Recursion leads to automatic variable blocking for dense linear-algebra algorithms

by F. G. Gustavson

We describe some modifications of the LAPACK dense linear-algebra algorithms using recursion. Recursion leads to automatic variable blocking. LAPACK's level-2 versions transform into level-3 codes by using recursion. The new recursive codes are written in FORTRAN 77, which does not support recursion as a language feature. Gaussian elimination with partial pivoting and Cholesky factorization are considered. Very clear algorithms emerge with the use of recursion. The recursive codes do exactly the same computation as the LAPACK codes, and a single recursive code replaces both the level-2 and level-3 versions of the corresponding LAPACK codes. We present an analysis of the recursive algorithm in terms of both FLOP count and storage usage. The matrix operands are more "squarish" using recursion. The total area of the submatrices used in the recursive algorithm is less than the total area used by the LAPACK level-3 right-/left-looking algorithms. We quantify the difference; we also quantify how the FLOPS are computed. Also, we show that the algorithms exhibit high performance on RISC-type processors. In fact, except for small matrices, the recursive version outperforms the level-3 LAPACK versions of DGETRF and DPOTRF on an RS/6000™ workstation. For the level-2 versions, the performance gain approaches a factor of 3. We also demonstrate that a change to the LAPACK DLASWP routine can improve the performance of both the recursive version and DGETRF by more than 15 percent.

1. Introduction

Recursion leads to automatic variable blocking for dense linear-algebra algorithms, e.g., the algorithms in ESSL, IMSL, LAPACK, MATLAB, and NAG [1–5]. By *variable* we mean that the block size changes during execution of the algorithm; we are not referring to the blocking of the variables of the algorithm. Blocking for the memory hierarchy is extremely important. Explicit blocking parameters should be combined with recursion if one wants to obtain near-optimal results for dense linear-algebra codes on today's RISC-type processors. However, we do not combine these blocking parameters with recursion in this paper. Our aim is to exhibit the implicit blocking that recursion imparts to certain algorithms and also to demonstrate its simplicity. We do this by closely examining the two LAPACK level-2 codes DGETF2 and

0018-8646/97/$5.00 © 1997 IBM

DPOTF2. Our form of recursion contains many forms of fixed blocking: right-looking, left-looking, JKI, etc. [6, 7]. Additionally, we include variable-size square blocking, a form of which has been shown by Toledo [8] to be superior to right-looking blocking for general matrix factorization. Our recursive codes are written in FORTRAN 77, which does not support recursion. This is accomplished by explicitly handling the recursion in the FORTRAN 77 code. Using recursion on the LAPACK level-2 codes DGETF2 and DPOTF2 automatically turns them into level-3 codes; the new recursive codes call only the level-3 BLAS (basic linear-algebra subprogram) routines DGEMM, DTRSM, and DSYRK. Additionally, the new level-2 codes (named RGETF2 and RPOTF2) outperform the LAPACK level-3 codes for the two example codes. To us, the clarity of the recursive form of the algorithm appears to be superior to that of the nonrecursive form. If the performance trend in the two example codes is nearly universal, one has a strong argument to replace all level-2 and level-3 codes with their level-2 recursive counterparts, codes with only level-3 BLAS calls.[1]

In the 1970s the algorithms of dense linear algebra were implemented in a systematic way by the LINPACK [9] project and were kept machine-independent partly through the introduction of the level-1 BLAS routines. Almost all of the computation was done by calling level-1 BLAS. For each machine, the set of level-1 BLAS would be implemented in a machine-specific manner to obtain high performance.

We briefly review the concepts behind level-2 and level-3 codes. The introduction in the late 1970s and early 1980s of vector machines brought about the development of LAPACK level-2 algorithms for dense linear algebra. A level-2 code is typified by the main level-2 BLAS, which is the multiplication of a matrix by a vector. These codes were meant to give improved performance over the dense linear-algebra codes in LINPACK, which were based on level-1 BLAS. A typical level-1 BLAS is a vector dot product or the adding of a multiple of one vector to another vector. Later on, in the late 1980s and early 1990s, with the introduction of RISC-type microprocessors and other machines with cache-type memories, we saw the development of LAPACK level-3 algorithms for dense linear algebra. A level-3 code is typified by the main level-3 BLAS, which is the multiplication of a matrix by a matrix. (The suffix i in level i refers to the number of nested "do looks" required to do the computation of the BLAS.) Like the level-2 codes, the level-3 codes were meant to improve performance over existing level-2 and level-1 codes on these newer machines.

[1] For RGETF2 there are calls to the level-1 BLAS routines IDAMAX and DSCAL.

For general LU decomposition, one factors an M by N matrix $\mathbf{A}$ using partial pivoting; $LU = P\mathbf{A}$. The Cholesky algorithm factors an N by N positive definite symmetric matrix $\mathbf{A}$; either $U^TU = \mathbf{A}$ or $LL^T = \mathbf{A}$. For both RGETF2 and RPOTF2, our recursion produces a binary tree with $N - 1$ nodes of depth $k + 1$, where $2^{k-1} < N \leq 2^k$. At each level i, 2^{i+1} calls are made to level-3 BLAS.

In the Cholesky algorithm (analogous results hold for Gaussian elimination), at level i each BLAS problem is square of size $n_i = \lfloor N/2^{i+1} \rfloor$ or $n_i + 1$. In going from level i to $i + 1$, the number of BLAS calls doubles and each problem size is halved. Hence, the total number of FLOPS done at each level goes down by a factor of 4. Suppose the MFLOP rate were constant at each level; then the computation time would follow a geometric series with ratio $r = 1/4$. However, the MFLOP rate of a square level-3 BLAS is only "constant" when the problem size becomes larger than a block size NB, which depends on architecture considerations [10]. As the problem size falls below NB and approaches 1, the MFLOP rate drops off drastically. This partly explains why our recursive method performs poorly for small matrices. In these cases our algorithms make most of their calls to level-3 BLAS, where each call has a small-square problem size. We mention that we can avoid this performance problem by "pruning the tree" at a high enough level, i.e., by calling a factor kernel. For large problems the geometric nature of the recursion "takes over," as the performance results demonstrate.

This paper introduces the total area of the BLAS operands as the basis of a new set of measures of the efficiency of a dense linear-algebra code. We denote the measures by *LLTA*, *RLTA*, and *RTA*, which stand respectively for left-looking total area, right-looking total area, and recursive total area. The new measures are used in Sections 2 and 3 to quantify just how much variable blocking improves upon left-/right-looking blocking. To be more specific, let $N = nNB$ so that $\mathbf{A}$ is represented as an n by n matrix of square blocks of size NB. Both the right-/left-looking and the recursive algorithms consist of n block factor steps and $n - 1$ calls to level-3 BLAS. The total FLOP count for all calls to level-3 BLAS at the $n - 1$ stages is the same for all of the algorithms. However, the operands (submatrices of $\mathbf{A}$) of the BLAS calls are always nearly square for the recursive algorithm. Since the FLOP count is maximized for square operands one can expect the total area of the operands for the recursive algorithm to be less than or equal to the total area of the operands for the right-/left-looking algorithms. This turns out to be true. In fact, for Cholesky, as n increases, the ratio $LLTA/RTA$ approaches $1 + N/9$. For LU factorization, as n increases, the ratio $RLTA/RTA$ approaches $4N/(3 \log_2 N)$.

In Section 2, we describe the recursive Cholesky algorithm by detailing and verifying the claims made in the above paragraphs. In Section 3, we describe recursive general factorization. This algorithm is similar to the recursive FORTRAN 90 algorithm of Toledo [8]. General factorization is done on an M by N matrix. Recursion works on the column dimension N. At recursion level i, 2^i calls are made to DTRSM and 2^i calls to DGEMM. As in Cholesky, each call to DTRSM is on a square problem of size n_i or $n_i + 1$. DGEMM has three matrix dimensions—m, n, and k. Each of the 2^i, n, k dimensions is either n_i or $n_i + 1$. However, the m dimensions are variable. Nonetheless, the total computation at each level again follows a variable "geometric progression" whose ratio is $r > 1/2$. In Section 4, we give some performance results comparing the new recursive algorithms to LAPACK algorithms DGETF2, DGETRF, and DPOTF2, DPOTRF on an IBM RS/6000* workstation. These results show that the recursive versions outperform both the level-2 and level-3 versions of LAPACK when the matrices do not fit into level-1 cache. For large problems, the performance gains are between 2.5 and 3.0 over level-2 codes and 1.02 to 1.10 over level-3 codes. The improvement given by our version of DLASWP versus the LAPACK version is more than 15% for large matrices.

