UPDATING AN LU FACTORIZATION AND ITS APPLICATION TO SCALABLE OUT-OF-CORE COMPUTATION

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

Abstract. We show how to compute an LU factorization of a matrix when the factors of a leading principle submatrix are already known. The approach incorporates pivoting akin to partial pivoting, a strategy we call *incremental pivoting*. A practical, scalable out-of-core implementation of the LU factorization that maintains reasonable numerical stability is built upon the result. Here we use the term scalable to mean that performance is maintained as the matrix size grows well beyond available (in-core) memory. By practical we mean that the implementation achieves high performance. 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. Experimental results demonstrate high performance on an Intel Itanium2 based server.

Key words. LU factorization, out-of-core algorithms, linear systems, pivoting, updating.

AMS subject classifications. 65F05, 65Y10.

1. Introduction. In this paper we consider two related special instances of the LU factorization of a nonsymmetric matrix, A. The first is the instance where matrix A is partitioned as

$$A \rightarrow \left(\begin{array}{c|c} B & C \\ \hline D & E \end{array}\right) \tag{1.1}$$

and a factorization of B is to be reused as the other parts of the matrix change. This is known as the updating of an LU factorization. The second is the instance where matrix A is so large that it does not fit in memory. This is known as the out-of-core (OOC) factorization of a matrix that is stored on disk.

A typical application. Applications arising in Boundary Element Methods (BEM) often lead to very large dense linear systems [7, 11]. The idea 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. By contrast, Finite Element Methods (FEM) put degrees of freedom throughout the three dimensional object. While FEM result in large sparse matrices, BEM 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 [7, 9, 11], leading to linear systems of that order.

Updating a factorization. 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 matrix presents the structure given in (1.1). Now, as the shape of the component is

[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.

modified, it is only the matrices C, D, and E that change together with the right-hand side vector of the corresponding linear system. Since the dimension of B is frequently much larger than those of the remaining three matrices, it is desirable to factorize B only once and to update the factorization as C, D, and E change. 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.

Out-of-core factorization. Often a BEM application leads to a problem that is sufficiently large that the matrix A does not fit into the memory of a single processor. For moderately sized problems the solution is to use a distributed memory parallel computer, calling library routines like those provided by ScaLAPACK [6]. Alternatively the matrix can be stored on disk and an OOC factorization routine be employed. Recent papers on the OOC solution of dense linear systems target parallel computers on which very large linear systems can be solved. ScaLAPACK provides prototype OOC implementations of solvers based on the Cholesky, LU, and QR factorizations [8].

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 provides an input-output (I/O) layer that manages the loading and storing of submatrices from/to disk. 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 on 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 provided by ScaLAPACK.

We developed the Parallel Linear Algebra Package (PLAPACK) in the mid-1990s [29] as an alternative to ScaLAPACK. An OOC extension to PLAPACK, POOCLAPACK, was introduced shortly later. POOCLAPACK has been used for the scalable parallel OOC implementation of tile-based Cholesky and QR factorizations [21, 14, 13].

The parallel OOC library efforts described above 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 [1, 20, 5, 22, 23]. For a survey see [26].

Pivoting and scalability. The fundamental problem with the OOC LU factorization is that in order to provide stability, partial pivoting must be incorporated. (Note however that LU factorization with partial pivoting is not a numerically stable algorithm [16]; it is only experience that taught us to trust it in practice.) 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. Finally, rows must be swapped, which is prohibitively expensive unless the rows in question are already in memory: for current architectures, a disk seek is six to eight *orders of magnitude* more expensive than a floating point operation. 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. 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.

A tiled approach to computing the LU factorization. It is widely recognized that working with so-called (square) tiles is preferable [26, 14, 13]. As the overall matrix size increases, the size of the tile brought into memory can be kept constant, which keeps the ratio between useful computation and I/O overhead constant. The problem with this approach is that it stands in the way of partial pivoting, since all the tiles containing part of a column have to be brought into main memory. In addition, rows that are being swapped need to be read from disk, requiring a large number of disk seeks if matrices are stored by column. 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.

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 for a scalable OOC LU factorization with pivoting. The implementation of this algorithm delivers both scalability and high-performance.

It has been brought to our attention that an unblocked OOC algorithm similar to our algorithm was reported in [31]. The key innovation of our approach lies with the insight that a blocked (high-performance) version of the algorithm in [31] requires, for the in-core factorization, a modification to the way the LAPACK blocked LU factorization is implemented. This modification yields a blocked version of the LU factorization that was part of LINPACK [10]. The OOC solver based on this insight incurs extra computation with a cost that is a lower order term.

