J. Parallel Distrib. Comput. 85 (2015) 3–31

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Reconstructing Householder vectors from Tall-Skinny QR

G. Ballard[a], J. Demmel[b], L. Grigori[c], M. Jacquelin[d], N. Knight[b,*], H.D. Nguyen[b]

[a] Sandia National Laboratories, United States
[b] UC Berkeley, United States
[c] INRIA Paris - Rocquencourt, France
[d] Lawrence Berkeley National Laboratory, United States

## HIGHLIGHTS

- We reconstruct Householder vectors representing the $Q$-factor from Tall-Skinny QR.
- Our approach has the same asymptotic communication efficiency as TSQR.
- Additionally, it enables more communication-efficient parallel QR algorithms.
- We also provide algorithmic improvements to the Householder QR and CAQR algorithms.

## ARTICLE INFO

*Article history:*
Received 14 January 2015
Received in revised form
7 May 2015
Accepted 9 June 2015
Available online 5 August 2015

*Keywords:*
QR decomposition
Dense linear algebra
Communication-avoiding algorithms

## ABSTRACT

The Tall-Skinny QR (TSQR) algorithm is more communication efficient than the standard Householder algorithm for QR decomposition of matrices with many more rows than columns. However, TSQR produces a different representation of the orthogonal factor and therefore requires more software development to support the new representation. Further, implicitly applying the orthogonal factor to the trailing matrix in the context of factoring a square matrix is more complicated and costly than with the Householder representation.

We show how to perform TSQR and then reconstruct the Householder vector representation with the same asymptotic communication efficiency and little extra computational cost. We demonstrate the high performance and numerical stability of this algorithm both theoretically and empirically. The new Householder reconstruction algorithm allows us to design more efficient parallel QR algorithms, with significantly lower latency cost compared to Householder QR and lower bandwidth and latency costs compared with Communication-Avoiding QR (CAQR) algorithm. Experiments on supercomputers demonstrate the benefits of the communication cost improvements: in particular, our experiments show substantial improvements over tuned library implementations for tall-and-skinny matrices. We also provide algorithmic improvements to the Householder QR and CAQR algorithms, and we investigate several alternatives to the Householder reconstruction algorithm that sacrifice guarantees on numerical stability in some cases in order to obtain higher performance.

## 1. Introduction

Because of the rising costs of communication (*i.e.,* data movement) relative to computation, so-called *communication-avoiding* algorithms – ones that perform roughly the same computation as alternatives but significantly reduce communication – often run with reduced running times on today's architectures. In particular, the standard algorithm for computing the QR decomposition of a tall and skinny matrix (one with many more rows than columns) is often bottlenecked by communication costs. A more recent algorithm known as Tall-Skinny QR (TSQR) is presented in [13] (the ideas go back to [20]) and overcomes this bottleneck by reformulating the computation. In fact, the algorithm is communication optimal, attaining the lower bound for communication costs of QR decomposition (up to a logarithmic factor in the number of processors) [7]. Not only is communication reduced in theory, but the algorithm has been demonstrated to perform better on a variety

* Corresponding author.
*E-mail addresses:* gmballa@sandia.gov (G. Ballard), demmel@cs.berkeley.edu (J. Demmel), laura.grigori@inria.fr (L. Grigori), mathias.jacquelin@lbl.gov (M. Jacquelin), knight@cs.berkeley.edu (N. Knight), hdnguyen@cs.berkeley.edu (H.D. Nguyen).

http://dx.doi.org/10.1016/j.jpdc.2015.06.003
0743-7315/© 2015 Elsevier Inc. All rights reserved.

of architectures, including multicore processors [32], graphics processing units [3], and grid computing systems [1].

The standard algorithm for QR decomposition, which is implemented in LAPACK [2], ScaLAPACK [9], and Elemental [11] is known as Householder-QR (given below as Algorithm 1). For tall and skinny matrices, the algorithm works column-by-column, computing a Householder vector and applying the corresponding transformation for each column in the matrix. When the matrix is distributed across a parallel machine, this requires one parallel reduction per column. The TSQR algorithm (given below as Algorithm 6), on the other hand, performs only one reduction during the entire computation. Therefore, TSQR requires asymptotically less inter-processor synchronization than Householder-QR on parallel machines (TSQR also achieves asymptotically higher cache reuse on sequential machines).

Computing the QR decomposition of a tall and skinny matrix is an important kernel in many contexts, including standalone least squares problems, eigenvalue and singular value computations, and Krylov subspace and other iterative methods. In addition, the tall and skinny factorization is a standard building block in the computation of the QR decomposition of general (not necessarily tall and skinny) matrices. In particular, most algorithms work by factoring a tall and skinny panel of the matrix, applying the orthogonal factor to the trailing matrix, and then continuing on to the next panel. Although Householder-QR is bottlenecked by communication in the panel factorization, it can apply the orthogonal factor as an aggregated Householder transformation efficiently, using matrix multiplication [26].

The Communication-Avoiding QR (CAQR) [13] algorithm uses TSQR to factor each panel of a general matrix. One difficulty faced by CAQR is that TSQR computes an orthogonal factor that is implicitly represented in a different format than that of Householder-QR. While Householder-QR represents the orthogonal factor as a set of Householder vectors (one per column), TSQR computes a tree of smaller sets of Householder vectors (though with the same total number of nonzero parameters). In CAQR, this difference in representation implies that the trailing matrix update is done using the implicit tree representation rather than matrix multiplication as possible with Householder-QR. From a software engineering perspective, this means writing and tuning more complicated code. Furthermore, from a performance perspective, for square matrices, the trailing matrix update within CAQR is less communication efficient than the update within Householder-QR by a logarithmic factor in the number of processors.

Building on a method introduced by Yamamoto [37], we show that the standard Householder vector representation may be recovered from the implicit TSQR representation for roughly the same cost as the TSQR itself. The key idea is that the Householder vectors that represent an orthonormal matrix can be computed via LU decomposition (without pivoting) of the orthonormal matrix subtracted from a diagonal sign matrix (see Section 3). We prove in Section 6 that this reconstruction is as numerically stable as Householder-QR (independent of the matrix condition number) and validate this proof with experimental results (see Section 7).

This reconstruction method allows us to get the best of the TSQR algorithm (avoiding synchronization) as well as the best of the Householder-QR algorithm (efficient trailing matrix updates via matrix multiplication). We show that this approach improves parallel performance both in theory and practice.

Based on our parallel performance cost model given in Section 2, our algorithms exhibit asymptotic improvements over both Householder-QR and CAQR. We divide the presentation of algorithms and cost analysis across two sections. In Section 4 we consider parallel algorithms for tall-and-skinny matrices distributed over 1D processor grids, including Householder-QR, TSQR, our Householder reconstruction algorithm called TSQR-HR, and several other related approaches. In Section 5 we consider parallel algorithms for general matrices distributed over 2D processor grids. We compare our general Householder reconstruction approach (CAQR-HR) with both Householder-QR and CAQR, and we also present new improvements for each algorithm. We use a more efficient trailing matrix update within CAQR that improves both the computation and communication costs of that algorithm (see Section 5.3), and we employ a two-level aggregation technique within Householder-QR to alleviate a memory bandwidth bottleneck (see Section 5.5). The latter optimization can be applied to our new CAQR-HR as well. We demonstrate in Section 8 that these algorithmic improvements can lead to speedups on real machines and problem sizes. In particular, we demonstrate significant benefits over tuned library implementations of Householder-QR on tall-and-skinny matrices, both with 1D and tall-and-skinny 2D processor grids.

A previous version of this manuscript has appeared [5] with supplemental material in a technical report [6]. The main contributions of this work (including contributions of the previous versions) are as follows:

- the Householder reconstruction method (TSQR-HR) presented in Section 3.1,
- consideration of cheaper conditionally-stable alternatives to TSQR-HR (see Section 4.4),
- analysis of interprocessor communication and local memory bandwidth costs for all algorithms (see Sections 4–5),
- proof of correctness and numerical stability of TSQR-HR (see Section 6),
- an improvement of the trailing matrix update within CAQR (see Section 5.3),
- introduction of two-level aggregation for Householder-based trailing matrix updates (see Section 5.5), and
- performance benchmarks and evaluation of proposed algorithms and existing alternatives on two Cray supercomputers (see Section 8).

The largest observed speedup of TSQR-HR over Householder-QR (1D-pdgeqrf) for the performance experiments is about 3.6× on "Edison" (for a 1.18 million by 32 matrix distributed across 2304 processors) and about 2.7× on "Hopper" (for a 4.72 million by 32 matrix distributed across 9216 processors).

## 2. Performance cost model

In this section, we detail our algorithmic performance cost model for parallel execution on $p$ processors, each with a cache of size $Z$ words. We will use an $\alpha$–$\beta$–$\nu$–$\gamma$ model that expresses algorithmic costs in terms of computation and both interprocessor and intraprocessor communication costs. In this model, $\alpha$ corresponds to the network latency (synchronization) cost, $\beta$ to the cost of transferring a single word (inverse network bandwidth) between two processors, $\nu$ to the cost of moving a word from a local processor's memory to its cache, and $\gamma$ for computing a floating-point operation where input and output operands reside in the cache. We refer to interprocessor data movement (corresponding to $\alpha$ and $\beta$) as horizontal communication and intraprocessor data movement (main memory to and from cache, corresponding to $\nu$) as vertical communication. We assume that $\alpha > \beta > \nu > \gamma$, which is justified both by current architectures as well as architectural trends that suggest that the gaps between these cost quantities are expanding [19].

### 2.1. Horizontal communication

In our model, the cost of an interprocessor data transfer is composed of a bandwidth cost and a latency cost. We assume the cost of a message of size $w$ words is $\alpha + \beta w$, where $\alpha$ is the per-message latency cost and $\beta$ is the per-word bandwidth cost. We ignore the network topology and measure the costs in parallel, so that the cost of two disjoint pairs of processors communicating the same-sized message simultaneously is the same as that of one message. We also assume that a processor can send a message at the same time as receiving a message (from a potentially different processor). For horizontal communication costs, we will provide analysis to determine the constants on leading-order terms, but we ignore lower order terms.

Our algorithmic analysis will depend on the costs of collective communication, particularly broadcasts, reductions, and all-reductions, and we consider both tree-based and bidirectional-exchange (or recursive doubling/halving) algorithms [17,34,10]. For arrays of size $w \geq p$, the personalized collectives scatter, gather, all-gather, and reduce-scatter can all be performed with communication cost

$$T_{\text{personalized}}(w, p) = \beta \cdot w + \alpha \cdot \log p, \tag{1}$$

assuming $p$ is a power of two (reduce-scatter also incurs a computational cost of $\gamma \cdot w$). Since a broadcast can be performed with scatter and all-gather, a reduction can be performed with reduce-scatter and gather, and an all-reduction can be performed with reduce-scatter and all-gather, the communication costs of these non-personalized collectives for large arrays are twice those given by Eq. (1):

$$T_{\text{non-personalized}}(w, p) = \beta \cdot 2w + \alpha \cdot 2 \log p.$$

We note that broadcasts and reductions can in fact be done with cost

$$T_{\text{fast-broadcast}}(w, p) = \beta \cdot w + \alpha \cdot \log p + 2\sqrt{(\beta \cdot w) \cdot (\alpha \cdot \log p)}$$

via the approach in [36]. For $w \geq p \geq 1$, $T_{\text{fast-broadcast}}(w, p) \leq T_{\text{non-personalized}}(w, p)$, and they are equivalent when $\alpha \cdot \log p = \beta \cdot w$. For simplicity we use only the $T_{\text{non-personalized}}(w, p)$ cost. Most of our algorithms will rely on broadcasts and (all-)reductions. An alternative approach would be to reformulate the algorithms to leverage personalized collectives via a cyclic distribution layout as in the Elemental library [11] rather than the block-cyclic layout we employ. This reformulation could reduce the network bandwidth and latency costs by a factor of up to two.

For small arrays ($w < p$), it is not possible to use pipelined or recursive-doubling algorithms, which require the subdivision of the message into many pieces, therefore collectives such as a binomial tree broadcast must be used. In this case, the communication cost of broadcast, reduction, and all-reduction is

$$T_{\text{small-coll}}(w, p) = \beta \cdot w \log p + \alpha \cdot \log p.$$

### 2.2. Vertical communication

We model the intraprocessor communication as data movement between main memory and a cache of size $Z$ words. We assume the cost of moving a single word between memory and cache is $\nu$, which corresponds to the inverse memory bandwidth. We do not distinguish the cost of reading words from memory to cache and the cost of writing words to memory from cache, and we do not distinguish between reading contiguous or noncontiguous sets of words. Although the cost analysis of our algorithms will quantify the interprocessor communication costs and computation costs to the leading order constant, for simplicity we will not track the leading constant on the number of words moved between memory and cache.

Furthermore, in order to shorten the cost expressions, we use two assumptions on the relative magnitude of $\nu$. First, as mentioned earlier, we assume that $\nu < \beta$: interprocessor communication is more expensive than memory-to-cache transfers. With this assumption, we need to quantify only those memory bandwidth cost terms that exceed the network bandwidth cost terms. Second, we assume that $\nu < \gamma \cdot \sqrt{Z}$: a computation that performs at least $\sqrt{Z}$ floating-point operations for every word read into cache will be compute bound rather than memory bound. Using this assumption, we will ignore all memory-bandwidth costs associated with computations achieving optimal data reuse. All computations we consider will be dense linear algebra computations, namely matrix multiplication, triangular solve with multiple right hand sides, QR factorization, and LU factorization; these computations cannot achieve a reuse factor of more than $O(\sqrt{Z})$ [7]. One can compensate for ignoring these particular memory-bandwidth costs by considering $\gamma$ to be the inverse of the computation rate achieved by BLAS-3 operations rather than the inverse of the peak hardware rate.

## 3. Householder reconstruction

In this section, we explain the mathematical ideas behind Householder reconstruction and present the relevant (sequential) algorithms. We first remind the reader of the Householder-QR algorithm and then provide two interpretations of the method for reconstructing Householder vectors. We also explain a related representation of orthogonal matrices proposed by Yamamoto [37], which was the inspiration for this work.

### 3.1. Householder-QR

We first present Householder-QR in Algorithm 1, following [25] so that each Householder vector has a unit diagonal entry. Recall that a Householder vector $y$ is chosen so that the application of its corresponding Householder transformation $I - (2/y^T y)yy^T$ annihilates a subset of the entries of a matrix. We use Matlab-style pseudocode and LAPACK [2] notation for the scalar quantities.[1] However, to simplify the presentation we depart from the LAPACK code in that there is no check for a zero norm of a subcolumn. The algorithm works column-by-column, computing a Householder vector to annihilate subdiagonal entries in the column and then updating the trailing matrix to the right with the Householder transformation.

Algorithm 1 assumes real-valued inputs, but it generalizes to complex values. In the complex case, transposes should be interpreted as conjugate-transposes, and recall that for complex values, $\text{sgn}(x) = x/|x|$ for $x \neq 0$. We follow the convention $\text{sgn}(0) = 1$ for both real and complex values. In the complex case, we note that there is another difference between Algorithm 1 and the LAPACK implementation. LAPACK chooses unitary transformations that result in real-valued diagonal entries of $R$, and $\tau$ is complex-valued in general. Algorithm 1 computes *Hermitian* unitary transformations, which implies that $\tau$ is real-valued but that the diagonal entries of $R$ are complex-valued in general.

While the algorithm works for general $m$ and $b$, it is most commonly used when $m \gg b$, such as a panel factorization within a square QR decomposition. In LAPACK terms, this algorithm

[1] This notation is not to be confused with the communication cost model given in Eq. (1).

**Algorithm 1** $[Y, \tau, R]$ = Householder-QR($A$)

```
Require: $A$ is $m \times b$
1: for $i = 1$ to $b$ do
     % Compute the Householder vector
2:     $R(i,i) = -\operatorname{sgn}(A(i,i)) \cdot \|A(i:m,i)\|_2$
3:     $\tau(i) = \frac{R(i,i)-A(i,i)}{R(i,i)}$
4:     $Y(i+1:m,i) = \frac{1}{A(i,i)-R(i,i)} \cdot A(i+1:m,i)$
     % Apply the Householder transformation to the trailing matrix
5:     $z = \tau(i)\cdot[A(i,i{+}1:b)+A(i{+}1:m,i)^T \cdot A(i{+}1:m,i{+}1:b)]$
6:     $R(i,i+1:b) = A(i,i+1:b) - z$
7:     $A(i{+}1:m,i{+}1:b) = A(i{+}1:m,i{+}1:b) - Y(i{+}1:m,i)\cdot z$
8: end for
Ensure: $A = \left(\prod_{i=1}^{n}(I-\tau_i y_i y_i^T)\right) R$ where $R$ is upper-triangular and
   $Y$ (the Householder vectors) has implicit unit diagonal, while $\tau$
   is an array of length $b$ with $\tau_i = 2/(y_i^T y_i)$
```

corresponds to `geqr2` and is used as a subroutine in `geqrf`. In this case, we also compute an upper triangular matrix $T$ so that

$$Q = \prod_{i=1}^{n}(I - \tau_i y_i y_i^T) = I - YTY^T,$$

which allows the application of $Q^T$ to the trailing matrix to be done efficiently using matrix multiplication. This representation of $Q$ as $I - YTY^T$ is known as the "compact WY representation" [26] because it was developed as a storage improvement over the "WY representation" [24], $Q = I - WY^T$ (where $W = YT$). Computing $T$ from $Y$ is done in LAPACK with `larft` (so it does not need to be stored either). However, assuming the conventions of Algorithm 1 are followed, $T^{-1}$ can also be computed from $Y^TY$ by solving the equation $Y^TY = T^{-1} + T^{-T}$ for $T^{-1}$. Since $Y^TY$ is symmetric and $T^{-1}$ is triangular, the off-diagonal entries are equivalent and the diagonal entries differ by a factor of 2 [29].

### 3.2. Householder reconstruction

We now present the main mathematical idea behind Householder vector reconstruction. The key insight is that the Householder vectors that transform a matrix $A$ to an upper triangular matrix $R$ can be computed more cheaply if $R$ is known ahead of time. We assume here that $A$ is full rank and ignore roundoff for the purposes of exposition; correctness and numerical stability of our algorithms are more carefully considered in Section 6. In this section, we explain two interpretations of the algorithm that computes a set of Householder vectors from a pair of matrices $A$ and $R$ (such that $A = QR$ for some orthogonal matrix $Q$). One can think of performing the Householder-QR algorithm on $A$ using "hints" from the $R$ matrix, or one can think of performing an LU decomposition of $A - R$.

#### 3.2.1. Householder-QR with hints

Algorithm 2 presents an algorithm that differs from Householder-QR (Algorithm 1) in only four lines. Lines 2 and 6 of Algorithm 1 are removed, as $R$ is already given. In line 5, instead of computing a matrix–vector product, that vector is computed from the corresponding row of $R$. Line 3 is necessary to handle the sign ambiguity in QR decompositions; Algorithm 2 outputs a diagonal sign matrix so that $Y$ matches the output of Householder-QR even if the input $R$ reflected different sign choices.

To see how $R$ can be used as hints throughout Algorithm 2, consider an $R$ that matches the sign choices of Householder-QR. Let $A^{(i)}$ be the partially factored matrix whose first $i$ columns form an upper triangular matrix. Since $Y$ is a lower triangular matrix, the $i$th row of the trailing matrix is not updated after the $i$th Householder reflection is applied. This implies that the $i$th row of $A^{(i)}$ is equal to the $i$th row of the final output; that is, $A^{(i)}(i,:) = R(i,:)$.

In order to compute the Householder vector $y_i$, we must compute the norm of $A^{(i-1)}(i:m,i)$ and scale each entry below the diagonal by the reciprocal of the norm. Computing this norm requires a reduction, but we can avoid this calculation because the norm is given by the absolute value of the diagonal entry $R(i,i)$.

Further, after the Householder vector $y_i$ is computed, the orthogonal reflector is applied to the trailing matrix in the form of a rank-one update. This update is given by

$$\begin{aligned} A^{(i)} &= \left(I - \tau_i y_i y_i^T\right) A^{(i-1)} = A^{(i-1)} - y_i \left(\tau y_i^T A^{(i-1)}\right) \\ &= A^{(i-1)} - y_i z_i \end{aligned} \tag{2}$$

where $y_i$ is the $i$th column of $Y$ and $\tau_i = 2/(y_i^T y_i)$. Note that the $i$th entry of $y_i$, or $Y(i,i)$, is always 1. Then by Eq. (2), we have

$$R(i,:) = A^{(i)}(i,:) = A^{(i-1)}(i,:) - y_i(i)\cdot z_i = A^{(i-1)}(i,:) - z_i$$

so that $z_i = A^{(i-1)}(i,:) - R(i,:)$. This implies that $z_i$ can be computed via subtraction rather than from the formula $z_i = \tau y_i^T A^{(i-1)}$, thereby avoiding the matrix–vector product.

**Algorithm 2** $[Y, \tau, S]$ = Householder-QR-with-Hints($A$, $R$)

```
Require: $A$ is $m \times b$, $R$ is upper triangular such that $R = Q^TA$ for
   some orthogonal $Q$
1: for $i = 1$ to $b$ do
     % Compute the Householder vector to annihilate sub-diagonal
     entries in the ith column
2:     $\tau(i) = \frac{R(i,i)-A(i,i)}{R(i,i)}$
3:     $S(i,i) = -\operatorname{sgn}(A(i,i))/\operatorname{sgn}(R(i,i))$ % Store sign information
4:     $Y(i+1:m,i) = \frac{1}{A(i,i)-R(i,i)} \cdot A(i+1:m,i)$
     % Compute z from the ith row of R
5:     $z = A(i,i+1:b) - S(i,i)\cdot R(i,i+1:b)$
     % Apply the Householder transformation to the trailing matrix
6:     $A(i{+}1:m,i{+}1:b) = A(i{+}1:m,i{+}1:b) - Y(i{+}1:m,i)\cdot z$
7: end for
Ensure: $A = \left(\prod_{i=1}^{b}(I-\tau(i)y_iy_i^T)\right)\begin{bmatrix} SR \\ 0 \end{bmatrix}$, where $y_i$ is $i$th column
   of $Y$; $\tau$ is an array of length $b$ with $\tau_i = 2/(y_i^Ty_i)$; $S$ is $b \times b$
   diagonal sign matrix
```

#### 3.2.2. LU decomposition of $A - R$

A second way to interpret Algorithm 2 is as an LU decomposition. Suppose there exists a pair of matrices $A$ and $R$ with QR decomposition $A = (I - YTY^T)R$, for some $Y$ (and upper triangular $T$ that depends only on $Y$). Then to compute $Y$, we can rearrange the equation to be

$$A - R = Y(-TY^TR). \tag{3}$$

Since $T$, $Y^T$, and $R$ are upper triangular, their product is also upper triangular. Further, since $Y$ is unit lower triangular, this equation gives the unique LU decomposition of $A - R$. That is, by performing LU on $A - R$, the Householder vectors are given by the computed lower triangular factor. Note that if $A$ is tall and skinny, then because of the sparsity of $R$, we have $Y^TR = Y_1^TR_1$, where $Y_1$ and $R_1$ and square submatrices. Fig. 1 illustrates the shapes of the relevant matrices in this compact representation.

However, due to both sign ambiguity and roundoff issues, we cannot expect to compute accurate Householder vectors for all pairs of matrices $A$ and $R$ using LU($A - R$), even if $Q^TA = R$ for an exactly orthogonal $Q$. Thus, in Algorithm 3, we restrict our attention to input matrices that are orthonormal. In this case, the related upper triangular factor $R$ must be diagonal with entries of unit

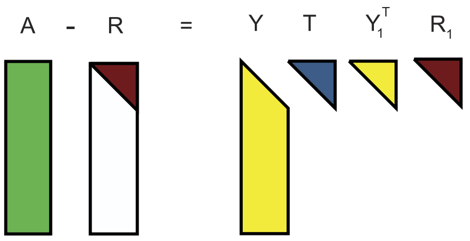


**Fig. 1.** Illustration of the matrix shapes in the LU decomposition of $A - R$ as given in Eq. (3).

modulus, reflecting only sign ambiguities. The key idea of our reconstruction algorithm in practice is that performing Householder-QR on an orthonormal matrix $Q$ is the same as performing an LU decomposition on $Q - S$, where $S$ is a diagonal sign matrix corresponding to the sign choices made inside the Householder-QR algorithm (this claim is proved explicitly in Lemma 6.2). See Section 4.4.1 for a description of the parallel algorithm for Modified-LU for general $A$ and $R$ that sacrifices numerical stability for better parallel performance.

Not only does the assumption that the input be orthonormal guarantee numerical stability, it also simplifies the algorithm. Ignoring lines 2 and 3, Algorithm 3 is exactly LU decomposition without pivoting. Note that with the choice of $S$, no pivoting is required since the effective diagonal entry will be at least 1 in absolute value and all other entries in the column are bounded by 1 (the matrix is orthonormal). This holds true throughout the entire factorization because the trailing matrix remains orthonormal, which we prove within Lemma 6.2.

```
Algorithm 3 [L, U, S] = Modified-LU(Q)
Require: Q is m × b orthonormal matrix
1: for $i = 1$ to $b$ do
2:     $S(i,i) = -\text{sgn}(Q(i,i))$
       % Set ith row of U
3:     $U(i,i) = Q(i,i) - S(i,i)$
4:     $U(i, i+1:b) = Q(i, i+1:b)$
       % Scale ith column of L by diagonal element
5:     $L(i+1:m, i) = \frac{1}{U(i,i)} \cdot Q(i+1:m, i)$
       % Perform Schur complement update
6:     $Q(i+1:m, i+1:b) = Q(i+1:m, i+1:b) - L(i+1:m, i) \cdot U(i, i+1:b)$
7: end for
Ensure: L is lower triangular with implicit unit diagonal, U is
    upper triangular, and S is diagonal so that $Q - S = LU$
```

The lower triangular output $L$ is exactly the Householder vector matrix $Y$. We note also that the upper triangular output $U$ of Algorithm 3 is equal to a product of matrices, including $T$ where $T^{-1} + T^{-T} = Y^TY$. Thus, we can compute $T = -US^{-1}Y_1^{-T}$ using triangular solve with multiple right hand sides rather than computing it from $Y^TY$. Our final algorithm applied to an $m \times b$ matrix $A$ consists of (1) computing $A = QR$ using TSQR, where $Q$ is $m \times b$, (2) applying Algorithm 3 to $Q$ to compute $L(=Y)$, $U$, and $S$, (3) computing $T$ from $U$, and (4) outputting the decomposition $A = (I - YTY_1^T)(SR)$.

### 3.3. Yamamoto's basis-kernel representation

The main goal of this work is to combine Householder-QR with CAQR; Yamamoto [37] proposes a scheme to achieve this. Motivated in part to improve the performance and programmability of a hybrid CPU/GPU implementation, Yamamoto suggests computing a representation of the orthogonal factor that triangularizes the panel that mimics the representation in Householder-QR.

As described by Sun and Bischof [33], there are many so-called "basis-kernel" representations of an orthogonal matrix. See also [30] for general discussion of block reflectors. For example, the Householder-QR algorithm computes a lower triangular matrix $Y$ such that $A = (I - YTY_1^T)R$, so that

$$Q = I - YTY^T = I - \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} T \begin{bmatrix} Y_1^T & Y_2^T \end{bmatrix}. \tag{4}$$

Here, $Y$ is called the "basis" and $T$ is called the "kernel" in this representation of the square orthogonal factor $Q$. However, there are many such basis-kernel representations if we do not restrict $Y_1$ and $T$ to be lower and upper triangular matrices, respectively.

