Communication-avoiding parallel and sequential QR factorizations

James Demmel, Laura Grigori,
Mark Hoemmen, and Julien Langou

May 30, 2008

Abstract

We present parallel and sequential dense QR factorization algorithms that are optimized to avoid communication. Some of these are novel, and some extend earlier work. Communication includes both messages between processors (in the parallel case), and data movement between slow and fast memory (in either the sequential or parallel cases).

Our first algorithm, Tall Skinny QR (TSQR), factors $m \times n$ matrices in a one-dimensional (1-D) block cyclic row layout, storing the Q factor (if desired) implicitly as a tree of blocks of Householder reflectors. TSQR is optimized for matrices with many more rows than columns (hence the name). In the parallel case, TSQR requires no more than the minimum number of messages $\Theta(\log P)$ between P processors. In the sequential case, TSQR transfers $2mn + o(mn)$ words between slow and fast memory, which is the theoretical lower bound, and performs $\Theta(mn/W)$ block reads and writes (as a function of the fast memory size W), which is within a constant factor of the theoretical lower bound. In contrast, the conventional parallel algorithm as implemented in ScaLAPACK requires $\Theta(n \log P)$ messages, a factor of n times more, and the analogous sequential algorithm transfers $\Theta(mn^2)$ words between slow and fast memory, also a factor of n times more. TSQR only uses orthogonal transforms, so it is just as stable as standard Householder QR. Both parallel and sequential performance results show that TSQR outperforms competing methods.

Our second algorithm, CAQR (Communication-Avoiding QR), factors general rectangular matrices distributed in a two-dimensional block cyclic layout. It invokes TSQR for each block column factorization, which both remove a latency bottleneck in ScaLAPACK's current parallel approach, and both bandwidth and latency bottlenecks in ScaLAPACK's out-of-core QR factorization. CAQR achieves modeled speedups of 2.1× on an IBM POWER5 cluster, 3.0× on a future petascale machine, and 3.8× on the Grid.

Contents

1 Introduction

We present communication-avoiding parallel and sequential algorithms for the QR factorization of dense matrices. The algorithms, which we call "Communication-Avoiding QR" (CAQR) in common, are based on reformulations of the QR factorization. They feature a decrease in the number of messages exchanged between processors (in the parallel case) and additionally the volume of data moved between different levels of the memory hierarchy (in the sequential case) during the factorization, at the cost of some redundant computations. This design is motivated by the exponentially growing gaps between floating-point arithmetic rate, bandwidth, and latency, for many different storage devices and networks on modern high-performance computers (see e.g., Graham et al. [20]).

General CAQR operates on dense matrices stored in a two-dimensional (2-D) block cyclic layout. It exploits what we call "Tall Skinny QR" (TSQR) for block column factorizations. TSQR takes its name from the "tall and skinny" input matrices for which it is optimized. Such matrices have a number of rows m which is much larger than the number of columns n. TSQR assumes they are stored in a 1-D block cyclic row layout. The algorithm works by reorganizing the QR decomposition to work like a reduction, executed on a tree. In this case, though, each reduction step is a local QR factorization. The tree's shape is a tuning parameter that makes the algorithm broadly applicable. For example, in the sequential case, a "flat tree" (linear chain) achieves the minimum total volume of data transferred and number of messages sent between levels of the memory hierarchy, for any sequential QR factorization of a matrix in that layout, or indeed in any layout. In the parallel distributed-memory case, TSQR on a binary tree minimizes the number of messages for any parallel QR factorization of a matrix in that layout, and has about the same message volume as the usual algorithm. We summarize our performance models of TSQR and CAQR in Tables 1 resp. 2. An important characteristic of TSQR is that it only uses orthogonal transformations, so it has the same accuracy of the QR factorization implemented in ScaLAPACK.

	Parallel		Sequential	
	TSQR	ScaLAPACK	TSQR	ScaLAPACK
# messages	$\log(P)$	$2n \log(P)$	$\frac{2mn}{W}$	$\Theta\left(\left(\frac{mn}{W}\right)^2\right)$
# words	$\frac{n^2}{2} \log(P)$	$\frac{n^2}{2} \log(P)$	$2mn$	$2mn + \frac{m^3 n^2}{W^2}$
# flops	$\frac{2mn^2}{P} + \frac{2}{3}n^3 \log(P)$	$\frac{2mn^2}{P} + \frac{n^2}{2} \log(P)$	$2mn^2$	$2mn^2 - \frac{2}{3}n^3$

Table 1: Performance models of the TSQR and ScaLAPACK QR factorizations on an $m \times n$ matrix with P processors (in the parallel case) or a fast memory size of W floating-point words (in the sequential case). Units of message volume are number of floating-point words. In the parallel case, everything (messages, words, and flops) is counted along the critical path.

TSQR is also useful as a stand-alone algorithm. Parallel or out-of-DRAM QR factorizations of tall and skinny matrices in block (cyclic) row format arise in

	Parallel		Sequential	
	TSQR	ScaLAPACK	TSQR	ScaLAPACK
# messages	$\frac{3}{2}\sqrt{P}\log P$	$\frac{3}{2}n\log P$	$\Theta\left(\frac{n^3}{W^{3/2}}\right)$	$\Theta\left(\frac{n^4}{W^2}\right)$
# words	$n^2\frac{\log P}{\sqrt{P}}$	$\frac{3n^2\log(P)}{4\sqrt{P}}$	$\Theta\left(\frac{n^3}{W^{1/2}}\right)$	$\frac{n^5}{W^2}$
# flops	$\Theta(n^3\frac{\log P}{P})$	$\Theta\left(\frac{n^3}{P}\right)$	$2n^3+\frac{2n^3}{\sqrt{P}}$	$\frac{4}{3}n^3$

Table 2: Performance models of CAQR on a square $n \times n$ matrix in a 2-D block layout on a grid of $\sqrt{P} \times \sqrt{P}$ processors (in the parallel case) or a fast memory size of W floating-point words (in the sequential case). In the parallel case, everything (messages, words, and flops) is counted along the critical path. Block size for the models in this table is $n/\sqrt{P}$ by $n/\sqrt{P}$. For the performance model when using square blocks of any dimension, refer to Equation (A 17) in Appendix E.

many different applications. Highly overdetermined least squares problems are a natural application, as well as certain Krylov subspace methods. See Section 3 for details. The usual QR factorization algorithms for these problems either minimize communication as we do but are not stable (because they use the normal equations, forming $Q = AR^{-1}$ where R is the Cholesky factor of A^TA), or are stable but communicate more (such as the usual Householder-based factorization, which is implemented in ScaLAPACK). TSQR, which we describe in more detail in later sections, both minimizes communication and is stable. See Section 10 for a discussion of the stability of various orthogonalization methods.

The tree-based QR idea itself is not novel (see for example, [10, 23, 38, 40]), but we have a number of optimizations and generalizations:

- Our algorithm can perform almost all its floating-point operations using any fast sequential QR factorization routine. In particular, we can achieve significant speedups by invoking Elmroth and Gustavson's recursive QR (see [15, 16]).
- We apply TSQR to the parallel factorization of arbitrary rectangular matrices in a two-dimensional block cyclic layout.
- We adapt TSQR to work on general reduction trees. This flexibility lets schedulers overlap communication and computation, and minimize communication for more complicated and realistic computers with multiple levels of parallelism and memory hierarchy (e.g., a system with disk, DRAM, and cache on multiple boards each containing one or more multicore chips of different clock speeds, along with compute accelerator hardware like GPUs).
- We prove in the case of a 1-D block layout that both our parallel and sequential algorithms minimize bandwidth and latency costs. We make the same conjecture, with supporting arguments, for the case of a 2-D block layout.

We note that the Q factor is 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 needed for solving least squares problems. Adjoining the right-hand side(s) to the matrix A, and taking the QR factorization of the result, requires only the R factor. Previous authors discuss this optimization. However, many of our applications require storing and working with the implicit representation of the Q factor. Furthermore, applying this implicit representation has nearly the same performance model as constructing an explicit Q factor with the same dimensions as A.

The rest of this report is organized as follows. Section 2 lists terms and abbreviations that we use throughout the report. Section 3 gives our motivations for (re-)discovering and expanding upon TSQR, and developing CAQR. After that, we begin the exposition of TSQR (the block column QR factorization) by demonstrating its matrix algebra informally in Section 4. Section 5 illustrates how TSQR is actually a reduction, introduces corresponding terminology, and discusses some design choices. After that, Section 6 shows how to save flops and storage and improve performance on the local QR factorizations in TSQR. We summarize our machine model in Section 7, and describe the TSQR algorithm and its performance model in Section 8. Section 9 includes performance models of competing "tall skinny" QR factorization algorithms, and Section 10 briefly compares their numerical stability. Discussion of the tall skinny case ends with Section 12, which gives parallel and sequential performance results for TSQR. We describe machines of interest for our TSQR experiments and models in Section 11, and show parallel and sequential benchmark results for TSQR and other QR factorization algorithms in Section 12.

Section 13 begins our description of the general 2-D block cyclic QR factorization algorithm CAQR. We construct performance models of CAQR and ScaLAPACK's QR factorization routine in Section 14. Then, we use these models to predict performance on a variety of realistic parallel machines in Section 15.

We conclude the main body of our work with two sections of theoretical lower bounds. Section 16 presents actual and conjectured lower bounds on the amount of communication a parallel or sequential Householder- or Givens-based QR factorization must perform. It shows that TSQR is optimal, and argues that CAQR is optimal as well. Section 17 cites lower bounds on the critical path length for a parallel QR factorization based on Householder reflections or Givens rotations. Both TSQR and CAQR attain these bounds.

The various Appendices present the algebra of our performance models in great detail.

2 List of terms and abbreviations

alpha-beta model A simple model for communication time, involving a latency parameter α and an inverse bandwidth parameter β: the time to

transfer a single message containing n words is $\alpha + \beta n$.

CAQR Communication-Avoiding QR – a parallel and/or *explicitly swapping* QR factorization algorithm, intended for input matrices of general shape. Invokes *TSQR* for panel factorizations.

CholeskyQR A fast but numerically unstable QR factorization algorithm for tall and skinny matrices, based on the Cholesky factorization of $A^T A$.

`DGEQRF` LAPACK QR factorization routine for general dense matrices of double-precision floating-point numbers. May or may not exploit shared-memory parallelism via a multithreaded BLAS implementation.

GPU Graphics processing unit.

Explicitly swapping Refers to algorithms explicitly written to save space in one level of the memory hierarchy ("fast memory") by using the next level ("slow memory") as swap space. Explicitly swapping algorithms can solve problems too large to fit in fast memory. Special cases include *out-of-DRAM* (a.k.a. out-of-core), *out-of-cache* (which is a performance optimization that manages cache space explicitly in the algorithm), and algorithms written for processors with non-cache-coherent local scratch memory and global DRAM (such as Cell).

Flash drive A persistent storage device that uses nonvolatile flash memory, rather than the spinning magnetic disks used in hard drives. These are increasingly being used as replacements for traditional hard disks for certain applications. Flash drives are a specific kind of solid-state drive (SSD), which uses solid-state (not liquid, gas, or plasma) electronics with no moving parts to store data.

Local store A user-managed storage area which functions like a cache (in that it is smaller and faster than main memory), but has no hardware support for cache coherency.

Out-of-cache Refers to algorithms explicitly written to save space in cache (or local store), by using the next larger level of cache (or local store), or main memory (DRAM), as swap space.

Out-of-DRAM Refers to algorithms explicitly written to save space in main memory (DRAM), by using disk as swap space. ("Core" used to mean "main memory," as main memories were once constructed of many small solenoid cores.) See *explicitly swapping*.

`PDGEQRF` ScaLAPACK parallel QR factorization routine for general dense matrices of double-precision floating-point numbers.

`PFDGEQRF` ScaLAPACK parallel out-of-core QR factorization routine for general dense matrices of double-precision floating-point numbers.

TSQR Tall Skinny QR – our reduction-based QR factorization algorithm, intended for "tall and skinny" input matrices (i.e., those with many more rows than columns).

3 Motivation for TSQR

3.1 Block iterative methods

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 [47, 17, 36]), as well as block iterative eigensolvers (for a summary of such methods, see [3, 30]). Many of these 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, 33, 28, 2, 39, 44]. 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 [24, 29].

3.2 s-step Krylov methods

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. Demmel et al. review the existing literature and discuss new advances in this area [13]. Such a method begins with an $n \times n$ matrix A and a starting vector v, and generates some basis for the Krylov subspace $\text{span}\{v, Av, A^2v, \ldots, A^sv\}$, using a small number of communication steps that is independent of s. Then, a QR factorization is used to orthogonalize the basis vectors.

The goal of combining s steps into one is to leverage existing basis generation algorithms that reduce the number of messages and/or the volume of communication between different levels of the memory hierarchy and/or different processors. These algorithms make the resulting number of messages independent of s, rather than growing with s (as in standard Krylov methods). However, this means that the QR factorization is now the communications bottleneck, at least in the parallel case: the current PDGEQRF algorithm in ScaLAPACK takes at least $s \log_2 P$ messages (in which P is the number of processors), instead of $\log_2 P$ messages for TSQR. Numerical stability considerations limit s, so that it is essentially a constant with respect to the matrix size m. Furthermore, a stable QR factorization is necessary in order to restrict the loss of stability caused by generating s steps of the basis without intermediate orthogonalization. This is an ideal application for TSQR, and in fact inspired its (re-)discovery.

3.3 Panel factorization in general QR

Householder QR decompositions of tall and skinny matrices also comprise the panel factorization step for typical QR factorizations of matrices in a more general, two-dimensional layout. This includes the current parallel QR code PDGEQRF in ScaLAPACK, as well as ScaLAPACK's out-of-DRAM QR factorization PFDGEQRF. 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. Replacing the existing panel factorization with TSQR would reduce this cost by a factor equal to the number of columns in a panel, thus removing the bottleneck. TSQR requires more floating-point operations, though some of this computation can be overlapped with communication. Section 13 will discuss the advantages of this approach in detail.

4 TSQR matrix algebra

In this section, we illustrate the insight behind the TSQR algorithm. TSQR uses a reduction-like operation to compute the QR factorization of an $m \times n$ matrix A, stored in a 1-D block row layout.[1] We begin with parallel TSQR on a binary tree of four processors ($P = 4$), and later show sequential TSQR on a linear tree with four blocks.

4.1 Parallel TSQR on a binary tree

The basic idea of using a reduction on a binary tree to compute a tall skinny QR factorization has been rediscovered more than once (see e.g., [10, 38]). We repeat it here in order to show its generalization to a whole space of algorithms. First, we decompose the $m \times n$ matrix A into four $m/4 \times n$ block rows:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}.$$

Then, we independently compute the QR factorization of each block row:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{pmatrix}.$$

This is "stage 0" of the computation, hence the second subscript 0 of the Q and R factors. The first subscript indicates the block index at that stage.

[1] The ScaLAPACK Users' Guide has a good explanation of 1-D and 2-D block and block cyclic layouts of dense matrices [5]. In particular, refer to the section entitled "Details of Example Program #1."

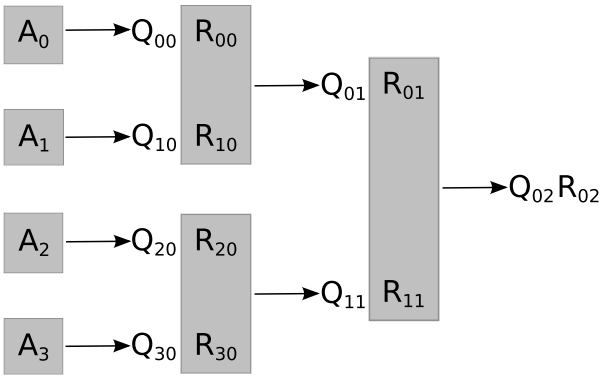


Figure 1: Execution of the parallel TSQR factorization on a binary tree of four processors. The gray boxes indicate where local QR factorizations take place. The Q and R factors each have two subscripts: the first is the sequence number within that stage, and the second is the stage number.

(Abstractly, we use the Fortran convention that the first index changes "more frequently" than the second index.) Stage 0 operates on the $P = 4$ leaves of the tree. We can write this decomposition instead as a block diagonal orthogonal matrix times a column of blocks:

$$A = \begin{pmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \cdot \left(\begin{array}{c} R_{00} \\ \hline R_{10} \\ \hline R_{20} \\ \hline R_{30} \end{array}\right),$$

although we do not have to store it this way. After this stage 0, there are $P = 4$ of the R factors. We group them into successive pairs $R_{i,0}$ and $R_{i+1,0}$, and do the QR factorizations of grouped pairs in parallel:

$$\left(\begin{array}{c} R_{00} \\ R_{10} \\ \hline R_{20} \\ R_{30} \end{array}\right) = \left(\begin{array}{c} \begin{pmatrix} R_{00} \\ R_{10} \end{pmatrix} \\ \hline \begin{pmatrix} R_{20} \\ R_{30} \end{pmatrix} \end{array}\right) = \left(\begin{array}{c} Q_{01}R_{01} \\ \hline Q_{11}R_{11} \end{array}\right).$$

As before, we can rewrite the last term as a block diagonal orthogonal matrix times a column of blocks:

$$\left(\begin{array}{c} Q_{01}R_{01} \\ \hline Q_{11}R_{11} \end{array}\right) = \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{11} \end{array}\right) \cdot \left(\begin{array}{c} R_{01} \\ \hline R_{11} \end{array}\right).$$

This is stage 1, as the second subscript of the Q and R factors indicates. We iteratively perform stages until there is only one R factor left, which is the root of the tree:

$$\begin{pmatrix} R_{01} \\ R_{11} \end{pmatrix} = Q_{02}R_{02}.$$

Equation (1) shows the whole factorization:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & Q_{10} & & \\ \hline & & Q_{20} & \\ \hline & & & Q_{30} \end{array}\right) \cdot \left(\begin{array}{c|c} Q_{01} & \\ \hline & Q_{11} \end{array}\right) \cdot Q_{02} \cdot R_{02}, \quad (1)$$

in which the product of the first three matrices has orthogonal columns, since each of these three matrices does. Note the binary tree structure in the nested pairs of R factors.

Figure 1 illustrates the binary tree on which the above factorization executes. Gray boxes highlight where local QR factorizations take place. By "local," we refer to a factorization performed by any one processor at one node of the tree; it may involve one or more than one block row. If we were to compute all the above Q factors explicitly as square matrices, each of the Q_{i0} would be $m/P \times m/P$, and Q_{ij} for $j > 0$ would be $2n \times 2n$. The final R factor would be upper triangular and $m \times n$, with $m - n$ rows of zeros. In a "thin QR" factorization, in which the final Q factor has the same dimensions as A, the final R factor would be upper triangular and $n \times n$. In practice, we prefer to store all the local Q factors implicitly until the factorization is complete. In that case, the implicit representation of Q_{i0} fits in an $m/P \times n$ lower triangular matrix, and the implicit representation of Q_{ij} (for $j > 0$) fits in an $n \times n$ lower triangular matrix (due to optimizations that will be discussed in Section 6).

Note that the maximum per-processor memory requirement is $\max\{mn/P, n^2 + O(n)\}$, since any one processor need only factor two $n \times n$ upper triangular matrices at once, or a single $m/P \times n$ matrix.

4.2 Sequential TSQR on a flat tree

Sequential TSQR uses a similar factorization process, but with a "flat tree" (a linear chain). It may also handle the leaf nodes of the tree slightly differently, as we will show below. Again, the basic idea is not new (see e.g., [40, 23]), but we will show how it fits into the same general framework as the parallel QR decomposition illustrated above, and also how this generalization expands the tuning space of QR factorization algorithms.

We start with the same block row decomposition as with parallel TSQR above:

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}$$

but begin with a QR factorization of A_0, rather than of all the block rows:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \begin{pmatrix} Q_{00}R_{00} \\ A_1 \\ A_2 \\ A_3 \end{pmatrix}.$$

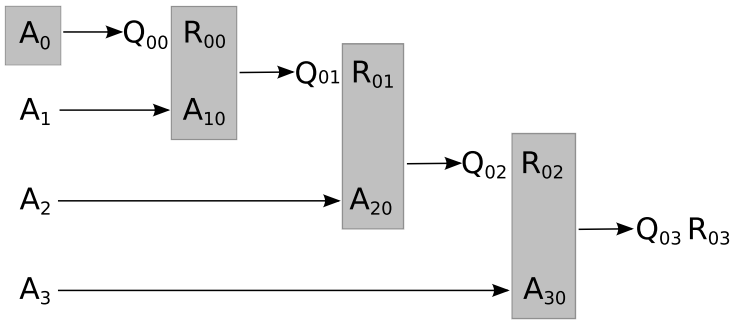


Figure 2: Execution of the sequential TSQR factorization on a flat tree with four submatrices. The gray boxes indicate where local QR factorizations take place The Q and R factors each have two subscripts: the first is the sequence number for that stage, and the second is the stage number.

This is "stage 0" of the computation, hence the second subscript 0 of the Q and R factor. We retain the first subscript for generality, though in this example it is always zero. We can write this decomposition instead as a block diagonal matrix times a column of blocks:

$$\begin{pmatrix} Q_{00}R_{00} \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & I & & \\ \hline & & I & \\ \hline & & & I \end{array}\right) \cdot \left(\begin{array}{c} R_{00} \\ \hline A_1 \\ \hline A_2 \\ \hline A_3 \end{array}\right).$$

We then combine R_{00} and A_1 using a QR factorization:

$$\begin{pmatrix} R_{00} \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c} R_{00} \\ \hline A_1 \\ \hline A_2 \\ \hline A_3 \end{array}\right) = \left(\begin{array}{c} Q_{01}R_{01} \\ \hline A_2 \\ A_3 \end{array}\right)$$

This can be rewritten as a block diagonal matrix times a column of blocks:

$$\left(\begin{array}{c} Q_{01}R_{01} \\ \hline A_2 \\ A_3 \end{array}\right) = \left(\begin{array}{c|c|c} Q_{01} & & \\ \hline & I & \\ \hline & & I \end{array}\right) \cdot \left(\begin{array}{c} R_{01} \\ \hline A_2 \\ \hline A_3 \end{array}\right).$$

We continue this process until we run out of A_i factors. The resulting factorization has the following structure:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = \left(\begin{array}{c|c|c|c} Q_{00} & & & \\ \hline & I & & \\ \hline & & I & \\ \hline & & & I \end{array}\right) \cdot \left(\begin{array}{c|c|c} Q_{01} & & \\ \hline & I & \\ \hline & & I \end{array}\right) \cdot \left(\begin{array}{c|c|c} I & & \\ \hline & Q_{02} & \\ \hline & & I \end{array}\right) \cdot \left(\begin{array}{c|c|c} I & & \\ \hline & I & \\ \hline & & Q_{03} \end{array}\right) R_{30}. \tag{2}$$

Here, the A_i blocks are $m/P \times n$. If we were to compute all the above Q factors explicitly as square matrices, then Q_{00} would be $m/P \times m/P$ and Q_{0j} for $j > 0$ would be $2m/P \times 2m/P$. The above I factors would be $m/P \times m/P$. The final R factor, as in the parallel case, would be upper triangular and $m \times n$, with $m - n$ rows of zeros. In a "thin QR" factorization, in which the final Q factor has the same dimensions as A, the final R factor would be upper triangular and $n \times n$. In practice, we prefer to store all the local Q factors implicitly until the factorization is complete. In that case, the implicit representation of Q_{00} fits in an $m/P \times n$ lower triangular matrix, and the implicit representation of Q_{0j} (for $j > 0$) fits in an $m/P \times n$ lower triangular matrix as well (due to optimizations that will be discussed in Section 6).

Figure 2 illustrates the flat tree on which the above factorization executes. Gray boxes highlight where "local" QR factorizations take place.

The sequential algorithm differs from the parallel one in that it does not factor the individual blocks of the input matrix A, excepting A_0. This is because in the sequential case, the input matrix has not yet been loaded into working memory. In the fully parallel case, each block of A resides in some processor's working memory. It then pays to factor all the blocks before combining them, as this reduces the volume of communication (only the triangular R factors need to be exchanged) and reduces the amount of arithmetic performed at the next level of the tree. In contrast, the sequential algorithm never writes out the intermediate R factors, so it does not need to convert the individual A_i into upper triangular factors. Factoring each A_i separately would require writing out an additional Q factor for each block of A. It would also add another level to the tree, corresponding to the first block A_0.

Note that the maximum per-processor memory requirement is $mn/P + n^2/2 + O(n)$, since only an $m/P \times n$ block and an $n \times n$ upper triangular block reside in fast memory at one time. We could save some fast memory by factoring each A_i block separately before combining it with the next block's R factor, as long as each block's Q and R factors are written back to slow memory before the next block is loaded. One would then only need to fit no more than two $n \times n$ upper triangular factors in fast memory at once. However, this would result in more writes, as each R factor (except the last) would need to be written to slow memory and read back into fact memory, rather than just left in fast memory for the next step.

In both the parallel and sequential algorithms, a vector or matrix is multiplied by Q or Q^T by using the implicit representation of the Q factor, as shown in Equation (1) for the parallel case, and Equation (2) for the sequential case. This is analogous to using the Householder vectors computed by Householder QR as an implicit representation of the Q factor.

