Parallel Out-of-Core Computation and Updating of the QR Factorization

UTCS Technical Report TR-03-40

PLAPACK Working Note #13

Brian C. Gunter
Center for Space Research
The University of Texas at Austin
Austin, TX 78712
gunter@csr.utexas.edu

Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
rvdg@cs.utexas.edu

September 10, 2003

Abstract

This paper discusses the high-performance parallel implementation of the computation and updating of QR factorizations of dense matrices, including problems large enough to require out-of-core computation, where the matrix is stored on disk. The algorithms presented here are scalable both in problem size and as the number of processors increases. Implementation using the Parallel Linear Algebra Package (PLAPACK) and the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) is discussed. The methods are shown to attain excellent performance, in some cases attaining roughly 80% of the peak of the architectures on which the experiments were performed.

1 Introduction

With the recent improvements in memory access, storage capacity, and processing power of high-performance computers, operating on large dense linear systems is not as daunting a task as it has been in the past. One such realm where this new capability has been of extreme value is in the Earth sciences, where large overdetermined Linear Least-Squares problems are encountered which can involve tens of thousands of parameters and millions of observations [16]. Since these systems have the potential to be mildly ill-conditioned, the method chosen to compute the least-squares solution is one utilizing the QR factorization, since it provides greater accuracy than the Method of Normal Equations. To tackle a problem of this size requires the use of an efficient and scalable implementation of the QR factorization that can take advantage of the power of modern day supercomputers. Initially, an in-core parallel implementation was developed, but later an out-of-core (OOC) implementation was created to handle even larger problems.

In reporting the results of our research, this paper makes the following contributions:

- It reviews the standard techniques for the high-performance implementation of the QR factorization based on the Householder transformation [4, 3, 13, 8].
- It extends these techniques to the problem of updating the QR factorization as additional batches of observations are added to the system.
- It demonstrates how the techniques used to update a QR factorization can be used to implement an OOC QR factorization algorithm that is more scalable than other previously proposed OOC approaches

for this problem [7]. In particular, while it was previously observed that so-called tiled approaches are more scalable than so-called "slab" approaches for the OOC computation of the Cholesky factorization [31, 26, 25, 17] and a (possibly unstable) variant of the LU factorization [27], only (nonscalable) slab approaches had been previously proposed for the OOC QR factorization.

- It discusses a parallel implementation of the algorithms using the Parallel Linear Algebra Package (PLAPACK) [32] and its out-of-core extension, the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) [26, 25, 17, 1].
- It reports excellent performance attained on massively parallel distributed memory supercomputers.

While we reported initial performance results for the implementations in a previous (conference) paper [19], this paper goes into considerably more depth.

This paper is structured as follows: Section 2 briefly describes the notation used in the paper. Section 3 reviews the QR factorization using Householder transformations. This includes a discussion of the block algorithm as well as the QR factorization's application to the least squares problem. Section 4 discusses how the standard QR factorization can be updated when new information becomes available. This technique later becomes a key component of the OOC algorithm, which is given in Section 5. The actual parallel implementation is discussed in Section 6. Results from experiments are reported in Section 7. Section 8 provides some final thoughts and conclusions.

2 Notation

In this paper, we adopt the following conventions: Matrices, vectors, and scalars are denoted by upper-case, lower-case, and lower-case Greek letters, respectively. The identity matrix will be denoted by I and e_1 will denote the first column of the identity matrix (in other words, the vector with first element equal to unity and all other elements equal to zero). The dimensions and lengths of such matrices and vectors will generally be obvious from context.

Algorithms in this paper are given in a notation that we have recently adopted as part of the FLAME project [15, 24]. If one keeps in mind that the double lines in the partitioned matrices and vectors relate to how far into the matrices and vectors the computation has proceeded, we believe the notation to be intuitively obvious. If not, we suggest that the reader consult some of these related papers.

3 Computing the QR factorization via Householder Transformations

Given an $m \times n$ real-valued matrix A, with $m \geq n$, the QR factorization is given by

$$A = QR$$

where the $m \times m$ matrix Q has mutually orthonormal columns ($Q^T Q = I$) and the $m \times n$ matrix R is upper triangular.

There are many different methods for computing the QR factorization, including those based on Givens rotations, orthogonalization via Gram-Schmidt and Modified Gram-Schmidt, and Householder transformations [13, 33]. For dense matrices, the method of choice depends largely on how the factorization is subsequently used, the stability of the system, and the dimensions of the matrix. For problems where $m \gg n$, the method based on Householder transformations is typically the algorithm of choice, especially when Q needs not be explicitly computed.

3.1 Householder transformations (reflectors)

Given the real-valued vector x of length m, partition

$$x = \left(\frac{\chi_1}{x_2} \right)$$

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right)$ and $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right)$

where A_{TL} is 0×0 and b_T has 0 elements

while $n(A_{BR}) \neq 0$ **do**

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array} \right) \text{ and } \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline b_2 \end{array} \right)$$

where α_{11} and β_1 are scalars

$$\left[\left(\begin{array}{c} 1 \\ \hline u_2 \end{array} \right), \eta, \beta_1 \right] := h \left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array} \right)$$

$$\alpha_{11} := \eta$$

$$a_{21} := u_2$$

$$w^T := a_{12}^T + u_2^T A_{22}$$

$$\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array} \right) := \left(\begin{array}{c} a_{12}^T - \beta_1 w^T \\ \hline A_{22} - \beta_1 u_2 w^T \end{array} \right)$$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array} \right) \text{ and } \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array} \right)$$

enddo

Figure 1: Unblocked Householder QR factorization.

where χ_1 equals the first element of x.