Yamamoto [37] chooses a basis-kernel representation that is easy to compute. For an $m \times b$ matrix $A$, let $A = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} R$ where $Q_1$ and $R$ are $b \times b$. Then define the basis-kernel representation

$$Q = I - \tilde{Y}\tilde{T}\tilde{Y}^T = I - \begin{bmatrix} Q_1 - I \\ Q_2 \end{bmatrix} \left[I - Q_1\right]^{-T} \begin{bmatrix} (Q_1 - I)^T & Q_2^T \end{bmatrix}, \tag{5}$$

where $I - Q_1$ is assumed to be nonsingular. It can be easily verified that $Q^TQ = I$ and $Q^TA = \begin{bmatrix} R \\ 0 \end{bmatrix}$; in fact, this is the representation suggested and validated by [8, Theorem 3]. Note that both the basis and kernel matrices $\tilde{Y}$ and $\tilde{T}$ are dense.

The main advantage of basis-kernel representations is that they can be used to apply the orthogonal factor (or its transpose) very efficiently using matrix multiplication. In particular, the computational complexity of applying $Q^T$ using any basis-kernel representation is the same to leading order, assuming $Y$ has the same dimensions as $A$ and $m \gg b$. Thus, it is not necessary to reconstruct the Householder vectors; from a computational perspective, finding any basis-kernel representation of the orthogonal factor computed by TSQR will do. Note also that in order to apply $Q^T$ with the representation in Eq. (5), we need to apply the inverse of $I - Q_1$, e.g., by performing an LU decomposition of this $b \times b$ matrix and then applying the inverses of the triangular factors using triangular solves.

The assumption that $I - Q_1$ is nonsingular can be dropped by replacing $I$ with a diagonal sign matrix $S$ chosen so that $S - Q_1$ is nonsingular [38]; in this case the representation becomes

$$QS = I - \tilde{Y}\tilde{T}\tilde{Y}^T = I - \begin{bmatrix} Q_1 - S \\ Q_2 \end{bmatrix} S \left[S - Q_1\right]^{-T} \begin{bmatrix} (Q_1 - S)^T & Q_2^T \end{bmatrix}.$$

## 4. 1D parallel algorithms

In this section, we present two existing 1D algorithms – Householder-QR in Section 4.1 and TSQR in Section 4.2 – and provide analysis for the computation and communication costs. In Section 4.3 we discuss our parallel algorithm for Householder reconstruction and provide the cost analysis. We also discuss alternative approaches in Section 4.4.

Table 1 summarizes the cost analysis for all the algorithms presented in this section. It presents the cost for QR of a $m \times b$ matrix done using 1D $p$ processor grids, and assumes each processor owns $m/p$ rows of the matrix locally. Note that Householder-QR sends a factor of $\Theta(b)$ more messages and communicates up to a factor $O(b)$ more words between memory and cache than all other algorithms. Note also that we include the constant on the leading order costs for flops as well as horizontal data movement, as those vary among algorithms.

For this and subsequent sections, we denote the processor grid as $\pi$ ($\pi(r)$ is the $r$th processor in a 1D layout and $\pi(r, c)$ is in the $r$th row and $c$th column of a 2D layout). Each processor will be assigned a set of matrix blocks; for example, $\pi(r, c)$ may own

**Table 1**
Costs of QR factorization of tall-skinny $m \times b$ matrix distributed over $p$ processors in 1D fashion. We assume these algorithms are used as panel factorizations in the context of a 2D algorithm applied to an $m \times n$ matrix. Thus, costs (except for TSQR) include those of computing $T$. Note that the costs for TSQR-HR differ from the corresponding table in the previous version of this work (see [5, Table I]) as it includes the new optimization described in Section 4.3.

| | Flops | Horiz. words | Messages | Vert. words |
|---|---|---|---|---|
| Householder-QR | $\frac{3mb^2}{p} - \frac{2b^3}{3p}$ | $\frac{b^2}{2} \log p$ | $2b \log p$ | $O\left(\max\left\{\frac{mb}{p}, \frac{mb^2}{p} - \frac{Z^2p}{m}\right\}\right)$ |
| TSQR | $\frac{2mb^2}{p} + \frac{2b^3}{3} \log p$ | $\frac{b^2}{2} \log p$ | $\log p$ | $O(mb/p)$ |
| TSQR + LU($A - R$) | $\frac{3mb^2}{p} + \frac{2b^3}{3} \log p$ | $\frac{b^2}{2} \log p$ | $3 \log p$ | $O(mb/p)$ |
| Yamamoto | $\frac{4mb^2}{p} + \frac{4b^3}{3} \log p$ | $b^2 \log p$ | $2 \log p$ | $O(mb/p)$ |
| TSQR-HR-simple | $\frac{5mb^2}{p} + \frac{4b^3}{3} \log p$ | $b^2 \log p$ | $3 \log p$ | $O(mb/p)$ |
| TSQR-HR | $\frac{4mb^2}{p} + \frac{4b^3}{3} \log p$ | $b^2 \log p$ | $2 \log p$ | $O(mb/p)$ |
| CholQR-HR | $\frac{2mb^2}{p} + 2b^3$ | $3b^2$ | $2 \log p$ | $O(mb/p)$ |
| CholQR2-HR | $\frac{4mb^2}{p} + 4b^3$ | $2b^2$ | $4 \log p$ | $O(mb/p)$ |

some block $A_{ij}$. Our algorithms will be implicitly parallel over the processor indices $r$ and $c$. The block and processor grid indices will be zero-based, unlike the matrix entry indices $A(k, l)$ which are one-based. We assume throughout this section that $m \bmod b = 0$, where the size of the input matrix is $m \times b$. Further, when giving costs for non-personalized collectives (broadcast, reduce, and all-reduce) we assume that $\beta \cdot b \leq \alpha \leq \beta \cdot b^2$, which implies that it is more efficient to use latency-efficient collectives for a message of size $b$ with cost $\alpha \cdot \log p + \beta \cdot b \log p$ but a bandwidth-efficient collective for a message of size $b^2$ with cost $2\alpha \cdot \log p + \beta \cdot 2b^2$. This assumption is also consistent with the restriction on bandwidth-efficient collectives that the message size is at least equal to the number of processors for most of the values of $b$ and $p$ that appear in our benchmarking studies.

### 4.1. *Householder QR*

Both ScaLAPACK [9] and Elemental [11] libraries include 1D parallelizations of Householder-QR. Assuming a 1D distribution across $p$ processors, the parallelization of Algorithm 1 requires horizontal communication at lines 2 and 5. Algorithm 4 demonstrates how these can be done in a parallel 1D layout via two pairs of reductions and broadcasts (in fact they can be reorganized into just two all-reductions). The algorithm is defined so each $\pi(r)$ owns a variable amount of rows $m_r$, but throughout the analysis in this section we will assume each $m_r \approx m/p$ (in the context of 2D algorithms $m_r \in [m/p, m/p + b]$).

Because these two all-reductions occur for each column in the matrix, the total latency cost of Algorithm 4 is $2b \log p$. This synchronization cost is a potential parallel scaling bottleneck, since it grows with the number of columns of the matrix and does not decrease with the number of processors. The bandwidth cost is dominated by the all-reduction of the $z$ vector, which has length $b - i$ at the $i$th step. Thus, the bandwidth cost of Householder-QR is $(b^2/2) \log p$. Note the all-reduction of such small vectors precludes the use of a more efficient collective. The computation within the algorithm is load balanced when each $m_r \approx m/p$, for a parallel cost of $(2mb^2 - 2b^3/3)/p$ flops.

The memory-bandwidth cost is dominated by the matrix–vector multiply of the local contribution to the $z$ vector as well as the rank-one update of the local trailing matrix. If the entire local matrix fits in cache ($m_r b \leq Z$), then the memory bandwidth cost is that of reading and writing the local matrix, $O(mb/p)$ words. If

**Algorithm 4** $[Y, R] = $ 1D-Householder-QR$(A, \pi)$

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns
    a subset of $m_r \geq b$ rows, $A_r$
1: for $i = 1$ to $b$ do
       % Compute the Householder vector
2:     A reduction from $\pi(:)$ to $\pi(0)$ computes $\|A(i:m, i)\|_2$
3:     $\pi(0)$ computes $R(i, i) = -\text{sgn}(A_0(i, i)) \cdot \|A(i:m, i)\|_2$
4:     $\pi(0)$ sets $R(i, i+1:b) = A_0(i, i+1:b)$
5:     $\pi(0)$ broadcasts $R(i, i)$ and $A_0(i, i)$ to all $\pi(:)$
6:     % Define range of local active rows on processor $r$
7:     Let $\text{rows}_0 = i+1 : m_0$ and $\text{rows}_r = 1 : m_r$ for $r \geq 1$
8:     $\pi(r)$ computes $Y_r(\text{rows}_r, i) = \frac{1}{A_0(i,i) - R(i,i)} \cdot A_r(\text{rows}_r, i)$
       % Apply the Householder transformation to the trailing matrix
9:     $\pi(:)$ compute $\tau(i) = \frac{R(i,i) - A_0(i,i)}{R(i,i)}$
10:    $\pi(r)$ computes $z_r = \tau(i) \cdot A_r(\text{rows}_r, i)^T \cdot A_r(\text{rows}_r, i+1:b)$
11:    A reduction from $\pi(:)$ to $\pi(0)$ computes $z = R(i, i+1:b) + \sum_{r=0}^{p-1} z_r$
12:    $\pi(0)$ broadcasts $z$ to $\pi(:)$
13:    $\pi(r)$ updates $A_r(\text{rows}_r, i+1:b) = A_r(\text{rows}_r, i+1:b) - A_r(\text{rows}_r, i) \cdot z$
14: end for
Ensure: $A = \left(\prod_{i=1}^{n}(I - \tau(i)Y(i:m, i)Y(i:m, i)^T)\right) R$ where $R$ is
    upper triangular and owned by $\pi(0)$, $Y$ is lower triangular and
    distributed in the same layout as $A$
```

the local matrix does not fit in cache ($m_r b > Z$), then the trailing matrix must be read from memory for each iteration until the trailing matrix is small enough to fit into cache, for a total cost of $O(mb^2/p - Z^2p/m)$ words. Thus, we write the total memory bandwidth cost as $O(\max\{mb/p, mb^2/p - Z^2p/m\})$ words. Note that if not even one column of the local matrix fits into cache ($m_r > Z$), then this expression simplifies to $mb^2/p$.

The overall cost of 1D Householder-QR is given by

$$T_{\text{1D-HhQR}}(m, b, p) = \gamma \cdot \frac{2mb^2 - 2b^3/3}{p} + \beta \cdot \frac{b^2}{2} \log p$$
$$+ \alpha \cdot 2b \log p + \nu \cdot O\left(\max\left\{mb/p, mb^2/p - Z^2p/m\right\}\right).$$

In order to efficiently use the Householder form provided by Householder-QR to perform a trailing matrix update, it is additionally necessary to form an upper-triangular matrix $T$, so that $I - YTY^T = \prod_{i=1}^{n}(I - \tau_i y_i y_i^T)$. We give a routine that computes

```
Algorithm 5 [T] = 1D-Form-T-from-Y(Y, π)
Require: $Y$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns a subset of $m_r \geq b$ rows, $Y_r$
1: $\pi(:)$ compute $Z_r = Y_r^T Y_r$
2: A reduction from $\pi(:)$ to $\pi(0)$ computes $Z = \sum_{r=0}^{p-1} Z_r$
3: $\pi(0)$ extracts upper-triangular $W$ such that $Z = W + W^T$
4: $\pi(0)$ (optionally) computes $T = W^{-1}$
Ensure: $T$ is upper triangular, is owned by $\pi(0)$, and satisfies $Y^T Y = T^{-1} + T^{-T}$.
```

this $T$ when $Y$ is in a 1D distribution, as Algorithm 5. The cost of computing $T$ from $Y$ stored in a 1D layout is dominated by the cost of forming $Y^TY$. This consists of a local multiply (using `syrk`) and a reduction, for a cost of

$$T_{\text{1D-T-fm-Y}}(m, b, p) = \gamma \cdot \frac{mb^2}{p} + \beta \cdot b^2 + \alpha \cdot 2\log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

We list the combined cost (of Householder-QR and of forming $T$) in the first row of Table 1, since the other algorithms do not require forming or provide alternative (cheaper) methods for computing $T$. We note that $T$ does not need to be inverted explicitly; $T^{-1}$ can be applied with triangular solves instead of triangular multiplication during trailing matrix updates.

### 4.2. Tall-Skinny QR (TSQR)

We present a simplified version of TSQR in Algorithm 6 with an illustration in Fig. 2: we assume the number of processors is a power of two and use a binary elimination reduction tree (TSQR can be performed with any tree, see [13, Algorithm 1]). The communication structure of Algorithm 6 is in fact a binomial tree rather than a binary tree. Note also that this algorithm is a reduction rather than an all-reduction (*i.e.*, the final $R$ resides on only one processor at the end of the algorithm). TSQR assumes the tall-skinny matrix $A$ is distributed in block row layout so that each of $p$ processors, $\pi(r)$ owns a $m_r \times n$ submatrix. Throughout the analysis, we will assume $m_r \approx m/p$. After each processor computes a local QR factorization of its submatrix (line 1), the algorithm works by reducing the $p$ remaining $n \times n$ triangles to one final upper triangular $R = Q^TA$ (lines 2–10). The $Q$ that triangularizes $A$ is stored implicitly as a tree of sets of Householder vectors, given by $\{Y_{r,k}\}$. In particular, $\{Y_{r,k}\}$ is the set of Householder vectors computed by $\pi(r)$ at the $k$th level of the tree. The $r$th leaf of tree, $Y_{r,0}$ is the set of Householder vectors which $\pi(r)$ computes by doing a local QR on its part of the initial matrix $A$.

In the case of a binary tree, every internal node of the tree consists of a QR factorization of two stacked $b \times b$ triangles (line 6). This sparsity structure can be exploited, saving a constant factor of computation compared to a QR factorization of a dense $2b \times b$ matrix. In fact, as of version 3.4, LAPACK has subroutines for exploiting this and similar sparsity structures (`tpqrt`). Furthermore, the Householder vectors generated during the QR factorization of stacked triangles have similar sparsity; the structure of the $Y_{r,k}$ for $k > 0$ is an identity matrix stacked on top of a triangle.

Because the set $\{Y_{r,k}\}$ constitutes the implicit tree representation of the orthogonal factor, it is important to note how the tree is distributed across processors since the $Q$ factor will be either applied to a matrix (see [13, Algorithm 2]) or constructed explicitly (see Section 4.2.1). For a given $Y_{r,k}$, $r$ indicates the processor number (both where it is computed and where it is stored), and $k$ indicates the level in the tree. Although $0 \leq r \leq p-1$ and $0 \leq k \leq \log p$, many of the $Y_{r,k}$ are null; for example, $Y_{r,\log p} = \emptyset$ for $r > 0$ and $Y_{p-1,k} = \emptyset$ for $k > 0$. In the case of the binary tree in Algorithm 6, processor 0 computes and stores $Y_{0,k}$ for $0 \leq k \leq \log p$ and thus requires $O(b^2 \log p)$ extra memory.

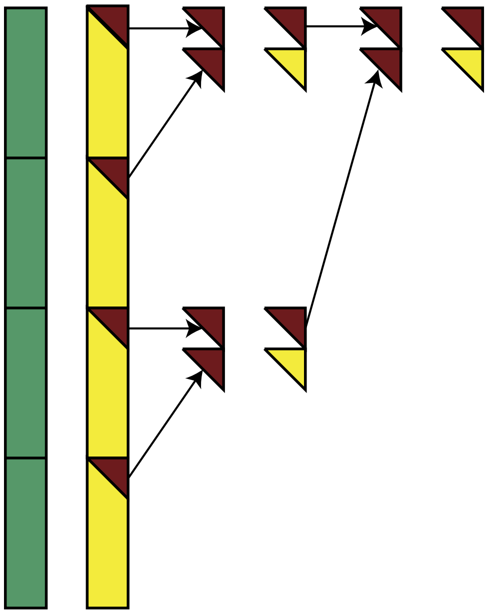


**Fig. 2.** Binary-tree TSQR on four processors. Light yellow represents Householder information ($\{Y_{r,k}\}$), dark brown represents upper triangular factors computed at each node in the tree, and arrows represent communication. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The costs and analysis of parallel TSQR (except for memory bandwidth) are given in [13,14]. We reproduce this analysis below and additionally take memory-bandwidth cost into consideration,

- The computational cost of TSQR comes from lines 1 and 6 in Algorithm 6. Line 1 corresponds to a QR factorization of a $m_r \times b$ matrix, with a flop count of $2(m/p)b^2 - 2b^3/3$ (each processor performs this step simultaneously). Line 6 corresponds to a QR factorization of a $b \times b$ triangle stacked on top of a $b \times b$ triangle. Exploiting the sparsity structure, the flop count is $2b^3/3$; this occurs at every internal node of the binary tree, so the total cost in parallel is $(2b^3/3)\log p$.
- The interprocessor communication costs of Algorithm 6 occur in lines 5 and 8. Since every $R_{r,k}$ is a $b \times b$ upper triangular matrix, the cost of a single message is $\alpha + \beta \cdot (b^2/2)$. This occurs at every internal node in the tree, so the total communication cost in parallel is a factor $\log p$ larger.
- In order to determine the memory-bandwidth costs, we assume that the sequential CAQR algorithm is used locally for both leaf and internal nodes of the tree. This implies that each leaf node requires $O(mb^2/(p\sqrt{Z}) + mb/p)$ words of vertical data movement and each internal node requires $O(b^3/\sqrt{Z} + b^2)$ words. The $mb^2/(p\sqrt{Z})$ and $b^3/\sqrt{Z}$ terms are dominated by computation, and the $b^2$ term is dominated by the horizontal bandwidth cost.

Overall, along the critical path, the costs of TSQR are:

$$T_{\text{TSQR}}(m, b, p) = \gamma \cdot \left(\frac{2mb^2}{p} + \frac{2b^3}{3}\log p\right) + \beta \cdot \left(\frac{b^2}{2}\log p\right) + \alpha \cdot \log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

We tabulate these costs in the second row of Table 1. Note that in the context of a 2D algorithm, using TSQR as the panel factorization

**Algorithm 6** $[\{Y_{r,k}\}, R] = \text{TSQR}(A, \pi)$

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns
    a subset of $m_r \geq b$ rows, $A_r$
 1: $\pi(r)$ computes $[Y_{r,0}, \bar{R}_r] = \text{Householder-QR}(A_r)$
 2: for $k = 1$ to $\lceil \log p \rceil$ do
 3:     if $r \equiv 0 \bmod 2^k$ and $r + 2^{k-1} < p$ then
 4:         $j = r + 2^{k-1}$
 5:         $\pi(r)$ receives $\bar{R}_j$ from $\pi(j)$
 6:         $\pi(r)$ computes $[Y_{r,k}, \bar{R}_r] = \text{Householder-QR}\left(\begin{bmatrix} \bar{R}_r \\ \bar{R}_j \end{bmatrix}\right)$
 7:     else if $r \equiv 2^{k-1} \bmod 2^k$ then
 8:         $\pi(r)$ sends $\bar{R}_r$ to $\pi(r - 2^{k-1})$
 9:     end if
10: end for
11: $\pi(0)$ sets $R = \bar{R}_0$
Ensure: $A = QR$ with $Q$ implicitly represented by $\{Y_{r,k}\}$
    distributed so that $R$ is stored by $\pi(0)$ and $Y_{r,k}$ is stored by $\pi(r)$
    for all $k$
```

implies that there is no $b \times bT$ matrix to compute; the update of the trailing matrix is performed differently.

Algorithm 6 can be augmented so that the $\{Y_{r,k}\}$ set of matrices are computed redundantly on multiple processors, by using a butterfly, as done in Algorithm 7. This butterfly algorithm performs redundant computation and uses redundant storage, but has the exact same cost along the critical path as Algorithm 6 and uses no more memory on any processor than $\pi(0)$ in Algorithm 6. It is effectively an all-reduction adaptation of the TSQR reduction done in the Algorithm 6. Storing the Householder vectors computed in the non-leaf nodes of the binary elimination tree (matrices $\{Y_{r,k}\}$ for $k > 0$) with some redundancy will make it possible to apply the TSQR tree to the trailing matrix in a more efficient fashion, as described in Section 5.3. The first iteration of the butterfly deals with the case of non-power of two $p$. If $p$ is not a power of two, the last $\bar{p} = p - 2^{\lfloor \log p \rfloor}$ processors swap their $R$ matrices with the first $\bar{p}$ processors, then the butterfly proceeds using the power-of-two-sized subset of processors $\pi(0 : p-\bar{p}-1)$. Handling this case is also the reason why the for loop on line 2 of Algorithm 7 is reversed.

**Algorithm 7** $[\{Y_{r,k}\}, R] = \text{Butterfly-TSQR}(A, \pi)$

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns
    a subset of $m_r \geq b$ rows, $A_r$
 1: $\pi(r)$ computes $[Y_{r,0}, \bar{R}_r] = \text{Householder-QR}(A_r)$
 2: for $k = \lceil \log p \rceil$ to 1 do
 3:     % Determine the neighbor of $\pi(r)$ for the kth level of the
    butterfly
 4:     $j = 2^k \lfloor \frac{r}{2^k} \rfloor + (r + 2^{k-1} \bmod 2^k)$
 5:     if $r < j$ and $j < p$ then
 6:         $\pi(r)$ swaps $\bar{R}_r$ for $\bar{R}_j$ with $\pi(j)$
 7:         $\pi(r)$ computes $[Y_{r,k}, \bar{R}_r] = \text{Householder-QR}\left(\begin{bmatrix} \bar{R}_r \\ \bar{R}_j \end{bmatrix}\right)$
 8:     else if $j < p$ then
 9:         $\pi(r)$ swaps $\bar{R}_r$ for $\bar{R}_j$ with $\pi(j)$
10:         $\pi(r)$ computes $[Y_{r,k}, \bar{R}_r] = \text{Householder-QR}\left(\begin{bmatrix} \bar{R}_j \\ \bar{R}_r \end{bmatrix}\right)$
11:     end if
12:     If $r \geq 2^{\lfloor \log p \rfloor}$ exit for loop after first iteration
13: end for
14: $R = \bar{R}_r$
Ensure: $A = QR$ with $Q$ implicitly represented by $\{Y_{r,k}\}$
Ensure: $R$ is stored redundantly on $\pi(1 : 2^{\lfloor \log p \rfloor})$, $Y_{r,0}$ is stored by
    $\pi(r)$, and $Y_{r,k}$ for $k \in [1, \lfloor \log p \rfloor]$ is stored redundantly on $\pi(j)$
    for each $j + 2^{k-1} = r \bmod 2^k$ and $j < 2^{\lfloor \log p \rfloor}$
```

### 4.2.1. Applying Q to a triangle

In many use cases of QR decomposition, an explicit orthogonal factor is not necessary; rather, we need only the ability to apply the matrix (or its transpose) to another matrix, as done in Section 5.2. For our purposes (see Section 4.3), we will also need to apply the orthogonal factor $Q$ to a tall-skinny upper triangular matrix (*i.e.*, an $m \times b$ matrix which is only nonzero in the upper triangle of the top $b \times b$ block). We can exploit the sparsity of such a matrix to reduce both computation and communication compared to the general case, and indeed this is the same technique as is used in forming the first $b$ columns of the explicit $m \times m$ orthogonal factor. In the case of constructing $Q$ explicitly, one can apply it to the first $b$ columns of the identity matrix (which is upper triangular); that has been previously presented (see [22, Figure 4]).[2] The structure of the algorithm is very similar to TSQR.

Algorithm 8 presents the method for applying $Q$ to an upper triangular $m \times b$ matrix $U$. Note that while we present Algorithm 8 as a binary tree, any tree shape is possible, as long as it is the same as the tree shape used in TSQR. While the nodes of the tree are computed from leaves to root, they will be applied in reverse order from root to leaves. Note that in order to minimize the computational cost, we exploit both the sparsity of $U$ at the root node and the sparsity structure of $\{Y_{r,k}\}$ at the inner tree nodes. In particular, since $U$ is upper triangular and $Y_{0,\lceil \log p \rceil}$ has the structure of identity stacked on top of an upper triangle, the output of the root node application in line 5 is a stack of two upper triangles. By induction, since every $Y_{r,k}$ for $k > 0$ has the same structure, all output $Q_{r,k}$ are triangular matrices. At the leaf nodes, when $Y_{r,0}$ is applied in line 11, the output is a dense $(m/p) \times b$ block.

Since the communicated matrices $\bar{B}_j$ are triangular just as $\bar{R}_j$ was triangular in the TSQR algorithm, Apply-TSQR-$Q$-to-Triangle incurs the identical computational and communication costs as TSQR. In particular, in lines 5 and 11 of Algorithm 8, we note that when $k > 0$, $Y_{r,k}$ is a $2b \times b$ matrix: the identity matrix stacked on top of an upper triangular matrix. Furthermore, $Q_{r,k}$ is an upper triangular matrix. Exploiting the structure of $Y_{r,k}$ and $Q_{r,k}$, the cost of line 5 is $2b^3/3$, which occurs at every internal node of the tree. Each $Y_{r,0}$ is a $(m/p) \times b$ lower triangular matrix of Householder vectors, so the cost of applying them to an upper triangular matrix in line 11 is $2(m/p)b^2 - 2b^3/3$. The computational cost of these two lines is the same as those of the previous step in Algorithm 6. Further, both the horizontal and vertical communication patterns of Algorithm 8 are also identical to Algorithm 6, so the communication cost is also the same as that of the previous step. So, we can apply $Q$ to any upper triangular $b \times b$ matrix (including the identity matrix) from the implicit form given by TSQR for the same cost as the TSQR itself,

$$T_{\text{AQtT}}(m, b, p) = T_{\text{TSQR}}(m, b, p).$$

## 4.3. TSQR with householder reconstruction

Given the algorithms of the previous sections, we now present the full approach for computing the QR decomposition of a tall-skinny matrix using TSQR and Householder reconstruction. That is, in this section we present an algorithm such that the format of the output of the algorithm is identical to that of Householder-QR. However, we argue that the latency and vertical communication costs are typically much less than those of performing Householder-QR. We first present a simple version of the algorithm (which first appeared in [5]), and then describe an optimization that improves the constants of the cost function.

[2] Similar ideas are used in LAPACK's routine `orgqr` that generates $Q$ from a set of Householder vectors.

**Algorithm 8** $B$ = Apply-TSQR-$Q$-to-Triangle($\{Y_{r,k}\}, U, \pi$)

