}^{(k+1)}\|_1 \leq \sum_{i\geq 1} \|A_{i,j}\|_1$,, for all i, j, k. This leads to the following bound:

$$\frac{\max_{i,j,k} \|A_{ij}^{(k)}\|_1}{\max_{i,j} \|A_{i,j}\|_1} \leq n.$$

From this we see that the criteria eliminates the potential for exponential growth due to the LU steps. Note that for a diagonally dominant matrix, the bound on the growth factor can be reduced to 2 [6].

C. MUMPS criterion

In LU decomposition with partial pivoting, the largest element of the column is use as the pivot. This method is stable experimentally, but the seeking of the maximum and the pivoting requires a lot of communications in distributed memory. Thus in an LU step of the *LU-QR Algorithm*, the LU decomposition with partial pivoting is limited to the local tiles of the panel (i.e., to the diagonal domain). The idea behind the MUMPS criterion is to estimate the quality of the pivot found locally compared to the rest of the column. The MUMPS criterion is one of the strategies available in MUMPS, although it is for symmetric indefinite matrices (LDL^T) [7].

At step k of the *LU-QR Algorithm*, let $L^{(k)}U^{(k)}$ be the LU decomposition of the diagonal domain and $A_{ij}^{(k)}$ be the value of the tile A_{ij} at the beginning of step k. Let $local\_max_k(j)$ be the largest element of the column j of the panel in the diagonal domain, $away\_max_k(j)$ be the largest element of the column j of the panel off the diagonal domain, and $pivot_k$ be the list of pivots used in the LU decomposition of the diagonal domain:

$$\begin{aligned} local\_max_k(j) &= \max_{\substack{\text{tiles } A_{i,k} \text{ on the} \\ \text{diagonal domain}}} \max_l |(A_{i,k})_{l,j}|, \\ away\_max_k(j) &= \max_{\substack{\text{tiles } A_{i,k} \text{ off the} \\ \text{diagonal domain}}} \max_l |(A_{i,k})_{l,j}|, \\ pivot_k(j) &= |U_{j,j}^{(k)}|. \end{aligned}$$

$pivot_k(j)$ represents the largest local element of the column j at step j of the LU decomposition with partial pivoting on the diagonal domain. Thus, we can express the growth factor of the largest local element of the column j at step j as: $growth\_factor_k(j) = pivot_k(j)/local\_max_k(j)$. The idea behind the MUMPS criterion is to estimate if the largest element outside the local domain would have grown the same way. Thus, we can define a vector $estimate\_max_k$ initialized to $away\_max_k$ and updated for each step i of the LU decomposition with partial pivoting like $estimate\_max_k(j) \leftarrow estimate\_max_k(j) \times growth\_factor_k(i)$. We consider that the LU decomposition with partial pivoting of the diagonal domain can be used to eliminate the rest of the panel if and only if all pivots are larger than the estimated maximum of the column outside the diagonal domain times a threshold α. Thus, the MUMPS criterion (as we implemented it) decides that step k of the *LU-QR Algorithm* will be an LU step if and only if:

$$\forall j, \alpha \times pivot_k(j) \geq estimate\_max_k(j). \tag{4}$$

D. Complexity

All criteria require the reduction of information of the off-diagonal tiles to the diagonal tile. Criteria (2) and (3) require the norm of each tile to be calculated locally (our implementation uses the 1-norm) and then reduced to the diagonal tile. Both criteria also require computing $\|A_{kk}^{-1}\|$. Since the LU factorization of the diagonal tile is computed, the norm can be approximated using the L and U factors by an iterative method in $O(n_b^2)$ operations. The overall

