

Designing LU-QR hybrid solvers for performance and stability

Mathieu Faverge¹, Julien Herrmann², Julien Langou³,
Bradley Lowery³, Yves Robert^{2,4} and Jack Dongarra⁴

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 floating-point 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 floating-point operations, and a more complicated update step that is not as parallel as a matrix-matrix product. Tiled QR algorithms [1]–[3] 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. In a nutshell, the 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. Technically, this is very similar to a step during a block LU factorization [4]. 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. A slight change in the algorithm, or in the matrix layout across the nodes, might call for a time-consuming and error-prone process of code adaptation. For each version, one must identify, and adequately implement, inter-node versus intra-node kernels. This dramatically complicates the task of the programmers if they rely on a manual approach. We solve this problem by relying on the PaRSEC software [5]–[7], 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 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 [8].

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:

- The introduction of new LU-QR hybrid algorithms;
- The design of several robustness criteria, with bounds on the induced growth factor;
- A comprehensive experimental evaluation of the best trade-offs between performance and numerical stability;
- The 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 exper-

imental 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:

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

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	
<i>factor A</i>	2/3	GETRF	4/3	GEQRT
<i>eliminate B</i>	$(n-1)$	TRSM	$2(n-1)$	TSQRT
<i>apply C</i>	$(n-1)$	TRSM	$2(n-1)$	TSMQR
<i>update D</i>	$2(n-1)^2$	GEMM	$4(n-1)^2$	UNMQR

Table I: Computational cost of each kernel. The unit is n_b^3 floating-point 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

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;
```

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 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.

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} .

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:  $A_{ij} \leftarrow GEMM(A_{ik}, A_{kj}, A_{ij})$ ;
```

- *Eliminate:* $A_{ik} \leftarrow TRSM(A_{kk}, A_{ik})$ solves in-place, the upper triangular system such that $A_{ik} \leftarrow A_{ik}U_{kk}^{-1}$ where U_{kk} is stored in the upper part of A_{kk} .
- *Apply:* $A_{kj} \leftarrow SWPTRSM(A_{kk}, A_{kj})$ solves the unit lower triangular system such that $A_{kj} \leftarrow L_{kk}^{-1}P_{kk}A_{kj}$ where L_{kk} is stored in the (strictly) lower part of A_{kk} .
- *Update:* $A_{ij} \leftarrow GEMM(A_{ik}, A_{kj}, A_{ij})$ is a general matrix-matrix multiplication $A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$.

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 floating-point operations.

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

Algorithm 3: Step k of the HQR factorization

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

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 submatrix 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 [8] 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 algorithm is entirely characterized by its elimination list, which is the ordered list of all the eliminations $elim(i, eliminator(i, k), k)$ that are executed. 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 our hybrid *LU-QR Algorithm*, any combination of reduction trees of the HQR algorithm described in [8] is available. It is then possible to use an intra-domain reduction tree to

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  
   $\lfloor$   $A_{eliminator(i,k),j} \leftarrow$   
     $UNMQR(A_{eliminator(i,k),j}, A_{eliminator(i,k),k});$   
     $A_{i,j}, A_{eliminator(i,k),j} \leftarrow$   
     $TSMQR(A_{i,j}, A_{eliminator(i,k),j}, A_{i,k});$ 
```

(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  
   $\lfloor$   $A_{eliminator(i,k),j} \leftarrow$   
     $UNMQR(A_{eliminator(i,k),j}, A_{eliminator(i,k),k});$   
     $A_{i,j} \leftarrow UNMQR(A_{i,j}, A_{i,k});$   
 $A_{i,k}, A_{eliminator(i,k),k} \leftarrow$   
 $TTQRT(A_{i,k}, A_{eliminator(i,k),k});$   
for  $j = k + 1$  to  $n - 1$  do  
   $\lfloor$   $A_{i,j}, A_{eliminator(i,k),j} \leftarrow$   
     $TTMQR(A_{i,j}, A_{eliminator(i,k),j}, A_{i,k});$ 
```

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.

C. LU step variants

In the following, we describe several other variants of the LU step.

1) *Variant (A2)*: It consists of first performing a QR factorization of the diagonal tile and proceeds pretty much as in (A1) thereafter.