Starting with LINPACK and continuing with LAPACK, the algorithms of dense linear algebra were kept machine-independent through the use of the BLAS. As machines became more complex in the design of their memory hierarchies, it became necessary to increase the scope of the BLAS routines from level 1 to levels 2 and 3. The algorithms in LINPACK were redesigned; the result was LAPACK. However, modularity between the BLAS routines and the algorithms was preserved. Nonetheless, there is a basic pattern to the calling of BLAS in many dense linear-algebra algorithms, which is typified by right-looking matrix factorization. The pattern is this: For as long as any columns remain to be factored, factor the next block of k columns followed by a rank k update of all trailing columns. The LAPACK level-3 codes call a level-2 routine to perform the factor step and a level-3 routine to perform the rank k update. Hence, the operands of the level-3 BLAS calls are related. This then suggests that modularity between LAPACK code design and its BLAS calls should be re-examined.

This paper further demonstrates that the BLAS calls in many dense linear-algebra algorithms are related, and it raises a question as to whether that relationship can be exploited. The answer appears to be both yes and no. The yes answer requires that a change be made in the way the original matrix is stored. If this is done, the BLAS routines must be changed to reflect the new storage arrangement. The "new" storage format is not actually new, in the sense that it has been advocated by many

```
Cholesky factor A(1:N, 1:N)

if (N=1) then
   A(1,1) = SQRT(A(1,1))
else
   N1 = N/2
   N2 = N-N1
   J1 = N1+1
   Cholesky factor (A(1:N1, 1:N1))
   solve A(1:N1, 1:N1) X = A(1:N1, J1:N) (DTRSM)
   update A(J1:N, J1:N) = A(J1:N, J1:N) -X^T X (DSYRK)
   Cholesky factor (A(J1:N, J1:N))
   end if
```

Figure 1

High-level description of the recursive Cholesky algorithm.

authors at various times. The format is a blocked format, which is a special case of the block-cyclic format, and that suggests a key observation: The various calls to the BLAS routines of a dense linear-algebra code encounter the same submatrix operands over and over again. To take advantage of this fact, one can rearrange the storage format of the original matrix to blocked format just once, so that each BLAS call receives its submatrices stored in an optimal way. Then, on completion of the dense linear-algebra algorithm, it is necessary only to rearrange the blocked storage format of the matrix back to the original, column-oriented FORTRAN storage format.

Currently, some BLAS implementations do exactly this. The matrix operands are copied to a more suitable data structure and then the BLAS is executed on this copy. However, this copy procedure, although very effective, has to be done for every call. The repeated copy can be avoided if the original data are in the copied form to begin with. So, having the input data to a BLAS in an optimal form actually makes the design and implementation of the BLAS simpler. In essence, the memory-management aspect of the BLAS is no longer present. The burden has been shifted to the algorithm designer to provide an appropriate blocking parameter *NB*. However, for dense linear algebra this is already being done.

The no answer refers to perhaps being unable to add the blocked data typed to the FORTRAN or C language and/or to suitably modify the current BLAS to accept blocked submatrix operands. Also, as mentioned, the LAPACK design can be changed, thereby keeping the original FORTRAN/C input data structures. The latter approach is perhaps more realistic.

2. Recursive Cholesky factorization

In **Figure 1**, we give the algorithm. For simplicity we assume that the N by N matrix **A** is positive definite and

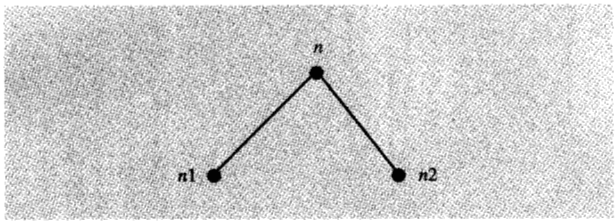

Figure 2

Part of a tree diagram that describes the recursive Cholesky algorithm ($n1 + n2 = n$ and $n1 = n/2$).

```
do J = 1,N,NB
  JB = MIN(NB,N-J+1)
  call DSYRK('U','T',JB,J-1,-ONE,A(1,J),LDA,ONE,A(J,J),LDA)
  call DPOTF2('U',JB,A(J,J),LDA,INFO)
  if( J+JB.LE.N ) then
    call DGEMM('T','N',JB,N-J-JB+1,J-1,-ONE,A(1,J),LDA,
$                A(1,J+JB),LDA,ONE,A(J,J+JB),LDA)
    call DTRSM('L','U','T','N',JB,N-J-JB+1,ONE,A(J,J),LDA,
$                A(J,J+JB),LDA)
  end if
end do
```

Figure 3

Main loop of LAPACK routine DPOTRF (*uplo* = '*U*').

is stored as upper triangular (*uplo* = '*U*'). We use the colon notation to describe submatrices, as in [11].

In the **else** clause there are two recursive calls, one on matrix **A**(1:$N1$, 1:$N1$), the other on matrix **A**($J1$:N, $J1$:N), and a call to the level-3 BLAS routines DTRSM and DSYRK. To handle the recursion explicitly in FORTRAN 77, we store three integers (*ISW*, *J*, *N*) for each recursion level *i*. *ISW* denotes a switch having values 0, 1, 2, denoting whether one should make the first recursive call, the second recursive call, or return from the current recursion level; *J* denotes its diagonal position in the global matrix **A**; and *N* denotes the current size of the submatrix. The space needed for the stack is minuscule. To handle matrices up to size $N = 2^{k-1}$ requires space for $3 \log_2 N$ integers.

• *Analysis of recursive Cholesky factorization*

Suppose we are at recursive level *i* and the current problem size is *n*. According to Figure 1, one executes the **else** clause unless $n = 1$. In **Figure 2**, we depict this situation as a node (at level *i*) in a tree with two branches to level $i + 1$ which denote the two recursive calls. In between these recursive calls there are calls to DTRSM and DSYRK. Thus, at each node that is not a leaf there is one call each to DTRSM and DSYRK. The size of the DTRSM problem is $n1$ by $n2$, and the size of the DSYRK problem is $n2$ by $n1$. If *n* is even, DTRSM and DSYRK perform $n1^3$ multiply–adds (MAs), and $n1(n1 + 2)n/2$ MAs if $n1$ is odd.

The number of MAs needed to Cholesky-factor an *n* by *n* matrix is $n(n^2 - 1)/6$, and the number of MAs needed to Cholesky-factor the $n1$ by $n1$ and $n2$ by $n2$ submatrices is $n(n^2 - 4)/24$ (if *n* is even) and $n(n^2 - 1)/24$ (if *n* is odd).