The Householder vector associated with x is defined as the vector

$$u = \left(\begin{array}{c} 1 \\ \hline x_2/\nu_1 \end{array} \right),$$

where $\nu_1 = \chi_1 + \text{sign}(\chi_1)\|x\|_2$. If $\beta = \frac{2}{u^T u}$ then $(I - \beta u u^T)x = \eta e_1$, annihilating all but the first element of x. Here $\eta = -\text{sign}(\chi_1)\|x\|_2$. The transformation $I - \beta u u^T$, with $\beta = 2/u^T u$ is referred to as a Householder transformation or reflector.

Let us introduce the notation $[u, \eta, \beta] := h(x)$ as the computation of the above mentioned η, u, and β from vector x and the notation $H(x)$ for the transformation $(I - \beta u u^T)$ where $[u, \eta, \beta] = h(x)$. An important feature of $H(x)$ is that it is orthonormal ($H(x)^T H(x) = H(x)H(x)^T = I$) and symmetric ($H(x)^T = H(x)$).

3.2 A simple algorithm for the QR factorization via Householder transformations

The computation of the QR factorization commences as described in Figure 1. The idea is that Householder transformations are computed to successively annihilate elements below the diagonal of matrix A. The Householder vectors are stored below the diagonal over the elements of A that have been so annihilated. Upon completion, matrix R has overwritten the upper triangular part of the matrix, while the Householder vectors are stored in the lower trapezoidal part of the matrix. The scalars β discussed above are stored in the vector b.

If the matrix Q is explicitly desired, it can be formed by computing the first n columns of $H_1 H_2 \cdots H_n$ where H_i equals the ith Householder transformation computed as part of the factorization described above. In our application, we do not need to form Q explicitly and thus will not discuss the issue further.

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $T \rightarrow \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right)$
where A_{TL} is 0×0 and T_T has 0 rows

while $n(A_{BR}) \neq 0$ **do**
Determine block size k
Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ T_2 \end{array}\right)$$

where A_{11} is $k \times k$ and T_1 has k rows

$$\left[\left(\frac{A_{11}}{A_{21}}\right), \eta, b_1\right] := \left[\left(\frac{\{Y\backslash R\}_{11}}{Y_{21}}\right), \eta, b_1\right] = QR\left(\left(\frac{A_{11}}{A_{21}}\right)\right)$$

Compute T_1 from $\left[\left(\frac{\{Y\backslash R\}_{11}}{Y_{21}}\right), \eta, b_1\right]$

$$\left(\frac{A_{12}}{A_{22}}\right) := \left(I + \left(\frac{Y_{11}}{Y_{21}}\right) T_1^T \left(\begin{array}{c|c} Y_{11}^T & Y_{21}^T \end{array}\right)\right)\left(\frac{A_{12}}{A_{22}}\right)$$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ T_1 \\ \hline T_2 \end{array}\right)$$

enddo

Figure 2: Blocked Householder QR factorization.

3.3 A high-performance (blocked) algorithm for the QR factorization

It is well-known that high performance can be achieved in a portable fashion by casting algorithms in terms of matrix-matrix multiplication [2, 10, 12]. We now review how to do so for the QR factorization [3, 2].

Two observations play a key role:

- Let $u_1, \ldots, u_k$ equal the first k Householder vectors computed as part of the factorization and $\beta_1, \ldots, \beta_k$ the corresponding scalars. Then $H_k \cdots H_2 H_1 = I + YTY^T$ where Y is a $n \times k$ unit lower-trapezoidal matrix, T is a $k \times k$ upper-triangular matrix, and the jth column of the lower-trapezoidal part of Y equals u_j.
- The QR factorization of the first k columns of A yields the same k vectors u_k and same values in the upper triangular part of those k columns as would a full QR factorization.

Notice that this suggests the blocked algorithm given in Figure 2.

Note 1 *Notice that the algorithm stores the "T" matrices that are part of the block Householder transformation $I + YTY^T$. This is a distinct departure from traditional implementations like those found in LAPACK [2] and will allow us to avoid recomputation of those matrices as part of the OOC implementation of the QR factorization.*

3.4 Solving multiple Linear Least-Squares problems

Given a real-valued $m \times n$ matrix A and vector y of length m, the linear least-squares problem is generally stated as

$$\min_x \|y - Ax\|_2$$

where the desired result is a vector x that minimizes the above expression. It is well-known that the minimizing vector x can be found by computing the QR factorization $A = QR$, computing $z = Q^T y$, and solving $Rx = z_T$ where z_T denotes the first n elements of z.

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$ and $T \rightarrow \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right)$
  where A_{TL} is 0×0 and B_T and T_T have 0 rows

while $n(A_{BR}) \neq 0$ **do**
  Determine block size k
  Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline T_2 \end{array}\right)$$

  where A_{11} is $k \times k$ and B_1 and T_1 have k rows

$$\left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) := \left(I + \left(\begin{array}{c} Y_{11} \\ \hline Y_{21} \end{array}\right) T_1^T \left(\begin{array}{c|c} Y_{11}^T & Y_{21}^T \end{array}\right)\right) \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right)$$

NOTE: Here Y_{11} refers to the lower triangular part of A_{11} and Y_{21} to A_{21}

  Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline\hline T_2 \end{array}\right)$$

enddo

Figure 3: Forward substitution-like of right-hand-side matrix B.

