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

BRIAN C. GUNTER and ROBERT A. VAN DE GEIJN
The University of Texas at Austin

This article 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 "realizable" peak of the architectures on which the experiments were performed.

Categories and Subject Descriptors: G.1.10 [**Numerical Analysis**]: Applications; G.4 [**Mathematical Software**]—*Parallel and vector implementations*; J.2 [**Physical Sciences and Engineering**]—*Earth and atmospheric sciences*

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: Linear algebra, dense systems, linear least squares

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, particularly with regards to the determination of high resolution

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 and the Delores Liebmann Fellowship program.
For more information on PLAPACK and POOCLAPACK visit http://www.cs.texas.edu/users/plapack.

Authors' addresses: B. C. Gunter, Center for Space Research, The University of Texas at Austin, Austin, TX 78712; email: gunter@csr.utexas.edu; R. A. van de Geijn, Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712; email: rvdg@cs.utexas.edu.

gravity field models. The estimation of these models involves the solution of large overdetermined linear least-squares problems, which often contain tens of thousands of parameters and millions of observations [Gunter et al. 2001b; Condi et al. 2003]. 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 [Gunter 2000], but later an out-of-core (OOC) implementation was created to handle even larger problems.

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

—It reviews the standard techniques for the high-performance implementation of the QR factorization based on the Householder transformation [Dongarra et al. 1986; Bischof and Van Loan 1987; Schreiber and Van Loan 1989; Elmroth and Gustavson 1998, 2000, 2001; Bjorck 1996].
—It extends these techniques to the problem of updating the QR factorization as additional batches of observations are added to the system [Dongarra et al. 1979].
—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 [D'Azevedo and Dongarra 1997; Rabani and Toledo 2001; Toledo and Rabani 2002]. 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 [Toledo and Gustavson 1996; Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a] and a possibly unstable variant of the LU factorization [Scott 1993], 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) [van de Geijn 1997] and its out-of-core extension, the Parallel Out-of-Core Linear Algebra Package (POOCLAPACK) [Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a; Alpatov et al. 1997].
—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 [Gunter et al. 2001a], this article goes into considerably more depth.

This article is structured as follows: Section 2 briefly describes the notation used. 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 article 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 the context.

Algorithms in this article are given in a notation that we have recently adopted as part of the FLAME project [Gunnels et al. 2001; Quintana-Ortí and van de Geijn 2003]. The double lines in the partitioned matrices and vectors relate to how far into the matrices and vectors the computation has proceeded, indicating which parts are in their factored form and which parts are in their original form. We believe the notation to be intuitive, but suggest that the reader consult some of these related papers for further clarification.

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 [Golub and Van Loan 1996; Watkins 1991]. 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 does not need to be explicitly computed.

3.1 Householder Transformations (Reflectors)

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

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

where χ_1 equals the first element of x.

```
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 $0 \times 0$ and $b_T$ has 0 elements

while $n(A_{BR}) \neq 0$ do
    Repartition
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ 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 $\alpha_{11}$ and $\beta_1$ 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
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right)$ 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
```

Fig. 1. Unblocked Householder QR factorization.

We define the Householder vector associated with x 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 one column at a time. 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.

