UPDATING AN LU FACTORIZATION AND ITS APPLICATION TO SCALABLE OUT-OF-CORE [*]

THIERRY JOFFRAIN[†], ENRIQUE S. QUINTANA-ORTÍ[‡], AND ROBERT VAN DE GEIJN[†]

Abstract. How to achieve a scalable out-of-core implementation of the LU factorization while maintaining numerical stability akin to partial pivoting is a decades-old question. Here we use the term scalable to mean that performance is maintained as the matrix size grows well beyond available (in-core) memory. We show how partitioning the matrix into tiles and restricting the permutation of rows to be between at most two such tiles achieve these goals. The resulting method enables the solution of large-scale problems using limited computational resources and/or reduces the cost of an architecture by reducing the amount of memory that needs to be purchased. An implementation using the Formal Linear Algebra Methods Environment (FLAME) Application Programming Interface (API) is described. Performance experiments demonstrate high performance on an Intel Itanium2® based server.

Key words. LU factorization, Out-of-Core algorithms, linear systems, pivoting.

AMS subject classifications. 65F05, 65Y10.

1. Introduction.

Numerical linear algebra is an area that is prone to consuming vast amounts of computer memory. When the data structures that store the problem are too large to fit in the memory of the computer, the solution is to rely on disk storage. The reason is simple: disk space is cheaper and available in a larger quantity. Although such additional storage on disk can be accessed via virtual memory, careful modification of the algorithms is generally required to attain high performance. Such strategy leads to so-called out-of-core (OOC) algorithms.

In this paper we consider the solution of nonsymmetric dense linear systems via the LU factorization with pivoting. As the computation of this factorization is the real challenge, we do not discuss the solution of the resulting triangular systems.

1.1. Target applications.

Practical applications arising in Boundary Element Methods (BEM) often lead to very large dense linear systems. The idea there is that by placing the discretization on the boundary of a three-dimensional object, the degrees of freedom are restricted to a two-dimensional surface. In contrast, Finite Element Methods (FEM) set degrees of freedom throughout the three dimensional object. While the FEM methods result in large sparse matrices, BEM methods result in dense matrices, with a much smaller dimension. Here "smaller" is a relative term: problems with hundreds of thousands or even millions of degrees of freedom are not uncommon [6, 8, 11], leading to linear systems of that order.

For many of these applications the goal is to optimize a feature of an object. For example, BEM may be used to model the radar signature of an airplane. In an effort to minimize this signature, it may be necessary to optimize the shape of a certain component of the airplane. If the degrees of freedom associated with this component are ordered last among all degrees of freedom, the linear system may present the

[*]This paper expands on results reported in an earlier conference paper [19].

[2]Dept. of Computer Sciences, The University of Texas, Austin, TX 78712; phone: 512-4719720, fax: 512-4718885, {joffrain,rdvg}@cs.utexas.edu.

[3]Dept. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12071 Castellón, Spain; phone: +34 964-728257, fax: +34 964-728486, quintana@icc.uji.es.

structure

$$\begin{pmatrix} B & C \\ D & E \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}. \tag{1.1}$$

Now, as the shape of the component is modified, it is only the matrices C, D, and E that change, together with the right-hand side of the equation. Since the dimensions of B are frequently much larger than those of the remaining three matrices, it becomes of interest to factorize B only once, and to *update the factorization* as C, D, and E change. We note here that a standard LU factorization with partial pivoting does not provide a convenient solution to this problem, since the rows to be swapped during the application of the permutations may not lie only within B.

Remark: *The methodology proposed in this paper applies to the problem stated in (1.1). It is the insights regarding this simpler problem that provide the key to a high-performance implementation of the scalable OOC LU factorization with pivoting.*

1.2. OOC LU factorization. A commonly used library for sequential computers and conventional shared-memory parallel computers since the 1990s is the Linear Algebra Package (LAPACK) [1]. This package does not explicitly include OOC capabilities, although on machines with virtual memory the library can be used to solve problems larger than fit in-core.

Very large problems are typically solved on distributed-memory supercomputers. Therefore recent papers on OOC solution of dense linear systems typically target parallel computers, on which a version of LAPACK called ScaLAPACK [5] can be used. This extension of LAPACK does provide prototype OOC implementations of some of the ScaLAPACK routines, including solvers for general and symmetric positive definite linear systems using, respectively, the LU and Cholesky factorizations, and solvers for linear least squares problems via the QR factorization [7].

A more thorough effort to add OOC capabilities to ScaLAPACK was provided by SOLAR [27], a portable library for scalable OOC linear algebra computations. This library uses ScaLAPACK routines for in-core computation, and also provides an input-ouput (I/O) layer that manages matrix I/O. SOLAR achieves better I/O rates by allowing a different storage scheme for matrices on disk from that used in-core by ScaLAPACK. Impressive performance is reported for up to four nodes of an IBM SP-2. Lack of performance on a larger number of nodes is in part blamed on nonscalability of some of the in-core parallel kernels used there.

We developed the Parallel Linear Algebra Package (PLAPACK) in the mid-1990s [28] as an alternative to ScaLAPACK. An OOC extension to PLAPACK, POOCLAPACK, was introduced shortly later. To date, that infrastructure has been used for the parallel OOC implementation of tile-based Cholesky and QR factorizations [16, 15].

It should be noted that the above described parallel OOC library efforts are in addition to a number of parallel OOC implementations of individual operations or machine specific libraries for dense linear systems reported in the literature [2, 22, 4, 23, 24]. For a complete survey, see [26].

1.3. OOC computation and stability. The fundamental problem with the OOC LU factorization is that in order to provide stability, partial pivoting has been traditionally used with the algorithm. (Note however that LU factorization with partial pivoting is not a numerically stable algorithm [18]; it is only practice that taught us to trust it.) If partial pivoting is used, when choosing the row to be swapped, the entire column from which it is chosen must have been updated up to the same stage of the computation. Moreover, so as to make access to this column inexpensive, the part of the column on and below the diagonal must be in memory. As a result, so-called slab approaches have been adopted that proceed by bringing blocks of columns of the matrix into memory at a time. However, these methods are inherently not scalable: as the overall matrix problem grows, the row dimension of the slab increases, and the number of columns that can fit in memory decreases. Since the ratio between the computation performed with a slab and the I/O required for that computation is proportional to the number of columns in the slab, eventually the cost of I/O becomes significant.

Thus, it was widely recognized that working with so-called (square) tiles was preferable [26, 16, 15]. As the overall matrix size increases, the size of the tile brought into memory can be kept constant, and thus also the ratio between the computation and I/O overhead. The problem with this approach is that it stands in the way of partial pivoting, since all the tiles containing part of a column would have to be brought into main memory. One solution to this has been to abandon partial pivoting and to pivot only within the tile on the diagonal, in the hope that this does not affect the accuracy of the solution. However, in general, this solution is not numerically satisfactory.

1.4. A one-tile approach for the LU factorization. Our own approach is different: In order to solve the problem stated in (1.1) we introduce the method of incremental pivoting, which maintains many of the benefits of partial pivoting. The insights we gain from studying this simpler problem result in a relatively simple, yet powerful design of a scalable OOC LU factorization. The implementation of this algorithm delivers both scalability and high-performance. It has been brought to our attention that an unblocked algorithm similar to our algorithm was reported in [29].

As the computation of an OOC LU factorization goes back to the time when memories could only hold a few Kilobytes or less, our approach provides a solution to a problem that dates back to the early days of computing.

1.5. Overview. This paper expands on results reported in an earlier conference paper [19]. It is organized as follows: In Section 2 we review two algorithms for computing the LU factorization with partial pivoting. Next, in Section 3, we discuss how to update an LU factorization by considering the factorization of a 2×2 blocked matrix. This study will help us to present a scalable OOC algorithm for the LU factorization with incremental pivoting in Section 4. Numerical stability is discussed in Section 5 and performance is reported in Section 6. Concluding remarks are given in the final section.