Let $n_i = \lfloor N/2^i \rfloor$. At level *i* there will be 2^i nodes, each of which has size n_i or $n_i + 1$. Let α_i be the number of nodes of size n_i and β_i be the number of nodes of size $n_i + 1$. Note that $\alpha_i n_i + \beta_i(n_i + 1) = N$, and since $\alpha_i + \beta_i = 2^i$, we have $\beta_i = N - 2^i n_i$. What we have just stated follows easily using induction. For $i = 0$, $n_0 = N$, $\alpha_0 = 1$, and $\beta_0 = 0$. Suppose the result is true for $j = i$. There are two cases, depending on whether n_j is even or odd. Suppose n_j is even. Then $n_{j+1} = n_j/2$ and each α_j node is doubled. Similarly, the β_j nodes all have size $n_j + 1$ and its two children become size n_{j+1} and size $n_{j+1} + 1$ at level $j + 1$. Hence, $\alpha_{j+1} = 2\alpha_j + \beta_j$ and $\beta_{j+1} = \beta_j$. Also, $N = \alpha_j n_j + \beta_j(n_j + 1) = 2\alpha_j n_{j+1} + \beta_j n_{j+1} + \beta_j(n_{j+1} + 1) = \alpha_{j+1} n_{j+1} + \beta_{j+1}(n_{j+1} + 1)$. If n_j is odd, then $n_{j+1} = (n_j - 1)/2$ and $n_{j+1} + 1 = (n_j + 1)/2$. The α_j nodes split into nodes of size n_{j+1} and $n_{j+1} + 1$, while the β_j nodes double with size $n_{j+1} + 1$. Hence, $\alpha_{j+1} = \alpha_j$, $\beta_{j+1} = \alpha_j + 2\beta_j$, and it easily follows that $N = \alpha_{j+1} n_{j+1} + \beta_{j+1}(n_{j+1} + 1)$ and $\alpha_{j+1} + \beta_{j+1} = 2^{j+1}$. This completes the induction proof. For any $N > 0$ there exists *k* such that $2^{k-1} < N \leq 2^k$. For these *N*, the binary tree will have depth $k + 1$. Each of the leaves corresponds to the **if** clause of Figure 1. At level $k - 1$, $n_{k-1} = 1$ and $N = 2^{k-1} + \beta_{k-1}$. This means that there are α_{k-1} leaves at level $k - 1$ and $2\beta_{k-1}$ leaves at level *k*. We have just proved Theorem 1.

Theorem 1

The recursive Cholesky factor algorithm gives rise to a binary tree with *N* leaves. There are $k + 1$ levels, where *k* is defined by $2^{k-1} < N \leq 2^k$, $i = 0, \cdots, k$. Let $n_i = \lfloor N/2^i \rfloor$. At each level *i* there are 2^i nodes, and α_i of these nodes denote a Cholesky factor problem of size n_i. The remaining $\beta_i = 2^i - \alpha_i$ nodes denote Cholesky factor problems of size $n_i + 1$. Also, $\alpha_i n_i + \beta_i(n_i + 1) = N$. At level $i = k - 1$, $n_i = 1$ and $\beta_i > 0$ unless $N = 2^k$ when $n_k = 1$ and $\alpha_k = N$. Assuming $N \neq 2^k$, there are α_i leaves at level *i* and $2\beta_i$ leaves at level *k*.

In going from level *i* to level $i + 1$, the number of Cholesky factorizations doubles, but their size is halved. This means that the total number of FLOPS decreases by a factor of 4 in going down one tree level. More precisely, when *n* is even, the factor ratio is $4 + 12/(n^2 - 4)$; it is exactly 4 if *n* is odd. Since MAs are conserved, we conclude

that $n(n^2 - 1)/6 = n^3/8 + n(n^2 - 4)/24$ (n even) and $n(n^2 - 1)/6 = n(n - 1)(n + 1)/8 + n(n^2 - 1)/24$ (n odd) must be identities (which they are). These identities succinctly quantify the MA count of the **else** clause. DTRSM and DSYRK consume exactly three times the number of MAs of the two recursive calls if n is odd, and an MA of ratio $3 + 12/(n^2 - 4)$ if n is even. These assertions lead to Theorem 2.

Theorem 2
Let $TMA(i)$ be the Total MA count at level i of the 2^i Cholesky subproblems of sizes n_i and $n_i + 1$ where $n_i = \lfloor N/2^i \rfloor$. Also, let $TTS(i)$ be the Total of the dTrsm plus dSyrk MA counts at level i. We have $TMA(i) \geq 4TMA(i + 1)$ and $TTS(i) \geq 3TMA(i + 1)$.

• *Comparison of left-looking blocking versus recursive blocking*
Let $N = nNB$ so that **A** is represented as an n by n block matrix of block size NB. The left-looking and the recursive algorithms consist of n block-factor steps and $n - 1$ calls to DTRSM and DSYRK. Additionally, the left-looking algorithm calls DGEMM $n - 2$ times. Consider the n block-factor steps. Both the left-looking and the recursive algorithms access the same operands, which are the diagonal blocks. Since they require a total of only n blocks, we do not include them below in the formulas for $LLTA$ and RTA. (Please refer to the Introduction, where $LLTA$ and RTA are defined.) Here we use a new concept of BLAS operand total area to measure the efficiency of a dense linear-algebra algorithm. The total FLOP count for all calls is the same for both algorithms. Each call to DTRSM, DSYRK, or DGEMM can be considered a series of block matrix operations on square blocks of size NB. Each of these block operations is either a DSYRK, DTRSM, or DGEMM operation. We now compute the total area of the operands of the $n - 1$ DTRSM, DSYRK, and DGEMM calls for both the left-looking and the recursive algorithms. The operands for the recursive algorithm are nearly square. For a fixed area the FLOP count is maximized for square operands. Since both algorithms do the same number of FLOPS, one can expect that the total area of the operands for the recursive algorithm is less than or equal to the total area of the operands for the left-looking algorithm. For $n > 2$, this turns out to be true. And for $N = 2$, the operands of the left-looking and recursive algorithms are the same. In **Figure 3**, we give the LAPACK left-looking algorithm DPOTRF, and, in **Figure 4**, the LAPACK level-2 algorithm DPOTF2. Suppose we set $NB = 1$ in DPOTRF. Then the call to DSYRK becomes the DDOT computation of DPOTF2. Similarly, the DGEMM and DTRSM calls in DPOTRF become the DGEMV and DSCAL computations of DPOTF2. Thus, routine DPOTF2 is a special case of routine DPOTRF; namely, the case $NB = 1$. In **Figure 5**,

```
do J = 1,N
  A(J,J) = A(J,J) -DDOT(J-1,A(1,J),1,A(1,J),1)
  A(J,J) = SQRT(A(J,J))
  if(J.LT.N) then
   call DGEMV('T',J-1,N-J,-ONE,A(1,J+1),LDA,
$              A(1,J),1,ONE,A(J,J+1),LDA)
   call DSCAL(N-J,ONE/A(J,J),A(J,J+1),LDA)
  end if
end do
```

Figure 4
Main loop of LAPACK routine DPOTF2 (*uplo* = '*U*').

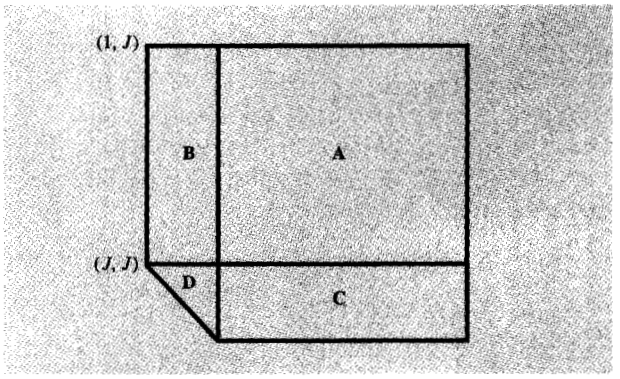

Figure 5
Matrices processed by DPOTRF at step *J*.

we give a computational snapshot of the processing done by DPOTRF at block step $J = jNB + 1$ of Figure 3. DSYRK updates triangular matrix $\mathbf{D} = \mathbf{D} - \mathbf{B}^T\mathbf{B}$, where matrices **B** and **D** have sizes jNB by NB and NB by NB. DGEMM updates matrix $\mathbf{C} = \mathbf{C} - \mathbf{B}\mathbf{A}^T$, where matrices **A** and **C** have sizes jNB by $(n - j - 1)NB$ and NB by $(n - j - 1)NB$. DTRSM solves $\mathbf{D}^T\mathbf{C} = \mathbf{C}$. During block step J, **B** and **D** are used as DSYRK operands; **B**, **A**, and **C** are used as DGEMM operands; and **D** and **C** are used as DTRSM operands. Hence, **A** is used once, while **B**, **C**, and **D** are each used twice. The total area of the operands used is $[(j + 2)(n - j + 1) - 2]NB^2$ for $1 < j < n - 1$. For $j = 0$, DTRSM uses nNB^2 area, and for $j = n - 1$, DSYRK uses nNB^2 area. Summing from $j = 0$ to $n - 1$ gives

$$LLTA(N) = n(n - 1)(n + 10)NB^2/6. \quad (1)$$

Now we compute the total area of the operands for the recursive Cholesky algorithm. Between two recursive calls (problem size in $N = nNB$) there is a call to DTRSM