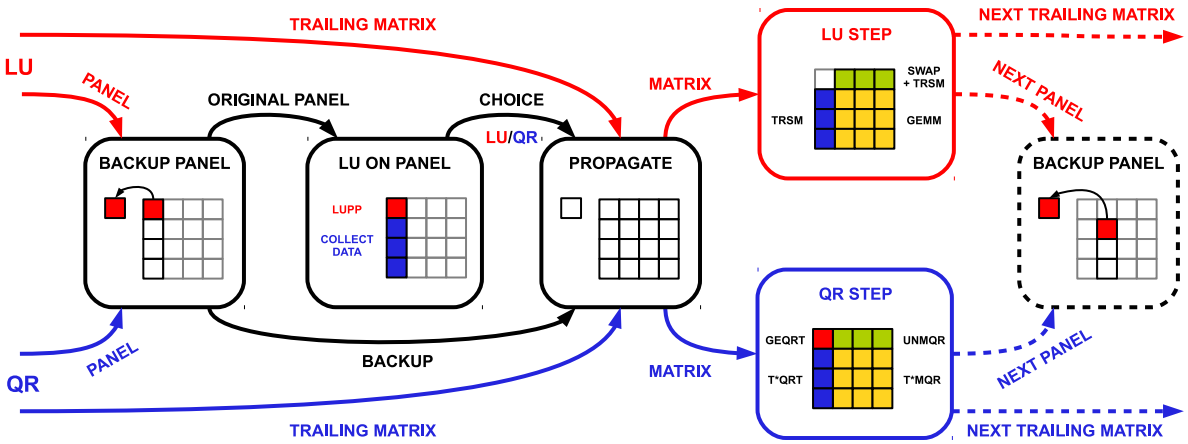

Figure 1: Dataflow of one step of the algorithm.

complexity for both criteria is $O(n \times n_b^2)$. Criterion (4) requires the maximum of each column be calculated locally and then reduced to the diagonal tile. The complexity of the MUMPS criterion is also $O(n \times n_b^2)$ comparisons.

The Sum criterion is the strictest of the three criteria. It also provides the best stability with linear growth in the norm of the tiles in the worst case. The other two criteria have similar worst case bounds. The growth factor for both criteria are bound by the growth factor of partial (threshold) pivoting. The Max criterion has a bound for the growth factor on the norm of the tiles that is analogous to partial pivoting. The MUMPS criteria does not operate at the tile level, but rather on scalars. If the estimated growth factor computed by the criteria is a good estimate, then the growth factor is no worse than partial (threshold) pivoting.

IV. Implementation

As discussed in section I, we have implemented the *LU-QR Algorithm* on top of the PaRSEC runtime. This choice implied major difficulties due to the parameterized task graph representation exploited by the PaRSEC runtime. This representation being static, a solution had to be developed to allow for dynamism in the graph traversal. To solve this issue, a layer of selection tasks has been inserted between each elimination step of the algorithm. These tasks are executed only once a control flow has been sent to them after the criterion selection. Thus, they delay the decision to send the data to the next elimination step until a choice has been made, in order to guarantee that data follow the correct path. These are the *Propagate* tasks on Figure 1. Note that these tasks, as well as *Backup Panel* tasks, can receive the same data from two different paths. In the PaRSEC runtime, tasks are created only when one of their dependencies is solved, then by graph construction they are enabled only when the previous elimination step has already started, hence they will receive their data only from the correct path.

Figure 1 describes the connection between the different stages of one elimination step of the algorithm. These stages are described below:

- Backup Panel: This is a set of tasks that collect the tiles of the panel from the previous step. Since an LU factorization will be performed in-place for criterion computation, a backup of the tiles on the diagonal domain is created and directly sent to the *Propagate* tasks in case a QR elimination step is needed. All tiles belonging to the panel are forwarded to the *LU On Panel* tasks.
- LU On Panel: Once the backup is done, the criterion is computed. The first node computes the U matrix related to this elimination step. This could be done through LU factorization with or without pivoting. We decided to exploit the multi-threaded recursive-LU kernel from the PLASMA library to enlarge the pivot search space while keeping good efficiency [8]. All other nodes compute information required for the criterion (see section III). Then, an all-reduce operation is performed to exchange the information, so that everyone can take and store the decision in a local array. Once the decision is known, data on panel are forwarded to the appropriate *Propagate* tasks and a control flow triggers all to release the correct path in the dataflow.
- Propagate: These tasks, one per tile, receive the decision from the previous stage through a control flow, and are responsible for forwarding the data to the computational tasks of the selected factorization. The tasks belonging to the panel (assigned to the first nodes) have to restore the data back to their previous state if QR elimination is chosen. In all cases, the backup is destroyed upon exit of these tasks.