```
Require: {$Y_{r,k}$} is a set of lower-triangular matrices computed by Algorithm 6 so that $Y_{r,k}$ is stored by $\pi(r)$ for $r \in [0, p-1]$, $Y_{r,0}$ is $m_r \times b$, while $Y_{r,k}$ for $k > 0$ is $b \times b$
Require: $U$ is $b \times b$ and upper triangular
 1: $\pi(0)$ sets $\bar{B}_0 = U$
 2: for $k = \lceil \log p \rceil$ down to 1 do
 3:     if $r \equiv 0 \bmod 2^k$ and $r + 2^{k-1} < p$ then
 4:         $j = r + 2^{k-1}$
 5:         $\pi(r)$ computes $\begin{bmatrix} \bar{B}_r \\ \bar{B}_j \end{bmatrix}$ = Apply-Householder-Q $\left(Y_{r,k}, \begin{bmatrix} \bar{B}_r \\ 0 \end{bmatrix}\right)$
 6:         $\pi(r)$ sends $\bar{B}_j$ to $\pi(j)$
 7:     else if $i \equiv 2^{k-1} \bmod 2^k$ then
 8:         $\pi(r)$ receives $\bar{B}_r$ from processor $r - 2^{k-1}$
 9:     end if
10: end for
11: $\pi(r)$ computes $B_r$ = Apply-Q-to-Triangle $\left(Y_{r,0}, \begin{bmatrix} \bar{B}_r \\ 0 \end{bmatrix}\right)$
Ensure: $B = Q^T \begin{bmatrix} U \\ 0 \end{bmatrix}$ is $m \times b$ matrix distributed in block row layout; $B_r$ is $\pi(r)$'s block
```

The simple method, given as Algorithm 9, is to perform TSQR (line 1), construct the tall-skinny $Q$ factor explicitly (line 2), and then compute the Householder vectors that represent that orthogonal factor using Modified-LU (line 3). Note that the parallel version of Modified-LU consists of performing the algorithm on the top square block on one processor, broadcasting the upper triangular factor to all processors, and computing all other rows of the lower triangular factor with independent triangular solves. The $R$ factor is computed in line 1 and the Householder vectors (the columns of $Y$) are computed in line 3. An added benefit of the approach is that the triangular $T$ matrix, which allows for block application of the Householder vectors, can be computed very cheaply. That is, a triangular solve involving the upper triangular factor from Modified-LU computes the $T$ so that $A = (I - YTY_1^T)R$. To compute $T$ directly from $Y$ (as is necessary if Householder-QR is used) requires $O(nb^2)$ flops; here the triangular solve involves $O(b^3)$ flops. Our approach for computing $T$ is given in line 4, and line 5 ensures sign agreement between the columns of the (implicitly stored) orthogonal factor and rows of $R$. As described in more detail in [5], TSQR-HR($A$) as presented in Algorithm 9, where $A$ is $m$-by-$b$, incurs the following costs (ignoring lower order terms):

$$T_{\text{TSQR-HR-simple}}(m, b, p) = \gamma \cdot \left(\frac{5mb^2}{p} + \frac{4b^3}{3} \log p\right) + \beta \cdot b^2 \log p + \alpha \cdot 4 \log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

We present in Algorithm 10 an optimized approach that reduces both communication and computation. The key to the optimization is that only the top block of $Q$ need to be formed explicitly in order to perform Modified-LU, and the final two global steps can be fused. Algorithm 10 implements the QR factorization of the tall-skinny matrix $A$ of dimension $m \times b$ as

$$A = \begin{bmatrix} A(1:b, 1:b) \\ A(b+1:m, 1:b) \end{bmatrix} = \begin{bmatrix} A(1:b, 1:b) \\ Q_1 R_1 \end{bmatrix} = \begin{bmatrix} I_b & \\ & Q_1 \end{bmatrix} \begin{bmatrix} A(1:b, 1:b) \\ R_1 \end{bmatrix} = \begin{bmatrix} I_b & \\ & Q_1 \end{bmatrix} Q_2 \tilde{R} = Q\tilde{R},$$

**Algorithm 9** $[Y, T, R]$ = TSQR-HR-simple($A, \pi$)

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns a subset of $m_r \geq b$ rows, $A_r$
 1: $[\{Y_{r,k}\}, \tilde{R}]$ = TSQR($A, \pi$)
 2: $Q$ = Apply-TSQR-$Q$-to-Triangle($\{Y_{r,k}\}, I, \pi$)
 3: $[Y, U, S]$ = Modified-LU($Q, \pi$)
 4: $\pi(0)$ computes $T = -USY_1^{-T}$
 5: $\pi(0)$ computes $R = S\tilde{R}$
Ensure: $A = (I - YTY_1^T)R$, where $Y$ is $m \times b$ unit lower triangular with the same distribution as $A$, $Y_1$ is the top $b \times b$ block of $Y$, and $T$ and $R$ are $b \times b$ and upper triangular and reside on $\pi(0)$
```

where $I_b$ is the identity matrix of dimension $b \times b$, $\tilde{R}$ is of dimension $b \times b$, $Q_1$ is of dimension $(m-b) \times b$, and $Q_2$ is of dimension $2b \times b$. The factor $Q$ of dimension $m \times b$ is

$$Q = \begin{bmatrix} Q_2(1:b, 1:b) \\ Q_1 Q_2(b+1:2b, 1:b) \end{bmatrix},$$

where $Q_2(b+1:2b, 1:b)$ is upper triangular. The factorization $A(b+1:m, 1:b) = Q_1 R_1$ is computed by using TSQR (line 1), where $Q_1$ is implicitly represented by $\{Y_{i,k}\}$ and $R_1$ is stored by processor 0. Then processor 0 computes the QR factorization of $\begin{bmatrix} A(1:b, 1:b) \\ R_1 \end{bmatrix}$ (line 2) and forms the orthogonal factor explicitly. This ensures that processor 0 has the first $b \times b$ block of $Q$ and can compute with no extra communication $Y_1$ and $U$ by using Modified-LU (line 3).

**Algorithm 10** $[Y, T, R]$ = TSQR-HR($A, \pi$)

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns a subset of $m_r \geq b$ rows, $A_r$
 1: $[\{Y_{i,k}\}, R_1]$ = TSQR($A(b+1:m, 1:b), \pi$)
 2: $\pi(0)$ computes $[Q_2, \tilde{R}]$ = Householder QR $\begin{bmatrix} A(1:b, 1:b) \\ R_1 \end{bmatrix}$
 3: $\pi(0)$ computes $[Y_1, U, S]$ = Modified-LU($Q_2(1:b, 1:b)$)
 4: $\pi(0)$ computes $W = Q_2(b+1:2b, 1:b)U^{-1}$
 5: $Y(b+1:m, :)$ = Apply-TSQR-$Q$-to-Triangle $(\{Y_{i,k}\}, W, \pi)$
 6: $\pi(0)$ computes $T = -USY_1^{-T}$
 7: $\pi(0)$ computes $R = S\tilde{R}$
Ensure: $A = (I - YTY_1^T)R$, where $Y$ is $m \times b$ unit lower triangular with the same distribution as $A$, $Y_1$ is the top $b \times b$ block of $Y$, and $T$ and $R$ are $b \times b$ and upper triangular and reside on the same processor who owns $Y_1$.
```

On $p$ processors, Algorithm 10 incurs the costs of running a TSQR, an Apply-TSQR-$Q$-to-Triangle, which we derived in Sections 4.2 and 4.2.1 respectively, a local QR and explicit construction of $Q_2$, as well as three additional costs:

1. $[Y_1, U, S]$ = Modified-LU($Q_2(1:b, 1:b)$)
2. $W = Q_2(b+1:2b, 1:b)U^{-1}$
3. $\pi(0)$ computes $T = -USY_1^{-T}$ and $R = S\tilde{R}$.

All these additional operations are performed on processor 0 and their flop counts are $O(b^3)$. Thus, the optimized TSQR-HR($A$) as presented in Algorithm 10, where $A$ is $m$-by-$b$ has the following costs (ignoring lower order terms):

$$T_{\text{TSQR-HR}}(m, b, p) = \gamma \cdot \left(\frac{4mb^2}{p} + \frac{4b^3}{3} \log p\right) + \beta \cdot b^2 \log p + \alpha \cdot 2 \log p + \nu \cdot O\left(\frac{mb}{p}\right)$$

which is listed in the third-to-last row of Table 1.

### 4.4. Alternative approaches

Alternative approaches can be used to compute the Householder vectors of a QR factorization, with varied performance and stability properties. We first discuss the LU$(A-R)$ algorithm which avoids having to reconstruct the explicit $Q$ matrix, therefore is always cheaper than TSQR-HR. However LU$(A-R)$ is not as numerically stable. Yamamoto's approach can also be used to compute a similar representation of $Q$, however it does not return results in the compact $WY$ representation [26]. Another approach is to use Cholesky-QR to compute the $R$ matrix and apply LU$(A-R)$ to compute the Householder vectors. This is the cheapest approach, but is also not as stable due to the sensitivity of Cholesky factorization to the condition number of input matrix. Finally we discuss iterative refinement techniques to improve the numerical stability of the cheaper conditionally-stable algorithms in Section 4.4.3.

#### 4.4.1. LU decomposition of $A-R$

Computing the Householder vectors basis-kernel representation via Algorithm 10 requires applying the implicit orthogonal factor to an upper triangular matrix, doubling the cost of TSQR. It is possible to compute this representation more cheaply, though at the expense of losing numerical stability. Suppose $A$ is full rank, $R=\begin{bmatrix} R_1 \\ 0 \end{bmatrix}$ is the upper triangular factor computed by TSQR, and $A-R$ is also full rank. Then there exists a Householder representation of the orthogonal factor such that $A=(I-YTY_1^T)R_1$, which implies that the unique LU decomposition of $A-R$ is given by $A-R=Y(-TY_1^TR_1)$, since $Y$ is unit lower triangular and $T, Y_1^T$, and $R_1$ are all upper triangular. The assumption that $A-R$ is full rank can be dropped by choosing an appropriate diagonal sign matrix $S$ and computing the LU decomposition of $A-\begin{bmatrix} SR_1 \\ 0 \end{bmatrix}$. We present this approach in Algorithm 11.

**Algorithm 11** $[L, U, S] = \text{Modified-LU}(A, R)$

```
Require: $A$ is $m \times b$, $R$ is upper triangular such that $\begin{bmatrix} R \\ 0 \end{bmatrix} = Q^TA$
    for some orthogonal $Q$
1: for $i = 1$ to $b$ do
2:     $S(i,i) = -\operatorname{sgn}(A(i,i))/\operatorname{sgn}(R(i,i))$
       % Set ith row of U
3:     $U(i, i+1:b) = A(i, i+1:b) - S(i,i)R(i, i+1:b)$
       % Scale ith column of L by diagonal element
4:     $L(i+1:m, i) = \frac{1}{U(i,i)} \cdot A(i+1:m, i)$
       % Perform Schur complement update
5:     $A(i+1:m, i+1:b) = A(i+1:m, i+1:b) - L(i+1:m, i) \cdot U(i, i+1:b)$
6: end for
Ensure: $L$ is lower triangular with implicit unit diagonal, $U$ is
    upper triangular, and $S$ is diagonal so that $A - \begin{bmatrix} SR \\ 0 \end{bmatrix} = LU$
```

The advantage of this approach is that $Y$ can be computed without constructing $Q$ explicitly, which is as expensive as TSQR itself. Thus the cost of this approach as detailed in the third row Table 1 is

$$T_{\text{LU(A-R)}}(m,b,p) = \gamma \cdot \left(\frac{3mb^2}{p} + \frac{2b^3}{3}\log p\right) + \beta \cdot \left(\frac{b^2}{2}\log p\right) + \alpha \cdot 3\log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

Not surprisingly, choosing the signs to match the choices of the Householder-QR algorithm applied to $A$ guarantees that the matrix is nonsingular. In fact, the signs can be computed during the course of a modified LU decomposition algorithm very similar to Algorithm 3 (designed for general matrices rather than only orthonormal). Unfortunately, this method is not numerically stable. An intuition for the instability comes from considering a low-rank matrix $A$. Because the QR decomposition of $A$ is not unique, the TSQR algorithm and the Householder-QR algorithm may compute different factor pairs $(Q_{\text{TSQR}}, R_{\text{TSQR}})$ and $(Q_{\text{Hh}}, R_{\text{Hh}})$. Performing an LU decomposition using $A$ and $R_{\text{TSQR}}$ to recover the Householder vectors that represent $Q_{\text{Hh}}$ is hopeless if $R_{\text{TSQR}} \neq R_{\text{Hh}}$, even in exact arithmetic. In floating point arithmetic, an ill-conditioned $A$ corresponds to a factor $R$ which is sensitive to roundoff error, so reconstructing the Householder vectors that would produce the same $R$ as computed by TSQR is unstable. However, if $A$ is well-conditioned, then this method is sufficient; one can view Algorithm 3 as applying the method to an orthonormal matrix (which is perfectly conditioned). See Section 7 for a discussion of experiments where the instability occurs.

One method of decreasing the sensitivity of $R$ is to employ column pivoting. That is, after performing TSQR, apply QR with column pivoting to the $b \times b$ upper triangular factor $R$, yielding a factorization $A = Q_{\text{TSQR}}(\tilde{Q}\tilde{R}P)$, or $AP = (Q_{\text{TSQR}}\tilde{Q})\tilde{R}$, where the diagonal entries of $R$ decrease monotonically in absolute value. We have observed experimentally that performing LU decomposition of $AP - \begin{bmatrix} S\tilde{R} \\ 0 \end{bmatrix}$ is a more stable operation than not using column pivoting, though it is still not as robust as Algorithm 10.

#### 4.4.2. Yamamoto's basis-kernel representation

Note that the LU factorization required in Yamamoto's approach (see Section 3.3) is equivalent to performing Modified-LU$(Q_1)$. In Algorithm 10, the Modified-LU algorithm is applied to an $m \times b$ matrix rather than to only the top $b \times b$ block; since no pivoting is required, the only difference is the update of the bottom $m-b$ rows with a triangular solve. Thus, ignoring signs, the Householder basis-kernel representation in Eq. (4) can be obtained from the representation given in Eq. (5) with two triangular solves: if the LU factorization gives $I - Q_1 = LU$, then $Y = \tilde{Y}U^{-1}$ and $T = UL^{-T}$. Indeed, performing these two operations and handling the signs correctly gives Algorithm 10.

While Yamamoto's approach avoids performing the triangular solve on $Q_2$, it still involves performing both TSQR and Apply-TSQR-Q-to-Triangle (to form $Q$ explicitly) for a total cost (as shown in the fourth row of Table 1) of

$$T_{\text{Yamamoto}}(m,b,p) = T_{\text{TSQR}}(m,b,p) + T_{\text{AQtT}}(m,b,p) = \gamma \cdot \left(\frac{4mb^2}{p} + \frac{4b^3}{3}\log p\right) + \beta \cdot b^2\log p + \alpha \cdot 2\log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

The main advantages of TSQR-HR over Yamamoto's algorithm are that the storage of the basis-kernel representation is more compact (since $Y$ is unit lower triangular and $T$ is upper triangular) and that this basis-kernel representation is backward compatible with (Sca)LAPACK and other libraries using the compact $WY$ representation [26], offering greater performance portability.

#### 4.4.3. Other related work

In this section we discuss some cheaper but less stable strategies for obtaining Householder QR representations. The Cholesky QR algorithm is a simple algorithm to compute QR factorization which is also communication optimal since it uses only the matrix multiplication kernel, and the Cholesky factorization can be done locally on a single node. However it is known that the Cholesky QR algorithm is not as numerically stable. First, the computed $Q$ factor is not guaranteed to be orthogonal. Second, Cholesky factorization is very sensitive to the condition number of input matrix [21], and it may fail when the condition number of input matrix is above $\varepsilon^{-1/2}$ where $\varepsilon$ is machine epsilon.

Fukaya et al. [18] proposed to perform the Cholesky QR algorithm twice to improve the numerical quality of computed results. The algorithm is called CholeskyQR2. It is demonstrated in the paper that the algorithm is stable in practice and in theory [39] when the condition number of input matrix is smaller than $\varepsilon^{-1/2}$, in particular, both the residual $\|A - QR\|_F/\|A\|_F$ and the orthogonality $\|Q^TQ - I\|_F/\sqrt{n}$ are close to $\varepsilon$. However CholeskyQR2 returns $Q$ in explicit format. As noted in [18], a simple modification using our proposed LU decomposition of $Q - S$ can produce results in Householder format.

**Algorithm 12** $[Y, T, R] = \text{CholQR2-HR}(A, \pi)$

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns a subset of $m_r \geq b$ rows, $A_r$ where $\text{cond}(A) < \sqrt{\varepsilon}$
1: $\pi(r)$ computes $Z_r = A_r^T A_r$
2: An all-reduction along $\pi(:)$ computes $Z = \sum_{r=0}^{p-1} Z_r$
3: $\pi(:)$ compute $R = \text{Cholesky}(Z)$
4: $\pi(r)$ computes $B_r = A_r R^{-1}$ and $\tilde{Z}_r = B_r^T B_r$
5: An all-reduction along $\pi(:)$ computes $\tilde{Z} = \sum_{r=0}^{p-1} \tilde{Z}_r$
6: $\pi(:)$ compute $R_2 = \text{Cholesky}(\tilde{Z})$
7: $\pi(:)$ compute $[\tilde{Y}, U, S] = \text{Modified-LU}(B_r, R_2)$
8: $\pi(r)$ computes $Y_r = B_r U^{-1}$
9: $\pi(:)$ compute $R = SR_2R$
10: $\pi(:)$ computes $T = -UR^{-1}\tilde{Y}^{-T}$
Ensure: $A = (I - YT\tilde{Y}^T)R$, where $Y$ is $m \times b$ unit lower triangular with the same distribution as $A$, $\tilde{Y}$ is the top $b \times b$ block of $Y$, and $T$ and $R$ are $b \times b$ and upper triangular and copies reside on each processor.
```

The $R$ factors computed by CholQR2-HR are exactly the same as those computed by CholeskyQR2. Moreover, with the assumption that the input matrix is not ill-conditioned, *i.e.*, $\text{cond}(A) < \varepsilon^{-1/2}$, the Q factor $B_r$ computed from Cholesky QR factorization of $A$ is also well-conditioned. Ideally, the condition number of $B_r$ can be close to 1 if $A$ is very well-conditioned. Therefore the Householder vectors reconstructed using the LU decomposition of $B_r - R_2$ can be stable. CholQR2-HR has almost the same total cost as CholeskyQR2, which is $4mb^2 + O(b^3)$ flops. However, both CholeskyQR2 and CholQR2-HR may not proceed to completion when the input matrix is ill-conditioned, *i.e.*, $\text{cond}(A) > \varepsilon^{-1/2}$.

CholQR2-HR requires two all-reduction operations (lines 2 and 5) to compute $B$ and $Z$. Note that we can exploit the symmetry of $B_r$ and $Z_r$ to reduce the communication cost of each all-reduction operation to $\gamma \cdot b^2 + \beta \cdot b^2 + \alpha \cdot 2\log p$. Therefore, the overall cost of CholQR2-HR is given by:

$$T_{\text{CholQR2-HR}}(m, b, p) = \gamma \cdot \left(\frac{4mb^2}{p} + 4b^3\right) + \beta \cdot 2b^2 + \alpha \cdot 4\log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

In some cases, if we know that input matrix $A$ is well-conditioned and/or computed results are not required to be of high accuracy, we can simply get $R$ from Cholesky QR factorization and use LU decomposition of $A - R$ presented above to reconstruct the Householder vectors.

CholQR-HR costs half of CholQR2-HR in terms of floating-point operations, with total cost

$$T_{\text{CholQR-HR}}(m, b, p) = \gamma \cdot \left(\frac{2mb^2}{p} + 2b^3\right) + \beta \cdot 3b^2 + \alpha \cdot 2\log p + \nu \cdot O\left(\frac{mb}{p}\right).$$

**Algorithm 13** $[Y, T, R] = \text{CholQR-HR}(A, \pi)$

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns a subset of $m_r \geq b$ rows, $A_r$ where $\text{cond}(A) < \sqrt{\varepsilon}$
1: $\pi(r)$ computes $Z_r = A_r^T A_r$
2: An all-reduction along $\pi(:)$ to $\pi(0)$ computes $Z = \sum_{r=0}^{p-1} Z_r$ and broadcasts $\tilde{A} = A(1:b, 1:b)$
3: $\pi(:)$ compute $R = \text{Cholesky}(Z)$
4: $\pi(:)$ compute $[\tilde{Y}, U, S] = \text{Modified-LU}(\tilde{A}, R)$
5: $\pi(r)$ computes $Y_r = A_r U^{-1}$
6: $\pi(:)$ compute $R = SR$
7: $\pi(:)$ compute $T = -UR^{-1}\tilde{Y}^{-T}$
Ensure: $A = (I - YT\tilde{Y}^T)R$, where $Y$ is $m \times b$ unit lower triangular with the same distribution as $A$, $\tilde{Y}$ is the top $b \times b$ block of $Y$, and $T$ and $R$ are $b \times b$ and upper triangular and copies reside on each processor.
```

One all-reduction of size $(3/2)b^2$ is performed to distribute both $Z$ and $\tilde{A}$ to all processors: processor 0 contributes $Z_0$ and $\tilde{A}$ while processor $r \neq 0$ contributes $Z_r$ and 0. CholQR-HR is the fastest of all the 1D algorithms presented in this paper. Since it uses $R$ computed directly from Cholesky QR factorization, CholQR-HR is also the least accurate algorithm.

In [4], Auckenthaler et al. proposed to dynamically adjust the block size so that the resulting matrix panels are well conditioned enough to obtain an accurate R factor from Cholesky QR algorithm. They also proposed a technique to compute Householder vectors from $A$ and $R$, which is very similar to the LU decomposition of $A - R$. The algorithm is called HouseGenk and starts with an initial heuristic block size of 16 columns. First, they check the stability of the computed R factor by $\left(\sum_{j=1}^{i-1} R_{i,j}^2\right)/R_{i,i}^2 \leq \epsilon_{fallback}$, where $\epsilon_{fallback}$ is set to 1 to avoid any substantial accuracy losses. If this criterion is not satisfied then the block size is decreased by 1. This technique is shown in the paper to work well in practice for well-conditioned matrices. For ill-conditioned matrices, HouseGenk is still able to compute a QR decomposition. However its running time increases substantially since the block size need to be changed more frequently.

We also propose two more approaches that are slightly more expensive but also more stable. Instead of performing Cholesky-QR in the first step of CholQR-HR and CholQR2-HR, we propose using TSQR and refer to the resulting algorithm as TSQR-AR. The TSQR step requires twice as many flops and a less efficient reduction operation, but guarantees the numerical stability of that step. We show in Algorithm 14 the substitution of TSQR in the CholQR2-HR approach, although the iterative refinement step could also alternatively use Cholesky-QR.

## 5. 2D parallel algorithms

In this section, we discuss 2D parallel QR algorithms that employ $p_r$-by-$p_c$ processor grids where the $m$-by-$n$ matrix $A$ is distributed block-cyclically with square $b \times b$ blocks. The 2D algorithms assume that $b \leq m/p_r, n/p_c$ and that $m \bmod b = n \bmod b = 0$. The algorithms refer to processors in the 2D $p_r$-by-$p_c$ processor grid as $\pi(i, j)$. All the algorithms employ one of the 1D algorithms from Section 4 to factor each matrix panel, then update the trailing matrix in a variety of ways. Throughout our memory-bandwidth analysis of the 2D algorithms, we assume that the local part of the matrix owned by each processor does not fit into cache (*i.e.*, $mn/p > Z$). If the entire problem fits into the processors'

**Table 2**
Costs of QR factorization of $m \times n$ matrix distributed over $p$ processors in 2D fashion. Here we assume a square processor grid ($p_r = p_c$), and we choose block sizes for each algorithm so as to minimize the costs. The computation cost for all algorithms is the same, $\frac{2mn^2-2n^3/3}{p}$. The omitted cost in the Vertical Words column for Two-Level-CAQR-HR implies all the vertical communication costs of this algorithm correspond to ideal reuse or are dominated by horizontal communication costs.

| | Horiz. words | Messages | Vert. words | Block sizes |
|---|---|---|---|---|
| 2D-Householder-QR | $\frac{2mn+n^2/4}{\sqrt{p}}$ | $n \log p$ | $O\left(\frac{mn^{3/2}}{p^{3/4}}\right)$ | $\frac{\sqrt{n}}{p^{1/4}}$ |
| Two-Level-2D-Householder-QR | $\frac{2mn+n^2/2}{\sqrt{p}}$ | $n \log p$ | $O\left(\frac{mn^{4/3}}{p^{5/6}}\right)$ | $\frac{n^{1/3}}{p^{1/3}}, \frac{n^{2/3}}{p^{1/6}}$ |
| CAQR | $\frac{2mn+n^2 \log p}{\sqrt{p}}$ | $\frac{7}{2}\sqrt{p}\log^3 p$ | $O\left(\frac{mn\log^2 p}{\sqrt{p}}\right)$ | $\frac{n}{\sqrt{p}\log^2 p}$ |
| Butterfly-CAQR | $\frac{2mn+3n^2/4}{\sqrt{p}}$ | $\frac{7}{2}\sqrt{p}\log^2 p$ | $O\left(\frac{mn\log p}{\sqrt{p}}\right)$ | $\frac{n}{\sqrt{p}\log p}$ |
| CAQR-HR | $\frac{2mn+n^2/2}{\sqrt{p}}$ | $5\sqrt{p}\log^2 p$ | $O\left(\frac{mn\log p}{\sqrt{p}}\right)$ | $\frac{n}{\sqrt{p}\log p}$ |
| Two-Level-CAQR-HR | $\frac{2mn+3n^2/4}{\sqrt{p}}$ | $5\sqrt{p}\log^2 p$ | – | $\frac{n}{\sqrt{p}\log p}, \frac{n}{\sqrt{p}}$ |

**Algorithm 14** $[Y, T, R] = \text{TSQR-AR2}(A, \pi)$

```
Require: $A$ is an $m \times b$ matrix of which $\pi(r)$ for $r \in [0, p-1]$ owns
    a subset of $m_r \geq b$ rows, $A_r$ where $\text{cond}(A) < \varepsilon^{-1}$
1: $R = \text{Butterfly-TSQR}(A, \pi)$
2: $\pi(r)$ computes $B_r = (A_r R^{-1})^T (A_r R^{-1})$
3: An all-reduction along $\pi(:)$ computes $B = \sum_{r=0}^{p-1} B_r$
4: $\pi(:)$ compute $R_2 = \text{Cholesky}(B)$
5: $\pi(:)$ compute $[\tilde{Y}, U, S] = \text{Modified-LU}(B, R_2)$
6: $\pi(r)$ computes $Y_r = B_r U^{-1}$
7: $\pi(:)$ compute $R = SR_2R$
8: $\pi(:)$ compute $T = -UR^{-1}\tilde{Y}^{-T}$
Ensure: $A = (I - YT\tilde{Y}^T)R$, where $Y$ is $m \times b$ unit lower triangular
    with the same distribution as $A$, $\tilde{Y}$ is the top $b \times b$ block of $Y$,
    and $T$ and $R$ are $b \times b$ and upper triangular and copies reside on
    each processor.
```

caches, then the vertical bandwidth cost is always dominated by the horizontal bandwidth cost (by the assumption $\nu < \beta$).

We first study the standard 2D block-cyclic Householder QR algorithm in Section 5.1. This algorithm has latency and memory-bandwidth overheads due to its use of 1D-Householder-QR on each matrix panel. Further, its trailing matrix update incurs a memory-bandwidth overhead when the local matrix does not fit entirely into cache. We then study the CAQR algorithm ([13, Algorithm 2]) in Section 5.2, which employs TSQR on each panel and as a result avoids the latency and memory-bandwidth overheads present in the 1D-Householder-QR algorithm. However, the trailing matrix update in CAQR incurs a $\log p$ overhead in interprocessor communication with respect to 2D-Householder-QR for square matrices. We introduce an optimization to the trailing matrix update in CAQR, which results in an algorithm we call Butterfly-CAQR that has nearly the same trailing matrix update communication cost as 2D-Householder-QR. Then we introduce CAQR-HR which employs TSQR-HR to factor each panel with roughly twice the cost of TSQR and performs the update in the exact same fashion (and cost) as 2D-Householder-QR. Lastly, we show that two-level blocking may be used to eliminate the memory-bandwidth overhead involved in the trailing matrix update of 2D-Householder-QR as well as CAQR-HR. We summarize the results of our analysis in Table 2 by listing the computation, interprocessor bandwidth, interprocessor latency, and memory-bandwidth costs of each algorithm on a square processor grid ($p_r = p_c = \sqrt{p}$).

### 5.1. *Householder QR*

We refer to the 2D algorithm that uses Householder-QR as the panel factorization as 2D-Householder-QR. In ScaLAPACK terms, this algorithm corresponds to `pxgeqrf`. The algorithm involves performing a factorization of each panel via 1D-Householder-QR (Algorithm 4), forming $T$ via Algorithm 5, followed by updating the trailing matrix.

**Algorithm 15** $[B] = \text{2D-Householder-Update}(A, Y, T, \pi)$