Alternatively, one can think of this as follows: Append y to A to form $\left(\begin{array}{c|c} A & y \end{array}\right)$. Compute the QR factorization $A = QR$, storing the Householder vectors and R over A. Update y by applying the Householder transformations used to compute R to vector y, which overwrites y with z. Finally, solve $Rx = z_T$ with the first n elements of the updated y. This second approach is reminiscent of how a linear system can be solved by appending the right-hand-side vector to the system and performing an LU factorization (or, equivalently, Gaussian elimination) on the appended system, followed by a back-substitution step.

Finally, if there exists a set of right-hand-sides, one can simultaneously solve a linear least-squares problem with A and columns of B by the following approach: Append B to A to form $\left(\begin{array}{c|c} A & B \end{array}\right)$. Compute the QR factorization $A = QR$, storing the Householder vectors and R over A. Update B by applying the Householder transformations used to compute R to matrix B, which overwrites B with Z. Finally, solve $RX = Z_T$ with the first n rows of the updated B. It is this second operation with a right-hand-side B that we will encounter in the OOC implementation of the QR factorization. An algorithm for the first, forward substitution-like, step is given in Figure 3.

Note 2 *Notice that because we store the "T" matrices that are part of the block Householder transformation* $I + YTY^T$*, they need not be recomputed as part of the "forward substitution" step on matrix* B*. This is again a distinct departure from traditional implementations like those found in LAPACK.*

4 Updating the QR Factorization

Frequently, the linear equations used in the least squares problem are collected incrementally. For example, if the observations from a particular instrument are only collected or contributed on a monthly basis, it would be useful to combine each new batch of data into the existing solution without having to recombine all of the

Partition $R \rightarrow \left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array} \right)$, $C \rightarrow \left(\begin{array}{c||c} C_L & C_R \end{array} \right)$, and $b \rightarrow \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right)$
where R_{TL} and C_L are 0×0 and b_T has 0 elements

while $n(R_{BR}) \neq 0$ **do**
Repartition

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} R_{00} & r_{01} & R_{02} \\ \hline\hline r_{10}^T & \rho_{11} & r_{12}^T \\ \hline R_{20} & r_{21} & R_{22} \end{array} \right),$$

$$\left(\begin{array}{c||c} C_L & C_R \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} C_0 & c_1 & C_2 \end{array} \right), \text{ and } \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline\hline \beta_1 \\ \hline b_2 \end{array} \right)$$

where ρ_{11} and β_1 are scalars and c_1 is a column

$$\left[\left(\frac{1}{u_2} \right), \eta, \beta_1 \right] := h \left(\frac{\rho_{11}}{c_1} \right)$$

$\rho_{11} := \eta$
$c_1 := u_2$
$w^T := r_{12}^T + u_2^T C_2$

$$\left(\frac{r_{12}^T}{C_2} \right) := \left(\frac{r_{12}^T - \beta_1 w^T}{C_2 - \beta_1 u_2 w^T} \right)$$

Continue with

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} R_{00} & r_{01} & R_{02} \\ \hline r_{10}^T & \rho_{11} & r_{12}^T \\ \hline\hline R_{20} & r_{21} & R_{22} \end{array} \right),$$

$$\left(\begin{array}{c||c} C_L & C_R \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} C_0 & c_1 & C_2 \end{array} \right), \text{ and } \left(\begin{array}{c} b_T \\ \hline\hline b_B \end{array} \right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline\hline b_2 \end{array} \right)$$

enddo

Figure 4: Unblocked update to a QR factorization.

previous data again. In other words, it would be nice to update the QR factorization rather than compute it from the original data to which the new data has been added. We now show how the QR factorization can be updated as additional batches of equations (i.e., observations) become available.

4.1 Factorization

Let us assume that we have computed Q and R such that $A = QR$, overwriting A with the Householder vectors and upper triangular matrix R, and storing the "T" matrices in matrix T. Thus, we have available $A = \{Y \backslash R\}$ and T. Now, consider the QR factorization of matrix

$$\left(\begin{array}{c} A \\ C \end{array} \right) = \bar{Q}\bar{R} \tag{1}$$

A key observation is that the QR factorization of

$$\left(\begin{array}{c} R \\ C \end{array} \right) \tag{2}$$

produces the same upper triangular matrix $\bar{R}$ as does the factorization in (1). If we are not interested in explicitly forming $\bar{Q}$ and are satisfied with storing the Householder vectors required to first compute the QR factorization of A and next the QR factorization in (2), then we have an approach for computing the QR factorization of an updated system. The unblocked and blocked algorithm for doing so is given in Figures 4 and 5, respectively.

