SIAM J. SCI. COMPUT.
Vol. 34, No. 1, pp. A206–A239

COMMUNICATION-OPTIMAL PARALLEL AND SEQUENTIAL QR AND LU FACTORIZATIONS*

JAMES DEMMEL†, LAURA GRIGORI‡, MARK HOEMMEN§, AND JULIEN LANGOU¶

Abstract. We present parallel and sequential dense QR factorization algorithms that are both *optimal* (up to polylogarithmic factors) in the amount of communication they perform and just as *stable* as Householder QR. We prove optimality by deriving new lower bounds for the number of multiplications done by "non-Strassen-like" QR, and using these in known communication lower bounds that are proportional to the number of multiplications. We not only show that our QR algorithms attain these lower bounds (up to polylogarithmic factors), but that existing LAPACK and ScaLAPACK algorithms perform asymptotically more communication. We derive analogous communication lower bounds for LU factorization and point out recent LU algorithms in the literature that attain at least some of these lower bounds. The sequential and parallel QR algorithms for tall and skinny matrices lead to significant speedups in practice over some of the existing algorithms, including LAPACK and ScaLAPACK, for example, up to 6.7 times over ScaLAPACK. A performance model for the parallel algorithm for general rectangular matrices predicts significant speedups over ScaLAPACK.

Key words. linear algebra, QR factorization, LU factorization

AMS subject classifications. 15A23, 65F25, 65F20

DOI. 10.1137/080731992

1. Introduction.

The large and increasing costs of communication motivate redesigning algorithms to avoid it whenever possible. In the parallel case, communication refers to messages between processors, which may be sent over a network or via a shared memory. In the sequential case, communication refers to data movement between different levels of the memory hierarchy. In both the parallel and sequential cases we model the time to communicate a message of n words as $\alpha + \beta n$, where α is the latency and β is the reciprocal bandwidth. Many authors have pointed out technology trends causing floating point to become faster at an exponentially higher rate than bandwidth, and bandwidth at an exponentially higher rate than latency (see, e.g., Graham, Snir, and Patterson [25]). Thus the importance of minimizing communication will only grow.

We present parallel and sequential dense QR factorization algorithms that both (1) attain known communication lower bounds (sometimes only up to polylogarithmic factors) for the total number of words moved and the total number of messages, and (2) are just as numerically stable as conventional Householder QR. Some of the algorithms are novel, and some extend earlier work. The first set of algorithms, "Tall Skinny QR" (TSQR), are for matrices with many more rows than columns, and the second set, "Communication-Avoiding QR" (CAQR), are for general rectangular matrices. The

*Submitted to the journal's Methods and Algorithms for Scientific Computing section August 4, 2008; accepted for publication (in revised form) October 11, 2011; published electronically February 2, 2012.

http://www.siam.org/journals/sisc/34-1/73199.html

†Mathematics and EECS, University of California, 831 Evans Hall, Berkeley, CA 94720 (demmel@eecs.berkeley.edu).

‡Laboratoire de Recherche en Informatique, INRIA Saclay-Ile de France, Bat 490, Universite Paris-Sud 11, Orsay, 91405 France (Laura.Grigori@inria.fr).

§Sandia National Laboratories, PO Box 500, Albuquerque, NM 87185 (mhoemme@sandia.gov).

¶Department of Mathematical Sciences, University of Colorado at Denver and Health Sciences Center, Denver, CO 80202 (julien.langou@ucdenver.edu).

TABLE 1.1

Performance models of parallel TSQR, ScaLAPACK's parallel QR factorization `PDGEQRF`*, sequential TSQR, and blocked sequential Householder QR on an $m \times n$ matrix, along with lower bounds on the number of flops, words, and messages. The parallel algorithms assume P processors and $m/P \geq n$. The sequential algorithms assume fast memory size W, $m \gg n$, and $W \geq 3n^2/2$. $\widetilde{W} = W - n(n+1)/2$, which is at least about $\frac{2}{3}W$. The sequential Householder QR algorithm analyzed here is ScaLAPACK's out-of-DRAM* `PFDGEQRF`*, which requires that the entire panel and at least one column of the trailing matrix fit in fast memory: $mb > W$, for panel width b. In Appendix* A*, we analyze LAPACK's in-DRAM blocked sequential QR factorization* `DGEQRF`*, which lacks that restriction; there, we show that* `DGEQRF` *as well requires asymptotically more memory traffic than sequential CAQR.*

	Par. TSQR	`PDGEQRF`	Lower bound
# flops	$\frac{2mn^2}{P} + \frac{2n^3}{3}\log P$	$\frac{2mn^2}{P} - \frac{2n^3}{3P}$	$\Theta\left(\frac{mn^2}{P}\right)$
# words	$\frac{n^2}{2}\log P$	$\frac{n^2}{2}\log P$	$\frac{n^2}{2}\log P$
# messages	$\log \mathbf{P}$	$\mathbf{2n}\log \mathbf{P}$	$\log \mathbf{P}$
	Seq. TSQR	Householder QR	Lower bound
# flops	$2mn^2$	$2mn^2$	$\Theta(mn^2)$
# words	$2mn$	$\frac{m^2n^2}{2W}$	$2mn$
# messages	$2mn/\widetilde{W}$	$mn^2/2W$	$2mn/W$

algorithms have significantly lower latency cost (i.e., fewer messages) in the parallel case, and significantly lower latency and bandwidth costs (i.e., fewer words moved) in the sequential case, than existing algorithms in LAPACK [1] and ScaLAPACK [8].

It will be easy to see that our parallel and sequential TSQR implementations communicate as little as possible (see Table 1.1 and section 4). For CAQR we need a different approach: in [7] lower bounds on communication (both the total number of words moved and the total number of messages) are derived for a variety of dense linear algebra algorithms, including QR and LU, assuming they are implemented using non-Strassen-like algorithms, and assuming the QR algorithm satisfies certain technical assumptions, discussed more carefully in section 4. All these lower bounds are proportional to the number of multiplications performed. Here (see section 4) we derive lower bounds on the number of multiplications required by any non-Strassen-like implementation of QR (in a sense made formal later). These lower bounds are within modest constant multiples of the number of operations used by current algorithms. This in turn provides the communication lower bounds in Tables 1.2 and 1.3. We show that CAQR attains these lower bounds (sometimes only up to polylogarithmic factors).

Tables 1.1 through 1.3 summarize our performance models and lower bounds for TSQR, CAQR, and the sequential and parallel blocked QR factorizations represented in LAPACK, respectively, ScaLAPACK. Our model of computation looks the same for the parallel and sequential cases, with $running\ time = \#flops \times time\_per\_flop + \#words\_moved \times (1/bandwidth) + \#messages \times latency$, where the last two terms constitute the communication. We do not model overlap of communication and computation, which while important in practice can at most improve the running time by a factor of 2, whereas we are looking for asymptotic improvements. In the tables we give the #flops, #words moved, and #messages as functions of the number of rows m and columns n (assuming $m \geq n$), the number of processors P in the parallel case, and the size of fast memory W in the sequential case. In Tables 1.1 and 1.3, we make some assumptions on W for simplicity. We relax these assumptions in Appendix A and show that nevertheless blocked Householder QR as implemented in LAPACK's `DGEQRF` routine cannot be optimal, unlike sequential CAQR. Furthermore, to make these tables

TABLE 1.2

Performance models of parallel CAQR and ScaLAPACK's parallel QR factorization PDGEQRF *on an $m \times n$ matrix and a square $n \times n$ matrix with P processors, along with lower bounds on the number of flops, words, and messages. The matrix is stored in a two-dimensional $P_r \times P_c$ block cyclic layout with square $b \times b$ blocks. We choose b, P_r, and P_c optimally and independently for each algorithm.*

$m \times n$ matrix	Par. CAQR	PDGEQRF	Lower bound
# flops	$\frac{2mn^2}{P} + \frac{2n^3}{3}$	$\frac{2mn^2}{P} + \frac{2n^3}{3}$	$\Theta\left(mn^2/P\right)$
# words	$\sqrt{\frac{mn^3}{P}} \log P - \frac{1}{4}\sqrt{\frac{n^5}{mP}} \log\left(\frac{nP}{m}\right)$	$\sqrt{\frac{mn^3}{P}} \log P - \frac{1}{4}\sqrt{\frac{n^5}{mP}} \log\left(\frac{nP}{m}\right)$	$\Theta\left(\sqrt{mn^3}/\sqrt{P}\right)$
# messages	$\mathbf{\frac{1}{4}\sqrt{\frac{nP}{m}} \log^2\left(\frac{mP}{n}\right) \cdot \log\left(P\sqrt{\frac{mP}{n}}\right)}$	$\mathbf{\frac{n}{4} \log\left(\frac{mP^5}{n}\right) \log\left(\frac{mP}{n}\right)}$	$\mathbf{\Theta\left(\sqrt{nP}/\sqrt{m}\right)}$
$n \times n$ matrix			
# flops	$4n^3/3P$	$4n^3/3P$	$\Theta\left(n^3/P\right)$
# words	$(3n^2/4\sqrt{P}) \log P$	$(3n^2/4\sqrt{P}) \log P$	$\Theta\left(n^2/\sqrt{P}\right)$
# messages	$\mathbf{(3/8)\sqrt{P} \log^3 P}$	$\mathbf{(5n/4) \log^2 P}$	$\mathbf{\Theta\left(\sqrt{P}\right)}$

TABLE 1.3

Performance models of sequential CAQR and blocked sequential Householder QR on an $m \times n$ matrix and on a square $n \times n$ matrix with fast memory size W, along with lower bounds on the number of flops, words, and messages. As in Table 1.1*, the sequential Householder QR algorithm analyzed here is ScaLAPACK's out-of-DRAM* PFDGEQRF*, which requires that at least two columns of the matrix fit in fast memory (i.e., that $W \geq 2m$). See Appendix* A *for an analysis of LAPACK's in-DRAM blocked sequential QR factorization* DGEQRF*, which does not have this requirement yet also requires asymptotically more memory traffic than sequential CAQR.*

$m \times n$ matrix	Seq. CAQR	Householder QR	Lower bound
# flops	$2mn^2 - \frac{2n^3}{3}$	$2mn^2 - \frac{2n^3}{3}$	$\Theta(mn^2)$
# words	$\mathbf{3\frac{mn^2}{\sqrt{W}}}$	$\mathbf{\frac{m^2n^2}{2W}}$	$\mathbf{\Theta\left(\frac{mn^2}{\sqrt{W}}\right)}$
# messages	$\mathbf{12\frac{mn^2}{W^{3/2}}}$	$\mathbf{\frac{mn^2}{2W}}$	$\mathbf{\Theta\left(\frac{mn^2}{W^{3/2}}\right)}$
$n \times n$ matrix			
# flops	$\frac{4n^3}{3}$	$\frac{4n^3}{3}$	$\Theta(n^3)$
# words	$\mathbf{3\frac{n^3}{\sqrt{W}}}$	$\mathbf{\frac{n^4}{3W}}$	$\mathbf{\Theta\left(\frac{n^3}{\sqrt{W}}\right)}$
# messages	$\mathbf{12\frac{n^3}{W^{3/2}}}$	$\mathbf{\frac{n^3}{2W}}$	$\mathbf{\Theta\left(\frac{n^3}{W^{3/2}}\right)}$

easier to read: we omit most lower-order terms, make boldface the terms where the new algorithms differ significantly from Sca/LAPACK, and make the optimal choice of matrix layout for each parallel algorithm. The latter means optimally choosing the block size b as well as the processor grid dimensions $P_r \times P_c$ in the two-dimensional block cyclic layout. (See section 3 for discussion of these parameters and detailed performance models for general layouts.) We note that although our new algorithms perform slightly more floating point operations than LAPACK and ScaLAPACK, they have the same highest order terms in their floating point operation counts.

Tables 1.1 and 1.2 present the parallel performance models for TSQR, CAQR on general rectangular matrices, and CAQR on square matrices, respectively. First, Table 1.1 shows that parallel TSQR requires only $\log P$ messages, which is both optimal and a factor $2n$ smaller than for ScaLAPACK's parallel QR factorization

PDGEQRF. Parallel TSQR and PDGEQRF both communicate the minimimum number of words possible, namely, $\frac{n^2}{2} \log P$. Parallel TSQR performs more flops, but this represents a lower order term when n is small compared to m/P. Table 1.2 shows that parallel CAQR needs only $\Theta(\sqrt{nP/m})$ messages (ignoring polylogarithmic factors) on a general $m \times n$ rectangular matrix, which is both optimal and a factor $\Theta(\sqrt{mn/P})$ fewer messages than ScaLAPACK. Note that $\sqrt{mn/P}$ is the square root of the size of the local matrix stored in each processor's local memory, up to a small constant factor. Table 1.2 also presents the same comparison for the special case of a square $n \times n$ matrix.

Next, Tables 1.1 and 1.3 present the sequential performance models for TSQR and CAQR on general rectangular matrices and CAQR on square matrices, respectively. Table 1.1 compares sequential TSQR with sequential blocked Householder QR. These tables show performance models for ScaLAPACK's out-of-DRAM QR factorization routine PFDGEQRF, for which fast memory is DRAM and slow memory is disk. Appendix A shows the slightly different memory traffic model for LAPACK's QR factorization routine DGEQRF, for which fast memory is cache and slow memory is DRAM. Sequential TSQR transfers fewer words between slow and fast memory: $2mn$, which is both optimal and a factor $mn/(4W)$ fewer words than transferred by blocked Householder QR. Note that mn/W is how many times larger the matrix is than the fast memory size W. Furthermore, TSQR requires fewer messages: at most about $3mn/W$, which is close to optimal and $\Theta(n)$ times lower than Householder QR.

Table 1.3 compares sequential CAQR and sequential blocked Householder QR on a general rectangular matrix. Sequential CAQR transfers fewer words between slow and fast memory: $\Theta(mn^2/\sqrt{W})$, which is both optimal and a factor $\Theta(m/\sqrt{W})$ fewer words transferred than blocked Householder QR. Note that $m/\sqrt{W} = \sqrt{m^2/W}$ is the square root of how many times larger a square $m \times m$ matrix is than the fast memory size W. Sequential CAQR also requires fewer messages: $12mn^2/W^{3/2}$, which is optimal. The analysis of the sequential blocked Householder QR algorithm in Tables 1.1 and 1.3 assumes for simplicity that that at least two columns of the matrix fit in fast memory (that is, $W \geq 2m$); Appendix A relaxes this restriction but nevertheless shows that sequential blocked Householder QR requires asymptotically more memory traffic than sequential CAQR. Finally, Table 1.3 also presents the same comparison for the special case of a square $n \times n$ matrix.