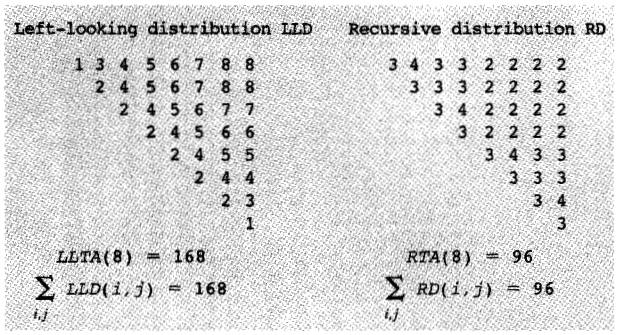

Figure 6

Comparison of total areas for LLD and RD for $n = 8$.

with triangle size $N1 = (n/2)NB$ and rectangle size $N1$ by $N2 = N - N1$. The call to DTRSM is followed by a call to DYSRK with the same $N1$ by $N2$ rectangle and a triangle of size $N2$. The total area is $N1(N1 + 1)/2 + 2N1N2 + N2(N2 + 1)/2$. It follows that the recursive total area satisfies the equation

$$RTA(n) = 2RTA(k) + k(3k + 1) \qquad n \text{ even} = 2k, \tag{2}$$

$$RTA(n) = RTA(k) + RTA(k + 1) + (3k + 1)(k + 1) \qquad n \text{ odd} = 2k + 1. \tag{3}$$

Let $n = \Sigma_{i \geq 0} n_i 2^i$ be the base-2 representation of n and $L = 1 + \lfloor \log_2 n \rfloor$. Then, using (2) and (3), one finds

$$RTA(N) = \left[3n^2/2 - n/2 - 2^L + Ln - \sum_{n_i \geq 0} n_i 2^i \left(i/2 + \sum_{j>i} n_j \right) \right] NB^2. \tag{4}$$

In particular, if n is a power of 2, $n = 2^k$, then

$$RTA(N) = [3(2^{2k-1} - 2^{k-1}) + k \cdot 2^{k-1}]NB^2.$$

Using Equations (1) and (4), we compute the data in **Table 1** for the values of $LLTA$, RTA, and $LLTA/RTA$. Table 1 shows that $RTA(n) < LLTA(n)$ when $n > 2$. It is instructive to consider, as in **Figure 6**, a particular value for n, say $n = 8$, and exhibit the distribution of the matrix operand blocks that sum to TA. For $n = 8$ and $NB = 100$ one would be computing the Cholesky factor of an 800 by 800 matrix using a blocked algorithm where the blocks are square of order 100.

The block submatrix $\mathbf{U}_{ij}$ of Figure 6 is used $LLD(i, j)$ or $RD(i, j)$ times as a matrix operand by DGEMM, DTRSM, or DSYRK when DPOTRF or recursive Cholesky is executed. Table 1 shows that when $n > 10$, the recursive algorithm uses fewer than half the number of blocks used by the left-looking algorithm. For large n the ratio $LLTA/RTA$ approaches $1 + N/9$. This fact can be deduced from (1) and (4). This limit has more meaning for level-2 codes, because then $n = N$.

The analysis can also be used to compare the data movement between a level-2 and a level-3 LAPACK code. Take the above example of $N = 800$. Using Equation (1) with $n = 800$ and $NB = 1$ and $n = 8$, $NB = 100$, one can compute that the $LLTA$ level-2, level-3 ratio is 51.36. Similarly, the $LLTA$ level-2, RTA ratio is 89.89, and the $LLTA$ level-3, RTA ratio is 1.75. The results of this section are now stated as Theorem 3.

Theorem 3

Let $N = nNB$. The Cholesky LAPACK left-looking algorithm DPOTRF (DPOTF2 when $NB = 1$) makes $n - 1$ calls to DSYRK and DTRSM and $n - 2$ calls to DGEMM. The total area of the matrix operands for these calls is $LLTA(N)$. The recursive Cholesky factor algorithm makes $n - 1$ calls to DSYRK and DTRSM. The total area of the matrix operands for these calls is $RTA(N)$. For $n > 2$, $RTA(N) < LLTA(N)$. The $LLTA(N)/RTA(N)$ ratio is approximately $(n + 10)/[9 + 3k/(n - 1)]$, where $k = \log_2 n$.

3. Recursive LU factorization with partial pivoting

In **Figure 7** we give the algorithm. Without loss of generality, we assume that **A** is M by N where $M \geq N$. [If $N > M$, apply the algorithm to $A_{11} = \mathbf{A}(1{:}M, 1{:}M)$. It returns $PA_{11} = L_{11}U_{11}$. Let $A_{12} = \mathbf{A}(1{:}M, M + 1{:}N)$. Now solve $L_{11}X = PA_{12}$ for X.] In the **else** clause there are two recursive calls, one on matrix $\mathbf{A}(1{:}M, 1{:}N1)$, the other on matrix $\mathbf{A}(J1{:}M, J1{:}N)$. There are two calls to DLASWP on matrices $\mathbf{A}(1{:}M, J1{:}N)$ and $\mathbf{A}(J1{:}M, 1{:}N1)$ and a call to level-3 BLAS routines DTRSM and DGEMM. We can use the same three integers (ISW, J, N) at each recursion level i to handle the recursion explicitly in FORTRAN 77. $ISW = 0$ means make first recursive call, $ISW = 1$ means perform a forward-interchange, solve, update, and make the second recursive call, and $ISW = 2$ means perform a backward interchange and return; J denotes the diagonal position of N in the global matrix **A**; and N denotes the current column dimension of the submatrix.

• *Analysis of recursive LU factorization*

The analysis is not as simple as in Cholesky factorization because there is a variable M in addition to the recursion variable N. According to Figure 7, we execute the **else** clause unless $N = 1$. In **Figure 8**, we depict this situation as a node (at level i) in a tree with two branches to level $i + 1$ which denote the two recursive calls. Between these recursive calls there are calls to DLASWP, DTRSM, and

```
LU factor PA(1:M, 1:N) = LU

if (N = 1) then
   compute PA = LU; i.e., find pivot, interchange it and scale
else
   N1 = N/2
   N2 = N-N1
   J1 = N1+1
   LU factor P1 A(1:M, 1:N1) = LU(1:M, 1:N1) (recursive call)
   forward pivot A(1:M, J1:N) = P1 A(1:M, J1:M) (DLASWP)
   solve L(1:N1, 1:N1) X = A(1:N1, J1:N) (DTRSM)
   update A(J1:M, J1:N) = A(J1:M, J1:N) = L(J1:M, 1:N1) U(1:N1, J1:N)(DGEMM)
   LU factor P2 A(J1:M, J1:N) = LU (J1:M, J1:N) (recursive call)
   back pivot A(J1:M, 1:N1) = P2 A(J1:M, 1:N1) (DLASWP)
end if
```

Figure 7

High-level description of the recursive LU algorithm.