2. The LU factorization with partial pivoting.

Given an $n \times n$ matrix A, its LU factorization with partial pivoting is given by

$$PA = LU, \tag{2.1}$$

where P is a permutation matrix of order n, L is $n \times n$ lower triangular, and U is $n \times n$ upper triangular. We will denote the computation of P, L, and U by

$$[A,\ p] := [\{L\backslash U\},\ p] = LU(A), \tag{2.2}$$

where $\{L\backslash U\}$ is the matrix whose strictly lower and upper triangular parts equal L and U, respectively. Here we recognize that L has ones on the diagonal, which need not be stored, and that the factors L and U can be stored overwriting the original contents of A. The permutation matrix is generally stored in a vector p of n integers.

```
Algorithm: $[A,\ p] := [\{L\backslash U\},\ p] = \mathrm{LU}_{\mathrm{unb}}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$ and $p_T$ has 0 elements
while $n(A_{TL}) < n(A)$ do
    Repartition
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$
        where $\alpha_{11}$ and $\pi_1$ are scalars

    LINPACK algorithm:
        Compute $\pi_1$
        $\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right) :=$
            $P(\pi_1)\left(\begin{array}{c|c} \alpha_{11} & a_{12}^T \\ \hline a_{21} & A_{22} \end{array}\right)$
        $\alpha_{11} := \mu_{11} = \alpha_{11}$ (no-op)
        $a_{12}^T := u_{12}^T = a_{12}^T$ (no-op)
        $a_{21} := l_{21} = a_{21}/\mu_{11}$
        $A_{22} := A_{22} - l_{21}u_{12}^T$

    LAPACK algorithm:
        Compute $\pi_1$
        $\left(\begin{array}{c||c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) :=$
            $P(\pi_1)\left(\begin{array}{c||c|c} a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
        $\alpha_{11} := \mu_{11} = \alpha_{11}$ (no-op)
        $a_{12}^T := u_{12}^T = a_{12}^T$ (no-op)
        $a_{21} := l_{21} = a_{21}/\mu_{11}$
        $A_{22} := A_{22} - l_{21}u_{12}^T$

    Continue with
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline \pi_1 \\ \hline p_2 \end{array}\right)$
endwhile
```

FIG. 2.1. *Unblocked algorithms for the LU factorization.*

2.1. Unblocked right-looking LU factorization. The LU factorization is obtained by means of a triangularization procedure also known as Gaussian elimination [12]: A sequence of permutation matrices $P_1, P_2, \ldots, P_n$ and Gauss (or elementary) transformation matrices $L_1, L_2, \ldots, L_n$ are computed to reduce matrix A to upper triangular form (i.e., to reduce the original matrix to U). Gauss transforms only depart from the identity matrix in the subdiagonal elements of one of its columns and are trivial to invert (L_i^{-1} only differs from L_i in the subdiagonal elements in the ith column, which have their signs inverted). It is the accumulation of the Gauss transforms that becomes L, with the peculiarity that the ith column of L equals the ith column of L_i.

The computations that need to be performed in the Gaussian elimination procedure can be organized in different manners. Several variants of the LU factorization are identified in [25, 13] which differ in the order the computations are performed and would produce the same numerical results if exact arithmetic was employed.

Two algorithms used to compute the LU factorization of a matrix A are given in Fig. 2.1. In the algorithm, $n(\cdot)$ stands for the number of columns of a matrix; $P(\pi_1)$ is the permutation matrix constructed by interchanging the first and the π_1-th row of the identity matrix; and variables μ_{11}, u_{12}^T, and l_{21} are only introduced to relate the computations to the various parts of L and U. We believe the rest of the notation to be intuitive. Both algorithms correspond to what is usually known as the right-looking variant; that is, an algorithm which, at a given stage, updates the current column of the matrix by means of a Gauss transform and then applies this transformation to the part of the matrix to the right of this column. However, they differ in the part of the matrix to which the permutations are applied. The algorithm on the left

```
**Algorithm:** $[A,\ p] := [\{L\backslash U\},\ p] = \mathrm{LU}_{\mathrm{blk}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$ and $p_T$ has 0 elements
while $n(A_{TL}) < n(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$ and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
        where $A_{11}$ is $b \times b$ and $p_1$ has $b$ elements

    $\left[\left(\frac{A_{11}}{A_{21}}\right),\ p_1\right] := \left[\left(\frac{\{L_{11}\backslash U_{11}\}}{L_{21}}\right),\ p_1\right] = LU\left(\left(\frac{A_{11}}{A_{21}}\right)\right)$ % LAPACK algorithm
    $\left(\frac{A_{12}}{A_{22}}\right) := P(p_1)\left(\frac{A_{12}}{A_{22}}\right)$
    $A_{12} := U_{12} = L_{11}^{-1}A_{12} = \mathrm{trilu}\,(A_{11})^{-1}\,A_{12}$
    $A_{22} := A_{22} - L_{21}U_{12} = A_{22} - A_{21}A_{12}$

    Continue with
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$ and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
endwhile
```

FIG. 2.2. *LINPACK blocked algorithm for the LU factorization built upon an LAPACK panel factorization.*

corresponds to the implementation available in LINPACK [10], which computes an LU factorization of the form

$$L_n^{-1}P_n \cdots L_2^{-1}P_2L_1^{-1}P_1A = U. \tag{2.3}$$

Thus, the lower triangular matrix L is not explicitly available from this algorithm. Nevertheless, by permuting the rows of the Gauss transforms, the permutations matrices can be moved to the right, obtaining an LU factorization of the form

$$\hat{L}_n^{-1} \cdots \hat{L}_2^{-1}\hat{L}_1^{-1}P_n \cdots P_2P_1A = L^{-1}PA = U. \tag{2.4}$$

This is the idea behind the algorithm on the right of Fig. 2.1, which corresponds to an implementation that mimics the code from LAPACK [1], and delivers an LU factorization as defined in (2.2). Hereafter we will refer to the two implementations in Fig. 2.1 as the *LINPACK (unblocked) algorithm* and the *LAPACK (unblocked) algorithm*.

In case A is $m \times n$, with $m > n$, the two previous algorithms produce a "rectangular" LU factorization $PA = LU$, where P is $m \times m$, L is $m \times n$ lower trapezoidal, and U is $n \times n$. Matrices L and U are also stored overwriting A.

2.2. Blocked right-looking LU factorization. It is well-known that high-performance can be achieved in a portable fashion by casting algorithms in terms of matrix-matrix multiplication [20, 17, 21, 14]. We next review how to do so for the LINPACK algorithm. The reason to prefer this approach over the one in LAPACK will become clear in Section 3.

A right-looking blocked algorithm for the LU factorization that combines the LINPACK and LAPACK algorithms is presented in Fig. 2.2. There, $\mathrm{trilu}\,(A_{11})$ denotes the lower triangular matrix with ones on the diagonal and whose strictly lower

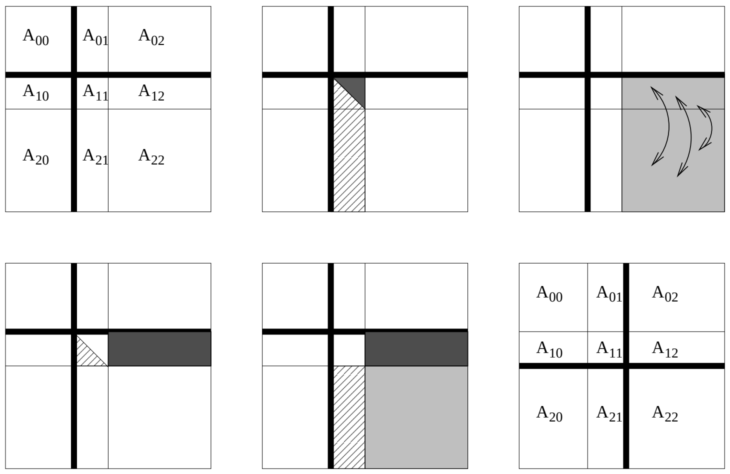

FIG. 2.3. *Computational steps in the LINPACK blocked algorithm for the LU factorization built upon an LAPACK panel factorization. Top row, from left to right: Initial state; LAPACK LU factorization of current panel; application of permutations. Bottom row, from left to right:* $A_{12} := U_{12} = L_{11}^{-1} A_{12}$; $A_{22} := A_{22} - L_{21} U_{12}$; *final state.*

triangular part corresponds to that of A_{11}. At each iteration of the algorithm, the current panel of columns, $\left(\frac{A_{11}}{A_{21}}\right)$, is factored using the LAPACK algorithm. Then, the parts of A to the right of this panel are updated with respect to the LU factors of the current column panel. By using an LAPACK algorithm for the factorization of the current panel, all pivots computed during the factorization of that panel can be applied to the remainder of the matrix. However, the pivots are only applied to $\left(\frac{A_{12}}{A_{22}}\right)$ (but not to $\left(\frac{A_{10}}{A_{20}}\right)$), which corresponds to the LINPACK manner of pivoting the matrix. After that, U_{12} is computed by solving the lower triangular system with multiple right-hand sides as $A_{12} := U_{12} = L_{11}^{-1} A_{12}$. Finally, A_{22} is updated by $A_{22} := A_{22} - L_{21} U_{12}$. This process is illustrated in Fig. 2.3.