Notation. We start indexing elements of vectors and matrices at 0. Capital letters, lower case letter, and lower case Greek letters will be used to denote matrices, vectors, and scalars, respectively. The identity matrix of order n is denoted by I_n.

Overview. This paper expands on results reported in an earlier conference paper [17]. It is organized as follows: in Section 2 we review algorithms for computing the LU factorization with partial pivoting. In Section 3, we discuss how to update an LU factorization by considering the factorization of a 2×2 blocked matrix. This study helps us 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. We hereafter assume that the reader is familar with Gauss transforms, their properties, and how they are used to factor a matrix.

2. The LU factorization with partial pivoting.

Given an $n\times n$ matrix A, its LU factorization with partial pivoting is given by $PA = LU$. Here P is a permutation matrix of order n, L is $n \times n$ unit lower triangular, and U is $n \times n$ upper triangular. We will denote the computation of P, L, and U by

$$[A, p] := [\{L\backslash U\}, p] = \mathrm{LU}(A), \tag{2.1}$$

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

Solving the linear system $Ax = b$ now becomes a matter of solving $Ly = Pb$ followed by $Ux = y$. These two stages are referred to as *forward substitution* and *backward substitution*, respectively.

2.1. Unblocked right-looking LU factorization. Two unblocked algorithms for computing the LU factorization with partial pivoting are given in Fig. 2.1. There, $n(\cdot)$ stands for the number of columns of a matrix; the thick lines in the matrices/vectors denote how far computation has progressed; $\textsc{Pivot}(x)$ determines the element in x with largest magnitude, swaps that element with the top element, and returns the index of the element that was swapped; and $P(\pi_1)$ is the permutation matrix constructed by interchanging row 0 and row π_1 of the identity matrix. The dimension of a permutation matrix will not be specified since it is always obvious from the context in which it is used. We believe the rest of the notation to be intuitive [4, 2]. Both algorithms correspond to what is usually known as the right-looking variant. Upon completion matrices L and U overwrite A.

The LINPACK variant, $\mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LIN}}$ hereafter, computes the factorization as a sequence of Gauss transforms interleaved with pivot matrices:

$$L_{n-1}\left(\begin{array}{c|c} I_{n-1} & 0 \\ \hline 0 & P(\pi_{n-1}) \end{array}\right)\cdots L_1\left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & P(\pi_1) \end{array}\right) L_0 P(\pi_0) A = U.$$

For the LAPACK variant, $\mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}$, it is recognized that by swapping those rows of matrix L that were already computed and stored to the left of the column that is currently being eliminated, the order of the Gauss transforms and the pivot matrices can be rearranged so that $P(p)A = LU$. Here $P(p)$, with $p = \left(\begin{array}{c|c|c} \pi_0 & \cdots & \pi_{n-1} \end{array}\right)^T$, denotes the $n \times n$ permutation

$$\left(\begin{array}{c|c} I_{n-1} & 0 \\ \hline 0 & P(\pi_{n-1}) \end{array}\right)\cdots\left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & P(\pi_1) \end{array}\right) P(\pi_0).$$

Both algorithms will execute to completion even if an exact zero is encountered on the diagonal of U. This is important since it is possible that matrix B in (1.1) is singular even if A is not.

The difference between the two algorithms becomes most obvious when forward substitution is performed. For the LINPACK variant forward substitution requires the application of permutations and Gauss transforms to be interleaved. For the LAPACK algorithm, the permutations can be applied first, after which a clean lower triangular solve yields the desired (intermediate) result: $Ly = P(p)b$. Depending on whether the LINPACK or the LAPACK variant was used for the LU factorization, we denote the forward substitution stage respectively by $y := \mathrm{FS}^{\mathrm{LIN}}(A, p, b)$ or $y := \mathrm{FS}^{\mathrm{LAP}}(A, p, b)$, where A and p are assumed to be the outputs of the corresponding factorization.

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 [18, 15, 19, 12]. In Fig. 2.2 we show LINPACK(-like) and LAPACK blocked algorithms, $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LIN}}$ and $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LAP}}$ respectively, both built upon an LAPACK unblocked algorithm. The former algorithm really combines the LAPACK style of pivoting, within the factorization of a panel of width b, with the LINPACK style of pivoting. The two 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 $A_{22} := A_{22} - L_{21}U_{12}$, which is known to achieve high performance.