```
Require: $A$ is $m \times n$ and distributed block-cyclically on $p = p_r \cdot p_c$
    processors, so $\pi(r, c)$ owns $A_{rc}$. $Y$ is $m \times b$, lower-triangular,
    and distributed block-cyclically over a column of $p_r$ processors
    $\pi(:, 0)$ so that $Y_r$ is owned by $\pi(r, 0)$. $T$ is the upper-triangular
    kernel of $Y$ owned by $\pi(0, 0)$.
    % Update trailing matrix using all p processors
1: $\pi(0, 0)$ broadcasts $T$ to $\pi(:, :)$
2: $\pi(r, 0)$ broadcasts $Y_r$ across row $\pi(r, :)$
3: $\pi(r, c)$ computes $\widetilde{W}_{rc} = Y_r^T \cdot A_{rc}$
4: All-reduce $W_c = \sum_r \widetilde{W}_{rc}$ along column $\pi(:, c)$
5: $\pi(r, c)$ computes $B_{rc} = A_{rc} - Y_r \cdot T^T \cdot W_c$
Ensure: $B = (I - YTY^T)A$ is distributed in the same layout as $A$
```

Because we will use the same trailing matrix update in other 2D algorithms in this paper, we first give Algorithm 15 describing the distributed Householder update to a 2D-distributed matrix. The number of blocks each processor owns in a cyclic layout may be different. However, in the following analysis we will make the simplifying assumption that each processor owns $(m/b)/p_r$ block rows and $(n/b)/p_c$ block columns, which, more generally, yields costs that are correct to leading order as long as $b \ll m/p_r, n/p_c$. We derive the cost of this update step-by-step:

1. line 1: $\pi(0, 0)$ broadcasts $T$ to $\pi(:, :)$: $\beta \cdot b^2 + \alpha \cdot 2 \log p$
2. line 2: $\pi(r, 0)$ broadcasts $Y_r$ across row $\pi(r, :)$: $\beta \cdot \frac{2mb}{p_r} + \alpha \cdot 2 \log p_c$
3. line 3: $\pi(r, c)$ computes $\widetilde{W}_{rc} = Y_r^T \cdot A_{rc}$: $\gamma \cdot \frac{2mnb}{p} + \nu \cdot O\left(\frac{mn}{p}\right)$.
   Each block $\widetilde{W}_{rc}$ is computed on $\pi(r, c)$, using $A_{rc}$, which it owns, and $Y_r$, which it just received from $\pi(r, 0)$. Unless $A_{rc}$ fits entirely into cache and has already been loaded previously, this matrix must be read into cache, yielding an important memory-bandwidth overhead.
4. line 4: All-reduce $W_c = \sum_r \widetilde{W}_{rc}$ along column $\pi(:, c)$: $\gamma \cdot \frac{2nb}{p_c} + \beta \cdot \frac{2nb}{p_c} + \alpha \cdot 2 \log p_r$
5. line 5: $\pi(r, c)$ computes $B_{rc} = A_{rc} - Y_r \cdot T^T \cdot W_c$: $\gamma \cdot \frac{2mnb+2nb^2}{p} + \nu \cdot O\left(\frac{mn}{p}\right)$.
   Since in our case, $m \geq n$, it is faster to first multiply $T$ by $W_c$ rather than $Y_{ri}$ by $T$. When multiplying $T$ by $W_c$, we can exploit the triangular structure of $T$, to lower the flop count by a factor of two. Again, assuming $A_{rc}$ is not cache resident, this multiplication requires loading this matrix into cache and writing the result.

The overall cost of the update is the sum of the above, which is to leading order no larger than

$$T_{\text{2D-Hh-Upd}}(m,n,b,p_r,p_c) = \gamma \cdot \frac{4mnb}{p} + \beta \cdot \left(\frac{2mb}{p_r} + \frac{2nb}{p_c}\right) + \alpha \cdot (2\log p + 2\log p_r + 2\log p_c) + \nu \cdot O\left(\frac{mn}{p}\right).$$

**Algorithm 16** $[Y, R] = $ 2D-Householder-QR$(A, \pi)$

```
Require: A is m × n and distributed block-cyclically on $p = p_r \cdot p_c$ processors with block size b, so that each $b \times b$ block $A_{ij}$ is owned by $\pi(r, c)$ where $r = (i \bmod p_r)$ and $c = p_c \cdot (j{+}1 \bmod p_c)$.
1: for $i = 0$ to $n/b - 1$ do
2:     Let $\pi^{(i)}(r,c) = \pi((r{+}i) \bmod p_r, (c{+}i) \bmod p_c)$ for $r \in [0, \min(m/b - i, p_r{-}1)]$ and $c \in [0, \min(n/b{-}i, p{-}1))]$
       % Use 1D Householder QR on a matrix panel using column of $p_r$ processors
3:     $[Y_{i:m/b-1,i}, R_{ii}] = $ 1D-Householder-QR$(A_{i:m/b-1,i}, \pi^{(i)}(:,0))$
4:     $[T_i] = $ 1D-Form-T-from-Y$(Y_{i:m/b-1,i}, \pi^{(i)}(:,0))$
5:     $[A_{i:m/b-1,i+1:n/b-1}] = $ 2D-Householder-Update$(A_{i:m/b-1,i+1:n/b-1}, Y_{i:m/b-1,i}, T_i, \pi^{(i)})$
6:     Set $R_{i,i+1:n/b-1} = A_{i,i:n/b-1}$
7: end for
Ensure: $A = \left(\prod_{i=1}^{n/b}(I - Y_{:,i}T_iY_{:,i}^T)\right) R$ where Y and R are distributed in the same layout as A.
```

Using Algorithm 4 for the panel factorization and Algorithm 15 for the trailing matrix update, it is straightforward to formulate the full 2D-Householder-QR routine, Algorithm 16. The overall cost of 2D-Householder-QR, which includes panel factorizations and trailing matrix updates, is given to leading order by the sum of three costs at each loop iteration, where at iteration $i$ we denote the number of active processor rows as $p_r^{(i)} = \min(p_r, m/b - i)$, columns involved in the update as $p_c^{(i)} = \min(p_c, n/b - i - 1)$, and total processors involved in the update as $p^{(i)} = p_r^{(i)} \cdot p_c^{(i)}$,

$$T_{\text{2D-HhQR}}(m,n,b,p_r,p_c) = \sum_{i=0}^{n/b-1}\Big[T_{\text{1D-HhQR}}(m-ib, b, p_r^{(i)}) + T_{\text{1D-T-fm-Y}}(m-ib, b, p_r^{(i)}) + T_{\text{2D-Hh-Upd}}(m-ib, n-ib, b, p_r^{(i)}, p_c^{(i)})\Big].$$

The 1D algorithm costs require that the number of rows owned by each active $\pi(r)$ is $m_r = (m-ib)/p_r^{(i)}$, however, in a block-cyclic layout the number of rows of the trailing matrix that each processor owns is variable in the range $\lfloor (m/b - i)/p_r^{(i)} \rfloor b \le m_r \le \lceil (m/b - i)/p_r^{(i)} \rceil b$. The difference is lower order when $b$ is sufficiently small, so we assume $b \le m/(p_r \log p)$. Further, the update makes the same assumption about the number of columns in the updated matrix, which is again variable, so we further assume $b \le n/(p_c \log p)$. With these assumptions we can expand the above expression as

$$T_{\text{2D-HhQR}}(m,n,b,p_r,p_c) = \sum_{i=0}^{n/b-1}\Bigg[\gamma \cdot \left(\frac{2(m-ib)b^2 - 2b^3/3}{p_r^{(i)}} + \frac{(m-ib)b^2}{p_r^{(i)}} + \frac{4(m-ib)(n-ib)}{p^{(i)}}\right) + \beta \cdot \left(\frac{b^2}{2}\log p_r^{(i)} + \frac{2(m-ib)b}{p_r^{(i)}} + \frac{2(n-ib)b}{p_c^{(i)}}\right) + \alpha \cdot 2b \log p_r^{(i)} + \nu \cdot O\left(\max\left\{\frac{(m-ib)b}{p_r^{(i)}}, \frac{(m-ib)b^2}{p_r^{(i)}} - \frac{Z^2 p_r^{(i)}}{m-ib}\right\} + \frac{(m-ib)(n-ib)}{p^{(i)}}\right)\Bigg].$$

When $n/p_c, m/p_r \gg b$, a number of the computation and interprocessor bandwidth terms with $p_r^{(i)}, p_c^{(i)}$, or $p^{(i)}$ in the denominator decrease at every iteration. This is due to the fact that picking $b = n/p_c$ or $b = m/p_r$ corresponds to a blocked layout, which is not well load-balanced (the costs stay the same at each iteration, even though the active part of the matrix gets smaller). Therefore, the assumption that $b \le \frac{1}{\log p}\min(n/p_c, m/p_r)$ allows us to focus on the first iterations as well as drop $b^2$-dependent computation cost terms associated with the panel factorization at each iteration. The above expression for leading order cost therefore simplifies to

$$\begin{aligned} T_{\text{2D-Hh-QR}}(m,n,b,p_r,p_c) &= \sum_{i=0}^{n/b-1}\Bigg[\gamma \cdot \frac{4(m-ib)(n-ib)b}{p} + \beta \cdot \left(\frac{b^2}{2}\log p_r^{(i)} + \frac{2(m-ib)b}{p_r} + \frac{2(n-ib)b}{p_c}\right) + \alpha \cdot 2b\log p_r + \nu \cdot O\left(\max\left\{\frac{mn}{p}, \frac{mb^2}{p_r} - \frac{Z^2 p_r}{m}\right\}\right)\Bigg] \\ &= \gamma \cdot \frac{2mn^2 - 2n^3/3}{p} + \beta \cdot \left(\frac{nb}{2}\log p_r + \frac{2mn - n^2}{p_r} + \frac{n^2}{p_c}\right) + \alpha \cdot 2n\log p_r + \nu \cdot O\left(\max\left\{\frac{mn^2}{pb}, \frac{mnb}{p_r} - \frac{Z^2 np_r}{mb}\right\}\right). \end{aligned}$$

As a result of keeping track of memory-bandwidth, we now see that there are overheads beyond those associated with ideal reuse in each local operation or those associated with interprocessor communication (which our model ignores). The second overhead in the max, $\nu \cdot O\left(mnb/p_r - Z^2 np_r/(mb)\right)$ arises whenever the matrix panel does not fit in cache, and so can be eliminated by setting $b \le Zp_r/m$. However, the first memory-bandwidth overhead in the max, $\nu \cdot O\left(mn^2/(pb)\right)$, which is associated with the trailing matrix updates, is present whenever the local portion of the entire matrix does not fit in cache (which we expect to generally be the case), and grows inversely with the block size, since the number of trailing matrix updates increases. Choosing $b = Zp_r/m$ causes the latter overhead to become dominant in this case.

In the case when the matrix is nearly square $m \approx n$, we can pick $p_r = p_c = \sqrt{p}$, and select $b = \sqrt{n}/p^{1/4}$, which guarantees that the second term in the max does not dominate the first (although a better cost may be achieved by picking $b$ so that the panel fits into cache, $Z \ge mb/p_r$, in this case meaning $Z \ge n^{3/2}/p^{3/4}$) to obtain the leading order costs

$$T_{\text{2D-Hh-QR}}\left(m, n, \frac{n}{\sqrt{p}\log p}, \sqrt{p}, \sqrt{p}\right) = \gamma \cdot (2mn^2 - 2n^3/3)/p + \beta \cdot (2mn + n^2/4)/\sqrt{p} + \alpha \cdot n\log p + \nu \cdot O\left(\frac{mn^{3/2}}{p^{3/4}}\right).$$

Note that these latency and interprocessor communication costs match those of [13,9], with exceptions coming from the use of more efficient collectives. We present the costs of 2D-Householder-QR in the first row of Table 2. The memory-bandwidth cost associated with the trailing matrix update can be alleviated by the use of multiple levels of blocking, as we show in Section 5.5.

### 5.2. Communication-Avoiding (CAQR)

The 2D algorithm that uses TSQR for panel factorizations is known as CAQR. In order to update the trailing matrix within CAQR, the implicit orthogonal factor computed from TSQR need to be applied as a tree in the same order as it was computed. See [13, Algorithm 2] for a description of this process, or see [6, Algorithm 4] for pseudocode that matches the binary tree in Algorithm 6. We refer to this application of implicit $Q^T$ as Apply-TSQR-$Q^T$ and describe how it is done on one 1D column of $p$ processors in Algorithm 17. The algorithm has the same tree dependency flow structure as TSQR but requires a bidirectional exchange between paired nodes in the tree. We note that in internal nodes of the tree it is possible to exploit the additional sparsity structure of $Y_{r,k}$ (an identity matrix stacked on top of a triangular matrix), which our implementation does via the use of the LAPACK v3.4 + routine `tpmqrt`.

**Algorithm 17** $[B]$ = Apply-TSQR-$Q^T(A, \{Y_{r,k}\}, \pi)$

```
Require: $A$ is $m \times n$ matrix distributed over $p$ processors; $A_r$ is $\pi(r)$'s block
Require: $\{Y_{r,k}\}$ is the implicit tree TSQR representation of $m \times m$ orthogonal matrix $Q$ with each $Y_{r,k}$ having $b$ columns and stored by $\pi(r)$
1: $\pi(r)$ computes $B_r$ = Apply-Householder-$Q^T(Y_{r,0}, A_r)$
2: Let $\bar{B}_r$ be the first $b$ rows of $B_r$
3: for $k = 1$ to $\lceil \log p \rceil$ do
4:     if $r \equiv 0 \bmod 2^k$ and $r + 2^{k-1} < p$ then
5:         $j = r + 2^{k-1}$
6:         $\pi(r)$ receives $\bar{B}_j$ from $\pi(j)$
7:         $\pi(r)$ computes $\begin{bmatrix} \bar{B}_r \\ \bar{B}_j \end{bmatrix}$ = Apply-Householder-$Q^T\left(Y_{r,k}, \begin{bmatrix} \bar{B}_r \\ \bar{B}_j \end{bmatrix}\right)$
8:         $\pi(r)$ sends $\bar{B}_j$ back to $\pi(j)$
9:     else if $i \equiv 2^{k-1} \bmod 2^k$ then
10:        $\pi(r)$ sends $\bar{B}_r$ to processor $r - 2^{k-1}$
11:        $\pi(r)$ receives updated rows $\bar{B}_r$ from processor $r - 2^{k-1}$
12:        $\pi(r)$ sets the first $b$ rows of $B_r$ to $\bar{B}_r$
13:    end if
14: end for
Ensure: $B = Q^T A$ distributed in the same layout as $A$
```

In the following analysis we will again make the simplifying load-balance assumption that each processor owns exactly $m/p$ rows. We calculate the cost of Algorithm 17 step-by-step:

1. line 1: $\pi(r)$ computes $B_r$ = Apply-Householder-$Q^T(Y_{r,0}, A_r)$: $\gamma \cdot \left(\frac{4mnb}{p} - 2nb^2\right) + \nu \cdot O\left(\frac{mn}{p}\right)$.
   The application of the $Y_{r,0}$ Householder matrices formed by the leaves of the TSQR reduction tree to the trailing matrix can be done locally. However, it requires reading $Y_{r,0}$ and $A_r$ from memory, which has an associated memory-bandwidth overhead.
2. lines 6, 8, 6, and 12: $\pi(r)$ sends or receives $\bar{B}_j$ or $\bar{B}_r$: $\beta \cdot 2nb \log p + \alpha \cdot 2 \log p$.
   At every one of $\log p$ levels of the tree, $\pi(0)$ sends and receives $b$ rows, which may not be overlapped since local work must happen between the send and receive.
3. line 7: $\pi(r)$ computes $\begin{bmatrix} \bar{B}_r \\ \bar{B}_j \end{bmatrix}$ = Apply-Householder-$Q^T$ $\left(Y_{r,k}, \begin{bmatrix} \bar{B}_r \\ \bar{B}_j \end{bmatrix}\right)$: $\gamma \cdot 2nb^2 \log p$.
   At every one of $\log p$ levels of the tree, $\pi(0)$ applies the update to $2b$ rows. This step does not incur an asymptotic vertical bandwidth overhead in our model, since the same number of rows must be communicated over the network, so we keep track only of the computation cost.

Overall Algorithm 17 therefore has the cost

$$T_{\text{TSQR-AQT}}(m, n, b, p) = \gamma \cdot \left(\frac{4mnb}{p} + 2nb^2 \log p\right) + \beta \cdot 2nb \log p + \alpha \cdot 2 \log p + \nu \cdot O\left(\frac{mn}{p}\right).$$

**Algorithm 18** $[\{Y_{r,k}^{(i)}\}, R]$ = CAQR$(A, \pi)$