We are now ready to describe each step:

a) LU Step*:* If the numerical criterion is met by the panel computation, the update step is performed. On the nodes with the *diagonal* row, the update is made according to the factorization used on the panel. Here, a swap is performed with all tiles of the local panel, and then a triangular solve is applied to the first row. On all other nodes, a block LU algorithm is used to performed the update. This means that the panel is updated with $TRSM$ tasks, and the trailing sub-matrix is updated with $GEMM$ tasks. This avoids the row pivoting between the nodes usually

performed by the classical LU factorization algorithm with partial pivoting, or by tournament pivoting algorithms [9].

b) QR STEP*:* If the numerical criterion is not met, a QR factorization has to be performed. Many solutions could be used for this elimination step. We chose to exploit the HQR method implementation presented in [4]. This allowed us to experiment with different kinds of reduction trees, so as to find the most adapted solution to our problem. Our default tree (which we use in all of our experiments) is a hierarchical tree made of GREEDY reduction trees inside nodes and a FIBONACCI reduction tree between the nodes. The FIBONACCI tree between nodes has been chosen for its short critical path and its good pipelining of consecutive trees, in case some QR steps are performed in sequence. The GREEDY reduction tree is favored within a node. A two-level hierarchical approach is natural when considering multicore parallel distributed architectures. (See [4] for more details on the reduction trees).

To implement the *LU-QR Algorithm* within the PARSEC framework, two extensions had to be implemented within the runtime. The first extension allows the programmer to generate data during the execution with the OUTPUT keywords. This data is then inserted to the tracking system of the runtime to follow its path in the dataflow. This has been used to generate the backup on the fly, and to limit the memory peak of the algorithm. The second extension is for the end detection of the algorithm. Due to its distributed nature, PARSEC goes over all the domain space of each type of task and uses a predicate, namely *the owner computes* rule, to decide if a task is local or not. Local tasks are counted and the end of the algorithm is detected when all of them have been executed. Due to the dynamism in the *LU-QR Algorithm*, the size of the domain space is larger than the number of tasks that will be really executed. Thus, a function to dynamically increase/decrease the number of local tasks has been added, so that the *Propagate* task of each node updates the local counter according to the elimination step chosen.

V. EXPERIMENTS

The purpose of this section is to present numerical experiments for the hybrid *LU-QR Algorithm*, and to highlight the trade-offs between stability and performance that can be achieved by tuning the threshold α in the robustness criterion (see Section III).

A. Experimental framework

We used *Dancer*, a parallel machine hosted at the Innovative Computing Laboratory (ICL) in Knoxville, to run the experiments. This cluster has 16 multi-core nodes, each equipped with 8 cores, and an Infiniband 10G interconnection network. The nodes feature two Intel Westmere-EP E5606 CPUs at 2.13GHz. The system is running the Linux 64bit operating system, version 3.7.2-x86_64. The software was compiled with the Intel Compiler Suite 2013.3.163. BLAS kernels were provided by the MKL library and OpenMPI 1.4.3 has been used for the MPI communications by the PARSEC runtime. Each computational thread is bound to a single core using the HwLoc 1.7.1 library. If not mentioned otherwise, we will use all 16 nodes and the data will be distributed according to a 4-by-4 2D-block-cyclic distribution. The theoretical peak performance of the 16 nodes is 1091 GFlop/s.

For each experiment, we consider a square tiled-matrix A of size N-by-N, where $N = n \times n_b$. The tile size n_b has been fixed to 240 for the whole experiment set, because this value was found to achieve good performance for both LU and QR steps. We evaluate the backward stability by computing the HPL3 accuracy test of the High-Performance Linpack benchmark:

$$HPL3 = \frac{\|Ax - b\|_\infty}{\|A\|_\infty \|x\|_\infty \times \epsilon \times N},$$

where x is the computed solution and ϵ is the machine precision. Each test is run with double precision arithmetic. For performance, we point out that the number of floating point operations executed by the hybrid algorithm depends on the number of LU and QR steps performed during the factorization. Thus, for a fair comparison, we assess the efficiency by reporting the *normalized* GFlop/s performance computed as

$$\text{GFlop/s} = \frac{\frac{2}{3}N^3}{\text{EXECUTION TIME}},$$

where $\frac{2}{3}N^3$ is the number of operations for LU with partial pivoting and EXECUTION TIME is the execution time of the algorithm. With this formula, QR factorization will only achieve half of the performance due to the $\frac{4}{3}N^3$ operations of the algorithm.