We also describe the implementation and performance results of these algorithms. The efficient implementation of the QR factorizations of tall and skinny matrices distributed in a one-dimensional layout is very important, since this operation arises in a wide range of applications. We cite three important examples. First, block iterative methods frequently compute the QR factorization of a tall and skinny dense matrix. This includes algorithms for solving linear systems $Ax = B$ with multiple right-hand sides, as well as block iterative eigensolvers. Second, recent research has reawakened an interest in alternate formulations of Krylov subspace methods, called *s-step Krylov methods*, in which some number s steps of the algorithm are performed all at once, in order to reduce communication. For new s-step methods and a literature review, see Hoemmen [30]. Some s-step methods, after generating a basis for the Krylov subspace, use a QR factorization to orthogonalize the basis vectors. This is an ideal application for TSQR and in fact inspired its (re)discovery. Third, Householder QR decompositions of tall and skinny matrices also make up the panel factorization step for typical QR factorizations of matrices in a more general, two-dimensional layout. This includes the current parallel QR factorization routine PDGEQRF in ScaLAPACK, as well

as ScaLAPACK's out-of-DRAM QR factorization `PFDGEQRF` [15]. Both algorithms use a standard column-based Householder QR for the panel factorizations, but in the parallel case this is a latency bottleneck, and in the out-of-DRAM case it is a bandwidth bottleneck. Our CAQR algorithm for computing the QR factorization of general rectangular matrices uses TSQR for its panel factorization and removes the latency bottleneck in the parallel case and the bandwidth bottleneck in the sequential case.

The main insight behind the TSQR algorithm is to perform the QR factorization of a tall skinny matrix as a reduction operation over row blocks. This idea itself is not novel (see, for example, [2, 9, 13, 24, 28, 35, 41, 42, 43]), but we have a number of optimizations and generalizations:

- Our algorithms can perform almost all their floating-point operations using any fast sequential QR factorization routine. For example, we can use blocked Householder transformations exploiting BLAS 3 operations, or invoke Elmroth and Gustavson's recursive QR (see [20, 21]).
- We use TSQR as a building block for CAQR. Any reduction tree can be used by TSQR during the panel factorization, and this triggers the update of the trailing matrix in CAQR. We discuss in particular the parallel, respectively, sequential, CAQR factorization of arbitrary rectangular matrices in a two-dimensional block cyclic layout.
- Most significantly, we prove optimality for both our parallel and sequential algorithms, with a one-dimensional layout for TSQR and two-dimensional block layout for CAQR, i.e., that they minimize bandwidth and latency costs. This assumes $\Theta(n^3)$ (non-Strassen-like) algorithms and is usually shown in an asymptotic or "big O" sense.
- We describe special cases in which existing sequential algorithms by Elmroth and Gustavson [21] and also LAPACK's `DGEQRF` attain minimum bandwidth, though not minimum latency.
- We observe that there are alternative LU algorithms in the literature that attain at least some of these communication lower bounds: [26, 27] describe stable parallel and sequential LU algorithms attaining both bandwidth and latency lower bounds.
- We outline how to extend both algorithms and optimality results to certain kinds of hierarchical architectures, with either multiple levels of memory hierarchy or multiple levels of parallelism (e.g., where each node in a parallel machine consists of other parallel machines, such as multicore). In the case of TSQR we do this by adapting it to work on general reduction trees.

We note that the Q factor will be represented as a tree of smaller Q factors, which differs from the traditional layout. Many previous authors did not explain in detail how to apply a stored TSQR Q factor, quite possibly because this is not required for solving a single least squares problem. However, many of our applications require storing and working with the implicit representation of the Q factor. Our performance models show that applying this tree-structured Q has about the same cost as the traditionally represented Q.

The rest of this paper is organized as follows. Section 2 presents TSQR, describing its parallel and sequential optimizations, performance models, comparisons to LAPACK and ScaLAPACK, and how it can be adapted to other architectures. Section 3 presents CAQR analogously. Section 4 presents our lower bounds for TSQR and for CAQR (as well as LU). Section 6 describes related work. Section 7 describes open problems and future work. Finally, Appendix A extends the memory traffic analysis of sequential blocked Householder QR to LAPACK's QR factorization `DGEQRF`. This

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_0 R_0 \\ Q_1 R_1 \\ Q_2 R_2 \\ Q_3 R_3 \end{pmatrix} \qquad \left(\begin{array}{c} R_0 \\ R_1 \\ \hline R_2 \\ R_3 \end{array}\right) = \left(\begin{array}{c} \begin{pmatrix} R_0 \\ R_1 \end{pmatrix} \\ \hline \begin{pmatrix} R_2 \\ R_3 \end{pmatrix} \end{array}\right) = \left(\begin{array}{c} Q_{01} R_{01} \\ \hline Q_{23} R_{23} \end{array}\right)$$

(a) First step of parallel TSQR (b) Second step of parallel TSQR

FIG. 2.1. *First two steps of parallel TSQR.*

paper is based on the technical report [16], to which we leave many of the detailed derivations of the algorithms and performance models.

2. TSQR. In this section, we present TSQR, a family of algorithms for the QR factorization of an $m \times n$ matrix A, stored in a one-dimensional block row layout[1] $A = [A_1; A_2; \ldots; A_p]$, where A_i is $m_i \times n$. We use MATLAB-style notation to indicate that the row blocks A_i are vertically stacked. We assume $m_i \geq n$ and typically $m \gg n$ (hence "tall and skinny"). Our parallel TSQR algorithm requires only $\log P$ messages on P processors, as opposed to $\Theta(n \log P)$ messages for standard Householder QR. Our sequential TSQR algorithm only moves $2mn$ words between fast and slow memory, which is optimal and asymptotically less than sequential Householder QR.

We begin in section 2.1 by describing parallel TSQR. Section 2.2 shows sequential TSQR. Section 2.3 illustrates how differently shaped trees can be used to construct more general TSQR algorithms that improve performance on different computer architectures. Section 2.4 illustrates the three primitive operations on which all TSQR algorithms are based and shows how these can be implemented to save memory, avoid unnecessary floating-point operations, and improve performance. Finally, section 2.5 shows that TSQR communicates less than numerically equally stable QR factorizations and is about as fast as the fastest numerically unstable alternative.

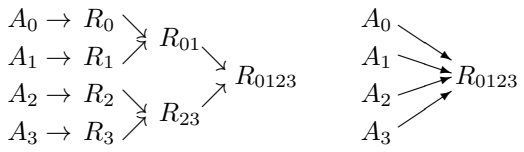
2.1. Parallel TSQR. In this section, we show a simple parallel TSQR algorithm. We begin by illustrating the algorithm for the case of $P = 4$ processors. Later, we will give the general version of the algorithm and derive its performance model. Suppose that the $m \times n$ input matrix A is divided into four row blocks $A = [A_0; A_1; A_2; A_3]$, where A_i is $m_i \times n$ with $m_i \geq n$. Each row block A_i is stored on a different processor[2] i. First, each processor computes the QR factorization of its row block, as in Figure 2.1(a). Then, processors work in pairs, combining their local R factors by computing the QR factorization of the $2n \times n$ matrix $[R_0; R_1]$, respectively, $[R_2; R_3]$, as in Figure 2.1(b). Thus, $[R_0; R_1]$ is replaced by R_{01} and $[R_2; R_3]$ is replaced by R_{23}. Here and later, the subscripts on a matrix like R_{ij} refer to the original row blocks A_i and A_j on which they depend. Finally, the $2n \times n$ QR factorization $[R_{01}; R_{23}] = Q_{0123} R_{01234}$ is computed.

We claim that R_{0123} is the R factor in the QR factorization of the original matrix $A = [A_0; A_1; A_2; A_3]$. To see this, we combine all steps above into (2.1), which expresses the entire computation as a product of intermediate orthonormal factors:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_0 & & & \\ \hline & Q_1 & & \\ \hline & & Q_2 & \\ \hline & & & Q_3 \end{array}\right) \cdot \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{23} \end{array}\right) \cdot Q_{0123} \cdot R_{0123}. \tag{2.1}$$

[1] See the *ScaLAPACK Users' Guide* [8] for a description of one-dimensional and two-dimensional layouts.

[2] We present the parallel algorithm using a distributed-memory programming model, but the algorithm is equally valid in a shared-memory model.



(a) Parallel TSQR (b) Householder QR

FIG. 2.2. *Graphical comparison of parallel TSQR and Householder QR.*

For this product to make sense, we must choose the dimensions of the intermediate Q factors consistently. They can all be square, or when all $m_i \geq n$, they can all have n columns (in which case each R factor will be $n \times n$). (The usual representation of Q factors by Householder reflectors encodes both possibilities.) In either case, we have expressed A as a product of (block diagonal) orthogonal matrices (which must therefore also be orthogonal) and the triangular matrix R_{0123}. By uniqueness of the QR decomposition (modulo signs of diagonal entries of R_{0123}), this is the QR decomposition of A.

We abbreviate this algorithm with the simple notation of Figure 2.2(a), which makes the binary tree apparent. The notation has the following meaning: if one or more arrows point to the same matrix, that matrix is the R factor of the matrix obtained by stacking all the matrices at the other ends of the arrows atop one another. This notation not only helps visualize the parallelism in the algorithm (all QR decompositions at the same depth in the tree may be done in parallel), but implies that *any* tree leads to a valid QR decomposition. For example, the conventional Householder QR decomposition may be expressed as the "trivial" tree in Figure 2.2(b).

Algorithm 1. Parallel TSQR factorization.

Require: Π is the set of P processors; my processor's index is i
Require: Tree with P leaves and height L, describing communication pattern
Require: $m \times n$ matrix A in block row layout; A_i is processor i's block
1: Compute QR factorization $A_i = Q_{i,0}R_{i,0}$
2: **for** k from 1 to L = **do**
3: **if** I have $q - 1 > 0$ neighbors in the tree at this level = **then**
4: Send $R_{i,k-1}$ to each neighbor not myself
5: Receive $R_{j,k-1}$ from each neighbor j not myself
6: Stack the $R_{j,k-1}$ from all neighbors (incl. my $R_{i,k-1}$), in j order, into the $qn \times n$ matrix $C_{i,k}$, and factor $C_{i,k} = Q_{i,k}R_{i,k}$
7: **else**
8: $R_{i,k} := R_{i,k-1}$, and $Q_{i,k} := I_{n\times n}$ ▷ The latter is stored implicitly
9: **end if**
10: **end for**
Ensure: $R_{i,L}$ is the R factor of A, for all processors i
Ensure: Q factor is implicitly represented by the tree of intermediate Q factors $\{Q_{i,k}\}$: $i \in \Pi$, $k \in \{0, 1, \ldots, L\}\}$.

Now, we describe the parallel TSQR factorization algorithm in more detail and show how to apply the Q factor or compute its explicit representation. We also give performance models of each. Algorithm 1 shows the parallel TSQR factorization of a matrix A distributed in a block row layout among P processors. It computes an R factor which is duplicated over all P processors and a Q factor which is stored implicitly in a distributed way.

On Line 1 of Algorithm 1, each processor computes a QR factorization of an $m/P \times n$ matrix, at a cost of $2mn^2/P - 2n^3/3$ flops. If we use a binary tree to factor the matrix, and if P is a power of two, then each subsequent stage of the factorization involves pairs of processors exchanging $n \times n$ upper triangular factors, and redundantly computing the QR factorization of the $2n \times n$ matrix formed by the two upper triangular matrices. If we count operations only along the critical path of the algorithm, each of $\log P$ stages requires one message of $n(n-1)/2$ words, and $2n^3/3$ floating-point operations. Thus, the run time of TSQR is estimated to be

$$\text{Time}_{\text{Par. TSQR}}(m,n,P) = \left(\frac{2mn^2}{P} + \frac{2n^3}{3}\log P\right)\gamma + \left(\frac{1}{2}n^2 \log P\right)\beta + (\log P)\,\alpha, \tag{2.2}$$

where γ is the floating-point throughput, β the inverse message bandwidth, and α the message latency. For detailed derivations of the floating-point operation counts, see Demmel et al. [16].

Algorithm 1 stores the Q factor implicitly as a tree of intermediate factors. To apply the resulting Q factor to an $m \times r$ matrix (with $r \leq m_i$ for all i), we execute a similar procedure as the factorization but apply the tree in reverse order of its computation (from root to leaves, rather than from leaves to root). Applying the transpose or the conjugate transpose (in the complex arithmetic case) of the implicitly stored Q factor involves applying the tree in the same order as in the factorization (from leaves to root). The performance model for each of these cases is similar to that of the factorization (equation (2.2)). Details are in Demmel et al. [16]. Computing an explicit representation of the Q factor as an $m \times n$ matrix requires applying the implicitly represented Q factor to the first n columns of the $m \times n$ identity matrix (distributed in block row fashion so that processor i gets an $m_i \times n$ row block). It is not possible, as far as we know, to compute the explicit Q factor in place (in a way analogous to LAPACK's `ORGQR` routine).

2.2. Sequential TSQR. In this section, we show a sequential TSQR algorithm. We begin by illustrating the algorithm for the case of $P = 4$ row blocks. Then, we derive a performance model for the general case, where fast memory has a capacity of W floating-point words. We show an example for the case of $P = 4$ row blocks, so that $A = [A_0; A_1; A_2; A_3]$. Sequential TSQR begins by computing the QR factorization of A_0, as in Figure 2.3(a). It then computes the QR factorization $[R_0; A_1] = Q_{01}R_{01}$, as in Figure 2.3(b). The algorithm continues this process for each row block in turn, first with the QR factorization $[R_{01}; A_2] = Q_{012}R_{012}$, and finally with the QR factorization $[R_{012}; A_3] = Q_{0123}R_{0123}$. One can show using a product representation analogous to (2.1) that R_{0123} is the R factor in the QR factorization of A and that the tree of intermediate Q factors implicitly represents the Q factor in the QR factorization of A.

Using the visual notation introduced in section 2.1, sequential TSQR corresponds to the "flat tree" of Figure 2.4. The idea of sequential TSQR is that if fast memory can only hold a little more than a fraction m/P of the rows of A (a little more than $m/4$ for the above example), then all computations occur in fast memory, and the algorithm need only read the entire matrix once from slow memory and write it back once. This is the minimal amount of data movement possible for a QR factorization. We omit writing out the sequential TSQR algorithm here; it is analogous to Algorithm 1. Sequential TSQR performs the same number of floating-point operations

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_0 R_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} \qquad \begin{pmatrix} R_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} \begin{pmatrix} R_0 \\ A_1 \end{pmatrix} \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{01} R_{01} \\ \\ A_2 \\ A_3 \end{pmatrix}$$

(a) First step of sequential TSQR (b) Second step of sequential TSQR

FIG. 2.3. *First two stages of sequential TSQR.*

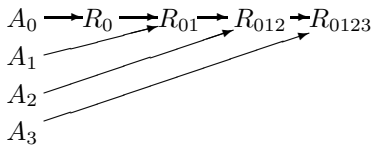


FIG. 2.4. *Graphical representation of sequential TSQR.*

as sequential Householder QR, namely, $2mn^2 - 2n^3/3$ flops. However, it performs less communication than Householder QR, as we will show in section 2.5. Sequential TSQR transfers $2mn - n(n+1)/2 + mn^2/\widetilde{W}$ words between slow and fast memory, in which $\widetilde{W} = W - n(n+1)/2$, and performs $2mn/\widetilde{W}$ transfers between slow and fast memory. Thus the run time for sequential TSQR is given by

$$\text{Time}_{\text{Seq. TSQR}}(m,n,W) = \left(2mn^2 - \frac{2n^3}{3}\right)\gamma + \left(2mn - \frac{n(n+1)}{2} + \frac{mn^2}{\widetilde{W}}\right)\beta + \left(\frac{2mn}{\widetilde{W}}\right)\alpha, \tag{2.3}$$

where γ is the floating-point throughput, β the inverse bandwidth between slow and fast memory (we assume read and write bandwidth are the same; modeling asymmetric bandwidth is a simple exercise), and α the latency between slow and fast memory. Since $\widetilde{W} \gtrsim 2W/3$, the number of messages $2mn/\widetilde{W} \lesssim 3mn/W$.