```
Require: $A$ is $m \times n$ and distributed block-cyclically on $p = p_r \cdot p_c$ processors with block size $b$, so that each $b \times b$ block $A_{ij}$ is owned by $\pi(r, c)$ where $r = (i \bmod p_r)$ and $c = p_c \cdot (j+1 \bmod p_c)$.
1: for $i = 0$ to $n/b-1$ do
2:     Let $\pi^{(i)}(r, c) = \pi((r+i) \bmod p_r, (c+i) \bmod p_c)$ for $r \in [0, \min(m/b - i, p_r-1)]$ and $c \in [0, \min(n/b-i, p-1))]$
       % Use 1D TSQR on a matrix panel using column of $p_r$ processors
3:     $\left[\{Y_{r,k}^{(i)}\}, R_{ii}\right]$ = TSQR$(A_{i:m/b-1,i}, \pi^{(i)}(:, 0))$
       % Broadcast implicit orthogonal matrix across processor rows
4:     $\pi^{(i)}(r, 0)$ broadcasts $Y_{r,k}^{(i)}$ to $\pi^{(i)}(r, :)$
       % Apply 1D trailing matrix updates on all processor columns
5:     Let bcols$_c = \{i + c, i + 2c, \dots, n/b - p_c + c\}$
6:     $\left[A_{i:m/b-1,\text{bcols}_c}\right]$ = Apply-TSQR-$Q^T(\{Y_{r,k}^{(i)}\}, A_{i:m/b-1,\text{bcols}_c}, \pi^{(i)}(:, c))$
7:     Set $R_{i,i+1:n/b-1} = A_{i,i:n/b-1}$
8: end for
Ensure: $A = QR$ where $Q$ is orthogonal and stored implicitly by $\{Y_{r,k}^{(i)}\}$ and $R$ is triangular and distributed in the same layout as $A$.
```

CAQR involves performing TSQR on each processor panel followed by a trailing matrix update via Algorithm 17. We present CAQR in Algorithm 18, which outputs the triangular matrix $R$ and an implicit representation of the orthogonal factor $Q$. It is possible to output $Q$ explicitly by applying each TSQR tree $\{Y_{r,k}\}$ to the identity as done in Algorithm 8, but this results in additional leading-order computation and communication terms. The overall cost of CAQR is given to leading order by the sum of TSQR and Apply-TSQR-$Q^T$ over all iterations, plus the cost of broadcasting the tree $\{Y_{r,k}^{(i)}\}$ on line 4 (which has at most $b^2\lceil(m/b - i)/p_r^{(i)}\rceil + b^2 \log p_r^{(i)}/2$ entries on processor $\pi^{(i)}(0, 0)$) where we again keep track of the number of active processor rows in $\pi^{(i)}$ as $p_r^{(i)}$, columns as $p_c^{(i)}$:

$$T_{\text{CAQR}}(m, n, b, p_r, p_c) = \sum_{i=0}^{n/b-1} \Bigg[ T_{\text{TSQR}}(m - ib, b, p_r^{(i)}) + T_{\text{TSQR-AQT}}\left(m - ib, b\left\lceil \frac{n/b - i}{p_c^{(i)}} \right\rceil, b, p_r^{(i)}\right) + \beta \cdot \left(2b^2\lceil(m/b - i)/p_r^{(i)}\rceil + b^2 \log p_r^{(i)}\right) + \alpha \cdot 2 \log p_c^{(i)} \Bigg].$$

Assuming $b \leq \frac{1}{\log p} \min(m/p_r, n/p_c)$, the leading order costs are

$$T_{\text{CAQR}}(m, n, b, p_r, p_c) = \sum_{i=0}^{n/b-1} \Bigg[ \gamma \cdot \frac{4(m - ib)(n - ib)b}{p} + \beta \cdot \left(\frac{2(n - ib)b}{p_c} \log p_r + \frac{2(m - ib)b}{p_r}\right) + \alpha \cdot (3 \log p_r + 4 \log p_c) + \nu \cdot O\left(\frac{(m - ib)(n - ib)}{p}\right) \Bigg]$$
$$= \gamma \cdot \left(\frac{2mn^2 - 2n^3/3}{p} + \frac{n^2 b}{p_c} \log p_r\right)$$

$$+\beta\cdot\left(\frac{n^2}{p_c}\log p_r+\frac{2mn-n^2}{p_r}\right)$$
$$+\alpha\cdot\left(\frac{3n}{b}\log p_r+\frac{4n}{b}\log p_c\right)+\nu\cdot O\left(\frac{mn^2}{pb}\right).$$

See [13] for a discussion of these costs and [14] for detailed analysis. Note that the bandwidth cost is slightly lower here due to the use of more efficient broadcasts. If we pick $p_r = p_c = \sqrt{p}$ (assuming $m \approx n$) and $b = \frac{n}{\sqrt{p}\log^2 p}$ then we obtain the leading order costs

$$T_{\text{CAQR}}\left(m, n, \frac{n}{\sqrt{p}\log^2 p}, \sqrt{p}, \sqrt{p}\right) = \gamma\cdot\left(\frac{2mn^2-2n^3/3}{p}\right)$$
$$+\beta\cdot\left(\frac{2mn+n^2\log p}{\sqrt{p}}\right)+\alpha\cdot\left(\frac{7}{2}\sqrt{p}\log^3 p\right)$$
$$+\nu\cdot O\left(\frac{mn\log^2 p}{\sqrt{p}}\right).$$

We choose $b$ to preserve the leading constants of the computational cost. Note that $b$ need to be chosen smaller here than in Section 5.1 due to the costs associated with Apply-TSQR-Q$^T$.

It is possible to reduce the costs of Apply-TSQR-Q$^T$ further using ideas from efficient recursive doubling/halving collectives; see Section 5.3 for more details. Another important practical optimization for CAQR is pipelining the trailing matrix updates [15], though we do not consider this idea here as it cannot be applied with the Householder reconstruction approach.

### 5.3. Butterfly-CAQR

We also present an alternative method for improving the CAQR trailing matrix update that does not reconstruct the Householder form. A major drawback with performing the update via a binary tree algorithm is heavy load imbalance. This problem may be resolved by exploiting the fact that each column of the trailing matrix may be updated independently and subdividing the columns among more processors to balance out the work. This can be done with ideal load balance using a butterfly communication network instead of a single binomial tree. The main idea of this approach is to use a recursive halving and recursive doubling approach over the columns of the trailing matrix while applying the TSQR tree. The Householder tree data may be replicated explicitly via communication or can be computed redundantly during the factorization phase via a butterfly network TSQR (see Algorithm 7) for no extra parallel cost.

Algorithm 19 uses the redundantly stored representation $\{Y_{r,k}\}$ produced by Algorithm 7 to perform the trailing matrix update more efficiently (see Fig. 3). In particular, at each recursive step of the butterfly network in Algorithm 19, the working part of the columns of the trailing matrix is partitioned in half and subdivided amongst pairs of processor rows. In the binary tree application method (Algorithm 17), half the processors are assigned work at each successive level of the tree. In the butterfly algorithm, all the processors are assigned half the work at each recursive level of the butterfly (line 15 of Algorithm 19). As a result, the number of columns each processor works on reduces in half at each successive level in the butterfly. The first block of the if statement on line 2 handles the case of a non-power of two number of processors, by performing a two processor butterfly update to complete the work needed by the processors whose ranks are higher than the nearest power of two. This preprocessing step is done at most once, as all further recursive calls will subdivide a power-of-two number of processors into two halves of the same size. We note that Algorithm 19 is different from the butterfly algorithm for the trailing matrix application presented in [22], which applies the redundant copies of $Y_{i,k}$ to the columns of the trailing matrix redundantly.

**Algorithm 19** $[B]$ = Scatter-Apply-TSQR-Q$^T$($\{Y_{r,k}\}, A, \pi$)

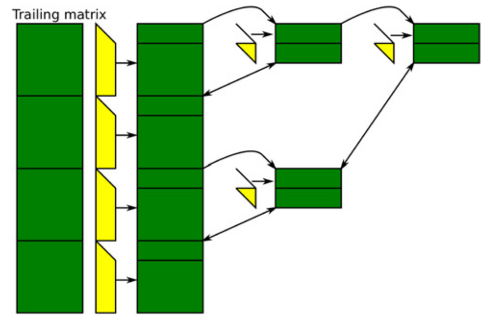
```
Require: $A$ is $m \times n$ matrix distributed over $p$ processors; $A_r$ is $\pi(r)$'s block
Require: $\{Y_{r,k}\}$ is the implicit tree Butterfly-TSQR representation of $m \times m$ orthogonal matrix $Q$ with each $Y_{r,k}$ having $b$ columns and stored by $\pi(r)$
1: Let $\bar{B}_r$ be the first $b$ rows of $B_r$
2: if $\lceil \log p \rceil \neq \lfloor \log p \rfloor$ then
3:     Let $\bar{p} = p - 2^{\lfloor \log p \rfloor}$
4:     if $r < \bar{p}$ or $r \geq p - \bar{p}$ then
5:         $[B]$ = Scatter-Apply-TSQR-Q$^T$($Y_{r,\lfloor \log p \rfloor}, A, [\pi(r \bmod \bar{p}), \pi((r \bmod \bar{p}) + \bar{p})]$)
6:     end if
7:     if $r < p - \bar{p}$ then
8:         $[B]$ = Scatter-Apply-TSQR-Q$^T$($\{Y_{r,k}\}, B, \pi(0 : p-\bar{p}-1)$)
9:     end if
10: else if $p > 1$ then
11:     Let $\bar{B}_r = [\bar{B}_{r0}, \bar{B}_{r1}]$ where each block is $b$-by-$n/2$
12:     $\rho = (r < p/2)$,   $\delta = (r \geq p/2)$,   $q = \delta \cdot p/2$
13:     $\pi(r)$ swaps $\bar{B}_{r\rho}$ with $\bar{B}_{j\delta}$ from $\pi(r + (\rho - \delta) \cdot p/2)$
14:     $\begin{bmatrix} \bar{B}_{r\delta} \\ \bar{B}_{j\delta} \end{bmatrix}$ = Apply-Householder-Q$^T$ $\left(Y_{r,\log p}, \begin{bmatrix} \bar{B}_{r\delta} \\ \bar{B}_{j\delta} \end{bmatrix}\right)$
15:     $[\bar{B}_{r\delta}]$ = Scatter-Apply-TSQR-Q$^T$($\{Y_{r,k}\}, \bar{B}_{r\delta}, \pi(q : q+p/2-1)$)
16:     $\pi(r)$ swaps $\bar{B}_{j\delta}$ with $\bar{B}_{r\rho}$ from $\pi(r + (\rho - \delta) \cdot p/2)$
17: end if
Ensure: $B = Q^T A$ is distributed the same way as $A$ and where $Q$ is the orthogonal matrix implicitly represented by $\{Y_{r,k}\}$
```

**Algorithm 20** $[R]$ = Butterfly-CAQR($A, \pi$)

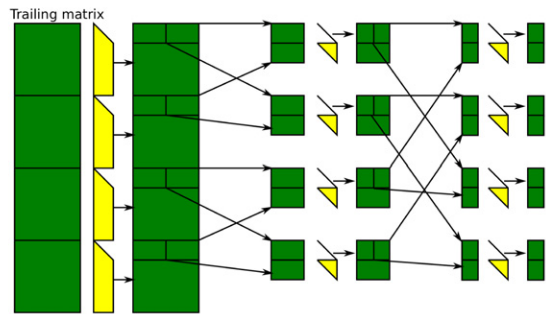
```
Require: $A$ is $m \times n$ and distributed block-cyclically on $p = p_r \cdot p_c$ processors with block size $b$, so that each $b \times b$ block $A_{ij}$ is owned by $\pi(r, c)$ where $r = (i \bmod p_r)$ and $c = p_c \cdot (j{+}1 \bmod p_c)$.
1: for $i = 0$ to $n/b-1$ do
2:     Let $\pi^{(i)}(r, c) = \pi((r{+}i) \bmod p_r, (c{+}i) \bmod p_c)$
           for $r \in [0, \min(m/b - i, p_r{-}1)]$ and $c \in [0, \min(n/b{-}i, p{-}1))]$
       % Use 1D Butterfly-TSQR on a matrix panel using column of $p_r$ processors
3:     $[\{Y_{r,k}\}, R_{ii}]$ = Butterfly-TSQR($A_{i:m/b-1,i}, \pi^{(i)}(:, 0)$)
4:     $\pi^{(i)}(r, 0)$ broadcasts $Y_{r,k}$ to $\pi^{(i)}(r, :)$
5:     Let $\text{bcols}_c = \{i + c, i + 2c, \ldots, n/b - p_c + c\}$
6:     $\pi^{(i)}(r, c)$ computes $A_{i:m/b-1,\text{bcols}_c}$ = Apply-Householder-Q$^T$($Y_{r,0}, A_{i:m/b-1,\text{bcols}_c}$)
7:     $[A_{i,\text{bcols}_c}]$ = Scatter-Apply-TSQR-Q$^T$($\{Y_{r,k}\}, A_{i,\text{bcols}_c}, \pi^{(i)}(:, c)$)
8:     Set $R_{i,i+1:n/b-1} = A_{i,i:n/b-1}$
9: end for
Ensure: $A = QR$ where $Q$ is some orthogonal matrix while $R$ is triangular and distributed in the same layout as $A$.
```

The benefit achieved by Algorithm 19 over the tree update in Algorithm 17 is a factor of $O(\log p)$ in horizontal communication and computation cost. The algorithm requires two swaps of size $nb/2$ on each recursive level on lines 13 and 16 and an application of a Householder update to a $2b \times n/2$ matrix on line 14. Given a power-of-two number of processors $p$, the cost of Algorithm 17 is given by the recurrence

$$T_{\text{rec-SAQT}}(n, b, p) = T_{\text{rec-SAQT}}(n/2, b, p/2) + \gamma\cdot nb^2$$
$$+\beta\cdot nb+\alpha\cdot 2.$$



(a) Illustration of Algorithm 17. This approach can suffer from computational and communication load imbalance.



(b) Illustration of Algorithm 19. This approach balances the computation and communication across all processors in the column but requires Householder information be stored redundantly (using Butterfly-TSQR, Algorithm 7).

**Fig. 3.** Two algorithms for applying (implicit) $Q^T$ from TSQR to the part of the trailing matrix owned by a single processor column.

Note that there is no extra memory bandwidth cost (all vertical communication costs are dominated by the computation and horizontal bandwidth costs). Since the base case requires no work, $T_{\text{rec-SAQT}}(n, b, 1) = 0$, we have

$$T_{\text{rec-SAQT}}(n, b, 2) = \gamma \cdot nb^2 + \beta \cdot nb + \alpha \cdot 2,$$

$$T_{\text{rec-SAQT}}(n, b, p) \le \gamma \cdot 2nb^2 + \beta \cdot 2nb + \alpha \cdot 2 \log p.$$

By combining this cost with the cost of the preprocessing step when $p$ is not a power of two, we obtain a bound on the leading order overall cost

$$\begin{aligned} T_{\text{SAQT}}(n, b, p) &\le T_{\text{rec-SAQT}}(n, b, 2) + T_{\text{rec-SAQT}}(n, b, 2^{\lfloor \log p \rfloor}) \\ &\le \gamma \cdot 3nb^2 + \beta \cdot 3nb + \alpha \cdot 2 \log p. \end{aligned}$$

Algorithm 20 gives the full 2D QR algorithm that employs Butterfly-TSQR on each panel and Scatter-Apply-TSQR-Q$^T$ on the trailing matrix update. It also broadcasts $\{Y_{r,k}\}$ analogously to Algorithm 18 and performs the local updates at the leaves of the TSQR tree, for a total cost of

$$\begin{aligned} T_{\text{BCAQR}}(m, n, b, p_r, p_c) = \sum_{i=0}^{n/b-1} \Bigg[ & T_{\text{Butterfly-TSQR}}(m - ib, b, p_r^{(i)}) \\ & + T_{\text{SAQT}}\left(m - ib, b\left\lceil \frac{n/b - i}{p_c^{(i)}} \right\rceil, b, p_r^{(i)}\right) \\ & + \gamma \cdot \frac{4mnb}{p^{(i)}} + \beta \cdot \left(2b^2 \lceil (m/b - i)/p_r^{(i)} \rceil + b^2 \log p_r^{(i)}\right) \\ & + \alpha \cdot 2 \log p_c^{(i)} + \nu \cdot O\left(\frac{(m - ib)(n - ib)}{p}\right) \Bigg]. \end{aligned}$$

As before, assuming $b \le \frac{1}{\log p} \min(m/p_r, n/p_c)$, the leading order cost becomes

$$\begin{aligned} T_{\text{BCAQR}}(m, n, b, p_r, p_c) = {} & \gamma \cdot \frac{2mn^2 - 2n^3/3}{p} \\ & + \beta \cdot \left(\frac{nb}{2} \log p_r + \frac{3n^2}{2p_c} + \frac{2mn - n^2}{p_r}\right) \\ & + \alpha \cdot \left(\frac{3n}{b} \log p_r + \frac{4n}{b} \log p_c\right) + \nu \cdot O\left(\frac{mn^2}{pb}\right). \end{aligned}$$

Since the computational cost associated with Scatter-Apply-TSQR-Q$^T$ is not leading order, unlike the tree update done in Algorithm 17, the block size for the square processor grid case, can be picked to be the same as in 2D-Householder-QR, namely $b = \frac{n}{\sqrt{p} \log p}$, achieving better horizontal and vertical communication as well as latency costs than CAQR,

$$\begin{aligned} T_{\text{BCAQR}}\left(m, n, \frac{n}{\sqrt{p} \log p}, \sqrt{p}, \sqrt{p}\right) = {} & \gamma \cdot \left(\frac{2mn^2 - 2n^3/3}{p}\right) \\ & + \beta \cdot \left(\frac{2mn + 3n^2/4}{\sqrt{p}}\right) \\ & + \alpha \cdot \frac{7}{2} \sqrt{p} \log^2 p + \nu \cdot O\left(\frac{mn \log p}{\sqrt{p}}\right). \end{aligned}$$

These costs are shown in the fourth row of Table 2. The computation cost of the Butterfly-CAQR is the same as 2D-Householder-QR and CAQR. The interprocessor bandwidth cost differs in the $O(n^2/\sqrt{p})$ term, which is a factor of 3 higher than 2D-Householder-QR, but a factor of $O(\log p)$ lower than CAQR. The latency cost is a factor of $O(n/\log p)$ lower than 2D-Householder-QR, a reduction achieved due to the use of TSQR, and a factor of $O(\log p)$ lower than CAQR, due to the fact that the trailing matrix update incurs $O(\log p)$ less computational cost, allowing a larger block size. The memory bandwidth cost is asymptotically the same as that of 2D-Householder-QR, but a factor of $O(\log p)$ better than CAQR, again since a larger block size can be used.

## 5.4. CAQR-HR

**Algorithm 21** $[Y, R] = \text{CAQR-HR}(A, \pi)$

```
Require: $A$ is $m \times n$ and distributed block-cyclically on $p = p_r \cdot p_c$
    processors with block size $b$, so that each $b \times b$ block $A_{ij}$ is owned
    by $\pi(i, j) = (i \bmod p_r) + p_r \cdot (j \bmod p_c)$
1: for $i = 0$ to $n/b - 1$ do
2:     Let $\pi^{(i)}(r, c) = \pi((r+i) \bmod p_r, (c+i) \bmod p_c)$
           for $r \in [0, \min(m/b - i, p_r - 1)]$ and $c \in [0, \min(n/b - i, p - 1))]$
       % Compute TSQR and reconstruct Householder representation using proc. column $\pi(:, i)$
3:     $[Y_{i:m/b-1,i}, T_i, R_{ii}] = $ 1D-TSQR-HR($A_{i:m/b-1,i}, \pi^{(i)}(:, 0)$)
4:     $[A_{i:m/b-1,i+1:n/b-1}] = $ 2D-Householder-Update($A_{i:m/b-1,i+1:n/b-1}, Y_{i:m/b-1,i}, T_i, \pi^{(i)}$)
5:     Set $R_{i,i+1:n/b-1} = A_{i,i:n/b-1}$
6: end for
Ensure: $A = \left(\prod_{i=0}^{n/b-1} (I - Y_{:,i} T_i Y_{:,i}^T)\right) R$, where $Y$ is $m \times n$ unit
    lower triangular and $R$ is $m \times n$ upper triangular, both with the
    same distribution as $A$
```

We refer to the 2D algorithm that uses TSQR-HR for panel factorizations as CAQR-HR. Because Householder-QR and TSQR-HR generate the same representation as output of the panel factorization, the trailing matrix update can be performed in exactly the same way. The key difference between 2D-Householder-QR and CAQR-HR, presented in Algorithm 21, is the subroutine call for the panel factorization (line 3). Additionally, it is no longer necessary to compute the $T$ matrix from $Y$ as done in 2D-Householder-QR, as the TSQR-HR algorithm computes it in $O(b^3)$ operations.

Algorithm 21 operates with block size $b$ on an $m \times n$ matrix $A$, with $m \geq n$ and $m, n \equiv 0 \pmod b$ distributed on a 2D $p_r$-by-$p_c$ processor grid. At iteration $i$ there are $p_r^{(i)}$ active rows and $p_c^{(i)}$ active columns in $\pi^{(i)}$. Over all $n/b$ iterations, the cost of the algorithm is

$$T_{\text{CAQR-HR}} = \sum_{i=0}^{n/b-1} \left[ T_{\text{TSQR-HR}}(m - ib, b, p_r^{(i)}) + T_{\text{2D-Hh-Upd}}(m - ib, n - ib, b, p_r^{(i)}, p_c^{(i)}) \right].$$

Assuming $b \leq \frac{1}{\log p} \min(m/p_r, n/p_c)$, the computation and memory bandwidth costs associated with the panel factorization becomes low-order (by analogy to the CAQR analysis), while the leading order interprocessor communication costs are a combination of the trailing matrix update (which is the same as in 2D-Householder-QR) and TSQR-HR,

$$T_{\text{CAQR-HR}} = \gamma \cdot \frac{2mn^2 - 2n^3/3}{p} + \beta \cdot \left( nb \log p_r + \frac{2mn - n^2}{p_r} + \frac{n^2}{p_c} \right) + \alpha \cdot \left( \frac{4n}{b} \log p_r + \frac{2n}{b} \log p_c + \frac{2n}{b} \log p \right) + \nu \cdot O\left( \frac{mn^2}{pb} \right).$$

In the case when the matrix is nearly square $m \approx n$, it is again sensible to pick $p_r = p_c = \sqrt{p}$ and $b = n/(\sqrt{p} \log p)$, obtaining

$$T_{\text{CAQR-HR}}\left(m, n, \frac{n}{\sqrt{p}\log p}, \sqrt{p}, \sqrt{p}\right) = \gamma \cdot \frac{2mn^2 - 2n^3/3}{p} + \beta \cdot \frac{2mn + n^2/2}{\sqrt{p}} + \alpha \cdot 5\sqrt{p} \log^2 p + \nu \cdot O\left( \frac{mn \log p}{\sqrt{p}} \right).$$

These costs are shown in the fifth row of Table 2.

Comparing the leading order costs of CAQR-HR with the existing approaches, we note again the $O(n \log p)$ latency cost incurred by the 2D-Householder-QR algorithm. CAQR and CAQR-HR eliminate this synchronization bottleneck and reduce the latency cost to be independent of the number of columns of the matrix. Further, both the bandwidth and latency costs of CAQR-HR are factors of $O(\log p)$ lower than CAQR (when $m \approx n$). As previously discussed, CAQR includes an extra leading order bandwidth cost term, $\beta \cdot O(n^2 \log p/\sqrt{p})$, as well as a computational cost term, $\gamma \cdot O((n^2 b/p_c) \log p_r)$ that requires the choice of a smaller block size and leads to an increase in the latency cost. Our next step will be to demonstrate a technique to eliminate the $\nu \cdot O\left(\frac{mn \log p}{\sqrt{p}}\right)$ memory bandwidth overhead present in CAQR-HR.

## 5.5. Two-level aggregation

The Householder-QR algorithm attains an efficient trailing matrix update by aggregating Householder vectors into panels (the compact-WY representation) and applying the update via Algorithm 15. However, in the memory-bandwidth cost analysis of the use of this update 2D-Householder-QR, which also applies to CAQR-HR, we see that a memory-bandwidth overhead arises due to the cost of moving the trailing matrix to and from cache every time it is updated. We now present an approach which avoids this overhead by applying the update in multiple stages, exploiting the fact that only the part of the trailing matrix corresponding to the next panel need to be updated. This technique leverages the flexibility of the Householder form of the update, by aggregating multiple updates into a larger one. This "two-level" aggregation is possible for any basis-kernel representation [33, Corollary 2.8], but the Householder form is particularly convenient as it maintains trapezoidal structure of the basis and triangular structure of the kernel. For Yamamoto's representation the aggregation requires the formation of a block-trapezoidal basis and block-triangular kernel.

Given the Householder vector reconstruction technique, two-level aggregation makes it possible to decouple the block sizes used for the trailing matrix update from the width of each TSQR. We detail this nested blocked approach in Algorithm 23, which employs Algorithm 21 as a subroutine. Algorithm 23 aggregates the Householder vectors simultaneously on every matrix column, exploiting the fact that each vector is broadcast along the row in Algorithm 15, which saves some interprocessor communication. This two-level aggregation is analogous to the two-level blocking technique employed in [31] for 2.5D LU factorization, albeit only on a 2D grid of processors.

**Algorithm 22** $[T^{-1}]$ = 2D-Form-Inverse-$T$-from-$Y(Y, \pi)$

```
Require: $Y$ is $m \times l$ and replicated over columns of $p_r$-by-$p_c$, $\pi$ so
    the subset of rows $Y_r$ is owned by $\pi(r,c)$ for each $c$. Assumes
    $l \bmod p_c = 0$.
1: $\pi(r,c)$ computes $\widetilde{W}_c = Y_r(:, cl/p_c + 1 : (c+1)l/p_c)^T \cdot Y_r$
2: All-reduce $W_c = \sum_c \widetilde{W}_c$ along $\pi(:,c)$
3: All-gather $W = [W_1, W_2, \ldots, W_{p_c - 1}]$ along $\pi(r,:)$
4: $\pi(r,c)$ extracts upper-triangular $T^{-1}$ from $W$
Ensure: $T^{-1}$ is $l \times l$ and upper triangular, owned by each $\pi(r,c)$,
    and satisfies $Y^T Y = T^{-1} + T^{-T}$.
```

Two-level aggregation requires forming a larger kernel $T$. The cost of forming this kernel becomes a leading order computational cost term if done using 1D-Form-$T$-from-$Y$. Our implementation takes this simpler approach, as the extra local work was not seen to be a bottleneck in our experiments. However, it is possible to compute $T$ more efficiently using a 2D processor grid, which we demonstrate in Algorithm 22. This algorithm assumes that $Y$ is replicated on each processor column, which is the case in Algorithm 23, computes a subset of columns of $Y^T Y$ on each processor column then all-gathers the $l \times l$ result and outputs $T^{-1}$ on all processors (to avoid the $O(b^3)$ computational cost of inverting $T$). The interprocessor communication cost of the algorithm involves an all-reduce of size $l \times l/p_c$ and an all-gather of a symmetric $l \times l$ matrix, while the memory-bandwidth and flop terms are incurred only by the computation of $\widetilde{W}_c = Y_r(:, cl/p_c + 1 : (c+1)l/p_c)^T \cdot Y_r$. Thus the leading order cost of Algorithm 22 is

$$T_{\text{2D-invT-from-Y}}(m, l, p_r, p_c) = \gamma \cdot \frac{2ml^2}{p_c} + \beta \cdot l^2/2 + \alpha \cdot (2 \log p_r + \log p_c) + \nu \cdot O(ml/p_c).$$

In deriving the overall cost of Algorithm 23 we assume that, as before $b_1 \leq \frac{1}{\log p} \min(m/p_r, n/p_c)$, while $b_2 \leq \min(m/p_r, n/p_c)$. The latter assumption is needed to ensure that $O(mn/p)$ memory is used (otherwise the local part of the aggregated $Y$ becomes larger than the local matrix). We first note that due to the assumption $b_2 \leq \min(m/p_r, n/p_c)$ the computational cost of line 8 in Algorithm 23, $\gamma \cdot O(n^2 b_2/p)$ over all iterations, is low order. The rest of the steps in the updates done in Algorithm 23 have effectively the same cost as the total cost of updates done in 2D-Householder-QR, since the cost of an all-reduce in our model is the same as that of a reduce-scatter followed by an allgather. The overall cost of the two-level algorithm (keeping into account that at iteration $j$ of the

**Algorithm 23** $[Y, R] =$ Two-Level-CAQR-HR$(A, \pi)$