Table 1 Values of *RTA*, *LLTA*, and *LLTA/RTA* for various *n*.

n	2	3	4	5	6	7	8	9	10
RTA(*n*)	4	12	22	37	54	74	96	124	154
LLTA(*n*)	4	13	28	50	80	119	168	228	300
LLTA/RTA	1.00	1.08	1.27	1.35	1.48	1.61	1.75	1.84	1.95
n	11	12	13	14	15	16	17	18	19
RTA(*n*)	187	222	261	302	346	392	445	500	558
LLTA(*n*)	385	484	598	728	875	1040	1224	1428	1653
LLTA/RTA	2.06	2.18	2.29	2.41	2.53	2.65	2.75	2.86	2.96
n	20	30	40	50	60	70	80	90	100
RTA(*n*)	618	1382	2456	3828	5494	7472	9752	12332	15206
LLTA(*n*)	1900	5800	13000	24500	41300	64400	94800	133500	181500
LLTA/RTA	3.07	4.20	5.29	6.40	7.52	8.62	9.72	10.83	11.94
n	200	300	400	500	600				
RTA(*n*)	60512	135858	241224	376522	542016				
LLTA(*n*)	1393000	4634500	10906000	21207500	36539000				
LLTA/RTA	23.020	34.113	45.211	56.325	67.413				
n	700	800	900	1000					
RTA(*n*)	737454	962848	1218222	1503544					
LLTA(*n*)	57900500	86292000	122713500	168165000					
LLTA/RTA	78.514	89.622	100.732	111.846					

DGEMM. After completion of the second recursive call, there is a second call to DLASWP. In this analysis we neglect the cost of the two calls to DLASWP. We can see then that at each node that is not a leaf, there are two calls to DLASWP and single calls to DTRSM and DGEMM. The number of MAs needed to LU-factor an m by n matrix is $n(n-1)[m-(n+1)/3]/2$, and the number of MAs needed to LU-factor both the $(m, n1)$ and $(m-n1, n2)$ submatrices is $n(n-2)[m-(5n+4)/12]/4$ if n is even and $(n-1)\{(n-1)m-[(5n-3)(n+1)]/12\}/4$ if n is odd. When $n = 2k$, the MA cost of both DTRSM and DGEMM is $k^2[m-(k+1)/2]$, and

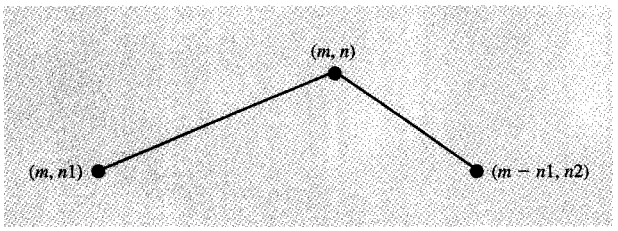

Figure 8

Part of a tree diagram that describes the recursive LU factorization algorithm; $n1 + n2 = n$ and $n1 = n/2$.

when $n = 2k + 1$, the MA cost of both DTRSM and DGEMM is $k(k + 1)[m - (k + 1)/2]$. Again, the MAs are conserved so that we have the identity that the number of MAs needed to factor an (n, m) problem equals the number of MAs needed to perform DTRSM and DGEMM plus the number of MAs needed to factor both an $(m, n1)$ and an $(m - n1, n2)$ problem.

Let $n_i = \lfloor N/2^i \rfloor$. At level i there will be 2^i nodes, each of which will have column dimension n_i or $n_i + 1$. Let α_i be the number of nodes of size n_i and β_i be the number of nodes of size $n_i + 1$. Again we have that $\alpha_i + \beta_i = 2^i$, $\alpha_i n_i + \beta_i(n_i + 1) = N$ and $\beta_i = N - 2^i n_i$. However, the 2^i ms at level i are variable, because the m size of the right branch in Figure 8 depends on n_i. Let $M(\alpha_i)$ be the set of the α_i ms and $M(\beta_i)$ be the set of the β_i ms. If n_i is even, then $\alpha_{i+1} = 2\alpha_i + \beta_i$, $\beta_{i+1} = \beta_i$, $M(\alpha_{i+1}) = M(\alpha_i) \cup \{M(\alpha_i) - n_i/2\} \cup M(\beta_i)$ and $M(\beta_{i+1}) = \{M(\beta_i) - n_i/2\}$. If n_i is odd, then $\alpha_{i+1} = \alpha_i$, $\beta_{i+1} = \alpha_i + 2\beta_i$, $M(\alpha_{i+1}) = M(\alpha_i)$, and $M(\beta_{i+1}) = M(\beta_i) \cup \{M(\alpha_i) - (n_i - 1)/2\} \cup \{M(\beta_i) - (n_i + 1)/2\}$. This specification follows from Figure 8. We now use induction to establish these results. For $i = 0$, $n_0 = N$, $\alpha_0 = 1$, $\beta_0 = 0$, $M(\alpha_0) = \{M\}$, and $M(\beta_0) = \varnothing$. We want to show that $\alpha_i n_i + \beta_i(n_i + 1) = N$, where $\alpha_i + \beta_i = 2^i$. We also indicate how the 2^i different m_i change. Suppose the result is true for $j = i$. The result $\alpha_{i+1} n_{i+1} + \beta_{i+1}(n_{i+1} + 1)$ follows exactly as it did in Section 2. The result about $M(\alpha_{i+1})$ and $M(\beta_{i+1})$ is straightforward. Suppose n_i is even. Let $m \in M(\alpha_i)$. According to Figure 8, there will be two ms at level $i + 1$, namely m and $m - n_i/2$. If $m \in M(\beta_i)$, then, since $n_i + 1$ is odd, its left branch will have $n_{i+1} = n_i/2$; thus, this m belongs to $M(\alpha_{i+1})$. The right branch of (m, n_i) will have $n_{i+1} = n_i/2 + 1$, and so this m produces for β_{i+1} a value $m - n_i/2$. Suppose n_i is odd. Let $m \in M(\alpha_i)$. The left branch has node $(m, (n_i - 1)/2)$, so $m \in M(\alpha_{i+1})$. The right branch has an n value $n_{i+1} + 1$ and an m value $[m - (n_i - 1)/2]$. Hence, this m value belongs to $M(\beta_{i+1})$. Finally, let $m \in M(\beta_i)$. The n value of both children is $n_{i+1} + 1 = (n_i + 1)/2$. The corresponding m values are m and $m - (n_i + 1)/2$. Both of these values belong to $M(\beta_{i+1})$. The argument in Section 2 about recursion variable n at level k, where k is defined by $2^{k-1} < N \le 2^k$, is the same here. We have thus proved the following theorem.

Theorem 4

The recursive LU-factor algorithm gives rise to a binary tree with N leaves. There are $k + 1$ levels, where k is defined by $2^{k-1} < N \le 2^k$, $i = 0, \cdots, k$. Let $n_i = \lfloor N/2^i \rfloor$. At each level i there are 2^i nodes, and α_i of these nodes is an (m, n_i) LU-factor problem where $m \in M(\alpha_i)$. The remaining $\beta_i = 2^i - \alpha_i$ nodes is an $(m, n_i + 1)$ LU-factor problem where $m \in M(\beta_i)$. At level $i = k - 1$, $n_i = 1$ and $\beta_i > 0$ unless $N = 2^k$, and then $n_k = 1$ and $\alpha_k = N$. Assuming $N \neq 2^k$, there are α_i leaves at level i and $2\beta_i$ leaves at level k.

In going from level i to level $i + 1$, the number of LU factorizations doubles, but each of their n sizes is halved. The m sizes are variable at both level i and level $i + 1$. Thus, all we can say is that the total FLOP count at level i for all 2^i LU-factor problems is more than twice the total FLOP count of all 2^{i+1} LU-factor problems at level $i + 1$. This result follows from examining the MA count for an (m, n) problem versus the MA count for its two children. The (m, n) MA count is $n(n - 1)[m - (n + 1)/3]/2$, and if n is even, the MA count for the children is $n(n - 2)[m - (5n + 4)/12]/4$. For n odd, the MA count for the children is $(n - 1)^2[m - (5n - 3)(n + 1)/12(n - 1)]/4$. In either case, by inspection, the (m, n) MA count for each node is more than twice the count for the children. This establishes the following theorem.