Partition $R \rightarrow \left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right)$, $C \rightarrow \left(\begin{array}{c||c} C_L & C_R \end{array}\right)$, and $T \rightarrow \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right)$
where R_{TL} and C_L are 0×0 and T_T has 0 rows

while $n(R_{BR}) \neq 0$ **do**
Determine block size k
Repartition

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} R_{00} & R_{01} & R_{02} \\ \hline\hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right),$$

$$\left(\begin{array}{c} C_L \\ \hline\hline C_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_0 & C_1 & C_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline T_2 \end{array}\right)$$

where A_{11} is $k \times k$, C_1 has k columns and T_1 has k rows

$$\left[\left(\frac{R_{11}}{C_1}\right), \eta, b_1\right] := \left[\left(\frac{\{0\backslash R\}_{11}}{Y_1}\right), \eta, b_1\right] = QR\left(\left(\frac{R_{11}}{C_1}\right)\right)$$

Compute T_1 from $\left[\left(\frac{I}{Y_1}\right), \eta, b_1\right]$

$$\left(\frac{R_{12}}{C_2}\right) := \left(I + \left(\frac{I}{Y_1}\right) T_1^T \left(\begin{array}{c|c} I & Y_1^T \end{array}\right)\right)\left(\frac{R_{12}}{C_2}\right)$$

Continue with

$$\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline\hline R_{20} & R_{21} & R_{22} \end{array}\right),$$

$$\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_0 & C_1 & C_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline\hline T_2 \end{array}\right)$$

enddo

Figure 5: Blocked update to a QR factorization.

Note 3 *Notice that the algorithm is explicitly designed to take advantage of the zeros below the diagonal of R. As a result, factoring A followed by an update of the factorization requires essentially no more computation than the factorization in (1). Also, the Householder vectors that are stored below the diagonal are not overwritten. It is the case that an additional vector b is required to store the "β"s for the unblocked algorithm and an additional matrix is required to store the triangular "T" matrices for the blocked algorithm.*

4.2 Solving multiple Linear Least-Squares problems

If we now wish to compute multiple Linear Least-Squares solutions, one for each of the systems of linear equations defined by picking one column of the right-hand-side of

$$\left(\begin{array}{c} A \\ C \end{array}\right) X = \left(\begin{array}{c} B \\ D \end{array}\right)$$

the following approach will yield the desired result:

- Append $\left(\begin{array}{c|c} A & B \\ C & D \end{array}\right)$.
- Overwrite A with its factors $\{Y \backslash R\}$, also computing matrix T, as in Figure 2.
- Overwrite $\left(\begin{array}{c} R \\ C \end{array}\right)$ with $\left(\begin{array}{c} \bar{R} \\ Y^{(C)} \end{array}\right)$, also computing $T^{(C)}$ as in Figure 5.

Partition $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$, $C \rightarrow \left(\begin{array}{c||c} C_L & C_R \end{array}\right)$, and $T \rightarrow \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right)$
where C_L has 0 columns, and B_T and T_T have 0 rows

while $n(C_R) \neq 0$ **do**
Determine block size k
Repartition

$$\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_0 & C_1 & C_2 \end{array}\right),$$

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline\hline T_1 \\ \hline T_2 \end{array}\right)$$

where C_1 has k columns and B_1 and T_1 have k rows

$$\left(\begin{array}{c} B_1 \\ \hline D \end{array}\right) := \left(I + \left(\begin{array}{c} I \\ \hline C_1 \end{array}\right) T_1^T \left(\begin{array}{c|c} I & C_1^T \end{array}\right)\right) \left(\begin{array}{c} B_1 \\ \hline D \end{array}\right)$$

Continue with

$$\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_0 & C_1 & C_2 \end{array}\right),$$

$$\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right), \text{ and } \left(\begin{array}{c} T_T \\ \hline\hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline\hline T_2 \end{array}\right)$$

enddo

Figure 6: Forward substitution consistent with the QR factorization of an updated matrix.

- Update B by forward substitution as in Figure 3.
- Update $\begin{pmatrix} B \\ D \end{pmatrix}$ by forward substitution using the Householder transformations computed as part of the update of R, as in Figure 6.
- Solve $\bar{R}X = B_T$ where B_T denotes the top n rows of the updated matrix B.

5 Out-of-Core Algorithms

Having now described the in-core algorithm, we can apply a similar strategy for problems that are too large to fit in the available memory of the machine. To deal with these problems, we have developed an out-of-core algorithm that allows us to store the bulk of the matrix components on disk, while only working on select pieces in-core at any one time. The algorithm we will outline here is unique in that it is both scalable and efficient.

5.1 Out-of-core QR factorization

Traditional OOC algorithms of the QR factorization have used a slab approach, in which the OOC matrix is processed by bringing into memory one or more slabs (blocks of columns) of the matrix at a time [6, 21, 7, 31, 27]. The problem with this technique is that it is inherently not scalable in the following sense: As the row dimension, m, of A becomes larger and larger, the width of the slab you can bring into memory becomes proportionally smaller. As m reaches into the millions, the number of columns able to be brought into memory numbers only in the dozens, even on today's powerful machines with large memories.

The alternative that we have found to the slab approach is to work with the matrix as a collection of tiles, where a tile is a submatrix that is roughly square. As was shown for the OOC Cholesky factorization [31, 26, 25, 17], a tiled approach provides true scalability. We will see that the processing of these tiles becomes a