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

while $n(A_{BR}) \neq 0$ do
    Determine block size $k$
    Repartition
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$ 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 $T_1$ 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 $\left[\left(\frac{\{Y\backslash R\}_{11}}{Y_{21}}\right), \eta, b_1\right]$
    $\left(\frac{A_{12}}{A_{22}}\right) := \left(I + \left(\frac{Y_{11}}{Y_{21}}\right) T_1^T \left( Y_{11}^T \,|\, Y_{21}^T \right)\right) \left(\frac{A_{12}}{A_{22}}\right)$
    ============================================================
    Continue with
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$ 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
```

Fig. 2. Blocked Householder QR factorization.

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.

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 [Anderson et al. 1992; Dongarra et al. 1990, 1991]. We now review how to do so for the QR factorization [Schreiber and Van Loan 1989; Anderson et al. 1992].

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 $H_1 H_2 \cdots H_k = I + YTY^T$ 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 the 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 avoids having to recompute those matrices as part of the OOC implementation of the QR factorization, and results in a small but noticeable increase in performance [Elmroth and Gustavson 1998, 2000, 20001]. It should be noted that when factoring tall, thin matrices, the performance gain from saving the T matrices is much greater, in some cases a factor of three or more. The cases examined in this study, in particular the OOC algorithm presented later, work primarily on roughly square matrices, which explains why the performance gain is not as dramatic. In addition, while not implemented for this study, further optimizations can be gained for certain types of problems by formulating the T matrices in terms of Level-3 operations [Elmroth and Gustavson 2000] as opposed to the traditional method, which only incorporates Level-2 operations [Schreiber and Van Loan 1989].*

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.

Alternatively, one can think of this as follows: Append y to A to form $(A \mid y)$. 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 $(A \mid B)$. 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. *Again, 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 (see Note 1 for details).*

```
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 $0 \times 0$ and $B_T$ and $T_T$ have 0 rows

while $n(A_{BR}) \neq 0$ do
    Determine block size $k$
    Repartition
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$,
        $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$, 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
        $\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right)$,
        $\left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$, 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
```

Fig. 3. Blocked update of the right-hand-side matrix B using a "forward substitution-like" approach.

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 previous data. We now review how the QR factorization can be updated as additional batches of equations (i.e., observations) become available [Dongarra et al. 1979; Golub and Van Loan 1996; Watkins 1991].

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

$$\begin{pmatrix} A \\ C \end{pmatrix} = \bar{Q}\bar{R}. \tag{1}$$

```
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 $0 \times 0$ and $b_T$ has 0 elements

while $n(R_{BR}) \neq 0$ do
  Repartition
    $\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} R_{00} & r_{01} & R_{02} \\ \hline\hline r_{10}^T & \rho_{11} & r_{12}^T \\ \hline R_{20} & r_{21} & R_{22} \end{array}\right)$,
    $\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_0 & c_1 & C_2 \end{array}\right)$, 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 $\rho_{11}$ and $\beta_1$ are scalars and $c_1$ is a column
  ---------------------------------------------------------------
  $\left[\left(\begin{array}{c} 1 \\ \hline u_2 \end{array}\right), \eta, \beta_1\right] := h\left(\begin{array}{c} \rho_{11} \\ \hline c_1 \end{array}\right)$
  $\rho_{11} := \eta$
  $c_1 := u_2$
  $w^T := r_{12}^T + u_2^T C_2$
  $\left(\begin{array}{c} r_{12}^T \\ \hline C_2 \end{array}\right) := \left(\begin{array}{c} r_{12}^T - \beta_1 w^T \\ \hline C_2 - \beta_1 u_2 w^T \end{array}\right)$
  ---------------------------------------------------------------
  Continue with
    $\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} R_{00} & r_{01} & R_{02} \\ \hline r_{10}^T & \rho_{11} & r_{12}^T \\ \hline\hline R_{20} & r_{21} & R_{22} \end{array}\right)$,
    $\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_0 & c_1 & C_2 \end{array}\right)$, 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
```

Fig. 4. Unblocked update to a QR factorization.

A key observation is that the QR factorization of

$$\begin{pmatrix} R \\ C \end{pmatrix} \tag{2}$$

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.

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.*

```
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 $0 \times 0$ and $T_T$ has 0 rows