2.3. More general TSQR algorithms. All TSQR algorithms use at least two of the three primitive operations described in section 2.4 to combine the blocks, until a single R factor remains. This combination process forms a tree, with the row blocks A_i as the leaves and the final R factor as the root. The algorithms differ only in the tree topology and the use of parallel processors. For example, parallel TSQR above uses a binary tree, and sequential TSQR uses a flat tree. More general tree shapes can be used to minimize communication and tune TSQR performance for different computer architectures. For example, see Demmel et al. [18] for a tree shape that minimizes both communication between processors and the volume of memory traffic between main memory and each processor's cache, in a shared-memory implementation. Demmel et al. [16, section 4.3] show a different tree shape that optimizes for the case of multiple processors sharing a single connection to memory.

Parallel TSQR looks like a reduction operation over intermediate R factors and is indeed a reduction if we only desire the final R factor of A. This is because the QR factorization is unique (in exact arithmetic), up to the choice of the signs of the diagonal entries of R. The latter can always be chosen nonnegative in a numerically stable way; see, e.g., Demmel et al. [17]. However, if we want the Q factor as well, TSQR is not a reduction, since applying the Q factor or computing its explicit representation requires using the same tree as when factoring the matrix A. Nevertheless, we can use the same performance tuning techniques as with standard parallel reductions in

order to discover efficient tree shapes automatically. See, e.g., Nishtala, Almási, and Caşcaval [39].

2.4. Primitive operations. All TSQR variants perform floating-point computations only in the following three primitive local operations:

1. Compute the QR factorization of the $m_i \times n$ row block A_i (with $m_i \geq n$).
2. Compute the QR factorization of $[R_i; A_j]$, where R_i is an $n \times n$ upper triangular matrix and A_j is an $m_j \times n$ row block (with $m_j \geq n$).
3. Compute the QR factorization of $[R_1; R_2; \ldots; R_k]$, where $k \geq 2$ and $R_1, \ldots, R_k$ are all $n \times n$ upper triangular matrices.

Each of the three primitive factorizations is computed "in place," that is, by overwriting the input. This can be done without introducing any new nonzero entries. Only n scaling factors (corresponding to the `TAU` output of LAPACK's `GEQRF`) must be stored for each QR factorization. These operations can exploit faster QR algorithms, such as Schreiber and Van Loan's YTY^T representation [45] or Elmroth and Gustavson's recursive QR [21]. See Demmel et al. [16] for details.

2.5. TSQR versus alternative algorithms. In this section, we compare performance models of parallel and sequential QR to those of alternative QR factorization algorithms. These include (blocked) Householder QR, classical and modified Gram–Schmidt (CGS, respectively, MGS), and CholeskyQR (which computes the Cholesky factorization $R^T R = A^T A$ and forms $Q = AR^{-1}$). In summary, TSQR is as numerically accurate as Householder QR but communicates asymptotically less than Householder QR. TSQR also communicates only a small constant factor more than the much less accurate CholeskyQR algorithm. We only outline our approach here and give details in Demmel et al. [16, Section 9].

The version of sequential Householder QR we model here was optimized to minimize data movement when fast memory is DRAM and slow memory is disk ("out-of-DRAM"). This routine, `PFDGEQRF` [14], was designed to exploit ScaLAPACK's parallelism for the panel factorization and for updates of trailing matrix panels. We assume here that it is running sequentially, since we are interested only in modeling memory traffic. `PFDGEQRF` is left looking, as usual with out-of-DRAM algorithms (left-looking schemes write less than right-looking schemes, and disk write bandwidth is often less than read bandwidth). It keeps two panels in memory: a left panel of fixed width b, and the current panel being factored, whose width c can expand to fill the available memory. Details may be found in [14] and Demmel et al. [16, Appendix F]. In the latter, we choose b and c to minimize disk traffic. Here, we summarize the performance model in Table 2.1 for optimal choices of b and c.

The implementation of `PFDGEQRF` requires $mb < W$: the left panel and at least one column of the trailing matrix must fit in fast memory. LAPACK's right-looking sequential blocked QR factorization routine `DGEQRF` does not have this requirement: the current panel need not fit in fast memory. Nevertheless, `DGEQRF` also does not attain the communication lower bound, for almost all reasonable matrix dimensions and fast memory sizes. See Appendix A for a detailed derivation.

Examining Table 2.1, we see that all parallel algorithms have the same highest order term in their flop counts, $2mn^2/P$, and also use the same bandwidth, $\frac{n^2}{2} \log P$, but that parallel TSQR sends a factor of $2n$ fewer messages than the only stable alternative (`PDGEQRF`), and is about as fast as the fastest unstable method (CholeskyQR). In other words, only parallel TSQR is simultaneously fastest and stable. Examining Table 2.1, we see a similar story, with sequential TSQR sending a factor of about $\frac{mn}{4W}$

TABLE 2.1

Performance models of various parallel and sequential QR algorithms for matrices with $m \gg n$. `PFDGEQRF` is our model of Householder QR; W is the fast memory capacity, and $\widetilde{W} = W - n(n+1)/2$. Lower-order terms omitted.

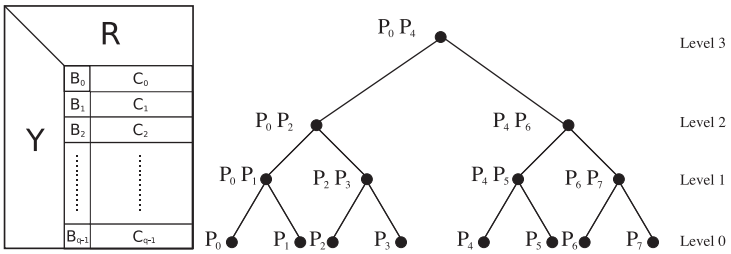
Parallel algorithm	# flops	# messages	# words
TSQR	$\frac{2mn^2}{P} + \frac{2n^3}{3}\log(P)$	$\log(P)$	$\frac{n^2}{2}\log(P)$
`PDGEQRF`	$\frac{2mn^2}{P} - \frac{2n^3}{3P}$	$2n\log(P)$	$\frac{n^2}{2}\log(P)$
MGS	$\frac{2mn^2}{P}$	$2n\log(P)$	$\frac{n^2}{2}\log(P)$
CGS	$\frac{2mn^2}{P}$	$2n\log(P)$	$\frac{n^2}{2}\log(P)$
CholeskyQR	$\frac{2mn^2}{P} + \frac{n^3}{3}$	$\log(P)$	$\frac{n^2}{2}\log(P)$
Sequential algorithm	**# flops**	**# messages**	**# words**
TSQR	$2mn^2 - \frac{2n^3}{3}$	$\frac{2mn}{W}$	$2mn - \frac{n(n+1)}{2} + \frac{mn^2}{W}$
`PFDGEQRF`	$2mn^2 - \frac{2n^3}{3}$	$\frac{2mn}{W} + \frac{mn^2}{2W}$	$\frac{m^2n^2}{2W} - \frac{mn^3}{6W} + \frac{3mn}{2} - \frac{3n^2}{4}$
MGS	$2mn^2$	$\frac{2mn^2}{\widetilde{W}}$	$\frac{3mn}{2} + \frac{m^2n^2}{2\widetilde{W}}$
CholeskyQR	$2mn^2 + \frac{n^3}{3}$	$\frac{6mn}{W}$	$3mn$

fewer words and $n/4$ fewer messages than the only stable alternative, `PFDGEQRF`. Note that mn/W gives how many times larger the matrix A is than the fast memory capacity. Since we assume $W \geq n^2$, sequential TSQR sends fewer words than sequential CholeskyQR.

3. Communication-avoiding QR—CAQR. We present the CAQR algorithm for computing the QR factorization of an m-by-n matrix A, with $m \geq n$. Stated most simply, CAQR simply implements the right-looking QR factorization using TSQR as the panel factorization. The update of the trailing matrix after each panel factorization is triggered by the shape of the reduction tree used by TSQR.

We discuss parallel CAQR and compare its performance to ScaLAPACK. We also show, given m, n, and the number of processors P, how to choose a distribution of the input matrix to minimize running times of both algorithms; our proof of CAQR's optimality depends on these choices. Then we do the same for sequential CAQR and an out-of-DRAM algorithm from ScaLAPACK, whose floating point operations are counted sequentially. We also discuss other sequential QR algorithms, including showing that recursive QR routines of Elmroth and Gustavson [21] also minimize bandwidth, though possibly not latency.

3.1. Parallel CAQR. We describe parallel CAQR algorithm and a few details most relevant to the complexity but refer the reader to [16, section 13] for details. The matrix A is stored on a two-dimensional grid of processors $P = P_r \times P_c$ in a two-dimensional block-cyclic layout, with blocks of dimension $b \times b$. We assume that all the blocks have the same size; we can always pad the input matrix with zero rows and columns to ensure this is possible. For a detailed description of the two-dimensional block cyclic layout, see [8]. CAQR is based on TSQR in order to minimize communication. At each step of the factorization, TSQR is used to factor a panel of columns, and the resulting Householder vectors are applied to the rest of the matrix. The block column QR factorization as performed in `PDGEQRF` is the latency bottleneck of the current ScaLAPACK QR algorithm. Replacing this block column factorization with TSQR based on a binary tree, and adapting the rest of the algorithm to work with TSQR's representation of the panel Q factors, removes the bottleneck.



(a) Step j of CAQR (b) TSQR reduction tree

FIG. 3.1. *Step j of CAQR factorization* (a), *and an example of a binary TSQR reduction tree with eight processors* (b). *First, the current panel of width b, $B = [B_0; B_1; \cdots ; B_{q-1}]$ is factorized using TSQR. Here, q is the number of blocks in the current panel. Second, the trailing matrix, $C = [C_0; C_1; \cdots ; C_{q-1}]$, is updated.*

CAQR is defined iteratively. We assume that the first $j-1$ iterations of the CAQR algorithm have been performed. That is, $j-1$ panels of width b have been factored and the trailing matrix has been updated. The active matrix at step j (that is, the part of the matrix which needs to be worked on) is of dimension $(m-(j-1)b) \times (n-(j-1)b) = m_j \times n_j$.

Figure 3.1(a) shows the execution of the QR factorization. For the sake of simplicity, we suppose that processors $0, \ldots, P_r - 1$ lie in the column of processes that hold the current panel j and that P_r is a power of 2. The $m_j \times b$ matrix $B = [B_0; B_1; \ldots ; B_{q-1}]$ represents the current panel j. The $m_j \times (n_j - b)$ matrix $C = [C_0; C_1; \ldots ; C_{q-1}]$ is the trailing matrix that needs to be updated after the TSQR factorization of B. For each processor i, the first b rows of its first block row of B and C are B_i and C_i, respectively.

We first introduce some notation to help us refer to different parts of a binary TSQR reduction tree. TSQR takes place in $(\log_2 P_r + 1)$ steps, starting from the bottom level $k = 0$ of a binary tree. Each node of the binary tree is associated with a set of processors. We use the following notation:

- $level(i, k) = \left\lfloor \frac{i}{2^k} \right\rfloor$ denotes the node at level k of the reduction tree which is assigned to a set of processors that includes processor i.
- $first\_proc(i, k) = 2^k level(i, k)$ is the index of the "first" processor associated with the node $level(i, k)$ at stage k of the reduction tree. In a reduction, it receives the messages from its neighbors and performs the local computation.
- $target\_first\_proc(i, k) = first\_proc(i, k) + 2^{k-1}$ is the index of the processor with which $first\_proc(i, k)$ exchanges data in a reduction at level k.

A binary TSQR reduction tree for eight processors is shown in Figure 3.1(b). For example, the processors P_4 and P_6 are affected to the right node at level $k = 2$. With the above notation, the processors in the range $i = 4, \ldots, 7$ can compute easily the two processors affected to this node, that is, $first\_proc(i, 2) = 4$ and $target\_first\_proc(i, 2) = 6$.

Algorithm 2 outlines the right-looking parallel QR decomposition. At iteration j, first, the block column j is factored using TSQR. After the block column factorization is complete, the trailing matrix is updated as follows. The update corresponding to the QR factorization at the leaves of the TSQR tree is performed locally on every processor. The updates corresponding to the upper levels of the TSQR tree are performed between groups of neighboring trailing matrix processors. Note that only one of the trailing matrix processors in each neighbor group continues to be involved

Algorithm 2. Right-looking parallel CAQR factorization.

1: **for** $j = 1$ to $n/b =$ **do**
2: The column of processors that holds panel j computes a TSQR factorization of this panel. The Householder vectors are stored in a tree-like structure.
3: Each processor p that belongs to the column of processes holding panel j broadcasts along its row of processors the $m_j/P_r \times b$ matrix that holds the two sets of Householder vectors. It also broadcasts two arrays of size b each, containing the Householder multipliers τ_p.
4: Each processor in the same process row as processor p, $0 \leq p < P_r$, forms T_{p0} and updates its local trailing matrix C using T_{p0} and Y_{p0}. (This computation involves all processors.)
5: **for** $k = 1$ to $\log P_r$, the processors that lie in the same row as processor p, where $0 \leq p < P_r$ equals $first\_proc(p,k)$ or $target\_first\_proc(p,k)$, respectively. = **do**
6: Processors in the same process row as $target\_first\_proc(p,k)$ form $T_{level(p,k),k}$ locally. They also compute pieces of $W = Y^T_{level(p,k),k} C_{target\_first\_proc(p,k)}$, leaving the results distributed. This computation is overlapped with the communication in Line 7.
7: Each processor in the same process row as $first\_proc(p,k)$ sends to the processor in the same column and belonging to the row of processors of $target\_first\_proc(p,k)$ the local pieces of $C_{first\_proc(p,k)}$.
8: Processors in the same process row as $target\_first\_proc(p,k)$ compute pieces of

$$W = T^T_{level(p,k),k} \left(C_{first\_proc(p,k)} + W \right).$$

9: Each processor in the same process row as $target\_first\_proc(p,k)$ sends to the processor in the same column and belonging to the process row of $first\_proc(p,k)$ the local pieces of W.
10: Processors in the same process row as $first\_proc(p,k)$ and $target\_first\_proc(p,k)$ each complete the rank-b updates $C_{first\_proc(p,k)} := C_{first\_proc(p,k)} - W$ and $C_{target\_first\_proc(p,k)} := C_{target\_first\_proc(p,k)} - Y_{level(p,k),k} \cdot W$ locally. The latter computation is overlapped with the communication in Line 9.
11: **end for**
12: **end for**

in successive trailing matrix updates. This allows overlap of computation and communication, as the uninvolved processors can finish their computations in parallel with successive reduction stages. Table 3.1 expresses the performance model over a rectangular $P_r \times P_c$ grid of processors. A detailed derivation of the model is given in [16]. According to the table, the number of arithmetic operations and words transferred is roughly the same between parallel CAQR and ScaLAPACK's parallel QR factorization, but the number of messages is a factor b times lower for CAQR.