Theorem 5

Let $TMA(i)$ be the total MA count at level i of the 2^i LU subproblems of sizes (m_{ij}, n_i) and $(m_{ij}, n_i + 1)$, where $n_i = \lfloor N/2^i \rfloor$ and $1 \le j \le 2^i$. Also, let $TTG(i)$ be the Total of the MA counts at level i for dTrsm plus dGemm. We have $TMA(i) \ge 2TMA(i + 1)$ and $TTG(i) \ge TMA(i + 1)$.

The interpretation of Theorem 5 is that the FLOP count decreases according to a variable geometric series of ratio $r > 1/2$ as one goes down one level in the recursion tree.

- *Comparison of right-looking blocking versus recursive blocking for LU factorization*

Let $N = nNB$ so that **A** is represented as an n by n block matrix of block size NB. Both the right-looking and the recursive algorithms consist of n block-factor steps and $n - 1$ calls to both DTRSM and DGEMM. Consider the n block-factor steps. Both the right-looking and the recursive algorithms access the same operands, which are the n diagonal column blocks. Since they require a total of only n column blocks, we do not include them below in the

```
do J=1,MIN(M,N),NB
  JB=MIN(MIN(M,N)-J+1,NB)
  call DGETF2(M-J+1,JB,A(J,J),LDA,IPIV(J),INFO)
  do I=J,MIN(M,J+JB-1)
    IPIV(I)=J - 1 + IPIV(I)
  end do
  call DLASWP(J-1,A,LDA,J,J+JB-1,IPIV,1)
  if( J+JB.LE.N ) then
    call DLASWP(N-J-JB+1,A(1,J+JB),LDA,J,J+JB-1,IPIV,1)
    call DTRSM('L','L','N','U',JB,N-J-JB+1,ONE,A(J,J),LDA,
$               A(J,J+JB),LDA)
   if( J+JB.LE.M ) then
     call DGEMM('N','N',M-J-JB+1,N-J-JB+1,JB,-ONE,A(J+JB,J),
$              LDA,A(J,J,+JB),LDA,ONE,A(J+JB,J+JB),LDA)
   end if
  end if
end do
```

Figure 9

Main loop of LAPACK routine DGETRF.

formulas for *RLTA* and *RTA*. (Please refer to the Introduction, where *RLTA* and *RTA* are defined.) Here we are using a new concept of BLAS operand total area to measure the efficiency of a dense linear-algebra algorithm. The total FLOP count for all these calls is the same for both algorithms. Each call to DTRSM or DGEMM can be considered a series of block matrix operations on square blocks of size *NB*. Each block operation is either a DTRSM or a DGEMM call. We now compute the total area of the operands of the $n - 1$ DTRSM and DGEMM calls for both the right-looking and the recursive algorithms. The operands for the recursive algorithm are nearly square. For a fixed area the FLOP count is maximized for square operands. Since both algorithms do the same number of FLOPS, one can expect that the total area of the operands for the recursive algorithm is less than or equal to the total area of the operands for the right-looking algorithm. For $n > 3$, this turns out to be true.

In **Figure 9**, we show the LAPACK right-looking algorithm DGETRF. Note that if one calls DGETRF with $NB = 1$, then functionally the calls to DTRSM and DGEMM become, respectively, a no-operation and a call to DGER. Also, the two calls to DLASWP become a no-operation and a call to DSWAP. Hence, the following analysis also applies to DGETF2 if one sets $NB = 1$. In **Figure 10**, we detail the DGETRF computation during a single block step $J = jNB + 1$, $0 \le j < n$ of Figure 9. After factorization of the column panel at J, J (which we neglect in this analysis), there is a call to DLASWP to interchange pivot rows in **B**, **C**. Both the right-looking and the recursive algorithms access the same total area of operands in their calls to DLASWP; namely, $n(n - 1)NB^2$ total area. However, the pattern of access is different for the two algorithms. For DGETRF, at each block step J, the two calls to DLASWP access $(n - 1)NB^2$ area. For recursive LU, at each tree node (see Figure 8) the two calls to DLASWP access $2N1N2NB^2$ area. On the basis of these remarks, we may neglect these contributions to the total area. Next DTRSM is called with matrix operands **D**, **B** [DTRSM sizes are *NB* and $(n - j - 1)NB$], followed by a call to DGEMM with matrix operands **A**, **B**, **C** [DGEMM sizes are $(m - j - 1)NB$, $(n - j - 1)NB$, NB]. DTRSM solves $\mathbf{DB} = \mathbf{B}$, where **D** is unit lower triangular and DGEMM updates $\mathbf{C} = \mathbf{C} - \mathbf{AB}$. Hence **A**, **C**, **D** are each used once, while **B** is used twice. The total area of the operands used is $[n - j + (m - j)(n - j) - 1]NB^2$ for $0 \le j < n$. Summing from $j = 0$ to $n - 1$ gives

$$RLTA(M, N) = \{[n(n + 1)/2](m - n/3 + 4/3) - m - 1\}NB^2. \quad (5)$$

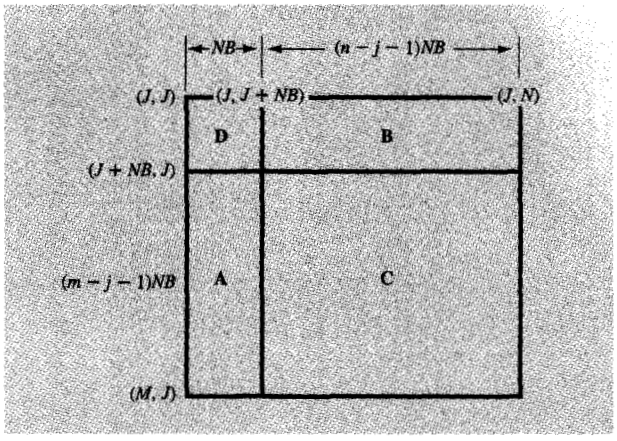

Figure 10

Matrices processed by DGETRF at step J.

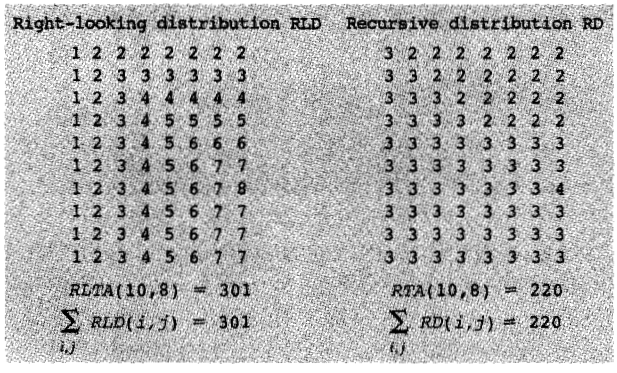

Figure 11

Comparison of total areas for RLD and RD for $m, n = 10, 8$.

Now we compute the total area for recursive LU. Between two recursive calls [problem size is $(M, N) = (mNB, nNB)$] there is a call to DTRSM with triangle size $N1 = (n/2)NB$ and rectangle size $N1$ by $N2 = N - N1$. The call to DTRSM is followed by a call to DGEMM with rectangles $M - N1$ by $N1$, $N1$ by $N2$, and $M - N1$ by $N2$. The total area is $N1(N1 + 1)/2 + 2N1N2 + N(M - N1)$. Let $n = 2k$ or $2k + 1$ be even or odd. It follows that recursive total area RTA satisfies the following equations:

$$RTA(m, 2k) = RTA(m, k) + RTA(m - k, k) + k[2m + (k + 1)/2], \quad (6)$$

$$RTA(m, 1) = 0;$$

$$RTA(m, 2k + 1) = RTA(m, k) + RTA(m - k, k + 1) + 5k(k + 1)/2 + (2k + 1)(m - k). \quad (7)$$