4.3 TSQR on general trees

The above two algorithms are extreme points in a large set of possible QR factorization methods, parametrized by the tree structure. Our version of TSQR is novel because it works on any tree. In general, the optimal tree may depend

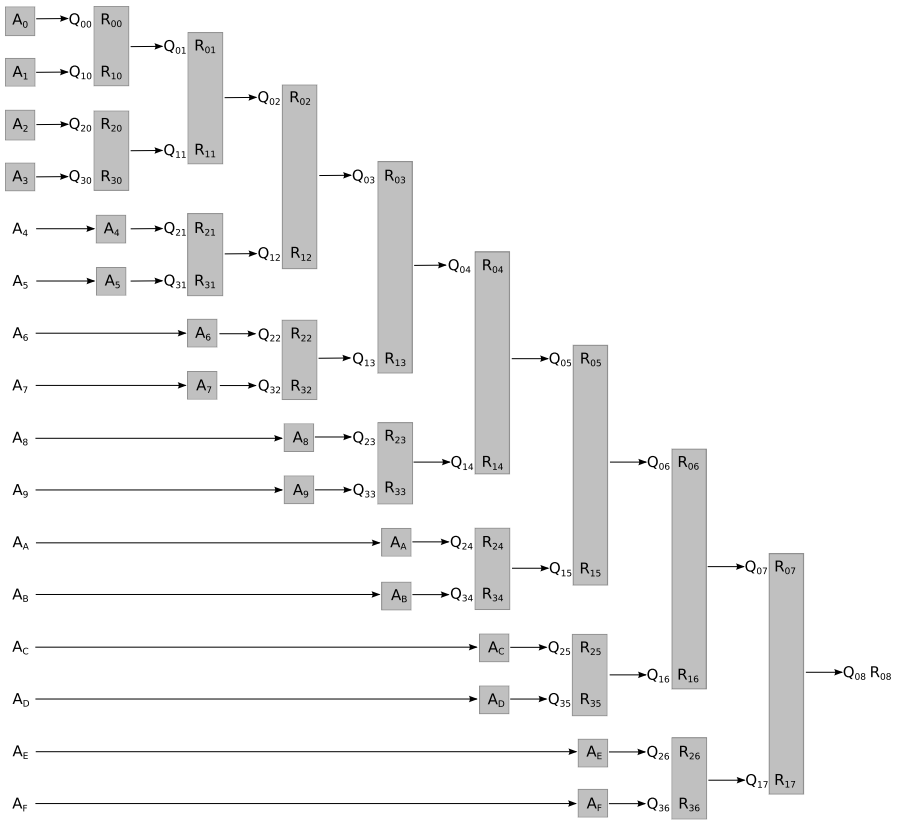


Figure 3: Execution of a hybrid parallel / out-of-core TSQR factorization. The matrix has 16 blocks, and four processors can execute local QR factorizations simultaneously. The gray boxes indicate where local QR factorizations take place. We number the blocks of the input matrix A in hexadecimal to save space (which means that the subscript letter A is the number 10_{10}, but the non-subscript letter A is a matrix block). The Q and R factors each have two subscripts: the first is the sequence number for that stage, and the second is the stage number.

on both the architecture and the matrix dimensions. This is because TSQR is a reduction (as we will discuss further in Section 5). Trees of types other than binary often result in better reduction performance, depending on the architecture (see e.g., [35]). Throughout this paper, we discuss two examples – the binary tree and the flat tree – as easy extremes for illustration. We will show that the binary tree minimizes the number of stages and messages in the parallel case, and that the flat tree minimizes the number and volume of input matrix reads and writes in the sequential case. Section 5 shows how to perform TSQR on any tree. Methods for finding the best tree in the case of TSQR are future work. Nevertheless, we can identify two regimes in which a "nonstandard" tree could improve performance significantly: parallel memory-limited CPUs, and large distributed-memory supercomputers.

The advent of desktop and even laptop multicore processors suggests a revival of parallel out-of-DRAM algorithms, for solving cluster-sized problems while saving power and avoiding the hassle of debugging on a cluster. TSQR could execute efficiently on a parallel memory-limited device if a sequential flat tree were used to bring blocks into memory, and a parallel tree (with a structure that reflects the multicore memory hierarchy) were used to factor the blocks. Figure 3 shows an example with 16 blocks executing on four processors, in which the factorizations are pipelined for maximum utilization of the processors. The algorithm itself needs no modification, since the tree structure itself encodes the pipelining. This is, we believe, a novel extension of the parallel out-of-core QR factorization of Gunter et al. [23], as their method uses ScaLAPACK's existing parallel QR (PDGEQRF) to factor the blocks in memory.

TSQR's choice of tree shape can also be optimized for modern supercomputers. A tree with different branching factors at different levels could naturally accommodate the heterogeneous communication network of a cluster of multicores. The subtrees at the lowest level may have the same branching factor as the number of cores per node (or per socket, for a multisocket shared-memory architecture).

Note that the maximum per-processor memory requirement of all TSQR variations is bounded above by

$$\frac{qn(n+1)}{2} + \frac{mn}{P},$$

in which q is the maximum branching factor in the tree.

5 TSQR as a reduction

Section 4 explained the algebra of the TSQR factorization. It outlined how to reorganize the parallel QR factorization as a tree-structured computation, in which groups of neighboring processors combine their R factors, perform (possibly redundant) QR factorizations, and continue the process by communicating their R factors to the next set of neighbors. Sequential TSQR works in a similar way, except that communication consists of moving matrix factors between slow

and fast memory. This tree structure uses the same pattern of communication found in a reduction or all-reduction. Thus, effective optimization of TSQR requires understanding these operations.

5.1 Reductions and all-reductions

Reductions and all-reductions are operations that take a collection as input, and combine the collection using some (ideally) associative function into a single item. The result is a function of all the items in the input. Usually, one speaks of (all-) reductions in the parallel case, where ownership of the input collection is distributed across some number P of processors. A reduction leaves the final result on exactly one of the P processors; an all-reduction leaves a copy of the final result on all the processors. See, for example, [22].

In the sequential case, there is an analogous operation. Imagine that there are P "virtual processors." To each one is assigned a certain amount of fast memory. Virtual processors communicate by sending messages via slow memory, just as the "real processors" in the parallel case communicate via the (relatively slow) network. Each virtual processor owns a particular subset of the input data, just as each real processor does in a parallel implementation. A virtual processor can read any other virtual processor's subset by reading from slow memory (this is a "receive"). It can also write some data to slow memory (a "send"), for another virtual processor to read. We can run programs for this virtual parallel machine on an actual machine with only one processor and its associated fast memory by scheduling the virtual processors' tasks on the real processor(s) in a way that respects task dependencies. Note that all-reductions and reductions produce the same result when there is only one actual processor, because if the final result ends up in fast memory on any of the virtual processors, it is also in fast memory on the one actual processor.

The "virtual processors" argument may also have practical use when implementing (all-) reductions on clusters of SMPs or vector processors, multicore out-of-core, or some other combination consisting of tightly-coupled parallel units with slow communication links between the units. A good mapping of virtual processors to real processors, along with the right scheduling of the "virtual" algorithm on the real machine, can exploit multiple levels of parallelism and the memory hierarchy.

5.2 (All-) reduction trees

Reductions and all-reductions are performed on directed trees. In a reduction, each node represents a processor, and each edge a message passed from one processor to another. All-reductions have two different implementation strategies:

- "Reduce-broadcast": Perform a standard reduction to one processor, followed by a broadcast (a reduction run backwards) of the result to all processors.

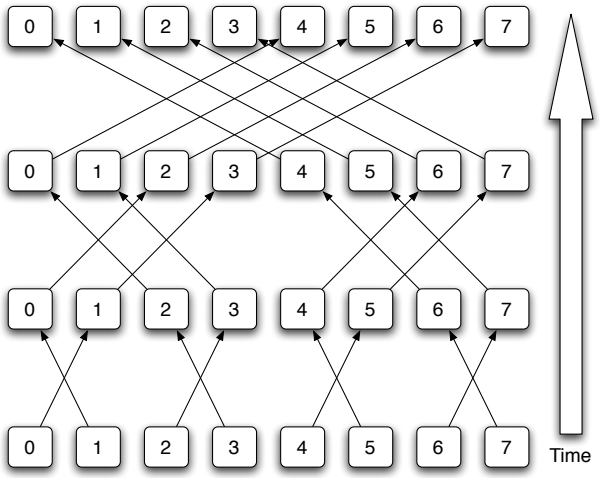


Figure 4: Diagram of a parallel butterfly all-reduction on a binary tree of eight processors. Each arrow represents a message from one processor to another. Time moves upwards.

- "Butterfly" method, with a communication pattern like that of a fast Fourier transform.

The butterfly method uses a tree with the following recursive structure:

- Each leaf node corresponds to a single processor.
- Each interior node is an ordered tuple whose members are the node's children.
- Each edge from a child to a parent represents a complete exchange of information between all individual processors at the same positions in the sibling tuples.

We call the processors that communicate at a particular stage *neighbors*. For example, in a a binary tree with eight processors numbered 0 to 7, processors 0 and 1 are neighbors at the first stage, processors 0 and 2 are neighbors at the second stage, and processors 0 and 4 are neighbors at the third (and final) stage. At any stage, each neighbor sends its current reduction value to all the other neighbors. The neighbors combine the values redundantly, and the all-reduction continues. Figure 4 illustrates this process. The butterfly all-reduction can be extended to any number of processors, not just powers of two.

The reduce-broadcast implementation requires about twice as many stages as the butterfly pattern (in the case of a binary tree) and thus as much as twice the latency. However, it reduces the total number of messages communicated per level of the tree (not just the messages on the critical path). In the

case of a binary tree, reduce-broadcast requires at most $P/2$ messages at any one level, and $P \log(P)/2$ total messages. A butterfly always generates P messages at every level, and requires $P \log(P)$ total messages. The choice between reduce-broadcast and butterfly depends on the properties of the communication network.

5.3 TSQR-specific (all-) reduction requirements

TSQR uses an (all-) reduction communication pattern, but has requirements that differ from the standard (all-) reduction. For example, if the Q factor is desired, then TSQR must store intermediate results (the local Q factor from each level's computation with neighbors) at interior nodes of the tree. This requires reifying and preserving the (all-) reduction tree for later invocation by users. Typical (all-) reduction interfaces, such as those provided by MPI or OpenMP, do not allow this (see e.g., [22]). They may not even guarantee that the same tree will be used upon successive invocations of the same (all-) reduction operation, or that the inputs to a node of the (all-) reduction tree will always be in the same order.

6 Optimizations for local QR factorizations

Although TSQR achieves its performance gains because it optimizes communication, the local QR factorizations lie along the critical path of the algorithm. The parallel cluster benchmark results in Section 12 show that optimizing the local QR factorizations can improve performance significantly. In this section, we outline a few of these optimizations, and hint at how they affect the formulation of the general CAQR algorithm in Section 13.

6.1 Structured QR factorizations

Many of the inputs to the local QR factorizations have a particular structure. In the parallel case, they are vertical stacks of $n \times n$ upper triangular matrices, and in the sequential case, at least one of the blocks is upper triangular. In this section, we show how to modify a standard dense Householder QR factorization in order to exploit this structure. This can save a factor of $5\times$ flops and (at least) $3\times$ storage, in the parallel case, and a factor of $2\times$ flops and (up to) $2\times$ storage in the sequential case. We also show how to perform the trailing matrix update with these *structured QR factorizations*, as it will be useful for Section 13.

Suppose that we have two upper triangular matrices R_0 and R_1, each of size 5×5. (The notation here is generic and not meant to correspond to a specific stage of TSQR. This is extended easily enough to the case of q upper triangular matrices, for $q = 2, 3, \ldots$.) Then, we can write their vertical concatenation as follows, in which an x denotes a structural nonzero of the matrix, and empty

spaces denote zeros:

$$\begin{pmatrix} R_0 \\ R_1 \end{pmatrix} = \begin{pmatrix} x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \\ x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \end{pmatrix}. \tag{3}$$

Note that we do not need to store the ones on the diagonal explicitly. The Q factor effectively overwrites R_1 and the R factor overwrites R_0.

The approach used for performing the QR factorization of the first block column affects the storage for the Householder vectors as well as the update of any trailing matrices that may exist. In general, Householder transformations have the form $I - \tau_j v_j v_j^T$, in which the Householder vector v_i is normalized so that $v_i(1) = 1$. This means that $v_i(1)$ need not be stored explicitly. Furthermore, if we use *structured Householder transformations*, we can avoid storing and computing with the zeros in Equation (3). As the Householder vector always has the same nonzero pattern as the vector from which it is calculated, the nonzero structure of the Householder vector is trivial to determine.

For a $2n \times n$ rectangular matrix composed of $n \times n$ upper triangular matrices, the i-th Householder vector v_i in the QR factorization of the matrix is a vector of length $2n$ with nonzeros in entries $n+1$ through $n+i$, a one in entry i, and zeros elsewhere. If we stack all n Householder vectors into a $2n \times n$ matrix, we obtain the following representation of the Q factor (not including the τ array of multipliers):

$$\begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & 1 & & \\ & & & 1 & \\ & & & & 1 \\ x & x & x & x & x \\ & x & x & x & x \\ & & x & x & x \\ & & & x & x \\ & & & & x \end{pmatrix}. \tag{4}$$

Algorithm 1 shows a standard, column-by-column sequential QR factorization of the $qn \times n$ matrix of upper triangular $n \times n$ blocks, using structured Householder reflectors. To analyze the cost, consider the components:

1. House(w): the cost of this is dominated by finding the norm of the vector w and scaling it.

Algorithm 1 Sequential QR factorization of $qn \times n$ matrix A, with structure as in Equation (3)

1: **for** $j = 1$ to n **do**
2: Let $\mathcal{I}_j$ be the index set $\{j, n+1 : n+j, \ldots, (q-1)n+1 : (q-1)n+j\}$
3: $w := A(\mathcal{I}_j, j)$ ▷ Gather pivot column of A into w
4: $[\tau_j, v] := \text{House}(w)$ ▷ Compute Householder reflection, normalized so that $v(1) = 1$
5: $X := A(\mathcal{I}_j, j+1 : n)$ ▷ Gather from A into X. One would normally perform the update in place; we use a copy to improve clarity.
6: $X := (I - \tau_j v v^T)X$ ▷ Apply Householder reflection
7: $A(\mathcal{I}_j \setminus \{j\}, j) := v(2 : end)$ ▷ Scatter $v(2 : end)$ back into A
8: $A(\mathcal{I}_j, j+1 : n) := X$ ▷ Scatter X back into A
9: **end for**

2. Applying a length n Householder reflector, whose vector contains k nonzeros, to an $n \times b$ matrix A. This is an operation $(I - \tau v v^T)A = A - v(\tau(v^T A))$.

Appendix A counts the arithmetic operations in detail. There, we find that the total cost is about

$$\frac{2}{3}(q-1)n^3$$

flops, to factor a $qn \times n$ matrix (we showed the specific case $q = 2$ above). The flop count increases by about a factor of $3\times$ if we ignore the structure of the inputs.

6.2 BLAS 3 structured Householder QR

Representing the local Q factor as a collection of Householder transforms means that the local QR factorization is dominated by BLAS 2 operations (dense matrix-vector products). A number of authors have shown how to reformulate the standard Householder QR factorization so as to coalesce multiple Householder reflectors into a block, so that the factorization is dominated by BLAS 3 operations. For example, Schreiber and Van Loan describe a so-called YT representation of a collection of Householder reflectors [42]. BLAS 3 transformations like this are now standard in LAPACK and ScaLAPACK.

We can adapt these techniques in a straightforward way in order to exploit the structured Householder vectors depicted in Equation (4). Schreiber and Van Loan use a slightly different definition of Householder reflectors: $\rho_j = I - 2v_j v_j^T$, rather than LAPACK's $\rho_j = I - \tau_j v_j v_j^T$. Schreiber and Van Loan's Y matrix is the matrix of Householder vectors $Y = [v_1\, v_2\, \ldots\, v_n]$; its construction requires no additional computation as compared with the usual approach. However, the T matrix must be computed, which increases the flop count by a constant factor. The cost of computing the T factor for the $qn \times n$ factorization above is about $qn^3/3$. Algorithm 2 shows the resulting computation. Note that the T factor

Algorithm 2 Computing Y and T in the (Y, T) representation of a collection of n Householder reflectors. Modification of an algorithm in [42] so that $P_j = I - \tau_j v_j v_j^T$.

Require: n Householder reflectors $\rho_j = I - \tau_j v_j v_j^T$
1: **for** $j = 1$ to n **do**
2: **if** $j = 1$ **then**
3: $Y := [v_1]$
4: $T := [-\tau_j]$
5: **else**
6: $z := -\tau_j (T(Y^T v_j))$
7: $Y := \begin{pmatrix} Y & v_j \end{pmatrix}$
8: $T := \begin{pmatrix} T & z \\ 0 & -\tau_j \end{pmatrix}$
9: **end if**
10: **end for**
Assert: Y and T satisfy $\rho_1 \cdot \rho_2 \cdot \ldots \rho_n = I + YTY^T$

requires $n(n-1)/2$ additional storage per processor on which the T factor is required.

6.3 Recursive Householder QR

In Section 12, we show large performance gains obtained by using Elmroth and Gustavson's recursive algorithm for the local QR factorizations [16]. The authors themselves observed that their approach works especially well with "tall thin" matrices, and others have exploited this effect in their applications (see e.g., [40]). The recursive approach outperforms LAPACK because it makes the panel factorization a BLAS 3 operation. In LAPACK, the panel QR factorization consists only of matrix-vector and vector-vector operations. This suggests why recursion helps especially well with tall, thin matrices. Elmroth and Gustavson's basic recursive QR does not perform well when n is large, as the flop count grows cubically in n, so they opt for a hybrid approach that divides the matrix into panels of columns, and performs the panel QR factorizations using the recursive method.

Elmroth and Gustavson use exactly the same representation of the Q factor as Schreiber and Van Loan [42], so the arguments of the previous section still apply.

6.4 Trailing matrix update

Section 13 will describe how to use TSQR to factor matrices in general 2-D layouts. For these layouts, once the current panel (block column) has been factored, the panels to the right of the current panel cannot be factored until the transpose of the current panel's Q factor has been applied to them. This is called a *trailing matrix update*. The update lies along the critical path of

the algorithm, and consumes most of the floating-point operations in general. This holds regardless of whether the factorization is left-looking, right-looking, or some hybrid of the two.[2] Thus, it's important to make the updates efficient.

The trailing matrix update consists of a sequence of applications of local Q^T factors to groups of "neighboring" trailing matrix blocks. (Section 5 explains the meaning of the word "neighbor" here.) We now explain how to do one of these local Q^T applications. (Do not confuse the local Q factor, which we label generically as Q, with the entire input matrix's Q factor.)

Let the number of rows in a block be M, and the number of columns in a block be N. We assume $M \geq N$. Suppose that we want to apply the local Q^T factor from the above $qN \times N$ matrix factorization, to two blocks C_0 and C_1 of a trailing matrix panel. (This is the case $q = 2$, which we assume for simplicity.) We divide each of the C_i into a top part and a bottom part:

$$C_i = \begin{pmatrix} C_i(1:N,:) \\ C_i(N+1:M,:) \end{pmatrix} = \begin{pmatrix} C_i' \\ C_i'' \end{pmatrix}.$$

Our goal is to perform the operation

$$\begin{pmatrix} R_0 & C_0' \\ R_1 & C_1' \end{pmatrix} = \begin{pmatrix} QR & C_0' \\ & C_1' \end{pmatrix} = Q \cdot \begin{pmatrix} R & \hat{C}_0' \\ & \hat{C}_1' \end{pmatrix},$$

in which Q is the local Q factor and R is the local R factor of $[R_0; R_1]$. Implicitly, the local Q factor has the dimensions $2M \times 2M$, as Section 4 explains. However, it is not stored explicitly, and the implicit operator that is stored has the dimensions $2N \times 2N$. We assume that processors P_0 and P_1 each store a redundant copy of Q, that processor P_2 has C_0, and that processor P_3 has C_1. We want to apply Q^T to the matrix

$$C = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix}.$$

First, note that Q has a specific structure. If stored explicitly, it would have the form

$$Q = \begin{pmatrix} U_{00} & & U_{01} & \\ & I_{M-N} & & \mathbf{0}_{M-N} \\ U_{10} & & U_{11} & \\ & \mathbf{0}_{M-N} & & I_{M-N} \end{pmatrix},$$

in which the U_{ij} blocks are each $N \times N$. This makes the only nontrivial computation when applying Q^T the following:

$$\begin{pmatrix} \hat{C}_0' \\ \hat{C}_1' \end{pmatrix} := \begin{pmatrix} U_{00}^T & U_{10}^T \\ U_{01}^T & U_{11}^T \end{pmatrix} \cdot \begin{pmatrix} C_0' \\ C_1' \end{pmatrix}. \tag{5}$$

We see, in particular, that only the uppermost N rows of each block of the trailing matrix need to be read or written. Note that it is not necessary to

[2] For descriptions and illustrations of the difference between left-looking and right-looking factorizations, see e.g., [14].

construct the U_{ij} factors explicitly; we need only operate on C'_0 and C'_1 with Q^T.

If we are using a standard Householder QR factorization (without BLAS 3 optimizations), then computing Equation (5) is straightforward. When one wishes to exploit structure (as in Section 6.1) and use a local QR factorization that exploits BLAS 3 operations (as in Section 6.2), more interesting load balance issues arise. We will discuss these in the following section.

6.4.1 Trailing matrix update with structured BLAS 3 QR

An interesting attribute of the YT representation is that the T factor can be constructed using only the Y factor and the τ multipliers. This means that it is unnecessary to send the T factor for updating the trailing matrix; the receiving processors can each compute it themselves. However, one cannot compute Y from T and τ in general.

When the YT representation is used, the update of the trailing matrices takes the following form:

$$\begin{pmatrix} \hat{C_0}' \\ \hat{C_1}' \end{pmatrix} := \left(I - \begin{pmatrix} I \\ Y_1 \end{pmatrix} \cdot \quad T^T \quad \cdot \begin{pmatrix} I \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C'_0 \\ C'_1 \end{pmatrix}.$$

Here, Y_1 starts on processor P_1, C'_0 on processor P_2, and C'_1 on processor P_3. The matrix T must be computed from τ and Y_1; we can assume that τ is on processor P_1. The updated matrices $\hat{C_0}'$ and $\hat{C_1}'$ are on processors P_2 resp. P_3.

There are many different ways to perform this parallel update. The data dependencies impose a directed acyclic graph (DAG) on the flow of data between processors. One can find the the best way to do the update by realizing an optimal computation schedule on the DAG. Our performance models can be used to estimate the cost of a particular schedule.

Here is a straightforward but possibly suboptimal schedule. First, assume that Y_1 and τ have already been sent to P_3. Then,

P_2's tasks:

- Send C'_0 to P_3
- Receive W from P_3
- Compute $\hat{C_0}' = C'_0 - W$

P_3's tasks:

- Compute the T factor and $W := T^T(C'_0 + Y_1^T C'_1)$
- Send W to P_2
- Compute $\hat{C_1}' := C'_1 - Y_1 W$

However, this leads to some load imbalance, since P_3 performs more computation than P_2. It does not help to compute T on P_0 or P_1 before sending it to P_3, because the computation of T lies on the critical path in any case. We will see in Section 13 that part of this computation can be overlapped with the communication.

For $q \geq 2$, we can write the update operation as

$$\begin{pmatrix} \hat{C_0}' \\ \hat{C_1}' \\ \vdots \\ \hat{C_{q-1}}' \end{pmatrix} := \left(I - \begin{pmatrix} I_{N \times N} \\ Y_1 \\ \vdots \\ Y_{q-1} \end{pmatrix} T^T \begin{pmatrix} I_{N \times N} & Y_1^T & \cdots & Y_{q-1}^T \end{pmatrix} \right) \begin{pmatrix} C_0' \\ C_1' \\ \vdots \\ C_{q-1}' \end{pmatrix}.$$

If we let

$$D := C_0' + Y_1^T C_1' + Y_2^T C_2' + \cdots + Y_{q-1}^T C_{q-1}'$$

be the "inner product" part of the update operation formulas, then we can rewrite the update formulas as

$$\begin{aligned} \hat{C_0}' &:= C_0' - T^T D, \\ \hat{C_1}' &:= C_1' - Y_1 T^T D, \\ &\vdots \\ \hat{C_{q-1}}' &:= C_{q-1}' - Y_{q-1} T^T D. \end{aligned}$$

As the branching factor q gets larger, the load imbalance becomes less of an issue. The inner product D should be computed as an all-reduce in which the processor owning C_i receives Y_i and T. Thus, all the processors but one will have the same computational load.

7 Machine model

7.1 Parallel machine model

Throughout this work, we use the "alpha-beta" or latency-bandwidth model of communication, in which a message of size n floating-point words takes time $\alpha + \beta n$ seconds. The α term represents message latency (seconds per message), and the β term inverse bandwidth (seconds per floating-point word communicated). Our algorithms only need to communicate floating-point words, all of the same size. We make no attempt to model overlap of communication and computation, but we do mention the possibility of overlap when it exists. Exploiting overlap could potentially speed up our algorithms (or any algorithm) by a factor of two.

We predict floating-point performance by counting floating-point operations and multiplying them by γ, the inverse peak floating-point performance, also known as the floating-point throughput. The quantity γ has units of seconds per flop (so it can be said to measure the bandwidth of the floating-point hardware). If we need to distinguish between adds and multiplies on one hand, and divides on the other, we use γ for the throughput of adds and multiplies, and γ_d for the throughput of divides.

When appropriate, we may scale the peak floating-point performance prediction of a particular matrix operation by a factor, in order to account for the

measured best floating-point performance of local QR factorizations. This generally gives the advantage to competing algorithms rather than our own, as our algorithms are designed to perform better when communication is much slower than arithmetic.

7.2 Sequential machine model

We also apply the alpha-beta model to communication between levels of the memory hierarchy in the sequential case. We restrict our model to describe only two levels at one time: fast memory (which is smaller) and slow memory (which is larger). The terms "fast" and "slow" are always relative. For example, DRAM may be considered fast if the slow memory is disk, but DRAM may be considered slow if the fast memory is cache. As in the parallel case, the time to complete a transfer between two levels is modeled as $\alpha + \beta n$. We assume that user has explicit control over data movement (reads and writes) between fast and slow memory. This offers an upper bound when control is implicit (as with caches), and also allows our model as well as our algorithms to extend to systems like the Cell processor (in which case fast memory is an individual local store, and slow memory is DRAM).

We assume that the fast memory can hold W floating-point words. For any QR factorization operating on an $m \times n$ matrix, the quantity

$$\frac{mn}{W}$$

bounds from below the number of loads from slow memory into fast memory (as the method must read each entry of the matrix at least once). It is also a lower bound on the number of stores from fast memory to slow memory (as we assume that the algorithm must write the computed Q and R factors back to slow memory). Sometimes we may refer to the block size P. In the case of TSQR, we usually choose

$$P = \frac{mn}{3W},$$

since at most three blocks of size P must be in fast memory at one time when applying the Q or Q^T factor in sequential TSQR (see Section 4).

In the sequential case, just as in the parallel case, we assume all memory transfers are nonoverlapped. Overlapping communication and computation may provide up to a twofold performance improvement. However, some implementations may consume fast memory space in order to do buffering correctly. This matters because the main goal of our sequential algorithms is to control fast memory usage, often to solve problems that do not fit in fast memory. We usually want to use as much of fast memory as possible, in order to avoid expensive transfers to and from slow memory.

8 TSQR implementation

In this section, we describe the TSQR factorization algorithm in detail. We also build a performance model of the algorithm, based on the machine model in Section 7 and the operation counts of the local QR factorizations in Section 6.

8.1 Reductions and all-reductions

In Section 5, we gave a detailed description of (all-)reductions. We did so because the TSQR factorization is itself an (all-)reduction, in which additional data (the components of the Q factor) is stored at each node of the (all-)reduction tree. Applying the Q or Q^T factor is also a(n) (all-)reduction.

If we implement TSQR with an all-reduction, then we get the final R factor replicated over all the processors. This is especially useful for Krylov subspace methods. If we implement TSQR with a reduction, then the final R factor is stored only on one processor. This avoids redundant computation, and is useful both for block column factorizations for 2-D block (cyclic) matrix layouts, and for solving least squares problems when the Q factor is not needed.