Consider hereafter that A is composed of $n = \nu \cdot b$ columns (that is, n is an exact multiple of the block size, b). Then, the *LINPACK blocked algorithm* in Fig. 2.2 computes an LU factorization of the form

$$\tilde{L}_{\nu}^{-1} \tilde{P}_{\nu} \cdots \tilde{L}_{2}^{-1} \tilde{P}_{2} \tilde{L}_{1}^{-1} \tilde{P}_{1} A = U, \tag{2.5}$$

where $\tilde{L}_i$ and $\tilde{P}_i$ are the factors obtained from the factorization of the ith panel of A, composed of b columns, by means of the LAPACK algorithm.

Remark: *The LINPACK blocked algorithm can be easily converted into an LAPACK blocked algorithm by applying the permutations computed during the factorization of the current panel also to the blocks to the left of this panel, that is,*

$$\left(\frac{A_{10}}{A_{20}}\right) := P(p_1) \left(\frac{A_{10}}{A_{20}}\right).$$

All these blocked algorithms attain high performance on modern architectures with (multiple levels of) cache memory by casting the bulk of the computation in terms of the matrix-matrix multiplication $A_{22} := A_{22} - L_{21} U_{12}$.

3. The LU factorization with incremental pivoting.

In this section we discuss how to compute the LU factorization of the matrix

$$\begin{pmatrix} B & C \\ D & E \end{pmatrix} \tag{3.1}$$

in such a way that the LU factorization with partial pivoting of B,

$$PB = LU, \tag{3.2}$$

can be reused if C, D, and/or E change. For simplicity, we consider all four submatrices in (3.1) to be of size $t \times t$, with t an exact multiple of the block size b: $t = \tau \cdot b$. For reference, factoring the matrix in (3.1) using the standard LU factorization with partial pivoting costs $\frac{2}{3}(2t)^3 = \frac{16}{3}t^3$ flops (floating-point arithmetic operations). In this expression (and any other following referred to computational costs) we neglect those terms of lower-order complexity, including the cost of pivoting the rows. Similarly, we will also ignore lower-order terms in storage requirements.

We propose to employ the following procedure, consisting of 5 steps, which computes an *LU factorization with incremental pivoting* of the matrix in (3.1):

Step 1: Factor B**.** Compute the LU factorization with partial pivoting

$$PB = LU, \tag{3.3}$$

overwriting B with L and U.

Step 2: Update C consistently with the factorization of B:

$$C := L^{-1}PC. \tag{3.4}$$

Step 3: Factor $\left(\frac{U}{D}\right)$**.** Compute the LU factorization with partial pivoting

$$\bar{P}\left(\frac{U}{D}\right) = \left(\frac{\bar{L}_1}{L_2}\right)\bar{U} = \bar{L}\bar{U}. \tag{3.5}$$

Additional storage is required at this stage to store $\bar{L}_1$ as U is stored in the array for B, which also contains the lower triangular matrix L from the LU factorization of the previous step. Matrix $\bar{U}$ overwrites the triangular factor U in B.

Step 4: Update $\left(\frac{C}{E}\right)$ consistently with the factorization of $\left(\frac{U}{D}\right)$. For that purpose, first apply the pivots generated in the previous step as

$$\left(\frac{C}{E}\right) := \bar{P}\left(\frac{C}{E}\right), \tag{3.6}$$

and then compute

$$\left(\frac{C}{E}\right) := \left(\frac{\bar{L}_1^{-1}C}{E - \bar{L}_2(\bar{L}_1^{-1}C)}\right). \tag{3.7}$$

Step 5: Factor E**.** Finally, compute the LU factorization with partial pivoting

$$\hat{P}E = \hat{L}\hat{U}, \tag{3.8}$$

overwriting E with $\hat{L}$ and $\hat{U}$.

Operation	Cost (in flops)		
	W/out exploiting structure in U	LINPACK blocked	LAPACK blocked
1: Factorize B	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$
2: Update C	t^3	t^3	t^3
3: Factorize $\left(\frac{U}{D}\right)$	$\frac{5}{3}t^3$	$t^3 + \frac{3}{2}t^2b$	$t^3 + \frac{3}{2}t^2b$
4: Update $\left(\frac{C}{E}\right)$	$3t^3$	$2t^3 + tb^2$	$3t^3$
5: Factorize E	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$
Total	$7t^3$	$\frac{16}{3}t^3 + \frac{3}{2}t^2b + tb^2$	$\frac{19}{3}t^3 + \frac{3}{2}t^2b$

TABLE 3.1

Computational cost (in flops) of the different approaches to compute the LU factorization of the matrix in (1.1).

Let us now analyze the cost of the above procedure. A first, naive approach computes the factorization in Step 3 ignoring any zeroes below the diagonal of U. Correspondingly, the update in Step 4 does not take advantage of the special structure either. This approach results in the costs stated in the column marked as "W/out exploiting structure in U" in Table 3.1, and requires $\frac{5}{3}t^3$ extra flops compared with the LU factorization with partial pivoting of (3.1). These additional flops correspond to the factorization of B in Step 1 and the following update of C in Step 2, which are basically "recomputed" during Steps 3 and 4. As discussed in Step 3, the procedure requires additional storage for a lower triangular $t \times t$ matrix.

The approach we describe in the next two subsections reduces both the computational and storage costs by exploiting the upper triangular structure of U during the LU factorization of $\left(\frac{U}{D}\right)$ and the corresponding update of $\left(\frac{C}{E}\right)$.

3.1. Exploiting structure in Step 3. A blocked algorithm that exploits the upper triangular structure of U is given in Fig. 3.1. The notation $\text{triu}(\cdot)$ there stands for the upper triangular part of a matrix; thus, for example, $\text{triu}(U_{11}) := \text{triu}(G_1)$ denotes that the upper triangular part of G_1 is copied into the upper triangular part of U_{11}. At each iteration of this algorithm, the panel of columns consisting of $\left(\frac{U_{11}}{D_1}\right)$ is factored using the LAPACK algorithm. Here we use the array G to store the blocks on the diagonal (that is, L_1) and thus ensure that the lower triangular factor L from Step 1 is not destroyed. In total, this requires an extra storage capacity for τ lower triangular blocks of dimension $b \times b$ each, which is negligible if $b \ll t$. The subdiagonal block L_2 is stored overwriting the elements annihilated in D_1.

After the current panel is factorized, the corresponding part of U and the remaining columns of D are updated with respect to the computed LU factors. As in the LINPACK blocked algorithm in Fig. 2.2, the LAPACK algorithm is used for the factorization of the current panel, but the pivots are only applied to $\left(\frac{U_{12}}{D_2}\right)$. After that, U_{12} is computed by solving the lower triangular system $U_{12} := L_1^{-1}U_{12}$, and D_2 is updated as $D_2 := D_2 - L_2U_{12}$. This process is illustrated in Fig. 3.2.

```
**Algorithm:** $\left[\left(\frac{U}{D}\right),\ G,\ p\right] := \left[\left(\frac{\bar{L}_1\backslash\bar{U}}{L_2}\right),\ G,\ p\right] = \mathrm{LU}^{UD}_{\mathrm{blk}}\left(\left(\frac{U}{D}\right)\right)$

**Partition** $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline U_{BL} & U_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $G \rightarrow \left(\begin{array}{c} G_T \\ \hline G_B \end{array}\right)$, and $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
    **where** $U_{TL}$ is $0 \times 0$, $D_L$ has 0 columns, $G_T$ has 0 rows, and $p_T$ has 0 elements
**while** $n(U_{TL}) < n(U)$ **do**
    **Determine block size** $b$
    **Repartition**
        $\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline U_{BL} & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline U_{10} & U_{11} & U_{12} \\ \hline U_{20} & U_{21} & U_{22} \end{array}\right)$, $\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right)$,
        $\left(\begin{array}{c} G_T \\ \hline G_B \end{array}\right) \rightarrow \left(\begin{array}{c} G_0 \\ \hline G_1 \\ \hline G_2 \end{array}\right)$, **and** $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
            **where** $U_{11}$ **is** $b \times b$, $D_1$ **has** $b$ **columns,** $G_1$ **has** $b$ **rows, and** $p_1$ **has** $b$ **elements**
    ─────────────────────────
    $G_1 := \mathrm{triu}\,(U_{11})$
    $\left[\left(\frac{G_1}{D_1}\right),\ p_1\right] := \left[\left(\frac{\{L_1\backslash U_1\}}{L_2}\right),\ p_1\right] = LU\left(\left(\frac{G_1}{D_1}\right)\right)$   % LAPACK algorithm
    $\left(\frac{U_{12}}{D_2}\right) := P(p_1)\left(\frac{U_{12}}{D_2}\right)$
    $U_{12} := L_1^{-1}U_{12} = \mathrm{trilu}\,(G_1)^{-1}\,U_{12}$
    $D_2 := D_2 - L_2U_{12} = D_2 - D_1U_{12}$
    $\mathrm{triu}\,(U_{11}) := \mathrm{triu}\,(G_1)$
    ─────────────────────────
    **Continue with**
        $\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline U_{BL} & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline U_{10} & U_{11} & U_{12} \\ \hline U_{20} & U_{21} & U_{22} \end{array}\right)$, $\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right)$,
        $\left(\begin{array}{c} G_T \\ \hline G_B \end{array}\right) \leftarrow \left(\begin{array}{c} G_0 \\ \hline G_1 \\ \hline G_2 \end{array}\right)$, and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ \hline p_2 \end{array}\right)$
**endwhile**
```