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 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Repartition

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

where α_{11} and π_1 are scalars

LINPACK variant:

$$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \textsc{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

if $\alpha_{11} \neq 0$ **then**

$$\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right)$$

$a_{21} := a_{21}/\alpha_{11}$

$A_{22} := A_{22} - a_{21}a_{12}^T$

endif

LAPACK variant:

$$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \pi_1\right] := \textsc{Pivot}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

if $\alpha_{11} \neq 0$ **then**

$$\left(\begin{array}{c|c|c} a_{10}^T & & a_{12}^T \\ \hline A_{20} & & A_{22} \end{array}\right) := P(\pi_1)\left(\begin{array}{c|c|c} a_{10}^T & & a_{12}^T \\ \hline A_{20} & & A_{22} \end{array}\right)$$

$a_{21} := a_{21}/\alpha_{11}$

$A_{22} := A_{22} - a_{21}a_{12}^T$

endif

Continue with

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

endwhile

Fig. 2.1. *LINPACK and LAPACK unblocked algorithms for the LU factorization.*

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 0×0 and p_T has 0 elements

while $n(A_{TL}) < n(A)$ **do**

Determine block size b

Repartition

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

where A_{11} is $b \times b$ and p_1 has b elements

LINPACK variant:

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right),\ p_1\right] := \left[\left(\begin{array}{c} \{L\backslash U\}_{11} \\ \hline L_{21} \end{array}\right),\ p_1\right] = \mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right)$$

$A_{12} := U_{12} = L_{11}^{-1}A_{12}$

$A_{22} := A_{22} - L_{21}U_{12}$

LAPACK variant:

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right),\ p_1\right] := \left[\left(\begin{array}{c} \{L\backslash U\}_{11} \\ \hline L_{21} \end{array}\right),\ p_1\right] = \mathrm{LU}_{\mathrm{UNB}}^{\mathrm{LAP}}\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$$

$$\left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right) := P(p_1)\left(\begin{array}{c|c|c} A_{10} & & A_{12} \\ \hline A_{20} & & A_{22} \end{array}\right)$$

$A_{12} := U_{12} = L_{11}^{-1}A_{12}$

$A_{22} := A_{22} - L_{21}U_{12}$

Continue with

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

endwhile

Fig. 2.2. *LINPACK and LAPACK blocked algorithms for the LU factorization built upon an LAPACK unblocked factorization.*

3. Updating an LU factorization. In this section we discuss how to compute the LU factorization of the matrix in (1.1) in such a way that the LU factorization with partial pivoting of B can be reused if D, C, and E change. For simplicity, we consider all four submatrices in (1.1) to be of size $t \times t$, with t an exact multiple of the block size b. For reference, factoring the matrix in (1.1) 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 future computational cost estimates) we neglect insignificant terms of lower-order complexity, including the cost of pivoting the rows.

3.1. Basic procedure. We propose employing the following procedure, consisting of 5 steps, which computes an *LU factorization with incremental pivoting* of the matrix in (1.1):

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

$$[B, p] := [\{L \backslash U\}, p] = \mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LAP}}(B).$$

(This step is skipped if B was already factored. If the factors are to be used for future updates to C, D, and E, then U needs to be saved since it is overwritten by subsequent steps.)

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

$$C := \mathrm{FS}^{\mathrm{LAP}}(B, p, C).$$

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

$$\left[\left(\frac{\{\bar{L} \backslash \bar{U}\}}{D}\right), \bar{p}\right] := \mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LIN}}\left(\frac{U}{D}\right).$$

Here $\bar{U}$ overwrites the upper triangular part of B (where U was stored before this operation). The lower triangular matrix $\bar{L}$ that results needs to be stored separately, since both L, computed in Step 1, and $\bar{L}$ are needed during the forward substitution stage when solving a linear system. More on this in Section 3.2.

Step 4: Update $\left(\frac{C}{E}\right)$ consistent with the factorization of $\left(\frac{U}{D}\right)$:

$$\left(\frac{C}{E}\right) := \mathrm{FS}^{\mathrm{LIN}}\left(\left(\frac{\bar{L}}{D}\right), \bar{p}, \left(\frac{C}{E}\right)\right).$$

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

$$[E, \hat{p}] := \mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LAP}}(E).$$

Let us analyze the cost of the above procedure. For now, the factorization in Step 3 does not take advantage of any zeroes below the diagonal of U. Correspondingly, the forward substitution in Step 4 does not take advantage of a special structure in $\bar{L}$, that will become obvious in Section 3.2, either. This approach results in the costs stated in the column as "Basic procedure" in Table 3.1, and requires $\frac{5}{3}t^3$ extra flops

Operation	Approximate cost (in flops)		
	Basic procedure	SA LINPACK procedure	SA LAPACK procedure
1: Factor 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: Factor $\left(\frac{U}{D}\right)$	$\frac{5}{3}t^3$	$t^3 + \frac{3}{2}t^2 b$	$t^3 + \frac{3}{2}t^2 b$
4: Update $\left(\frac{C}{E}\right)$	$3t^3$	$2t^3 + t^2 b$	$3t^3$
5: Factor E	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$	$\frac{2}{3}t^3$
Total	$7t^3$	$\frac{16}{3}t^3 + \frac{5}{2}t^2 b$	$\frac{19}{3}t^3 + \frac{3}{2}t^2 b$

TABLE 3.1

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

compared with the LU factorization with partial pivoting of (1.1). 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. The procedure also requires additional storage for the $t \times t$ lower triangular matrix $\bar{L}$ computed in Step 3.