When choosing b, P_r, and P_c to minimize the run time, they must satisfy the following conditions: $1 \leq P_r, P_c \leq P$, $P_r \cdot P_c = P$, $1 \leq b \leq \frac{m}{P_r}$, and $1 \leq b \leq \frac{n}{P_c}$. For simplicity we will assume that P_r evenly divides m and that P_c evenly divides n. These values are chosen simultaneously to minimize the approximate number of words sent, $n^2/P_c + mn/P_r$, and the approximate number of messages, $5n/b$, where for simplicity we temporarily ignore logarithmic factors and lower order terms in Table 3.1. This suggests the ansatz

$$P_r = K \cdot \sqrt{\frac{mP}{n}} \ , \ P_c = \frac{1}{K} \cdot \sqrt{\frac{nP}{m}}, \ \text{ and } b = B \cdot \sqrt{\frac{mn}{P}}, \tag{3.1}$$

for general values of K and $B \leq \min\{K, 1/K\}$, since we can thereby explore all

TABLE 3.1

Performance models of parallel CAQR and ScaLAPACK's `PDGEQRF` *routine when factoring an $m \times n$ matrix, distributed in a two-dimensional block cyclic layout on a $P_r \times P_c$ grid of processors with square $b \times b$ blocks. All terms are counted along the critical path. In this table, "flops" only includes floating-point additions and multiplications, not floating-point divisions. Some lower order terms are omitted. We generally assume $m \geq n$.*

	Parallel CAQR
# messages	$\frac{3n}{b} \log P_r + \frac{2n}{b} \log P_c$
# words	$\left(\frac{n^2}{P_c} + \frac{bn}{2}\right) \log P_r + \left(\frac{mn - n^2/2}{P_r} + 2n\right) \log P_c$
# flops	$\frac{2n^2(3m-n)}{3P} + \frac{bn^2}{2P_c} + \frac{3bn(2m-n)}{2P_r} + \left(\frac{4b^2 n}{3} + \frac{n^2(3b+5)}{2P_c}\right) \log P_r - b^2 n$
	ScaLAPACK's `PDGEQRF`
# messages	$3n \log P_r + \frac{2n}{b} \log P_c$
# words	$\left(\frac{n^2}{P_c} + bn\right) \log P_r + \left(\frac{mn - n^2/2}{P_r} + \frac{bn}{2}\right) \log P_c$
# flops	$\frac{2n^2(3m-n)}{3P} + \frac{bn^2}{2P_c} + \frac{3bn(2m-n)}{2P_r} - \frac{b^2 n}{3P_r}$

possible values of b, P_r, and P_c. Using the substitutions in (3.1), the flop count (neglecting lower order terms, including the division counts) becomes

$$\frac{mn^2}{P}\left(2 - B^2 + \frac{3B}{K} + \frac{BK}{2}\right) - \frac{n^3}{P}\left(\frac{2}{3} + \frac{3B}{2K}\right) + \frac{mn^2}{P} \log\left(K \cdot \sqrt{\frac{mP}{n}}\right)\left(\frac{4B^2}{3} + \frac{3BK}{2}\right). \tag{3.2}$$

We wish to choose B and K so as to minimize the flop count. We know at least that we need to eliminate the dominant $mn^2 \log(\dots)$ term, so that parallel CAQR has the same asymptotic flop count as ScaLAPACK's `PDGEQRF`. This is because we know that CAQR performs at least as many floating-point operations (asymptotically) as `PDGEQRF`, so matching the highest order terms will help minimize CAQR's flop count. To make the high order terms of (3.2) match the $2mn^2/P - 2n^3/(3P)$ flop count of ScaLAPACK's parallel QR routine, while minimizing communication as well, we can pick $K = 1$ and $B = o(\log^{-1}(\sqrt{mP/n}))$; for simplicity we will use $B = \log^{-2}(\sqrt{mP/n})$.

Using the substitutions in (3.1) and the above choices of B and K, the results are shown in Table 1.2, which also shows the results for ScaLAPACK, whose analogous analysis appears in [16, section 15], and the communication lower bounds, which are discussed in section 4. In summary, if we choose b, P_r, and P_c independently and optimally for both algorithms, the two algorithms match in the number of flops and words transferred, but CAQR sends a factor of $\Theta(\sqrt{mn/P})$ messages fewer than ScaLAPACK QR. This factor is the local memory requirement on each processor, up to a small constant.

3.2. Sequential CAQR. As stated above, sequential CAQR is just a right-looking QR factorization with TSQR used for the panel factorization. (In fact, left-looking QR with TSQR has the same costs, as we show in [16, Appendix C]. We describe only the right-looking algorithm here for simplicity.) We also assume that the m by n matrix A is stored in a $P_r \times P_c$ two-dimensional blocked layout, with individual $\frac{m}{P_r}$ by $\frac{n}{P_c}$ blocks stored contiguously in memory with $m \geq n$ and $\frac{m}{P_r} \geq \frac{n}{P_c}$.

For TSQR to work as analyzed we need to choose P_r and P_c large enough for one such $\frac{m}{P_r}$ by $\frac{n}{P_c}$ block to fit in fast memory, plus a bit more. For CAQR we will need to choose P_r and P_c a bit larger, so that a bit more than three such blocks fit in fast memory; this is in order to perform an update on two such blocks in the trailing matrix given Householder vectors from TSQR occupying $\frac{mn}{P_r P_c} + \frac{n^2}{2P_c^2}$ words, or at most $\frac{4mn}{P}$ altogether. In other words, we need $4mn/P \leq W$ or $P \geq 4mn/W$.

Leaving details to [16, Appendix C], we summarize the complexity analysis by

$$T_{\text{seq. CAQR}}(m,n,P_c,P_r) \leq \left(\frac{3}{2}P(P_c-1)\right)\alpha + \left(\frac{3}{2}mn\left(P_c+\frac{4}{3}\right)-\frac{1}{2}n^2P_c\right)\beta + \left(2n^2m-\frac{2}{3}n^3\right)\gamma, \tag{3.3}$$

where we have ignored lower order terms and used P_r as an upper bound on the number of blocks in each panel since this increases the run time only slightly and is simpler to evaluate than for the true number of blocks $P_r - \lfloor (J-1)\frac{nP_r}{mP_c} \rfloor$.

Now we choose P, P_r, and P_c to minimize the run time. From the above formula for $T_{\text{seq. CAQR}}(m,n,P_c,P_r)$, we see that the run time is an increasing function of P_r and P_c, so that we would like to choose them as small as possible, within the limits imposed by the fast memory size $P \geq \frac{4mn}{W}$. So we choose $P = \frac{4mn}{W}$ (assuming here and elsewhere that the denominator evenly divides the numerator). But we still need to choose P_r and P_c subject to $P_r \cdot P_c = P$.

Examining $T_{\text{seq. CAQR}}(m,n,P_c,P_r)$ again, we see that if P is fixed, the run time is also an increasing function of P_c, which we therefore want to minimize. But we are assuming $\frac{m}{P_r} \geq \frac{n}{P_c}$, or $P_c \geq \frac{nP_r}{m}$. The optimal choice is therefore $P_c = \frac{nP_r}{m}$ or $P_c = \sqrt{\frac{nP}{m}}$, which also means $\frac{m}{P_r} = \frac{n}{P_c}$, i.e., the blocks in the algorithm are square. This choice of $P_r = \frac{2m}{\sqrt{W}}$ and $P_c = \frac{2n}{\sqrt{W}}$ therefore minimizes the runtime, yielding

$$T_{\text{Seq. CAQR}}(m,n,W) \leq \left(12\frac{mn^2}{W^{3/2}}\right)\alpha + \left(3\frac{mn^2}{\sqrt{W}}+\right)\beta + \left(2mn^2-\frac{2}{3}n^3\right)\gamma. \tag{3.4}$$

We note that the bandwidth term is proportional to $\frac{mn^2}{\sqrt{W}}$, and the latency term is W times smaller, both of which match (to within constant factors) the lower bounds on bandwidth and latency to be described in section 4.

The results of this analysis are shown in Table 1.3. That table also shows the results for an out-of-DRAM algorithm `PFDGEQRF` from ScaLAPACK, whose internal block sizes b and c have been chosen to minimize disk traffic, and where we count the floating point operations sequentially (see [16, Appendix F]). We analyze LAPACK's `DGEQRF` routine in detail in Appendix A, where we show that this routine also is not bandwidth optimal for most reasonable matrix dimensions and fast memory sizes.

3.3. Other bandwidth minimizing sequential QR algorithms. In this section we discuss two variants of Elmroth's and Gustavson's recursive sequential QR factorization: `RGEQR3` and `RGEQRF` [21]. We describe special cases in which these algorithms also minimize bandwidth, although they do not minimize latency.

The fully recursive routine `RGEQR3` is analogous to Toledo's fully recursive LU routine [47]. Both routines factor the left half of the matrix (recursively), use the resulting factorization of the left half to update the right half, and then factor the right half (recursively again). The base case consists of a single column. When applied to an $m \times n$ matrix, `RGEQR3` returns the Q factor in the form $I - YTY^T$, where Y is

the $m \times n$ lower triangular matrix of Householder vectors and T is an $n \times n$ upper triangular matrix. A simple recurrence for the number of memory references of either `RGEQR3` or Toledo's algorithm is

$$
\begin{aligned}
B(m,n) &= \begin{cases} B\left(m, \frac{n}{2}\right) + B\left(m - \frac{n}{2}, \frac{n}{2}\right) + O\left(\frac{mn^2}{\sqrt{W}}\right) & \text{if } mn > W \text{ and } n > 1, \\ mn & \text{if } mn \le W, \\ m & \text{if } m > W \text{ and } n = 1 \end{cases} \\
&\le \begin{cases} 2B\left(m, \frac{n}{2}\right) + O\left(\frac{mn^2}{\sqrt{W}}\right) & \text{if } mn > W \text{ and } n > 1, \\ mn & \text{if } mn \le W, \\ m & \text{if } m > W \text{ and } n = 1 \end{cases} \qquad (3.5) \\
&= O\left(mn^2/\sqrt{W}\right) + mn.
\end{aligned}
$$

So RGEQR3 attains our bandwidth lower bound. (The mn term must be included to account for the case when $n < \sqrt{W}$, since each of the mn matrix entries must be accessed at least once.) However, `RGEQR3` performs a factor greater than one times as many floating point operations as sequential Householder QR.

Now we consider `RGEQRF`. This is a right-looking algorithm that differs from LAPACK's `DGEQRF` only in how it performs the panel factorization: `RGEQRF` uses `RGEQR3`, and `DGEQRF` uses `DGEQR2`, respectively. Let b be the panel width in either algorithm. It is easy to see that a reasonable estimate of the number of memory references just for the updates by all the panels is the number of panels n/b times the minimum number of memory references for the average size update $\Theta(\max(mn, mnb/\sqrt{W}))$, or $\Theta(\max(mn^2/b, mn^2/\sqrt{W}))$. Thus we need to pick b at least about as large as $\sqrt{W}$ to attain the desired lower bound $O(mn^2/\sqrt{W})$.

Concentrating now on `RGEQRF`, we get from inequality (3.5) that the n/b panel factorizations using `RGEQR3` cost at most an additional number of memory references,

$$
\#\text{ memory references} = O\left(\frac{n}{b} \cdot \left[\frac{mb^2}{\sqrt{W}} + mb\right]\right) = O\left(\frac{mnb}{\sqrt{W}} + mn\right).
$$

This is $O(mn)$ if we pick $b = \sqrt{W}$. Thus the total number of memory references for `RGEQRF` with $b = \sqrt{W}$ is $O(mn^2/\sqrt{W} + mn)$, which attains the desired lower bound.

`RGEQR3` does not always minimize latency. For example, consider applying `RGEQR3` to a single panel with $n = \sqrt{W}$ columns and $m > W$ rows, stored in a block-column layout with $\sqrt{W}$-by-$\sqrt{W}$ blocks stored columnwise, as above. Then a recurrence for the number of messages `RGEQR3` requires is

$$
\begin{aligned}
L(m,n) &= \begin{cases} L\left(m, \frac{n}{2}\right) + L\left(m - \frac{n}{2}, \frac{n}{2}\right) + O\left(\frac{m}{\sqrt{W}}\right) & \text{if } n > 1, \\ O\left(\frac{m}{\sqrt{W}}\right) & \text{if } n = 1 \end{cases} \\
&= O\left(\frac{mn}{\sqrt{W}}\right) = O(m) \text{ when } n = \sqrt{W},
\end{aligned}
$$

which is larger than the minimum $O(mn/W) = O(m/\sqrt{W})$ attained by sequential TSQR when $n = \sqrt{W}$.

In contrast to `DGEQRF`, `RGEQRF`, and `RGEQR3`, CAQR minimizes flops, bandwidth, and latency for all values of W.

4. Arithmetic and communication lower bounds for QR factorization. In this section we give a lower bound on the number of arithmetic operations performed by any "non-Strassen-like" implementation of QR decomposition. We then combine this with the communication lower bounds for QR decomposition in [7, section 4], which is proportional to the number of arithmetic operations, to establish a communication lower bound for QR decomposition, subject to certain technical assumptions discussed below, that is met by our CAQR algorithm.

Now we discuss the technical assumptions under which our lower bounds hold and how to interpret the results of this section. The original technical report on which this paper is based [17] proved a communication lower bound for QR decomposition but made an implicit assumption that not all QR implementations necessarily satisfy. The results in [7, section 4] examine QR decomposition more closely and prove the desired communication bound given two different kinds of technical assumptions. Not all QR algorithms, even the ones we propose here, necessarily satisfy these assumptions. We believe that these assumptions are not necessary for the communication lower bound to hold, in part because it would be very surprising if QR decomposition turned out to be "easier" than matrix multiplication, for which the lower bounds do hold. Also, the arithmetic lower bounds proven here (to which the communication bounds are proportional) do not require these assumptions. Therefore, these communication lower bounds establish natural targets for any QR algorithm to attain, which is the main goal of this paper.