- *Factor*: $A_{kk} \leftarrow GEQRF(A_{kk})$ is a QR factorization $A_{kk} = Q_{kk}U_{kk}$, where Q_{kk} is never constructed explicitly and we instead store the Householder reflector V_{kk} . The output matrices V_{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 such that $A_{ik} \leftarrow A_{ik}U_{kk}^{-1}$ where U_{kk} is stored in the upper part of A_{kk} .
- *Apply*: $A_{kj} \leftarrow ORMQR(A_{kk}, A_{kj})$ performs $A_{kj} \leftarrow Q_{kk}^T A_{kj}$ where Q_{kk}^T is applied using V_{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-matrix multiplication $A_{ij} \leftarrow A_{ij} -$

$$A_{ik}A_{kj}.$$

The *Eliminate* and *Update* steps are the exact same as in (A1). The (A2) variant has the same data dependencies as (A1) and therefore the same level of parallelism. A benefit of (A2) over (A1) is that if the criterion test decides that the step is a QR step, then the factorization of A_{kk} is not discarded but rather used to continue the QR step. A drawback of (A2) is that the *Factor* and *Apply* steps are twice as expensive as the ones in (A1).

2) *Variants (B1) and (B2)*: Another option is to use the so-called *block LU factorization* [4]. The result of this formulation is a factorization where the U factor is block upper triangular (as opposed to upper triangular), and the diagonal tiles of the L factor are identity tiles. The *Factor* step can either be done using an LU factorization (variant (B1)) or a QR factorization (variant (B2)). The *Eliminate* step is $A_{ik} \leftarrow A_{ik}A_{kk}^{-1}$. There is no *Apply* step. And the *Update* step is $A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$.

The fact that row k is not updated provides two benefits: (i) A_{kk} does not need to be broadcast to these tiles, simplifying the communication pattern; (ii) The stability of the LU step can be determined by considering only the growth factor in the Schur complement of A_{kk} . One drawback of (B1) and (B2) is that the final matrix is not upper triangular but only block upper triangular. This complicates the use of these methods to solve a linear system of equations. The stability of (B1) and (B2) has been analyzed in [4].

We note that (A2) and (B2) use a QR factorization during the *Factor* step. Yet, we still call this an LU step. This is because all four LU variants mentioned use the Schur complement to update the trailing sub-matrix. The mathematical operation is: $A_{ij} \leftarrow A_{ij} - A_{ik}A_{kk}^{-1}A_{kj}$. In practice, the *Update* step for all four variants looks like $A_{ij} \leftarrow A_{ij} - A_{ik}A_{kj}$, since A_{kk}^{-1} is somehow applied to A_{ik} and A_{kj} during the preliminary *update* and *eliminate* steps. The Schur update dominates the cost of an LU factorization and therefore all variants are more efficient than a QR step. Also, we have the same level of parallelism for the update step: embarrassingly parallel. In terms of stability, all variants would follow closely the analysis of Section V-C. We do not consider further variants (A2), (B1), and (B2) in this paper, since they are all very similar, and only study Algorithm 2, (A1).

D. Comments

1) *Solving systems of linear equations*: To solve systems of linear equations, we augment A with the right-hand side b to get $\tilde{A} = (A, b)$ and apply all

transformations to \tilde{A} . Then an N -by- N triangular solve is needed. This is the approach we used in our experiments. We note that, at the end of the factorization, all needed information about the transformations is stored in place of A , so, alternatively, one can apply the transformations on b during a second pass.

2) *No restriction on N* : In practice, N does not have to be a multiple of n_b . We keep this restriction for the sake of simplicity. The algorithm can accommodate any N and n_b with some clean-up codes, which we have written.

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 a Bruck’s all-reduce algorithm [9]) 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

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

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:

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

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) [10]. 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:

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

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 [10]. 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 sub-matrix. 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:

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

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 [10].

C. MUMPS criterion

In LU decomposition with partial pivoting, the largest element of the column is used 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) [11], and Amestoy et al. [12] provided us with their scalar criterion for the LU case.