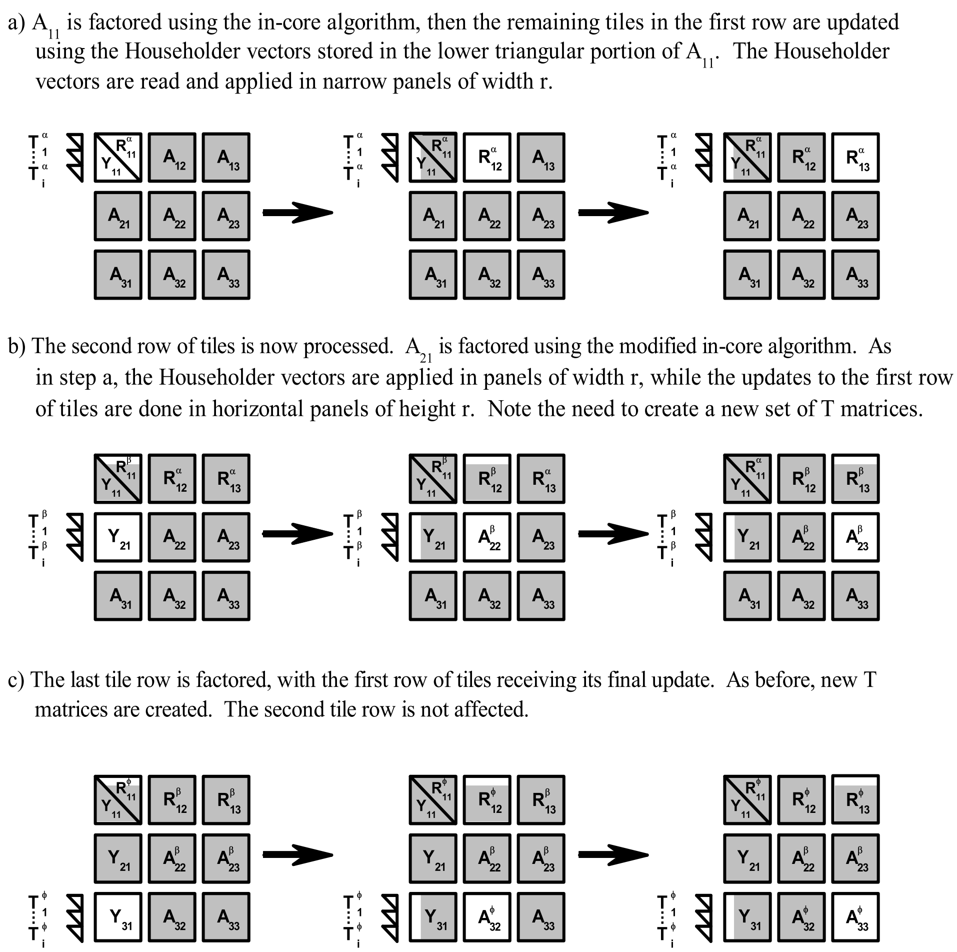


Figure 7: Factoring the first row of tiles using the out-of-core approach. Grey regions indicates components that reside on disk.

simple application of the algorithms in Figures 2, 3, 5, and 6. The high performance achieved with these in-core procedures is maintained when processing the tiles, providing the same benefits to the OOC approach. As the problem size increases, additional tiles are simply added to the system, without adversely affecting performance.

To demonstrate this, we begin in much the same way as we did with the in-core algorithm, except now the matrix A resides entirely on disk and not in memory. We partition the matrix into a series of tiles, as illustrated in Figure 7. For the purposes of this example, we will assume that the matrix A is square and divided into nine tiles of size $t \times t$, forming a 3×3 grid of tiles.

1. The first tile, A_{11}, is read into memory and factored using the in-core algorithm in Figure 2. Upon completion, A_{11} is written to disk. For now, the "T" matrices created during this step are kept in-core.

   Notice that as the dimension t becomes larger, the cost of reading and writing A_{11}($O(t^2)$) improves because it is amortized over the useful computation ($O(t^3)$), i.e., the QR factorization. Consequently, the larger t becomes, the less significant the I/O overhead.

2. Next, tile A_{12} is brought into memory. It is updated consistent with the factorization of A_{11}, using the Householder vectors that have overwritten the lower triangular part of A_{11} and the "T" matrices still in memory. In other words, the algorithm in Figure 3 is employed. Once updated, A_{12} is written back to disk.

   On the surface, it would thus appear that two tiles must be in memory, consequently limiting the tile dimension t. However, a closer look at the update in the body of the loop of the algorithm in Figure 3 shows that only a panel of columns of A_{11} needs to be brought into memory, which can be discarded as soon as it has been used to update A_{12}. Thus, at most $t \times k$ elements of A_{11} need to be in memory at a time. The cost of bringing these elements into memory is amortized over $O(kt^2)$ computation. Similarly, the $O(t^2)$ cost of bringing A_{12} into memory is amortized over $O(t^3)$ computations. The larger t becomes, the less significant the I/O overhead.

   The remaining tiles in the first row are processed similarly. Once the entire first row has been processed, the "T" matrices computed in the factorization of A_{11} can be written to disk.

3. After processing the first row, A_{21} is brought into memory. It must be updated together with A_{11} according to the algorithm in Figure 5, generating a new set of "T" matrices. Once updated, A_{21} is written to disk, while the now generated "T" matrices are kept in memory.

   Again, it would appear that two tiles must be in memory, thus limiting the tile dimension t. However, the update in the body of the loop of the algorithm in Figure 5 shows that only a panel of rows of A_{11} needs to be brought into memory, which can be written back to disk as soon as it has been used to update A_{21}. Thus, at most $k \times t$ elements of A_{11} need to be in memory at a time. The cost of bringing these elements into memory is amortized over $O(kt^2)$ computations. Similarly, the $O(t^2)$ cost of bringing A_{21} into memory is amortized over $O(t^3)$ computations. The larger t, the less significant the I/O overhead.

4. Once A_{21} is updated, A_{22} is brought into memory, to be updated according to Step 3 above, using the algorithm in Figure 6.