We describe how to reduce both the computational and storage requirements by exploiting the upper triangular structure of U during the LU factorization of $\left(\frac{U}{D}\right)$ and the corresponding update of $\left(\frac{C}{E}\right)$.

3.2. Exploiting the structure in Step 3. A blocked algorithm that exploits the upper triangular structure of U is given in Fig. 3.1 and illustrated in Fig. 3.2. We name this algorithm $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}$ to reflect that it computes a "structure-aware" (SA) LU factorization. At each iteration of the algorithm, the panel of b columns consisting of $\left(\frac{U_{11}}{D_1}\right)$ is factored using the LAPACK blocked algorithm $\mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LAP}}$. As part of the factorization, U_{11} is overwritten by $\{\bar{L}\backslash\bar{U}\}_{11}$. However, in order to preserve the strictly lower triangular part of U_{11} (which stores part of the matrix L that was computed in Step 1), the matrix $\bar{L}_{11}$ is actually stored in the $b \times b$ submatrix $\bar{L}_1$ of the $t \times b$ array $\bar{L}$. As in the LINPACK blocked algorithm in Fig. 2.2, the LAPACK and LINPACK styles of pivoting are combined: the current panel of columns are pivoted using the LAPACK approach but the pivots from this factorization are only applied to $\left(\frac{U_{12}}{D_2}\right)$.

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

Remark. *An SA-LAPACK blocked algorithm for Step 3 only differs from that in Fig. 3.1 in that, 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 LU factorization in Step 1, pivoting*

Algorithm: $\left[\begin{pmatrix} U \\ \hline D \end{pmatrix}, \bar{L}, \bar{p}\right] := \mathrm{LU}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}\begin{pmatrix} U \\ \hline D \end{pmatrix}$

Partition $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right)$, $\bar{L} \rightarrow \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right)$, $D \rightarrow \left(\begin{array}{c|c} D_L & D_R \end{array}\right)$, $\bar{p} \rightarrow \left(\begin{array}{c} \bar{p}_T \\ \hline \bar{p}_B \end{array}\right)$

where U_{TL} is 0×0, $\bar{L}_T$ has 0 rows, D_L has 0 columns, and $\bar{p}_T$ has 0 elements

while $n(U_{TL}) < n(U)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\begin{array}{c} \bar{p}_T \\ \hline \bar{p}_B \end{array}\right) \rightarrow \left(\begin{array}{c} \bar{p}_0 \\ \hline \bar{p}_1 \\ \hline \bar{p}_2 \end{array}\right)$$

where U_{11} is $b \times b$, $\bar{L}_1$ has b rows, D_1 has b columns, and $\bar{p}_1$ has b elements

$$\left[\begin{pmatrix} \{\bar{L}_1 \backslash U_{11}\} \\ \hline D_1 \end{pmatrix}, \bar{p}_1\right] := \mathrm{LU}_{\mathrm{BLK}}^{\mathrm{LAP}}\begin{pmatrix} U_{11} \\ \hline D_1 \end{pmatrix}$$

$$\begin{pmatrix} U_{12} \\ \hline D_2 \end{pmatrix} := P(\bar{p}_1)\begin{pmatrix} U_{12} \\ \hline D_2 \end{pmatrix}$$

$$U_{12} := \bar{L}_1{}^{-1} U_{12}$$

$$D_2 := D_2 - D_1 U_{12}$$

Continue with

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline 0 & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline 0 & U_{11} & U_{12} \\ \hline 0 & 0 & U_{22} \end{array}\right), \left(\begin{array}{c} \bar{L}_T \\ \hline \bar{L}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} D_L & D_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} D_0 & D_1 & D_2 \end{array}\right), \left(\begin{array}{c} \bar{p}_T \\ \hline \bar{p}_B \end{array}\right) \leftarrow \left(\begin{array}{c} \bar{p}_0 \\ \hline \bar{p}_1 \\ \hline \bar{p}_2 \end{array}\right)$$

endwhile

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

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 (see Fig. 3.2).*

3.3. Revisiting the update in Step 4. The same optimizations made in Step 3 must now be carried over to the update of $\begin{pmatrix} C \\ \hline E \end{pmatrix}$. The algorithm for this is given in Fig. 3.3. Computation corresponding to zeroes is avoided so that the cost of performing the update is $2t^3 + t^2 b$ flops, or $2t^3$ flops if $b \ll t$, which saves t^3 flops with respect to the basic procedure.