Using (6) and (7) we can find a partial solution for RTA. Let $L = 1 + \lfloor \log_2 n \rfloor$. Then

$$RTA(M, N) = \{[(L + 1)n - 2^L]m + f(n)\}NB^2, \quad (8)$$

where

$$f(2^k) = 2^k[k(-2 \cdot 2^k + 1) + 5(2^k - 1)]/4. \quad (9)$$

In particular, if $n = 2^k$ is a power of 2 and $m = n$, then

$$RTA(N, N) = 2^{k-2}[k(2^{k+1} + 1) + 5(2^k - 1)]NB^2. \quad (10)$$

Using (5), (6), and (7), we compute the data in **Table 2** for the values of $RLTA(m, n)$, $RTA(m, n)$, and $RLTA(m, n)/RTA(m, n)$. Table 2 shows that $RTA(m, n) < RLTA(m, n)$ when $n > 3$. It is instructive to consider, as in **Figure 11**, a particular value for (m, n) and exhibit the distribution of the matrix operand blocks that sum to TA. This pattern is general. For $(m, n) = (10, 8)$ and $NB = 100$, one would be computing the LU factorization of a 1000 by 800 matrix using a blocked algorithm where the blocks are square of order 100. The block submatrix $\mathbf{A}_{ij}$ is used $RLD(i, j)$ or $RD(i, j)$ times as a matrix operand by either DTRSM or DGEMM when either DGETRF or RGETRF is executed. Table 2 shows that for $n = 17$ the recursive algorithm uses fewer than half the number of blocks used by the right-looking algorithm. An approximation to (8) when $M = N$ is

$$RTA(N, N) = 0.25n[\log_2 n(2n + 1) + 5(n - 1)]NB^2. \quad (11)$$

Thus, for any N the ratio of $RLTA/RTA$ is approximated by using (11). This analysis can also be used to compare the data movement between a level 2- and a level-3 code. Take the above example of $(M, N) = (1000, 800)$. Using Equation (5) with $(m, n) = (1000, 800)$ and $NB = 1$ and $(m, n) = (10, 8)$, $NB = 100$, one can compute that the $RLTA$ level-2, level-3 ratio is 78.201. Similarly, the level-2 $RLTA$, RTA ratio is 106.994, and the level-3 $RLTA$, RTA ratio is 1.368. The results of this section are now stated as Theorem 6.

Theorem 6

Let $(M, N) = (mNB, nNB)$. The LU right-looking LAPACK algorithm DGETRF (DGETF2 when $NB = 1$) and the recursive LU algorithm make $n - 1$ calls to both DTRSM and DGEMM. The total area of the matrix operands for these calls is respectively $RLTA(M, N)$ and $RTA(M, N)$; see Equations (5) and (8), (9). For $n > 3$, $RTA(M, N) < RLTA(M, N)$, and the $RLTA(N, N)/RTA(N, N)$ ratio is approximately $4n/(3 \log_2 n)$.

Table 2 Values of *RTA*, *RLTA*, and *RLTA/RTA* for various m, n.

m, n	2,2	3,3	4,4	5,5	6,6	7,7	8,8	9,9	10,10
$RTA(m, n)$	5	16	33	57	89	127	172	225	289
$RLTA(m, n)$	5	16	35	64	105	160	231	320	429
$RLTA/RTA$	1.00	1.00	1.06	1.12	1.18	1.26	1.34	1.42	1.48
m, n	11,11	12,12	13,13	14,14	15,15	16,16	17,17	18,18	19,19
$RTA(m, n)$	359	439	524	618	719	828	946	1080	1219
$RLTA(m, n)$	560	715	896	1105	1344	1615	1920	2261	2640
$RLTA/RTA$	1.56	1.63	1.71	1.79	1.87	1.95	2.03	2.09	2.17
m, n	20,20	30,30	40,40	50,50	60,60	70,70	80,80	90,90	100,100
$RTA(m, n)$	1373	3343	6316	10275	15191	21227	28492	36768	46125
$RLTA(m, n)$	3059	9889	22919	44149	75579	119209	177039	251069	343299
$RLTA/RTA$	2.23	2.96	3.63	4.30	4.98	5.62	6.21	6.83	7.44

m, n	200,200	300,300	400,400	500,500	600,600
$RTA(m, n)$	204500	485749	897900	1434981	2123048
$RLTA(m, n)$	2706599	9089899	21493199	41916499	72359799
$RLTA/RTA$	13.235	18.713	23.937	29.210	34.083

m, n	700,700	800,800	900,900	1000,1000
$RTA(m, n)$	2949567	3911200	5007675	6239212
$RLTA(m, n)$	114823099	171306399	243809699	334332999
$RLTA/RTA$	38.929	43.799	48.687	53.586

m, n	4,2	6,3	8,4	10,5	12,6	14,7	16,8	18,9	20,10
$RTA(m, n)$	9	31	65	117	185	267	364	486	629
$RLTA(m, n)$	9	31	71	134	225	349	511	716	969
$RLTA/RTA$	1.00	1.00	1.09	1.15	1.22	1.31	1.40	1.47	1.54

m, n	40,20	60,30	80,40	100,50	200,100
$RTA(m, n)$	3133	7783	14956	24575	113325
$RLTA(m, n)$	7239	23809	55679	107849	848199
$RLTA/RTA$	2.31	3.06	3.72	4.39	7.48

m, n	400,200	600,300	800,400	1000,500
$RTA(m, n)$	513300	1232149	2293100	3678981
$RLTA(m, n)$	6726599	22634899	53573199	104541499
$RLTA/RTA$	13.105	18.370	23.363	28.416

4. Experimental results

The recursive DPOTF2 and DGETF2 algorithms which we name RPOTF2 and RGETF2 have been implemented and tested. See Appendix A and Appendix B for algorithms RPOTF2 and RGETF2. We wish to verify our conjecture that DPOTF2 and RGETF2 outperform both DPOTF2, DPOTRF and DGETF2, DGETRF for large matrices. In all experiments we use $M = N$ and $LDA = M + 1$. This experimental verification demonstrates that the variable square blocking that recursion automatically imparts to the algorithm does indeed lead to higher performance than conventional fixed blocking of the right- or left-looking variety.[2] Please note that level-3 codes use blocking and our recursive versions do not. Two examples make this point clear. Suppose we consider $N = 1000$. For DGETRF, this is the TPP (Toward Peak Performance) benchmark [12]. The default block size of LAPACK is 64. DGETRF makes 16 calls to DGETF2, 15 calls to DTRSM and DGEMM, and 30 calls to DLASWP. On the other hand, the experiments use pure recursion; i.e., the block size is 1. Thus, there are 999 calls to DTRSM, DGEMM, 1998 calls to DLASWP, and 1000 calls to IDAMAX and DSCAL. Now suppose $N = 50$. Here DGETRF makes a single call to DGETF2, whereas RGETF2 makes 49 calls to DTRSM, DGEMM, 98 calls to DLASWP, and 50 calls to IDAMAX and DSCAL. Our point is that for peak performance one must include explicit blocking with recursion (see also the last paragraph of the Introduction). This is especially true for small matrices, where level-3 performance drops off drastically because of the nature of the cubic function and the calling overheads and error checking. The point we have just made helps explain our performance results and demonstrates our main conclusion: The automatic variable blocking that recursion imparts to these two algorithms leads to higher performance than conventional fixed blocking of the right- or left-looking variety.

[2] DGETRF uses a right-looking algorithm; DPOTRF uses a left-looking algorithm.

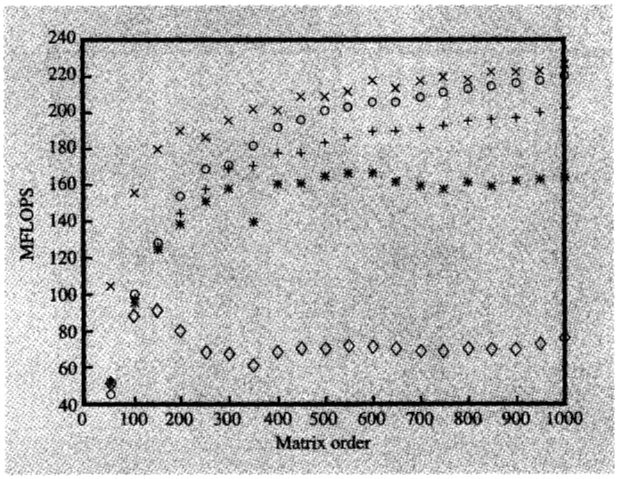

Figure 12

MFLOP performance versus matrix order of five algorithms for LU factorization. The plot points (×, o, +, *, ◇) are for DGEF, RGETF2, DGETRF(C), DGETRF(R), and DGETF2, respectively.