while $n(R_{BR}) \neq 0$ do
    Determine block size $k$
    Repartition
        $\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} R_{00} & R_{01} & R_{02} \\ \hline\hline R_{10} & R_{11} & R_{12} \\ \hline R_{20} & R_{21} & R_{22} \end{array}\right)$,
        $\left(\begin{array}{c} C_L \\ \hline\hline C_R \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} C_0 & C_1 & C_2 \end{array}\right)$, 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
        $\left(\begin{array}{c||c} R_{TL} & R_{TR} \\ \hline\hline R_{BL} & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} R_{00} & R_{01} & R_{02} \\ \hline R_{10} & R_{11} & R_{12} \\ \hline\hline R_{20} & R_{21} & R_{22} \end{array}\right)$,
        $\left(\begin{array}{c||c} C_L & C_R \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} C_0 & C_1 & C_2 \end{array}\right)$, 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
```

Fig. 5. Blocked update to a QR factorization.

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

$$\begin{pmatrix} A \\ C \end{pmatrix} X = \begin{pmatrix} B \\ D \end{pmatrix},$$

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 $\left(\begin{smallmatrix} R \\ C \end{smallmatrix}\right)$ with $\left(\begin{smallmatrix} \bar{R} \\ Y^{(C)} \end{smallmatrix}\right)$, also computing $T^{(C)}$ as in Figure 5.
—Update B by forward substitution as in Figure 3.
—Update $\left(\begin{smallmatrix} B \\ D \end{smallmatrix}\right)$ 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.