```
Require: $A$ is $m \times n$ and distributed block-cyclically on $p = p_r \cdot p_c$ processors with block size $b_1$, so that each $b_1 \times b_1$ block $A_{ij}$ is owned by $\pi(i,j) = (i \bmod p_r) + p_r \cdot (j \bmod p_c)$. Let $b_2 = kb_1$ be an integer ($k$) multiple of $b_1$ and $n \bmod b_2 = 0$.
1:  for $j = 0$ to $n/b_2 - 1$ do
2:      Let $\pi^{(j)}(r,c) = \pi(r{+}jk \bmod p_r, c{+}jk \bmod p_c)$ for $r \in [0, \min(m/b_1 - jk, p_r{-}1)]$ and $c \in [0, \min(n/b_1{-}jk, p{-}1))]$
        % Compute Householder-QR on a $b_2$-width matrix panel using full processor grid
3:      $\left[Y(jb_2{+}1 : m, jb_2{+}1 : (j{+}1)b_2), R(jb_2{+}1 : (j{+}1)b_2, jb_2{+}1 : (j{+}1)b_2)\right]$ $= \text{CAQR-HR}(A(jb_2{+}1 : m, jb_2{+}1 : (j{+}1)b_2), \pi^{(j)})$
4:      Form $\bar{Y} = Y(jb_2{+}1 : m, jb_2{+}1 : (j{+}1)b_2)$ on each proc. column $\pi^{(j)}(:,c)$ by concatenating panels of $\bar{Y}$ which were broadcast within the above call on line 2 of Algorithm 15
5:      $[Z_j] =$ 2D-Form-Inverse-$T$-from-$Y(\bar{Y}, \pi^{(j)}(:,c))$
        % Update trailing matrix using all processors
6:      $\pi^{(j)}(r,c)$ computes $\widetilde{W}_{rc} = \bar{Y}_r^T \cdot A_{rc}$
7:      Reduce-scatter $W_c = \sum_r \widetilde{W}_{rc}$ along $\pi(:,c)$ so that $\pi^{(j)}(r,c)$ owns a subset of columns $W_{rc}$
8:      $\pi^{(j)}(r,c)$ computes $V_{rc} = Z_j^{-T} \cdot W_{rc}$
9:      All-gather $V_c = [W_{1c}, W_{2c}, \dots, W_{p_r-1,c}]$ along $\pi(:,c)$
10:     $\pi^{(j)}(r,c)$ computes $A_{rc} = A_{rc} - \bar{Y}_r \cdot V_c$
11:     Set $R(jb_2{+}1 : (j{+}1)b_2, (j{+}1)b_2{+}1 : n) = A(jb_2{+}1 : (j{+}1)b_2, (j{+}1)b_2{+}1 : n)$
12: end for
Ensure: $A = \left(\prod_{j=0}^{n/b_2-1}(I - Y_{:,jk:(j+1)k-1} Z_j^{-T} Y_{:,jk:(j+1)k-1}^T)\right) R$, where $Y$ is $m \times n$ unit lower triangular and $R$ is $m \times n$ upper triangular, both with the same distribution as $A$
```

loop in Algorithm 23 $p_r^{(j)}$ processor rows and $p_c^{(j)}$ processor columns are active in $\pi^{(j)}$ and $p^{(j)} = p_r^{(j)} p_c^{(j)}$) is therefore given by

$$\begin{aligned}
&T_{\text{2L-CAQR-HR}}(m, n, b_1, b_2, p_r, p_c)\\
&= \sum_{j=0}^{n/b_2}\Bigg[T_{\text{CAQR-HR}}(m - jb_2, b_2, b_1, p_r^{(j)}, p_c^{(j)})\\
&\quad + T_{\text{2D-invT-fm-Y}}(m - jb_2, b_2, p_r^{(j)}, p_c^{(j)})\\
&\quad + \gamma \cdot \frac{4(m - jb_2)(n - jb_2)b_2}{p^{(j)}} + \beta \cdot \frac{2(n - jb_2)b_2}{p_c^{(j)}}\\
&\quad + \alpha \cdot (3\log p_c^{(j)} + 2\log p_r^{(j)}) + \nu \cdot O\left(\frac{(m - jb_2)(n - jb_2)}{p^{(j)}}\right)\Bigg]\\
&= \gamma \cdot \frac{2mn^2 - 2n^3/3}{p}\\
&\quad + \beta \cdot \left(nb_1 \log p_r + nb_2/4 + \frac{2mn - n^2}{p_r} + \frac{n^2}{p_c}\right)\\
&\quad + \alpha \cdot \left(\frac{4n}{b_1}\log p_r + \frac{2n}{b_1}\log p_c + \frac{2n}{b_1}\log p\right.\\
&\quad \left. + \frac{3n}{b_2}\log p_c + \frac{2n}{b_2}\log p_r\right) + \nu \cdot O\left(\frac{mnb_2}{pb_1} + \frac{mn^2}{pb_2}\right)
\end{aligned}$$

where the term $\beta \cdot nb^2/4$ comes from the 2D algorithm for forming $T^{-1}$, while all the other terms are either from CAQR-HR or the update done at the second level within the pseudocode in Algorithm 23. On a square processor grid, if we select $b_1 = n/(\sqrt{p}\log p)$ and $b_2 = n/\sqrt{p}$, the memory bandwidth terms go away since they become asymptotically no greater than the interprocessor bandwidth cost, and we obtain an overall cost of

$$\begin{aligned}
T_{\text{2L-CAQR-HR}}\left(m, n, \frac{n}{\sqrt{p}\log p}, \frac{n}{\sqrt{p}}, \sqrt{p}, \sqrt{p}\right) &= \gamma \cdot \frac{2mn^2 - 2n^3/3}{p}\\
&\quad + \beta \cdot \frac{2mn + 3n^2/4}{\sqrt{p}} + \alpha \cdot 5\sqrt{p}\log^2 p,
\end{aligned}$$

which appears in the sixth row of Table 2.

The same two-level algorithmic blocking optimization may be applied to 2D-Householder-QR, simply be replacing the call to CAQR-HR on line 3 in Algorithm 23 with a call to 2D-Householder-QR. We refer to this algorithm as Two-Level-2D-Householder-QR; the analysis of its costs is essentially the same as above, yielding the leading order expression,

$$\begin{aligned}
T_{\text{2L-2D-HQR}}(m, n, b_1, b_2, p_r, p_c) &= \gamma \cdot \frac{2mn^2 - 2n^3/3}{p}\\
&\quad + \beta \cdot \left(nb_1 \log p_r + nb_2/4 + \frac{2mn - n^2}{p_r} + \frac{n^2}{p_c}\right)\\
&\quad + \alpha \cdot 2n\log p_r + \nu \cdot O\left(\max\left\{\frac{mnb_2}{pb_1}\right.\right.\\
&\quad \left.\left. + \frac{mn^2}{pb_2}, \frac{mnb_1}{p_r} - \frac{Z^2 np_r}{mb_1}\right\}\right).
\end{aligned}$$

On a $\sqrt{p}$-by-$\sqrt{p}$ processor grid, if we assume that the cache size $Z$ is small, so that the second term in the max does not vanish, we can make the three terms contributing to the memory bandwidth cost, namely $mnb_2/(pb_1)$ associated with the $m \times b_2$ trailing matrix within 2D-Householder-QR, $mn^2/(pb_2)$ associated with the trailing matrix of the aggregated update, and $mnb_1/\sqrt{p}$ associated with the 1D-Householder-QR, equal by selecting $b_1 = n^{1/3}/p^{1/3}$ and $b_2 = n^{2/3}/p^{1/6}$, yielding the total cost:

$$\begin{aligned}
&T_{\text{2L-2D-Hh-QR}}\left(m, n, \frac{n^{1/3}}{p^{1/3}}, \frac{n^{2/3}}{p^{1/6}}, \sqrt{p}, \sqrt{p}\right)\\
&= \gamma \cdot (2mn^2 - 2n^3/3)/p + \beta \cdot (2mn + n^2/2)/\sqrt{p}\\
&\quad + \alpha \cdot n\log p + \nu \cdot O\left(\frac{mn^{4/3}}{p^{5/6}}\right),
\end{aligned}$$

which appears in the second row of Table 2.

We implemented Two-Level-2D-Householder-QR employing ScaLAPACK to factor each thin panel. We note that ScaLAPACK could be modified to use this algorithm with the addition of a second algorithmic blocking factor. Additionally, we implemented a two-level aggregated version of Yamamoto's algorithm, whose theoretical costs are the same as $T_{\text{2L-CAQR-HR}}$. In all of our implementations the larger kernel was formed using a 1D algorithm and therefore our implementations incur a slightly higher computational cost than the ones reported above. These two-level algorithms obtain a significant performance improvement over their single-level aggregated counterparts, as we discuss in Section 8.

## 6. Correctness and stability

In order to reconstruct the lower trapezoidal Householder vectors that constitute an orthonormal matrix (up to column signs), we can use an algorithm for LU decomposition without pivoting on an associated matrix (Algorithm 3). Lemma 6.1 shows by uniqueness that this LU decomposition produces the Householder vectors representing the orthogonal matrix in exact arithmetic. Given the explicit orthogonal factor, the LU method is cheaper in terms of both computation and communication than constructing the vectors via Householder-QR.

**Lemma 6.1.** *Given an orthonormal $m \times b$ matrix $Q$, let the compact QR decomposition of $Q$ given by the Householder-QR algorithm (Algorithm 1) be*

$$Q = \left( \begin{bmatrix} I_n \\ 0 \end{bmatrix} - YTY_1^T \right) S,$$

*where $Y$ is unit lower triangular, $Y_1$ is the top $b \times b$ block of $Y$, and $T$ is the upper triangular $b \times b$ matrix satisfying $T^{-1} + T^T = Y^T Y$. Then $S$ is a diagonal sign matrix, and $Q - \begin{bmatrix} S \\ 0 \end{bmatrix}$ has a unique LU decomposition given by*

$$Q - \begin{bmatrix} S \\ 0 \end{bmatrix} = Y \cdot (-TY_1^T S). \tag{6}$$

**Proof.** Since $Q$ is orthonormal, it has full rank and therefore has a unique QR decomposition with positive diagonal entries in the upper triangular matrix. This decomposition is $Q = Q \cdot I_n$, so the Householder-QR algorithm must compute a decomposition that differs only by sign. Thus, $S$ is a diagonal sign matrix. The uniqueness of $T$ is guaranteed by [33, Example 2.4].

We obtain Eq. (6) by rearranging the Householder QR decomposition. Since $Y$ is unit lower triangular and $T$, $Y_1^T$, and $S$ are all upper triangular, we have an LU decomposition. Since $Y$ is full column rank and $T$, $Y_1^T$, and $S$ are all nonsingular, $Q - \begin{bmatrix} S \\ 0 \end{bmatrix}$ has full column rank and therefore has a unique LU decomposition with a unit lower triangular factor. □

Note that Lemma 6.1 extends to complex-valued matrices $Q$ with orthonormal columns. In this case, the sign matrix $S$ is a diagonal matrix with unit modulus diagonal entries.

Unfortunately, given an orthonormal matrix $Q$, the sign matrix $S$ produced by Householder-QR is not known, so we cannot run a standard LU algorithm on $Q - \begin{bmatrix} S \\ 0 \end{bmatrix}$. Lemma 6.2 shows that by running the Modified-LU algorithm (Algorithm 3), we can cheaply compute the sign matrix $S$ during the course of the algorithm.

**Lemma 6.2.** *In exact arithmetic, Algorithm 3 applied to an orthonormal $m \times b$ matrix $Q$ computes the same Householder vectors $Y$ and sign matrix $S$ as the Householder-QR algorithm (Algorithm 1) applied to $Q$.*

**Proof.** Consider applying one step of Modified-LU (Algorithm 3) to the orthonormal matrix $Q$, where we first set $S(1, 1) = -\operatorname{sgn}(Q(1, 1))$ and subtract it from $Q(1, 1)$ (this computes the diagonal entry of $U$). Note that since all other entries of the first column $Q(:, 1)$ are less than 1 in absolute value and the absolute value of the diagonal entry has been increased by 1, the diagonal entry is the maximum entry. Following the LU algorithm, all entries below the diagonal are scaled by the reciprocal of the diagonal element: for $2 \le i \le m$,

$$Y(i, 1) = \frac{Q(i, 1)}{Q(1, 1) + \operatorname{sgn}(Q(1, 1))}, \tag{7}$$

where $Y$ is the computed lower triangular factor. The first row of the upper triangular factor $U$ is set to be the first row of $Q$. The Schur complement is updated as follows: for $2 \le i \le m$ and $2 \le j \le b$,

$$\tilde{Q}(i, j) = Q(i, j) - \frac{Q(i, 1)Q(1, j)}{Q(1, 1) + \operatorname{sgn}(Q(1, 1))}. \tag{8}$$

Now consider applying one step of the Householder-QR algorithm (Algorithm 1). To match the LAPACK notation, we let $\alpha = Q(1, 1)$, $\beta = -\operatorname{sgn}(\alpha) \cdot \|Q(:, 1)\| = -\operatorname{sgn}(\alpha)$, and $\tau = \frac{\beta-\alpha}{\beta} = 1 + \alpha \operatorname{sgn}(\alpha)$, where we use the fact that the columns of $Q$ have unit norm. The Householder vector $y = Y(:, 1)$ is computed by setting the diagonal entry to 1 and scaling the other entries of $Q(:, 1)$ by $\frac{1}{\alpha-\beta} = \frac{1}{\alpha+\operatorname{sgn}(\alpha)}$. Thus, computing the Householder vector matches computing the column of the lower triangular matrix from Modified-LU in Eq. (7).

Next, we consider the update of the trailing matrix: $\tilde{Q} = (I - \tau yy^T)Q$. Since $Q$ has orthonormal columns, the dot product of the Householder vector with the $j$th column is given by

$$\begin{aligned} y^T Q(:, j) &= \sum_{i=1}^{m} y(i) Q(i, j) \\ &= Q(1, j) + \sum_{i=2}^{m} \frac{Q(i, 1)}{\alpha + \operatorname{sgn}(\alpha)} Q(i, j) \\ &= Q(1, j) - \frac{Q(1, 1)Q(1, j)}{\alpha + \operatorname{sgn}(\alpha)} \\ &= \left(1 - \frac{\alpha}{\alpha + \operatorname{sgn}(\alpha)}\right) Q(1, j). \end{aligned}$$

The identity

$$(1 + \alpha \operatorname{sgn}(\alpha)) \left(1 - \frac{\alpha}{\alpha + \operatorname{sgn}(\alpha)}\right) = 1$$

implies $\tau y^T Q(:, j) = Q(1, j)$. Then the trailing matrix update ($2 \le i \le m$ and $2 \le j \le b$) is given by

$$\tilde{Q}(i, j) = Q(i, j) - y(i)(\tau y^T Q(:, j)) = Q(i, j) - \frac{Q(i, 1)Q(1, j)}{\alpha + \operatorname{sgn}(\alpha)},$$

which matches the Schur complement update from Modified-LU in Eq. (8).

Finally, consider the $j$th element of first row of the trailing matrix ($j \ge 2$): $\tilde{Q}(1, j) = Q(1, j) - y(1)(\tau y^T Q(:, j)) = 0$. Note that the computation of the first row is not performed in the Modified-LU algorithm. The corresponding row is not changed since the diagonal element is $Y(1, 1) = 1$. However, in the case of Householder-QR, since the first row of the trailing matrix is zero, and because the Householder transformation preserves column norms, the updated $(m - 1) \times (b - 1)$ trailing matrix is itself an orthonormal matrix. Thus, by induction, the rest of the two algorithms perform the same computations. □

Note that Lemma 6.2 extends to the complex-valued case, with transposes interpreted as conjugate-transposes.

In order to bound the error in the decomposition of a general matrix using Householder reconstruction, we will use Lemma 6.4 as well as the stability bounds on the TSQR or "AllReduce" algorithm provided in [28]. We restate those results here, using the notation $\gamma_c = c\varepsilon/(1 - c\varepsilon)$ with $\varepsilon$ being machine precision.

**Lemma 6.3** ([28]). *Let $\hat{R}$ be the computed upper triangular factor of $m \times b$ real-valued matrix $A$ obtained via the AllReduce algorithm using a binary tree of height $L$ ($m/2^L \ge b$). Then there exists an orthonormal matrix $Q$ such that*

$$\|A - Q\hat{R}\|_F \le f_1(m, b, L, \varepsilon)\|A\|_F$$

*and*

$$\|Q - \hat{Q}\|_F \le f_2(m, b, L, \varepsilon),$$

*where $f_1(m, b, L, \varepsilon) \simeq b\gamma_{c \cdot \frac{m}{2^L}} + Lb\gamma_{c \cdot 2b}$ and $f_2(m, b, L, \varepsilon) \simeq \left(b\gamma_{c \cdot \frac{m}{2^L}} + Lb\gamma_{c \cdot 2b}\right)\sqrt{b}$, for $b\gamma_{c \cdot \frac{m}{2^L}} \ll 1$ and $b\gamma_{c \cdot 2b} \ll 1$, where $c$ is a small constant.*

The following lemma shows that the computation performed in Algorithm 3 is more stable than general LU decomposition. Because it is performed on an orthonormal matrix, the growth factor in the upper triangular factor is bounded by 2, and we can bound the Frobenius norms of both computed factors by functions of the number of columns. Further, we can stably compute the $\tilde{T}$ factor (needed to apply a blocked Householder update) from the upper triangular LU factor, a computationally cheaper method than using the Householder vectors themselves.

**Lemma 6.4.** *In floating point arithmetic, given an orthonormal $m \times b$ matrix $Q$, Algorithm 3 computes factors $S$, $\tilde{Y}$, and $\tilde{T}$ such that*

$$\left\| QS - \left( \begin{bmatrix} I \\ 0 \end{bmatrix} - \tilde{Y}\tilde{T}\tilde{Y}_1^T \right) \right\|_F \le f_3(b, \varepsilon)$$

*where*

$$f_3(b, \varepsilon) = 2\gamma_b \left( b^2 + \left(1 + \sqrt{2}\right) b \right)$$

*and $Y_1$ is given by the first $b$ rows of $Y$.*

**Proof.** This result follows from the standard error analysis of LU decomposition and triangular solve with multiple right hand sides, and the properties of the computed lower and upper triangular factors.

First, we use the fact that for LU decomposition of $Q - \begin{bmatrix} S \\ 0 \end{bmatrix}$ into triangular factors $\tilde{Y}$ and $\tilde{U}$, we have (see [21, Section 9.3])

$$\left\| \left( Q - \begin{bmatrix} S \\ 0 \end{bmatrix} \right) - \tilde{Y}\tilde{U} \right\|_F \le \gamma_b \|\tilde{Y}\|_F \|\tilde{U}\|_F.$$

Second, we compute $\tilde{T} = (\tilde{U}S)Y_1^{-T}$ using the standard algorithm for triangular solve with multiple right hand sides to obtain (see [21, Section 8.1])

$$\left\| \tilde{U} - \tilde{T}\tilde{Y}_1^T S \right\|_F = \left\| \tilde{U}S - \tilde{T}\tilde{Y}_1^T \right\|_F \le \gamma_b \|\tilde{T}\|_F \|\tilde{Y}_1^T\|_F.$$

Finally, we bound the norms of the computed triangular factors in exact arithmetic. At each step of the Modified-LU algorithm, the trailing matrix is an orthogonal matrix (before modifying the diagonal element). Each column of $\tilde{Y}$ is computed by scaling down elements of a unit vector and setting the diagonal element to 1. Thus, the sum of squares of elements in a column is bounded by 2, and $\|\tilde{Y}\|_F \le \sqrt{2b}$. Note that this also implies $\|\tilde{Y}_1\|_F \le \sqrt{2b}$. Each row of $\tilde{U}$ is computed by increasing the absolute value of the diagonal element by 1, but leaving the other elements to the right of the diagonal unchanged. The sum of squares of elements in a row of a matrix with orthonormal columns is at most 1. Thus, the sum of squares of elements in a row of $\tilde{U}$ is bounded by 4, and $\|\tilde{U}\|_F \le 2\sqrt{b}$. From [8, Theorem 13], we have $\|\tilde{T}\|_F < b + 1$.

Therefore, altogether we have

$$\begin{aligned}
\left\| QS - \left( \begin{bmatrix} I \\ 0 \end{bmatrix} - \tilde{Y}\tilde{T}\tilde{Y}_1^T \right) \right\|_F &= \left\| Q - \left( \begin{bmatrix} S \\ 0 \end{bmatrix} - \tilde{Y}\tilde{T}\tilde{Y}_1^T S \right) \right\|_F \\
&= \left\| \left( Q - \begin{bmatrix} S \\ 0 \end{bmatrix} \right) - \tilde{Y}\tilde{U} + \tilde{Y}\left( \tilde{U} - \tilde{T}\tilde{Y}_1^T S \right) \right\|_F \\
&\le \left\| \left( Q - \begin{bmatrix} S \\ 0 \end{bmatrix} \right) - \tilde{Y}\tilde{U} \right\|_F + \|\tilde{Y}\|_F \left\| \tilde{U} - \tilde{T}\tilde{Y}_1^T S \right\|_F \\
&\le \gamma_b \|\tilde{Y}\|_F \|\tilde{U}\|_F + \gamma_b \|\tilde{Y}\|_F \|\tilde{T}\|_F \|\tilde{Y}_1^T\|_F \\
&\le \gamma_b \left( \sqrt{8}b + 2b(b+1) \right). \quad \square
\end{aligned}$$

**Theorem 6.5.** *Let $\hat{R}$ be the computed upper triangular factor of $m \times b$ matrix $A$ obtained via the TSQR algorithm using a binary tree of height $L$ $(m/2^L \ge b)$, and let $\tilde{Q} = I - \tilde{Y}\tilde{T}\tilde{Y}_1^T$ and $\tilde{R} = S\hat{R}$ where $\tilde{Y}$, $\tilde{T}$, and $S$ are the computed factors obtained from the Householder reconstruction algorithm. Then*

$$\|A - \tilde{Q}\tilde{R}\|_F \le F_1(m, b, L, \varepsilon)\|A\|_F$$

*and*

$$\|I - \tilde{Q}^T\tilde{Q}\|_F \le F_2(m, b, L, \varepsilon)$$

*where*

$$\begin{aligned}
F_1(m, b, L, \varepsilon) \simeq\ & (b\gamma_{c \cdot \frac{m}{2^L}} + Lb\gamma_{c \cdot 2b})(1 + \sqrt{b}) \\
& + 2\gamma_b \left( b^2 + \left(1 + \sqrt{2}\right) b \right)
\end{aligned}$$

*and*

$$F_2(m, b, L, \varepsilon) \simeq 2(b\gamma_{c \cdot \frac{m}{2^L}} + Lb\gamma_{c \cdot 2b})\sqrt{b} + 4\gamma_b \left( b^2 + \left(1 + \sqrt{2}\right) b \right)$$

*for $b\gamma_{c \cdot \frac{m}{2^L}} \ll 1$ and $b\gamma_{c \cdot 2b} \ll 1$.*

**Proof.** From Lemmas 6.3 and 6.4, there exists an exactly orthonormal matrix $Q$ such that

$$\begin{aligned}
\|A - \tilde{Q}\tilde{R}\|_F &= \|A - Q\hat{R} - (Q - \hat{Q})\hat{R} - (\hat{Q} - \tilde{Q}S)\hat{R}\|_F \\
&\le \|A - Q\hat{R}\|_F + \|Q - \hat{Q}\|_F \|\hat{R}\|_F + \|\hat{Q} - \tilde{Q}S\|_F \|\hat{R}\|_F \\
&\le f_1(m, b, L, \varepsilon)\|A\|_F + f_2(m, b, L, \varepsilon)\|\hat{R}\|_F + f_3(b, \varepsilon)\|\hat{R}\|_F \\
&\le F_1(m, b, L, \varepsilon)\|A\|_F.
\end{aligned}$$

The approximation of $F_1$ also uses [28, Lemma 1]: $\|\hat{R}\|_F \simeq \|A\|_F$ assuming $b\gamma_{c \cdot \frac{m}{2^L}} \ll 1$ and $b\gamma_{c \cdot 2b} \ll 1$.

Further, since $Q$ is exactly orthonormal, we have

$$\begin{aligned}
\|I - \tilde{Q}^T\tilde{Q}\|_F &= \|Q^T(Q - \tilde{Q}S) + (Q - \tilde{Q}S)^T\tilde{Q}S\|_F \\
&\le \|Q - \tilde{Q}S\|_F \left(1 + \|\tilde{Q}\|_F\right) \\
&\le \left(\|Q - \hat{Q}\|_F + \|\hat{Q} - \tilde{Q}S\|_F\right)\left(1 + \|\tilde{Q}\|_F\right) \\
&\le (f_2(m, b, L, \varepsilon) + f_3(b, \varepsilon))\left(1 + \|\tilde{Q}\|_F\right) \\
&\le F_2(m, b, L, \varepsilon). \quad \square
\end{aligned}$$

## 7. Numerical experiments

In this section we present numerical results of TSQR-HR. Experiments were conducted on two representative sets of test matrices. The first set is used to check the stability of the algorithm on single panels represented by tall and skinny matrices, while the second set focuses on the factorization of full matrices panel by panel.

The orthogonality of $\tilde{Q}$ is measured by $\|I - \tilde{Q}^T\tilde{Q}\|_F$. The stability of the factorization is also measured by the norm-wise relative backward error $\|A - \tilde{Q}\tilde{R}\|_F/\|A\|_F$. We found similar results for column-wise relative errors.

### 7.1. Tall and skinny matrices

In this section, we use matrices which are formed by $A = Q \cdot R_\rho$, where $Q$ and $R$ are computed via QR decomposition of an $m \times b$ matrix with independent and identically distributed entries chosen from a normal distribution. $R_\rho$ is obtained by setting the $\lfloor \frac{n}{2} \rfloor$th diagonal element of an upper triangular matrix $R$ to a small parameter value $\rho$. This experimental setup is used to vary the

**Table 3**
Residual and orthogonality error for 5 approaches on tall and skinny matrices ($m = 1000, n = 200$) constructed using the parameter $\rho$ resulting in condition number $\kappa$. Errors of $+\infty$ indicate failure within a Cholesky factorization.

| $\rho$ | $\kappa$ | TSQR-HR (Algorithm 10) | | Yamamoto's approach (Algorithm from [37]) | | LU$(A-R)$ (Algorithm 11) | | LU$(A-R)$ (Algorithm 11) with iterative refinement | | CholQR2-HR (Algorithm 12) | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\frac{\|A-\tilde{Q}\tilde{R}\|_F}{\|A\|_F}$ | $\|I-\tilde{Q}^T\tilde{Q}\|_F$ | $\frac{\|A-\tilde{Q}\tilde{R}\|_F}{\|A\|_F}$ | $\|I-\tilde{Q}^T\tilde{Q}\|_F$ | $\frac{\|A-\tilde{Q}\tilde{R}\|_F}{\|A\|_F}$ | $\|I-\tilde{Q}^T\tilde{Q}\|_F$ | $\frac{\|A-\tilde{Q}\tilde{R}\|_F}{\|A\|_F}$ | $\|I-\tilde{Q}^T\tilde{Q}\|_F$ | $\frac{\|A-\tilde{Q}\tilde{R}\|_F}{\|A\|_F}$ | $\|I-\tilde{Q}^T\tilde{Q}\|_F$ |
| 1e−01 | 5.1E02 | 2.2E−15 | 6.8E−15 | 2.2E−15 | 6.8E−15 | 3.9E−14 | 5.3E−15 | 1.1E−15 | 2.6E−15 | 1.1E−15 | 2.6E−15 |
| 1e−02 | 5.0E03 | 2.7E−15 | 1.1E−14 | 2.7E−15 | 1.1E−14 | 3.2E−13 | 6.7E−15 | 1.1E−15 | 2.6E−15 | 1.1E−15 | 2.6E−15 |
| 1e−03 | 5.2E04 | 2.3E−15 | 9.3E−15 | 2.3E−15 | 9.3E−15 | 4.6E−12 | 5.0E−15 | 1.0E−15 | 2.7E−15 | 1.0E−15 | 2.8E−15 |
| 1e−04 | 5.1E05 | 3.2E−15 | 1.5E−14 | 3.2E−15 | 1.5E−14 | 3.6E−11 | 6.7E−15 | 1.1E−15 | 2.8E−15 | 1.1E−15 | 2.9E−15 |
| 1e−05 | 5.2E06 | 2.4E−15 | 9.5E−15 | 2.4E−15 | 9.5E−15 | 3.8E−10 | 6.7E−15 | 1.1E−15 | 2.6E−15 | 1.1E−15 | 3.0E−15 |
| 1e−06 | 5.0E07 | 2.2E−15 | 8.0E−15 | 2.2E−15 | 8.0E−15 | 3.8E−09 | 6.2E−15 | 1.1E−15 | 2.9E−15 | 1.1E−15 | 2.7E−15 |
| 1e−07 | 5.1E08 | 2.3E−15 | 9.1E−15 | 2.3E−15 | 9.1E−15 | 3.7E−08 | 6.9E−15 | 1.1E−15 | 2.8E−15 | 1.0E−15 | 2.6E−15 |
| 1e−08 | 5.1E09 | 2.2E−15 | 8.5E−15 | 2.2E−15 | 8.5E−15 | 3.8E−07 | 5.6E−15 | 1.1E−15 | 2.8E−15 | 1.1E−15 | 2.6E−15 |
| 1e−09 | 5.2E10 | 2.3E−15 | 9.3E−15 | 2.3E−15 | 9.3E−15 | 4.1E−06 | 5.1E−15 | 1.1E−15 | 2.8E−15 | 1.0E−15 | 2.8E−15 |
| 1e−10 | 5.0E11 | 2.3E−15 | 8.9E−15 | 2.3E−15 | 8.9E−15 | 3.1E−05 | 7.2E−15 | 1.1E−15 | 2.7E−15 | $+\infty$ | $+\infty$ |
| 1e−11 | 5.2E12 | 2.2E−15 | 8.8E−15 | 2.2E−15 | 8.8E−15 | 3.4E−04 | 5.9E−15 | 1.1E−15 | 2.7E−15 | 1.0E−15 | 2.8E−15 |
| 1e−12 | 5.1E13 | 2.4E−15 | 9.3E−15 | 2.4E−15 | 9.3E−15 | 3.6E−03 | 4.5E−15 | 1.0E−15 | 2.8E−15 | 1.1E−15 | 2.5E−15 |
| 1e−13 | 5.0E14 | 2.7E−15 | 1.2E−14 | 2.7E−15 | 1.2E−14 | 2.9E−02 | 5.6E−15 | 1.1E−15 | 2.7E−15 | $+\infty$ | $+\infty$ |
| 1e−14 | 3.8E15 | 2.5E−15 | 1.1E−14 | 2.5E−15 | 1.1E−14 | 2.3E−01 | 5.4E−15 | 1.1E−15 | 2.7E−15 | $+\infty$ | $+\infty$ |
| 1e−15 | 4.7E15 | 2.3E−15 | 8.7E−15 | 2.3E−15 | 8.7E−15 | 3.7E−01 | 6.0E−15 | 1.1E−15 | 4.6E−15 | $+\infty$ | $+\infty$ |

condition number $\kappa$ of the matrix and demonstrate instability of the cheaper method described in Section 4.4.1.

Table 3 shows a comparison between multiple approaches: TSQR-HR (Algorithm 10), Yamamoto's approach (Algorithm from [37]) described in Section 3.3, LU$(A-R)$ given in Algorithm 11 in Section 4.4.1, LU$(A-R)$ coupled with the iterative refinement as described in Algorithm 14 and CholQR2-HR given in Algorithm 12 (which is Cholesky QR with the same iterative refinement procedure).