B. Results for random matrices

We start with the list of the algorithms used for comparison with the *LU-QR Algorithm*. For fairness, they are all implemented within the PaRSEC framework:

- LU NoPiv, which performs pivoting only inside the diagonal tile but no pivoting across tiles (known to be both efficient and unstable)
- LU IncPiv, which performs incremental pairwise pivoting across all tiles in the elimination panel [1], [2] (still efficient but not stable either)
- Several instances of the hybrid *LU-QR Algorithm*, for different values of the robustness parameter α. Recall that the algorithm performs pivoting only across the diagonal domain, hence involving no remote communication nor synchronization.
- HQR, the Hierarchical QR factorization [4], with the same configuration as in the QR steps of the *LU-QR Algorithm*: GREEDY reduction trees inside nodes and FIBONACCI reduction trees between the nodes.

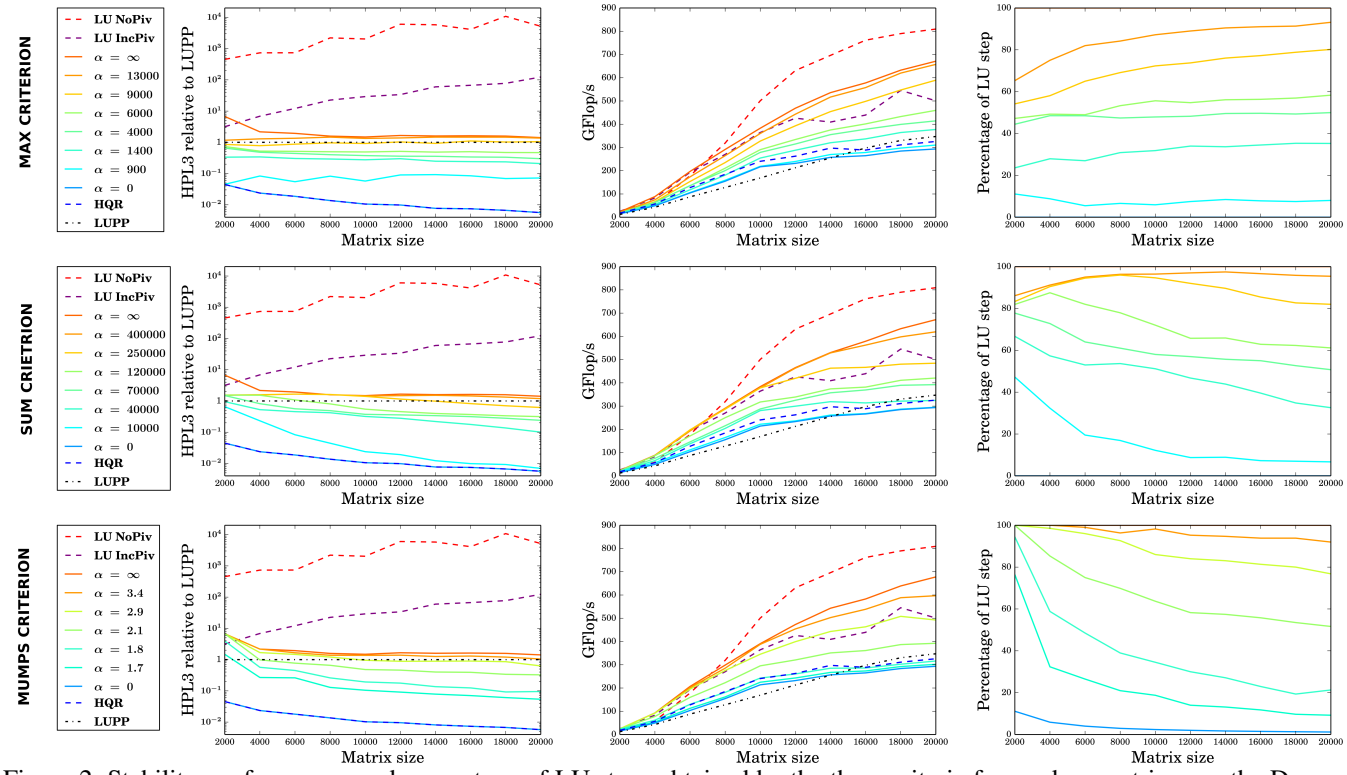

Figure 2: Stability, performance, and percentage of LU steps obtained by the three criteria for random matrices on the Dancer platform (4x4 grid).