FIG. 3.1. *LINPACK blocked LU factorization of* $\left(U^T,\ D^T\right)^T$ *built upon an LAPACK panel factorization.*

The algorithm provides an LU factorization of the form

$$\tilde{L}_\tau^{-1}\tilde{P}_\tau \cdots \tilde{L}_2^{-1}\tilde{P}_2\tilde{L}_1^{-1}\tilde{P}_1\left(\frac{U}{D}\right) = \left(\frac{\bar{U}}{0}\right), \tag{3.9}$$

where $\tilde{L}_i$ and $\tilde{P}_i$ correspond to those factors produced from the factorization of the ith panel, composed of the ith block on the diagonal of U, of dimension $b \times b$, and the ith column block of D, of dimension $b \times t$.

The cost of the algorithm is $t^3 + \frac{2}{3}t^2b$ flops, which corresponds to having considered U as a block upper triangular matrix with blocks on the diagonal of dimension $b \times b$. Therefore, if $b \ll t$ the cost becomes t^3 and $\frac{2}{3}t^3$ flops are saved with respect to the naive algorithm.

Remark: *Using the LAPACK blocked algorithm to compute the LU factorization in Step 3 delivers the same cost for this step, that is,* $t^3 + \frac{2}{3}t^2b$ *flops. However, let us review the pivoting applied in this algorithm. At a certain iteration, after the LU factorization of the current panel is computed, the pivots have to be applied to* $\left(\frac{U_{10}}{D_0}\right)$ *as well. Now, as* U_{10} *contains part of the lower triangular factor* L *from the*

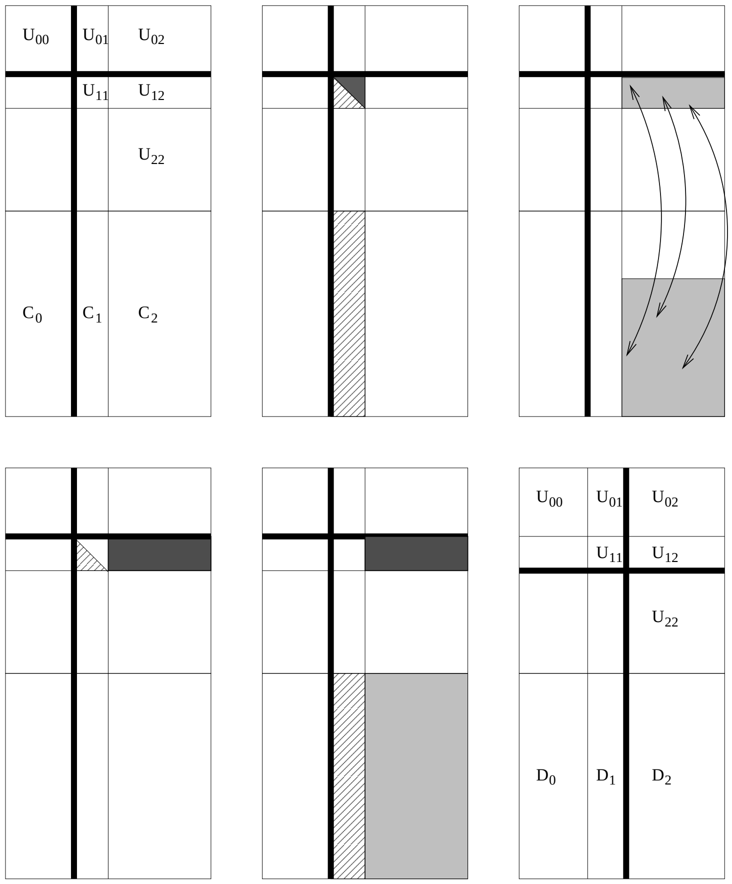

FIG. 3.2. *Computational steps in the LINPACK blocked algorithm for the LU factorization of* $\left(U^T,\ D^T\right)^T$ *built upon an LAPACK panel factorization. Top row, from left to right: Initial state; LAPACK LU factorization of current panel; application of permutations. Bottom row, from left to right:* $U_{12} := L_1^{-1} U_{12}$*;* $D_2 := D_2 - L_2 U_{12}$*; final state.*

LU factorization in Step 1, pivoting the rows in this block and those in D_0 *requires the use of an additional* $t \times t$ *lower triangular matrix in order to preserve* L*.*

3.2. Revisiting the update in Step 4. Given $\left(\frac{C}{E}\right)$, in this stage we need to update this matrix as

$$\left(\frac{C}{E}\right) := \tilde{L}_\tau^{-1} \tilde{P}_\tau \cdots \tilde{L}_2^{-1} \tilde{P}_2 \tilde{L}_1^{-1} \tilde{P}_1 \left(\frac{C}{D}\right), \tag{3.10}$$

where $\tilde{L}_i$ and $\tilde{P}_i$ correspond to those in (3.9). The algorithm for this purpose is given in Fig. 3.3. The update of $\left(\frac{C_1}{E}\right)$ performed in this algorithm is the same that was applied to $\left(\frac{U_{12}}{D_2}\right)$ in Fig. 3.1 as in the operations

$$\begin{array}{rcl} C_1 & := & \operatorname{trilu}\left(G_1\right)^{-1} C_1 \quad \text{and} \\ E & := & E - D_1 C_1, \end{array} \tag{3.11}$$

trilu(G_1) corresponds to L_1 and C_1 stores L_2.

The cost of performing the update in this manner is $2t^3 + tb^2$ flops, or $2t^3$ flops if $b \ll t$, saving thus t^3 flops with respect to the naive approach.

Remark: *Applying the LAPACK algorithm in this step requires the same amount of computations as the naive algorithm: the pivoting applied by the LAPACK algorithm in Step 3 destroys the structure of the lower triangular matrix, which cannot be easily recovered for the update in this step.*

Remark: *The algorithm described in this subsection is the same that would be applied to a right-hand side matrix* $\left(\frac{C}{E}\right)$ *when solving the lower triangular linear system resulting from the factorization in (3.9).*

The computational costs of the three approaches described in this section, namely, the naive algorithm, the LINPACK blocked algorithm, and the LAPACK blocked algorithm are summarized in Table 3.1. Assuming $b \ll t$, the use of the LINPACK blocked approach for Steps 2 and 3 effectively reduces the cost of the procedure for computing the LU factorization with incremental pivoting to that of the LU factorization with partial pivoting. The other two approaches, however, present a significant overhead in the number of computations. In addition, the LINPACK blocked algorithm only requires additional storage for τ lower triangular matrices of dimension $b \times b$ each, while the two other approaches need extra space for a $t \times t$ lower triangular matrix.

3.3. Putting it all together. We now review one last time how it all fits together by considering Figs. 3.5 and 3.4. In Fig. 3.4 all the algorithms for Steps 1–4 are condensed into one algorithm. By executing only one of the boxes in the body of the loop, the indicated step is performed. Columns in Fig. 3.5 illustrate the individual Steps 1–4 from Section 3.

4. Out-of-Core LU factorization.

In this section we first discuss traditional slab algorithms for the LU factorization. A simple analysis of the I/O cost shows the lack of scalability of these methods in the problem size. We then show how the insights from the previous section can be used to implement a scalable OOC LU factorization with incremental pivoting for matrices of arbitrary size.

4.1. Slab approaches. As mentioned, conventional OOC algorithms proceed by bringing entire columns of the matrix to be factored into memory in order to allow partial pivoting. This facilitates both the search for the pivot row and the swapping of rows.