```
Partition  $B \rightarrow \left(\frac{B_T}{B_B}\right)$, $C \rightarrow \left( C_L \,\|\, C_R \right)$, and $T \rightarrow \left(\frac{T_T}{T_B}\right)$
  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
        $\left( C_L \,\|\, C_R \right) \rightarrow \left( C_0 \,\|\, C_1 \,|\, C_2 \right)$,
        $\left(\frac{B_T}{B_B}\right) \rightarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$, and $\left(\frac{T_T}{T_B}\right) \rightarrow \begin{pmatrix} T_0 \\ \hline T_1 \\ \hline T_2 \end{pmatrix}$
      where  $C_1$ has $k$ columns and $B_1$ and $T_1$ have $k$ rows
    ----------------------------------------------------------------
    $\left(\frac{B_1}{D}\right) := \left( I + \left(\frac{I}{C_1}\right) T_1^T \left( I \,|\, C_1^T \right) \right) \left(\frac{B_1}{D}\right)$
    ----------------------------------------------------------------
    Continue with
        $\left( C_L \,\|\, C_R \right) \leftarrow \left( C_0 \,|\, C_1 \,\|\, C_2 \right)$,
        $\left(\frac{B_T}{B_B}\right) \leftarrow \begin{pmatrix} B_0 \\ \hline B_1 \\ \hline B_2 \end{pmatrix}$, and $\left(\frac{T_T}{T_B}\right) \leftarrow \begin{pmatrix} T_0 \\ \hline T_1 \\ \hline T_2 \end{pmatrix}$
enddo
```

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

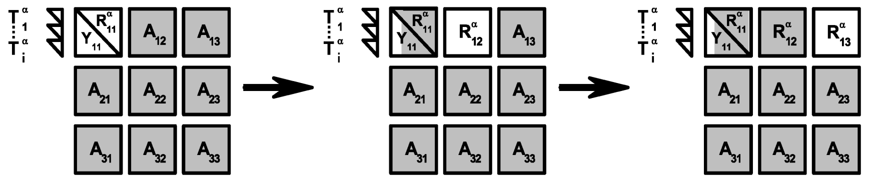
5. OUT-OF-CORE ALGORITHMS

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.

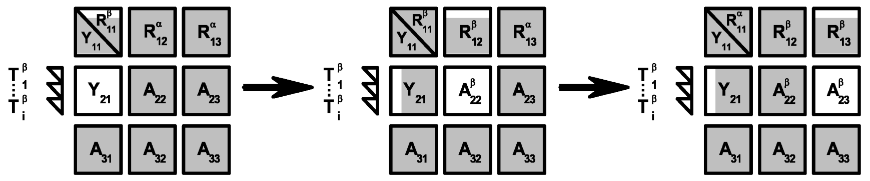
5.1 Out-of-Core QR Factorization

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 [Coleman et al. 1992; Klimkowski and van de Geijn 1995; D'Azevedo and Dongarra 1997; Toledo and Gustavson 1996; Scott 1993; Toledo and Rabani 2002]. 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.

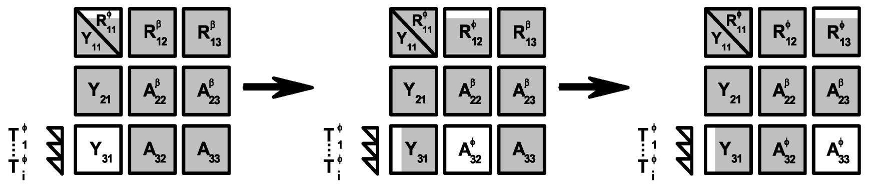
The alternative that we have found to the slab approach is to work with the matrix as a collection of tiles, where a tile is a submatrix that is roughly square. As was shown for the OOC Cholesky factorization [Toledo and Gustavson 1996;



(a) A_{11} is factored using the in-core algorithm, then the remaining tiles in the first row are updated using the Householder vectors stored in the lower triangular portion of A_{11}. The Householder vectors are read and applied in narrow panels of width r.



(b) The second row of tiles is now processed. A_{21} is factored using the modified in-core algorithm. As in step a, the Householder vectors are applied in panels of width r, while the updates to the first row of tiles are done in horizontal panels of height r. Note the need to create a new set of T matrices.



(c) The last tile row is factored, with the first row of tiles receiving its final update. As before, new T matrices are created. The second tile row is not affected.

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

Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a], a tiled approach provides true scalability. We will see that the processing of these tiles becomes a 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

($O(t^3)$)—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}, which can be discarded as soon as it has been used to update A_{12}, needs to be brought into memory. 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 $O(kt^2)$ computations. 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 newly 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}, which can be written back to disk as soon as it has been used to update A_{21}, needs to be brought into memory. 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 is 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 becomes, 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 A_{12} in row panels of height k is also updated. 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 5.1–5 start to repeat: A_{22} is factored as A_{11} was in Step 5.1. 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 5.1 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 the 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 rows of tiles were 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 article, 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 [Hendrickson and Womble 1994; Stewart 1990]. Moreover, to ensure load-balance as the active part of the matrix shrinks, an overdecomposition and wrapping of the matrix is typically employed [Lichtenstein and Johnsson 1992; Strazdins 1998; van de Geijn 1997; Dongarra et al. 1994].