The $T$ matrix used to update the trailing matrix is computed (when possible) as $T = -US^{-1}Y_1^{-T}$ ($U$ coming from Algorithm 3), or directly from $Y$ as detailed in Algorithm 5.

While Yamamoto's approach is as stable as TSQR-HR (Algorithm 9), LU$(A-R)$ (Algorithm 11) provides unsatisfactory backward errors which grow with the condition number of the matrix. Note that Cholesky QR is indubitably the lowest cost factorization alternative, however it is unstable.

The iterative refinement procedure applied to LU$(A-R)$ corrects the errors for all problem matrices in the set. When used in conjunction with Cholesky QR, iterative refinement obtains stable results as long as the condition number is lower than 1E10. Note that this approach corresponds to CholQR2-HR (Algorithm 12).

As can be seen from Table 3, for all test cases both the orthogonality and factorization errors of are of order $10^{-15}$ for TSQR-HR(Algorithm 10), which is close to $\epsilon = 2^{-52}$ of double precision. This result demonstrates the numerical stability of this approach on tall and skinny matrices with varying condition numbers.

### 7.2. Square matrices

We now present numerical results for the QR factorization of square matrices using a panel-by-panel factorization. The matrices are generated similarly to [12], where the set is chosen from well-known anomalous matrices. Details are given in Table 4. Most are of size 1000-by-1000 (except the ARC130 and FS_541_1 matrices, which are respectively of order 130 and 541), with various condition numbers ($\kappa$), some of them being very ill-conditioned (*i.e.*, having a condition number much larger than the inverse of machine precision). We tried several panel sizes ranging from 2 to 256 columns and report only the largest errors and their associated panel widths.

Results corresponding to LU$(A-R)$ (Algorithm 11) when computing $T = -UR^{-1}Y_1^{-T}$ or getting $T$ from Algorithm 5, $R$ coming from regular Householder QR, are reported in Table 5. It can be seen that the first approach is not as stable as the second approach. Hence, we will only compute $T$ from Algorithm 5 in the next set of experiments.

Results from Table 6 show that TSQR-HR is numerically stable in terms of backward errors when computing the QR factorization of full matrices, regardless of the condition number of the matrix, as suggested by Theorem 6.5. Again, Yamamoto's approach displays similar results. To the contrary, the alternative approach of LU$(A-R)$ does not yield numerically stable results in practice, confirming what was observed on tall and skinny matrices. As observed earlier, Cholesky QR does not yield backward stable results on this test set. However, contrary to what was observed in the previous experiment, using refinement on these pathological matrices does not overcome the errors of LU$(A-R)$ on all matrices in this set. Indeed, if $R$ becomes rank deficient, matrix $B$ at step 5 of Algorithm 12 becomes unsuitable for Cholesky factorization.

Altogether, these two sets of experiments demonstrate the numerical stability of TSQR-HR, confirming the theoretical results of Section 6. We demonstrate that the similar approach proposed by Yamamoto is similarly stable. However, we also show example matrices for which the cheaper Householder reconstruction algorithms fail to achieve the same accuracy as TSQR-HR.

## 8. Performance

The complexity analyses in the preceding sections motivate several performance studies. Our experiments in the following subsections will test several hypotheses. First, we study whether TSQR-HR and the conditionally stable approaches in Section 4.4.3 can outperform 1D Householder QR. Next, we evaluate whether the scatter-apply approach and Householder reconstruction can improve the performance of the CAQR trailing matrix update. Lastly, we ask whether two-level aggregation can improve performance, especially for smaller distribution block sizes.

### 8.1. Experimental setup

We targeted two supercomputers, Hopper [23] and Edison [16], at NERSC (Oakland, CA). The following table summarizes some key aspects of their hardware configurations (see Table 7).

All experiments use one MPI process per core, *i.e.*, no multithreading. (Hereafter, 'processor' refers to both an MPI process and a core.)

We compared our implementations with routines from ScaLAPACK [9] (via Intel MKL 11.2.1 and Cray LibSci 13.0.1). MKL and LibSci were pre-installed on Hopper and Edison; we built our codes

**Table 4**
Description of the full matrices used in the experiments.

| Id | Matrix type | Size | $\kappa$ |
|---|---|---|---|
| 1 | $A = 2 * rand(m) - 1$ | 1000-by-1000 | 2.106e + 03 |
| 2 | $A = diag((10 * eps)^{\frac{i}{m}}) * rand(m)$ | 1000-by-1000 | 7.731e + 17 |
| 3 | Golub-Klema-Stewart: $A(i, i) = \frac{1}{\sqrt{i}}, A(i, j > i) = \frac{1}{\sqrt{j}}, A(i, j < i) = 0$ | 1000-by-1000 | 2.242e + 20 |
| 4 | Break 1 distribution: $A = gallery('randsvd', n, 1e9, 2)$ | 1000-by-1000 | 1.00e + 09 |
| 5 | Break 9 distribution: $A = gallery('randsvd', n, 1e9, 1)$ | 1000-by-1000 | 1.00e + 09 |
| 6 | $A = orth(rand(n)) \cdot diag([100, 10, linspace(1e-8, 1e-2, n-2)])$ $A = A \cdot orth(rand(n))$ | 1000-by-1000 | 1.00e + 10 |
| 7 | $v = linspace(1, 1e-3, n); v(51 : end) = 0;$ $A = orth(rand(n)) \cdot diag(v) \cdot orth(rand(n)) + 0.1 * v(50) * rand(n)$ | 1000-by-1000 | 8.464e + 04 |
| 8 | $U\Sigma V^T$ with exponential distribution | 1000-by-1000 | 4.155e + 19 |
| 9 | The devil's stairs matrix | 1000-by-1000 | 2.275e + 19 |
| 10 | KAHAN matrix, a trapezoidal matrix | 1000-by-1000 | 5.642e + 56 |
| 11 | Matrix ARC130 from Matrix Market | 130-by-130 | 6.054e + 10 |
| 12 | Matrix FS_541_1 from Matrix Market | 541-by-541 | 4.468e + 03 |
| 13 | BAART Test problem: Fredholm integral equation of the first kind | 1000-by-1000 | 5.251e + 18 |
| 14 | BLUR Test problem: digital image deblurring. A is a symmetric, doubly block Toeplitz matrix | 961-by-961 | 3.075e + 01 |
| 15 | DERIV2 Test problem: computation of the second derivative | 1000-by-1000 | 1.216e + 06 |
| 16 | Matrix Ex1-CMRS | 1000-by-500 | 1.398e + 17 |
| 17 | Matrix Ex2-RST | 1024-by-512 | 3.276e + 16 |
| 18 | FOXGOOD Test problem: severely ill-posed problem | 1000-by-1000 | 5.742e + 20 |
| 19 | GRAVITY Test problem: 1-D gravity surveying model problem | 1000-by-1000 | 8.797e + 20 |
| 20 | HEAT Test problem: inverse heat equation | 1000-by-1000 | 1.070e + 232 |
| 21 | PARALLAX Stellar parallax problem with 28 fixed, real observations | 26-by-1000 | 4.629e + 14 |
| 22 | PHILLIPS Test problem: discretization of the "famous" first-kind Fredholm integral equation devised by D. L. Phillips | 1000-by-1000 | 2.641e + 10 |
| 23 | SHAW Test problem: one-dimensional image restoration model | 1000-by-1000 | 2.441e + 21 |
| 24 | SPIKES Test problem with a "spiky" solution | 1000-by-1000 | 5.796e + 21 |
| 25 | TOMO is a 2D tomography test problem | 961-by-961 | 1.091e + 17 |

**Table 5**
Residual and orthogonality error of full matrices ($m = 1000$) from LU($A - R$) (Algorithm 11) when computing $T = -UR^{-1}Y_1^{-T}$ or getting $T$ from Algorithm 5. Matrix descriptions can be found in Table 4. Numbers in parentheses indicate the block size for which the maximum error was observed.

| Id | LU($A - R$) ($T = -UR^{-1}Y_1^{-T}$) | | LU($A - R$) ($T$ from Algorithm 5) | |
|---|---|---|---|---|
| | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ |
| 1 | 3.1E−15 (256) | 3.0E−13 (256) | 3.5E−015 (256) | 2.9E−14 (2) |
| 2 | 3.4E−11 (256) | 3.9E−10 (256) | 2.8E−015 (256) | 3.1E−14 (2) |
| 3 | 1.4E119 (256) | 7.0E120 (256) | 0.0E+000 (2) | 0.0E+00 (2) |
| 4 | 5.9E−15 (256) | 6.3E−07 (256) | 4.5E−014 (256) | 2.9E−14 (2) |
| 5 | 3.3E−08 (32) | 7.4E−07 (256) | 4.3E−014 (256) | 2.9E−14 (2) |
| 6 | 8.9E−12 (256) | 1.6E−10 (256) | 1.5E−015 (256) | 2.9E−14 (2) |
| 7 | 2.0E−14 (256) | 3.8E−13 (256) | 2.4E−015 (256) | 2.8E−14 (2) |
| 8 | 2.7E07 (256) | 6.7E07 (256) | 1.9E−015 (256) | 2.8E−14 (2) |
| 9 | 8.2E07 (256) | 5.9E09 (256) | 2.9E−015 (256) | 2.9E−14 (2) |
| 10 | 1.3E61 (256) | 3.8E61 (256) | 0.0E+000 (2) | 0.0E+00 (2) |
| 11 | 5.0E−18 (16) | 9.1E−14 (64) | 4.0E−019 (32) | 1.8E−15 (2) |
| 12 | 7.2E−16 (32) | 1.8E−15 (2) | 6.1E−016 (256) | 1.8E−15 (2) |
| 13 | 3.2E−01 (256) | 9.3E−01 (256) | 3.1E−007 (256) | 2.8E−14 (2) |
| 14 | 8.1E−16 (32) | 1.0E−14 (256) | 8.2E−016 (16) | 3.9E−15 (2) |
| 15 | 5.2E−10 (256) | 1.2E−09 (256) | 1.0E−012 (256) | 5.6E−14 (2) |
| 16 | 6.3E06 (128) | 7.1E10 (128) | 4.3E−007 (256) | 3.4E−14 (2) |
| 17 | 6.9E−10 (256) | 2.5E−05 (256) | 1.4E−014 (256) | 4.5E−14 (2) |
| 18 | 1.3E95 (32) | 1.4E91 (32) | 8.2E−004 (256) | 2.8E−14 (2) |
| 19 | 4.6E259 (16) | 7.5E234 (16) | 2.1E−003 (256) | 2.8E−14 (2) |
| 20 | 1.4E03 (256) | 7.5E06 (256) | 3.9E−002 (256) | 3.4E−14 (2) |
| 21 | 2.1E17 (32) | 4.1E17 (32) | 1.2E−011 (32) | 2.3E−15 (16) |
| 22 | 7.1E−08 (256) | 5.0E−07 (256) | 5.3E−011 (256) | 3.4E−14 (2) |
| 23 | 3.0E170 (16) | 3.5E161 (16) | 7.0E−004 (256) | 2.8E−14 (2) |
| 24 | 6.6E101 (32) | 2.3E92 (32) | 2.3E−003 (256) | 2.8E−14 (2) |
| 25 | 6.2E−15 (64) | 1.7E02 (256) | 2.4E−015 (256) | 2.4E−14 (2) |

using the Intel C++ compiler (version 15.0.1.133) using the `-fast` flag. (For math library support, we linked our codes against both MKL and LibSci.)

All routines use double-precision arithmetic. Our routines and ScaLAPACK's use a block-cyclic matrix layout with square blocks and use column-major order to store each processor's local submatrix.

We test the strong and weak scalability of the implementations. Strong scaling tests an algorithm's ability to solve a fixed problem faster by increasing the available parallelism. Weak scaling tests an algorithm's ability to solve increasingly large problems in the same amount of time (or other resources) by increasing the available parallelism. In strong scaling experiments, we quantify performance by effective flop rate, computed by dividing $2mn^2$ −

**Table 6**
Residual and orthogonality error of full matrices ($m = 1000$). Descriptions of the matrices can be found in Table 4. Errors of $+\infty$ indicate failures within a Cholesky factorization. Numbers in parentheses indicate the block size for which the maximum error was observed.

| Id | TSQR-HR (Algorithm 10) ($T = -US^{-1}Y_1^{-T}$) | | Yamamoto's approach (Algorithm from [37]) | | LU($A - R$) (Algorithm 11) ($T$ from Algorithm 5) | | LU($A - R$) (Algorithm 11) with refinement ($T$ from Algorithm 5) | | CholQR2-HR (Algorithm 12) ($T$ from Algorithm 5) | |
|---|---|---|---|---|---|---|---|---|---|---|
| | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ | $\|A - \tilde{Q}\tilde{R}\|_F / \|A\|_F$ | $\|I - \tilde{Q}^T\tilde{Q}\|_F$ |
| 1 | 4.4E−15 (256) | 2.8E−14 (2) | 4.4E−15 (256) | 2.8E−14 (2) | 3.5E−15 (256) | 2.9E−14 (2) | 3.3E−15 (256) | 2.8E−14 (2) | 3.3E−15 (256) | 2.8E−14 (2) |
| 2 | 2.0E−15 (256) | 2.7E−14 (2) | 2.0E−15 (256) | 2.7E−14 (2) | 2.8E−15 (256) | 3.1E−14 (2) | 2.8E−15 (256) | 2.6E−14 (2) | 2.8E−15 (256) | 2.7E−14 (2) |
| 3 | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | $+\infty$ (256) | $+\infty$ (256) | $+\infty$ (256) | $+\infty$ (256) |
| 4 | 1.0E−14 (256) | 2.9E−14 (2) | 1.0E−14 (256) | 2.9E−14 (2) | 4.5E−14 (256) | 2.9E−14 (2) | 7.1E−15 (256) | 2.8E−14 (2) | $+\infty$ (64) | $+\infty$ (64) |
| 5 | 1.0E−14 (256) | 2.9E−14 (2) | 1.0E−14 (256) | 2.9E−14 (2) | 4.3E−14 (256) | 2.9E−14 (2) | 6.5E−15 (256) | 2.9E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 6 | 1.5E−15 (256) | 2.9E−14 (2) | 1.5E−15 (256) | 2.9E−14 (2) | 1.5E−15 (256) | 2.9E−14 (2) | 1.4E−15 (256) | 2.9E−14 (2) | 1.5E−15 (256) | 2.9E−14 (2) |
| 7 | 1.4E−15 (256) | 2.8E−14 (2) | 1.4E−15 (256) | 2.8E−14 (2) | 2.4E−15 (256) | 2.8E−14 (2) | 2.5E−15 (256) | 2.8E−14 (2) | 2.5E−15 (256) | 2.9E−14 (2) |
| 8 | 2.1E−15 (256) | 2.8E−14 (2) | 2.1E−15 (256) | 2.8E−14 (2) | 1.9E−15 (256) | 2.8E−14 (2) | 1.7E−15 (256) | 2.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 9 | 2.5E−15 (256) | 2.9E−14 (2) | 2.5E−15 (256) | 2.9E−14 (2) | 2.9E−15 (256) | 2.9E−14 (2) | 3.1E−15 (256) | 2.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 10 | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | 0.0E00 (2) | $+\infty$ (256) | $+\infty$ (256) | $+\infty$ (256) | $+\infty$ (256) |
| 11 | 8.8E−19 (16) | 2.1E−15 (2) | 8.8E−19 (16) | 2.1E−15 (2) | 4.0E−19 (32) | 1.8E−15 (2) | 3.3E−19 (32) | 1.5E−15 (2) | 3.8E−19 (32) | 1.8E−15 (2) |
| 12 | 5.9E−16 (64) | 1.9E−15 (256) | 5.9E−16 (64) | 1.9E−15 (256) | 6.1E−16 (256) | 1.8E−15 (2) | 6.1E−16 (256) | 1.8E−15 (2) | 6.1E−16 (256) | 1.8E−15 (2) |
| 13 | 1.6E−15 (32) | 2.9E−14 (2) | 1.6E−15 (32) | 2.9E−14 (2) | 3.1E−07 (256) | 2.8E−14 (2) | 1.3E−15 (256) | 2.9E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 14 | 1.1E−15 (32) | 4.7E−15 (2) | 1.1E−15 (32) | 4.7E−15 (2) | 8.2E−16 (16) | 3.9E−15 (2) | 7.5E−16 (256) | 3.7E−15 (2) | 8.4E−16 (16) | 4.0E−15 (2) |
| 15 | 2.8E−15 (256) | 4.7E−14 (2) | 2.8E−15 (256) | 4.7E−14 (2) | 1.0E−12 (256) | 5.6E−14 (2) | 9.8E−15 (256) | 4.3E−14 (2) | 9.6E−15 (256) | 4.7E−14 (2) |
| 16 | 1.9E−15 (256) | 4.9E−14 (2) | 1.9E−15 (256) | 4.9E−14 (2) | 4.3E−07 (256) | 3.4E−14 (2) | 1.5E−15 (256) | 5.2E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 17 | 3.8E−15 (32) | 5.1E−14 (2) | 3.8E−15 (32) | 5.1E−14 (2) | 1.4E−14 (256) | 4.5E−14 (2) | 1.5E−14 (256) | 4.4E−14 (2) | $+\infty$ (16) | $+\infty$ (16) |
| 18 | 2.4E−15 (256) | 2.8E−14 (2) | 2.4E−15 (256) | 2.8E−14 (2) | 8.2E−04 (256) | 2.8E−14 (2) | 2.6E−15 (32) | 2.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 19 | 2.4E−15 (256) | 2.9E−14 (2) | 2.4E−15 (256) | 2.9E−14 (2) | 2.1E−03 (256) | 2.8E−14 (2) | 1.9E−15 (256) | 2.9E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 20 | 2.7E−15 (256) | 3.9E−14 (2) | 2.7E−15 (256) | 3.9E−14 (2) | 3.9E−02 (256) | 3.4E−14 (2) | 1.3E−15 (256) | 3.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 21 | 8.9E−16 (16) | 2.6E−15 (16) | 8.9E−16 (16) | 2.6E−15 (16) | 1.2E−11 (32) | 2.3E−15 (16) | $+\infty$ (256) | $+\infty$ (256) | $+\infty$ (256) | $+\infty$ (256) |
| 22 | 1.0E−15 (32) | 3.7E−14 (2) | 1.0E−15 (32) | 3.7E−14 (2) | 5.3E−11 (256) | 3.4E−14 (2) | 1.3E−15 (32) | 4.1E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 23 | 2.6E−15 (256) | 2.8E−14 (2) | 2.6E−15 (256) | 2.8E−14 (2) | 7.0E−04 (256) | 2.8E−14 (2) | 2.3E−15 (256) | 2.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 24 | 7.3E−16 (32) | 2.8E−14 (2) | 7.3E−16 (32) | 2.8E−14 (2) | 2.3E−03 (256) | 2.8E−14 (2) | 3.5E−16 (128) | 2.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |
| 25 | 2.2E−15 (256) | 2.3E−14 (2) | 2.2E−15 (256) | 2.3E−14 (2) | 2.4E−15 (256) | 2.4E−14 (2) | 1.2E−15 (256) | 2.8E−14 (2) | $+\infty$ (256) | $+\infty$ (256) |

**Table 7**
Test machines used in performance experiments. (*): a subset of Hopper's nodes actually have 64 GB each.

| | Hopper | Edison |
|---|---|---|
| Make and Model | Cray XE6 | Cray XC30 |
| Compute nodes | 2 × 12-core AMD "Magny-Cours" (2.1 GHz) | 2 × 12-core Intel "Ivy Bridge" (2.4 GHz) |
| L1, L2 caches (private) | 64 KB, 512 KB (per core) | 64 KB, 256 KB (per core) |
| L3 caches (shared) | 6 MB (per 6 cores) | 30 MB (per 12 cores) |
| Memory (per node) | 32* GB DDR3 (1333 MHz) | 64 GB DDR3 (1866 MHz) |
| Interconnect | Cray "Gemini" (3D torus) | Cray "Aries" (Dragonfly) |

**Table 8**
List of 1D routines in our benchmark suite.

| TSQR-HR | Our implementation of Algorithm 10 |
|---|---|
| TSQR-HR-simple | Our implementation of Algorithm 9 |
| 1D-Yamamoto | Our implementation of Yamamoto's 1D approach (see Sections 3.3 and 4.4.2) |
| 1D-pdgeqrf | ScaLAPACK `pdgeqrf`, an implementation of 1D-Householder-QR (Algorithm 4) with block-size $b$ (*i.e.*, using a $b$-by-$b$ block-cyclic layout) |
| CholQR-HR | Our implementation of Algorithm 13 |
| CholQR2-HR | Our implementation of Algorithm 12 |
| TSQR-AR | Algorithm 13 with TSQR instead of Cholesky-QR |
| TSQR-AR2 | Our implementation of Algorithm Algorithm 14 |

$2n^3/3$ by measured runtime, and in weak scaling experiments, we quantify performance by effective flop rate per processor, computed by dividing $(2mn^2 - 2n^3/3)/p$ by measured runtime. Thus, ideal strong scaling would correspond to a line with unit slope, while ideal weak scaling would correspond to a flat horizontal line.

We note that a subset of the experiments on Hopper were reported in our previous work [5], and some of those results do not match those reported in this section. We repeated all benchmarks for consistent data in this paper and highlight median run times as opposed to best observed, as reported earlier. In addition, the software on Hopper, including LibSci, has been updated since our previous collection of data, which we believe accounts for much of the discrepancy. We observed similar performance using MKL and LibSci on both platforms; we only present the MKL data here.

## 8.2. 1D routines

We benchmarked 8 different 1D routines by varying three parameters: $m$, the number of matrix rows, $b$, the distribution block size, and $p$, the number of processors; the number of matrix columns was fixed, $n = b$, and the processors were arranged in a $p$-by-1 grid. A brief description of each routine follows in Table 8. We present data for the case $b = 32$ and several combinations of four problem sizes $m = 512 \cdot (12 \cdot 2^i)^2$ for $i \in \{0, \ldots, 3\}$ and four (sub)machine sizes $p = (12 \cdot 2^j)^2$ for $j \in \{0, \ldots, 3\}$. In particular, we consider seven of the sixteen $(i, j)$ pairs: a strong-scaling experiment, $i = 1$ and $j \in \{0, \ldots, 3\}$, and a weak-scaling experiment, $i = j \in \{0, \ldots, 3\}$. The values of $m$ and $p$ used in our 1D experiments are much larger than in typical panel factorizations within 2D algorithms for square problems on today's machines. However, such large 1D QR factorizations are commonly encountered in sparse iterative methods like GMRES (see, e.g., [27]). Each algorithm was run at least 5 times on at least 3 different allocations. Throughout these experiments, 'different allocations' really means the allocations associated with different jobs submitted to the batch processing system: there is no guarantee in general that different jobs will actually use different nodes.

### 8.2.1. Stable routines

In Fig. 4, we compare TSQR, TSQR-HR, TSQR-HR-simple, 1D-Yamamoto, and 1D-pdgeqrf. (Note that all of these routines except 1D-Yamamoto and TSQR return their $Q$-factors in the usual Householder representation.) Across experiments, TSQR outperforms all other algorithms and 1D-pdgeqrf demonstrates the lowest performance, which is expected due to the higher latency and memory bandwidth costs of Householder QR with respect to TSQR. We also note that TSQR-HR shows improved performance over TSQR-HR-simple on Edison, demonstrating that the optimization of Algorithm 10 over Algorithm 9 presented in Section 4.3 is beneficial in practice. Our implementation of Yamamoto's approach shows comparable performance to TSQR-HR, as expected from the cost analysis given in Table 1. Comparing median runtimes, the largest observed speedup of TSQR-HR over Householder-QR (1D-pdgeqrf) for these experiments is about 3.6× on Edison (with $i = j = 2$) and about 2.7× on Hopper (with $i = j = 3$).

In both weak and strong scaling experiments, we observe that the performance of not only Householder-QR but also the TSQR-based routines does not scale well to the largest processor counts (particularly to 9216). For the TSQR-based routines loss of efficiency is explained by the increase in the number of levels in the binary trees ($O(\log p)$) relative to the number of floating point operations per processor ($O(mb^2/p)$), which stays the same for weak scaling and even decreases for strong scaling. However, scalability to 576 processors already permits efficient panel factorizations for square matrices on square processor grids with a total of $576^2$ (a third of a million) processors.

### 8.2.2. Conditionally stable routines

In Fig. 5, we compare CholQR-HR, CholQR2-HR, TSQR-AR, TSQR-AR2, and TSQR-HR. (Note that TSQR-HR, actually an unconditionally stable routine, is repeated here for comparison.) Currently, our CholQR-HR and CholQR2-HR implementations differ slightly from Algorithms 13 and 12. In CholQR-HR, we perform a reduce followed by a broadcast, instead of an all-reduce (see line 2 of Algorithm 13). Similarly, in CholQR2-HR, we twice perform a reduce followed by a broadcast, instead of two all-reduces (see lines 2 and 5 of Algorithm 12). Thus, we expect that the performance of CholQR-HR and CholQR2-HR could further improve by exploiting these latency-saving optimizations.

We observe across experiments that CholQR-HR performs the best (and is the least stable as shown in Section 7) and TSQR-HR performs the worst (and is the most stable). We also note that Fig. 5 demonstrates the performance cost of iterative refinement, which can be less than 2×. Comparing median runtimes, the largest observed speedup of CholQR-HR over TSQR-HR is about 8.5× on Edison ($i = 1, j = 3$) and about 4.1× on Hopper ($i = 1, j = 0$); additionally, the largest observed speedup of CholQR-HR over 1D-pdgeqrf is about 31.0× on Edison ($i = 1$,

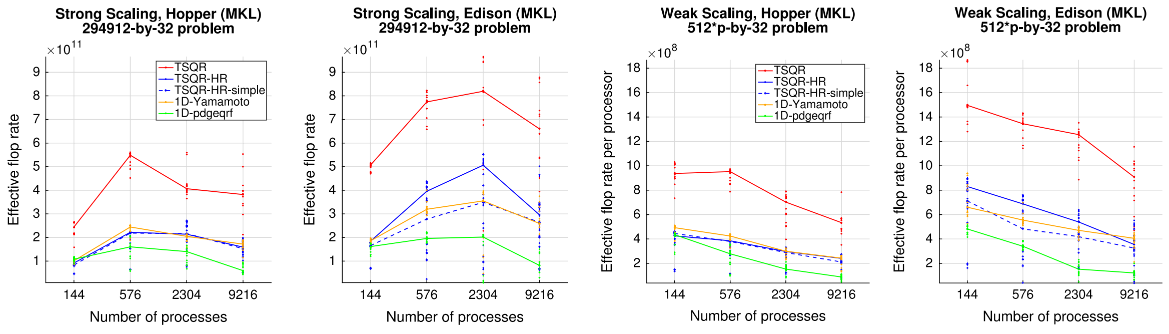


**Fig. 4.** Strong and weak scaling of TSQR-HR, TSQR-HR-simple, 1D-Yamamoto, and 1D-pdgeqrf. Plot lines connect median points.

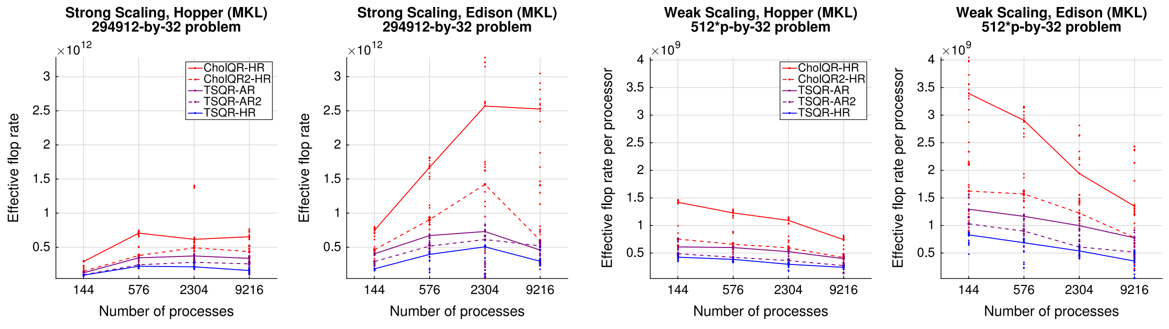


**Fig. 5.** Strong and weak scaling of CholQR-HR, CholQR2-HR, TSQR-AR, TSQR-AR2, and TSQR-HR. Plot lines connect median points.

**Table 9**
List of 2D routines in our benchmark suite.

| CAQR | Our implementation of Algorithm 18 |
|---|---|
| Butterfly-CAQR | Our implementation of Algorithm 20 |
| CAQR-HR | Our implementation of Algorithm 21 |
| 2D-pdgeqrf-ref | 2D-Householder-QR (Algorithm 16) implemented using 1D-pdgeqrf for the panel factorization and our own update (Algorithm 15) |
| 2D-Yamamoto | Our implementation of Yamamoto's 2D approach (see Sections 3.3 and 4.4.2) |
| 2D-pdgeqrf | ScaLAPACK `pdgeqrf`, an implementation of 2D-Householder-QR (Algorithm 16) with block-size $b$ |

$j = 3$) and about 11.0× on Hopper ($i = j = 2$). Thus for tall-skinny factorizations of well-conditioned matrices, conditionally stable routines offer a considerable performance improvement with respect to the performance of any stable algorithm we tested.

## 8.3. 2D routines

We benchmarked 8 different 2D routines by varying three parameters: $m$ and $n$, the number of rows and columns in the matrix, $b$, the distribution block size, and $p_r$ and $p_c$, the number of rows and columns in the process grid (there are $p = p_r p_c$ processors total). A brief description of each routine follows in Table 9. In practice, the problem size parameters $m$ and $n$ are often given by the application, while the user (or software library) is responsible for picking $b$, $p_r$, and $p_c$ to maximize performance: in other words, $b$, $p_r$, and $p_c$ are tuning parameters. In Section 5, we selected $p_r = p_c$ for square matrices and picked a $b$ based on the cost derived expressions. We maintain the square grid selection $p_r = p_c$ for square matrix experiments (Sections 8.3.1, 8.3.2, 8.3.4 and 8.3.5), but tune $b$ over a range of values, since the practical cost tradeoffs are not fully described by our model (e.g. we consider only asymptotic vertical communication costs). In particular, we studied combinations of the four problem sizes $m = 1536 \cdot 12 \cdot 2^i$ for $i \in \{0, \ldots, 3\}$ and the four (sub)machine sizes $p = (12 \cdot 2^j)^2$ for $j \in \{0, \ldots, 3\}$. We consider seven of the sixteen $(i, j)$ pairs: a strong-scaling experiment, $i = 1$ and $j \in \{0, \ldots, 3\}$, and a weak-scaling experiment, $i = j \in \{0, \ldots, 3\}$. For each of these $(i, j)$ pairs and for each $k \in \{3, \ldots, 7\}$, we benchmarked each algorithm with block-size $b = 2^k$ on at least three different allocations, and display results for the value of $k = \log_2(b)$ with the minimal median runtime.