   It would appear that now A_{21}, A_{12}, and A_{22} must all be in memory simultaneously. However, the update in the body of the loop in Figure 6 requires only a panel of columns of A_{21} and a panel of rows of A_{12} to be in memory. Thus only A_{22} needs to be kept in memory, while panels of the other two matrices are streamed from disk. The cost of the I/O involved is $O(kt)$ per iteration of the loop, which is amortized over $O(kt^2)$ computations. Meanwhile, the $O(t^2)$ cost of bringing A_{22} into memory is amortized over $O(t^3)$ computations. The larger t, the less significant the I/O overhead.

   The remaining tiles in the second row are processed similarly.

5. The third row of tiles is handled in the same manner as described in Steps 3 and 4. A_{31} is first factored with A_{11}, creating a new set of "T" matrices and overwriting A_{31} with the corresponding Householder vectors. A_{32} is brought into memory and updated with column panels from A_{31}, while also updated A_{12} in row panels of height k. The same is done for A_{33} and A_{13}. Note that only the first and third row of tiles are affected by these operations.

6. Now, Steps 1–5 start to repeat: A_{22} is factored as A_{11} was in Step 1. The remaining tiles in the second row are processed as described in Step 2. The tiles in the third row below the tiles on the diagonal are processed as in Step 3, and the remaining tiles in the third row are processed as in Steps 4 and 5. After this, it is back to Step 1 with A_{33} and so forth.

This process is illustrated in Figure 7. In that figure, it is the unshaded part of the matrix that is in memory at a typical stage of the algorithm.

Note 4 *In principle, most of memory can be dedicated to storing a single $t \times t$ tile. This allows t to be as large as possible, which then improves the indicated ratios of I/O to useful computation.*

5.2 Out-of-core updating

To update an existing OOC solution with a new set of equations is straightforward, as the OOC algorithm was designed to handle problems of arbitrary length. The new data is simply divided into the appropriate tiles and combined with the existing solution in the same manner that the 2nd and 3rd row of tiles was handled in the previous section.

6 Implementation

In this section, we discuss some practical considerations related to the actual implementation of the OOC algorithms.

6.1 Parallel implementation

So far in this paper, parallel implementation has not been explicitly discussed. We now give a few details.

It is well-known that a scalable implementation of dense linear algebra operations requires the use of a so-called two-dimensional matrix distribution [20, 29]. Moreover, to ensure load-balance as the active part of the matrix shrinks, an overdecomposition and wrapping of the matrix is typically employed [23, 30, 32, 9].

The observation now is that if one has parallel implementations of the algorithms in Figures 1–6, then the parallel implementation of the OOC algorithm becomes straight-forward. The parallel implementation of the QR factorization is well-understood, and is available as part of our PLAPACK package, as well as part of the Scalable Linear Algebra Package (ScaLAPACK) [5]. Since the remaining algorithms are, in essense, merely variations of the QR factorization, we implemented them as modifications of the PLAPACK QR factorization. Further modifications had to be made so that the panels being streamed from disk were read into memory and/or written out to disk at the appropriate time. Our POOCLAPACK package facilitates this read operation. Finally, a routine that manages the processing of the tiles was also written.

6.2 Optimizing I/O performance

The OOC method described in Section 5 advocates a single-tile method, in which most of memory is dedicated to a single tile. As argued, this is desirable because the I/O overhead decreases as the tile size increases. While this is easy to justify theoretically, in this section we point out a practical consideration which suggests that keeping two tiles in memory may lead to better performance, if the implementation is to be kept simple.

First, a few details about the storage of matrices. The matrices assigned locally to each processor as part of tiles are stored in memory in column-major order. Similarly, on disk, they are stored in column-major order. More precisely, the columns are stored so that if a panel of columns is read by a processor from disk, they are all contiguous in memory. This makes the reading of a tile and of panels of columns relatively cheap, since I/O carries a large start-up cost (latency). In other words, a panel of columns can be read essentially at peak bandwidth. By contrast, the reading of a panel of rows is generally staged as the reading of individual columns of that panel, incurring a latency related cost for each such column. This makes the reading of a panel of rows prohibitively expensive.

The update in Step 2 in Section 5.1 requires only column panels (of Householder vectors) to be read from disk. By contrast, the operations in Steps 3 and 4 require panels of rows to be brought in. Thus, it becomes advantageous to bring the entire tile from which panels of rows are to be used into memory, leading to a two-tile OOC algorithm. It is this approach that was actually implemented by us and used to obtain the performance numbers described in the next section.

Note 5 *By cleverly transposing tiles above the diagonal after processing, it is possible to implement the one-tile approach while still only reading panels of columns. We did not do so in an effort to keep the implementation simple.*

7 Performance

In this section, we demonstrate that the presented algorithm attains very high performance on distributed memory parallel architectures.

7.1 Target architectures

The POOCLAPACK implementation of the OOC QR factorization and update algorithm is portable essentially to any platform that supports the Message-Passing Interface [14, 28] and the Basic Linear Algebra Subprograms [22, 11, 10]. To date, the implementation has been ported to the SGI Origin 3000 and Linux PC cluster environments, in addition to the Cray T3E and IBM P-series systems.