For reference, we also include a comparison with PDGEQRF: this is the LUPP algorithm (LU with partial pivoting across all tiles of the elimination panel) from the reference ScaLAPACK implementation.

Figure 2 summarizes all results for random matrices. It is organized as follows: each row corresponds to one criterion. Within a row:

- the first column shows the relative stability (ratio of HPL3 value divided by HPL3 value for LUPP)
- the second column shows the GFlop/s performance
- the third column shows the percentage of LU steps during execution

Results are average values obtained on a set of 100 random matrices (we observe a very small standard deviation, less than 2%).

For each criterion, we experimentally chose a set of values of α that provides a representative range of ratios for the number of LU and QR steps. As explained in Section III, for each criterion, the smaller the α is, the tighter the stability requirement. Thus, the numerical criterion is met less frequently and the hybrid algorithm processes fewer LU steps. A current limitation of our approach is that we do not know how to auto-tune the best range of values for α, which seems to depend heavily upon matrix size and available degree of parallelism. In addition, the range of useful α values is quite different for each criterion.

For random matrices, we observe in Figure 2 that the stability of LU NoPiv and LU IncPiv is not satisfactory. We also observe that, for each criterion, small values of α result in better stability, to the detriment of performance. For $\alpha = 0$, *LU-QR Algorithm* processes only QR steps, which leads to the exact same stability as the HQR Algorithm and almost the same performance results. The difference between the performance of *LU-QR Algorithm* with $\alpha = 0$ and HQR comes from the cost of the decision making process steps (saving the panel, computing the LU factorization with partial pivoting on the diagonal domain, computing the choice, and restoring the panel). Figure 2 shows that the overhead due to the decision making process is approximately equal to 10% for the three criteria. This overhead, computed when QR eliminations are performed at each step, is primarily due to the backup/restore steps added to the critical path when QR is chosen. Performance impact of the criterion computation itself is negligible, as one can see by comparing performance of the random criterion to the MUMPS and Max criteria.

For $\alpha = \infty$, the *LU-QR Algorithm* processes only LU steps, with a stability slightly inferior to that of LUPP. When the matrix size increases, the relative stability results of the *LU-QR Algorithm* with $\alpha = \infty$ tends to 1, which means (somewhat unexpectedly) that, on random matrices, processing an LU factorization with partial pivoting on the diagonal domain followed by a flat tree elimination of the rest of the panel is almost as stable as an LU factorization with partial pivoting on the whole panel.

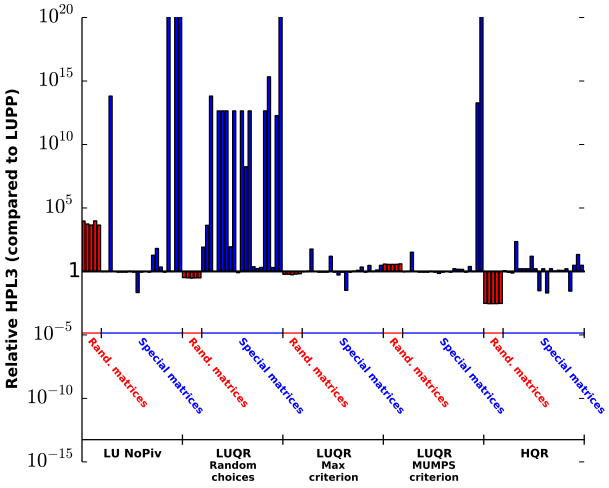

Figure 3: Stability on special matrices.