Consider a left-looking blocked algorithm for the LU factorization with partial pivoting of an $n \times n$ matrix A partitioned as

$$A = \left(\begin{array}{c|c|c} A_0 & A_1 & A_2 \end{array} \right) = \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right), \tag{4.1}$$

where A_{00}, A_{11}, and A_{22} are square blocks of order r, s, and $n-s-r$, respectively. During the current iteration of the left-looking algorithm, slab A_1 (of width s) is first loaded into memory. The cost of this in terms of I/O operations (hereafter, iops), either loads or stores from the disk, is ns. Next, this slab is updated with respect to the Gauss transforms that were employed to introduce zeros in the subdiagonal entries of A_0 (of width r). Loading the lower trapezoidal part of A_0 into memory requires $nl - l^2/2$ additional iops. In practice, in order to allow the use of BLAS-3

```
Algorithm: $\left[\left(\frac{C}{E}\right)\right] := \mathrm{Update}^{CE}_{\mathrm{blk}}\left(\left(\frac{C}{E}\right),\ D,\ G,\ p\right)$

Partition $C \rightarrow \left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $G \rightarrow \left(\begin{array}{c} G_T \\ \hline G_B \end{array}\right)$, and $p \rightarrow \left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right)$
    where $D_L$ has 0 columns, $C_T$ and $G_T$ have 0 rows, and $p_T$ has 0 elements
while $n(D_L) < n(D)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ C_2 \end{array}\right)$, $\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right)$,
        $\left(\begin{array}{c} G_T \\ \hline G_B \end{array}\right) \rightarrow \left(\begin{array}{c} G_0 \\ \hline G_1 \\ G_2 \end{array}\right)$, and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \rightarrow \left(\begin{array}{c} p_0 \\ \hline p_1 \\ p_2 \end{array}\right)$
        where $D_1$ has $b$ columns, $C_1$ and $G_1$ have $b$ rows, and $p_1$ has $b$ elements

    $\left(\frac{C_1}{E}\right) := P(p_1)\left(\frac{C_1}{E}\right)$
    $C_1 := \mathrm{trilu}\,(G_1)^{-1}\,C_1$
    $E := E - D_1 C_1$

    Continue with
        $\left(\begin{array}{c} C_T \\ \hline C_B \end{array}\right) \leftarrow \left(\begin{array}{c} C_0 \\ C_1 \\ \hline C_2 \end{array}\right)$, $\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right)$,
        $\left(\begin{array}{c} G_T \\ \hline G_B \end{array}\right) \leftarrow \left(\begin{array}{c} G_0 \\ G_1 \\ \hline G_2 \end{array}\right)$, and $\left(\begin{array}{c} p_T \\ \hline p_B \end{array}\right) \leftarrow \left(\begin{array}{c} p_0 \\ p_1 \\ \hline p_2 \end{array}\right)$
endwhile
```

FIG. 3.3. *Update of $\left(C^T,\ E^T\right)^T$ consistent with the LINPACK blocked LU factorization of $\left(U^T,\ D^T\right)^T$.*

operations, A_0 is loaded into memory b columns at a time, and as soon as each one of these columns has been utilized, it is discarded from memory. Finally, $\left(\frac{A_{11}}{A_{21}}\right)$ is factorized, and A_1 is written back to disk. We refer to this algorithm as a one-slab approach.

In total, assuming that A is composed of n/s slabs, this algorithm performs approximately

$$\begin{aligned} \textstyle\sum_{i=0}^{(n/s)-1}\left[ns + nis - (is)^2/2 + ns\right] &\approx 2n^2 + n\tfrac{(n/s)^2}{2}s - \tfrac{(n/s)^3}{3}\tfrac{s^2}{2} \\ &= \tfrac{n^3}{3s} + 2n^2 \text{ iops} \end{aligned} \tag{4.2}$$

and $2/3n^3$ flops. Thus, the ratio of disk access overhead to useful computation is given by

$$\frac{n^3/(3s) + 2n^2}{2/3n^3} \approx \frac{1}{2s}, \tag{4.3}$$

showing that the performance we can expect from the algorithm improves with the width of the slab, s.

Now, we note that there are memory constraints: The memory of the system must be able to simultaneously hold two slabs, one of width s for the current panel A_1, and

Algorithm:	$[B,\ p] := \mathrm{LU}_{\mathrm{blk}}(B)$	$[C] := \mathrm{Update}^{C}_{\mathrm{blk}}(B,\ C,\ p)$
	$\left[\left(\frac{U}{D}\right),\ G,\ q\right] := \mathrm{LU}^{UD}_{\mathrm{blk}}\left(\left(\frac{U}{D}\right)\right)$	$\left[\left(\frac{C}{E}\right)\right] := \mathrm{Update}^{CE}_{\mathrm{blk}}\left(\left(\frac{C}{E}\right),\ D,\ G,\ q\right)$

```
Partition $X \to \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right)$, $X \in \{B, U\}$, $D \to \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$,
$Y \to \left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right)$, $Y \in \{C, G\}$, $z \to \left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right)$, $z \in \{p, q\}$
    where $X_{TL}$ is $0 \times 0$, $D_L$ has 0 columns, $Y_T$ has 0 rows, and $z_T$ has 0 elements
while $n(X_{TL}) < n(X)$ do
    Determine block size $b$
    Repartition
    $\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \to \left(\begin{array}{c|c|c} X_{00} & X_{01} & X_{02} \\ \hline X_{10} & X_{11} & X_{12} \\ \hline X_{20} & X_{21} & X_{22} \end{array}\right)$, $X \in \{B, U\}$,
    $\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \to \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right)$,
    $\left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right) \to \left(\begin{array}{c} Y_0 \\ \hline Y_1 \\ \hline Y_2 \end{array}\right)$, $Y \in \{C, G\}$, $\left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right) \to \left(\begin{array}{c} z_0 \\ \hline z_1 \\ \hline z_2 \end{array}\right)$, $z \in \{p, q\}$
        where $X_{11}$ is $b \times b$, $D_1$ has $b$ columns,
        $Y_1$ has $b$ rows, and $z_1$ has $b$ elements
    =====================================================================
    Step 1:
        $\left[\left(\frac{B_{11}}{B_{21}}\right),\ p_1\right] := LU\left(\left(\frac{B_{11}}{B_{21}}\right)\right)$
            (Note: $B_{11} := \{L_{11} \backslash U_{11}\}$)
        $\left(\frac{B_{12}}{B_{22}}\right) := P(p_1)\left(\frac{B_{12}}{B_{22}}\right)$
        $B_{12} := L_{11}^{-1} B_{12}$
        $B_{22} := B_{22} - B_{21} B_{12}$
    Step 2:
        (Note: $B_{11}$ contains $\{L_{11} \backslash U_{11}\}$)
        $\left(\frac{C_1}{C_2}\right) := P(p_1)\left(\frac{C_1}{C_2}\right)$
        $C_1 := L_{11}^{-1} C_1$
        $C_2 := C_2 - B_{21} C_1$
    Step 3:
        $G_1 :=$ triu $(U_{11})$
        $\left[\left(\frac{G_1}{D_1}\right),\ q_1\right] := LU\left(\left(\frac{G_1}{D_1}\right)\right)$
            (Note: $G_1 = \{L_1 \backslash U_1\}$)
        $\left(\frac{U_{12}}{D_2}\right) := P(q_1)\left(\frac{U_{12}}{D_2}\right)$
        $U_{12} := L_1^{-1} U_{12}$
        $D_2 := D_2 - D_1 U_{12}$
        triu $(U_{11}) :=$ triu $(G_1)$
    Step 4:
        $\left(\frac{C_1}{E}\right) := P(q_1)\left(\frac{C_1}{E}\right)$
        $C_1 :=$ trilu $(G_1)^{-1}\, C_1$
        $E := E - D_1 C_1$
    =====================================================================
    Continue with
    $\left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline X_{BL} & X_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} X_{00} & X_{01} & X_{02} \\ \hline X_{10} & X_{11} & X_{12} \\ \hline X_{20} & X_{21} & X_{22} \end{array}\right)$, $X \in \{B, U\}$,
    $\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right)$,
    $\left(\begin{array}{c} Y_T \\ \hline Y_B \end{array}\right) \leftarrow \left(\begin{array}{c} Y_0 \\ \hline Y_1 \\ \hline Y_2 \end{array}\right)$, $Y \in \{C, G\}$, and $\left(\begin{array}{c} z_T \\ \hline z_B \end{array}\right) \leftarrow \left(\begin{array}{c} z_0 \\ \hline z_1 \\ \hline z_2 \end{array}\right)$, $z \in \{p, q\}$
endwhile
```