Here are the technical assumptions (see [7, section 4] for details). We assume that the algorithm uses blocked Householder (or Givens) transformations; write such a blocked transformation as $I - UTU^T$. Furthermore, the algorithm must apply such a transformation as $A := A - U(TU^TA) = A - UZ$, i.e., $Z = TU^TA$ is computed first. The lower bound only counts the multiplications performed in all the matrix multiplications UZ, which is a significant fraction of the total work. So far, these assumptions apply to any algorithm discussed in this paper. To establish the lower bound, either one of the following two additional technical assumptions is needed: (1) Assuming A is $m \times n$, the total number of entries of all Z matrices produced during the algorithm is $O(mn)$. This must hold, for example, if one uses only one Householder transformation per column to zero it out below the diagonal, but not necessarily for all variations of CAQR. (2) First, the algorithm must make "forward progress" [7, section 4.2.2]; roughly, this means that it is allowed to zero out any entry only once. Second, all T matrices must be 1×1, i.e., no blocking is allowed. This sounds like a drastic limitation, but in fact there are some CAQR variants that do satisfy it. For example, if we perform CAQR without blocking any of the local updates (which does not change the communication costs in either the sequential or parallel model), then the lower bound result of [7, Theorem 4.10] applies, and this variant of CAQR attains that lower bound.

Existing implementations of QR decomposition of an m-by-n matrix with $m \geq n$ all take $\Theta(mn^2)$ flops. Our main contribution is to show that this is necessary for *any* non-Strassen-like algorithm (in a sense made formal below) and to establish that the constant factor is at least $\frac{1}{3}$.

In fact, our approach can be used to get arithmetic lower bounds for any one-sided factorization (including LU and QR), on dense or sparse matrices, but we leave this generalization to future work [6].

From the definition of the QR decomposition, we see that columns i of R and Q only depend mathematically on columns 1 through i of A. Furthermore, $R = Q^TA$ implies that the first $i-1$ entries of column i of R are given by $R(1 : i-1, i) =$

$(Q(:, 1:i-1))^T \cdot A(:, i)$, i.e., the product of a matrix that depends on the first $i-1$ columns of A, with the ith column of A.

The independence of these two sets of parameters (first $i-1$ columns of A, and column i of A) will let us describe the non-Strassen-like algorithms for which we can get a lower bound. Suppose $B(x)$ is an m-by-n linear operator parametrized by the vector x (in our case, the entries of columns 1 to $i-1$ of A) and that we want to multiply $z = B(x) \cdot y$, where each entry of y is an independent parameter (in our case, the entries of column i of A). We assume the directed acyclic graph (DAG) implementing $z = B(x) \cdot y$ has the following properties.

DEFINITION 4.1 (non-Strassen-like multiplication $z = B(x) \cdot y$). *We label the input nodes of the DAG as x_{in} or y_{in} with no overlap allowed. We give the same label to the edges carrying their values to inputs of other nodes. If more computations just on x_{in} nodes are performed to compute other functions just of x, label these nodes and their output edges by x_{in} too (we will not count these operations in our lower bound). We label output nodes as z_{out}. The other nodes performing computations are labeled as follows:*

1. *A node multiplying an x_{in} edge and a y_{in} edge is labeled $mul(x_{in}, y_{in})$ and produces an output edge labeled y_{homo}, which is short for "linear homogeneous function of y with coefficients that are functions of x."*
2. *A node multiplying an x_{in} edge and a y_{homo} edge is labeled $mul(x_{in}, y_{homo})$ and produces an output edge also labeled y_{homo}.*
3. *A node multiplying anything (an $f \in \{x_{in}, y_{in}, y_{homo}\}$) by a constant like 3.1416 is labeled $mul(f, 3.1416)$, and its output edge is labeled x_{in} or y_{homo} as appropriate.*
4. *A node that adds or subtracts its inputs is labeled either $add/sub(x_{in})$ if both its inputs are x_{in} (in which case its output edge is x_{in}) or labeled $add/sub(y_{homo})$ if both its inputs are y_{in} and/or y_{homo} (in which case the output edge is y_{homo}).*

In particular, no nodes that add or subtract one x_{in} edge with either a y_{in} or y_{homo} edge are permitted. That is because these cannot contribute to creating a y_{homo}, which is what each z_{out} must finally be. Similarly, we are not allowed to compute quadratic or higher degree functions (or reciprocals or square roots ...) of y_{in} or y_{homo}, because these are also not y_{homo}.

This also captures the idea of non-Strassen-like multiplication, because column y cannot be combined with other columns to participate in later operations. In other words, no intermediate results can depend on data on which the final result does not depend, namely, on data from other columns.

Our lower bound depends on the following result:

LEMMA 4.2. *Consider performing the multiplication $z = B(x) \cdot y$ using an algorithm satisfying Definition 4.1. Suppose that as a subset of $R^{m \times n}$, the set of all $B(x)$ has dimension $d \le m \cdot n$ or is a union of various manifolds, at least one of which has dimension d. Then at least d multiplications (nodes labeled $mul(\cdot)$ in the above definition) are needed to implement the multiplication $z = B(x) \cdot y$.*

Proof. Consider running the algorithm on all possible inputs (x, y). Use the DAG symbolically to express each of the m z_{out} nodes as the symbolic sum $z(i) = \sum_{j=1}^{n} f_{ij}(x) \cdot y(j)$, and create the point $F(x) = (f_{11}(x), \ldots, f_{mn}(x))$ in $R^{m \times n}$, which is an explicit representation of the $B(x)$ that is multiplied by y. The set $\mathcal{F}$ of all points $F(x)$ has to have dimension d (or some manifold it contains must have dimension d). Only $mul(y_{in}, x_{in})$ and $mul(y_{homo}, x_{in})$ nodes can introduce a dependence of an output on a new function of x; let $\#muls$ be the number of such nodes. Thus $F(x)$

can only depend on $\#muls$ different functions of x. In other words, the set $\mathcal{F}$ is the image of the composition of two functions $F(x) = g(h(x))$, where $h(x)$ maps the set of all x values to R^q, where $q \leq \#muls$ is the number of different functions of x that the DAG can use, and $g(\cdot)$ maps from R^q to $R^{m\times n}$. So $F(x)$ must lie in a set of dimension at most q (since $g(\cdot)$ and $h(\cdot)$ are smooth functions), and thus $\#muls \geq q \geq d$ as desired. ∎

We note that this result would not apply to a Strassen-like algorithm used to multiply $B(x)$ times many vectors y_i, because such algorithms would form linear combinations of different y_i, which is excluded by our model.

This lemma let us prove the following lower bound on the number of multiplications needed to compute R factor of the QR decomposition.

THEOREM 4.3. *Consider computing the QR decomposition of the m-by-n real dense matrix A, with $m \geq n$, where the computation of just the strict upper triangle of R uses an algorithm satisfying Definition* 4.1. *Then the number of multiplications needed to compute the R factor is at least*

$$G(m,n) \equiv \frac{mn(n-1)}{2} - \frac{n(n^2-1)}{6} = \frac{n^2}{2}\left(m - \frac{n}{3}\right) - \frac{mn}{2} + \frac{n}{6}. \tag{4.1}$$

Proof. We observe that $R = Q^T \cdot A$ implies that $R(1:i-1,i)$ is the product $(Q(:,1:i-1))^T \cdot A(:,i)$, and that $Q(:,1:i-1)$ depends only on columns $1:i-1$ of A. Thus Lemma 4.2 applies to the computation of each column of the upper triangle of R. The dimension of the set of m-by-i real orthogonal matrices is the dimension of the Stiefel manifold, namely $mi - i(i+1)/2$. Applying Lemma 4.2, summing from $i = 1$ to $n-1$, yields the claimed lower bound on the number of multiplications. ∎

We emphasize that Theorem 4.3 is quite general, covering CGS, MGS, and CholeskyQR, as well as various conceivable hybrids (QR computed using (block) Householder transformations is considered below). When $m \gg n$, CholeskyQR does $\frac{mn(n+1)}{2} + \frac{n^3}{6} + O(n^2)$ multiplications just to compute R, which agrees with the lower bound in the highest order term $\frac{mn^2}{2}$. When $m = n$, the lower bound underestimates the number of multiplications by about a factor of two (versus CholeskyQR) or three (versus Gram–Schmidt), because we are not including the work to compute Q in our lower bound. This is good enough for our Big-Omega lower bounds.

Now we combine this result with communication lower bounds for orthogonal decompositions in [7].

THEOREM 4.4. *Consider computing the QR decomposition of the m-by-n real dense matrix A with $m \geq n$, where the computation of just the strict upper triangle of R uses an algorithm satisfying Definition* 4.1, *and also that the algorithm satisfies the technical assumptions discussed above* [7, section 4]. *Let $G(m,n)$ be defined as in Theorem* 4.3. *Suppose that the computation is done on a computer with a single large (slow) memory and another smaller (fast) memory of size W words, in which the arguments and result of any arithmetic operation must reside. Then the number of words moved between the fast and slow memory is at least*

$$\#\text{words\_moved} \geq \frac{G(m,n)}{8W^{1/2}} - W = \Omega\left(\frac{mn^2}{W^{1/2}}\right)$$

and the number of messages containing these words that are sent between fast and slow memory is at least

$$\#\text{messages} \geq \frac{G(m,n)}{8W^{3/2}} - 1 = \Omega\left(\frac{mn^2}{W^{3/2}}\right)$$

Proof. This follows either from [7, Lemma 4.1] or from [7, Theorem 4.10], depending on the technical assumptions made. ∎

Analogous to Corollary 3.2 in [7], this result may be extended to the case of a parallel distributed memory computer, where we assume either that (1) the load is balanced, i.e., each of P processors does $G(m,n)/P$ multiplications, or (2) the memory is balanced, i.e., each of the P processors stores mn/P words. Then at least one processor must send or receive at least

$$\begin{aligned}\#\text{words\_moved} &\geq \frac{G(m,n)/P}{8(mn/P)^{1/2}} - \frac{mn}{P} \\ &= \Omega\left(\frac{n^2}{P^{1/2}}\right) \qquad \text{if } m = n\end{aligned}$$

number of words with some other processor(s) and use at least the following number of messages to do so:

$$\begin{aligned}\#\text{messages} &\geq \frac{G(m,n)/P}{8(mn/P)^{3/2}} - 1 = \Omega\left(\sqrt{\frac{Pn}{m}}\right) \\ &= \Omega(P^{1/2}) \qquad \text{if } m = n.\end{aligned}$$

We note that when $m > Pn$, the lower bounds become vacuous. This occurs just when TSQR only needs $\log_2 P$ messages along its critical path, each of size $n^2/2$ (which is consistent with each processor sending and/or receiving $\Theta(1)$ messages of this size).

Section 4.1 in [7] discusses CholeskyQR, CGS and MGS. The steps of CholeskyQR (forming A^TA, and doing its Cholesky factorization) individually satisfy the lower bound $\#muls/(8\sqrt{W}) - W$ in [7, Theorem 2.2]. Since one could conceivably save some memory traffic by computing the Cholesky factorization of A^TA "on the fly" without writing it out to memory, one can use the technique of *imposing reads and writes* of [7, section 5.1.1], in which one writes out A^TA and reads it back in, even if this is not required. This reduces the lower bound to $\#muls/(8\sqrt{W}) - W - 2n^2$, which is asymptotically the same, and lets us use the lower bound $\#mul \geq G(m,n)$ from Theorem 4.3. Since implementations of A^TA and Cholesky exist that attain these communication lower bounds, they can be combined to do CholeskyQR optimally.

Section 4.1 in [7] also discusses CGS and MGS and derives the lower bound $\#\text{words\_moved} \geq \#muls/(8\sqrt{W}) - W$ just for the computation of R, again letting us use $\#muls \geq G(m,n)$. Existing implementations of CGS and MGS do not attain these lower bounds.

These results justify the communication lower bounds presented in Tables 1.2 and 1.3, which are expressed using $\Theta(\cdot)$ notation for simplicity. Table 1.2 assumes the matrix is sufficiently rectangular (n sufficiently larger than m/p) to not use TSQR, so that the lower bounds above are not vacuous. The communication lower bounds for parallel TSQR in Table 1.1 follow from the need to do reductions on P locally computed n-by-n R factors, and the communication lower bounds for sequential TSQR in Table 1.1 follow from the need to read the entire matrix at least once from slow memory.

Finally, we briefly mention another, simpler reduction-based approach to lower bounds, which is most easily applied to LU decomposition. The simple identity

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix} \tag{4.2}$$

shows that any routine that can do LU factorization can be used to multiply square matrices (of 1/3 the size). Thus, any lower bound for communication (or arithmetic) for matrix-multiplication becomes a lower bound (modulo the factor 1/3 to some power) for LU decomposition. Thus, the lower bounds for communication by Hong and Kung [31] and by Irony, Toledo, and Tiskin [32] apply to LU decomposition as well. (The communication lower bounds for LU in [7] take a different but related approach.)

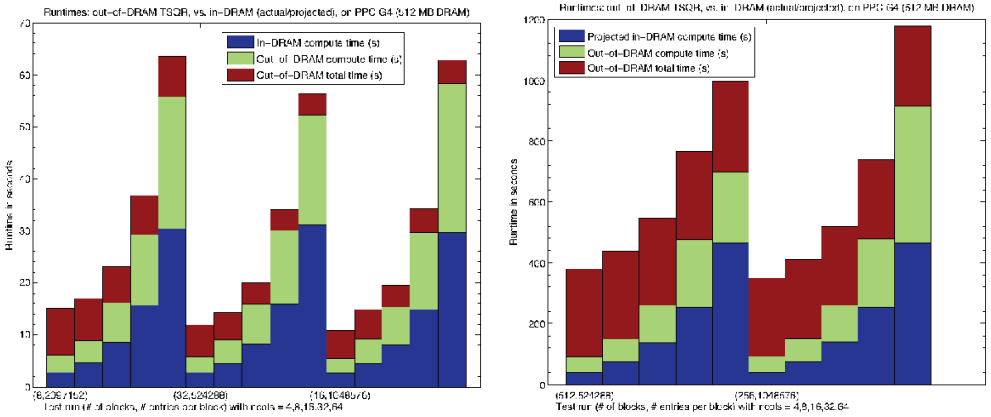
5. Experimental results. In this section we present the performance of sequential and parallel TSQR on several computational systems. We also use the performance model of CAQR in Table 3.1 to predict its performance and compare it to `PDGEQRF`. Detailed performance evaluation on two different parallel machines, an existing IBM POWER5 and a grid formed by 128 processors linked together by the Internet, can be found in the technical report [16].