8.2 Factorization

We now describe the TSQR factorization for the 1-D block row layout. We omit the obvious generalization to a 1-D block cyclic row layout. Algorithm 3 shows an implementation of the former, based on an all-reduction. (Note that running Algorithm 3 on a matrix on a 1-D block cyclic layout still works, though it performs an implicit block row permutation on the Q factor.) The procedure computes an R factor which is duplicated over all the processors, and a Q factor which is stored implicitly in a distributed way. Line 1 makes use of a block representation of the Q factor: it is stored implicitly as a pair $(Y_{i,0}, \tau_{i,0})$, in which $Y_{i,0}$ is the unit lower trapezoidal matrix of Householder vectors and $\tau_{i,0}$ are the associated scaling factors. We overwrite the lower trapezoid of A_i with $Y_{i,0}$ (not counting the unit part). The matrix $R_{i,k}$ is stored as an $n \times n$ upper triangular matrix for all stages k.

8.2.1 Performance model

In Appendix D, we develop a performance model for parallel TSQR on a binary tree. Appendix B does the same for sequential TSQR on a flat tree.

A parallel TSQR factorization on a binary reduction tree performs the following computations along the critical path: One local QR factorization of a fully dense $m/P \times n$ matrix, and $\log P$ factorizations, each of a $2n \times n$ matrix consisting of two $n \times n$ upper triangular matrices. The factorization requires $\log P$ messages, each of size $n^2/2 + O(n)$.

Sequential TSQR on a flat tree requires $2mn/W$ messages between fast and slow memory, and moves a total of $2mn$ words. It performs about $2mn^2$ arithmetic operations.

Algorithm 3 TSQR, block row layout

Require: All-reduction tree with height $L = \log_2 P$

Require: $i \in \Pi$: my processor's index

Require: The $m \times n$ matrix A is distributed in a 1-D block row layout over the processors Π.

Require: A_i is the block of rows belonging to processor i.

1: Compute $[Q_{i,0}, R_{i,0}] := qr(A_i)$
2: **for** k from 1 to $\log_2 P$ **do**
3: **if** I have any neighbors at this level **then**
4: Send (non-blocking) $R_{i,k-1}$ to each neighbor not myself
5: Receive (non-blocking) $R_{j,k-1}$ from each neighbor j not myself
6: Wait until the above sends and receives complete ▷ Note: *not* a global barrier.
7: Stack the $R_{j,k-1}$ from all neighbors (including my own $R_{i,k-1}$), by order of processor ids, into an array C
8: Compute $[Q_{i,k}, R_{i,k}] := qr(C)$
9: **else**
10: $R_{i,k} := R_{i,k-1}$
11: $Q_{i,k} := I_{n \times n}$ ▷ Stored implicitly
12: **end if**
13: Processor i has an implicit representation of its block column of $Q_{i,k}$. The blocks in the block column are $n \times n$ each and there are as many of them as there are neighbors at stage k (including i itself). We don't need to compute the blocks explicitly here.
14: **end for**

Assert: $R_{i,L}$ is the R factor of A, for all processors $i \in \Pi$.

Assert: The Q factor is implicitly represented by $\{Q_{i,k}\}$: $i \in \Pi$, $k \in \{0, 1, \ldots, L\}\}$.

8.3 Applying Q or Q^T to vector(s)

Just like Householder QR, TSQR computes an implicit representation of the Q factor. One need not generate an explicit representation of Q in order to apply the Q or Q^T operators to one or more vectors. In fact, generating an explicit Q matrix requires just as many messages as applying Q or Q^T. (The performance model for applying Q or Q^T is an obvious extension of the factorization performance model, so we omit it.) Furthermore, the implicit representation can be updated or downdated, by using standard techniques (see e.g., [19]) on the local QR factorizations recursively. The s-step Krylov methods mentioned in Section 3 employ updating and downdating extensively.

In the case of the "thin" Q factor (in which the vector input is of length n), applying Q involves a kind of broadcast operation (which is the opposite of a reduction). If the "full" Q factor is desired, then applying Q or Q^T is a kind of all-to-all (like the fast Fourier transform). Computing $Q \cdot x$ runs through the nodes of the (all-)reduction tree from leaves to root, whereas computing $Q^T \cdot y$

runs from root to leaves.

9 Other "tall skinny" QR algorithms

There are many other algorithms besides TSQR for computing the QR factorization of a tall skinny matrix. They differ in terms of performance and accuracy, and may store the Q factor in different ways that favor certain applications over others. In this section, we model the performance of the following competitors to TSQR:

- Four different Gram-Schmidt variants
- CholeskyQR (see [45])
- Householder QR, with a block row layout

Each includes parallel and sequential versions. For Householder QR, we base our parallel model on the ScaLAPACK routine PDGEQRF, and the sequential model on the out-of-core ScaLAPACK routine PFDGEQRF (which we assume is running on just one processor). In the subsequent Section 10, we summarize the numerical accuracy of these QR factorization methods, and discuss their suitability for different applications.

In the parallel case, CholeskyQR and TSQR have comparable numbers of messages and communicate comparable numbers of words, but CholeskyQR requires a constant factor fewer flops along the critical path. However, the Q factor computed by TSQR is always numerically orthogonal, whereas the Q factor computed by CholeskyQR loses orthogonality proportionally to $\kappa_2(A)^2$. The variants of Gram-Schmidt require at best a factor n more messages than these two algorithms, and lose orthogonality at best proportionally to $\kappa_2(A)$.

9.1 Gram-Schmidt orthogonalization

Gram-Schmidt has two commonly used variations: "classical" (CGS) and "modified" (MGS). Both versions have the same floating-point operation count, but MGS performs them in a different order to improve stability. We will show that a parallel implementation of MGS uses at best $2n \log(P)$ messages, in which P is the number of processors, and a sequential implementation moves at best $m \cdot (mn/W)^2 + O(mn)$ words between fast and slow memory, in which W is the fast memory capacity. In contrast, parallel TSQR requires only $O(\log(P))$ messages, and sequential TSQR only moves $O(mn)$ words between fast and slow memory.

9.1.1 Left- and right-looking

Just like many matrix factorizations, both MGS and CGS come in left-looking and right-looking variants. To distinguish between the variants, we append "_L" resp. "_R" to the algorithm name to denote left- resp. right-looking. We show

all four combinations as Algorithms 4–7. Both right-looking and left-looking variants loop from left to right over the columns of the matrix A. At iteration k of this loop, the left-looking version only accesses columns 1 to k inclusive, whereas the right-looking version only accesses columns k to n inclusive. Thus, right-looking algorithms require the entire matrix to be available, which forbids their use when the matrix is to be generated and orthogonalized one column at a time. (In this case, only left-looking algorithms may be used.) We assume here that the entire matrix is available at the start of the algorithm.

Right-looking Gram-Schmidt is usually called "row-oriented Gram-Schmidt," and by analogy, left-looking Gram-Schmidt is usually called "column-oriented Gram-Schmidt." We use the terms "right-looking" resp. "left-looking" for consistency with the other QR factorization algorithms in this paper.

9.1.2 Parallel Gram-Schmidt

MGS_L (Algorithm 5) requires about $n/4$ times more messages than MGS_R (Algorithm 4), since the left-looking algorithm's data dependencies prevent the use of matrix-vector products. CGS_R (Algorithm 6) requires copying the entire input matrix; not doing so results in MGS_R (Algorithm 4), which is more numerically stable in any case. Thus, for the parallel case, we favor MGS_R and CGS_L for a fair comparison with TSQR.

In the parallel case, all four variants of MGS and CGS listed here require

$$\frac{2mn^2}{P} + O\left(\frac{mn}{P}\right)$$

arithmetic operations, and involve communicating

$$\frac{n^2}{2}\log(P) + O(n\log(P))$$

floating-point words in total. MGS_L requires

$$\frac{n^2}{2}\log(P) + O(n\log(P))$$

messages, whereas the other versions only need $2n\log(P)$ messages. Table 3 shows all four performance models.

9.1.3 Sequential Gram-Schmidt

For one-sided factorizations in the out-of-slow-memory regime, left-looking algorithms require fewer writes than their right-looking analogues, and about as many reads (see e.g., [46]). Thus, it only pays to look at left-looking Gram-Schmidt. The communication pattern of sequential modified and classical Gram-Schmidt is close enough to that of sequential Householder QR, that we can analyze the general pattern once. We do this in Appendix F. The optimizations described in that section apply to Gram-Schmidt. Table 4 shows performance models for the left-looking Gram-Schmidt variants in the sequential case.

Parallel algorithm	# flops	# messages	# words
Right-looking MGS	$2mn^2/P$	$2n \log(P)$	$\frac{n^2}{2} \log(P)$
Left-looking MGS	$2mn^2/P$	$\frac{n^2}{2} \log(P)$	$\frac{n^2}{2} \log(P)$
Right-looking CGS	$2mn^2/P$	$2n \log(P)$	$\frac{n^2}{2} \log(P)$
Left-looking CGS	$2mn^2/P$	$2n \log(P)$	$\frac{n^2}{2} \log(P)$

Table 3: Arithmetic operation counts, number of messages, and total communication volume (in number of words transferred) for parallel left-looking and right-looking variants of CGS and MGS. Reductions are performed using a binary tree on P processors. Lower-order terms omitted.

Sequential algorithm	# flops	# messages	# words
MGS_L	$2mn^2$	$\frac{1}{2}\left(\frac{mn}{W}\right)^2$	$2mn + \frac{m^3n^2}{W^2}$
CGS_L	$2mn^2$	$\frac{1}{2}\left(\frac{mn}{W}\right)^2$	$2mn + \frac{m^3n^2}{W^2}$

Table 4: Arithmetic operation counts, number of reads and writes, and total communication volume (in number of words read and written) for sequential left-looking CGS and MGS. W is the fast memory capacity in number of floating-point words. Lower-order terms omitted.

9.1.4 Reorthogonalization

One can improve the stability of CGS by reorthogonalizing the vectors. The simplest way is to make two orthogonalization passes per column, that is, to orthogonalize the current column against all the previous columns twice. We call this "CGS2." This method only makes sense for left-looking Gram-Schmidt, when there is a clear definition of "previous columns." Normally one would orthogonalize the column against all previous columns once, and then use some orthogonality criterion to decide whether to reorthogonalize the column. As a result, the performance of CGS2 is data-dependent, so we do not model its performance here. In the worst case, it can cost twice as much as CGS_L. Section 10 discusses the numerical stability of CGS2 and why "twice is enough."

Algorithm 4 Modified Gram-Schmidt, right-looking

Require: A: $m \times n$ matrix with $m \geq n$

1: **for** $k = 1$ to n **do**
2: $\quad R(k,k) := \|A(:,k)\|_2$
3: $\quad Q(:,k) := A(:,k)/R(k,k)$
4: $\quad R(k,k+1:n) := Q(:,k)^T \cdot A(:,k+1:n)$
5: $\quad A(:,k+1:n) := A(:,k+1:n) - R(k,k+1:n) \cdot Q(:,k)$
6: **end for**

Algorithm 5 Modified Gram-Schmidt, left-looking

Require: A: $m \times n$ matrix with $m \geq n$

1: **for** $k = 1$ to n **do**
2: $v := A(:, k)$
3: **for** $j = 1$ to $k - 1$ **do** ▷ Data dependencies hinder vectorization
4: $R(j, k) := Q(:, j)^T \cdot v$ ▷ Change v to $A(:, k)$ to get CGS
5: $v := v - R(j, k) \cdot Q(:, j)$
6: **end for**
7: $R(k, k) := \|v\|_2$
8: $Q(:, k) := v / R(k, k)$
9: **end for**

Algorithm 6 Classical Gram-Schmidt, right-looking

Require: A: $m \times n$ matrix with $m \geq n$

1: $V := A$ ▷ Not copying A would give us right-looking MGS.
2: **for** $k = 1$ to n **do**
3: $R(k, k) := \|V(:, k)\|_2$
4: $Q(:, k) := V(:, k) / R(k, k)$
5: $R(k, k + 1 : n) := Q(:, k)^T \cdot A(:, k + 1 : n)$
6: $V(:, k + 1 : n) := V(:, k + 1 : n) - R(k, k + 1 : n) \cdot Q(:, k)$
7: **end for**

9.2 CholeskyQR

CholeskyQR (Algorithm 8) is a QR factorization that requires only one reduction [45]. In the parallel case, it requires $\log_2(P)$ messages, where P is the number of processors. In the sequential case, it reads the input matrix only once. Thus, it is optimal in the same sense that TSQR is optimal. Furthermore, the reduction operator is matrix-matrix addition rather than a QR factorization of a matrix with comparable dimensions, so CholeskyQR should always be faster than TSQR. Section 12 supports this claim with performance data on a cluster. Note that in the sequential case, P is the number of blocks, and we assume conservatively that fast memory must hold $2mn/P$ words at once (so that $W = 2mn/P$).

Algorithm	# flops	# messages	# words
Parallel CholeskyQR	$\frac{2mn^2}{P} + \frac{n^3}{3}$	$\log(P)$	$\frac{n^2}{2}\log(P)$
Sequential CholeskyQR	$2mn^2 + \frac{n^3}{3}$	$\frac{6mn}{W}$	$3mn$

Table 5: Performance model of the parallel and sequential CholeskyQR factorization. We assume $W = 2mn/P$ in the sequential case, where P is the number of blocks and W is the number of floating-point words that fit in fast memory. Lower-order terms omitted. All parallel terms are counted along the critical path.

Algorithm 7 Classical Gram-Schmidt, left-looking

Require: A: $m \times n$ matrix with $m \geq n$

1: **for** $k = 1$ to n **do**
2: $R(1 : k-1, k) := Q(:, 1 : k-1)^T \cdot A(:, k)$ ▷ This and the next statement are not vectorized in left-looking MGS.
3: $A(:, k) := A(:, k) - R(1 : k-1, k) \cdot Q(:, 1 : k-1)$ ▷ In the sequential case, one can coalesce the read of each block of $A(:, k)$ in this statement with the read of each block of $A(:, k)$ in the next statement.
4: $R(k, k) := \|A(:, k)\|_2$
5: $Q(:, k) := A(:, k)/R(k, k)$
6: **end for**

Algorithm 8 CholeskyQR factorization

Require: A: $m \times n$ matrix with $m \geq n$

1: $W := A^T A$ ▷ (All-)reduction
2: Compute the Cholesky factorization $L \cdot L^T$ of W
3: $Q := AL^{-T}$

Assert: $[Q, L^T]$ is the QR factorization of A

CholeskyQR begins by computing $A^T A$. In the parallel case, each processor i computes its component $A_i^T A_i$ locally. In the sequential case, this happens one block at a time. Since this result is a symmetric $n \times n$ matrix, the operation takes only $n(n+1)m/P$ flops. These local components are then summed using a(n) (all-)reduction, which can also exploit symmetry. The final operation, the Cholesky factorization, requires $n^3/3 + O(n^2)$ flops. (Choosing a more stable or robust factorization does not improve the accuracy bound, as the accuracy has already been lost by computing $A^T A$.) Finally, the $Q := AL^{-T}$ operation costs $(1 + n(n-1))(m/P)$ flops per block of A. Table 5 summarizes both the parallel and sequential performance models. In Section 10, we compare the accuracy of CholeskyQR to that of TSQR and other "tall skinny" QR factorization algorithms.

9.3 Householder QR

Householder QR uses orthogonal reflectors to reduce a matrix to upper tridiagonal form, one column at a time (see e.g., [19]). In the current version of LAPACK and ScaLAPACK, the reflectors are coalesced into block columns (see e.g., [42]). This makes trailing matrix updates more efficient, but the panel factorization is still standard Householder QR, which works one column at a time. These panel factorizations are an asymptotic latency bottleneck in the parallel case, especially for tall and skinny matrices. Thus, we model parallel Householder QR without considering block updates. In contrast, we will see that operating on blocks of columns can offer asymptotic bandwidth savings in sequential Householder QR, so it pays to model a block column version.

Parallel algorithm	# flops	# messages	# words
TSQR	$\frac{2mn^2}{P} + \frac{2}{3}n^3 \log(P)$	$\log(P)$	$\frac{n^2}{2} \log(P)$
PDGEQRF	$\frac{2mn^2}{P} + \frac{n^2}{2} \log(P)$	$2n \log(P)$	$\frac{n^2}{2} \log(P)$
MGS_R	$\frac{2mn^2}{P}$	$2n \log(P)$	$\frac{n^2}{2} \log(P)$
CGS_L	$\frac{2mn^2}{P}$	$2n \log(P)$	$\frac{n^2}{2} \log(P)$
CGS2	$\frac{4mn^2}{P}$	$4n \log(P)$	$n^2 \log(P)$
CholeskyQR	$\frac{2mn^2}{P} + \frac{n^3}{3}$	$\log(P)$	$\frac{n^2}{2} \log(P)$

Table 6: Performance model of various parallel QR factorization algorithms. "CGS2" means CGS with one reorthogonalization pass. Lower-order terms omitted. All parallel terms are counted along the critical path.

9.3.1 Parallel Householder QR

ScaLAPACK's parallel QR factorization routine, PDGEQRF, uses a right-looking Householder QR approach [6]. The cost of PDGEQRF depends on how the original matrix A is distributed across the processors. For comparison with TSQR, we assume the same block row layout on P processors.

PDGEQRF computes an explicit representation of the R factor, and an implicit representation of the Q factor as a sequence of Householder reflectors. The algorithm overwrites the upper triangle of the input matrix with the R factor. Thus, in our case, the R factor is stored only on processor zero, as long as $m/P \geq n$. We assume $m/P \geq n$ in order to simplify the performance analysis.

Section 6.2 describes BLAS 3 optimizations for Householder QR. PDGEQRF exploits these techniques in general, as they accelerate the trailing matrix updates. We do not count floating-point operations for these optimizations here, since they do nothing to improve the latency bottleneck in the panel factorizations.

In PDGEQRF, some processors may need to perform fewer flops than other processors, because the number of rows in the current working column and the current trailing matrix of A decrease by one with each iteration. With the assumption that $m/P \geq n$, however, all but the first processor must do the same amount of work at each iteration. In the tall skinny regime, "flops on the critical path" (which is what we count) is a good approximation of "flops on each processor."

Table 6 compares the performance of all the parallel QR factorizations discussed here. We see that 1-D TSQR and CholeskyQR save both messages and bandwidth over MGS_R and ScaLAPACK's PDGEQRF, but at the expense of a higher-order n^3 flops term.

9.3.2 Sequential Householder QR

LAPACK Working Note #118 describes an out-of-DRAM QR factorization PFDGEQRF, which is implemented as an extension of ScaLAPACK [11]. It uses ScaLAPACK's parallel QR factorization PDGEQRF to perform the current

Sequential algorithm	# flops	# messages	# words
TSQR	$2mn^2$	$\frac{6mn}{W}$	$2mn + \frac{3mn^2}{W} - \frac{n^2}{2}$
PFDGEQRF	$2mn^2$	$\frac{1}{2}\left(\frac{mn}{W}\right)^2 + O(n)$	$2mn + \frac{m^3n^2}{W^2}$
CholeskyQR	$2mn^2 + \frac{3mn^2}{2W} + \frac{n^3}{3}$	$\frac{9mn}{W}$	$3mn$

Table 7: Performance model of various sequential QR factorization algorithms. PFDGEQRF is our model of ScaLAPACK's out-of-DRAM QR factorization; W is the fast memory size. Lower-order terms omitted.

panel factorization in DRAM. Thus, it is able to exploit parallelism. We assume here, though, that it is running sequentially, since we are only interested in modeling the traffic between slow and fast memory. PFDGEQRF is a left-looking method, as usual with out-of-DRAM algorithms. The code 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. Appendix G describes the method in more detail, and Algorithm 16 in the Appendix gives an outline of the code.

The PFDGEQRF algorithm performs

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

floating-point arithmetic operations, just like any sequential Householder QR factorization. It transfers a total of about

$$2mn + \frac{m^3n^2}{W^2}$$

floating-point words between slow and fast memory, and accesses slow memory (counting both reads and writes) about

$$\frac{1}{2}\left(\frac{mn}{W}\right)^2 + O(n)$$

times. In contrast, sequential TSQR only requires $6mn/W$ slow memory accesses and only transfers

$$2mn + \frac{3mn^2}{W} - \frac{n(n+1)}{2}$$

words between slow and fast memory.

Table 7 compares the performance of the sequential QR factorizations discussed in this section, including our modeled version of PFDGEQRF.

10 Numerical stability of TSQR and other QR factorizations

In the previous section, we modeled the performance of various QR factorization algorithms for tall and skinny matrices on a block row layout. Our models show

that CholeskyQR should have better performance than all the other methods. However, numerical accuracy is also an important consideration for many users. For example, in CholeskyQR, the loss of orthogonality of the computed Q factor depends quadratically on the condition number of the input matrix (see Table 8). This is because computing the Gram matrix $A^T A$ squares the condition number of A. One can avoid this stability loss by computing and storing $A^T A$ in doubled precision. However, this doubles the communication volume. It also increases the cost of arithmetic operations by a hardware-dependent factor.

Algorithm	$\|I - Q^T Q\|_2$ bound	Assumption on $\kappa(A)$	Reference(s)
Householder QR	$O(\varepsilon)$	None	[19]
TSQR	$O(\varepsilon)$	None	[19]
CGS2	$O(\varepsilon)$	$O(\varepsilon\kappa(A)) < 1$	[1, 27]
MGS	$O(\varepsilon\kappa(A))$	None	[4]
CholeskyQR	$O(\varepsilon\kappa(A)^2)$	None	[45]
CGS	$O(\varepsilon\kappa(A)^{n-1})$	None	[27, 43]

Table 8: Upper bounds on deviation from orthogonality of the Q factor from various QR algorithms. Machine precision is ε. "Assumption on $\kappa(A)$" refers to any constraints which $\kappa(A)$ must satisfy in order for the bound in the previous column to hold.

Unlike CholeskyQR, CGS, or MGS, Householder QR is *unconditionally stable*. That is, the computed Q factors are always orthogonal to machine precision, regardless of the properties of the input matrix [19]. This also holds for TSQR, because the algorithm is composed entirely of no more than P Householder QR factorizations, in which P is the number of input blocks. Each of these factorizations is itself unconditionally stable. In contrast, the orthogonality of the Q factor computed by CGS, MGS, or CholeskyQR depends on the condition number of the input matrix. Reorthogonalization in MGS and CGS can make the computed Q factor orthogonal to machine precision, but only if the input matrix A is numerically full rank, i.e., if $O(\varepsilon\kappa(A)) < 1$. Reorthogonalization also doubles the cost of the algorithm.

However, sometimes some loss of accuracy can be tolerated, either to improve performance, or for the algorithm to have a desirable property. For example, in some cases the input vectors are sufficiently well-conditioned to allow using CholeskyQR, and the accuracy of the orthogonalization is not so important. Another example is GMRES. Its backward stability was proven first for Householder QR orthogonalization, and only later for modified Gram-Schmidt orthogonalization [21]. Users traditionally prefer the latter formulation, mainly because the Householder QR version requires about twice as many floating-point operations (as the Q matrix must be computed explicitly). Another reason is that most GMRES descriptions make the vectors available for orthogonalization one at a time, rather than all at once, as Householder QR would require (see e.g., [48]). (Demmel et al. review existing techniques and present new methods for rearranging GMRES and other Krylov subspace methods for use with

Householder QR and TSQR [13].)

We care about stability for two reasons. First, an important application of TSQR is the orthogonalization of basis vectors in Krylov methods. When using Krylov methods to compute eigenvalues of large, ill-conditioned matrices, the whole solver can fail to converge or have a considerably slower convergence when the orthogonality of the Ritz vectors is poor [24, 29]. Second, we will use TSQR in Section 13 as the panel factorization in a QR decomposition algorithm for matrices of general shape. Users who ask for a QR factorization generally expect it to be numerically stable. This is because of their experience with Householder QR, which does more work than LU or Cholesky, but produces more accurate results. Users who are not willing to spend this additional work already favor faster but less stable algorithms.

Table 8 summarizes known upper bounds on the deviation from orthogonality $\|I - Q^T Q\|_2$ of the computed Q factor, as a function of the machine precision ε and the input matrix's two-norm condition number $\kappa(A)$, for various QR factorization algorithms. Except for CGS, all these bounds are sharp. Smoktunowicz et al. demonstrate a matrix satisfying $O(\varepsilon\kappa(A)^2) < 1$ for which $\|I - Q^T Q\|_2$ is not $O(\varepsilon\kappa(A)^2)$, but as far as we know, no matrix has yet been found for which the $\|I - Q^T Q\|_2$ is $O(\varepsilon\kappa(A)^{n-1})$ bound is sharp [43].

In the table, "CGS2" refers to classical Gram-Schmidt with one reorthogonalization pass. A single reorthgonalization pass suffices to make the Q factor orthogonal to machine precision, as long as the input matrix is numerically full rank, i.e., if $O(\varepsilon\kappa(A)) < 1$. This is the source of Kahan's maxim, "Twice is enough" [37]: the accuracy reaches its theoretical best after one reorthogonalization pass (see also [1]), and further reorthogonalizations do not improve orthogonality. However, TSQR needs only half as many messages to do just as well as CGS2. In terms of communication, TSQR's stability comes for free.

11 Platforms of interest for TSQR experiments and models

11.1 A large, but composable tuning space

TSQR is not a single algorithm, but a space of possible algorithms. It encompasses all possible reduction tree shapes, including:

1. Binary (to minimize number of messages in the parallel case)
2. Flat (to minimize communication volume in the sequential case)
3. Hybrid (to account for network topology, and/or to balance bandwidth demands with maximum parallelism)

as well as all possible ways to perform the local QR factorizations, including:

1. (Possibly multithreaded) standard LAPACK (DGEQRF)

2. An existing parallel QR factorization, such as ScaLAPACK's PDGEQRF

3. A "divide-and-conquer" QR factorization (e.g., [15])

4. Recursive (invoke another form of TSQR)

Choosing the right combination of parameters can help minimize communication between any or all of the levels of the memory hierarchy, from cache and shared-memory bus, to DRAM and local disk, to parallel filesystem and distributed-memory network interconnects, to wide-area networks.

The huge tuning space makes it a challenge to pick the right platforms for experiments. Luckily, TSQR's hierarchical structure makes tunings *composable*. For example, once we have a good choice of parameters for TSQR on a single multicore node, we don't need to change them when we tune TSQR for a cluster of these nodes. From the cluster perspective, it's as if the performance of the individual nodes improved. This means that we can benchmark TSQR on a small, carefully chosen set of scenarios, with confidence that they represent many platforms of interest.

11.2 Platforms of interest

Here we survey a wide variety of interesting platforms for TSQR, and explain the key features of each that we will distill into a small collection of experiments.