FIG. 3.4. *All the algorithms for Steps 1–4 condensed into one algorithm.*

a second one of width b for a block of columns from A_0. Thus, if the total number of items that can be stored in memory is fixed at M, $(s + b)n \leq M$, or $s \leq M/n - b$, and the ratio of overhead to useful computation is approximately

$$\frac{1}{2s} \geq \frac{1}{2(M/n - b)}. \tag{4.4}$$

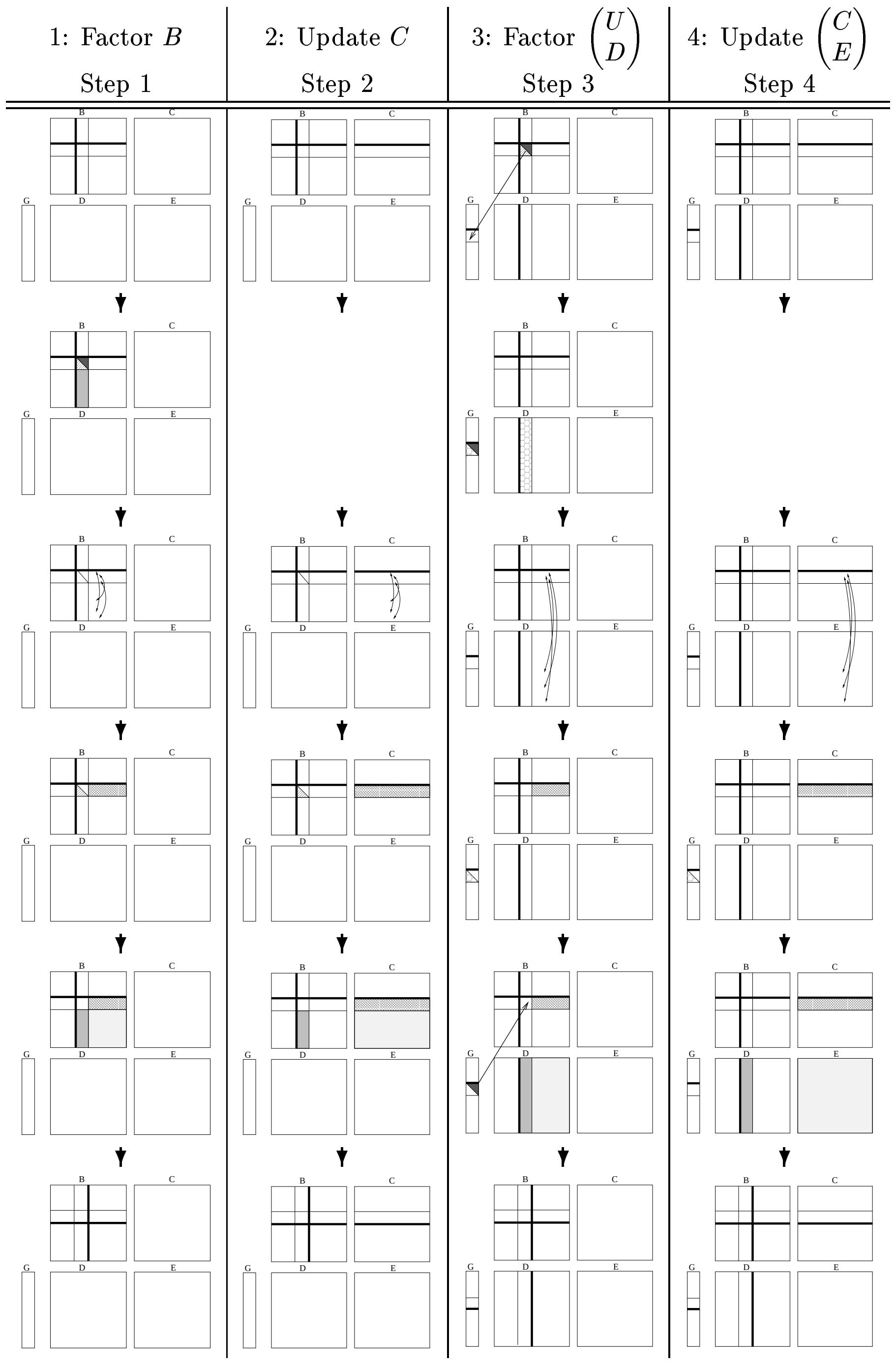
1: Factor B
Step 1
2: Update C
Step 2
3: Factor $\begin{pmatrix} U \\ D \end{pmatrix}$
Step 3
4: Update $\begin{pmatrix} C \\ E \end{pmatrix}$
Step 4
B
C
G
D
E

FIG. 3.5. *An illustration of Steps 1–4.*

The conclusion is that, as the problem size n grows, the overhead increases, ultimately leading to inefficiency.

Remark: *A slab-based right-looking algorithm for the LU factorization requires* *about twice as much I/O as a left-looking one as, at each iteration, a panel of the form* $\left(\frac{A_{11}}{A_{21}}\right)$ *must be brought into memory, factorized, and then* $\left(\frac{A_{12}}{A_{22}}\right)$ *must be loaded into memory b columns at a time, updated, and written back to disk. It is the need of writing back the matrix that doubles the I/O cost of the algorithm.*

4.2. A one-tile OOC algorithm. Now, let us re-examine the procedure for LU factorization with incremental pivoting of (3.1), where we assume that the combined matrix of dimension $2t \times 2t$ initially resides on disk.

Step 1: Factor B. This step requires t^2 iops for reading B from disk and t^2 to eventually write it back to disk.

Step 2: Update C. After reading C from disk, at a cost of t^2 iops, this matrix is permuted as $C := PC$, and updated by bringing blocks of b columns of the lower triangular part of B (which contains L) into memory one at a time. To explain this, consider the block of columns $\left(0^T,\ L_{11}^T,\ L_{21}^T\right)^T$, where L_{11} is lower triangular of dimension $b \times b$, and the conformal row partitioning $C = \left(C_0^T,\ C_1^T,\ C_2^T\right)^T$. Then, in this stage we need to perform the updates $C_1 := L_{11}^{-1} C_1$ and $C_2 := C_2 - L_{21} C_1$. Accessing L in this manner requires $t^2/2$ iops. At the end of this step, C is written back to disk with a cost of t^2 iops.

Step 3: Factor $\left(\frac{U}{D}\right)$. After reading D from disk, with a cost of t^2 iops, U and D are updated by bringing blocks of b rows of U into memory. Consider now the block of rows $(0,\ U_{11},\ U_{12})$, where U_{11} is upper triangular of dimension $b \times b$, and the conformal column partitioning $D = (D_0,\ D_1,\ D_2)$. During this stage, $\left(\frac{U_{11}}{D_1}\right)$ needs to be factored and $\left(\frac{U_{12}}{D_2}\right)$ is to be updated. This requires a total of t^2 iops (elements of U must be read and written back). Then, D is written back to disk at a cost of t^2 iops.

Step 4: Update $\left(\frac{C}{E}\right)$. Here, E is first read from disk at a cost of t^2 iops. The update is then performed by bringing into memory a block G_1 of dimension $b \times b$ from G, a block D_1 consisting of b rows from D, and a block C_1 of b columns from C. These are used to apply the pivots, compute $C_1 := \text{trilu}\,(G_1)^{-1}\, C_1$, and finally update $E := E - D_1 C_1$. This requires approximately $3t^2$ iops (as elements of C must be read and written back).

Step 5: Factor E. Finally E, which still resides in memory, is factored and written back to disk at a cost of t^2 iops.

Table 4.1 summarizes the number of iops for each of these steps. In this approach, at most one tile of dimension $t \times t$, and three blocks, of dimensions $b \times b$, $b \times t$, and $t \times b$ reside in memory at a given moment (see Step 4). Thus the classification as “one-tile”.