In this paper, we report performance on two architectures:

- The Cray T3E-600. The system on which the experiments were performed has 272 total processors, each with 128MB of available memory. The T3E operates at a peak theoretical performance of 600 millions of floating point operations per second per processor (MFLOPS/sec/proc). For reference, the matrix-matrix multiply operation (DGEMM) was benchmarked at 445 MFLOPS/sec/proc for the particular machine used in this study. The BLAS used was provided as part of the Cray Scientific Library. It should also be noted that since the T3E is a true 64-bit platform, all arithmetic was done using 64-bit precision.

- The IBM P690. The system on which the experiments were performed consists of SMP nodes, where each node consists of 16 Power4 (1.3 GHz) processors, with 32 GBytes of available memory. The P690s operate at four FLOPS per cycle for a peak theoretical performance of 5200 MFLOPS/sec/proc, with a DGEMM benchmark of 3723 MFLOPS/sec/proc. IBM's optimized Engineering and Scientific Subroutine Library (ESSL) was used in place of the standard BLAS library. Again, all computation was performed in 64-bit arithmetic.

7.2 Reporting performance

The operation count for a Householder transform based QR factorization of an $m \times m$ matrix is given by approximately $\frac{4}{3}m^3$ floating point operations. While our OOC algorithm requires more operations, due to the accumulation and application of the "T" matrices, it is this operation count that represents the *useful* computation.

Thus, given $T_p(m)$, the time in seconds required on p processors to factor an $m \times m$ matrix, the rate in MFLOPS/sec/proc at which the processors compute is given by the formula

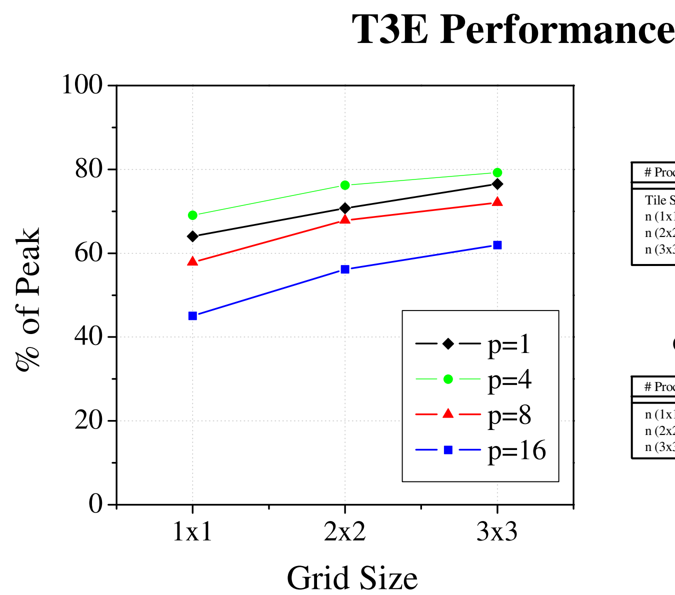
$$R_p(m) = \frac{\frac{4}{3}m^3}{T_p(m)} \times \frac{10^{-6}}{p}$$

Now, since the bulk of the computation is cast in terms of local matrix-matrix multiplications, the upper bound on $R_p(m)$ is given by the rate in MFLOPS/sec attained by DGEMM, which we will denote by R_{dgemm}. We consider this the peak performance that can be attained per processor. Now, we report the percentage of peak attained by our OOC implementation as the ratio

$$\frac{R_p(m)}{R_{dgemm}}.$$

7.3 Results

Figure 8 illustrates the performance of the OOC algorithm. In our experiment, we vary both the number of processors used and the problem size in the following way: Parameter t is chosen as the dimension of the tiles that will be kept in memory. Naturally, as the number of processors increases, the total available memory increases, and t can be increased. Factorizations of problems of size 1×1 tiles through 3×3 tiles were subsequently timed (reported as the Grid Size along the x-axis). The curves connect the data points

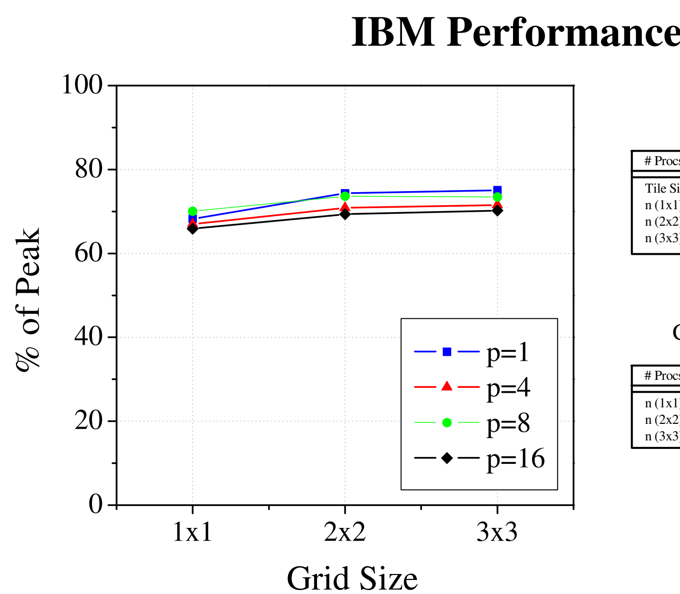


Problem Size

# Procs	1	4	16	64
Tile Size	2088	4704	8448	18432
n (1x1)	2088	4704	8448	18432
n (2x2)	4176	9408	16896	36864
n (3x3)	6264	14112	25344	55296

Compute Time (sec.)