Remark. *Applying the SA-LAPACK blocked algorithm in Step 3 destroys the structure of the lower triangular matrix, which cannot be recovered during the forward substitution stage in Step 4.*

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

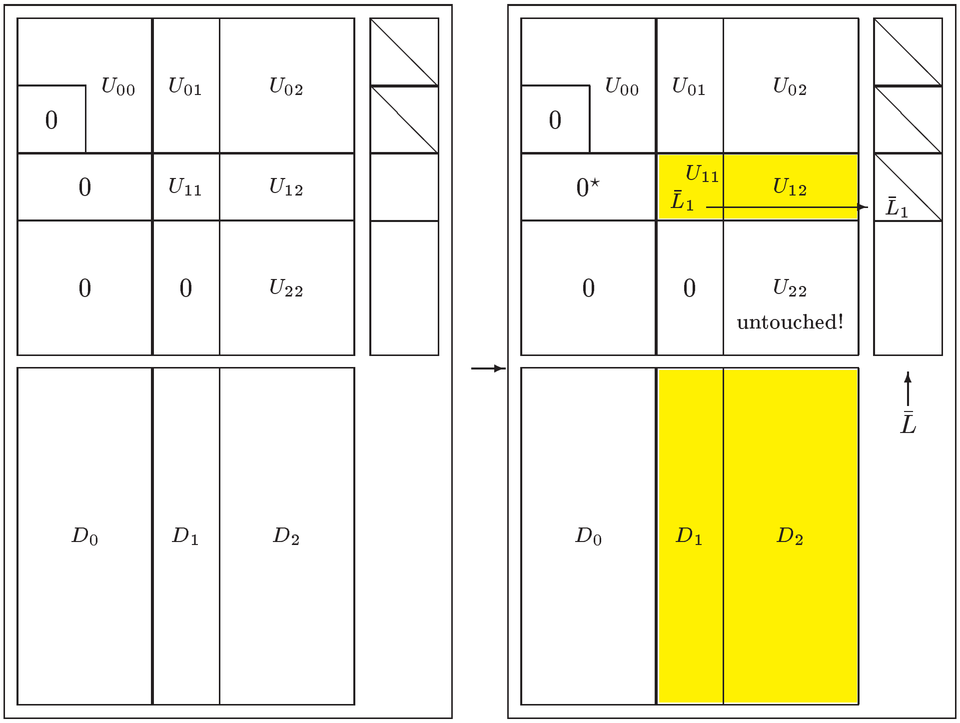

FIG. 3.2. *Illustration of an iteration of the SA-LINPACK blocked algorithm used in Step 3 and how it preserves most of the zeroes in U. The zeroes below the diagonal are preserved, except within the $b \times b$ diagonal blocks, where pivoting will fill below the diagonal. The shaded areas are the ones updated as part of the current iteration. The fact that U_{22} is not updated demonstrates how computation can be reduced. If the SA-LAPACK blocked algorithm is used, then nonzeroes will appear during this iteration in the block marked as $0^\star$, due to pivoting.*

lower triangular matrix.

4. Out-of-Core LU factorization. In this section it is shown how the insights from the previous section can be used to implement a scalable OOC LU factorization with incremental pivoting for matrices stored on disk. Next, we analyze the non-scalability of slab approaches for implementing the LU factorization with traditional partial pivoting. During the section we will consider A of dimension $n \times n$.

4.1. A tiled OOC algorithm. Assume for simplicity that $n = Nt$, where N is an integer, and consider the partitioning by tiles

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & \cdots & A_{0,N-1} \\ \hline \vdots & \ddots & \vdots \\ \hline A_{N-1,0} & \cdots & A_{N-1,N-1} \end{array}\right),$$

with all A_{ij} of size $t \times t$. Then the algorithm in Fig. 4.1 is a generalization of the algorithm described in Section 3 that computes the LU factorization of A with incremental pivoting. The algorithm is annotated with the cost of each operation (highest order term only) in terms of flops and iops (I/O operations), that is, loads and stores of data from and to disc.