5.1. Sequential TSQR tests. In Demmel et al. [18], we describe an implementation of the sequential TSQR algorithm of section 2.2 that minimizes memory traffic between cache and DRAM. TSQR occurs in that work as part of a communication-avoiding sparse iterative solver, where it shows significant speedups over competing algorithms. The implementation combines both sequential and parallel optimizations in complicated ways, so we do not discuss its performance further here.

We also developed an out-of-DRAM version of the sequential TSQR algorithm of section 2.2. It uses standard POSIX blocking file operations, with no attempt to overlap communication and computation. Exploiting overlap could at best double the performance. For implementation details, see Demmel et al. [16]. The point of this implementation was not maximum performance but to use TSQR to solve problems too large to fit in DRAM. We show that for such matrices, TSQR is a much better approach than relying on LAPACK's implicit use of virtual memory.

We benchmarked our out-of-DRAM sequential TSQR code on a single-CPU laptop. The 1.5 GHz PowerPC G4 CPU has 512 KB of L2 cache, 512 MB of DRAM on a 167 MHz bus, and one Fujitsu MHT2080AH 80 HB hard drive spinning at 5400 RPM. In our experiments, we first used both out-of-DRAM TSQR and standard LAPACK QR (`DGEQRF`) to factor a collection of matrices that use only slightly more than half of the total DRAM for the factorization. This was so that we could collect comparison timings. Then, we ran only out-of-DRAM TSQR on matrices too large to fit in DRAM or swap space, so that an out-of-DRAM algorithm is necessary to solve the problem at all. For the latter timings, we used a power law model to extrapolate the standard LAPACK QR timings up to the larger problem sizes, in order to estimate the run time if memory were unbounded. LAPACK's QR factorization swaps so much for out-of-DRAM problem sizes that its actual run times are many times larger than these extrapolated unbounded-memory run time estimates. In some cases we had to abandon the computation because it rendered the computer unusable.

Figure 5.1(a) shows the measured in-DRAM results on the laptop platform, and Figure 5.1(b) shows the (measured TSQR, extrapolated LAPACK) out-of-DRAM results on the same platform. In these figures, the amount of memory, and so the total number of matrix entries, is constant for all the experiments: $m \cdot n = 2^{24}$. This fixes the total volume of communication for all experiments. The number of blocks P used, and so the number of matrix entries per block mn/P, is the same for each group of five bars and is shown in a label under the horizontal axis. Within each group of 5 bars, we varied the number of matrix columns to be 4, 8, 16, 32, and 64. Note that we have not tried to overlap I/O and computation in this implementation. The trends in Figure 5.1(a) suggest that the extrapolation is reasonable: TSQR takes



(a) Measured data (b) Extrapolated runtime

FIG. 5.1. *Run times (in seconds) of out-of-DRAM TSQR compared against* (a) *measured data and* (b) *extrapolated (power-law) run time of standard QR (LAPACK's* `DGEQRF`*) on a single-CPU laptop. For the measured data, we limit memory usage to* 256 *MB (half of total system memory), so that we can collect* `DGEQRF` *performance data. The graphs show different choices of block dimensions and number of blocks* P*. The top of the blue bar is* (a) *the benchmarked total runtime for* `DGEQRF` *and* (b) *the extrapolated total runtime for* `DGEQRF`*, the top of the green bar is the benchmarked compute time for TSQR, and the top of the brown bar is the benchmarked total time for TSQR. Thus the height of the brown bar alone is the I/O time. Note that LAPACK starts and ends in DRAM (if it could fit in DRAM), and TSQR starts and ends on disk.*

about twice as much time for computation as does standard LAPACK QR, and the fraction of time spent in I/O is reasonable and decreases with problem size.

TSQR assumes that the matrix starts and ends on disk, whereas LAPACK starts and ends in DRAM. Thus, to compare the two, one could also estimate LAPACK performance with infinite DRAM but where the data starts and ends on disk. The height of the reddish-brown bars in Figures 5.1(a) and 5.1(b) is the I/O time for TSQR, which can be used to estimate the LAPACK I/O time. This is reasonable since the volume of communication in the two cases is the same, and the fact that the reddish-brown bars are of similar height for different values of P shows that the communication is bandwidth dominated. Add this to the blue bar (the LAPACK compute time) to estimate the run time when the LAPACK QR routine must load the matrix from disk and store the results back to disk.

Our results show that using an out-of-DRAM version of sequential TSQR makes it possible to factor matrices too large to fit in DRAM. If we attempt to use LAPACK and rely on the virtual memory system to factor these matrices, either the computer hangs or it takes orders of magnitude more time to complete. Furthermore, our mostly unoptimized implementation of sequential TSQR on matrices too large for DRAM requires only two times longer than LAPACK on a hypothetical computer with infinite DRAM.

5.2. Tests of parallel TSQR on a binary tree. We also present results for a parallel messsage-passing interface (MPI) implementation of TSQR on a binary tree. Rather than LAPACK's `DGEQRF`, the code uses a custom local QR factorization, `DGEQR3`, based on the recursive approach of Elmroth and Gustavson [21]. Tests show that `DGEQR3` consistently outperforms LAPACK's `DGEQRF` by a large margin for matrix dimensions of interest.

TABLE 5.1

Run time in seconds of various parallel QR factorizations on the Beowulf machine. The total number of rows $m = 100{,}000$ and the ratio $\lceil n/\sqrt{P} \rceil = 50$ (with P being the number of processors) were kept constant as P varied from 1 to 64. This illustrates weak scaling with respect to the square of the number of columns n in the matrix, which is of interest because the number of floating-point operations in sequential QR is $\mathcal{O}(mn^2)$. If an algorithm scales perfectly, then all the run times in that algorithm's column should be constant. Both Q and R factors were computed explicitly; in particular, for those codes which form an implicit representation of Q, the conversion to an explicit representation was included in the run time measurement.

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK (PDGEQRF)
1	1.02	4.14	3.73	7.17	9.68	12.63
2	0.99	4.00	6.41	12.56	15.71	19.88
4	0.92	3.35	6.62	12.78	16.07	19.59
8	0.92	2.86	6.87	12.89	11.41	17.85
16	1.00	2.56	7.48	13.02	9.75	17.29
32	1.32	2.82	8.37	13.84	8.15	16.95
64	1.88	5.96	15.46	13.84	9.46	17.74

We ran parallel TSQR on the following distributed-memory machines:

- Pentium III cluster ("Beowulf"), operated by the University of Colorado, Denver. It has 35 dual-socket 900 MHz Pentium III nodes with Dolphin interconnect. Peak floating-point rate is 900 Mflop/s per processor, network latency is less than 2.7 μs, benchmarked,[3] and network bandwidth is 350 MB/s, benchmarked upper bound.
- IBM BlueGene/L ("Frost"), operated by the National Center for Atmospheric Research. We use one BlueGene/L rack with 1024 700 MHz compute CPUs. Peak floating-point rate is 2.8 Gflop/s per processor, network[4] latency is 1.5 μs, hardware, and network one-way bandwidth is 350 MB/s, hardware.

The experiments compare many different implementations of a parallel QR factorization. TSQR was tested both with the recursive local QR factorization `DGEQR3`, and the standard LAPACK routine `DGEQRF`. Both CGS and MGS (by row) were timed.

Tables 5.1 and 5.2 show the results of two different performance experiments on the Pentium III cluster. In the first of these, the total number of rows $m = 100{,}000$ and the ratio $\lceil n/\sqrt{P} \rceil = 50$ (with P being the number of processors) were kept constant as P varied from 1 to 64. This was meant to illustrate weak scaling with respect to n^2 (the square of the number of columns in the matrix), which is of interest because the number of floating-point operations in sequential QR is $\mathcal{O}(mn^2)$. If an algorithm scales perfectly, then all the run times shown in that algorithm's column should be constant. Both the tall and skinny Q and the square R factors were computed explicitly; in particular, for those codes which form an implicit representation of Q, the conversion to an explicit representation was included in the run time measurement. The results show that TSQR scales better than CGS or MGS (by row) and significantly outperforms ScaLAPACK's QR. Also, using the recursive local QR in TSQR, rather than LAPACK's QR, more than doubles performance. CholeskyQR gets the best performance of all the algorithms, but at the expense of significant loss of orthogonality when the initial matrix A is ill-conditioned. Note that in this case (Q and R

[3] See http://www.dolphinics.com/products/benchmarks.html.

[4] The BlueGene/L has two separate networks—a torus for nearest-neighbor communication and a tree for collectives. The latency and bandwidth figures here are for the collectives network.

TABLE 5.2

Run time in seconds of various parallel QR factorizations on the Beowulf machine, illustrating weak scaling with respect to the total number of rows m in the matrix. The ratio $\lceil m/P \rceil = 100{,}000$ and the total number of columns $n = 50$ were kept constant as the number of processors P varied from 1 to 64. If an algorithm scales perfectly, then all the run times in that algorithm's column should be constant. For those algorithms which compute an implicit representation of the Q factor, that representation was left implicit.

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK (PDGEQRF)
1	0.45	3.43	3.61	7.13	7.07	7.26
2	0.47	4.02	7.11	14.04	11.59	13.95
4	0.47	4.29	6.09	12.09	13.94	13.74
8	0.50	4.30	7.53	15.06	14.21	14.05
16	0.54	4.33	7.79	15.04	14.66	14.94
32	0.52	4.42	7.85	15.38	14.95	15.01
64	0.65	4.45	7.96	15.46	14.66	15.33

TABLE 5.3

Run time in seconds of various parallel QR factorizations on the Frost machine on a $10^6 \times 50$ matrix. This metric illustrates strong scaling (constant problem size, but number of processors increases).

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK (PDGEQRF)
32	0.140	0.453	0.836	0.694	1.132	1.817
64	0.075	0.235	0.411	0.341	0.570	0.908
128	0.038	0.118	0.180	0.144	0.247	0.399
256	0.020	0.064	0.086	0.069	0.121	0.212

requested), CholeskyQR, CGS, and MGS perform half the flops of the Householder-based algorithms, TSQR `DGEQR3`, TSQR `DGEQRF`, and `PDGEQRF` ($2mn^2$ versus $4mn^2$).

Table 5.2 shows the results of the second set of experiments on the Pentium III cluster. In these experiments, we also illustrate weak scaling with respect to the total number of rows m in the matrix. For this, the ratio $\lceil m/P \rceil = 100,000$ and the total number of columns $n = 50$ were kept constant as the number of processors P varied from 1 to 64. Unlike in the previous set of experiments, for those algorithms which compute an implicit representation of the Q factor, that representation was left implicit. The results show that TSQR scales well. In particular, when using TSQR with the recursive local QR factorization, there is almost no performance penalty for moving from one processor to two, unlike with CGS, MGS, and ScaLAPACK's QR. Again, the recursive local QR significantly improves TSQR performance; here it is the main factor in making TSQR perform better than ScaLAPACK's QR. Note that in this case (only R requested), CholeskyQR, performs half the flops of all the others algorithm CGS, MGS, TSQR `DGEQR3`, TSQR `DGEQRF`, and `PDGEQRF` (mn^2 versus $2mn^2$).

Table 5.3 shows the results of the third set of experiments, which was performed on the BlueGene/L cluster Frost. These data show performance per processor (Mflop / s / (number of processors)) on a matrix of constant dimensions $10^6 \times 50$, as the number of processors was increased. This illustrates strong scaling. If an algorithm scales perfectly, then all the numbers in that algorithm's column should decrease proportionally to P, i.e., halve from row to row. For ScaLAPACK's QR factorization, we used `PDGEQRF`. We observe that using the recursive local QR factorization with TSQR makes it clearly outperform ScaLAPACK. Note that, in this case (Q

and R requested), CholeskyQR, CGS, and MGS perform half the flops of the Householder based algorithms, TSQR `DGEQR3`, TSQR `DGEQRF`, and `PDGEQRF` ($2mn^2$ versus $4mn^2$).

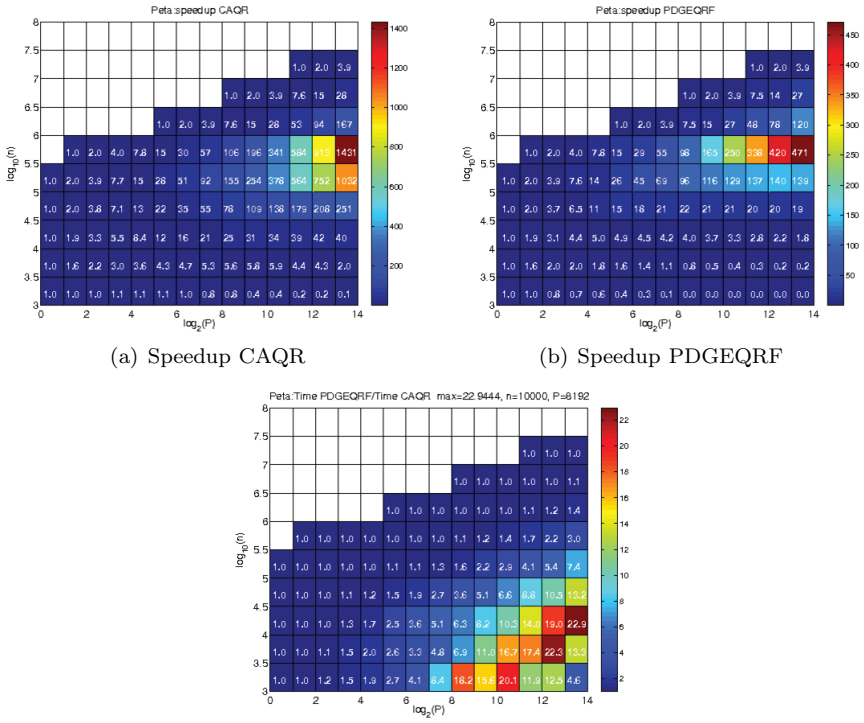
Both the Pentium III and BlueGene/L platforms have relatively slow processors with a relatively low-latency interconnect. TSQR was optimized for the opposite case of fast processors and expensive communication. Nevertheless, TSQR outperforms ScaLAPACK's QR by over 6.7 times on 16 processors (and 3.5 times on 64 processors) on the Pentium III cluster, and 4.0 times on 32 processors (and 3.3 times on 256 processors) on the BlueGene/L machine.

5.3. Performance estimation of parallel CAQR. We use the performance model developed in section 3.1 to estimate the performance of parallel CAQR on a model of a petascale machine. We expect CAQR to outperform ScaLAPACK, in part because it uses a faster algorithm for performing most of the computation of each panel factorization (`DGEQR3` versus `DGEQRF`) and in part because it reduces the latency cost. Our performance model uses the same time per floating-point operation for both CAQR and `PDGEQRF`. Hence our model evaluates the improvement due only to reducing the latency cost. Our projection of a petascale machine ("Peta") has 8192 processors. Each "processor" of Peta may itself be a parallel multicore node, but we consider it as a single fast sequential processor for the sake of our model. Here are the parameters we use: peak floating-point rate is 500 Gflop/s per processor, network latency is 10 μs, and network bandwidth is 4 GB/s.