11.2.1 Single-node parallel, and explicitly swapping

The "cluster of shared-memory parallel (SMP) nodes" continues to provide a good price-performance point for many large-scale applications. This alone would justify optimizing the single-node case. Perhaps more importantly, the "multicore revolution" seeks to push traditionally HPC applications into wider markets, which favor the single-node workstation or even the laptop over the expensive, power-hungry, space-consuming, difficult-to-maintain cluster. A large and expanding class of users may never run their jobs on a cluster.

Multicore SMPs can help reduce communication costs, but cannot eliminate them. TSQR can exploit locality by sizing individual subproblems to fit within any level of the memory hierarchy. This gives programmers explicit control over management of communication between levels, much like a traditional "out-of-core" algorithm.[3] TSQR's hierarchical structure makes explicit swap management easy; it's just another form of communication. It gives us an optimized implementation for "free" on platforms like Cell or GPUs, which require explicitly moving data into separate storage areas for processing. Also, it lets us easily and efficiently solve problems too large to fit in DRAM. This seems

[3] We avoid this label because it's an anachronism ("core" refers to main system memory, constructed of solenoids rather than transistors or DRAM), and because people now easily confuse "core" with "processing unit" (in the sense of "multicore"). We prefer the more precise term *explicitly swapping*, or "out-of-X" for a memory hierarchy level X. For example, "out-of-DRAM" means using a disk, flash drive, or other storage device as swap space for problems too large to solve in DRAM.

like an old-fashioned issue, since an individual node nowadays can accommodate as much as 16 GB of DRAM. Explicitly swapping variants of libraries like ScaLAPACK tend to be ill-maintained, due to lack of interest. However, we predict a resurgence of interest in explicitly-swapping algorithms, for the following reasons:

- Single-node workstations will become more popular than multinode clusters, as the number of cores per node increases.
- The amount of DRAM per node cannot scale linearly with the number of cores per node, because of DRAM's power requirements. Trying to scale DRAM will wipe out the power savings promised by multicore parallelism.
- The rise to prominence of mobile computing – e.g., more laptops than desktops were sold in U.S. retail in 2006 – drives increasing concern for total-system power use.
- Most operating systems do not treat "virtual memory" as another level of the memory hierarchy. Default and recommended configurations for Linux, Windows XP, Solaris, and AIX on modern machines assign only 1–3 times as much swap space as DRAM, so it's not accurate to think of DRAM as a cache for disk. Few operating systems expand swap space on demand, and expanding it manually generally requires administrative access to the machine. It's better for security and usability to ask applications to adapt to the machine settings, rather than force users to change their machine for their applications.
- *Unexpected* use of virtual memory swap space generally slows down applications by orders of magnitude. HPC programmers running batch jobs consider this a performance problem serious enough to warrant terminating the job early and sizing down the problem. Users of interactive systems typically experience large (and often frustrating) delays in whole-system responsiveness when extensive swapping occurs.
- In practice, a single application need not consume all memory in order to trigger the virtual memory system to swap extensively.
- Explicitly swapping software does not stress the OS's virtual memory system, and can control the amount of memory and disk bandwidth resources that it uses.
- Alternate storage media such as solid-state drives offer more bandwidth than traditional magnetic hard disks. Typical hard disk read or write bandwidth as of this work's publication date is around 60 MB/s, whereas Samsung announced in May 2008 the upcoming release of a 256 GB capacity solid-state drive with 200 MB/s read bandwidth and 160 MB/s write bandwidth [32]. Solid-state drives are finding increasing use in mobile devices and laptops, due to their lower power requirements. This will make out-of-DRAM applications more attractive by widening any bandwidth bottlenecks.

11.2.2 Distributed-memory machines

Avoiding communication is a performance-enhancing strategy for distributed-memory architectures as well. TSQR can improve performance on traditional clusters as well as other networked systems, such as grid and perhaps even volunteer computing. Avoiding communication also makes improving network reliability less of a performance burden, as software-based reliability schemes use some combination of redundant and/or longer messages. Many distributed-memory supercomputers have high-performance parallel filesystems, which increase the bandwidth available for out-of-DRAM TSQR. This enables reducing per-node memory requirements without increasing the number of nodes needed to solve the problem.

11.3 Pruning the platform space

For single-node platforms, we think it pays to investigate both problems that fit in DRAM (perhaps with explicit cache management), and problems too large to fit in DRAM, that call for explicit swapping to a local disk. High-performance parallel filesystems offer potentially much more bandwidth, but we chose not to use them for our experiments for the following reasons:

- Lack of availability of a single-node machine with exclusive access to a parallel filesystem
- On clusters, parallel filesystems are usually shared among all cluster users, which would make it difficult to collect repeatable timings.

For multinode benchmarks, we opted for traditional clusters rather than volunteer computing, due to the difficulty of obtaining repeatable timings in the latter case.

11.4 Platforms for experiments

We selected the following experiments as characteristic of the space of platforms:

- Single node, sequential, out-of-DRAM, and
- Distributed memory, in-DRAM on each node.

We ran sequential TSQR on a laptop with a single PowerPC CPU. It represents the embedded and mobile space, with its tighter power and heat requirements. Details of the platform are as follows:

- Single-core PowerPC G4 (1.5 GHz)
- 512 KB of L2 cache
- 512 MB of DRAM on a 167 MHz bus
- One Fujitsu MHT2080AH 80 HB hard drive (5400 RPM)

- MacOS X 10.4.11
- GNU C Compiler (gcc), version 4.0.1
- vecLib (Apple's optimized dense linear algebra library), version 3.2.2

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

1. Pentium III cluster ("Beowulf")
   - Operated by the University of Colorado at Denver and the Health Sciences Center
   - 35 dual-socket 900 MHz Pentium III nodes with Dolphin interconnect
   - Floating-point rate: 900 Mflop/s per processor, peak
   - Network latency: less than 2.7 μs, benchmarked[4]
   - Network bandwidth: 350 MB/s, benchmarked upper bound
2. IBM BlueGene/L ("Frost")
   - Operated by the National Center for Atmospheric Research
   - One BlueGene/L rack with 1024 700 MHz compute CPUs
   - Floating-point rate: 2.8 Gflop/s per processor, peak
   - Network[5] latency: 1.5 μs, hardware
   - Network one-way bandwidth: 350 MB/s, hardware

11.5 Platforms for performance models

In Section 15, we estimate performance of CAQR, our QR factorization algorithm on a 2-D matrix layout, on three different parallel machines: an existing IBM POWER5 cluster with a total of 888 processors ("IBM POWER5"), a future proposed petascale machine with 8192 processors ("Peta"), and a collection of 128 processors linked together by the internet ("Grid"). Here are the parameters we use in our models for the three parallel machines:

- IBM POWER5
  - 888 processors
  - Floating-point rate: 7.6 Gflop/s per processor, peak
  - Network latency: 5 μs
  - Network bandwidth: 3.2 GB/s
- Peta

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

[5] 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.

  - 8192 processors
  - Floating-point rate: 500 Gflop/s per processor, peak
  - Network latency: 10 μs
  - Network bandwidth: 4 GB/s
- Grid
  - 128 processors
  - Floating-point rate: 10 Tflop/s, peak
  - Network latency: 0.1 s
  - Network bandwidth: 0.32 GB/s

Peta is our projection of a future high-performance computing cluster, and Grid is our projection of a collection of geographically separated high-performance clusters, linked over a TeraGrid-like backbone. 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. Similarly, each "processor" of Grid is itself a cluster, but we consider it as a single very fast sequential processor.

12 TSQR performance results

12.1 Scenarios used in experiments

Previous work covers some parts of the tuning space mentioned in Section 11.3. Gunter et al. implemented an out-of-DRAM version of TSQR on a flat tree, and used ScaLAPACK's QR factorization PDGEQRF to factor in-DRAM blocks in parallel [23]. Pothen and Raghavan [38] and Cunha et al. [10] both benchmarked parallel TSQR using a binary tree on a distributed-memory cluster, and implemented the local QR factorizations with a single-threaded version of DGEQRF. All these researchers observed significant performance improvements over previous QR factorization algorithms.

We chose to run two sets of experiments. The first set covers the out-of-DRAM case on a single CPU. The second set is like the parallel experiments of previous authors in that it uses a binary tree on a distributed-memory cluster, but it improves on their approach by using a better local QR factorization (the divide-and-conquer approach – see [16]).

12.2 Sequential out-of-DRAM tests

We developed an out-of-DRAM version of TSQR that uses a flat reduction tree. It invokes the system vendor's native BLAS and LAPACK libraries. Thus, it can exploit a multithreaded BLAS on a machine with multiple CPUs, but the parallelism is limited to operations on a single block of the matrix. We used standard POSIX blocking file operations, and made no attempt to overlap

communication and computation. Exploiting overlap could at best double the performance.

We ran sequential tests on a laptop with a single PowerPC CPU, as described in Section 11.4. In our experiments, we first used both out-of-DRAM TSQR and standard LAPACK QR 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 extrapolated the standard LAPACK QR timings up to the larger problem sizes, in order to estimate the runtime if memory were unbounded. LAPACK's QR factorization swaps so much for out-of-DRAM problem sizes that its actual runtimes are many times larger than these extrapolated unbounded-memory runtime estimates. As mentioned in Section 11.2, once an in-DRAM algorithm begins swapping, it becomes so much slower that most users prefer to abort the computation and try solving a smaller problem. No attempt to optimize by overlapping communication and computation was made.

We used the following power law for the extrapolation:

$$t = A_1 b m^{A_2} n^{A_3},$$

in which t is the time spent in computation, b is the number of input matrix blocks, m is the number of rows per block, and n is the number of columns in the matrix. After taking logarithms of both sides, we performed a least squares fit of $\log(A_1)$, A_2, and A_3. The value of A_2 was 1, as expected. The value of A_3 was about 1.6. This is less than 2 as expected, given that increasing the number of columns increases the computational intensity and thus the potential for exploitation of locality (a BLAS 3 effect). We expect around two digits of accuracy in the parameters, which in themselves are not as interesting as the extrapolated runtimes; the parameter values mainly serve as a sanity check.

12.2.1 Results

Figure 5 shows the measured in-DRAM results on the laptop platform, and Figure 6 shows the (measured TSQR, extrapolated LAPACK) out-of-DRAM results on the same platform. In these figures, the number of blocks used, as well as the number of elements in the input matrix (and thus the total volume of communication), is the same for each group of five bars. We only varied the number of blocks and the number of columns in the matrix. For each graph, the total number of rows in the matrix is constant for all groups of bars. Note that we have not tried to overlap I/O and computation in this implementation. The trends in Figure 5 suggest that the extrapolation is reasonable: TSQR takes 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

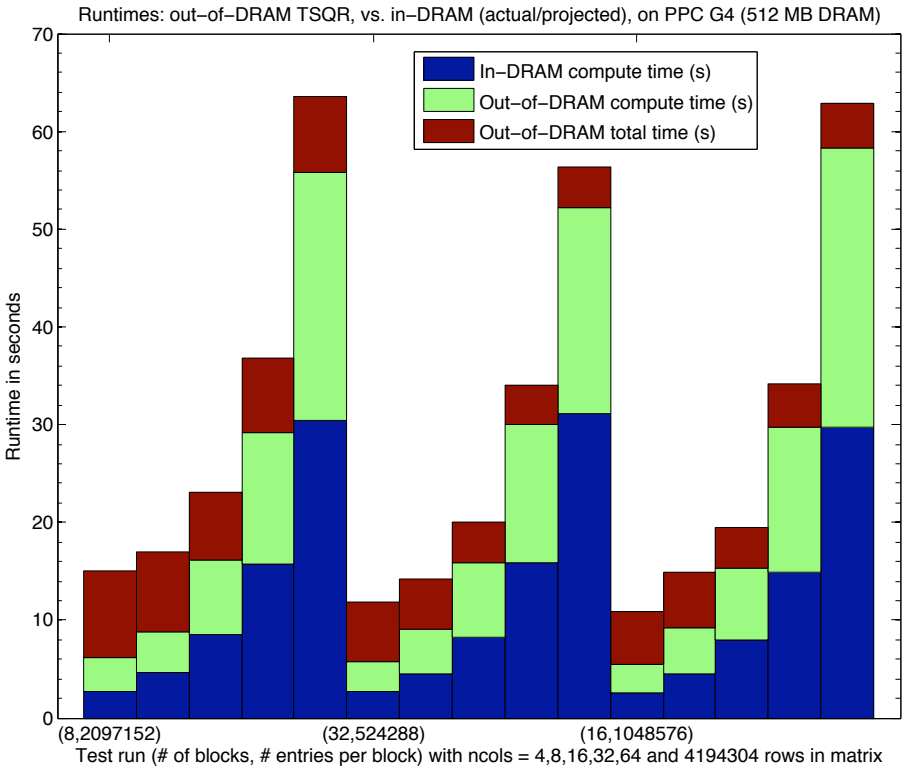


Figure 5: Runtimes (in seconds) of out-of-DRAM TSQR and standard QR (LAPACK's DGEQRF) on a single-processor laptop. All data is measured. We limit memory usage to 256 MB, which is half of the laptop's total system memory, so that we can collect performance data for DGEQRF. The graphs show different choices of block dimensions and number of blocks. The top of the blue bar is the benchmarked 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, and TSQR starts and ends on disk.

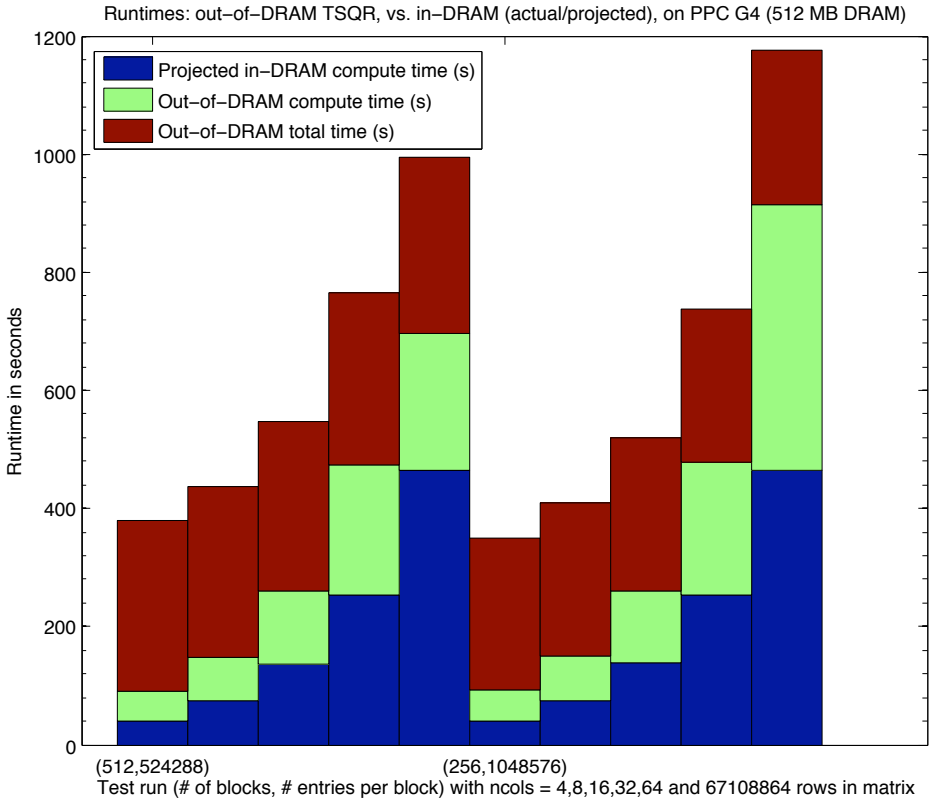


Figure 6: Measured runtime (in seconds) of out-of-DRAM TSQR, compared against extrapolated runtime (in seconds) of standard QR (LAPACK's DGEQRF) on a single-processor laptop. We use the data in Figure 5 to construct a power-law performance extrapolation. The graphs show different choices of block dimensions and number of blocks. The top of the blue bar is 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.

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK
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

Table 9: Runtime in seconds of various parallel QR factorizations on the Beowulf machine. The total number of rows $m = 100000$ 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 $\Theta(mn^2)$. If an algorithm scales perfectly, then all the numbers in that algorithm's column should be constant. Both the 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 runtime measurement.

on disk. The height of the reddish-brown bars in Figures 5 and 6 is the I/O time for TSQR, which can be used to estimate the LAPACK I/O time. Add this to the blue bar (the LAPACK compute time) to estimate the runtime when the LAPACK QR routine must load the matrix from disk and store the results back to disk.

12.2.2 Conclusions

The main purpose of our out-of-DRAM code is not to outperform existing in-DRAM algorithms, but to be able to solve classes of problems which the existing algorithms cannot solve. The above graphs show that the penalty of an explicitly swapping approach is about 2x, which is small enough to warrant its practical use. This holds even for problems with a relatively low computational intensity, such as when the input matrix has very few columns. Furthermore, picking the number of columns sufficiently large may allow complete overlap of file I/O by computation.

12.3 Parallel cluster tests

We also have results from a parallel 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 [16]. Tests show that DGEQR3 consistently outperforms LAPACK's DGEQRF by a large margin for matrix dimensions of interest.

We ran our experiments on two platforms: a Pentium III cluster ("Beowulf")

# procs	CholeskyQR	TSQR (DGEQR3)	CGS	MGS	TSQR (DGEQRF)	ScaLAPACK
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 10: Runtime 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 $\lceil m/P \rceil = 100000$ 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 numbers 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	TSQR		ScaLAPACK	
	(DGEQR3)	(DGEQRF)	(PDGEQRF)	(PDGEQR2)
32	690	276	172	206
64	666	274	172	206
128	662	316	196	232
256	610	322	184	218

Table 11: Performance per processor (Mflop / s / (# processors)) on a $10^6 \times 50$ matrix, on the Frost machine. This metric illustrates strong scaling (constant problem size, but number of processors increases). If an algorithm scales perfectly, than all the numbers in that algorithm's column should be constant. DGEQR3 is a recursive local QR factorization, and DGEQRF LAPACK's standard local QR factorization.

and on a BlueGene/L ("Frost"), both described in detail in Section 11.4. The experiments compare many different implementations of a parallel QR factorization. "CholeskyQR" first computes the product $A^T A$ using a reduction, then performs a QR factorization of the product. It is less stable than TSQR, as it squares the condition number of the original input matrix (see Table 8 in Section 10 for a stability comparison of various QR factorization methods). TSQR was tested both with the recursive local QR factorization DGEQR3, and the standard LAPACK routine DGEQRF. Both CGS and MGS were timed.

12.3.1 Results

Tables 9 and 10 show the results of two different performance experiments on the Pentium III cluster. In the first of these, the total number of rows $m = 100000$ 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 $\Theta(mn^2)$. If an algorithm scales perfectly, then all the numbers in that algorithm's column should be constant. Both the 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 runtime measurement. The results show that TSQR scales better than CGS or MGS, 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.

Table 10 shows the results of the second set of experiments on the Pentium III cluster. In these experiments, the ratio $\lceil m/P \rceil = 100000$ and the total number of columns $n = 50$ were kept constant as the number of processors P varied from 1 to 64. This was meant to illustrate weak scaling with respect to the total number of rows m in the matrix. If an algorithm scales perfectly, then all the numbers in that algorithm's column should be constant. 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.

Table 11 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, than all the numbers in that algorithm's column should be constant. Two different versions of ScaLAPACK's QR factorization were used: PDGEQR2 is the textbook Householder QR panel

factorization, and PDGEQRF is the blocked version which tries to coalesce multiple trailing matrix updates into one. The results again show that TSQR scales at least as well as ScaLAPACK's QR factorization, which unlike TSQR is presumably highly tuned on this platform. Furthermore, using the recursive local QR factorization with TSQR makes its performance competitive with that of ScaLAPACK.

12.3.2 Conclusions

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 6.8× on 16 processors (and 3.5× on 64 processors) on the Pentium III cluster, and successfully competes with ScaLAPACK's QR on the BlueGene/L machine.

13 Parallel 2-D QR factorization

The CAQR ("Communication Avoiding QR") algorithm uses TSQR to perform a parallel QR factorization of a dense matrix A on a two-dimensional grid of processors $P = P_r \times P_c$. The $m \times n$ matrix (with $m \geq n$) is distributed using a 2-D block cyclic layout over the processor grid, 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 2-D block cyclic layout of a dense matrix, please refer to [5], in particular to the section entitled "Details of Example Program #1." There is also an analogous sequential version of CAQR, which we describe in detail in Appendix C.

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. As we will show, 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, and adapting the rest of the algorithm to work with TSQR's representation of the panel Q factors, removes the bottleneck. We use the reduction-to-one-processor variant of TSQR, as the panel's R factor need only be stored on one processor (the pivot block's processor).

CAQR is defined inductively. 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 7 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

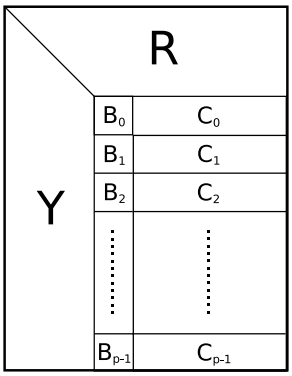


Figure 7: Step j of the QR factorization algorithm. First, the current panel of width b, consisting of the blocks B_0, B_1, ..., B_{p-1}, is factorized using TSQR. Here, p is the number of blocks in the current panel. Second, the trailing matrix, consisting of the blocks C_0, C_1, ..., C_{p-1}, is updated. The matrix elements above the current panel and the trailing matrix belong to the R factor and will not be modified further by the QR factorization.

that hold the current panel j. The $m_j \times b$ matrix B represents the current panel j. The $m_j \times (n_j - b)$ matrix C is the trailing matrix that needs to be updated after the TSQR factorization of B. For each processor p, we refer to the first b rows of its first block row of B and C as B_p and C_p respectively.

We first introduce some notation to help us refer to different parts of a binary TSQR reduction tree.

- $level(i,k) = \left\lfloor \frac{i}{2^k} \right\rfloor$ denotes the node at level k of the reduction tree which is assigned to a set of processors that includes processor i. The initial stage of the reduction, with no communication, is $k = 0$.
- $first\_proc(i,k) = 2^k level(i,k)$ is the index of the "first" processor associated with the node $level(i,k)$ at stage k of the reduction tree. In a reduction (not an all-reduction), it receives the messages from its neighbors and performs the local computation.
- $target(i,k) = first\_proc(i,k) + (i + 2^{k-1}) \mod 2^k$ is the index of the processor with which processor i exchanges data at level k of the butterfly all-reduction algorithm.
- $target\_first\_proc(i,k) = target(first\_proc(i,k)) = first\_proc(i,k) + 2^{k-1}$ is the index of the processor with which $first\_proc(i,k)$ exchanges data in an all-reduction at level k, or the index of the processor which sends its data to $first\_proc(i,k)$ in a reduction at level k.

Algorithm 9 outlines the j-th iteration of the QR decomposition. First, the block column j is factored using TSQR. We assume for ease of exposition that

TSQR is performed using a binary tree. After the block column factorization is complete, the matrices C_p are 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 as described in Section 6.4. Note that only one of the trailing matrix processors in each neighbor group continues to be involved 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.

We see that CAQR consists of $\frac{n}{b}$ TSQR factorizations involving P_r processors each, and $n/b - 1$ applications of the resulting Householder vectors. Table 12 expresses the performance model over a rectangular grid of processors. A detailed derivation of the model is given in Appendix E.

	Parallel CAQR
# messages	$\frac{3n}{b}\log(P_r) + \frac{2n}{b}\log(P_c)$
# words	$\left(\frac{bn}{2} + \frac{n^2}{P_c}\right)\log(P_r) + \left(\frac{mn-n^2/2}{P_r} + 2n\right)\log(P_c)$
# flops	$\frac{2n^2(3m-n)}{3P} + \frac{bn^2}{2P_c} + \frac{3bn(m-n/2)}{P_r} + \left(\frac{4b^2n}{3} + \frac{n^2(3b+5)}{2P_c}\right)\log(P_r) - b^2n$
# divisions	$\frac{mn-n^2/2}{P_r} + \frac{bn}{2}(\log(P_r) - 1)$
	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	$\left(\frac{2n^2}{3P}(3m-n) + \frac{bn^2}{2P_c} + \frac{1}{P_r}\left(3bn\left(m - \frac{n}{2}\right) - \frac{b^2n}{3}\right)\right)$
# divisions	$\frac{n}{P_r}\left(m - \frac{n}{2}\right)$

Table 12: Performance models of parallel CAQR and ScaLAPACK's PDGEQRF when factoring an $m \times n$ matrix, distributed in a 2-D 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 only, "flops" only includes floating-point additions and multiplications, not floating-point divisions. Some lower-order terms are omitted. We generally assume $m \geq n$.

The parallelization of the computation is represented by the number of multiplies and adds and by the number of divides, in Table 12. We discuss first the parallelization of multiplies and adds. The first term represents mainly the parallelization of the local Householder update corresponding to the leaves of the TSQR tree (the matrix matrix multiplication in step 3 of Algorithm 9). The second term corresponds to forming the T_{p0} matrices for the local Householder update in step 3 of the algorithm. The third term represents the QR factorization of a panel of width b that corresponds to the leaves of the TSQR tree (part of step 1) and part of the local rank-b update (triangular matrix matrix multiplication) in step 3 of the algorithm.

The fourth term in the number of multiplies and adds represents the redundant computation introduced by the TSQR formulation. In this term, the number of flops performed for computing the QR factorization of two upper triangular matrices at each node of the TSQR tree is $(2/3)nb^2\log(P_r)$ (step 1).

The number of flops in step $(4.a)$ is given by $(2/3)nb^2\log(P_r)$. The number of flops performed during the Householder updates issued by each QR factorization of two upper triangular matrices is $n^2(3b+5)/(2P_c)\log(P_r)$.

The runtime estimation in Table 12 does not take into account the overlap of computation and communication in steps $(4.a)$ and $(4.b)$ or the overlap in steps $(4.d)$ and $(4.e)$. Suppose that at each step of the QR factorization, the condition

$$\alpha + \beta\frac{b(n_j-b)}{P_c} > \gamma b(b+1)\frac{n_j-b}{P_c}$$

is fulfilled. This is the case for example when $\beta/\gamma > b+1$. Then the fourth non-division flops term that accounts for the redundant computation is decreased by $n^2(b+1)\log(P_r)/P_c$, about a factor of 3.