# Procs	1	4	16	64
n (1x1)	43	112	195	651
n (2x2)	308	818	1332	4176
n (3x3)	962	2657	4231	12776



Problem Size

# Procs	1	4	16	64
Tile Size	7900	21000	30000	42000
n (1x1)	7900	21000	30000	42000
n (2x2)	15800	42000	60000	84000
n (3x3)	23700	63000	90000	126000

Compute Time (sec.)

# Procs	1	4	16	64
n (1x1)	259	1238	1726	2518
n (2x2)	1900	9365	13130	19128
n (3x3)	6349	31286	44439	63770

Figure 8: Performance of the OOC algorithm on a Cray T3E and IBM P690.

corresponding to the number of processors indicated in the legend. As the figure shows, the performance is quite respectable.

It is interesting to note that as the problem size becomes larger, the performance improves. This can be explained by the fact that as the problem size increases, more of the computation is in the operations in Figures 3 and 6, which cast more of the computation in matrix-matrix multiplication, the operation that attains the highest performance.

There is a noticable difference between the two architectures regarding scalability as the number of processors is increased. This can largely be attributed to the fact that the I/O performance of the specific Cray T3E used for these experiments becomes a bottleneck as more processors access the disk simultaneously.

7.4 Further possible improvements

The use of asynchronous I/O (i.e., overlapping I/O with computation) was explored in a previous study on parallel OOC implementation of the Cholesky factorization [26, 25, 17]. While it was determined that a slight performance increase could be achieved on machines with slower I/O bandwidths, the complexity of the code required to do this was considered prohibitive for the algorithms presented in this paper. Advances in I/O technology with newer high performance machines also render this performance increase practically negligible. Consequently, asynchronous I/O was not used to achieve the performance numbers described in Figure 8.

While the results above were obtained using the Cray T3E and IBM P690, it should be noted that the performance of the algorithm on other platforms is comparable when examining the speed as a percentage of peak.

8 Conclusion

We have demonstrated that a modification of the standard in-core QR factorization algorithm, combined with a tile-based approach for out-of-core implementations, results in a highly efficient and powerful method for computing QR factorizations of large, dense matrices. We believe that the resulting implementation is unique in that it is scalable both as the number of processors is increased and as, for a fixed number of processors, the problem size is increased. The performance of these algorithms is impressive, reaching roughly 80% of peak in some cases.

The application of these algorithms has already proven valuable to the Earth Sciences, particularly with regard to the determination of the Earth's gravity field [16, 18]. In addition, the tile-based approach is well suited for other types of dense linear algebra operations, such as the Cholesky decomposition [17, 26, 25].

9 Acknowledgements

The authors would like to acknowledge the Texas Advanced Computing Center (TACC) for providing access to the IBM P690s and Cray T3E machines, along with other computing resources, used in the development of this study. Support for this research was provided by NASA grant NAG5-97213 and NSF grant ACI-0203685. Additional support for this work was provided by the NASA Earth System Science Fellowship program and the Dolores Liebmann Fellowship program.

More Information

For more information on PLAPACK and POOCLAPACK visit

`http://www.cs.utexas.edu/users/plapack`

References

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

[2] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[3] C. Bischof and C.F. van Loan. The wy representation for products of householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):s2-s13, January 1987.

[4] Ake Bjorck. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996.

[5] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[6] R. Coleman, B. Leback, R. Norin, D. Scott, and K. Van de Houten. Soz - a dense, out-of-core solver with partial pivoting for the ipsc/860: A case history. In *1992 Annual Users Conference*, 1992.

[7] 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.

[8] J.W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[9] Jack Dongarra, Robert van de Geijn, and David Walker. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput.*, 22(3), Sept. 1994.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[11] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[12] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.

[13] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[15] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[16] Brian C. Gunter. Parallel least squares analysis of simulated GRACE data. Master's thesis, Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin, 2000.

[17] Brian C. Gunter, Wesley C. Reiley, and Robert A. van de Geijn. Parallel out-of-core Cholesky and QR factorizations with POOCLAPACK. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2001.

[18] Brian C. Gunter, Byron D. Tapley, and Robert A. van de Geijn. Advanced parallel least squares algorithms for grace data processing. In *Proc. of the International Association of Geodesy (IAG) Conference*, Budapest, Hungary, September 2-7 2001.

[19] Brian C. Gunter, Robert A. van de Geijn, and Wesley C. Reiley. Parallel out-of-core cholesky and qr factorizations with pooclapack. In *Proc. of the 15th International Parallel and Distributed Processing Symposium*, San Francisco, CA, April 23-27 2001.

[20] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[21] Ken Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*, volume III - Algorithms and Applications, pages 29–33, 1995.

[22] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[23] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. Technical Report TR-04-92, Harvard University, Center for Research in Computing Technology, Jan. 1992.

[24] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft.*, 29(2):218–243, July 2003.

[25] Wesley C. Reiley. Efficient parallel out-of-core implementation of the Cholesky factorization. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Dec. 1999. Undergraduate Honors Thesis.

[26] Wesley C. Reiley and Robert A. van de Geijn. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Nov. 1999.

[27] David S. Scott. Parallel I/O and solving out-of-core systems of linear equations. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 123–130, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[28] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[29] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[30] P. E. Strazdins. Optimal load balancing techniques for block-cyclic decompositions for matrix factorization. Submitted to PDCN'98.

[31] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation. In *Proceedings of IOPADS '96*, 1996.

[32] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[33] D.S. Watkins. *Fundamentals of Matrix Computations*. J. Wiley and Sons, New York, 1991.