We next explain how to extend the one-tile factorization of (3.1) to a matrix of arbitrary size. For this purpose, consider the $n \times n$ matrix A partitioned as

$$A = \left(\ A_0 \;\Big\|\; A_1 \;\Big|\; A_2\ \right) = \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c} \check{A}_0 \\ \hline\hline \check{A}_1 \\ \hline \check{A}_2 \end{array}\right), \tag{4.5}$$

where A_{00}, A_{11}, and A_{22} are square blocks of order $n - t - r$, t, and $r = \rho \cdot t$, respectively. At a certain stage of a right-looking algorithm the first block of columns,

A_0, has already been factorized and the first blocks of rows, $\check{A}_0$, has been updated correspondingly. During the next iteration the following tasks are performed in the algorithm:

1. First, the LU factorization $PA_{11} = LU$ is computed.
2. Consider A_{12} partitioned by columns into ρ tiles of dimension $t \times t$ as $A_{12} = (C_0,\ C_1, \ldots, C_{\rho-1})$. Then, C_j, $0 \le j < \rho$, is updated with respect to the LU factorization of A_{11}; that is, $C_j := L^{-1}PC_j$.
3. Similarly, let $A_{21} = \left(D_0^T,\ D_1^T, \ldots, D_{\rho-1}^T\right)^T$ be a row partitioning of A_{21} into square tiles of order t. Each one of the matrices of the form $\left(U^T,\ D_i^T\right)$, $0 \le i < \rho$, is then factorized using the algorithm in Fig. 3.1 and overwriting these blocks with the results.
4. Finally, consider A_{22} partitioned into $\rho \times \rho$ tiles of dimension $t \times t$ and let $E_{i,j}$, $0 \le i,j < \rho$, denote the (i,j)th tile. At this stage we proceed to update $\left(\frac{C_j}{E_{i,j}}\right)$, $0 \le i,j < \rho$, with respect to the factorization of $\left(\frac{U}{D_i}\right)$ using the algorithm in Fig. 3.3.

It can be shown that this algorithm performs approximately $2/3n^3$ flops. Also, assuming that the matrix is composed of $(n/t) \times (n/t)$ tiles, it requires additional storage (on disk) for approximately $(n/t)^2/2 \cdot \tau = n^2/(2tb)$ lower triangular blocks of dimension $b \times b$ each.

Denoting by iops_{Si} the number of iops performed at Step i (see Table 4.1), the amount of I/O performed by the previous algorithm is given by

$$\sum_{k=0}^{(n/t)-1} \left[C_{S1} + \sum_{j=k+1}^{(n/t)-1} C_{S2} + \sum_{i=k+1}^{(n/t)-1} \left[C_{S3} + \sum_{j=k+1}^{(n/t)-1} C_{S4} \right]\right] \approx \frac{4}{3}n^3/t + \frac{19}{4}n^2 \text{ iops.}$$

The performance of this algorithm is therefore linked to the ratio

$$\frac{4/3n^3/t + 19/4n^2}{2/3n^3} \approx \frac{2}{t}, \tag{4.6}$$

showing that larger values of t benefit the performance.

Let us analyze now the scalability of the *one-tile OOC algorithm*. At most, this algorithm needs to hold in memory a tile of size $t \times t$, a lower triangular block of dimension $b \times b$, and two blocks with bt elements each. Therefore, t and b must satisfy $b^2/2 + 2tb + t^2 \le M$, showing that the problem size n does not affect the performance of the one-tile algorithm.

A left-looking one-tile OOC algorithm is also easily constructed using the building blocks from the factorization of (3.1). This algorithm would perform the same amount of I/O operations as the right-looking one.

5. Remarks on Numerical Stability.

The one-tile OOC algorithm carries out a sequence of row permutations (corresponding to the application of pivots) which are different from those that would be performed in an LU factorization with partial pivoting. Therefore, the numerical stability of this algorithm is also different. In this section we provide some remarks on the stability of the one-tile OOC algorithm.

The numerical (backward) stability of an algorithm that computes the LU factorization of a matrix A depends on the growth factor [18]

$$\rho_{gr} = \frac{\max_{i,j,k} \mid \alpha_{i,j}^{(k)} \mid}{\max_{i,j} \mid \alpha_{i,j} \mid}, \tag{5.1}$$

Operation	Cost (in iops)
1: Factorize B	$2t^2$
2: Update C	$2t^2 + t^2/2$
3: Factorize $\left(\frac{U}{D}\right)$	$3t^2$
4: Update $\left(\frac{C}{E}\right)$	$4t^2$
5: Factorize E	t^2
Total	$\frac{25}{2}t^2$

TABLE 4.1
I/O cost (in iops) of the different steps to compute the LU factorization of the matrix in (3.1).

where $\alpha_{i,j}$ stands for the (i, j) entry of A, and $\alpha_{i,j}^{(k)}$ denotes the corresponding entry of $A^{(k)}$, the transformed matrix after the kth stage of the LU factorization (that is, after having applied the first k Gauss transforms to it).

The growth of (5.1) during the LU factorization is basically determined by the problem size and the pivoting strategy employed in the algorithm. Thus, for example, the growth factors of complete, partial, and pairwise pivoting have been demonstrated to be bounded as $\rho_c \leq n^{1/2}(2 \cdot 3^{1/2} \cdots n^{1/n-1})$, $\rho_p \leq 2^{n-1}$, and $\rho_w \leq 4^{n-1}$, respectively [9]. (The bound for ρ_w has not been demonstrated to be sharp, tough.) This shows that the LU factorization combined with either partial or pairwise pivoting cannot be considered as a numerically backward stable algorithm. Nevertheless, matrices yielding exponentially large element growth are extremely rare, and through years of practice we have come to trust partial pivoting: Extensive experimentations have reported the factors to be invariably small, and statistical models experimentally showed that, on average, $\rho_c \approx n^{1/2}$, while $\rho_p \approx n^{2/3}$, and $\rho_w \approx n$ (see the references in [18]) inferring that, in practice, partial/pairwise pivoting are both numerically stable and pairwise pivoting can be expected to numerically behave only slightly worse than partial pivoting.

The one-tile OOC algorithm for the LU factorization described earlier employs partial pivoting within the diagonal tiles of the matrix (Step 1) and also when factorizing matrices of the form $\left(\frac{U}{D}\right)$ (Step 2). However, in doing so, we are applying a blocked variant of pairwise pivoting between the diagonal tile and each one of the corresponding subdiagonal tiles. Thus, we can expect an element growth for the one-tile OOC LU algorithm that is between those of partial and pairwise pivoting. In particular, if the tile size equals the problem size ($t = n$) our algorithm strictly employs partial pivoting, while if $t = 1$ the algorithm employs pure pairwise pivoting. Next we elaborate an experiment that provides evidence in support of this observation.

In Fig. 5.1 we report the element growths observed during the computation of the LU factorization of matrices of dimensions $n = 500$, 1,000, and 1,500, using partial ($t = n$), incremental ($1 < t < n$), and pairwise pivoting ($t = 1$). The entries of the matrices are generated randomly, chosen from a uniform distribution in the interval $(0.0, 1.0)$. The experiment was carried out on an Intel Xeon processor using MATLAB® 7.0.0 (IEEE double-precision arithmetic). The results report the average element growth for 20 different matrices for each matrix dimension. The figure shows that the growth factor of incremental pivoting is smaller than that of pairwise pivoting and approximates that of partial pivoting as the tile size is increased.

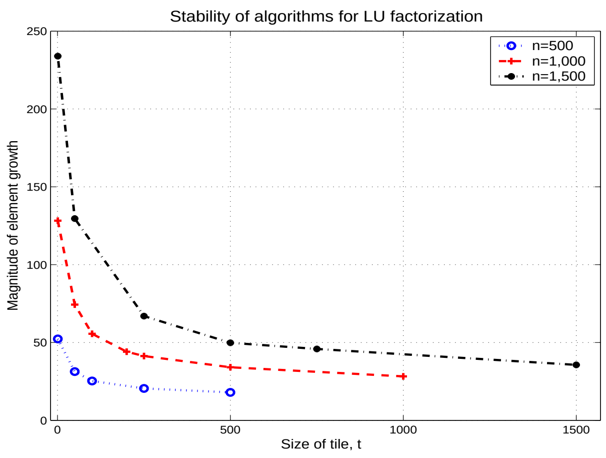

FIG. 5.1. *Element growth in the LU factorization using different pivoting techniques.*

For those who are not sufficiently satisfied with the element growth of incremental pivoting, we propose to perform a few refinement iterations of the solution to $Ax = b$ as this guarantees stability at a low computational cost [18]. We can combine this strategy with an estimation of the backward error $\|PA - LU\|_1$ *a posteriori*, at a cost of $O(n^2)$ flops, to determine whether iterative refinement is actually needed.