Algorithm: $\left[\left(\frac{C}{E}\right)\right] := \mathrm{FS}_{\mathrm{BLK}}^{\mathrm{SA-LIN}}\left(\left(\frac{\bar{L}}{D}\right), \bar{p}, \left(\frac{C}{E}\right)\right)$

Partition $C \rightarrow \left(\frac{C_T}{C_B}\right)$, $\bar{L} \rightarrow \left(\frac{\bar{L}_T}{\bar{L}_B}\right)$, $D \rightarrow \left(D_L \,\|\, D_R \right)$, and $\bar{p} \rightarrow \left(\frac{\bar{p}_T}{\bar{p}_B}\right)$
  where C_T and $\bar{L}_T$ have 0 rows, D_L has 0 columns, and $\bar{p}_T$ has 0 elements

while $n(D_L) < n(D)$ **do**
  Determine block size b
  Repartition
    $\left(\frac{C_T}{C_B}\right) \rightarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$, $\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \rightarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right)$,
    $\left(D_L \,\|\, D_R \right) \rightarrow \left(D_0 \,\|\, D_1 \,|\, D_2 \right)$, and $\left(\frac{\bar{p}_T}{\bar{p}_B}\right) \rightarrow \left(\begin{array}{c} \bar{p}_0 \\ \hline \bar{p}_1 \\ \hline \bar{p}_2 \end{array}\right)$
    where C_1 and $\bar{L}_1$ have b rows, D_1 has b columns,
    and $\bar{p}_1$ has b elements

  $\left(\frac{C_1}{E}\right) := P(\bar{p}_1)\left(\frac{C_1}{E}\right)$
  $C_1 := \bar{L}_1^{-1} C_1$
  $E := E - D_1 C_1$

  Continue with
    $\left(\frac{C_T}{C_B}\right) \leftarrow \left(\begin{array}{c} C_0 \\ \hline C_1 \\ \hline C_2 \end{array}\right)$, $\left(\frac{\bar{L}_T}{\bar{L}_B}\right) \leftarrow \left(\begin{array}{c} \bar{L}_0 \\ \hline \bar{L}_1 \\ \hline \bar{L}_2 \end{array}\right)$,
    $\left(D_L \,\|\, D_R \right) \leftarrow \left(D_0 \,|\, D_1 \,\|\, D_2 \right)$, and $\left(\frac{\bar{p}_T}{\bar{p}_B}\right) \leftarrow \left(\begin{array}{c} \bar{p}_0 \\ \hline \bar{p}_1 \\ \hline \bar{p}_2 \end{array}\right)$

endwhile

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

The total number of flops of the OOC algorithm is approximately given by

$$\sum_{k=0}^{N-1}\left(\frac{2}{3}t^2 + \sum_{j=k+1}^{N-1} t^3 + \sum_{i=k+1}^{N-1}\left(t^3 + \sum_{j=k+1}^{N-1} 2t^3\right)\right) \approx \frac{2}{3}\left(\frac{n}{t}\right)^3 t^3 = \frac{2}{3}n^3,$$

while the total number of iops is roughly given by

$$\sum_{k=0}^{N-1}\left(2t^2 + \sum_{j=k+1}^{N-1} 2t^2 + \sum_{i=k+1}^{N-1}\left(2t^2 + bt/2 + \sum_{j=k+1}^{N-1} 4t^2\right)\right) \approx \frac{4}{3}\left(\frac{n}{t}\right)^3 t^2 = \frac{4n^3}{3t}.$$

Thus, the ratio of iops to flops is approximately

$$\frac{\frac{4n^3}{3t}}{\frac{2}{3}n^3} \approx \frac{2}{t}$$

which demonstrates that the performance improves with the size of the tile, t.

Let us analyze now the scalability of the algorithm. At most, (during step OOC-4) three tiles of size $t \times t$ (for A_{ik}, A_{kj} and A_{ij}) and t/b lower triangular blocks of dimension $b \times b$ (for L_{ik}) need to be present in memory simultaneously. As the total

Step	Algorithm	Cost
	for $k = 0 : N-1$	
	load A_{kk}	t^2 iops
OOC-1	$[A_{kk}, p_{kk}] := \mathrm{LU}^{\mathrm{LAP}}_{\mathrm{BLK}}(A_{kk})$	$\frac{2}{3}t^3$ flops
	for $j = k+1 : N-1$	
	load A_{kj}	t^2 iops
OOC-2	$A_{kj} := \mathrm{FS}^{\mathrm{LAP}}(A_{kk}, p_{kk}, A_{kj})$	t^3 flops
	store A_{kj}	t^2 iops
	endfor	
	store A_{kk}, p_{kk}	t^2 iops
	for $i = k+1 : N-1$	
	load A_{kk}, A_{ik}	$2t^2$ iops
OOC-3	$\left[\left(\frac{A_{kk}}{A_{ik}}\right), L_{ik}, p_{ik}\right] := \mathrm{LU}^{\mathrm{SA-LIN}}_{\mathrm{BLK}}\left(\frac{A_{kk}}{A_{ik}}\right)$	t^3 flops
	store A_{kk}	t^2 iops
	for $j = k+1 : N-1$	
	load A_{kj}, A_{ij}	$2t^2$ iops
OOC-4	$\left[\left(\frac{A_{kj}}{A_{ij}}\right)\right] := \mathrm{FS}^{\mathrm{SA-LIN}}_{\mathrm{BLK}}\left(\left(\frac{L_{ik}}{A_{ik}}\right), p_{ik}, \left(\frac{A_{kj}}{A_{ij}}\right)\right)$	$2t^3$ flops
	store A_{kj}, A_{ij}	$2t^2$ iops
	endfor	
	store L_{ik}, p_{ik}	$bt/2$ iops
	endfor	
	endfor	