We evaluate the performance using matrices of size $n \times n$, distributed over a $P_r \times P_c$ grid of P processors using a two-dimensional block cyclic distribution, with square blocks of size $b \times b$. We estimate the best performance of CAQR and `PDGEQRF` for a given problem size n and a given number of processors P by finding the optimal values for the block size b and the shape of the grid $P_r \times P_c$ in the allowed ranges. The block size b is varied in the range $1, 5, 10, \ldots, 50, 60, \ldots, \min(200, m/P_r, n/P_c)$. The values for P, P_r, and P_c are chosen to be powers of two. When we evaluate the model, we set the floating-point performance value in the model so that the modeled floating-point rate is 80% of the machine's peak rate, so as to capture realistic performance on the local QR factorizations. The white regions in the plots signify that the problem needed more memory than available on the machine.

Figure 5.2 shows our performance estimates of CAQR and `PDGEQRF` on the Petascale machine, in which we display (a) the best speedup obtained by CAQR, with respect to the run time using the fewest number of processors with enough memory to hold the matrix (which may be more than one processor); (b) the best speedup obtained by `PDGEQRF`, computed similarly; and (c) the ratio of `PDGEQRF` run time to CAQR run time.

As can be seen in Figure 5.2(a), CAQR is expected to show good scalability for large matrices. For example, for $n = 10^{5.5}$, a speedup of 1431, measured with respect to the time on 2 processors, is obtained on 8192 processors. For $n = 10^6$ a speedup of 167, measured with respect to the time on 32 processors, is obtained on 8192 processors. CAQR leads to more significant improvements when the latency represents an important fraction of the total time; see the bottom right of Figure 5.2(c). The best improvement is a factor of 22.9, obtained for $n = 10^4$ and $P = 8192$. The speedup of the best CAQR compared to the best `PDGEQRF` for $n = 10^4$ when using at most $P = 8192$ processors is larger than 8, which is still an important improvement. The best performance of CAQR is obtained for $P = 4096$ processors and the best performance of `PDGEQRF` is obtained for $P = 16$ processors.



(a) Speedup CAQR

(b) Speedup PDGEQRF

(c) Comparison

FIG. 5.2. *Performance prediction comparing CAQR and* `PDGEQRF` *on Peta.*

TABLE 5.4

Estimated run time of `PDGEQRF` *divided by estimated runtime of parallel CAQR on a square* $n \times n$ *matrix, on the Peta platform for those values of* P *(number of processors) for which* `PDGEQRF` *performs the best for that problem size.*

$\log_{10} n$	Best $\log_2 P$ for `PDGEQRF`	CAQR speedup
3.0	1	1
3.5	2–3	1.1–1.5
4.0	4–5	1.7–2.5
4.5	7–10	2.7–6.6
5.0	11–13	4.1–7.4
5.5	13	3.0
6.0	13	1.4

Useful improvements are also obtained for larger matrices. For $n = 10^6$, CAQR outperforms `PDGEQRF` by a factor of 1.4. When the computation dominates the parallel time, Figure 5.2(c) predicts that there is no benefit from using CAQR. However, CAQR is never slower. For any fixed n, we can take the number of processors P for which `PDGEQRF` would perform the best, and measure the speedup of CAQR over `PDGEQRF` using that number of processors. We do this in Table 5.4, which predicts that CAQR always is at least as fast as `PDGEQRF` and often significantly faster (up to 7.4 times faster in some cases).

6. Related work. The central idea in this paper is factoring tall skinny matrices using a tree-based Householder QR algorithm. A number of authors previously figured

out the special case of a binary reduction tree for parallel QR. As far as we know, Golub et al. [24] were the first to suggest it, but their formulation requires $n \log P$ messages for QR of an $m \times n$ matrix on P processors. Pothen and Raghavan [41] were the first to implement parallel TSQR using only $\log P$ messages. Da Cunha et al. [13] independently rediscovered parallel TSQR.

Other authors have worked out variations of the algorithm we call "sequential TSQR" [9, 10, 28, 35, 42, 43]. They do not use it by itself but rather use it as the panel factorization step in the QR decomposition of general matrices. The references [9, 10, 28, 35, 42] refer to the latter algorithm as "tiled QR," which is the same as our sequential CAQR with square blocks. However, they use it in parallel on shared-memory platforms, especially single-socket multicore. They do this by exploiting the parallelism implicit in the directed acyclic graph of tasks. Often they use dynamic task scheduling, which we could use but do not discuss in this paper. Since the cost of communication in the single-socket multicore regime is low, these authors are less concerned than we are about minimizing latency; thus, they are not concerned about the latency bottleneck in the panel factorization, which motivates our parallel CAQR algorithm. We also model and analyze communication costs in more detail than previous authors did.

Here are recent examples of related work on sequential CAQR. Gunter and van de Geijn develop a parallel out-of-DRAM QR factorization algorithm that uses a flat tree for the panel factorizations [28]. Buttari et al. suggest using a QR factorization of this type to improve performance of parallel QR on commodity multicore processors [9]. Quintana-Orti et al. develop two variations on block QR factorization algorithms and use them with a dynamic task scheduling system to parallelize the QR factorization on shared-memory machines [42].

A different "cache-oblivious" approach is presented by Frens and Wise [22], based on a recursive two-dimensional decomposition of the matrix and using Morton ordering of the subblocks. This sequential algorithm minimizes communication (bandwidth and latency costs) across multiple levels of memory hierarchy, but at the cost of possibly tripling the number of floating point operations.

As far as we know, CAQR that uses TSQR based on any reduction tree as a building block, and in particular parallel CAQR (based on a binary tree), is novel. Nevertheless, there is a body of work on theoretical bounds on exploitable parallelism in QR factorizations. These bounds apply to both parallel TSQR and parallel CAQR if one replaces "matrix element" in the authors' work with "block" in ours. Cosnard, Muller, and Robert proved lower bounds on the critical path length $Opt(m, n)$ of any parallel QR algorithm of an $m \times n$ matrix based on Givens rotations [11]; it is believed that these apply to any QR factorization based on Householder or Givens rotations. Leoncini et al. show that any QR factorization based on Householder reductions or Givens rotations is P-complete [37]. The only known QR factorization algorithm in arithmetic NC (see [12]) is numerically highly unstable [19], and no work suggests that a stable arithmetic NC algorithm exists.

Hong and Kung [31] and Irony, Toledo, and Tiskin [32] proved lower bounds on communication for sequential and parallel matrix multiplication, and Ballard et al. [7] extended these results to most "direct" linear algebra problems (LU, QR, finding eigenvalues and eigenvectors, etc.), for dense or sparse matrices, and for sequential and parallel machines. Elmroth and Gustavson proposed a recursive QR factorization (see [20, 21]) which can also take advantage of memory hierarchies. For parallel LU with pivoting that minimizes communication, see [26, 27], and for sequential LU, see [47, 27].

Block iterative methods frequently compute the QR factorization of a tall and skinny dense matrix. This includes algorithms for solving linear systems $Ax = B$ with multiple right-hand sides (such as variants of GMRES, QMR, or CG [48, 23, 40]), as well as block iterative eigensolvers (for a summary of such methods, see [4, 36]). In practice, MGS orthogonalization is usually used when a (reasonably) stable QR factorization is desired. Sometimes unstable methods (such as CholeskyQR) are used when performance considerations outweigh stability. Eigenvalue computation is particularly sensitive to the accuracy of the orthogonalization; two recent papers suggest that large-scale eigenvalue applications require a stable QR factorization [29, 33]. Many block iterative methods have widely used implementations, on which a large community of scientists and engineers depends for their computational tasks. Examples include TRLAN (thick restart Lanczos), BLZPACK (block Lanczos), Anasazi (various block methods), and PRIMME (block Jacobi–Davidson methods) [49, 38, 34, 3, 5, 46].

7. Conclusions and open problems. We have presented lower bounds on the bandwidth costs (number of words moved) and latency costs (number of messages) for parallel and sequential dense QR factorization and presented some new and some old QR algorithms that attain these bounds. We have also presented a simple reduction argument to derive similar lower bounds for LU factorization and referred to LU algorithms in the literature that attain at least some of these bounds.

There are numerous ways in which one could hope to extend these results. For example, one could seek dense linear algebra algorithms for other problems, like eigenproblems, that attain the same communication lower bounds [7]. One could also seek algorithms attaining the corresponding lower bounds for sparse matrices [7]. Or one could seek analogous communication lower bounds for asymptotically faster dense linear algebra algorithms like those based on Strassen's algorithm, or indeed of any matrix multiplication algorithm, based on Raz's theorem converting any matrix multiplication algorithm to be "Strassen-like" (bilinear noncommutative) [44].

But the following question is perhaps of more practical importance. Our TSQR and CAQR algorithms have been described and analyzed in most detail for simple machine models: either sequential with two levels of memory hierarchy (fast and slow) or a homogeneous parallel machine, where each processor is itself sequential. Real computers are more complicated, with many levels of memory hierarchy and many levels of parallelism (multicore, multisocket, multinode, multirack...) all with different bandwidths and latencies. So it is natural to ask whether our algorithms and optimality proofs can be extended to these more general situations. We hinted at how TSQR could be extended to general reduction trees in section 2, which could in turn be chosen depending on the architecture. But we have not discussed CAQR, which we do here.

First suppose there are multiple, nested levels of memory hierarchy (with words in the cache at level i also contained inside the larger, slower cache at level $i+1$). Then by applying our lower bounds at each level (treating levels 1 through i as a single level, and levels $i+1$ and larger as another), we get lower bounds on communication between each pair of consecutive levels. It is known in the case of matrix multiplication that either recursive "cache-oblivious" algorithms can attain all these lower bounds or algorithms with explicit caching at each level.

It is natural to ask how this extends to QR. As mentioned in section 6, Frens and Wise [22] have a sequential cache oblivious QR algorithm that minimizes both latency and bandwidth costs across multiple layers of memory hierarchy, but at the cost of up to three times more flops. The sequential cache-oblivious algorithm due

to Elmroth and Gustavson [20, 21] minimizes only bandwidth costs, not latency. So it is also natural to ask whether CAQR can be organized to work with multiple levels of memory hierarchy; the challenge is that the subproblems into which CAQR decomposes the subproblem are not all "smaller CAQR" that would permit obvious recursion.

The problem becomes more challenging with multiple levels of parallelism as well. Finally, we have been assuming the machines are *homogeneous*, that all processing elements and memories (at the same level) have common processing rates, sizes, bandwidths, and latencies. But many architectures are *heterogeneous*, e.g., consist of multiple CPUs and GPUs with different properties. Deriving lower bounds, and algorithms that attain them, for this broad array of emerging architectures is future work.

Appendix A. Memory traffic for right-looking BLAS 3 QR.

In this section, we estimate the amount of data movement between fast and slow memory performed by LAPACK's sequential QR factorization `DGEQRF` for an $m \times n$ matrix with $m \geq n$. `DGEQRF` uses a right-looking BLAS 3 algorithm with panel width b. Its volume of data movement (if reads and writes are counted together, as we do) does not differ significantly from that of the left-looking out-of-core algorithm `PFDGEQRF` mentioned in section 3.2. That algorithm is described and analyzed in detail in [16, Appendix F]. The important difference is that `PFDGEQRF` requires $mb < W$: the panel of width b must fit in fast memory, and at least one column of the trailing matrix must also reside in fast memory. The LAPACK algorithm `DGEQRF` does not require this; in fact, not even an entire row of the matrix need fit in fast memory ($W < n$ is allowed). This matters because in some cases, choosing b so that the panel does *not* fit in fast memory reduces overall memory traffic. This counterintuitive result holds because the trailing matrix update also contributes to memory traffic: wider panels mean fewer trailing matrix updates, though they may result in more data movement from the panel factorizations.

We include this appendix because we wish to demonstrate that while `DGEQRF`'s greater freedom to choose the panel width may result in less data movement than the `PFDGEQRF` algorithm (see Case 3 in section A.2), both algorithms nevertheless still require asymptotically more data movement than sequential CAQR and than the lower bound, for almost all matrix dimensions and fast memory sizes of practical interest. We demonstrate this by estimating up to a small constant factor the volume of data movement performed by `DGEQRF`, as a function of the panel width b (which may range from 1 to n inclusive). In practice, the panel width resulting in best performance depends on the particular computer architecture. It is chosen in practice via performance tuning (a typical value may be 16 or 32). Instead, we will choose b here in order to minimize the volume of data movement, as a function of the matrix dimensions m and n and the fast memory capacity W. We will show that with an optimally chosen panel width, `DGEQRF` requires asymptotically more memory traffic than the lower bound $mn^2/\sqrt{W}$ for QR factorizations for nearly all matrix dimensions and fast memory sizes of interest.

A.1. Memory traffic as a function of panel width. We first estimate the volume of memory traffic for `DGEQRF` as a function of the panel width b. In section A.2, we will choose b to minimize memory traffic. In this appendix, we label the number of floating-point words transferred between slow and fast memory as $\text{Words}(m, n, W, b)$. In our model, this is a function of the matrix dimensions m and n, the fast memory size W, and the panel width b. In this analysis, we ignore insignificant constant factors and lower order terms. In particular, we neglect the fact that the panels and

trailing matrix shrink as the factorization progresses. Since we assume $m \geq n$, the shrinkage reduces memory traffic by no more than half (in the highest order term). We will estimate $\text{Words}(m, n, W, b)$ as a maximum of three terms. The analysis splits into two cases:

- $mb \leq W$ (the $m \times b$ panel fits in fast memory) and
- $mb > W$ (the $m \times b$ panel does *not* fit in fast memory).

Recall that the implementation of the left-looking out-of-core algorithm mentioned in section 3.2 and described in detail in [16, Appendix F] requires $mb < W$, so the second case above applies only to `DGEQRF`. The first case applies to both the out-of-core and in-core algorithms.

First, suppose that the $m \times b$ panel fits in fast memory: $mb \leq W$. Then, up to some small constant factor, the volume of memory traffic is given by

$$\text{Words}(m, n, W, b) = \max\left\{mn, \frac{mn^2}{\sqrt{W}}, \frac{mn^2}{b}\right\}. \tag{A.1}$$

The mn term comes from reading the matrix from slow memory into fast memory, which is a strict lower bound on memory traffic for the QR factorization (which depends on all the matrix entries). The $mn^2/\sqrt{W}$ and mn^2/b terms come from two things:

- the BLAS 3 update of the trailing matrix, at a cost of $\max\{mnb/\sqrt{W}, mn, mb\}$ words for each update; the total number of updates is n/b, so the total cost is $\max\{mn^2/\sqrt{W}, mn^2/b, mn\}$;
- the n/b panel factorizations, each of an $m \times b$ panel; each panel factorization costs mb words, for a total cost of mn.