The execution time for a square matrix ($m=n$), on a square grid of processors ($P_r = P_c = \sqrt{P}$) and with more lower order terms ignored, simplifies to:

$$T_{Par.\ CAQR}(n,n,\sqrt{P},\sqrt{P}) = \gamma\left(\frac{4n^3}{3P} + \frac{3n^2b}{4\sqrt{P}}\log(P)\right) + \beta\frac{3n^2}{4\sqrt{(P)}}\log(P) + \alpha\frac{5n}{2b}\log(P). \tag{6}$$

We will compare CAQR with the standard QR factorization algorithm implemented in ScaLAPACK in Section 14. Our models assume that the QR factorization does not use a look-ahead technique during the right-looking factorization. With the look-ahead right-looking approach, the communications are pipelined from left to right. At each step of factorization, we would model the latency cost of the broadcast within rows of processors as 2 instead of $\log(P_c)$.

14 Comparison of ScaLAPACK's QR and CAQR

Here, we compare the standard QR factorization algorithm implemented in ScaLAPACK and the new proposed CAQR approach. We suppose that we decompose a $m \times n$ matrix with $m \geq n$ which is distributed block cyclically over a P_r by P_c grid of processors, where $P_r \times P_c = P$. The two-dimensional block cyclic distribution uses square blocks of dimension $b \times b$. Equation (7) represents the runtime estimation of ScaLAPACK's QR, in which we assume that there is no attempt to pipeline communications from left to right and some lower order terms are omitted.

$$\begin{aligned} T_{SC}(m,n,P_r,P_c) \quad = \quad & \left[\tfrac{2}{3P}n^2(3m-n) + \tfrac{n^2b}{2P_c} + \tfrac{1}{P_r}(3(b+1)n(m-\tfrac{n}{2}) - nb\left(\tfrac{b}{3}+1.5\right)\right]\gamma + \\ & + \tfrac{1}{P_r}\left(mn - \tfrac{n^2}{2}\right)\gamma_d + \\ & + \log P_r\left[3n\left(1+\tfrac{1}{b}\right)\alpha + \left(\tfrac{n^2}{P_c} + n(b+2)\right)\beta\right] + \\ & + \log P_c\left[\tfrac{2n}{b}\alpha + \left(\tfrac{1}{P_r}\left(mn - \tfrac{n^2}{2}\right) + \tfrac{nb}{2}\right)\beta\right] \end{aligned} \tag{7}$$

When $P_r = P_c = \sqrt{P}$ and $m = n$, and ignoring more lower-order terms, Equation (7) simplifies to

$$T_{SC}(n, n, \sqrt{P}, \sqrt{P}) = \frac{4}{3}\frac{n^3}{P}\gamma + \frac{3}{4}\log P \frac{n^2}{\sqrt{P}}\beta + \left(\frac{3}{2} + \frac{5}{2b}\right) n \log P \alpha \qquad (8)$$

Consider Equation (7). The latency term we want to eliminate is $3n \log P_r \alpha$, which comes from the QR factorization of a panel of width b (PDGEQR2 routine). This involves first the computation of a Householder vector v spread over P_r processors (DGEBS2D and PDNRM2, which use a tree to broadcast and compute a vector norm). Second, a Householder update is performed (PDLARF) by applying $I - \tau v v^T$ to the rest of the columns in the panel. It calls DGSUM2D in particular, which uses a tree to combine partial sums from a processor column. In other words, the potential ScaLAPACK latency bottleneck is entirely in factoring a block column.

Comparing the CAQR performance model in Equation (6) with Equation (7) shows that CAQR overcomes PDGEQRF's latency bottleneck at the cost of some redundant computation. It performs a larger number of floating-point operations, given mainly by the lower order terms:

$$\gamma \left(\frac{4b^2 n}{3} + \frac{n^2(3b+5)}{2P_c}\right) \log P_r + \gamma_d \frac{nb}{2}(\log(P_r) - 1).$$

15 CAQR performance estimation

We use the performance model developed in the previous section to estimate the performance of parallel CAQR on three computational systems, IBM POWER5, Peta, and Grid, and compare it to ScaLAPACK's parallel QR factorization routine PDGEQRF. Peta is a model of a petascale machine with 8100 processors, and Grid is a model of 128 machines connected over the internet. Each processor in Peta and Grid can be itself a parallel machine, but our models consider the parallelism only between these parallel machines.

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 vs. 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.

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 2D block cyclic distribution, with square blocks of size $b \times b$. For each machine 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 matrix size n is varied in the range 10^3, $10^{3.5}$, 10^4, ..., $10^{7.5}$. The block size b is varied in the range 1, 5, 10, ..., 50, 60, ..., $\min(200, m/P_r, n/P_c)$. The number of processors is varied from 1 to the

largest power of 2 smaller than p_{max}, in which p_{max} is the maximum number of processors available in the system. The values for P_r and P_c are also chosen to be powers of two.

We describe now the parameters used for the three parallel machines. The available memory on each processor is given in units of 8-byte (IEEE 754 double-precision floating-point) words. When we evaluate the model, we set the γ 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. This estimate favors ScaLAPACK rather than CAQR, as ScaLAPACK requires more communication and CAQR more floating-point operations. The inverse network bandwidth β has units of seconds per word. The bandwidth for Grid is estimated to be the Teragrid backbone bandwidth of 40 GB/sec divided by p_{max}.

- **IBM POWER5:** $p_{max} = 888$, peak flop rate is 7.6 Gflop/s, $mem = 5{\cdot}10^8$ words, $\alpha = 5 \cdot 10^{-6}$ s, $\beta = 2.5 \cdot 10^{-9}$ s/word ($1/\beta = 400$ Mword/s $= 3.2$ GB/s).
- **Peta:** $p_{max} = 8192$, peak flop rate is 500 Gflop/s, $mem = 62.5{\cdot}10^9$ words, $\alpha = 10^{-5}$ s, $\beta = 2 \cdot 10^{-9}$ s/word ($1/\beta = 500$ Mword/s $= 4$ GB/s).
- **Grid:** $p_{max} = 128$, peak flop rate is 10 Tflop/s, $mem = 10^{14}$ words, $\alpha = 10^{-1}$ s, $\beta = 25 \cdot 10^{-9}$ s/word ($1/\beta = 40$ Mword/s $= .32$ GB/s).

There are 13 plots shown for each parallel machine. The first three plots display for specific n and P values our models of

- the best speedup obtained by CAQR, with respect to the runtime using the fewest number of processors with enough memory to hold the matrix (which may be more than one processor),
- the best speedup obtained by PDGEQRF, computed similarly, and
- the ratio of PDGEQRF runtime to CAQR runtime.

The next ten plots are divided in two groups of five. The first group presents performance results for CAQR and the second group presents performance results for PDGEQRF. The first two plots of each group of five display the corresponding optimal values of b and P_r obtained for each combination of n and P. (Since $P_c = P/P_r$, we need not specify P_c explicitly.) The last 3 plots of each group of 5 give the computation time to total time ratio, the latency time to total time ratio, and the bandwidth time to total time ratio.

The white regions in the plots signify that the problem needed too much memory with respect to the memory available on the machine. Note that in our performance models, the block size b has no meaning on one processor, because there is no communication, and the term $4n^3/(3P)$ dominates the computation. Thus, for one processor, we set the optimal value of b to 1 as a default.

CAQR leads to significant improvements with respect to PDGEQRF when the latency represents an important fraction of the total time, as for example

when a small matrix is computed on a large number of processors. On IBM POWER5, the best improvement is predicted for the smallest matrix in our test set ($n = 10^3$), when CAQR will outperform PDGEQRF by a factor of 9.7 on 512 processors. On Peta, the best improvement is a factor of 22.9, obtained for $n = 10^4$ and $P = 8192$. On Grid, the best improvement is obtained for one of the largest matrix in our test set $m = n = 10^{6.5}$, where CAQR outperforms PDGEQRF by a factor of 5.3 on 128 processors.

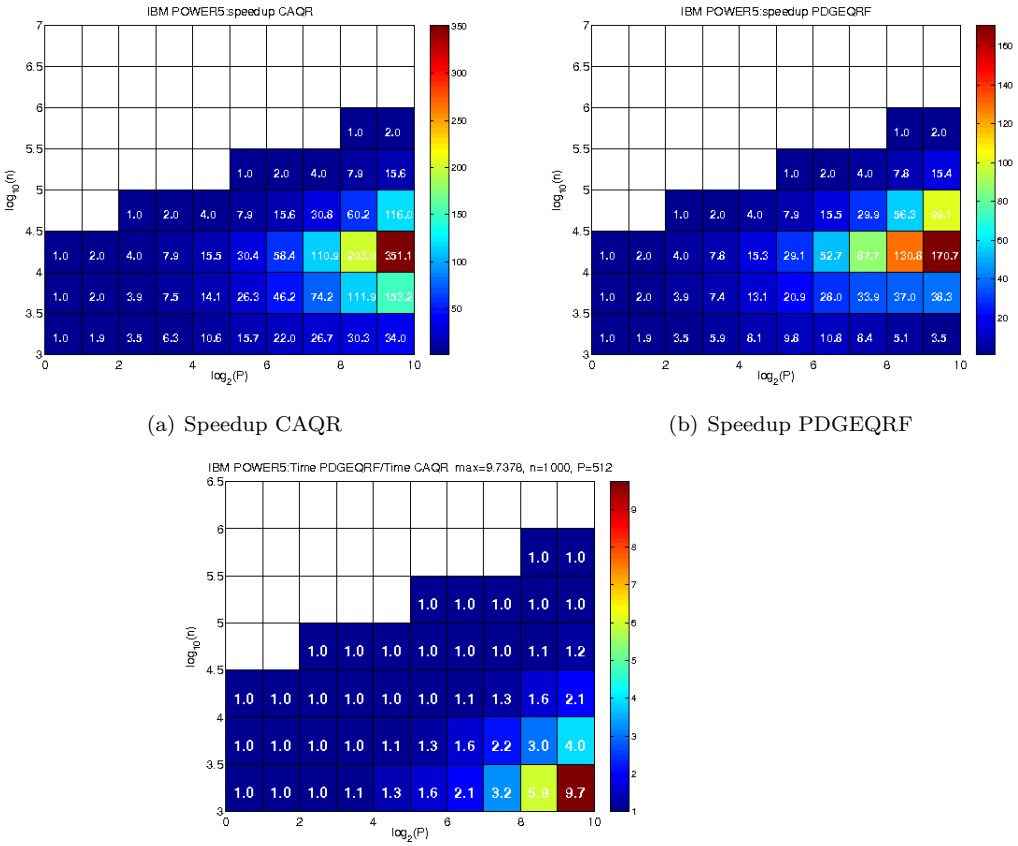
15.1 Performance prediction on IBM POWER5

Figures 8, 9, and 10 depict modeled performance on the IBM POWER 5 system. CAQR has the same estimated performance as PDGEQRF when the computation dominates the total time. But it outperforms PDGEQRF when the fraction of time spent in communication due to latency becomes significant. The best improvements are obtained for smaller n and larger P, as displayed in Figure 8(c), the bottom right corner. For the smallest matrix in our test set ($n = 10^3$), we predict that CAQR will outperform PDGEQRF by a factor of 9.7 on 512 processors. As shown in Figure 10(d), for this matrix, the communication dominates the runtime of PDGEQRF, with a fraction of 0.9 spent in latency. For CAQR, the time spent in latency is reduced to a fraction of 0.5 of the total time from 0.9 for PDGEQRF, and the time spent in computation is a fraction of 0.3 of the total time. This is illustrated in Figures 9(c) and 9(d).

Another performance comparison consists in determining the improvement obtained by taking the best performance independently for CAQR and PDGEQRF, when varying the number of processors from 1 to 512. For $n = 10^3$, the best performance for CAQR is obtained when using $P = 512$ and the best performance for PDGEQRF is obtained for $P = 64$. This leads to a speedup of more than 3 for CAQR compared to PDGEQRF. 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 13, which shows that CAQR always is at least as fast as PDGEQRF, and often significantly faster (up to 3× faster in some cases).

Figure 8 shows that CAQR should scale well, with a speedup of 351 on 512 processors when $m = n = 10^4$. A speedup of 116 with respect to the parallel time on 4 processors (the fewest number of processors with enough memory to hold the matrix) is predicted for $m = n = 10^{4.5}$ on 512 processors. In these cases, CAQR is estimated to outperform PDGEQRF by factors of 2.1 and 1.2, respectively.

Figures 9(b) and 10(b) show that PDGEQRF has a smaller value for optimal P_r than CAQR. This trend is more significant in the bottom left corner of Figure 10(b), where the optimal value of P_r for PDGEQRF is 1. This corresponds to a 1D block column cyclic layout. In other words, PDGEQRF runs faster by reducing the $3n \log P_r$ term of the latency cost of Equation (7) by choosing a small P_r. PDGEQRF also tends to have a better performance for a smaller block size than CAQR, as displayed in Figures 9(a) and 10(a). The optimal block size b varies from 1 to 15 for PDGEQRF, and from 1 to 30 for CAQR.



(a) Speedup CAQR

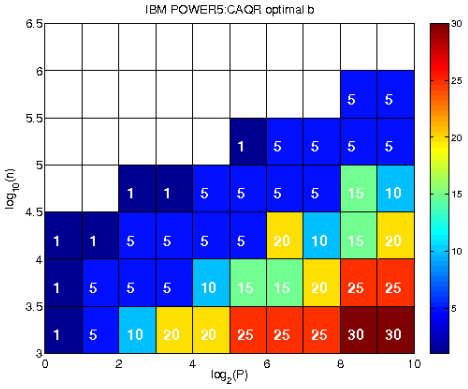
(b) Speedup PDGEQRF

(c) Comparison

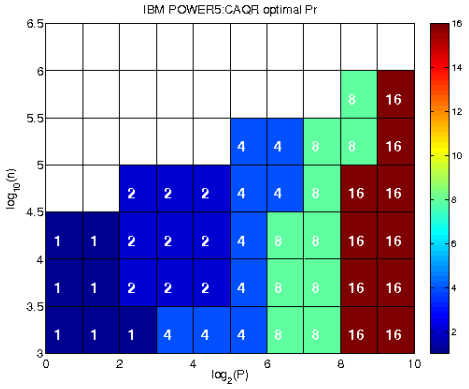
Figure 8: Performance prediction comparing CAQR and PDGEQRF on IBM POWER5.

$\log_{10} n$	Best $\log_2 P$ for PDGEQRF	CAQR speedup
3.0	6	2.1
3.5	8	3.0
4.0	9	2.1
4.5	9	1.2
5.0	9	1.0
5.5	9	1.0

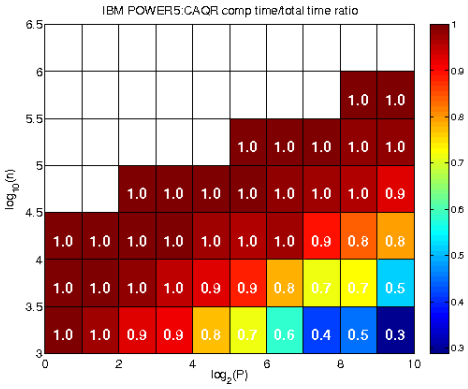
Table 13: Estimated runtime of PDGEQRF divided by estimated runtime of CAQR on a square $n \times n$ matrix, on the IBM POWER5 platform, for those values of P (number of processors) for which PDGEQRF performs the best for that problem size.



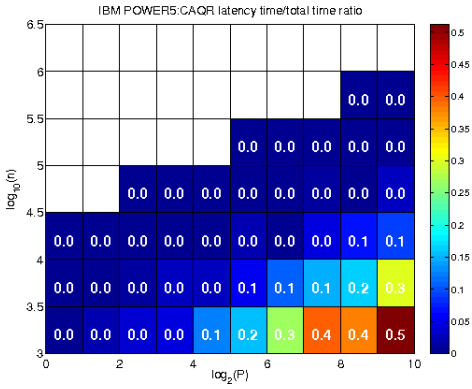
(a) Optimal b



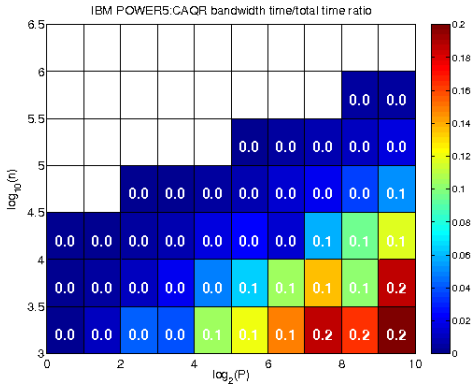
(b) Optimal P_r



(c) Fraction of time in computation

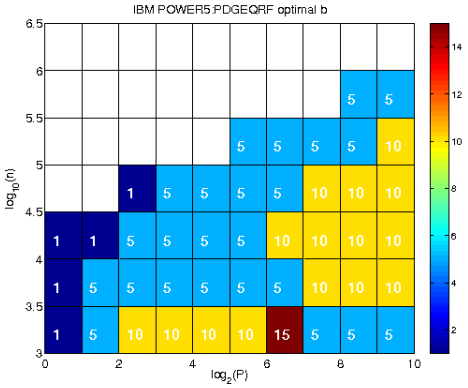


(d) Fraction of time in latency

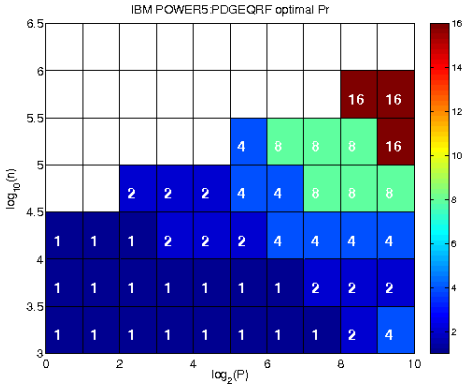


(e) Fraction of time in bandwidth

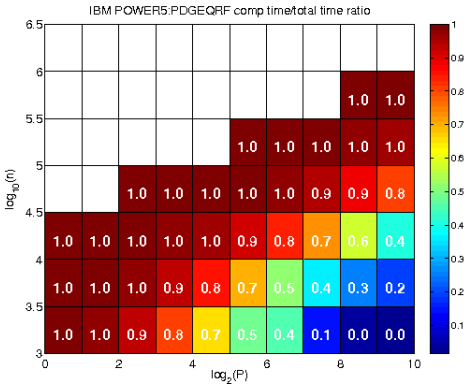
Figure 9: Performance prediction for CAQR on IBM POWER5.



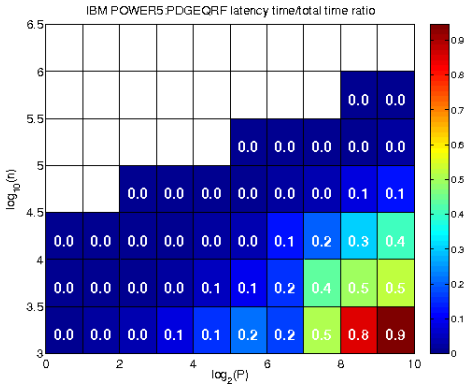
(a) Optimal b



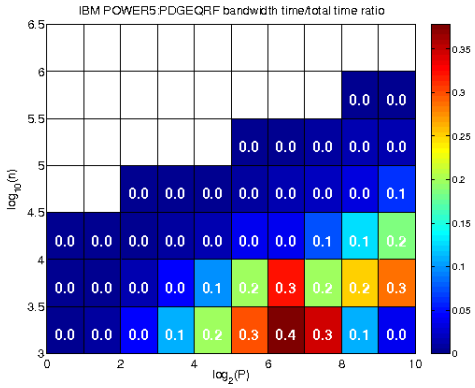
(b) Optimal P_r



(c) Fraction of time in computation



(d) Fraction of time in latency



(e) Fraction of time in bandwidth

Figure 10: Performance prediction for PDGEQRF on IBM POWER5.

15.2 Performance prediction on Peta

Figures 11, 12, and 13 show our performance estimates of CAQR and PDGEQRF on the Petascale machine. The estimated division of time between computation, latency, and bandwidth for PDGEQRF is illustrated in Figures 13(c), 13(d), and 13(e). In the upper left corner of these figures, the computation dominates the total time, while in the right bottom corner the latency dominates the total time. In the narrow band between these two regions, which goes from the left bottom corner to the right upper corner, the bandwidth dominates the time. CAQR decreases the latency cost, as can be seen in Figures 12(c), 12(d), and 12(e). There are fewer test cases for which the latency dominates the time (the right bottom corner of Figure 12(d)). This shows that CAQR is expected to be effective in decreasing the latency cost. The left upper region where the computation dominates the time is about the same for both algorithms. Hence for CAQR there are more test cases for which the bandwidth term is an important fraction of the total time.

Note also in Figures 13(b) and 12(b) that optimal P_r has smaller values for PDGEQRF than for CAQR. There is an interesting regularity in the value of optimal P_r for CAQR. CAQR is expected to have its best performance for (almost) square grids.

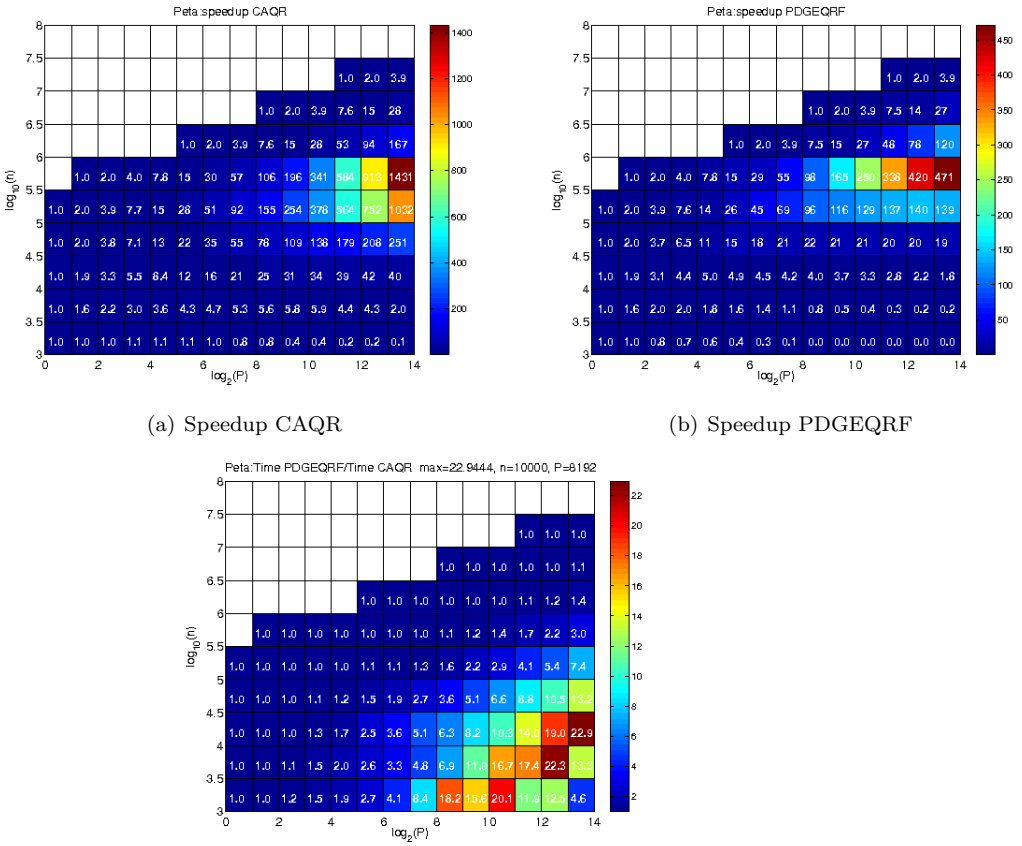
As can be seen in Figure 11(a), CAQR is expected to show good scalability for large matrices. For example, for $n = 10^{5.5}$, a speedup of 1431, measured with respect to the time on 2 processors, is obtained on 8192 processors. For $n = 10^{6.4}$ a speedup of 166, measured with respect to the time on 32 processors, is obtained on 8192 processors.

CAQR leads to more significant improvements when the latency represents an important fraction of the total time. This corresponds to the right bottom corner of Figure 11(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.

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, 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 14, which shows that CAQR always is at least as fast as PDGEQRF, and often significantly faster (up to 7.4× faster in some cases).

15.3 Performance prediction on Grid

The performance estimation obtained by CAQR and PDGEQRF on the Grid is displayed in Figures 14, 15, and 16. For small values of n both algorithms do



(a) Speedup CAQR

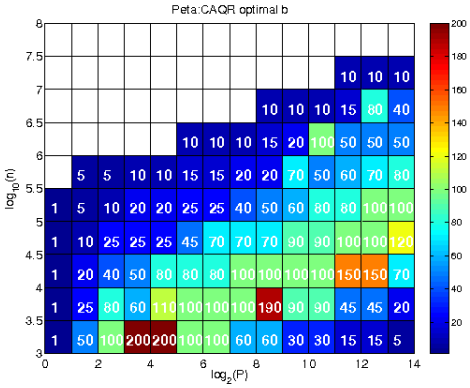
(b) Speedup PDGEQRF

(c) Comparison

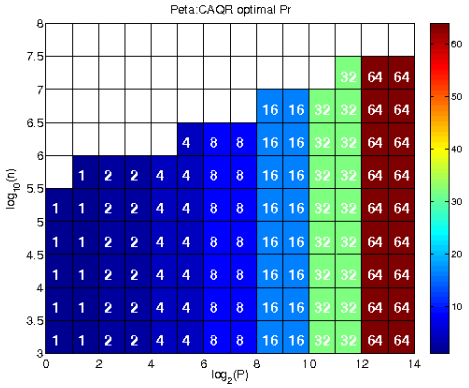
Figure 11: Performance prediction comparing CAQR and PDGEQRF on Peta.

$\log_{10} n$	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

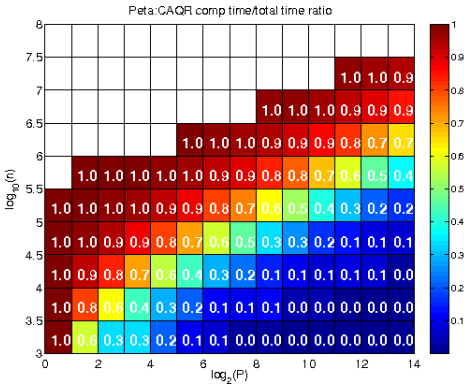
Table 14: Estimated runtime of PDGEQRF divided by estimated runtime of CAQR on a square $n \times n$ matrix, on the Peta platform, for those values of P (number of processors) for which PDGEQRF performs the best for that problem size.