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:

$$local_max_k(j) = \max_{\text{tiles } A_{i,k} \text{ on the diagonal domain}} \max_l |(A_{i,k})_{l,j}|,$$

$$away_max_k(j) = \max_{\text{tiles } A_{i,k} \text{ off the diagonal domain}} \max_l |(A_{i,k})_{l,j}|,$$

$$pivot_k(j) = |U_{j,j}^{(k)}|.$$

$pivot_k(j)$ 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:

$$\forall j, \alpha \times pivot_k(j) \geq estimate_max_k(j). \quad (4)$$

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 $O(n_b^2)$ floating-point operations. The overall 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 criterion 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. There are two major reasons for this choice: (i) it allows for easily targeting distributed architectures while concentrating only on the algorithm and not on implementation details such as data distribution and communications; (ii) previous implementations of the HQR algorithm [8] can be reused for QR elimination steps, and they include efficient reduction trees to reduce the critical path of these steps.

However, 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 only executed 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 [13]. 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 complete the description of 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 [14].

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 [8]. 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 [8] 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 into the tracking system of the runtime to follow its path in the dataflow. This is what has been used to generate the backup on the fly, and to limit the memory peak of the algorithm. A second extension has been made for the end detection of the algorithm. Due to its distributed nature, PARSEC goes over all the domain space of each type of task of the algorithm 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 actually be 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,

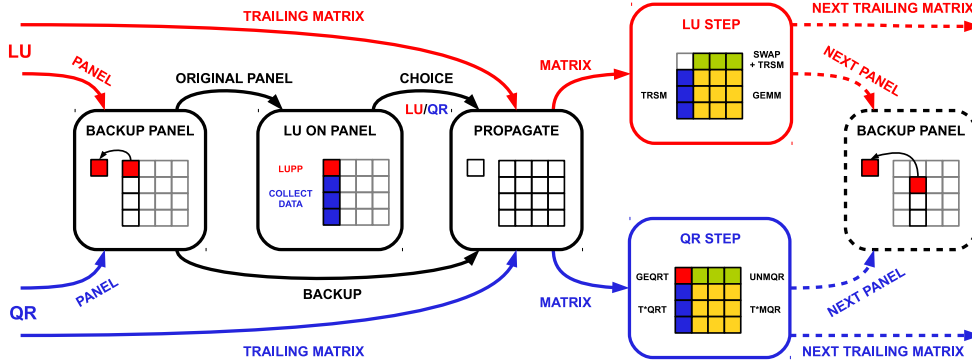


Figure 1: Dataflow of one step of the algorithm.

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/sec.

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 [15]:

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

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/sec performance computed as

$$\text{GFLOP/sec} = \frac{\frac{2}{3}N^3}{\text{EXECUTION TIME}},$$

where $\frac{2}{3}N^3$ is the number of floating-point 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 $\frac{4}{3}N^3$ floating-point 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*. All of these methods 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 [2], [3] (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 [8], 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.

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 [16].

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

- the first column shows the relative stability (ratio of HPL3 value divided by HPL3 value for LUPP)