### 8.3.1. Square matrix experiments

In Fig. 6, we compare CAQR-HR, 2D-Yamamoto, CAQR, and 2D-pdgeqrf, each with all possible optimizations enabled. For CAQR-HR and 2D-Yamamoto, we show the best median performance over both one-level and two-level aggregation configurations (see Section 8.3.5 for more details of the performance effects of this tuning). For CAQR, we show the best median performance over using both butterfly trailing matrix updates and binary-tree trailing matrix updates (see Section 8.3.4 for more details of the performance differences of these options).

The overall conclusions from this data are (1) 2D-pdgeqrf outperforms the other approaches almost everywhere, (2) CAQR-HR and 2D-Yamamoto show approximately the same performance, and (3) CAQR is outperformed by the other approaches almost everywhere. The superior performance of 2D-pdgeqrf suggests that the vendor-optimized implementation does not directly reflect 2D-Householder-QR (Algorithm 16), but contains further optimizations. In particular, we expect the MKL implementation of 2D-pdgeqrf overlaps the panel factorization with the trailing matrix update, which is not done in our algorithm descriptions or implementations.

The comparable performance of CAQR-HR and 2D-Yamamoto is expected, as the theoretical costs are identical and both use our implementations of the same building blocks. The inferior performance of CAQR is due to the inefficiency of the trailing matrix updates, which dominate the running time for square matrices. While further optimizations exist for the trailing matrix update (beyond our implementation) using the implicit representation of the orthogonal factors [15], we emphasize that

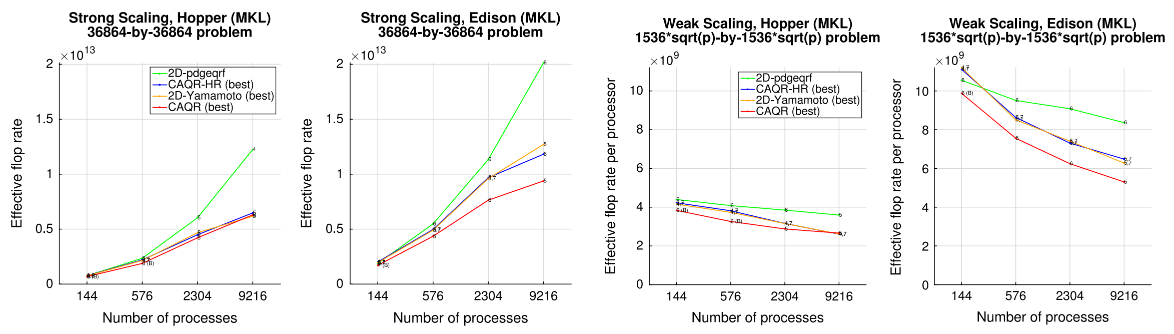


**Fig. 6.** Square experiments: strong and weak scaling of CAQR-HR, 2D-Yamamoto, CAQR, and 2D-pdgeqrf. Plot lines connect median points. Minimal $k = \log_2(b_1)$ and $l = \log_2(b_2)$ values shown.

Householder reconstruction obviates the need for developing and tuning such an implementation.

### 8.3.2. Panel-factorization comparison

In this section, we demonstrate the effect of changing the panel factorization from Householder-QR to TSQR-HR in the context of a 2D algorithm. In Fig. 7, we compare CAQR-HR not only to the vendor optimized 2D-pdgeqrf, but also to 2D-pdgeqrf-ref a routine that implements 2D-Householder-QR (Algorithm 16) by employing 1D-pdgeqrf for the panel factorization and the same trailing matrix update as CAQR-HR (our own implementation). We observe overall that the superiority of the panel factorization within CAQR-HR leads to a speedup over 2D-pdgeqrf-ref for square factorizations, and the effect is more pronounced on Edison than Hopper. The measured speed-ups are small for these square matrix experiments, however, the 1D experiments in Section 8.2 suggest greater benefit would be achieved for larger experiments, as the number of processors in each processor grid column and the number of rows in each matrix panel are increased.

### 8.3.3. Rectangular matrix experiments

Our experiments in this section consider rectangular problems on rectangular process grids, and in Fig. 8 we compare CAQR-HR, CAQR, and 2D-pdgeqrf routines. In particular, we studied combinations of the ten problem sizes $(m, n) = (2304 \cdot 512, 16 \cdot i)$ for $i \in \{1, \dots, 10\}$ and the ten process grids $(p_r, p_c) = (2304, j)$ for $j \in \{1, \dots, 10\}$. We fixed $b = 16$ and considered a weak scaling experiment with the ten $(i, j)$ pairs with $i = j$. We ran each algorithm at least ten times on at least three different allocations.

The idea of this experiment is to vary the amount of time spent in the panel factorization versus the time spent in the trailing matrix update. At the left end of the plot, we start with a matrix with 16 columns distributed over a 1D processor grid; as we move across to the right, we add processor columns, each storing 16 more columns. Thus, at the left end of the plot, the routines spend all their time in the one panel factorization, while at the right end of the plot, the routines spend more time in the trailing matrix updates. As expected, the fastest routine at the left is CAQR (which is just TSQR for the first data point) because it has the fastest panel factorization. However, as the trailing matrix updates become more important, CAQR-HR begins to outperform CAQR, demonstrating the benefit of Householder reconstruction and its efficiency during the trailing matrix update. Likewise, 2D-pdgeqrf is the slowest routine at the left end of the plot but eventually beats CAQR as more processor columns are added. For up to 10 processor columns, CAQR-HR outperforms 2D-pdgeqrf, but we expect that 2D-pdgeqrf would eventually surpass the performance of CAQR-HR based on the results from Section 8.3.1 for square matrices.

The performance improvement of 2D-pdgeqrf relative to CAQR-HR as more processors are added in the weak scaling experiment

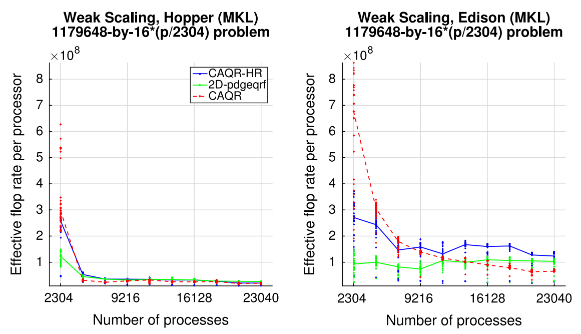


**Fig. 8.** Rectangular experiments: weak scaling of CAQR-HR, CAQR, and 2D-pdgeqrf. Plot lines connect median points.

confirms our hypothesis that the 2D-pdgeqrf implementation exploits overlap between the panel factorization and the trailing matrix update, as this is only achievable with a sufficiently large number of matrix panels. We additionally note this experiment does not reflect the best possible performance achievable by each implementation for the given rectangular matrices, but only the performance with the particular choice of block size, $b = 16$.

### 8.3.4. Improvements to CAQR

In this section we test the effectiveness of the optimization given in Algorithms 19 and 20 of Section 5.3 for CAQR. In Fig. 9, we compare Butterfly-CAQR with CAQR, and for reference we include data from 2D-pdgeqrf. Contrary to the performance observations made in our previous work [5], these experiments show little to no benefit of the scatter-apply update in practice. These differences are partially attributed to improvements in our implementation of CAQR, which removed unnecessary barriers and enabled overlap between the binary-tree TSQR (Algorithm 6) and trailing matrix update (Algorithm 17), alleviating the load imbalance intrinsic to each of these two steps.

### 8.3.5. Two-level aggregation

We benchmarked 3 different two-level 2D routines, each a modification of a "one-level" 2D routine discussed previously (cf. Tables 9 and 10). Relative to their one-level versions, the three two-level 2D algorithms have an additional tuning parameter, the aggregation block size $b_2$. (We sometimes refer to the distribution block size $b$ as $b_1$.) A brief description of each routine follows in Table 10. We repeated the square matrix experiments from Section 8.3. Moreover, we tuned $b_1$ and $b_2$ in tandem on the same problem/grid ($i = j = 2$) that we used to tune $b$ for the one-level algorithms: here, we consider $b_1 = 2^k$ for each $k \in \{3, \dots, 7\}$ and $b_2 = 2^l > b_1$ for each $l \in \{7, 8, 9\}$, and selected the $(b_1, b_2)$ pair with minimal median runtime (for each parameter 4-tuple $(i, j, k, l)$, we ran each algorithm at least once on at least three different allocations).

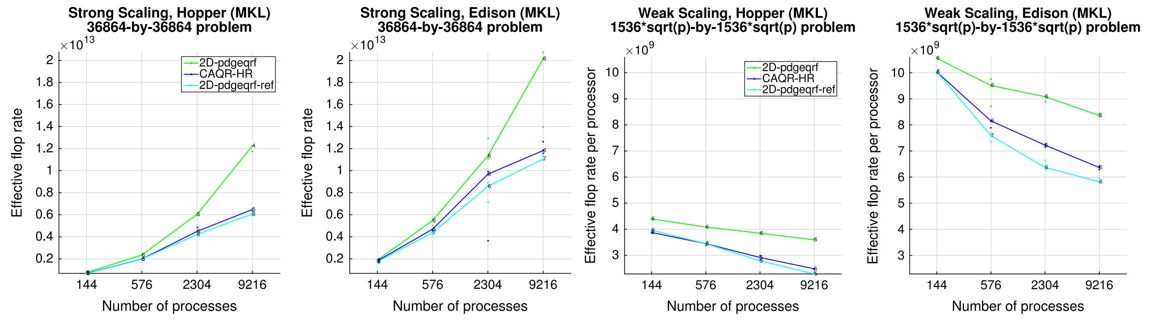


**Fig. 7.** Square experiments: strong and weak scaling of CAQR-HR, 2D-pdgeqrf-ref, CAQR, and 2D-pdgeqrf. Plot lines connect median points. Minimal $k = \log_2(b)$ values shown.

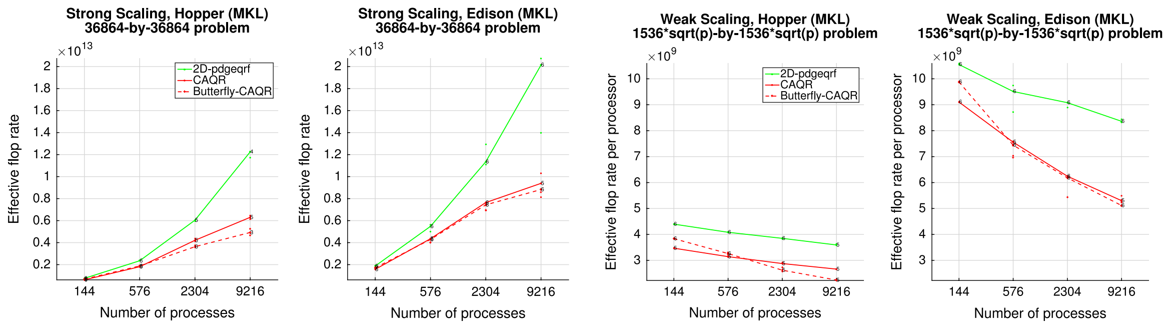


**Fig. 9.** Square experiments: strong and weak scaling of Butterfly-CAQR, CAQR, and 2D-pdgeqrf. Plot lines connect median points. Minimal $k = \log_2(b)$ values shown.

**Table 10**
List of two-level 2D routines in our benchmark suite.

| | |
|---|---|
| Two-Level-2D-Householder-QR | Our implementation of an algorithm described in Section 5.5 that calls 2D-pdgeqrf-ref for the (2D) 'panel' factorization |
| Two-Level-CAQR-HR | Our implementation of Algorithm 23 |
| Two-Level-2D-Yamamoto | Our implementation of a two-level aggregated version of Yamamoto's 2D approach (see Section 5.5) |

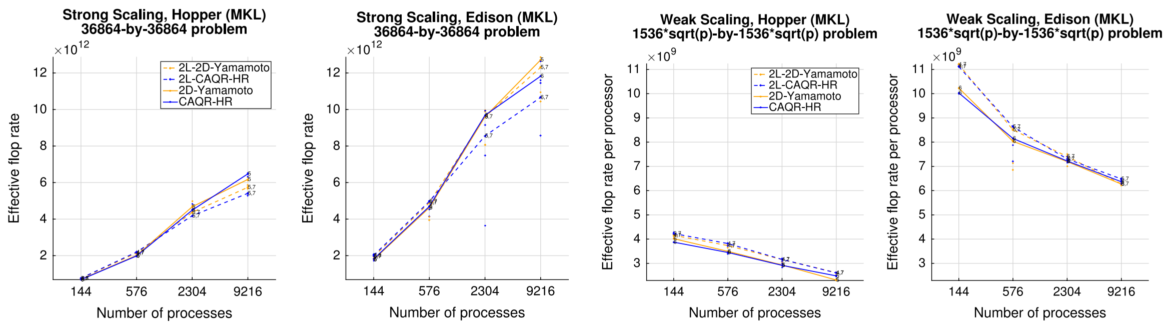


**Fig. 10.** Strong and weak scaling of Two-Level-CAQR-HR, CAQR-HR, Two-Level-2D-Yamamoto and 2D-Yamamoto. Plot lines connect median points. Minimal $k = \log_2(b_1)$ and $l = \log_2(b_2)$ values shown.

In Fig. 10, we compare Two-Level-CAQR-HR with CAQR-HR and Two-Level-2D-Yamamoto with 2D-Yamamoto. The two-level routines are tuned over $b_1 \neq b_2$ (so not inclusive of one-level configurations). Comparing median runtimes, the best performance benefit of Two-Level-CAQR-HR over CAQR-HR is about 1.1× on both Edison and Hopper ($i = 1, j = 0$), and less than 1.1× over 2D-pdgeqrf (same parameters), less pronounced than previously observed [5]. While two-level aggregation helped slightly in all weak scaling tests, it affected performance negatively in the large strong scaling tests, suggesting the second level of blocking should be used selectively.

However, if we restrict the distribution block-size for all algorithms, e.g., to $b_1 = 8$, and pick an aggregation block size $b_2 = 256$, the best performance benefit of Two-Level-CAQR-HR over CAQR-HR improves to about 3.4× on Edison and about 3.1× on Hopper, and the best performance benefit of Two-Level-CAQR-HR over 2D-pdgeqrf improves to about 3.4× on Edison and about 2.4× on Hopper, considering the same parameters $i = 1$ and $j = 0$. This benefit suggests that two-level blocking indeed reduces vertical communication overheads that appear in the single-level algorithms when using a small distribution block size. However, for the given experimental configurations, such a two-level algorithm achieves little or no speed-up over a single-level algorithm with a sufficiently large distribution block size. At this data point, the effective flop rate per processor of Two-Level-CAQR-HR is about 5.3 Gflop/s on Hopper (about 63% of peak) and about 14.0 Gflop/s on Edison (about 72% of peak).

## 9. Conclusion

In this paper, we introduce a method for recovering the Householder basis-kernel representation from any matrix in a stable and efficient manner. Our method was motivated by the desire to combine the efficiency of the TSQR algorithm with that of the trailing matrix updates within Householder-QR in the context of computing the QR factorization of general matrices.

We argue using a performance cost model that the savings from combining the approaches outweigh the extra cost required to reconstruct the Householder vectors for each panel factorization, observing asymptotic improvements over both existing approaches. We also demonstrate that these algorithmic improvements can lead to speedups over tuned library implementations of Householder-QR on real machines and problem sizes. Our experiments demonstrate significant benefits on tall-and-skinny matrices, both with 1D and tall-and-skinny 2D processor grids. For square matrices and processor grids, our panel factorization improvements had much less benefit overall, due to the dominating cost of the trailing matrix update.

Our approach provides a promising direction for heterogenous architectures (as suggested in [37]), where synchronization-avoidance and high granularity computation have even more pervasive effects on performance efficiency. Furthermore, because our approach recovers the standard representation of orthogonal matrices (as is used in libraries like LAPACK), we are able to reuse the existing software infrastructure and maintain performance portability.

Finally, we conjecture that the Householder reconstruction technique will enable the design of a 2.5D QR algorithm which is as stable as Householder QR and further reduces the bandwidth cost compared to parallel CAQR. We aim to reduce the cost for QR as done by Tiskin [35], except in a more practical manner, following the communication-optimal parallel algorithm for LU [31].

## Acknowledgments

We would like to thank Yusaku Yamamoto for sharing his slides from SIAM ALA 2012 with us and all anonymous reviewers of this work for their feedback.

Ballard was supported in part by an appointment to the Sandia National Laboratories Truman Fellowship in National Security Science and Engineering, sponsored by Sandia Corporation (a wholly owned subsidiary of Lockheed Martin Corporation) as Operator of Sandia National Laboratories under its U.S. Department of Energy Contract No. DE-AC04-94AL85000. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We also acknowledge DOE grants DE-SC0008700, DE-SC0010200, AC02-05CH11231, and DARPA grant HR0011-12-2-0016.

## References

[1] E. Agullo, C. Coti, J. Dongarra, T. Herault, J. Langou, QR factorization of tall and skinny matrices in a grid computing environment, in: Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on, 2010, pp. 1–11.
[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J.D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, PA, USA, 1992, also available from http://www.netlib.org/lapack/.
[3] M. Anderson, G. Ballard, J. Demmel, K. Keutzer, Communication-avoiding QR decomposition for GPUs, in: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium IPDPS'11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 48–58.
[4] T. Auckenthaler, T. Huckle, R. Wittmann, A blocked QR-decomposition for the parallel symmetric eigenvalue problem, Parallel Comput. 40 (7) (2014) 186–194. 7th Workshop on Parallel Matrix Algorithms and Applications.
[5] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H.D. Nguyen, E. Solomonik, Reconstructing Householder Vectors from Tall-Skinny QR, in: IEEE 28th International Parallel and Distributed Processing Symposium, 2014, pp. 1159–1170.
[6] G. Ballard, J. Demmel, L. Grigori, M. Jacquelin, H.D. Nguyen, E. Solomonik, Reconstructing householder vectors from Tall-Skinny QR, Tech. Rep., EECS Department, University of California, Berkeley, 2013.
[7] G. Ballard, J. Demmel, O. Holtz, O. Schwartz, Minimizing Communication in Linear Algebra, SIAM J. Matrix Anal. Appl. 32 (3).
[8] C.H. Bischof, X. Sun, On Orthogonal Block Elimination, Tech. Rep. MCS-P450-0794, Argonne National Laboratory, Argonne, IL, 1994.
[9] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, SIAM, Philadelphia, PA, USA, 1997.
[10] E. Chan, M. Heimlich, A. Purkayastha, R. van de Geijn, Collective communication: theory, practice, and experience, Concurrency Comput. Pract. Exp. 19 (13) (2007) 1749–1783.
[11] J. Poulson, B. Marker, R.A. van de Geijn, J.R. Hammond, N.A. Romero, Elemental: A new framework for distributed memory dense matrix computations, ACM Trans. Math. Softw. 39 (2) (2013) 13:1–13:24.
[12] J. Demmel, L. Grigori, M. Gu, H. Xiang, Communication Avoiding Rank Revealing QR Factorization with Column Pivoting, Tech. Rep. UCB/EECS-2013-46, EECS Department, University of California, Berkeley, 2013.
[13] J. Demmel, L. Grigori, M. Hoemmen, J. Langou, Communication-optimal parallel and sequential QR and LU factorizations, SIAM J. Sci. Comput. 34 (1) (2012) A206–A239.
[14] J. Demmel, L. Grigori, M.F. Hoemmen, J. Langou, Communication-Optimal Parallel and Sequential QR and LU Factorizations, Tech. Rep. UCB/EECS-2008-89, EECS Department, University of California, Berkeley, 2008.
[15] J. Dongarra, M. Faverge, T. HéRault, M. Jacquelin, J. Langou, Y. Robert, Hierarchical QR factorization algorithms for multi-core clusters, Parallel Comput. 39 (4–5) (2013) 212–232.
[16] Edison, NERSC's Cray XC30 System, http://www.nersc.gov/users/computational-systems/edison/, 2014.
[17] A. Farley, Broadcast time in communication networks, SIAM J. Appl. Math. 39 (2) (1980) 385–390.
[18] T. Fukaya, Y. Nakatsukasa, Y. Yanagisawa, Y. Yamamoto, CholeskyQR2: A simple and communication-avoiding algorithm for computing a Tall-skinny QR factorization on a large-scale parallel system, in: Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems ScalA'14, IEEE Press, Piscataway, NJ, USA, 2014, pp. 31–38.
[19] S.H. Fuller, L.I. Millett (Eds.), The Future of Computing Performance: Game Over or Next Level?, The National Academies Press, Washington, D.C., 2011, 200 pages. http://www.nap.edu.
[20] G.H. Golub, R.J. Plemmons, A. Sameh, Parallel block schemes for large-scale least-squares computations, in: R.B. Wilhelmson (Ed.), High-speed Computing: Scientific Applications and Algorithm Design, University of Illinois Press, Champaign, IL, USA, 1988, pp. 171–179.
[21] N.J. Higham, Accuracy and Stability of Numerical Algorithms, second ed., SIAM, Philadelphia, PA, 2002.
[22] M. Hoemmen, A Communication-avoiding, hybrid-parallel, rank-revealing orthogonalization method, in: Parallel Distributed Processing Symposium, IPDPS, 2011 IEEE International, 2011, pp. 966–977.
[23] Hopper, NERSC's Cray XE6 System, http://www.nersc.gov/users/computational-systems/hopper/, 2014.
[24] C. Bischof, C. Van Loan, The WY representation for products of Householder matrices, SIAM J. Sci. Stat. Comput. 8 (1).
[25] G. Golub, C. Van Loan, Matrix Computations, in: Johns Hopkins Studies in the Mathematical Sciences, Johns Hopkins University Press, 2012.
[26] R. Schreiber, C. Van Loan, A storage-efficient WY representation for products of householder transformations, SIAM J. Sci. Stat. Comput. 10 (1) (1989) 53–57.
[27] M. Mohiyuddin, M. Hoemmen, J. Demmel, K. Yelick, Minimizing communication in sparse matrix solvers, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis SC'09, ACM, New York, NY, USA, 2009, pp. 36:1–36:12.
[28] D. Mori, Y. Yamamoto, S.-L. Zhang, Backward error analysis of the AllReduce algorithm for Householder QR decomposition, Japan J. Ind. Appl. Math. 29 (1) (2012) 111–130.
[29] C. Puglisi, Modification of the householder method based on compact WY representation, SIAM J. Sci. Stat. Comput. 13 (3) (1992) 723–726.
[30] R. Schreiber, B. Parlett, Block reflectors: Theory and computation, SIAM J. Numer. Anal. 25 (1) (1988) 189–205.
[31] E. Solomonik, J. Demmel, Communication-optimal 2.5D matrix multiplication and LU factorization algorithms, in: Springer Lecture Notes in Computer Science, Proceedings of Euro-Par, Bordeaux, France, 2011, pp. 90–109.
[32] F. Song, H. Ltaief, B. Hadri, J. Dongarra, Scalable tile communication-avoiding QR factorization on multicore cluster systems, in: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis SC'10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–11.
[33] X. Sun, C. Bischof, A basis-kernel representation of orthogonal matrices, SIAM J. Matrix Anal. Appl. 16 (4) (1995) 1184–1196.
[34] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.
[35] A. Tiskin, Communication-efficient parallel generic pairwise elimination, Future Gener. Comput. Syst. 23 (2) (2007) 179–188.
[36] J.L. Träff, A. Ripke, Optimal broadcast for fully connected networks, in: L.T. Yang, O.F. Rana, B. Di Martino, J. Dongarra (Eds.), High Performance Computing and Communications, in: Lecture Notes in Computer Science, vol. 3726, Springer, Berlin, Heidelberg, 2005, pp. 45–56.
[37] Y. Yamamoto, Aggregation of the compact WY representations generated by the TSQR algorithm, in: Conference Talk Presented at SIAM Applied Linear Algebra, 2012.
[38] Y. Yamamoto, Personal communication, 2012.
[39] Y. Yamamoto, Y. Nakatsukasa, Y. Yanagisawa, T. Fukaya, Roundoff Error Analysis of the CholeskyQR2 Algorithm, Tech. Rep. 43, Department of Mathematical Informatics, University of Tokyo, 2014.

**Grey Ballard** is currently a Truman Fellow at Sandia National Labs in Livermore, CA. He received his Ph.D. in 2013 from the Computer Science Division (EECS Department) at the University of California Berkeley. He worked in the BeBOP group and Parallel Computing Laboratory under advisor James Demmel. Before coming to Berkeley, he received his B.S. in Mathematics and Computer Science at Wake Forest University in 2006 and his M.A. in Mathematics at Wake Forest in 2008. His research interests include numerical linear algebra, high performance computing, and computational science, particularly in developing algorithmic ideas that translate to improved implementations and more efficient software. His work has been recognized with the SIAM Linear Algebra Prize and two conference best paper awards, at SPAA and IPDPS, he received the C.V. Ramamoorthy Distinguished Research Award at UC Berkeley, and his Ph.D. thesis was recognized by the ACM Doctoral Dissertation Award—Honorable Mention.

**James Demmel** received his B.S. in Mathematics from Caltech in 1975 and his Ph.D. in Computer Science from UC Berkeley in 1983. After spending six years on the faculty of the Courant Institute, New York University, he joined the Computer Science Division and Mathematics Department at Berkeley in 1990, where he holds a joint appointment.

**Laura Grigori** is a senior research scientist at INRIA in France, where she is leading Alpines group, a joint group between INRIA, University of Pierre and Marie Curie, and CNRS, in Paris. Her field of expertise is in high performance scientific computing, numerical linear algebra, and combinatorial scientific computing. She has performed research in well-renowned institutions, such as INRIA, University of California at Berkeley and Lawrence Berkeley Laboratory. She is leading several projects in preconditioning, communication avoiding algorithms, and associated numerical libraries for large scale parallel/multicore machines, and she is a co-developer of SuperLU_DIST. She is the Program Director of the SIAM special interest group on supercomputing, January 2014–December 2015.

**Mathias Jacquelin** is a postdoctoral research fellow with the Scalable Solvers Group in the Computational Research Division at the Lawrence Berkeley National Laboratory since May 2013. He is currently working on highly scalable algorithms for solving large sparse symmetric linear systems. He obtained a Master's Degree in Computer Science from INSA of Lyon, France. He went on to the Ecole Normale Superieure of Lyon, to complete his Ph.D. in Computer Science, focusing on algorithms and scheduling, and how memory hierarchies should be handled. He then joined INRIA of Saclay, France, working on communication avoiding algorithms for dense linear algebra.

**Nicholas Knight** is a Ph.D. candidate at the University of California, Berkeley, advised by Professor James Demmel. He is interested in algorithmic complexity theory and scientific computing.

**Hong Diep Nguyen** is a postdoctoral scholar at UC Berkeley under the supervision of Prof. Jim Demmel. He did his Ph.D. at the Ecole Normale Superieure de Lyon, France under the supervision of Gilles Villard and Nathalie Revol. He is currently working on "The Reproducibility of Parallel Floating-Point Computations".