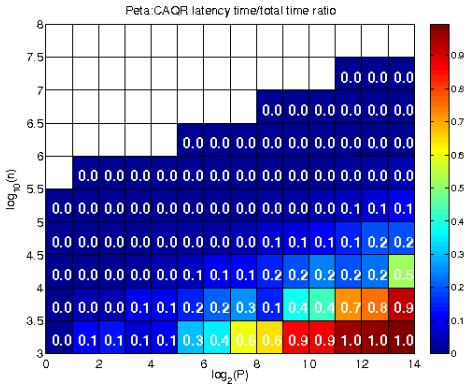
(a) Optimal b



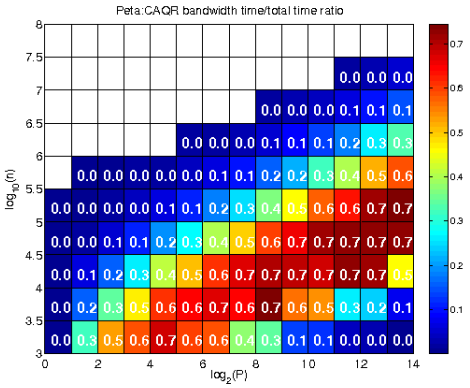
(b) Optimal P_r



(c) Fraction of time in computation

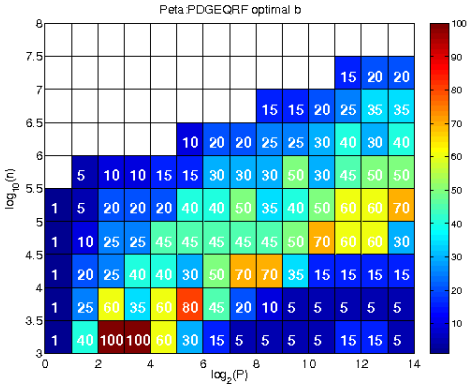


(d) Fraction of time in latency

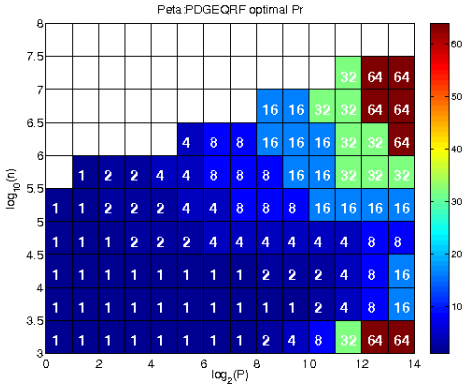


(e) Fraction of time in bandwidth

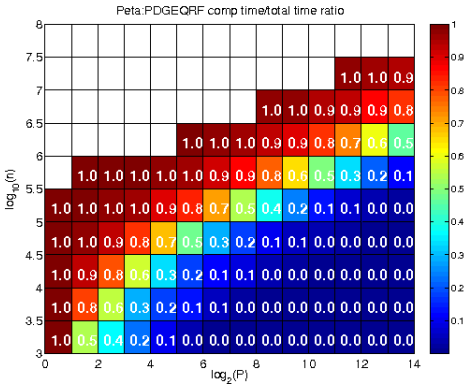
Figure 12: Performance prediction for CAQR on Peta.



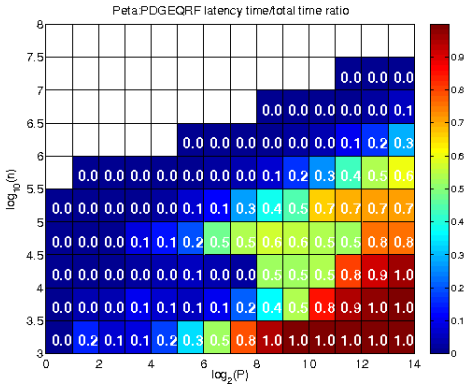
(a) Optimal b



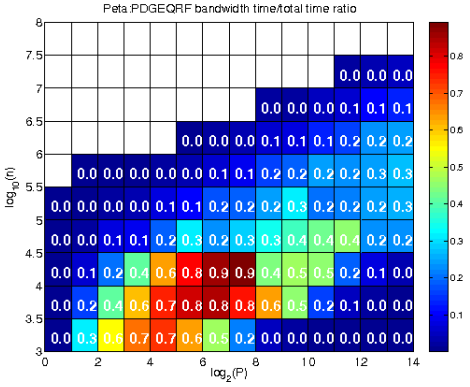
(b) Optimal P_r



(c) Fraction of time in computation



(d) Fraction of time in latency



(e) Fraction of time in bandwidth

Figure 13: Performance prediction for PDGEQRF on Peta.

not obtain any speedup, even on small number of processors. Hence we discuss performance results for values of n bigger than 10^5.

As displayed in Figures 15(a) and 16(a), the optimal block size for both algorithms is very often 200, the largest value in the allowed range. The optimal value of P_r for PDGEQRF is equal to 1 for most of the test cases (Figure 16(b)), while CAQR tends to prefer a square grid (Figure 15(b)). This suggests that CAQR can successfully exploit parallelism within block columns, unlike PDGEQRF.

As can be seen in Figures 16(c), 16(d), and 16(e), for small matrices, communication latency dominates the total runtime of PDGEQRF. For large matrices and smaller numbers of processors, computation dominates the runtime. For the test cases situated in the band going from the bottom left corner to the upper right corner, bandwidth costs dominate the runtime. The model of PDGEQRF suggests that the best way to decrease the latency cost with this algorithm is to use, in most test cases, a block column cyclic distribution (the layout obtained when $P_r = 1$). In this case the bandwidth cost becomes significant.

The division of time between computation, latency, and bandwidth has a similar pattern for CAQR, as shown in Figures 15(c), 15(d), and 15(e). However, unlike PDGEQRF, CAQR has as optimal grid shape a square or almost square grid of processors, which suggests that CAQR is more scalable.

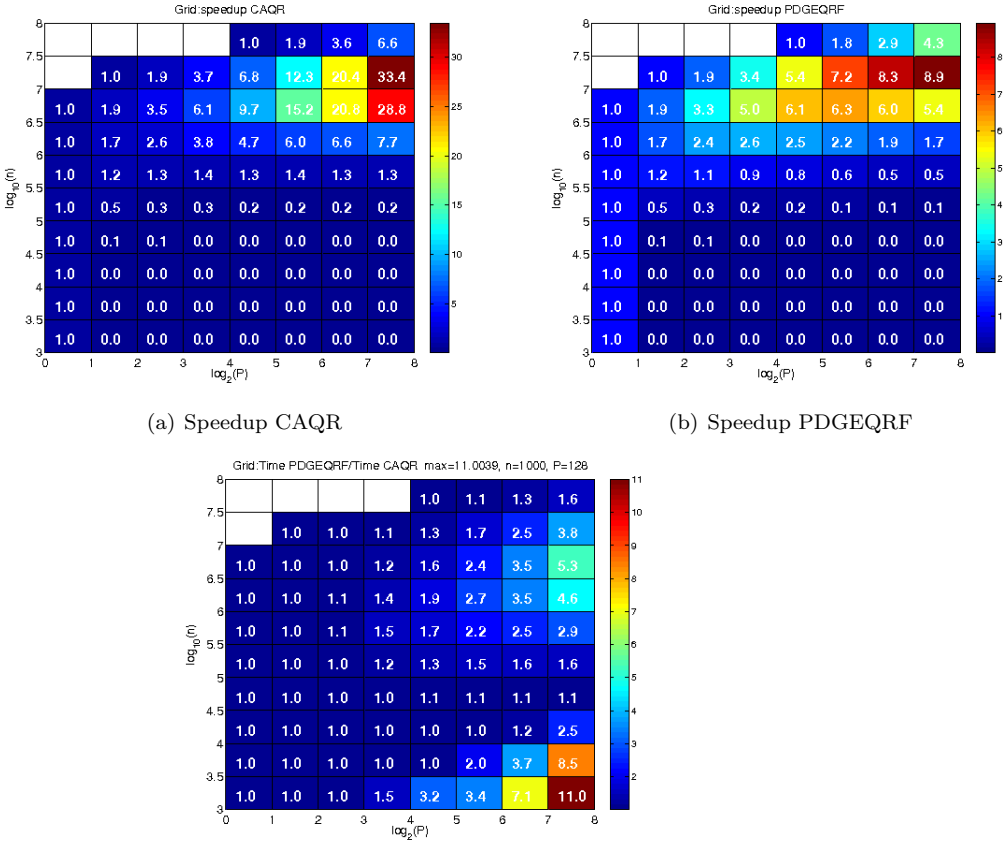
The best improvement is obtained for one of the largest matrix in our test set $m = n = 10^{6.5}$, where CAQR outperforms PDGEQRF by a factor of 5.3 on 128 processors. The speedup obtained by the best CAQR compared to the best PDGEQRF is larger than 4, and the best performance is obtained by CAQR on 128 processors, while the best performance of PDGEQRF is obtained on 32 processors.

CAQR is predicted to obtain reasonable speedups for large problems on the Grid, as displayed in Figure 14(a). For example, for $n = 10^7$ we note a speedup of 33.4 on 128 processors measured with respect to 2 processors. This represents an improvement of 1.6 over PDGEQRF. For the largest matrix in the test set, $n = 10^{7.5}$, we note a speedup of 6.6 on 128 processors, measured with respect to 16 processors. This is an improvement of 3.8 with respect to PDGEQRF.

As with the last model, 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 15, which shows that CAQR always is at least as fast as PDGEQRF, and often significantly faster (up to 3.8× faster in some cases).

16 Lower bounds on communication

In this section, we review known lower bounds on communication for parallel and sequential matrix-matrix multiplication of matrices stored in 1-D and 2-D block cyclic layouts. In Section 16.6, we will justify our conjecture that these bounds also apply to certain one-sided matrix factorizations (including LU and QR) of an $m \times n$ matrix A. We begin with 1-D layouts in Section 16.4, and



(a) Speedup CAQR

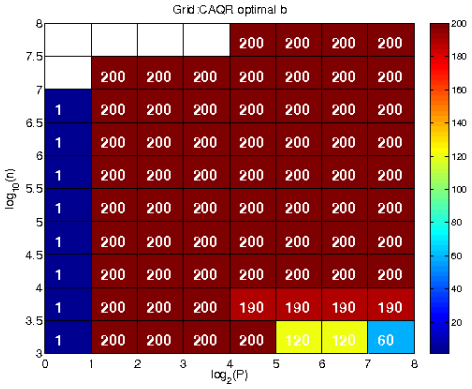
(b) Speedup PDGEQRF

(c) Comparison

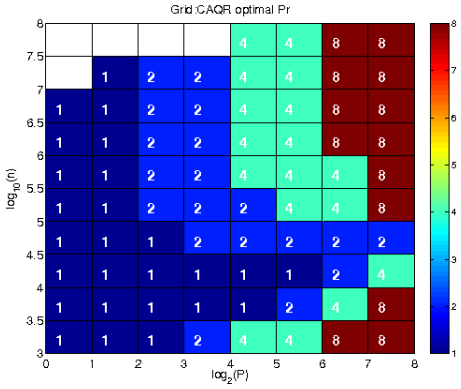
Figure 14: Performance prediction comparing CAQR and PDGEQRF on Grid.

$\log_{10} n$	Best $\log_2 P$ for PDGEQRF	CAQR speedup
6.0	3	1.4
6.5	5	2.4
7.0	7	3.8
7.5	7	1.6

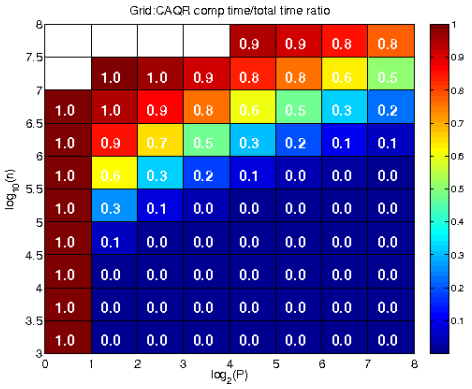
Table 15: Estimated runtime of PDGEQRF divided by estimated runtime of CAQR on a square $n \times n$ matrix, on the Grid platform, for those values of P (number of processors) for which PDGEQRF performs the best for that problem size.



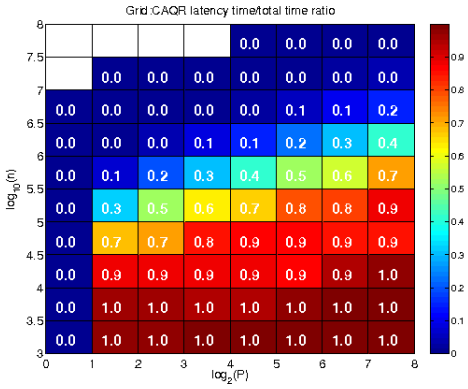
(a) Optimal b



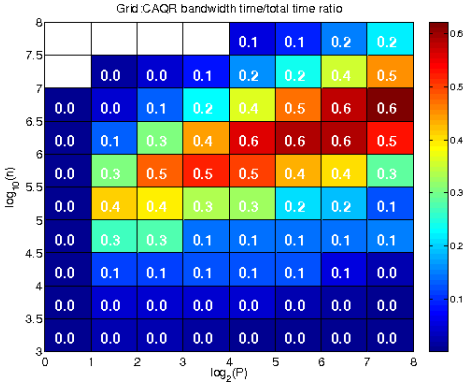
(b) Optimal P_r



(c) Fraction of time in computation

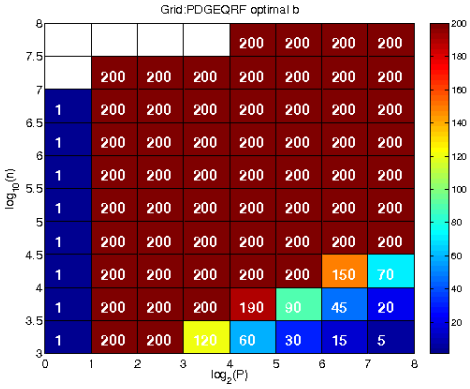


(d) Fraction of time in latency

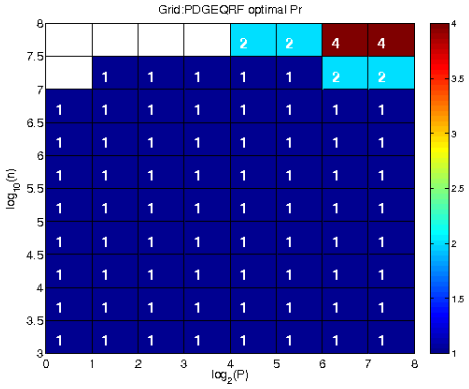


(e) Fraction of time in bandwidth

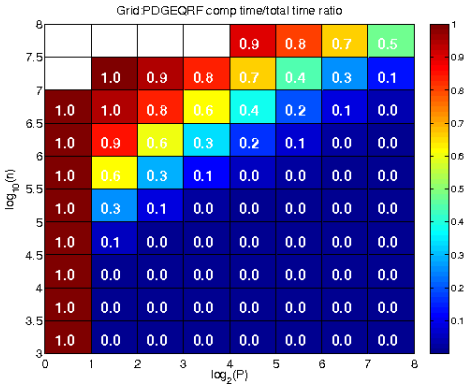
Figure 15: Performance prediction for CAQR on Grid.



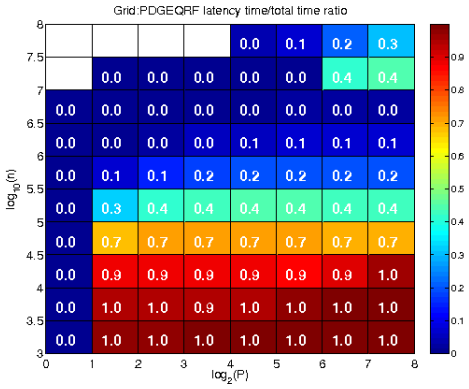
(a) Optimal b



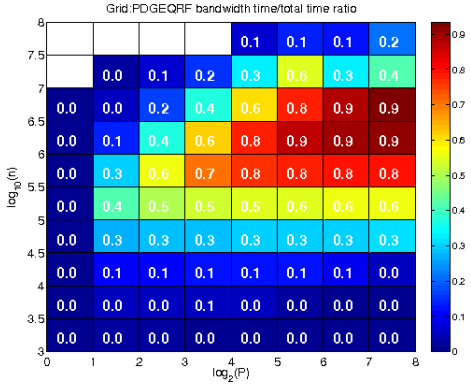
(b) Optimal P_r



(c) Fraction of time in computation



(d) Fraction of time in latency



(e) Fraction of time in bandwidth

Figure 16: Performance prediction for PDGEQRF on Grid.

follow with 2-D layouts, which are more extensively studied in the literature.

16.1 Assumptions

The matrix multiplication results only allow commutative and associative reorderings of the standard $O(n^3)$ algorithm:

$$C_{ij} = C_{ij} + \sum_k A_{ik} B_{kj}.$$

Alternate algorithms, such as Strassen's, are not permitted.

In the parallel case, we assume that each processor has a local memory of a fixed size W. The bounds govern the number of words that must be transferred between each processor's local memories, given any desired initial data distribution. We do not consider the cost of this initial distribution.

In the sequential case, the bounds govern the number of floating-point words transferred between slow and fast memory, given that fast memory has a fixed size W. We generally assume, unlike some authors we cite, that no part of the input matrix resides in fast memory at the start of the computation.

For the 1-D layouts, we can assume a block layout (only one block per processor) without loss of generality, as an implicit row permutation reduces block cyclic layouts to block layouts. We also generally assume that the blocks have at least as many rows as columns (i.e., $m/P \geq n$).

16.2 Prior work

Hong and Kung first developed a lower bound on the number of floating-point words moved between slow and fast memory for sequential matrix multiplication [26]. Others continued this work. Irony et al. simplified the proof and extended these bounds to include a continuum of parallel matrix distributions [25]. The latter work uses a proof technique which we conjecture could be extended to cover a variety of one-sided dense matrix factorizations, all of which require multiplying submatrices of the same magnitude as the matrix. These include both LU and QR. The bounds are known to hold for LU, as matrix-matrix multiplication can be reduced to LU using the following standard reduction:

$$\begin{pmatrix} I & 0 & -B \\ A & I & 0 \\ 0 & 0 & I \end{pmatrix} = \begin{pmatrix} I & & \\ A & I & \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} I & 0 & -B \\ & I & A \cdot B \\ & & I \end{pmatrix}. \tag{9}$$

Extending the bound to QR factorization is an open problem, as far as we know, but it is reasonable to conjecture that the same bound holds.

16.3 From bandwidth to latency

Given a lower bound on the number of words transferred, we can use the (local resp. fast) memory capacity W to derive a lower bound on the number of

messages, in both the parallel and sequential cases. This is because the best an algorithm can do in terms of the number of messages is to fill up its local resp. fast memory before it performs another message. So we can take any "inverse bandwidth" lower bound and divide it by W to get a "latency" lower bound.

16.4 1-D layouts

16.4.1 Sequential

In the sequential case, the $m \times n$ matrix A must be read from slow memory into fast memory at least once, if we assume that fast memory is empty at the start of the computation. Thus, the number of words transferred $B_{\text{seq,1-D}}(m, n, W)$ satisfies

$$B_{\text{seq,1-D}}(m, n, W) \geq mn. \tag{10}$$

(If we instead assume that fast memory may already contain part of the matrix, then the lower bound is $mn - W$.) The same reasoning as in Section 16.3 shows that the number of slow memory reads $L_{\text{seq,1-D}}(m, n, W)$ (the latency term) satisfies

$$L_{\text{seq,1-D}}(m, n, W) \geq mn/W. \tag{11}$$

16.4.2 Parallel

In the parallel case, we prefer for 1-D layouts to distinguish between the minimum number of messages per processor, and the number of messages along the critical path. For example, one can perform a reduction linearly, so that each processor only sends one message to the next processor. This requires $P - 1$ messages along the critical path, but only one message per processor. A lower bound on the minimum number of sends or receives performed by any processor is also a lower bound on the number of messages along the critical path. The latter is more difficult to analyze for 2-D layouts, so we only look at the critical path for 1-D layouts. By the usual argument that any nontrivial function of data distributed across P processors requires at least $\log_2 P$ messages to compute, the critical path length $C_{\text{1-D}}(m, n, P)$ satisfies

$$C_{\text{1-D}}(m, n, P) \geq \log_2 P. \tag{12}$$

This is also the number of messages required per processor along the critical path.

We make a conjecture about the bandwidth requirements of parallel Householder or Givens QR, based on the similar operation $C := A^T \cdot A$. Any one processor must communicate at least $n^2/2 + O(n)$ words with any other processor in order to complete this reduction operation. It seems reasonable that any one processor in a Householder or Givens QR factorization must transfer at least this many words between processors. This gives us a conjectured lower bound of

$$B_{\text{par,1-D}}(m, n, P) \geq n^2 + O(n) \tag{13}$$

on the number of words $B_{\text{par}}(m, n, P)$ sent and received by any one processor.

16.5 2-D layouts

16.5.1 Bandwidth

Irony et al. present the following lower bound on the number of words communicated between processors in parallel matrix-matrix multiplication, and between slow and fast memory in sequential matrix-matrix multiplication. Let A and B be the two square $n \times n$ matrices to be multiplied. If this operation uses $O(n^2/P)$ words of memory per processor, then at least one processor must send or receive

$$B_{\text{par,2-D}}(m, n, P) = \Omega\left(\frac{n^2}{\sqrt{P}}\right) \tag{14}$$

words. In the sequential case, if fast memory can contain W words,[6] then the number of floating-point words $B_{\text{seq,2-D}}(m, n, W)$ moved between slow and fast memory satisfies

$$B_{\text{seq,2-D}}(m, n, W) \geq \frac{n^3}{2\sqrt{2}W^{1/2}}, \tag{15}$$

if we assume that fast memory contains no part of the input matrix at the start of the computation.

16.5.2 Latency

In the sequential case, the total number of messages $L_{\text{seq,2-D}}(m, n, W)$ can be no better than the above bandwidth bound, divided by the fast memory capacity W. This gives a lower bound of

$$L_{\text{seq,2-D}}(m, n, W) \geq \frac{n^3}{2\sqrt{2}W^{1/2}} \tag{16}$$

messages. The same reasoning applies to the parallel case: since any one processor can hold at most W words, and one processor must send or receive $\Omega(n^2/\sqrt{P})$ words, then that processor must send

$$L_{\text{par,2-D}}(m, n, P) = \Omega(\sqrt{P}) \tag{17}$$

messages.

16.6 Extension to QR

Extending the above bounds to QR factorization is future work. It is straightforward to reduce matrix-matrix multiplication to Cholesky, but a reduction of Cholesky to QR is not immediately obvious. We could proceed as in Irony et al., by showing that the DAG of computations in the QR factorization implies the same communication bounds as in matrix-matrix multiplication. Alternately,

[6] Note that we use W to denote the fast memory size, whereas Irony et al. use W to denote that number of "elementary multiplications" performed per processor, and use M to denote the fast memory size.

we could try to demonstrate that any one-sided matrix factorization which uses a certain number of sufficiently large matrix-matrix multiplications internally must satisfy the matrix-matrix multiplication communication lower bounds. For the purposes of this paper, however, we assume in the 2-D case that the lower bounds on communication in matrix-matrix multiplication apply to QR up to a constant factor.

17 Lower bounds on parallelism

We base this paper on the premise that communication costs matter more than computation costs. Many authors developed parallel algorithms and bounds based on a PRAM model, which assumes that communication is essentially free. Their bounds are nevertheless of interest because they provide fundamental limits to the amount of parallelism that can be extracted, regardless of the cost of communication.

A number of authors have developed parallel QR factorization methods based on Givens rotations (see e.g., [41, 34, 8]). Givens rotations are a good model for a large class of QR factorizations, including those based on Householder reflections. This is because all such algorithms have a similar dataflow graph (see e.g., [31]), and are all based on orthogonal linear transformations (so they are numerically stable). Furthermore, these bounds also apply to methods that perform block Givens rotations, if we consider each block as an "element" of a block matrix.

17.1 Minimum critical path length

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 [7]. They assume any number of processors, any communication network, and any initial data distribution; in the extreme case, there many be mn processors, each with one element of the matrix. In their class of algorithms, a single step consists of computing one Givens rotation and zeroing out one matrix entry. Their first result concerns matrices for which $\lim_{m,n\to\infty} m/n^2 = 0$. This includes the case when $m = n$. Then the minimum critical path length is

$$2n + o(n). \tag{18}$$

A second complexity result is obtained for the case when $m \to \infty$ and n is fixed – that is, "tall skinny" matrices. Then, the minimum critical path length is

$$\log_2 m + (n-1)\log_2(\log_2 m) + o(\log_2(\log_2 m)). \tag{19}$$

The above bounds apply to 1-D and 2-D block (cyclic) layouts if we consider each "row" as a block row, and each "column" as a block column. One step in the computation involves computing one block Givens rotation and applying it (i.e., either updating or eliminating the current block). Then, Equation (18)

shows in the case of a square matrix that the critical path length is twice the number of block columns. (This makes sense, because the current panel must be factored, and the trailing matrix must be updated using the current panel factorization; these are two dependent steps.) In the case of a tall skinny matrix in a 1-D block row layout, Equation (19) shows that the critical path length is $\log_2(m/P)$, in which P is the number of processors. (The $(n-1)\log_2(\log_2 m)$ term does not contribute, because there is only one block column, so we can say that $n = 1$.)

17.2 Householder or Givens QR is P-complete

Leoncini et al. show that any QR factorization based on Householder reductions or Givens rotations is P-complete [31]. This means that if there exists an algorithm that can solve this problem using a number of processors polynomial in the number of matrix entries, in a number of steps polynomial in the logarithm of the number of matrix entries ("polylogarithmic"), then *all* tractable problems for a sequential computer (the set P) can be solved in parallel in polylogarithmic time, given a polynomial number of processors (the set NC). This "P equals NC" conclusion is considered unlikely, much as "P equals NP" is considered unlikely.

Note that one could compute the QR factorization of a matrix A by multiplying $A^T \cdot A$, computing the Cholesky factorization $R \cdot R^T$ of the result, and then performing $Q := AR^{-1}$. We describe this method ("CholeskyQR") in detail in Section 9. Csanky shows arithmetic NC algorithms for inverting matrices and solving linear systems, and matrix-matrix multiplication also has an arithmetic NC algorithm [9]. Thus, we could construct a version of CholeskyQR that is in arithmetic NC. However, this method is highly inaccurate in floating-point arithmetic. Not only is CholeskyQR itself inaccurate (see Section 10), Demmel observes that Csanky's arithmetic NC linear solver is so unstable that it loses all significant digits when inverting $3I_{n\times n}$ in IEEE 754 double-precision arithmetic, for $n \geq 60$ [12]. As far as we know, there exists no stable, practical QR factorization that is in arithmetic NC.

Appendix

A Structured local Householder QR flop counts

Here, we summarize floating-point operation counts for local structured Householder QR factorizations of various matrices of interest. We count operations for both the factorization, and for applying the resulting implicitly represented Q or Q^T factor to a dense matrix. We omit counts for BLAS 3 variants of structured Householder QR factorizations, as these variants require more floating-point operations. Presumably, the use of a BLAS 3 variant indicates that small constant factors and lower-order terms in the arithmetic operation count matter less to performance than exploiting locality.