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 straightforward. 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) [Choi et al. 1992]. 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 that suggests that keeping two tiles in memory may lead to better performance. The two tile approach also leads to a simpler implementation.

First, a few details about the storage of matrices. The matrices assigned locally to each processor as parts 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 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 essentially portable to any platform that supports the Message-Passing Interface [Gropp et al. 1994; Snir et al. 1996] and the Basic Linear Algebra Subprograms [Lawson et al. 1979; Dongarra et al. 1988, 1990]. 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 article 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 to be the peak performance that can be attained per processor, or the "realizable" peak of the system. The performance of our OOC implementation will be reported as a percentage of this realizable peak by the ratio

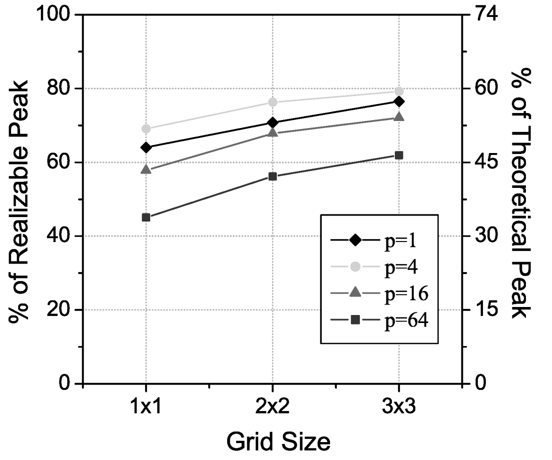
$$\frac{R_p(m)}{R_{dgemm}}.$$

Depending on the architecture, this realizable peak is 70–99% of the theoretical peak of the processor, which is defined by the clock speed multiplied by the number of floating point operations that can be performed per clock cycle. We believe that reporting performance relative to the realizable peak gives a clearer insight into the overhead incurred by the parts of the QR factorization that are not cast in terms of matrix-matrix multiplication, the overhead due to the parallelization, and the overhead due to I/O.

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

T3E Performance



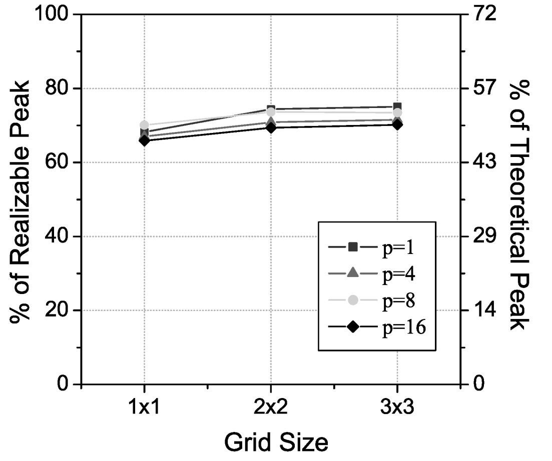
Problem Size

# Procs	1	4	8	16
Tile Size	2088	4704	8448	18432
n (1 x 1)	2088	4704	8448	18432
n (2 x 2)	4176	9408	16896	36864
n (3 x 3)	6264	14112	25344	55296

Compute Time (sec.)

# Procs	1	4	8	16
n (1 x 1)	43	112	195	651
n (2 x 2)	308	818	1332	4176
n (3 x 3)	962	2657	4231	12776

IBM Performance



Problem Size

# Procs	1	4	8	16
Tile Size	7900	21000	30000	42000
n (1 x 1)	7900	21000	30000	42000
n (2 x 2)	15800	42000	60000	84000
n (3 x 3)	23700	63000	90000	126000

Compute Time (sec.)

# Procs	1	4	8	16
n (1 x 1)	259	1238	1726	2518
n (2 x 2)	1900	9365	13130	19128
n (3 x 3)	6349	31286	44439	63770

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

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 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 casts 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 [Reiley and van de Geijn 1999; Reiley 1999; Gunter et al. 2001a]. 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 article. 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 above results 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 the realizable 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 the "realizable" 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 [Gunter 2000; Gunter et al. 2001b; Condi et al. 2003]. In addition, the tile-based approach is well suited for other types of dense linear algebra operations, such as the Cholesky decomposition [Gunter et al. 2001a; Reiley and van de Geijn 1999; Reiley 1999].

9. ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for providing their time and many insightful comments in reviewing this work.

REFERENCES

Alpatov, P., Baker, G., Edwards, C., Gunnels, J., Morrow, G., Overfelt, J., van de Geijn, R., and Wu, Y.-J. J. 1997. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*.

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

Bischof, C. and Van Loan, C. 1987. The WY representation for products of householder matrices. *SIAM J. Sci. Stat. Comput. 8*(1):s2-s13.

BJORCK, A. 1996. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

CHOI, J., DONGARRA, J. J., POZO, R., AND WALKER, D. W. 1992. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Press, 120–127.

COLEMAN, R., LEBACK, B., NORIN, R., SCOTT, D., AND DE HOUTEN, K. V. 1992. Soz - a dense, out-of-core solver with partial pivoting for the iPSC/860: A case history. In *1992 Annual Users Conference*.

CONDI, F., GUNTER, B., RIES, J., AND TAPLEY, B. 2003. Combining sea surface and terrestrial gravity data for global geopotential modelling and geoid determination. In *Eos Trans. AGU*, *84*(46), Fall Meet. Suppl., Abstract G31A-06.