FIG. 4.1. *Tiled OOC algorithm for the LU factorization with incremental pivoting (right-looking variant).*

number of matrix elements, M, that can be stored in memory is fixed, t and b must satisfy $bt/2 + 3t^2 \le M$, showing that the problem size n does not affect the scalability of the tiled OOC algorithm.

Notice that that there is some flexibility in the order in which the loops are arranged. We have not completely explored whether rearranging the loops affects the total amount of I/O.

4.2. A one-tile OOC algorithm. We note that the tiled OOC algorithm can be modified so that almost all of available memory is dedicated to a single tile, which allows the tiling parameter, t, to be larger. Clearly, during the factorization of the tile on the diagonal (step OOC-1) only that tile needs to be present in memory. While performing the forward substitution in step OOC-2, the matrix A_{kj} can occupy most of in-core memory while A_{kk} can be brought in a panel of columns of width b at a time. While computing step OOC-3, matrix A_{ik} must be kept in memory but A_{kk} can be read in panels of rows of height b. Finally, during the computation in step OOC-4, matrix A_{ik} must be kept in memory but A_{ij} and A_{kj} can be read b columns and b rows at a time, respectively. A detailed discussion of the same observations as they apply to a one-tile OOC Cholesky factorization can be found in [21].

4.3. 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. A detailed description of a practical left-looking blocked slab solver can be found in [20]. We briefly review it next.

The state of the matrix after k iterations of a left-looking blocked algorithm for the LU factorization with partial pivoting can be described as

$$A = \left(\begin{array}{c|c|c} \overbrace{A_{00}}^{A_0} & \overbrace{A_{01}}^{A_1} & \overbrace{A_{02}}^{A_2} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline \underbrace{A_{20}}_{ks} & \underbrace{A_{21}}_{s} & A_{22} \end{array}\right),$$

where A_{00}, A_{11}, and A_{22} are square blocks of order ks, s, and $n-(k+1)s$, respectively, and the thick line denotes how far computation has proceeded. Therefore, it is assumed that after k iterations A_0 has been completely updated with the final result while A_1 and A_2 are yet to be touched. During the current iteration, slab A_1 is loaded from disk into memory at a cost of ns iops. Next, forward substitution is performed with the columns of A_0 (and the corresponding pivot information). Loading the lower trapezoidal part of A_0 into memory requires $nks-(ks)^2/2$ iops. (In practice, in order to allow the use of BLAS-3 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 at a cost of another ns iops.

Assuming for simplicity that $n = Ns$, where N is an integer, this algorithm performs approximately

$$\textstyle\sum_{k=0}^{N-1}\left[ns + nks - (ks)^2/2 + ns\right] \approx \frac{n^3}{3s} + 2n^2 \quad \text{iops}$$

and $\frac{2}{3}n^3$ flops, and the ratio of disk access overhead to useful computation is given by

$$\frac{\frac{n^3}{3s}+2n^2}{\frac{2}{3}n^3} \approx \frac{1}{2s}.$$

This shows that the performance improves with the width of the slab, s.

At a given iteration in the algorithm, two slabs must be stored in memory: one of width s for the current panel A_1 and a second one of width b for a block of columns from A_0. Thus $(s+b)n \leq M$ and the ratio of overhead to useful computation is approximately

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

The conclusion is that as the problem size n grows, the overhead increases which ultimately leads to inefficiency. The approach is not scalable as n increases.

5. Remarks on Numerical Stability. The tiled OOC algorithm for the LU factorization with incremental pivoting 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 tiled OOC algorithm.