A.1 General formula

A.1.1 Factorization

Algorithm 1 in Section 6.1 shows a column-by-column Householder QR factorization of the $qn \times n$ matrix of upper triangular $n \times n$ blocks, using structured Householder reflectors. We can generalize this to an $m \times n$ matrix A with a different nonzero pattern, as long as the trailing matrix updates do not create nonzeros below the diagonal in the trailing matrix. This is true for all the matrix structures encountered in the local QR factorizations in this report. A number of authors discuss how to predict fill in general sparse QR factorizations; see, for example, [18].

The factorization proceeds column-by-column, starting from the left. For each column, two operations are performed: computing the Householder reflector for that column, and updating the trailing matrix. The cost of computing the Householder vector of a column $A(j, j : m)$ is dominated by finding the norm of $A(j, j : m)$ and scaling it. If this part of the column contains k_j nonzeros, this comprises about $4k_j$ flops, not counting comparisons. We assume here that the factorization never creates nonzeros in the trailing matrix; a necessary (but not sufficient) condition on k_j is that it is nondecreasing in j.

The trailing matrix update involves applying a length $m-j+1$ Householder reflector, whose vector contains k_j nonzeros, to the $m-j+1 \times c_j$ trailing matrix C_j. The operation has the following form:

$$(I - \tau v_j v_j^T)C_j = C_j - v_j(\tau_j(v^T C_j)),$$

in which v_j is the vector associated with the Householder reflector. The first step $v_j^T C_j$ costs $2c_j k_j$ flops, as we do not need to compute with the zero elements of v_j. The result is a $1 \times c_j$ row vector and in general dense, so scaling it by τ_j costs c_j flops. The outer product with v_j then costs $c_j k_j$ flops, and finally updating the matrix C_j costs $c_j k_j$ flops (one for each nonzero in the outer product). The total is $4c_j k_j + c_j$.

When factoring an $m \times n$ matrix, $c_j = n-j$. The total number of arithmetic operations for the factorization is therefore

$$\sum_{j=1}^{n} 4(n-j)k_j + 4k_j + (n-j) \text{ flops.} \tag{A 1}$$

A.1.2 Applying implicit Q or Q^T factor

Applying Q or Q^T to an $m \times c$ matrix is like performing n trailing matrix updates, except that the trailing matrix size c stays constant. This gives us an arithmetic operation count of

$$\sum_{j=1}^{n} (4ck_j + 4k_j + c) \tag{A 2}$$

flops. In the case that the original $m \times n$ matrix A was fully dense, we have $k_j = m - j + 1$ and thus the arithmetic operation count is

$$4(c+1)mn - 2(c+1)n^2 + O(mn) + O(cn).$$

A.2 Special cases of interest

A.2.1 One block – sequential TSQR

The first step of sequential TSQR involves factoring a single $m/P \times n$ input block. This is the special case of a full matrix, whose flop count is

$$\frac{2mn^2}{P} - \frac{2n^3}{3} + \frac{n^2}{2} + O(mn/P).$$

A.2.2 Two blocks – sequential TSQR

For a $2m/P \times n$ factorization with the top $m/P \times n$ block upper triangular and the lower block full, we have $k_j = 1 + m/P$, and thus the flop count of the local QR factorization is

$$\frac{2mn^2}{P} + \frac{2mn}{P} + O(n^2).$$

For the case $m = n$, this specializes to $k_j = 1 + n$ and thus the flop count is

$$\frac{2n^3}{P} + \frac{5}{2}n^2 + O\left(\frac{2n^2}{P}\right).$$

Thus, the structured approach requires only about half the flops of standard Householder QR on the same $2m/P \times n$ matrix.

A.2.3 Two or more blocks – parallel TSQR

For two $m/P \times n$ upper triangular blocks grouped to form a $2m/P \times n$ matrix, we have $k_j = 1 + j$ and therefore the flop count is

$$\frac{2}{3}n^3 + O(n^2).$$

Note that the flop count is independent of m in this case. We can generalize this to some number $q \geq 2$ of the $m/P \times n$ upper triangular blocks, which is useful for performing TSQR with tree structures other than binary. Here, q is the branching factor of a node in the tree. In that case, we obtain obtain $k_j = 1 + (q-1)j$ and therefore the flop count is

$$\frac{2}{3}(q-1)n^3 - \frac{2}{3}n^3 + O(qn^2).$$

Without the structured Householder optimization, this would be $2mn^2/P - 2n^3/3 + O(n^2) + O(qmn/P)$ flops. In the case $m = n$, the optimization saves $2/3$ of the arithmetic operations required by the standard approach.

B Sequential TSQR performance model

B.1 Factorization

We now derive the performance model for sequential TSQR. Assume that the matrix is $m \times n$ and divided into P row blocks, with $m/P \geq n$. We assume read and write bandwidth are the same, and equal to $1/\beta$. For simplicity of analysis, we assume no overlapping of computation and communication; overlap could potentially provide another twofold speedup. Sequential TSQR first performs one local QR factorization of the topmost $m/P \times n$ block alone, at the cost of

- about $\frac{2mn^2}{P}$ flops,
- one read from secondary memory of size mn/P, and
- one write to secondary memory, containing both the implicitly represented Q factor (of size $mn/P - n(n+1)/2$), and the τ array (of size n).

Then it does $P-1$ local QR factorizations of two $m/P \times n$ blocks grouped into a $2m/P \times n$ block. In each of these local QR factorizations, the upper $m/P \times n$ block is upper triangular, and the lower block is a full matrix. This totals to

- about $\frac{2(P-1)mn^2}{P}$ flops,
- $P-1$ reads of size $\frac{mn}{P}$ each, and
- $P-1$ writes to secondary memory, each containing both an implicitly represented Q factor (of size $\frac{mn}{p}$), and the τ array (of size n).

The resulting modeled runtime is

$$T_{\text{Seq. TSQR}}(m,n,P) = \alpha 2P + \beta(2mn + nP - n(n+1)/2) + \gamma 2mn^2. \quad \text{(A 3)}$$

Suppose that fast memory can only hold W words of data for sequential TSQR. The TSQR factorization requires holding two blocks of the matrix in fast memory at once, and applying Q or Q^T requires holding three blocks in fast memory at once. Thus, we need to choose P such that $P \geq \frac{3mn}{W}$. Naturally we must assume $W \geq n$, and can also assume $W < mn$. We take $P = 3mn/W$ so as to maximize the block size, and obtain a modeled runtime of

$$T_{\text{Seq. TSQR}}(m,n,W) = \alpha\frac{6mn}{W} + \beta\left(2mn + \frac{3mn^2}{W} - \frac{n(n+1)}{2}\right) + \gamma 2mn^2. \quad \text{(A 4)}$$

If we can overlap communication and computation, and if we assume that we can always hide latency, then this formula tells us how much memory bandwidth is needed to keep the floating-point units busy. This would require the bandwidth term above to be no less than the floating-point term, and therefore, that

$$\beta\left(1 + \frac{3n}{2W} - \frac{n+1}{4m}\right) \geq \gamma n.$$

The terms inside the parentheses on the left side are all 1 or less than 1 in most cases. Comparing β with γ is an expression of *machine balance*; here we see that sequential TSQR only requires a modest machine balance of $\beta \geq \gamma n$ in order to cover bandwidth costs.

B.2 Applying Q or Q^T

The Q and Q^T cases are distinguished only by the order of operations. Suppose we are applying Q or Q^T to the dense $m \times c$ matrix C. The top block row of C receives both a one-block update (Q_0 resp. Q_0^T is applied to it) and a two-block update (involving both Q_0 and Q_1), whereas the remaining block rows of C each receive only a two-block update.

The number of arithmetic operations is the same (modulo lower-order terms) as the number of flops in sequential TSQR itself. However, the communication pattern is different. The $m \times c$ matrix C must be read from secondary memory, and the $m \times c$ result (QC resp. $Q^T C$) written to disk. Each of these reads or writes involves an $m/P \times n$ block. Furthermore, the Q factors and their associated τ arrays must be read from secondary memory. This makes the total latency cost $\alpha 3P$ and the total bandwidth cost $\beta(2mc + Pn(n+1)/2 + Pn)$.

C Sequential CAQR performance model

C.1 Conventions and notation

Sequential CAQR operates on an $m \times n$ matrix, stored in a $P_r \times P_c$ 2-D block layout. We do not model a fully general block cyclic layout, as it is only helpful in the parallel case for load balancing. We assume without loss of generality that P_r evenly divides m and P_c evenly divides n. The dimensions of a single block of the matrix are therefore $M \times N$, in which $M = m/P_r$ and $N = n/P_c$. Furthermore, let $P = P_r \cdot P_c$ be the number of blocks (not the number of processors, as in the parallel case – here we only use one processor). Our convention is to use capital letters for quantities related to blocks and the block layout, and lowercase letters for quantities related to the whole matrix independent of a particular layout.

We assume that fast memory has a capacity of W floating-point words for direct use by the algorithms in this section. We neglect the additional space needed for workspace and bookkeeping.

C.2 Factorization outline

Algorithms 10 and 11 outline left-looking resp. right-looking variants of the sequential CAQR factorization. We will analyze both in this section, so as to pick the optimal approach in terms of communication. Both require the same number of floating-point operations, so it suffices to pick one when counting those (we choose the right-looking algorithm, as it is more intuitive). Note that we cannot merely assume that the algorithms perform the same number of

floating-point operations as a standard sequential Householder QR factorization, because CAQR uses different panel factorization and update methods.

C.3 Fast memory usage

The factorization operates on at most three blocks in fast memory at one time. The blocks are of size $M \times N$, so the block size is $m/P_r \times n/P_c = mn/P$. We maximize fast memory usage by taking $W = 3mn/P$.

C.4 Total arithmetic operations

Sequential CAQR is not merely a reordering of standard Householder QR, but actually performs different operations. Thus, we expect the floating-point operation count to be different from standard Householder QR. Both left-looking and right-looking sequential CAQR perform the same floating-point operations, just in a different order, so it suffices to count flops for the left-looking factorization. The total flop count is

$$\sum_{J=1}^{P_c} \left(2(m - J * (m/P_r) + m/P_r)(n/P_c)^2 + 2(n/P_c)^3(P_c - J) + 2(m/P_r)(n/P_c)^2(2(P_r - J + 1)(P_c - J))\right),$$

which sums to

$$4mn^2 - \frac{n^3}{P_c^2} - \frac{2mn^2}{P_c} + \frac{n^3}{P_c} + \frac{3mn^2}{P_r} - \frac{mn^2}{P} - \frac{2mn^2 P_c}{P_r}. \qquad \text{(A 5)}$$

In the case that $m = n$ (i.e., the matrix is square) and $P_r = P_c = \sqrt{P}$, this reduces to

$$2m^3 + \frac{2m^3}{\sqrt{P}} \pm O\left(\frac{2m^3}{P}\right)$$

flops. The standard Householder QR algorithm requires $2m^3$ flops. If we let fast memory capacity be W floating-point words, and assume that at most three $M \times M$ blocks of the matrix must fit in fast memory at once, then we can take $P = 3m^2/W$. We then see that sequential CAQR performs about

$$1 + \frac{\sqrt{W}}{\sqrt{3}m}$$

times more flops than the standard method. We must have $\sqrt{W} \leq m$ and thus sequential CAQR does no more than about 60% more flops than the usual Householder QR factorization. Indeed, often fast memory is large enough to hold more than one column or row of the matrix, and thus $\sqrt{W} \leq \sqrt{m}$.

C.5 Communication requirements

Here, we count the total volume and number of block read and write operations in both the left- and right-looking versions of sequential CAQR. Making a fair

count requires defining these two variants more precisely. For the right-looking algorithm, one can choose to sweep either in row order or in column order over the trailing matrix. The column-order option saves loads of the input matrix (as most input blocks of the trailing matrix are updated twice), whereas the row-order option saves loads of the current panel. Both save the same number of block I/O operations. We choose the column-order option in order to save bandwidth, as the input matrix blocks are larger than the Q factor blocks. Algorithm 12 shows the column-order right-looking factorization in more detail.

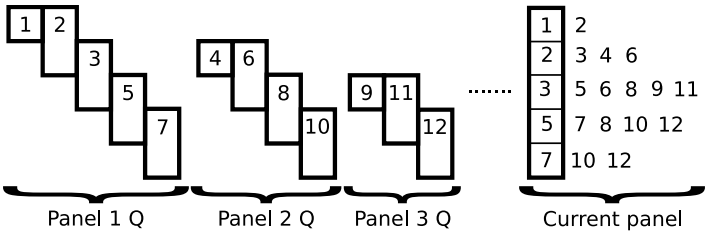


Figure 17: A possible reordering of the current panel updates in the left-looking sequential QR factorization. A number K on the left represents a block of a panel's Q factor, and a number on the right (attached to the "current panel") represents an update of a current panel block by the correspondingly numbered Q factor block. The ordering of the numbers is one possible ordering of updates. Note that the Q panels must be applied to the current panel in a way that respects left-to-right dependencies on a horizontal level. For example, the update represented by block 2 on the left must precede updates by blocks 3, 4, and 6. Blocks with no components on the same horizontal level are independent and can be reordered freely with respect to one another.

Left-looking sequential CAQR uses all but the last block of the current panel twice during updates. By expanding the number of current panel blocks held in fast memory and reordering the updates, one can increase reuse of current panel blocks. Figure 17 illustrates one possible reordering. We retain the usual ordering, however, so as to restrict the number of blocks in fast memory to three at once. Algorithm 13 shows the left-looking factorization in more detail.

For standard Householder QR, a left-looking factorization saves bandwidth in the out-of-DRAM regime. D'Azevedo et al. took the left-looking approach in their out-of-DRAM ScaLAPACK QR code, for example [11]. Sequential CAQR follows a different communication pattern, however, so we need not expect that a left-looking factorization saves communication. In fact, both our right-looking and left-looking approaches have about the same bandwidth requirements.

C.5.1 Right-looking communication

Communication volume The column-order right-looking algorithm requires the following communication volume:

$$\sum_{J=1}^{P_c}\left(2(P_r-J+1)MN-(P_r-J+1)N(N-3)/2+N+\sum_{K=J+1}^{P_c}\left(3MN-N(N-1)/2+\sum_{L=J+1}^{P_r}(3MN-N(\right.\right.$$

If we compute this sum and substitute $M \to m/P_r$, $N \to n/P_c$, $P_r- > P/P_c$, and $P- > 3mn/W$, we obtain a mess of positive and negative terms. We can neglect the negative terms in terms of asymptotic performance, since they come the upper triangle of the matrix. The positive terms in the sum are

$$\frac{7n}{4}+\frac{mn}{2}+\frac{3mnP_c}{2}+\frac{n^2P_c}{12}+\frac{9mn^2}{4W}+\frac{9mn^2}{4P_cW}+\frac{P_c^2W}{6}.$$

Of these, the most significant term is

$$3mnP_c/2.$$

It subsumes $n^2P_c/2$, as long as $m \geq n$. We see from this that as P_c grows, so do the bandwidth requirements. In the case of a square matrix on a square block layout (i.e., $P_c = \sqrt{P}$ and $m = n$), the most significant term in the whole sum (including negative terms) is

$$\frac{n^3}{4\sqrt{3}\sqrt{W}}. \tag{A 6}$$

Thus, sequential CAQR achieves the lower bound of $\Omega(n^3/\sqrt{W})$ conjectured in Section 16.

Number of block transfers The column-oriented right-looking algorithm reads and writes a total of

$$\sum_{J=1}^{P_c}\left(2(P_r-J+1)+\sum_{K=J+1}^{P_c}\left(3+\sum_{L=J+1}^{P_r}(3)\right)\right).$$

blocks between fast and slow memory. This sums to

$$\frac{P_c^2}{2}-\frac{P_c^3}{2}+\frac{P}{2}+3\frac{P_c^2P_r}{2}=\frac{P}{2}+\frac{3PP_c}{2}+\frac{P_c^2}{2}-\frac{P_c^3}{2}$$

block transfers. If we substitute $M \to m/P_r$, $N \to n/P_c$, $P_r- > P/P_c$, and $P- > 3mn/W$, we obtain

$$\frac{3mn}{2W}+\frac{9mnP_c}{2W}+\frac{P_c^2}{2}-\frac{P_c^3}{2}.$$

The most significant term is

$$\frac{9mnP_c}{2W}.$$

In the case that $m = n$ and $P_c = P$, all the positive terms together reduce to

$$\frac{3\sqrt{3}n^3}{2W^{3/2}} + \frac{3n^2}{W},$$

which shows that left-looking sequential CAQR satisfies the lower bound of $\Omega(n^3/W^{3/2})$ conjectured in Section 16.

C.5.2 Left-looking communication

The left-looking version of sequential CAQR has the following communication volume:

$$\sum_{J=1}^{P_c} \left(2(P_r - J + 1)MN - (P_r - J + 1)\frac{N(N-1)}{2} - N^2 + \sum_{K=1}^{J-1} \left(-N^2 + \sum_{L=K+1}^{P_r} \left(3MN - \frac{N(N-3)}{2} \right) \right) \right).$$

If we compute this sum and substitute $M \to m/P_r$, $N \to n/P_c$, $P_r -> P/P_c$, and $P -> 3mn/W$, we obtain a mess of positive and negative terms. We can neglect the negative terms in terms of asymptotic performance, since they come the upper triangle of the matrix. The positive terms in the sum are

$$n/2 + mn/2 + 3mnP_c/2 + n^2P_c/2 + 9mn^2/4W + P_cW.$$

Of these, the most significant term is

$$3mnP_c/2.$$

It subsumes $n^2P_c/2$, as long as $m \geq n$. We see from this that as P_c grows, so do the bandwidth requirements. In the case of a square matrix on a square block layout (i.e., $P_c = \sqrt{P}$ and $m = n$), we achieve the lower bound of $\Omega(n^3/\sqrt{W})$ conjectured in Section 16.

Number of block transfers Left-looking sequential CAQR performs the following number of block transfers between fast and slow memory:

$$\sum_{J=1}^{P_c} \left(2(P_r - J + 1) + \sum_{K=1}^{J-1} \left(\sum_{L=K}^{P_r} 3 \right) \right).$$

This sums to

$$\frac{3mn}{2W} + \frac{9mnP_c}{2W} + \frac{P_c^2}{2} - \frac{P_c^3}{2}$$

block transfers, which is exactly the same as in the column-oriented right-looking variant.

C.6 Applying Q or Q^T

Applying the Q or Q^T factor from sequential CAQR has almost the same cost as factoring the matrix. One need only subtract out the cost of the panel factorizations.

D Parallel TSQR performance model

We now derive the performance model for parallel TSQR on a binary tree of P processors. We restrict our performance analysis to the block row, reduction based Algorithm 3. The all-reduction-based version has the same number of flops on the critical path (the root process of the reduction tree), but it requires $2q$ parallel messages per level of the tree on a q-ary tree, instead of just $q-1$ parallel messages to the parent node at that level in the case of a reduction. For simplicity we assume that the number of processors P is a power of 2: $P = 2^{L-1}$ with $L = \log_2 p$.

Section 6 describes how to optimize the local QR factorizations, and Appendix A counts the number of arithmetic operations for these optimized local factorizations. We use the structured Householder reflectors discussed in Section 6.1, but do not include the BLAS 3 reformulation of structured Householder reflectors in Section 6.2.

D.1 Factorization

A parallel TSQR factorization on a binary reduction tree performs the following computations along the critical path: One local QR factorization of a fully dense $m/P \times n$ matrix, at cost $2mn^2/P - \frac{n^3}{3} + O(mn/P)$ flops, and and $\log(P)$ factorizations of a $2n \times n$ matrix consisting of two $n \times n$ upper triangular matrices, at cost $2n^3/3 + O(n^2)$ flops per factorization. The factorization requires $\log(P)$ messages, each of size $n(n+1)/2$.

E Parallel CAQR performance model

In this section, we model the performance the parallel CAQR algorithm described in Section 13. Parallel CAQR operates on an $m \times n$ matrix A, stored in a $P_r \times P_c$ 2-D block layout. We do not model a fully general block cyclic layout here, for simplicity. Assume that P_r evenly divides m and P_c evenly divides n. We introduce some notation in this section to simplify counting: let the dimensions of a single block of the matrix be $M \times N$, so that $M = m/P_r$ and $N = n/P_c$. Our convention here is to use capital letters for quantities related to blocks, and lowercase letters for quantities related to the whole matrix independent of a particular layout.

E.1 Factorization

First, we count the number of floating point arithmetic operations that CAQR performs along the critical path. We compute first the cost of computing the QR factorization using Householder transformations of a $m \times n$ matrix A (using DGEQR2). The cost of computing the jth Householder vector is given by the cost of computing its Euclidian norm and then by scaling the vector. This involves $3(m-j+1)$ flops and $(m-j+1)$ divides. The cost of updating the

trailing $A(j:m, j+1:n)$ matrix by $I - \tau v_j v_j^T$ is $4(n-j)(m-j+1)$. The total number of flops is:

$$3\sum_{j=1}^{n}(m-j+1) + 4\sum_{j=1}^{n-1}(n-j)(m-j+1) = 2mn^2 - \frac{2n^3}{3} + mn + \frac{n^2}{2} + \frac{n}{3} \approx 2mn^2 - \frac{2n^3}{3}$$

The total number of divides is around $mn - n^2/2$.

The Householder update of a matrix $(I - YT^TY^T)C$, where Y is a $m \times n$ unit lower trapezoidal matrix of Householder vectors and C is a $m \times q$ matrix, can be expressed as:

$$C = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \left(I - \begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix} \cdot \ T^T \ \cdot \begin{pmatrix} Y_0 \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C_0 \\ C_1 \end{pmatrix}$$

in which Y_0 is a $n \times n$ unit lower triangular matrix and Y_1 is a rectangular matrix. The total number of flops is around $qn(4m - n - 1) \approx qn(4m - n)$. We described in Section 6.4 how to perform the trailing matrix update. The breakdown of the number of flops in each step is:

- $W = Y_0^T C_0 \rightarrow n(n-1)q$ flops.
- $W = W + Y_1^T C_1 \rightarrow 2n(m-n)q$ flops.
- $W = T^T W \rightarrow n^2 q$ flops.
- $C_0 = C_0 - Y_0 W \rightarrow n^2 q$ flops.
- $C_1 = C_1 - Y_1 W \rightarrow 2n(m-n)q$ flops.

We consider now the computation of the upper triangular matrix T used in the $(I - YTY^T)$ representation of the Householder vectors (DLARFT routine). This consists of n transformations of the form $(I - \tau v_i v_i^T)$. Consider Y, a $m \times n$ unit lower trapezoidal matrix of Householder vectors. The matrix T is an upper triangular matrix of dimension $n \times n$ that is obtained in n steps. At step j, the first $j-1$ columns of T are formed. The j-th column is obtained as follows:

$$T(1:j, 1:j) = \begin{pmatrix} T(1:j-1, 1:j-1) & -\tau T(1:j-1, 1:j-1) Y^T(:, 1:j-1) v_j) \\ & \tau \end{pmatrix}$$

in which v_j is the jth Householder vector of length $m-j+1$. This is obtained by computing first $w = -\tau Y^T(:, 1:j-1)v_j$ (matrix vector multiply of $2(j-1)(m-j+1)$ flops) followed by the computation $T(1:j-1, j) = T(1:j-1, 1:j-1)w$ (triangular matrix vector multiplication of $(j-1)^2$ flops). The total cost of forming T is:

$$mn^2 - \frac{n^3}{3} - mn + \frac{n^2}{2} - \frac{n}{6} \approx mn^2 - \frac{n^3}{3}$$

The new QR factorization algorithm also performs Householder updates of the form

$$C = \begin{pmatrix} C_0 \\ C_1 \end{pmatrix} = \left(I - \begin{pmatrix} I \\ Y_1 \end{pmatrix} \cdot \quad T^T \quad \cdot \begin{pmatrix} I \\ Y_1 \end{pmatrix}^T \right) \begin{pmatrix} C_0 \\ C_1 \end{pmatrix}$$

in which Y_1 is a $n \times n$ upper triangular matrix and C is a $2n \times q$ matrix. The total number of flops is $3n^2q + 6nq$. The following outlines the number of floating-point operations corresponding to each operation performed during this update:

- $W = Y_1^T C_1 \rightarrow n(n+1)q$ flops.
- $W = W + C_0 \rightarrow nq$ flops.
- $W = T^T W \rightarrow n(n+1)q$ flops.
- $C_0 = C_0 - W \rightarrow nq$ flops.
- $C_1 = C_1 - Y_1 W \rightarrow n(n+2)q$ flops.

Forming the upper triangular matrix T used in the above Householder update corresponds now to computing $-\tau T(1:j-1, 1:j-1) Y_1^T(1:j-1, 1:j-1) v_j(n+1:n+j)$. v_j is the jth Householder vector composed of 1 in position j and nonzeros in positions $n+1, \ldots n+j+1$. First $w = -\tau Y_1^T(1:j-1, 1:j-1) v_j(n+1:2n)$ is computed (triangular matrix vector multiply of $j(j-1)$ flops), followed by $T(1:j-1, j) = T(1:j-1, 1:j-1) w$ (triangular matrix vector multiplication of $(j-1)^2$ flops). The total number of flops is

$$\sum_{j=1}^{n} j(j-1) + \sum_{j=1}^{n} (j-1)^2 \approx 2\frac{n^3}{3} \tag{A 7}$$

We are now ready to estimate the time of CAQR.