Next, suppose that the $m \times b$ panel does not fit in fast memory: $mb > W$. (We explained above why the algorithm will still work in this case. This is what distinguishes the analysis in this section from the analysis of the left-looking out-of-core factorization in section 3.2.) Then, up to some small constant factor, the volume of memory traffic is given by

$$\text{Words}(m, n, W, b) = \max\left\{mnb, \frac{mn^2}{\sqrt{W}}, \frac{mn^2}{b}\right\}. \tag{A.2}$$

The mnb term comes from the n/b panel factorizations, each of which requires the worst case of mb^2 memory accesses. The other two terms come from the n/b updates of the trailing matrix, at a cost of $\max\{mnb/\sqrt{W}, mn, mb\}$ words per update, for a total cost of $\max\{mn^2/\sqrt{W}, mn^2/b, mn\}$ words. Since $mnb > mn/b$, we leave out the mn/b term for equation (A.2).

A.2. Optimal panel width. In this section, we minimize the memory traffic formulae derived in section A.2. We do so by choosing the panel width b as a function of the matrix dimensions m and n and the fast memory size W. This means minimizing the one of (A.1) or (A.2) that results in the least memory traffic, depending on m, n, and W. We find the optimal b by considering five different cases of values of W:

1. $1 \leq W < n$ (not even one row fits in fast memory);
2. $n \leq W < m$ (at least one row, but not one column, fits in fast memory; recall that we assume $m \geq n$);
3. $m \leq W < m\sqrt{n}$ (at least one column, but less than $\sqrt{n}$ columns, fit in fast memory);

4. $m\sqrt{n} \leq W < mn$ (at least $\sqrt{n}$ columns, but not the whole matrix, fit in fast memory); and
5. $W \geq mn$ (the whole matrix fits in fast memory).

The factorization has minimal memory traffic when

$$\min_b \text{Words}(m, n, W, b) = \max\left\{mn, \frac{mn^2}{\sqrt{W}}\right\}. \tag{A.3}$$

For each case, we will point out whether it attains this minimum for the best possible choice of panel width b. We divide the cases into two groups: $W < m$ includes Cases 1 and 2, and $W \geq m$ includes Cases 3, 4, and 5.

Suppose first that $W < m$. Then $mb > W$ for any choice of panel width b, and equation (A.2) applies. To minimize that expression, choose b so $mnb = mn^2/b$, i.e., $b = \sqrt{n}$. Thus, $\min_b \text{Words}(m, n, W, b) = \max\{mn^{3/2}, mn^2/\sqrt{W}\}$. We can split this into two cases:

- Case 1 ($W < n \leq m$—not even one row fits in fast memory): Choose $b = \sqrt{n}$, so that $\min_b \text{Words}(m, n, W, b) = mn^2/\sqrt{W}$. This attains the desired lower bound of $\max\{mn, mn^2/\sqrt{W}\}$ (equation (A.3)).
- Case 2 ($n \leq W < m$—at least one row, but not one column, fits in fast memory): Choose $b = \sqrt{n}$, so that $\min_b \text{Words}(m, n, W, b) = mn^{3/2}$. This does *not* attain the desired lower bound of $\max\{mn, mn^2/\sqrt{W}\}$, since $n \leq W$.

Now suppose that $W \geq m$. One possibility is to choose b as large as possible subject to $mb <= W$ and $b <= n$, i.e., $b = \min\{W/m, n\}$. This yields

$$\begin{aligned} \min_b \text{Words}(m, n, W, b) &= \max\left\{mn, \frac{mn^2}{\sqrt{W}}, \frac{mn^2}{b}\right\} \\ &= \max\left\{mn, \frac{mn^2}{\sqrt{W}}, \frac{m^2n^2}{W}\right\} \\ &= \frac{m^2n^2}{W}. \end{aligned} \tag{A.4}$$

We justify the third line of (A.4) as follows. First, comparing $mn^2/\sqrt{W} < m^2n^2/W$ simplifies to $W < m^2$, which is true (since $m \geq n$ and the matrix does not fit in fast memory). Second, comparing $mn < m^2n^2/W$ simplifies to $W < mn$, which is also true (since the matrix does not fit in fast memory). Note that m^2n^2/W is the memory traffic model for the out-of-core algorithm `PFDGEQRF`. Allowing $b > W/m$ is impossible for that algorithm, but possible for `DGEQRF`. If we do so, then (A.2) applies. We can divide the analysis into three cases, based on the value of W:

- Case 3 ($m \leq W < m\sqrt{n}$): Choose $b = \sqrt{n} > W/m$. With that value of b, $\min_b \text{Words}(m, n, W, b) = mn^{3/2}$. As in Case 2 above, this does *not* attain the desired lower bound of $mn^2/\sqrt{W}$. Note that this case distinguishes LAPACK's QR factorization from the out-of-core left-looking factorization analyzed in section 3.2. There, we require that the panel and at least one column of the trailing matrix fit in fast memory, which constrains $b < W/m$. LAPACK's QR factorization allows the panel not to fit in fast memory.
- Case 4 ($m\sqrt{n} \leq W < mn$): Choose $b = W/m$. This means (A.4) applies. Since $W < mn$, $m^2n^2/W > mn$. Since $W \geq m\sqrt{n}$, $mn^2/\sqrt{W} \geq m^2n^{5/2} \geq mn$. Finally, $(m^2n^2/W)/(mn^2/\sqrt{W}) = m/\sqrt{W}$, and since $m\sqrt{n} \leq W$, $m/\sqrt{W} \geq m^{1/2}/n^{1/4} \geq 1$ (since we assume $m \geq n$). Thus, the memory traffic is given by $\min_b \text{Words}(m, n, W, b) = m^2n^2/W$, which exceeds the desired lower bound by a factor of $m/\sqrt{W} \geq m^{1/2}/n^{1/4} \geq m^{1/4}$.

- Case 5 ($mn \leq W$): Choose $b = n$, so that $\min_b \text{Words}(m, n, W, b) = mn$. This attains the desired lower bound, which is easy to see because in this case, the matrix fits entirely in fast memory. For this case, the optimal block size is $b = n$, meaning that the ordinary Householder (BLAS 2) QR factorization suffices for minimal memory traffic.

In summary, the LAPACK QR factorization requires asymptotically more memory traffic than the lower bound, when at least one row fits in fast memory, but the whole matrix does not fit in fast memory. The worst case is the the boundary between Cases 3 and 4, namely, when $W = m\sqrt{n}$. There, $\min_b \text{Words}(m, n, W, b) = mn^{3/2}$, which exceeds the lower bound $mn^2/\sqrt{W}$ by a factor of $\sqrt{W/n}$ when at least one row of the matrix fits in fast memory.

REFERENCES

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

[2] M. Baboulin, L. Giraud, S. Gratton, and J. Langou, *Parallel Tools for Solving Incremental Dense Least Squares Problems. Application to Space Geodesy*, Technical report UT-CS-06-582, University of Tennessee, Knoxville, 2006.

[3] J. Baglama, D. Calvetti, and L. Reichel, *Algorithm* 827: *Irbleigs: A MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix*, ACM Trans. Math. Software, 29 (2003), pp. 337–348.

[4] Z. Bai and D. Day, *Block Arnoldi method*, in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Vorst, eds., SIAM, Philadelphia, 2000, pp. 196–204.

[5] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist, *Anasazi webpage*, http://trilinos.sandia.gov/packages/anasazi.

[6] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing Communication in Numerical Linear Algebra*. Technical report UCB/EECS-2011-15, University of California, Berkeley, CA, 2011.

[7] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, *Minimizing communication in numerical linear algebra*, SIAM J. Matrix Anal. Appl., 32 (2011), pp. 866–901.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, 1997.

[9] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, *A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures*, Technical report UT-CS-07-600, University of Tennessee, Knoxville, 2007.

[10] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, *Parallel tiled QR factorization for multicore architectures*, Technical report UT-CS-07-598, University of Tennessee, Knoxville, 2007.

[11] M. Cosnard, J.-M. Muller, and Y. Robert, *Parallel QR decomposition of a rectangular matrix*, Numer. Math., 48 (1986), pp. 239–249.

[12] L. Csanky, *Fast parallel matrix inversion algorithms*, SIAM J. Comput., 5 (1976), pp. 618–623.

[13] R. D. D. Cunha, D. Becker, and J. C. Patterson, *New parallel (rank-revealing) QR factorization algorithms*, in Proceedings of the Euro-Par 2002. Parallel Processing: Eighth International Euro-Par Conference, Paderborn, Germany, 2002.

[14] E. F. D'Azevedo and J. J. Dongarra, *The Design and Implementation of the Parallel Out-of-Core ScaLAPACK LU, QR, and Cholesky Factorization Routines*, Technical report 118 CS-97-247, University of Tennessee, Knoxville, 1997.

[15] E. D'Azevedo and J. Dongarra, *The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines*, Concurrency Practice Experience, 12 (2000), pp. 1481–1483.

[16] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-Avoiding Parallel and Sequential QR and LU Factorizations: Theory and Practice*, Technical report UCB/EECS-2008-89, University of California, Berkeley, CA, 2008.

[17] J. W. Demmel, M. Hoemmen, Y. Hida, and E. J. Riedy, *Nonnegative diagonals and high performance on low-profile matrices from Householder QR*, SIAM J. Sci. Comput., 31 (2009), pp. 2832–2841.
[18] J. W. Demmel, M. Hoemmen, M. Mohiyuddin, and K. A. Yelick, *Minimizing communication in sparse matrix solvers*, in Proceedings of the 2009 ACM/IEEE Conference on Supercomputing, New York, 2009.
[19] J. W. Demmel, *Trading Off Parallelism and Numerical Stability*, Technical report UT-CS-92-179, University of Tennessee, Knoxville, 1992.
[20] E. Elmroth and F. Gustavson, *New serial and parallel recursive QR factorization algorithms for SMP systems*, in Proceedings of the Fourth International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems, B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, eds., Lecture Notes in Comput. Sci. 1541, Springer, New York, 1998, pp. 120–128.
[21] E. Elmroth and F. Gustavson, *Applying recursion to serial and parallel QR factorization leads to better performance*, IBM J. Res. Develop., 44 (2000), pp. 605–624.
[22] J. D. Frens and D. S. Wise, *QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism*, SIGPLAN Not., 38 (2003), pp. 144–154.
[23] R. W. Freund and M. Malhotra, *A block QMR algorithm for non-Hermitian linear systems with multiple right-hand sides*, Linear Algebra Appl., 254 (1997), pp. 119–157.
[24] G. H. Golub, R. J. Plemmons, and A. Sameh, *Parallel block schemes for large-scale least-squares computations*, in High-Speed Computing: Scientific Applications and Algorithm Design, R. B. Wilhelmson, ed., University of Illinois Press, Chicago, IL, 1988, pp. 171–179.
[25] S. L. Graham, M. Snir, and C. A. Patterson, eds., *Getting Up to Speed: The Future of Supercomputing*, National Academies Press, Washington, D.C., 2005.
[26] L. Grigori, J. W. Demmel, and H. Xiang, *Communication avoiding Gaussian elimination*, Proceedings of the ACM/IEEE SC08 Conference, 2008.
[27] L. Grigori, J. W. Demmel, and H. Xiang, *CALU: A Communication Optimal LU Factorization Algorithm*, Technical report UCB-EECS-2010-29, INRIA, 2010.
[28] B. C. Gunter and R. A. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Trans. Math. Software, 31 (2005), pp. 60–78.
[29] U. Hetmaniuk and R. Lehoucq, *Basis selection in LOBPCG*, J. Comput. Phys., 218 (2006), pp. 324–332.
[30] M. Hoemmen, *Communication-Avoiding Krylov Subspace Methods*, Ph.D. thesis, EECS Department, University of California, Berkeley, CA, 2010.
[31] J. W. Hong and H. T. Kung, *I/O complexity: The red-blue pebble game*, in STOC '81: Proceedings of the 13th Annual ACM Symposium on Theory of Computing, New York, 1981, pp. 326–333.
[32] D. Irony, S. Toledo, and A. Tiskin, *Communication lower bounds for distributed-memory matrix multiplication*, J. Parallel Distrib. Comput., 64 (2004), pp. 1017–1026.
[33] A. V. Knyazev, M. Argentati, I. Lashuk, and E. E. Ovtchinnikov, *Block Locally Optimal Preconditioned Eigenvalue Xolvers (BLOPEX) in HYPRE and PETSc*, Technical report UCDHSC-CCM-251P, University of California, Davis, 2007.
[34] A. V. Knyazev, *BLOPEX*, http://www-math.cudenver.edu/~aknyazev/software/BLOPEX.
[35] J. Kurzak and J. J. Dongarra, *QR Factorization for the CELL Processor*, Technical report UT-CS-08-616, University of Tennessee, Knoxville, 2008.
[36] R. Lehoucq and K. Maschhoff, *Block Arnoldi method*, in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Z. Bai, J. W. Demmel, J. J. Dongarra, A. Ruhe, and H. van der Vorst, eds., SIAM, Philadelphia, 2000, pp. 185–187.
[37] M. Leoncini, G. Manzini, and L. Margara, *Parallel complexity of numerically accurate linear system solvers*, SIAM J. Comput., 28 (1999), pp. 2030–2058.
[38] O. Marques, *BLZPACK*, http://crd.lbl.gov/~osni.
[39] R. Nishtala, G. Almási, and C. Caşcaval, *Performance without pain = productivity: Data layout and collective communication in UPC*, in Proceedings of the ACM SIGPLAN 2008 Symposium on Principles and Practice of Parallel Programming, 2008.
[40] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra Appl., 29 (1980), pp. 293–322.
[41] A. Pothen and P. Raghavan, *Distributed orthogonal factorization: Givens and Householder algorithms*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 1113–1134.
[42] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, F. G. V. Zee, and R. A. van de Geijn, *Scheduling of QR factorization algorithms on SMP and multi-core architectures*, in Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, Toulouse, France, 2008.

[43] E. Rabani and S. Toledo, *Out-of-core SVD and QR decompositions*, in Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing, Norfolk, VA, 2001.
[44] R. Raz, *On the complexity of matrix product*, SIAM J. Comput., 32 (2003), pp. 1356–1369.
[45] R. Schreiber and C. Van Loan, *A storage efficient WY representation for products of Householder transformations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 53–57.
[46] A. Stathopoulos, *PRIMME*, http://www.cs.wm.edu/~andreas/software.
[47] S. Toledo, *Locality of reference in LU decomposition with partial pivoting*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 1065–1081.
[48] B. Vital, *Étude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*, Ph.D. thesis, Université de Rennes I, Rennes, France, 1990.
[49] K. Wu and H. D. Simon, *TRLAN*, http://crd.lbl.gov/~kewu/ps/trlan_.html.