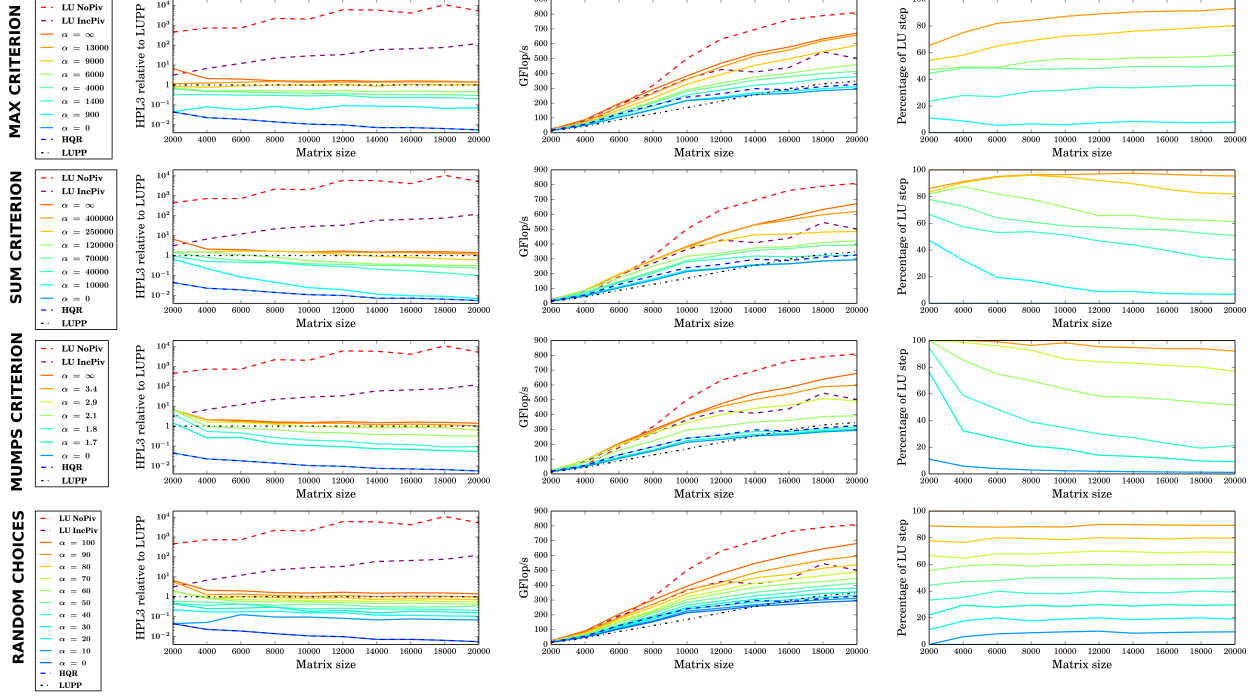


Figure 2: Stability, performance, and percentage of LU steps obtained by the three criteria and by random choices, for random matrices, on the Dancer platform (4x4 grid).

- the second column shows the GFLOP/sec performance
- the third column shows the percentage of LU steps during execution

The fourth row corresponds to a random choice between LU and QR at each step, and is intended to assess the performance obtained for a given ratio of LU vs QR steps. Plotted 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.

LU-QR Algorithm with $\alpha = \infty$ and LU NoPiv both process only LU steps. The only difference between both algorithms in term of error analysis is that LU NoPiv seeks for a pivot in the diagonal tile, while *LU-QR Algorithm* with $\alpha = \infty$ seeks for a pivot in the diagonal domain. This difference has a considerable impact in term of stability, in particular on random

Algorithm	α	Time	% LU steps	Fake GFLOP/sec	True GFLOP/sec	Fake % Peak Perf.	True % Peak Perf.
LU NoPiv		6.29	100.0	848.6	848.6	77.8	77.8
LU IncPiv		9.25	100.0	576.4	576.4	52.9	52.9
LUQR (MAX)	∞	7.87	100.0	677.7	677.7	62.1	62.1
LUQR (MAX)	13000	7.99	94.1	667.7	707.4	61.2	64.9
LUQR (MAX)	9000	8.62	83.3	619.0	722.2	56.8	66.2
LUQR (MAX)	6000	10.95	61.9	486.9	672.4	44.6	61.7
LUQR (MAX)	4000	12.43	51.2	429.0	638.4	39.3	58.5
LUQR (MAX)	1400	13.76	35.7	387.6	636.9	35.5	58.4
LUQR (MAX)	900	16.39	11.9	325.4	612.0	29.8	56.1
LUQR (MAX)	0	18.05	0.0	295.5	590.9	27.1	54.2
HQR		16.01	0.0	333.1	666.1	30.5	61.1
LUPP		15.30	100.0	348.6	348.6	32.0	32.0

Table II: Performance obtained by each algorithm, for $N = 20,000$, on the Dancer platform (4×4 grid). We only show the results for the *LU-QR Algorithm* with the Max criterion. The other criteria have similar performance. In column Fake GFLOP/sec, we assume all algorithms perform $\frac{2}{3}N^3$ floating-point operations. In column True GFLOP/sec, we compute the number of floating-point operations to be $(\frac{2}{3}f_{LU} + \frac{4}{3}(1 - f_{LU}))N^3$, where f_{LU} is the fraction of the steps that are LU steps (column 4).