D'AZEVEDO, E. F. AND DONGARRA, J. J. 1997. 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. (January)

DONGARRA, J., KAUFMANN, L., AND HAMMARLING, S. 1986. Squeezing the most out of eigenvalue solvers on high-performance computers. *Linear Algebra and It Applications* 77:113–136.

DONGARRA, J., VAN DE GEIJN, R., AND WALKER, D. 1994. Scalability issues affecting the design of a dense linear algebra library. *J. Parallel Distrib. Comput. 22*, 3 (September).

DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., AND STEWART, G. W. 1979. *LINPACK Users' Guide*. SIAM, Philadelphia.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft. 16*, 1 (March), 1–17.

DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND HANSON, R. J. 1988. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft. 14*, 1 (March), 1–17.

DONGARRA, J. J., DUFF, I. S., SORENSEN, D. C., AND VAN DER VORST, H. A. 1991. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA.

ELMROTH, E. AND GUSTAVSON, F. G. 1998. New serial and parallel recursive QR factorization algorithms for SMP systems. In *PARA*. 120–128.

ELMROTH, E. AND GUSTAVSON, F. G. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Dev. 44*, 4 (July), 605–624.

ELMROTH, E. AND GUSTAVSON, F. G. 2001. A faster and simpler recursive algorithm for the LAPACK routine DGELS. *BIT 41*, 5, 936–949.

GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, Baltimore, MD.

GROPP, W., LUSK, E., AND SKJELLUM, A. 1994. *Using MPI*. The MIT Press.

GUNNELS, J. A., GUSTAVSON, F. G., HENRY, G. M., AND VAN DE GEIJN, R. A. 2001. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft. 27*, 4 (December), 422–455.

GUNTER, B. C. 2000. Parallel least squares analysis of simulated GRACE data. CSR Technical Memoranda CSR-TM-00-05, The Center for Space Research, The University of Texas at Austin.

GUNTER, B. C., REILEY, W. C., AND VAN DE GEIJN, R. A. 2001a. 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.

GUNTER, B. C., TAPLEY, B. D., AND VAN DE GEIJN, R. A. 2001b. Advanced parallel least squares algorithms for GRACE data processing. In *Proceedings of the International Association of Geodesy (IAG) Conference*. Budapest, Hungary.

HENDRICKSON, B. A. AND WOMBLE, D. E. 1994. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput. 15*, 5, 1201–1226.

KLIMKOWSKI, K. AND VAN DE GEIJN, R. 1995. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*. Vol. III—Algorithms and Applications. 29–33.

LAWSON, C. L., HANSON, R. J., KINCAID, D. R., AND KROGH, F. T. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft. 5*, 3 (September), 308–323.

LICHTENSTEIN, W. AND JOHNSSON, S. L. 1992. Block-cyclic dense linear algebra. Tech. Rep. TR-04-92, Harvard University, Center for Research in Computing Technology. Jan.

QUINTANA-ORTÍ, E. S. AND VAN DE GEIJN, R. A. 2003. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft. 29*, 2 (July), 218–243.

Rabani, E. and Toledo, S. 2001. Out-of-core SVD and QR decompositions. In *In Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing (PARA)*. Norfolk, Virginia.

Reiley, W. C. 1999. Efficient parallel out-of-core implementation of the Cholesky factorization. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. (December) Undergraduate Honors Thesis.

Reiley, W. C. and van de Geijn, R. A. 1999. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Tech. Rep. CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin. (November)

Schreiber, R. and Van Loan, C. 1989. A storage-efficient WY representation for products of householder transformations. *SIAM J. Sci. Stat. Comput. 10*, 1 (January), 53–57.

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

Snir, M., Otto, S. W., Huss-Lederman, S., Walker, D. W., and Dongarra, J. 1996. *MPI: The Complete Reference*. The MIT Press.

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

Strazdins, P. 1998. Optimal load balancing techniques for block-cyclic decompositions for matrix factorization. Tech. Rep. TR-CS-98-10, Canberra 0200 ACT, Australia.

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

Toledo, S. and Rabani, E. 2002. Very large electronic structure calculations using an out-of-core filter-diagonalization method. *J. Comp. Phys. 180*, 256–269.

van de Geijn, R. A. 1997. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press.

Watkins, D. 1991. *Fundamentals of Matrix Computations*, 2nd ed. J. Wiley and Sons, New York.

Received July 2003; revised January 2004 and June 2004; accepted June 2004