6. Performance. While in theory a one-tile approach is ideal, in practice it is beneficial to implement the approach so that up to two full tiles are managed in memory. The reason for this is that during the factorization of $\left(\frac{U}{D}\right)$ and the update of $\left(\frac{C}{E}\right)$ (Steps 2 and 4) blocks of rows of U and C, respectively, must be read from disk. Since matrices are typically stored in column-major order, this incurs too much overhead and it is thus beneficial to bring U and C into memory in their entirety. It is this two-tile approach that we have implemented. The implementation utilizes the FLAME APIs [3], which allow the implementation to closely mirror the algorithms as presented in this paper.

Performance experiments were performed on a Intel Itanium2® (900 MHz) processor based workstation with 8 Gbytes of memory and capable of attaining 3.6 GFLOPS (10^9 flops per second). For reference, the algorithm for the (in-core) LU factorization in LAPACK delivered 3.1 GFLOPS for a square matrix of order 5,200 on this platform. No explicit overlapping is done in our algorithms.

In Fig. 6.1 we show the performance of a sequential implementation using square tiles of order t. An operation count of $2/3n^3$ for the LU factorization is used to compute the GFLOPS ratio. In other words, the extra computation performed by the algorithm is not counted as useful operations, and therefore decreases the effective rate of computation. The results show a remarkable scalability of the one-tile OOC algorithm which is not affected by the matrix size. Performance rivals that of the in-core LU factorization.

The graph is interpreted as follows: Square matrices of size $n \times n$ were factored.

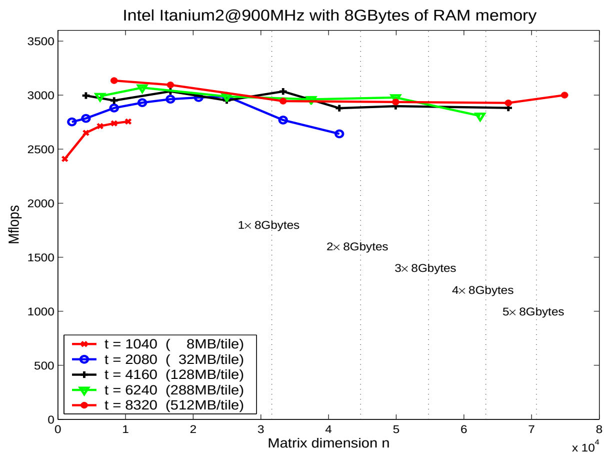

FIG. 6.1. *Performance of the one-tile OOC LU factorization algorithm with incremental pivoting.*

The dashed lines indicate multiples of available memory. We ran a number of experiments using tile sizes that were well below what could be accommodated by the available memory. Even when less that 10% of the available memory was used, excellent performance was attained. This hints at the fact that excellent performance can be attained even on systems with relatively little memory. We warn however that copies of tiles likely remained in memory even after being written back to disk, which may have affected (positively) the reported performance for small problems.

7. Conclusions. We have demonstrated that a modification of the standard in-core right-looking LU factorization algorithms, together with a unique tile-based approach, results in a powerful new method for solving large, dense linear systems via the LU factorization. In combination with other research of ours related to the Cholesky and QR factorizations, this completes a suite of truly scalable tile-based algorithms for OOC solution of dense linear systems and linear least-squares problems [15]. Although the pivoting strategy had to be modified, the proposed incremental pivoting strategy appears to retain much of the benefits of partial pivoting.

In order to create a production code using these techniques it would be highly advisable to add to the implementation iterative refinement, monitor element growth, and provide a condition number estimator since, in practice, as matrices become large, even slow element growth becomes a concern. The rule-of-thumb for LU factorization with partial pivoting is that $\log_{10}(n\kappa(A))$ digits of accuracy are lost, where $\kappa(A)$ is the condition number of the given matrix. We expect incremental pivoting to behave only slightly worse. Provided that at least some accuracy is retained in the solution, iterative refinement can be used to improve it.

The prototype implementation allows very large problems to be solved even on a single CPU, although much patience must be exercised. The largest problem we ran (74,800 $\times$ 74,800) required almost 26 hours to complete. Thus, a natural next step is to create SMP and distributed-memory parallel implementations of these codes.

Acknowledgments. This research was partially sponsored by NSF grants ACI-0203685, ACI-0305163 and CCF-0342369, and an equipment donation from Hewlett-Packard. Primary support for this work came from the Visiting Researcher program

of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin.

For further information on the one-tile OOC approach for the LU factorization, visit `http://www.cs.utexas.edu/users/flame`.

REFERENCES

[1] 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.

[2] G. A. Baker, *Implementation of Parallel Processing to Selected Problems in Satellite Geodesy*, PhD thesis, The University of Texas at Austin, 1998.

[3] P. Bientinesi, E. S. Quintana-Ortí, and R. A. van de Geijn, *Representing linear algebra algorithms in code: The FLAME APIs*, ACM Trans. Math. Soft., 31 (2005), pp. 27–59.

[4] J.-P. Brunet, P. Pederson, and S. L. Johnsson, *Load-balanced LU and QR factor and solve routines for scalable processors with scalable I/O*, in Proceedings of the 17th IMACS World Congress, Atlanta, Georgia, July 1994.

[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, IEEE Comput. Soc. Press, 1992, pp. 120–127.

[6] T. Cwik, R. van de Geijn, and J. Patterson, *The application of parallel computation to integral equation models of electromagnetic scattering*, Journal of the Optical Society of America A, 11 (1994), pp. 1538–1545.

[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] L. Demkowicz, A. Karafiat, and J. Oden, *Solution of elastic scattering problems in linear acoustics using* h-p *boundary element method*, Comp. Meths. Appl. Mech. Engrg, 101 (1992), pp. 251–282.

[9] J. Demmel, *Trading off parallelism and numerical stability*, LAPACK Working Note 52 CS-92-179, University of Tennessee, 1992.

[10] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, 1979.

[11] P. Geng, J. T. Oden, and R. van de Geijn, *Massively parallel computation for acoustical scattering problems using boundary element methods*, Journal of Sound and Vibration, 191 (1996), pp. 145–165.

[12] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, 3nd ed., 1996.

[13] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, *Flame: Formal linear algebra methods environment*, ACM Trans. Math. Soft., 27 (2001), pp. 422–455.

[14] J. A. Gunnels, G. M. Henry, and R. A. van de Geijn, *A family of high-performance matrix multiplication algorithms*, in Computational Science - ICCS 2001, Part I, V. N. Alexandrov, J. J. Dongarra, B. A. Juliano, R. S. Renner, and C. K. Tan, eds., Lecture Notes in Computer Science 2073, Springer-Verlag, 2001, pp. 51–60.

[15] B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the qr factorization*, ACM Trans. Math. Soft., (2005).

[16] B. C. Gunter, W. C. Reiley, and R. 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.

[17] F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling, *Superscalar GEMM-based level 3 BLAS – the on-going evolution of a portable and high-performance library*, in Applied Parallel Computing, Large Scale Scientific and Industrial Problems, B. K. et al., ed., Lecture Notes in Computer Science 1541, Springer-Verlag, 1998, pp. 207–215.

[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002.

[19] T. Joffrain, E. S. Quintana-Ortí, and R. A. van de Geijn, *Rapid development of high-performance out-of-core solvers for electromagnetics*, in PARA04. Submitted.

[20] B. Kågström, P. Ling, and C. V. Loan, *Gemm-based level 3 blas: High-performance model, implementations and performance evaluation benchmark*, LAPACK Working Note #107 CS-95-315, Univ. of Tennessee, Nov. 1995.

[21] ———, *GEMM-based level 3 BLAS: High performance model implementations and performance*

evaluation benchmark, ACM Trans. Math. Soft., 24 (1998), pp. 268–302.

[22] K. Klimkowski and R. van de Geijn, *Anatomy of an out-of-core dense linear solver*, in Proceedings of the International Conference on Parallel Processing 1995, vol. III - Algorithms and Applications, 1995, pp. 29–33.

[23] D. S. Scott, *Out of core dense solvers on Intel parallel supercomputers*, in Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, 1992, pp. 484–487.

[24] ———, *Parallel I/O and solving out-of-core systems of linear equations*, in Proceedings of the 1993 DAGS/PC Symposium, Hanover, NH, June 1993, Dartmouth Institute for Advanced Graduate Studies, pp. 123–130.

[25] G. W. Stewart, *Matrix Algorithms Volume 1: Basic Decompositions*, SIAM, 1998.

[26] S. Toledo, *A survey of out-of-core algorithms in numerical linear algebra*, in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1999.

[27] S. Toledo and F. 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.

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

[29] E. Yip, *Fortran subroutines for Out-of-Core solutions of linear systems*, Tech. Report CR-158142, NASA, 1979.