matrices. *LU-QR Algorithm* with $\alpha = \infty$ has a stability slightly inferior to that of LUPP and significantly better to that of LU NoPiv. When the matrix size increases, the relative stability results of the *LU-QR Algorithm* with $\alpha = \infty$ tends to 1, which means that, on random matrices, processing an LU factorization with partial pivoting on a diagonal domain followed by a direct elimination without pivoting for the rest of the panel is almost as stable as an LU factorization with partial pivoting on the whole panel. A hand-waving explanation would go as follows. The main instabilities are proportional to the small pivots encountered during a factorization. Using diagonal pivoting, as the factorization of the diagonal tile proceeds, one is left with fewer and fewer choices for a pivot in the tile. Ultimately, for the last entry of the tile in position (n_b, n_b) , one is left with no choice at all. When working on random matrices, after having performed several successive diagonal factorizations, one is bound to have encountered a few small pivots. These small pivots lead to a bad stability. Using a domain (made of several tiles) for the factorization significantly increases the number of choice for the pivot and it is not any longer likely to encounter a small pivot. Consequently diagonal domain pivoting significantly increases the stability of the *LU-QR Algorithm* with $\alpha = \infty$. When the local domain gets large enough (while being significantly less than N), the stability obtained on random matrices is about the same as partial pivoting.

When $\alpha = \infty$, our criterion is deactivated and our algorithm always performs LU step. We note that, when α is reasonable, (as opposed to $\alpha = \infty$), the algorithm is stable whether we use a diagonal domain or a diagonal tile. However using a diagonal domain increases the

chance of well-behaved pivot tile for the elimination, therefore using a diagonal domain (as opposed to a diagonal tile) increases the chances of an LU step.

Using random choices leads to results comparable to those obtained with the three criteria. However, since we are using random matrices in this experiment set, we need to be careful before drawing any conclusion on the stability of our algorithms. If an algorithm is not stable on random matrices, this is clearly bad. However we cannot draw any definitive conclusion if an algorithm is stable for random matrices.

Table II displays detailed performance results for the Max criterion with $N = 20,000$. In column Fake GFLOP/sec, we assume all algorithms perform $\frac{2}{3}N^3$ floating-point operations. In column True GFLOP/sec, we compute the number of floating-point operations to be $(\frac{2}{3}f_{LU} + \frac{4}{3}(1 - f_{LU}))N^3$, where f_{LU} is the fraction of the steps that are LU steps (column 4). For this example, we see that the *LU-QR Algorithm* reaches a peak performance of 677.7 GFLOP/sec (62.1% of the theoretical peak) when every step is an LU step. Comparing *LU-QR Algorithm* with $\alpha = 0$ and HQR shows the overhead for the decision making process is 12.7%. We would like the true performance to be constant as the number of QR steps increases. For this example, we see only a slight decrease in performance, from 62.1% for $\alpha = \infty$ to 54.2% for $\alpha = 0$. HQR maintains nearly the same true performance (61.1%) as the *LU-QR Algorithm* with $\alpha = \infty$. Therefore, the decrease in true performance is due largely to the overhead of restoring the panel.

C. Results for special matrices

For random matrices, we obtain a good stability with random choices, almost as good as with the three criteria. However, as mentioned above, we should draw no definite conclusion. To highlight the need for a smart criterion, we tested the hybrid *LU-QR Algorithm* on a collection of matrices that includes several pathological matrices on which LUPP fails because of large growth factors. This set of special matrices described in Table III includes ill-conditioned matrices as well as sparse matrices, and mostly comes from the Higham’s Matrix Computation Toolbox [10].

Figure 3 provides the relative stability (ratio of HPL3 divided by HPL3 for LUPP) obtained by running the hybrid *LU-QR Algorithm* on a set of 5 random matrices and on the set of special matrices. 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). 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 point out that we also experimented with the Fiedler matrix from Higham’s Matrix Computation Toolbox [10]. 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 of $(1 + \alpha)^{n-1}$ on the growth is quite pessimistic).