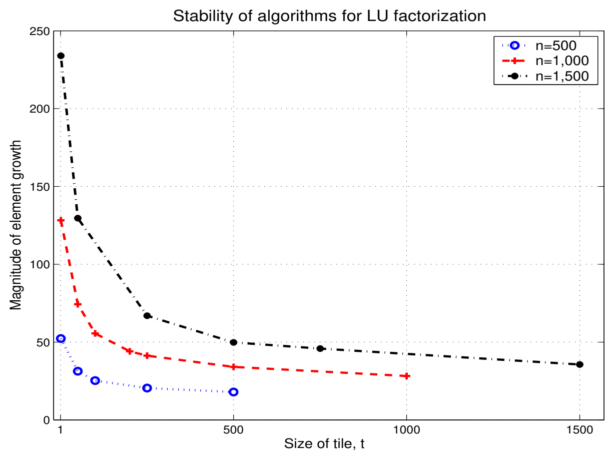

FIG. 5.1. *Element growth in the LU factorization using different pivoting techniques.*

The numerical (backward) stability of an algorithm that computes the LU factorization of a matrix A depends on the growth factor [25]

$$\rho = \frac{||L||||U||}{||A||}, \tag{5.1}$$

which is basically determined by the problem size and the pivoting strategy. For example, the growth factors of complete, partial, and *pairwise* ([30, p. 236]) 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 [24, 25]. Statistical models and extensive experimentations in [28] showed that, on average, $\rho_c \approx n^{1/2}$, $\rho_p \approx n^{2/3}$, and $\rho_w \approx n$, 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 tiled OOC algorithm applies partial pivoting during the factorizations in steps OOC-1 and OOC-3. Furthermore, tiles are annihilated pairwise in what can be considered a blocked (or tiled) version of pairwise pivoting. Thus, we can expect an element growth for the tiled OOC 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 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.

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 [16]. 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 in step OOC-3 blocks of rows of A_{kk} 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 A_{kk} into memory in its 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 3600 MFLOPS (10^6 flops per second). For reference, the algorithm for the (in-core) LU factorization in LAPACK delivered 3100 MFLOPS for a square matrix of order 5,200 on this platform. No explicit overlapping of I/O and computation is done in our algorithms.

In Fig. 6.1 we show the performance of a sequential implementation using tiles of size t. An operation count of $2/3n^3$ for the LU factorization is used to compute the MFLOPS 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. 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 [13]. 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.

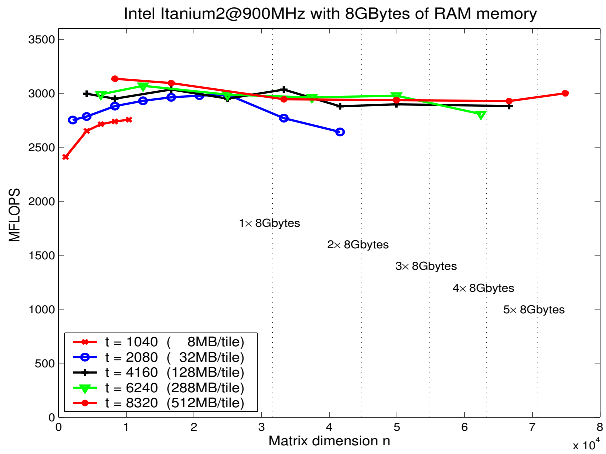

FIG. 6.1. *Performance of the tiled OOC algorithm for the LU factorization with incremental pivoting.*

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 *J. Tinsley Oden Faculty Fellowship Research 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] G. A. Baker, *Implementation of Parallel Processing to Selected Problems in Satellite Geodesy*, PhD thesis, The University of Texas at Austin, 1998.

[2] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn, *The science of deriving dense linear algebra algorithms*, ACM Trans. Math. Soft., 31 (2005), pp. 1–26.

[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] P. Bientinesi and R. van de Geijn, *Representing dense linear algebra algorithms: A farewell to indices*, SIAM Review. submitted.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[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] 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.

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

[14] 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.

[15] 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.

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

[17] T. Joffrain, E. S. Quintana-Ortí, and R. A. van de Geijn, *Rapid Development of High-Performance Out-of-Core Solvers*, LNCS 3732, Springer-Verlag, 2005, pp. 413–422.

[18] 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.

[19] ———, *GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark*, ACM Trans. Math. Soft., 24 (1998), pp. 268–302.

[20] 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.

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

[22] 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.

[23] ———, *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.

[24] D. C. Sorensen, *Analysis of pairwise pivoting in Gaussian elimination*, IEEE Trans. on Computers, c-34 (1985), pp. 274–278.

[25] G. W. Stewart, *Matrix Algorithms. Volume I: Basic Decompositions*, SIAM, Philadelphia, 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] L. N. Trefethen and R. S. Schreiber, *Average-case stability of Gaussian elimination*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 335–360.

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

[30] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, London, 1965.

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