C. Results for special matrices

We test the hybrid *LU-QR Algorithm* on a collection of matrices that includes several pathological matrices on which LUPP fails because of large growth factors. Figure 3 provides the relative stability (ratio of HPL3 divided by HPL3 for LUPP) obtained by running the hybrid *LU-QR Algorithm* with 5 random matrices (for comparison) and a set of special matrices from the Higham's Matrix Computation Toolbox [6] (see [5] for details). Matrix size is set to $N = 40,000$, and experiments were run on a 16-by-1 process grid. The parameter α has been set to 50 for the random criterion, $6,000$ for the Max criterion, and 2.1 for the MUMPS criterion (we do not report result for the Sum criterion because they are the same as they are for Max). In addition to the Max and MUMPS criteria, we report results obtained with *random choices* between LU and QR at each step of the algorithm: this is to assess the usefulness of the criteria. Figure 3 considers LU NoPiv, HQR and the *LU-QR Algorithm*. The first observation is that using random choices now leads to numerical instability. The Max criterion provides a good stability ratio on every tested matrix (up to 58 for the RIS matrix and down to 0.03 for the Invhess matrix). The MUMPS criterion also gives modest growth factor for the whole experiment set except for the Wilkinson and the Foster matrices, for which it fails to detect some "bad" steps.

We also experimented with the Fiedler matrix from the Higham's Matrix Computation Toolbox [6]. We observed that LU NoPiv and LUPP failed (due to small values rounded up to 0 and then illegally used in a division), while the Max and the MUMPS criteria provide HPL3 values ($\approx 5.16 \times 10^{-09}$ and $\approx 2.59 \times 10^{-09}$) comparable to that of HQR ($\approx 5.56 \times 10^{-09}$). This proves that our criteria can detect and handle pathological cases for which the generic LUPP algorithm fails.

D. Assessment of the three criteria

With respect to stability, while the three criteria behave similarly on random matrices, we observe different behaviors for special matrices. The MUMPS criterion provides good results for most of the tested matrices but not for all. If stability is the key concern, one may prefer to use the Max criterion (or the Sum criterion), which performs well for all special matrices (which means that the upper bound on the growth is quite pessimistic).

With respect to performance, we observe very comparable results, which means that the overhead induced by computing the criterion at each step is of the same order of magnitude for all criteria.

The overall conclusion is that all criteria bring significant improvement over LUPP in terms of stability, and over HQR in terms of performance. Tuning the value of the robustness parameter α enables the exploration of a wide range of stability/performance trade-offs.

VI. Related work

State-of-the-art QR factorizations use multiple eliminators per panel, in order to dramatically reduce the critical-path length of the algorithm. These algorithms are unconditionally stable, and their parallelization has been fairly well studied on shared memory systems [1], [2], [10] and on parallel distributed systems [4].

The reason for using LU kernels instead of QR kernels is performance: (i) LU performs half the number of operations of QR; (ii) LU kernels relies on GEMM kernels which are very efficient while QR kernels are more complex and much less tuned, hence not that efficient; and (iii) the LU update is much more parallel than the QR update. So all in all, LU is much faster than QR (as observed in the performance results of Section V). Because of the large number of communications and synchronizations induced by pivoting in the reference LUPP algorithm, *communication-avoiding* variants of LUPP have been introduced [11], but they have proven much more challenging to design because of stability issues. We review several approaches:

- *LUPP:* LU with partial pivoting is not a communication-avoiding scheme and its performance in a parallel distributed environment is low (see Section V). However, the LUPP algorithm is *stable in practice*, and we use it as a reference for stability.
- *LU NoPiv:* The most basic communication-avoiding LU algorithm is LU NoPiv. This algorithm is stable for block diagonal dominant matrices [6], but breaks down if it encounters a nearly singular diagonal tile, or loses stability if it encounters a diagonal tile whose smallest singular value is too small. Baboulin et al. [12] propose to apply a random transformation to the initial matrix, in order to use LU NoPiv while maintaining stability. This approach gives about the same performance as LU NoPiv, since preprocessing and postprocessing costs are negligible. However, for any matrix

which is rendered stable by this approach (i.e, LU NoPiv is stable), there exists a matrix which is rendered not stable. But in practice, this proves to be a valid approach.

• *LU IncPiv:* LU IncPiv is another communication-avoiding LU algorithm [1], [2]. Incremental pivoting is also called *pairwise pivoting*. The stability of the algorithm [1] is not sufficient and degrades as the number of tiles in the matrix increases (see our experimental results on random matrices). The method also suffers some of the same performance degradation of QR factorizations with multiple eliminators per panel, namely low-performing kernels, and some dependencies in the update phase.