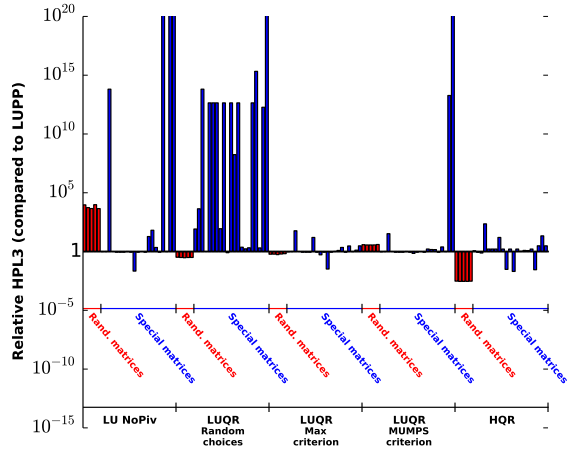


Figure 3: Stability on special matrices.

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 of the algorithm. These algorithms are unconditionally stable, and their parallelization has been fairly well studied on shared memory systems [1], [3], [17] and on parallel distributed systems [8].

The idea of mixing Gaussian transformations and orthogonal transformations has been considered once before. Irony and Toledo [18] present an algorithm for reducing a banded symmetric indefinite matrix to diagonal form. The algorithm uses symmetric Gaussian transformations and Givens rotations to maintain the banded symmetric structure and maintain similar stability to partial symmetric pivoting.

The reason for using LU kernels instead of QR kernels is performance: (i) LU performs half the number of floating-point 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

No.	Matrix	Description
1	house	Householder matrix, $A = eye(n) - \beta * v * v$
2	parter	Parter matrix, a Toeplitz matrix with most of singular values near Π . $A(i, j) = 1/(i - j + 0.5)$.
3	ris	Ris matrix, matrix with elements $A(i, j) = 0.5/(n - i - j + 1.5)$. The eigenvalues cluster around $-\Pi/2$ and $\Pi/2$.
4	condex	Counter-example matrix to condition estimators.
5	circul	Circulant matrix
6	hankel	Hankel matrix, $A = hankel(c, r)$, where $c = randn(n, 1)$, $r = randn(n, 1)$, and $c(n) = r(1)$.
7	compan	Companion matrix (sparse), $A = compan(randn(n + 1, 1))$.
8	lehmer	Lehmer matrix, a symmetric positive definite matrix such that $A(i, j) = i/j$ for $j \geq i$. Its inverse is tridiagonal.
9	dorr	Dorr matrix, a diagonally dominant, ill-conditioned, tridiagonal matrix (sparse).
10	demmel	$A = D * (eye(n) + 10 - 7 * rand(n))$, where $D = diag(1014 * (0 : n - 1)/n)$.
11	chebvand	Chebyshev Vandermonde matrix based on n equally spaced points on the interval $[0, 1]$.
12	invhess	Its inverse is an upper Hessenberg matrix.
13	prolate	Prolate matrix, an ill-conditioned Toeplitz matrix.
14	cauchy	Cauchy matrix.
15	hilb	Hilbert matrix with elements $1/(i + j - 1)$. $A = hilb(n)$.
16	lotkin	Lotkin matrix, the Hilbert matrix with its first row altered to all ones.
17	kahan	Kahan matrix, an upper trapezoidal matrix.
18	ortho	Symmetric eigenvector matrix: $A(i, j) = sqrt(2/(n + 1)) * sin(i * j * \pi/(n + 1))$
19	wilkinson	Matrix attaining the upper bound of the growth factor of GEPP.
20	foster	Matrix arising from using the quadrature method to solve a certain Volterra integral equation.
21	wright	Matrix with an exponential growth factor when Gaussian elimination with Partial Pivoting is used.

Table III: Special matrices in the experiment set.

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 [19], but they have proven much more challenging to design because of stability issues. In the following, we review several approaches:

A. 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.

B. LU NoPiv

The most basic communication-avoiding LU algorithm is LU NoPiv. This algorithm is stable for block diagonal dominant matrices [4], [10], 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. [20] 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. It is hard to be satisfied with this approach [20] because 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. Nevertheless, in practice, this proves to be a valid approach.

C. LU IncPiv

LU IncPiv is another communication-avoiding LU algorithm [2], [3]. Incremental pivoting is also called *pairwise pivoting*. The stability of the algorithm [2] 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.