1. The column of processes that holds panel j computes a TSQR factorization of this panel. The Householder vectors are stored in a tree as described in Section 8.

$$\left[\frac{2b^2 m_j}{P_r} + \frac{2b^3}{3} (\log P_r - 1) \right] \gamma + \left[\frac{m_j b}{P_r} + \frac{b^2}{2} (\log P_r - 1) \right] \gamma_d + \log P_r \alpha + \frac{b^2}{2} \log P_r \beta \tag{A 8}$$

2. Each processor p that belongs to the column of processes holding panel j broadcasts along its row of processors the $m_j/P_r \times b$ rectangular matrix that holds the two sets of Householder vectors. Processor p also broadcasts two arrays of size b each, containing the Householder factors τ_p.

$$2 \log P_c \alpha + \left(\frac{m_j b}{P_r} + 2b \right) \log P_c \beta \tag{A 9}$$

3. Each processor in the same row template as processor p, $0 \leq i < P_r$, forms T_{p0} (first two terms in the number of flops) and updates its local trailing matrix C using T_{p0} and Y_{p0} (last term in the number of flops). (This computation involves all processors and there is no communication.)

$$\left[b^2\frac{m_j}{P_r} - \frac{b^3}{3} + b\frac{n_j - b}{P_c}\left(4\frac{m_j}{P_r} - b\right)\right]\gamma \tag{A 10}$$

4. **for** $k = 1$ to $\log P_r$ **do**

   Processors that lie in the same row as processor p, where $0 \leq p < P_r$ equals $first\_proc(p,k)$ or $target\_first\_proc(p,k)$ perform:

   (a) Processors in the same template row as $target\_first\_proc(p,k)$ form locally $T_{level(p,k),k}$. They also compute local 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 (4b).

   $$\left[\frac{2b^3}{3} + b(b+1)\frac{n_j - b}{P_c}\right]\gamma \tag{A 11}$$

   (b) Each processor in the same row of the grid as $first\_proc(p,k)$ sends to the processor in the same column and belonging to the row of $target\_first\_proc(p,k)$ the local pieces of $C_{first\_proc(p,k)}$.

   $$\alpha + \frac{b(n_j - b)}{P_c}\beta \tag{A 12}$$

   (c) Processors in the same template row as $target\_first\_proc(p,k)$ compute local pieces of
   $W = T^T_{level(p,k),k}\left(C_{first\_proc(p,k)} + W\right)$.

   $$b(b+2)\frac{n_j - b}{P_c}\gamma \tag{A 13}$$

   (d) Each processor in the same template row as $target\_first\_proc(p,k)$ sends to the processor in the same column and belonging to the row template of $first\_proc(p,k)$ the local pieces of W.

   $$\alpha + \frac{b(n_j - b)}{P_c}\beta \tag{A 14}$$

   (e) Processors in the same template row as $first\_proc(p,k)$ complete locally the rank-b update $C_{first\_proc(p,k)} = C_{first\_proc(p,k)} - W$ (number of flops in Equation A 15). Processors in the same template row as $target\_first\_proc(p,k)$ complete locally the rank-b update $C_{target\_first\_proc(p,k)} = C_{target\_first\_proc(p,k)} - Y_{level(p,k),k} W$ (number of flops in Equation A 16). The latter computation is overlapped with the communication in (4d).

$$b\frac{n_j - b}{P_c}\gamma \tag{A 15}$$

$$b(b+2)\frac{n_j - b}{P_c}\gamma \tag{A 16}$$

end for

We can express the total computation time over a rectangular grid of processors $T(m, n, P_r, P_c)$ as a sum over the number of iterations of the previously presented steps. The number of messages is $n/b(3\log P_r + 2\log P_c)$. The volume of communication is:

$$\sum_{j=1}^{n/b}\left(\frac{b^2}{2}\log P_r + \frac{m_j b}{P_r}\log P_c + 2b\log P_c + \frac{2b(n_j - b)}{P_c}\log P_r\right) = \left(\frac{nb}{2} + \frac{n^2}{P_c}\right)\log P_r + \left(2n + \frac{mn - n^2/2}{P_r}\right)\log P_c$$

The total number of flops corresponding to each step is given by:

$$\begin{array}{lcl}
\sum_{j=1}^{n/b}(A\ 8) & \approx & \frac{2nmb - n^2b + nb^2}{P_r} + \frac{2b^2n}{3}(\log P_r - 1) \\
\sum_{j=1}^{n/b}(A\ 10) & \approx & \frac{1}{P}\left(2mn^2 - \frac{2}{3}n^3\right) + \frac{1}{P_r}\left(mnb + \frac{nb^2}{2} - \frac{n^2b}{2}\right) + \frac{n^2b}{2P_c} - \frac{b^2n}{3} \\
\sum_{j=1}^{n/b}(A\ 11) & \approx & \left(\frac{2b^2n}{3} + \frac{n^2(b+1)}{2P_c}\right)\log P_r \\
\sum_{j=1}^{n/b}(A\ 13) & \approx & \frac{n^2(b+2)}{2P_c}\log P_r \\
\sum_{j=1}^{n/b}(A\ 15) & \approx & \frac{n^2}{2P_c}\log P_r \qquad \sum_{j=1}^{n/b}(A\ 16) \approx \frac{n^2(b+2)}{2P_c}\log P_r
\end{array}$$

The total computation time of parallel CAQR can be estimated as:

$$\begin{array}{rcl}
T_{\text{Par. CAQR}}(m, n, P_r, P_c) & = & \left[\frac{2}{3P}n^2(3m - n) + \frac{n^2b}{2P_c} + \frac{1}{P_r}3bn\left(m - \frac{n}{2}\right) + \left(\frac{4b^2n}{3} + \frac{n^2(3b+5)}{2P_c}\right)\log P_r - b^2n\right]\gamma + \\
& & + \left[\frac{1}{P_r}\left(mn - \frac{n^2}{2}\right) + \frac{nb}{2}(\log P_r - 1)\right]\gamma_d + \\
& & + \log P_r\left[\frac{3n}{b}\alpha + \left(\frac{nb}{2} + \frac{n^2}{P_c}\right)\beta\right] + \\
& & + \log P_c\left[\frac{2n}{b}\alpha + \left(\frac{1}{P_r}\left(mn - \frac{n^2}{2}\right) + 2n\right)\beta\right]
\end{array} \tag{A 17}$$

F Sequential left-looking panel factorization

F.1 A common communication pattern

Many sequential QR factorization algorithms of interest share a common communication pattern. They are left-looking panel factorizations that keep two panels in fast memory: a left panel of width b, and the current panel being updated, of width c. For each current panel, the methods sweep from left to

right over the left panels, updating the current panel with each left panel in turn. They then factor the current panel and continue. If we model this communication pattern once, we can then get models for all such methods, just by filling in floating-point operation counts for each. We can do this both for the factorization and for applying the Q or Q^T factor, since the method for applying a Q or Q^T factor looks much like the factorization used to compute that factor.

We base this communication model on the out-of-DRAM QR factorization PFDGEQRF described in LAPACK Working Note #118 [11], in which standard Householder QR is used for the panel factorization. However, other "column-by-column" methods, such as Gram-Schmidt orthogonalization, may be used for the panel factorization without changing the model. This is because both the left panel and the current panel are in fast memory, so neither the current panel update nor the current panel factorization contribute to communication. (Hence, this unified model only applies in the sequential case, unless each processor contains an entire panel.)

Algorithm 14 gives an outline of the communication pattern for the factorization, without cleanup for cases in which c does not evenly divide n or b does not evenly divide the current column index minus one. This cleanup code does not contribute significantly to our performance model. The algorithm transfers a total of about

$$2mn + \frac{mn^2}{2c^2}$$

floating-point words between slow and fast memory. Thus, keeping b fixed and making c as large as possible optimizes for a bandwidth-limited regime. Making $b > 1$ can also provide some BLAS 3 benefits, but this does not contribute to our performance model.

Algorithm 15 outlines the communication pattern for applying the full Q^T factor to an $n \times r$ dense matrix. We assume this matrix might be large enough not to fit entirely in fast memory. Applying Q instead of Q^T merely changes the order of iteration over the Householder reflectors, and does not change the performance model, so we can restrict ourselves to applying Q^T without loss of generality.

F.2 Fast memory capacity

We assume that fast memory has a capacity of W floating-point words for direct use by the algorithms in this section. We neglect the additional space needed for workspace and bookkeeping.

F.3 Factorization

F.3.1 Fast memory usage

The factorization uses at most $(b + c)m$ words of fast memory at once. In order to maximize the amount of fast memory used, we take $(b + c)m = W$. The parameter b is typically a small constant chosen to increase the BLAS

3 efficiency, whereas one generally chooses c to fill the available fast memory. Thus, we may simplify the analysis by taking $b = 1$ and $c \approx W/m$. Note that we must have $W \geq 2m$ in order to be able to factor the matrix at all.

An important quantity to consider is mn/W. This is the theoretical lower bound on the number of reads from slow memory (it is a latency term). It also bounds from below the number of slow memory writes, assuming that we use the usual representation of the Q factor as a collection of dense Householder reflectors, and the usual representation of the R factor as a dense upper triangular matrix.

F.3.2 Number of words transferred

Algorithm 14 transfers about

$$\sum_{j=1}^{\frac{n}{c}} \left(2c\left(m - cj + 1\right) + \sum_{k=1}^{\frac{j-1}{b}} b\left(m - bk + 1\right) - \frac{bk(bk+1)}{2} \right) \tag{A 18}$$

floating-point words between slow and fast memory. If we take $b = 1$ and assume $m \geq n$, the most significant terms of the sum are

$$2mn + \frac{mn^2}{2c^2}.$$

The lower bound on the number of words read and written is $2mn$ (the entire matrix must be read and written). If we let $c = W/m$ (as an approximation for the case $b = 1$), we get a total of

$$2mn + \frac{m^3n^2}{W^2}$$

words transferred between slow and fast memory. Since $W \geq 2m$, we must read $2mn + mn^2/4$ words in the worst case. Note that $2mn$ is a lower bound (the matrix must be read from slow memory, and the results written back to slow memory).

F.3.3 Number of slow memory accesses

The total number of slow memory accesses performed by Algorithm 14 is about

$$\sum_{j=1}^{\frac{n}{c}} \left(2 + \sum_{k=1}^{\frac{j-1}{b}} 1 \right) = 2\frac{n}{c} - \frac{n}{2bc} + \frac{n^2}{2bc^2}. \tag{A 19}$$

If we take $b = 1$ and let $c = W/m$, this comes out to

$$\frac{3nW}{m} + \frac{m^2n^2}{2W^2}$$

accesses. We can see what this means by factoring out mn/W, which is the theoretical lower bound on the number of reads from slow memory (a latency term):

$$\frac{mn}{W}\left(\frac{3W^2}{m^2}+\frac{mn}{2W}\right)=\Theta\left(\left(\frac{mn}{W}\right)^2\right).$$

This shows that any sequential QR factorization with the above communication pattern is a factor of mn/W away from being optimal with respect to the number of slow memory reads.

F.4 Applying Q^T

F.4.1 Fast memory usage

Applying Q or Q^T uses at most $bm+cn$ words of fast memory at once. In order to maximize the amount of fast memory used, we take $bm+cn=W$. Note that this is different from the factorization, in case $m \neq n$. A lower bound on the number of reads from slow memory (a latency term) is $mn/W + nr/W$.

F.4.2 Number of words transferred

When we read the Q factor, we only have to read the lower trapezoid of the matrix Q into fast memory. If we neglect this and assume we have to read all mn entries of Q, then the algorithm must transfer about

$$\sum_{j=1}^{r/c}\left(2cn+\sum_{k=1}^{cj/b}bm\right)\approx\frac{mr^2}{2c}+2nr$$

words between slow and fast memory. The number of loads saved by not reading the upper triangle of Q is about

$$\sum_{j=1}^{r/c}\sum_{k=1}^{cj/b}\frac{b^2}{2}=\frac{r^2}{4c}$$

words, which is a lower-order term that we will neglect. The b parameter does not contribute significantly, so we can set $b=1$ and $c \approx W/m$, to get a total of about

$$\frac{mn}{W}\cdot\frac{r^2}{2}$$

words transferred between slow and fast memory.

F.4.3 Number of slow memory accesses

The total number of data transfers between slow and fast memory is about

$$\sum_{j=1}^{r/c}\left(2+\sum_{k=1}^{cj/b}1\right)=\frac{2r}{c}+\frac{r^2}{b}.$$

We see that b, the panel width of Q, affects latency much more than c, the panel width of B. In contrast, c affects bandwidth requirements much more than b. If we set $b = 1$ and $c \approx W/m$, we get

$$\frac{2mr}{W} + r^2.$$

G Sequential Householder QR

G.1 Factorization

LAPACK Working Note #118 describes an out-of-DRAM QR factorization PFDGEQRF, which is implemented as an extension of ScaLAPACK [11]. It uses ScaLAPACK's parallel QR factorization PDGEQRF to perform the current panel factorization in DRAM. Thus, it is able to exploit parallelism, either on a single shared-memory node or within a cluster (assuming a shared filesystem). It can also take advantage of the features of parallel filesystems for block reads and writes. In our performance analysis here, we assume that all operations in fast memory run sequentially, and also that the connection to slow memory is sequential.

PFDGEQRF is a left-looking method, as usual with out-of-DRAM algorithms. The code keeps two panels in fast memory: a left panel of fixed width b, and the current panel being factored, whose width c can expand to fill the available memory. Algorithm 16 gives an outline of the code, without cleanup for cases in which c does not evenly divide n or b does not evenly divide the current column index minus one. Algorithm 17 near the end of this section illustrates this "border cleanup" in detail, though we do not need this level of detail in order to model the performance of PFDGEQRF.

This sequential Householder QR code has the communication pattern described in Section F. This means that the latency and bandwidth performance models can be found in that section, and thus we need only count floating-point operations. In fact, the flop count is exactly the same as the usual sequential Householder QR algorithm, namely

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

PFDGEQRF merely performs the flops in a different order. Choosing c as large as possible optimizes bandwidth requirements, whereas increasing b reduces the number of reads.

G.2 Applying Q or Q^T

Section F.4 describes the communication pattern of applying the full Q or Q^T factor computed using the above factorization to an $n \times r$ matrix B. The floating-point operation count is about $2mnr$.

References

[1] N. N. Abdelmalek, *Round off error analysis for Gram–Schmidt method and solution of linear least squares problems*, BIT, 11 (1971), pp. 345–368.

[2] 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. Softw., 29 (2003), pp. 337–348.

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

[4] Å. Björck, *Solving linear least squares problems by Gram-Schmidt orthogonalization*, BIT, 7 (1967), pp. 1–21.

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

[6] J. Choi, J. J. Dongarra, S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley, *LAWN 80: the design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines*, LAPACK Working Note UT-CS-94-246, Oak Ridge National Laboratory, Sept. 1994.

[7] M. Cosnard, J.-M. Muller, and Y. Robert, *Parallel QR Decomposition of a Rectangular Matrix*, Numer. Math., 48 (1986), pp. 239–249.

[8] M. Cosnard and Y. Robert, *Complexite de la factorisation QR en parallele*, C.R. Acad. Sci., 297 (1983), pp. 549–552.

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

[10] R. D. da Cunha, D. Becker, and J. C. Patterson, *New parallel (rank-revealing) QR factorization algorithms*, in Euro-Par 2002. Parallel Processing: Eighth International Euro-Par Conference, Paderborn, Germany, August 27–30, 2002, 2002.

[11] 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*, LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville, Jan. 1997.

[12] J. Demmel, *Trading off parallelism and numerical stability*, Tech. Rep. UT-CS-92-179, University of Tennessee, June 1992. LAPACK Working Note #53.

[13] J. Demmel and M. Hoemmen, *Communication-avoiding Krylov subspace methods*, tech. rep., University of California Berkeley, Department of Electrical Engineering and Computer Science, in preparation.

[14] J. J. Dongarra, S. Hammarling, and D. W. Walker, *Key concepts for parallel out-of-core LU factorization*, Scientific Programming, 5 (1996), pp. 173–184.

[15] E. Elmroth and F. Gustavson, *New serial and parallel recursive QR factorization algorithms for SMP systems*, in Applied Parallel Computing. Large Scale Scientific and Industrial Problems., B. K. et al., ed., vol. 1541 of Lecture Notes in Computer Science, Springer, 1998, pp. 120–128.

[16] ——, *Applying recursion to serial and parallel QR factorization leads to better performance*, IBM Journal of Research and Development, 44 (2000), pp. 605–624.

[17] R. W. Freund and M. Malhotra, *A block QMR algorithm for non-Hermitian linear systems with multiple right-hand sides*, Linear Algebra and its Applications, 254 (1997), pp. 119–157. Proceedings of the Fifth Conference of the International Linear Algebra Society (Atlanta, GA, 1995).

[18] J. R. Gilbert and E. G. Ng, *Predicting structure in nonsymmetric sparse matrix factorization*, Tech. Rep. ORNL/TM-12205, Oak Ridge National Laboratory, 1992.

[19] G. H. Golub and C. F. V. Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, USA, third ed., 1996.

[20] S. L. Graham, M. Snir, and e. Cynthia A. Patterson, *Getting Up To Speed: The Future Of Supercomputing*, National Academies Press, Washington, D.C., USA, 2005.

[21] A. Greenbaum, M. Rozložník, and Z. Strakoš, *Numerical behavior of the modified Gram-Schmidt GMRES implementation*, BIT Numerical Mathematics, 37 (1997), pp. 706–719.

[22] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1999.

[23] B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Transactions on Mathematical Software, 31 (2005), pp. 60–78.

[24] U. Hetmaniuk and R. Lehoucq, *Basis selection in LOBPCG*, Journal of Computational Physics, 218 (2006), pp. 324–332.

[25] D. Irony, S. Toledo, and A. Tiskin, *Communication lower bounds for distributed-memory matrix multiplication*, J. Parallel Distrib. Comput., 64 (2004), pp. 1017–1026.

[26] H. Jia-Wei and H. T. Kung, *I/O complexity: The red-blue pebble game*, in STOC '81: Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, New York, NY, USA, 1981, ACM, pp. 326–333.

[27] A. Kiełbasiński, *Analiza numeryczna algorytmu ortogonalizacji Grama–Schmidta*, Seria III: Matematyka Stosowana II, (1974), pp. 15–35.

[28] A. Knyazev, *BLOPEX webpage*. `http://www-math.cudenver.edu/~aknyazev/software/BLOPEX/`.

[29] A. V. Knyazev, M. Argentati, I. Lashuk, and E. E. Ovtchinnikov, *Block locally optimal preconditioned eigenvalue xolvers (BLOPEX) in HYPRE and PETSc*, Tech. Rep. UCDHSC-CCM-251P, University of California Davis, 2007.

[30] R. Lehoucq and K. Maschhoff, *Block Arnoldi method*, in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000, pp. 185–187.

[31] M. Leoncini, G. Manzini, and L. Margara, SIAM J. Comput., 28 (1999), pp. 2030–2058.

[32] L. Lugmayr, *Samsung 256GB SSD is world's fastest*. `http://www.i4u.com/article17560.html`, 25 May 2008. Accessed 30 May 2008.

[33] O. Marques, *BLZPACK webpage*. `http://crd.lbl.gov/~osni/`.

[34] J. J. Modi and M. R. B. Clarke, *An alternative Givens ordering*, Numer. Math., (1984), pp. 83–90.

[35] 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.

[36] D. P. O'Leary, *The block conjugate gradient algorithm and related methods*, Linear Algebra and its Applications, 29 (1980), pp. 293–322.

[37] B. N. Parlett, *The Symmetric Eigenvalue Problem*, SIAM, Philadelphia, 1998.

[38] A. Pothen and P. Raghavan, *Distributed orthogonal factorization: Givens and Householder algorithms*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 1113–1134.

[39] T. Project, *Anasazi webpage*. `http://software.sandia.gov/Trilinos/packages/anasazi/`.

[40] 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, Virginia, SIAM, Mar. 2001.

[41] A. H. Sameh and D. J. Kuck, *On Stable Parallel Linear System Solvers*, Journal of the Association for Computing Machinery, 25 (1978), pp. 81–91.

[42] R. Schreiber and C. V. Loan, *A storage efficient WY representation for products of Householder transformations*, SIAM J. Sci. Stat. Comput., 10 (1989), pp. 53–57.

[43] A. Smoktunowicz, J. Barlow, and J. Langou, *A note on the error analysis of Classical Gram-Schmidt*, Numerische Mathematik, 105 (2006), pp. 299–313.

[44] A. Stathopoulos, *PRIMME webpage*. `http://www.cs.wm.edu/~andreas/software/`.

[45] A. Stathopoulos and K. Wu, *A block orthogonalization procedure with constant synchronization requirements*, SIAM Journal on Scientific Computing, 23 (2002), pp. 2165–2182.

[46] S. Toledo, *A survey of out-of-core algorithms in numerical linear algebra*, in External Memory Algorithms and Visualization, J. Abello and J. S. Vitter, eds., American Mathematical Society Press, Providence, RI, 1999, pp. 161–180.

[47] B. Vital, *Étude de quelques méthodes de résolution de problèmes linéaires de grande taille sur multiprocesseur*, Ph.D. dissertation, Université de Rennes I, Rennes, Nov. 1990.

[48] H. F. Walker, *Implementation of the GMRES and Arnoldi methods using Householder transformations*, Tech. Rep. UCRL-93589, Lawrence Livermore National Laboratory, Oct. 1985.

[49] K. Wu and H. D. Simon, *TRLAN webpage*. `http://crd.lbl.gov/~kewu/ps/trlan_.html`.

Algorithm 9 CAQR: j-th step

Require: Steps 1 to $j-1$ of CAQR have been completed.

1: 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 as described in Section 8.

2: Each processor p that belongs to the column of processes holding panel j broadcasts along its row of processors the $m_j/P_r \times b$ rectangular matrix that holds the two sets of Householder vectors. Processor p also broadcasts two arrays of size b each, containing the Householder multipliers τ_p.

3: 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.)

4: **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**

5: Processors in the same process row as $target\_first\_proc(p,k)$ form $T_{level(p,k),k}$ locally. They also compute local 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 6.

6: 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 $C_{first\_proc(p,k)}$.

7: Processors in the same process row as $target\_first\_proc(p,k)$ compute local pieces of

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

8: 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.

9: 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 8.

10: **end for**

Algorithm 10 Outline of left-looking sequential CAQR factorization

1: **for** $J = 1$ to P_c **do**
2: **for** $K = 1$ to $J - 1$ **do**
3: Update blocks $K : P_r$ of the current panel, using panel K
4: **end for**
5: Factor blocks $J : P_r$ of the current panel
6: **end for**

Algorithm 11 Outline of right-looking sequential CAQR factorization

1: **for** $J = 1$ to P_c **do**
2: Factor the current panel (blocks J to P_r)
3: Update blocks $J : P_r \times J+1 : P_c$ of the trailing matrix, using the current panel
4: **end for**

Algorithm 12 Column-order right-looking sequential CAQR factorization

1: **for** $J = 1$ to P_c **do**
2: Factor the current panel: $P_r - J + 1$ reads of each $M \times N$ block of A. One write of the first block's Q factor (size $MN - N(N-1)/2$). $P_r - J$ writes of the remaining Q factors (each of size $MN - N(N-3)/2$).
3: **for** $K = J + 1 : P_c$ **do** ▷ Update trailing matrix
4: Load first Q block (size $MN - N(N-1)/2$) of panel J
5: Load block A_{KK} (size MN)
6: Apply first Q block to A_{KK}
7: **for** $L = J + 1 : P_r$ **do**
8: Load current Q block (size $MN - N(N-3)/2$) of panel J
9: Load block A_{LK} (size MN)
10: Apply current Q block to $[A_{L-1,K}; A_{LK}]$
11: Store block $A_{L-1,K}$ (size MN)
12: **end for**
13: Store block $A_{P_r,K}$ (size MN)
14: **end for**
15: **end for**

Algorithm 13 Left-looking sequential CAQR factorization

1: **for** $J = 1$ to P_c **do**
2: **for** $K = 1$ to $J - 1$ **do** ▷ Loop across previously factored panels
3: Load first Q block (size $MN - N(N-1)/2$) of panel K
4: Load block A_{KJ} (size MN) ▷ Current panel's index is J
5: Apply first Q block to A_{KJ}
6: **for** $L = K + 1$ to P_r **do** ▷ Loop over blocks $K + 1 : P_r$ of current panel
7: Load current Q block (size $MN - N(N-3)/2$) of panel K
8: Load block A_{LJ} (size MN)
9: Apply current Q block to $[A_{L-1,J}; A_{LJ}]$
10: Store block $A_{L-1,J}$ (size MN)
11: **end for**
12: Store block $A_{P_r,J}$ (size MN)
13: **end for**
14: Factor blocks $J : P_r$ of the current panel: $P_r - J + 1$ reads of each $M \times N$ block of A. One write of the first block's Q factor (size $MN - N(N-1)/2$). $P_r - J$ writes of the remaining Q factors (each of size $MN - N(N-3)/2$).
15: **end for**

Algorithm 14 Sequential left-looking QR factorization

1: **for** $j = 1$ to $n - c$ step c **do**
2: Read current panel (columns $j : j + c - 1$)
3: **for** $k = 1$ to $j - 1$ step b **do**
4: Read left panel (columns $k : k + b - 1$)
5: Apply left panel to current panel
6: **end for**
7: Factor and write current panel
8: **end for**

[1]

Algorithm 15 Applying Q^T from Algorithm 14 to $n \times r$ matrix B

1: **for** $j = 1$ to $r - c$ step c **do**
2: Read current panel of B (columns $j : j + c - 1$)
3: **for** $k = 1$ to $j - 1$ step b **do**
4: Read left panel of Q (columns $k : k + b - 1$)
5: Apply left panel to current panel
6: **end for**
7: Write current panel of B
8: **end for**

[1]

Algorithm 16 Outline of ScaLAPACK out-of-DRAM QR factorization (PFDGEQRF)

1: **for** $j = 1$ to $n - c$ step c **do**
2: Read current panel (columns $j : j + c - 1$)
3: **for** $k = 1$ to $j - 1$ step b **do**
4: Read left panel (columns $k : k + b - 1$)
5: Apply left panel to current panel
6: **end for**
7: Factor and write current panel
8: **end for**

[1]

Algorithm 17 More detailed outline of ScaLAPACK out-of-DRAM Householder QR factorization (PFDGEQRF), with border cleanup

1: **for** $j = 1$ to $\left(\lfloor \frac{n}{c} \rfloor - 1\right) c + 1$ step c **do**
2: Read current panel (columns $j : j + c - 1$, rows $1 : m$)
3: **for** $k = 1$ to $\left(\lfloor \frac{j-1}{b} \rfloor - 1\right) b + 1$, step b **do**
4: Read left panel (columns $k : k + b - 1$, lower trapezoid, starting at row k)
5: Apply left panel to rows $k : m$ of current panel
6: **end for**
7: $k := \lfloor \frac{j-1}{b} \rfloor b + 1$
8: Read left panel (columns $k : j - 1$, lower trapezoid, starting at row k)
9: Apply left panel to rows $k : m$ of current panel
10: Factor current panel (rows $1 : m$)
11: Write current panel (rows $1 : m$)
12: **end for**
13: $j := \lfloor \frac{n}{c} \rfloor c + 1$
14: Read current panel (columns $j : n$, rows $1 : m$)
15: **for** $k = 1$ to $\left(\lfloor \frac{j-1}{b} \rfloor - 1\right) b + 1$, step b **do**
16: Read left panel (columns $k : k + b - 1$, lower trapezoid, starting at row k)
17: Apply left panel to rows $k : m$ of current panel
18: **end for**
19: $k := \lfloor \frac{j-1}{b} \rfloor b + 1$
20: Read left panel (columns $k : j - 1$, lower trapezoid, starting at row k)
21: Apply left panel to current panel
22: Factor current panel (rows $1 : m$)
23: Write current panel (rows $1 : m$)
[1]