• *CALU:* CALU [9] is a communication-avoiding LU. It uses tournament pivoting which has been proven to be *stable in practice* [9]. CALU shares the (good) properties of one of our LU steps: (i) low number of operations; (ii) use of efficient GEMM kernels; and (iii) embarrassingly parallel update. The advantage of CALU over our algorithm is essentially that it performs only LU steps, while our algorithm might need to perform some (more expensive) QR steps. The disadvantage is that, at each step, CALU needs to perform global pivoting on the whole panel, which then needs to be reported during the update phase to the whole trailing submatrix. There is no publicly available implementation of parallel distributed CALU, and it was not possible to compare stability or performance.

VII. Conclusion

Linear algebra software designers have been struggling for years to improve the parallel efficiency of LUPP (LU with partial pivoting), the de-facto choice method for solving dense systems. The search for good pivots throughout the elimination panel is the key for stability (and indeed both NoPiv and IncPiv fail to provide acceptable stability), but dramatically decreases the overall performance of the factorization.

Communication-avoiding and critical-path reducing algorithms prove very relevant on today's architectures. In our experiments, our HQR factorization [4] based of QR kernels obtains similar performance as ScaLAPACK LUPP, while performing 2x more operations, using slower sequential kernels, and being undermined by a less parallel update phase. In this paper, stemming from the key observation that LU steps and QR steps can be mixed during a factorization, we present the *LU-QR Algorithm* whose goal is to accelerate the HQR algorithm by introducing some LU steps whenever these do not compromise stability. The hybrid algorithm represents dramatic progress in a long-standing research problem. By restricting to pivoting inside the diagonal domain, i.e., locally, but by doing so only when the robustness criterion forecasts that it is safe (and going to a QR step otherwise), we improve performance while guaranteeing stability. And we provide a continuous range of trade-offs between LU NoPiv (efficient but only stable for diagonally-dominant matrices) and QR (always stable but twice as costly and with less performance).

This work opens several research directions. There are many variants and extensions of the hybrid algorithm that can be envisioned (some are described in [5]). Another goal would be to derive LU algorithms with several eliminators per panel (just as for HQR) to decrease the critical-path length, provided the availability of a reliable robustness test to ensure stability.

References

[1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.

[2] G. Quintana-Ortí, E. S. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan, "Programming matrix algorithms-by-blocks for thread-level parallelism," *ACM Trans. Math. Softw.*, vol. 36, no. 3, pp. 1–26, 2009.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. 1, pp. 37–51, 2012.

[4] J. Dongarra, M. Faverge, T. Hérault, M. Jacquelin, J. Langou, and Y. Robert, "Hierarchical QR factorization algorithms for multi-core cluster systems," *Parallel Computing*, vol. 39, no. 4-5, pp. 212–232, 2013.

[5] M. Faverge, J. Herrmann, J. Langou, B. Lowery, Y. Robert, and J. Dongarra, "Designing LU-QR hybrid solvers for performance and stability," LAPACK Working Note 282, October 2013.

[6] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM Press, 2002.

[7] I. S. Duff and S. Pralet, "Strategies for scaling and pivoting for sparse symmetric indefinite problems," *SIAM J. Matrix Anal. Appl.*, vol. 27, no. 2, pp. 313–340, 2005.

[8] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "Achieving numerical accuracy and high performance using recursive tile lu factorization with partial pivoting," *Concurrency and Computation: Practice and Experience*, 2013, available online.

[9] L. Grigori, J. W. Demmel, and H. Xiang, "CALU: a communication optimal LU factorization algorithm," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1317–1350, 2011.

[10] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, "Tiled QR factorization algorithms," in *Proc. ACM/IEEE SC11 Conference*. ACM Press, 2011.

[11] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *SIAM J. Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012.

[12] M. Baboulin, J. J. Dongarra, J. Herrmann, and S. Tomov, "Accelerating linear system solutions using randomization techniques," *ACM Trans. Mathematical Software*, vol. 39, no. 2, pp. 8:1–8:13, 2013.