D. CALU

CALU [14] is a communication-avoiding LU. It uses tournament pivoting which has been proven to be *stable in practice* [14]. CALU shares the (good) properties of one of our LU steps: (i) low number of floating-point 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. CALU is known to be stable in practice [21], [22]. Performance results of CALU in parallel distributed are presented in [21]. Performance results of CALU on a single multicore node are presented in [22].

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 it induces several short-length communications that dramatically decrease the overall performance of the factorization.

Communication-avoiding algorithms are a recent alternative which proves very relevant on today's architectures. For example, in our experiments, our HQR factorization [8] based of QR kernels ends with similar performance as ScaLAPACK LUPP while performing 2x more floating-point operations, using slower sequential kernels, and 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). For some classes of matrices (e.g., tile diagonally dominant), the *LU-QR Algorithm* will only perform LU steps.

This work opens several research directions. First, as already mentioned, the choice of the robustness parameter α is left to the user, and it would be very interesting to be able to auto-tune a possible range of values as a function of the problem and platform parameters. Second, there are many variants and extensions of the hybrid algorithm that can be envisioned. Several have been mentioned in Section II, and many others could be tried. Another goal would be to derive LU algorithms with several eliminators per panel (just as for HQR) to

decrease the critical path, provided the availability of a reliable robustness test to ensure stability.

Acknowledgment

The research of Julien Langou and Bradley R. Lowery was fully supported by the National Science Foundation grant # NSF CCF 1054864. The research of Yves Robert was supported in part by the French ANR *Rescue* project and by the Department of Energy # DOE de-sc0010682. Yves Robert is with Institut Universitaire de France. The authors thank Thomas Herault for his help with PaRSEC, and Jean-Yves L'Excellent for discussions on stability estimators within the MUMPS software.

REFERENCES

- [1] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "Parallel tiled QR factorization for multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 13, pp. 1573–1590, 2008. [Online]. Available: <http://dx.doi.org/10.1002/cpe.1301>
- [2] —, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, pp. 38–53, 2009.
- [3] 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.
- [4] J. W. Demmel, N. J. Higham, and R. S. Schreiber, "Block lu factorization," *Numerical Linear Algebra with Applications*, vol. 2, no. 2, pp. 173–190, 1995.
- [5] 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.
- [6] —, "DAGuE: A generic distributed DAG engine for high performance computing," in *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, 2011.
- [7] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *12th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC'11)*, 2011.
- [8] J. Dongarra, M. Faverge, T. Héroult, 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.

- [9] J. Bruck, C.-T. Ho, E. Upfal, S. Kipnis, and D. Weathersby, "Efficient algorithms for all-to-all communications in multiport message-passing systems," *IEEE Trans. Parallel Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, 1997.
- [10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM Press, 2002.
- [11] 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.
- [12] P. Amestoy, J.-Y. L'Excellent, and S. Pralet, "Personal communication," Jan. 2013.
- [13] 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.
- [14] 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.
- [15] J. J. Dongarra, P. Luszczek, and A. Petit, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 803–820, 2003.
- [16] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petit, K. Stanley, D. Walker, and R. Whaley, "ScaLAPACK: a portable linear algebra library for distributed memory computers — design issues and performance," *Computer Physics Communications*, vol. 97, no. 1–2, p. 1–15, 1996.
- [17] H. Bouwmeester, M. Jacquelin, J. Langou, and Y. Robert, "Tiled QR factorization algorithms," in *Proc. ACM/IEEE SC11 Conference*. ACM Press, 2011.
- [18] D. Irony and S. Toledo, "The snap-back pivoting method for symmetric banded indefinite matrices," *SIAM journal on matrix analysis and applications*, vol. 28, no. 2, pp. 398–424, 2006.
- [19] 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.
- [20] 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.
- [21] L. Grigori, J. W. Demmel, and H. Xiang, "Communication avoiding Gaussian elimination," in *Proc. ACM/IEEE SC08 Conference*, 2008.
- [22] S. Donfack, J. Dongarra, M. Faverge, M. Gates, J. Kurzak, P. Luszczek, and I. Yamazaki, "On algorithmic variants of parallel Gaussian elimination: Comparison of implementations in terms of performance and numerical properties," LAPACK Working Note 280, Jul. 2013.