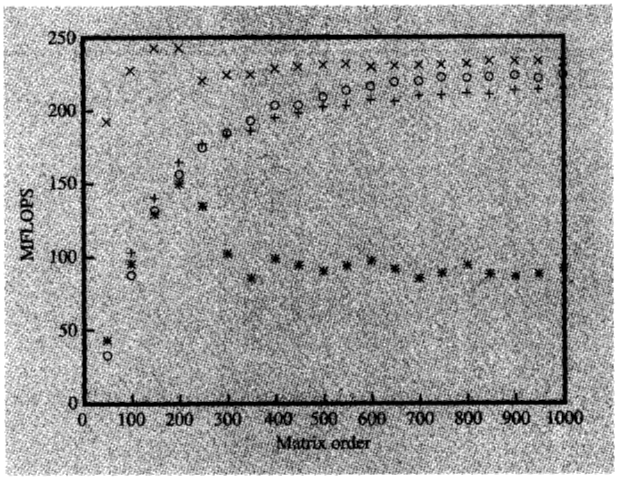

Figure 14

MFLOP performance versus matrix order of four algorithms for Cholesky factorization. (Case *uplo* = *'L'*.) The plot points (×, o, +, *) are for DPOF, RPOTF2, DPOTRF, and DPOTF2, respectively.

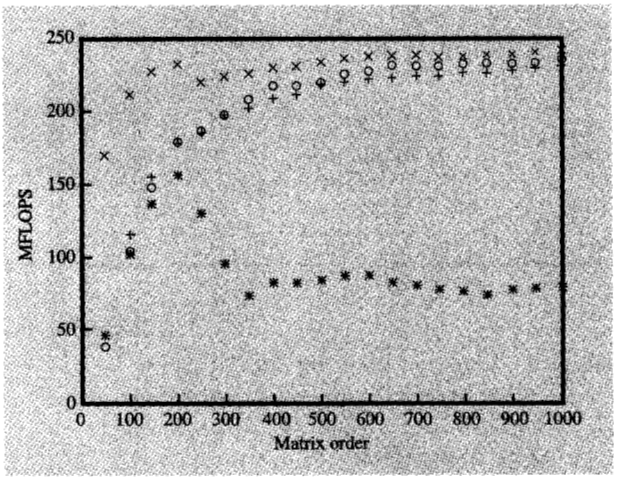

Figure 13

MFLOP performance versus matrix order of four algorithms for Cholesky factorization. (Case *uplo* = *'U'*.) The plot points (×, o, +, *) are for DPOF, RPOTF2, DPOTRF, and DPOTF2, respectively.

A single set of experiments was done on an IBM RS/6000 workstation; see **Figures 12**, **13**, and **14**. The results establish experimentally our main conclusion stated above. We now briefly describe algorithms RPOTF2 and RGETF2. The coding style is that of LAPACK. In fact, we took the public-domain LAPACK codes DPOTF2 and DGETF2 and imbedded the recursive codes of Figures 1 and 7 into those routines. Also, we have added many detailed comments. We have also modified LAPACK auxiliary routine DLASWP by interchanging the order of the two **do** loops that make up this code. (As mentioned, this gives rise to a performance gain of more than 15% for large matrices.) The performance times were obtained using the real-time clock in the machine; hence, all times are wall-clock times and therefore system overheads are included. In [8], Toledo describes a similar experiment, but for only one matrix size, $N = 1000$. He uses $LDA = 1007$ and 1024. For $LDA = 1024$, the effective cache size shrinks dramatically as many congruence-class slots go unused. In this case, it is very beneficial to use the purely recursive version of the algorithm. In a second experiment, Toledo considers matrices of order $N = 200$ to 2000. It was only for matrices at or above size $n = 300$ that his recursive algorithm began to show superiority over DGETRF. Our experiments show that the crossover is around $n = 100$. For DGETF2 the crossover also occurs around size 100. Our experiments plot performance of matrices 50 to 1000 in steps of 50. We made two to four runs and took the minimum of the wall-clock times.

In Figure 12 we compare DGETF2, DGETRF(C, R), RGETF2, and DGEF. The choppiness in the graphs is partly due to a bad LDA problem; e.g., DGETRF(R) for $n = 350$. For Cholesky, we consider both values of *uplo* = *'L'* and *'U'*, and we compare RPOTF2 results to DPOTF2, DPOTRF and ESSL DPOF. (See Figures 13 and 14.) For *uplo* = *'U'*, routine RPOTF2 outperforms DPOTF2, DPOTRF when the matrix size exceeds

$n = 100$ and at about $n = 200$. The gain over DPOTF2 approaches 3 and the gain over DPOTRF approaches 1.02 as N approaches 1000. Routine RPOTF2 always has less performance than the ESSL DPOF; its gain approaches 0.98 as n approaches 1000. For *uplo* = '*L*', the results are similar. The crossover points are around 150 and 250 for DPOTF2 and DPOTRF. The gains approach 2.5 and 1.05 over DPOTF2 and DPOTRF as N approaches 1000. For ESSL DPOF the gain approaches 0.97.

Here is a further explanation of these results. Variable blocking that is produced by recursion is superior if it is combined with blocking. Nonetheless, the ESSL routines DGEF and DPOF outperform the recursive versions. Now, the ESSL routines use a large block size that is greater than 100 and hence make few calls to DTRSM, DGEMM, and their factor kernels. For small matrix sizes, ESSL routines outperform the recursive routines by wide margins. The purely recursive routines do not make up this loss.

To verify this point, we make a minor change to RPOTF2 and RGETF2 by introducing a blocking parameter NB. For $n \leq NB$, a factor kernel is called; otherwise the algorithm is the same. This minor change results in excellent performance for n values $\leq NB$ and better performance for $n > NB$. We substituted the factor kernels from ESSL. With this change the performance of RPOTF2 and RGETF2 becomes about equal to that of DPOF and DGEF from ESSL.

5. Conclusions

Routines RPOTF2 and RGETF2 should be used in place of the LAPACK DPOTF2 and DGETF2 routines. One could modify these recursive routines to include a factor kernel. At present, these routines factor a 1 by 1 matrix or an m by 1 matrix. In their present form there is no blocking parameter to choose. Thus, a user cannot make a poor blocking choice. We have shown that level-2 routines can possibly be made level-3 by introducing recursion. This is certainly true for the Cholesky algorithm and for general LU factorization.

The use of recursion for dense linear-algebra algorithms is a powerful blocking technique. When it is combined with explicit blocking of the memory hierarchy it becomes even more powerful, although we have not considered that aspect in this paper. Instead we have concentrated on pure recursion and the automatic variable blocking that is implicit in using it. The results are surprisingly good. In fact, they outperform the corresponding level-3 LAPACK routines. However, for small matrices, recursion suffers and hence it alone does not present a universal answer.

As a by-product of this work, we have discovered a way to improve a library such as LAPACK by taking advantage of the relationship between submatrix operands of multiple BLAS calls in a LAPACK algorithm. The BLAS routines have to be modified to accept block submatrix operands. A new BLAS implementation may become simpler, and perhaps it will be possible to provide a generic version of these BLAS routines with LAPACK. Currently the GEMM-based BLAS group at the University of Umea and J. Wasnieski at Uni-C in Denmark are considering such a project.

Acknowledgment

Thanks to Don Coppersmith, who derived Equations (4), (8), and (9), and to Anshul Gupta and Sivan Toledo for useful discussions about this work. Thanks also to Bernard Rudin for a careful reading and suggestions for improving clarity.

Appendix A: RPOTF2

```
      SUBROUTINE RPOTF2( UPLO, N, A, LDA, INFO )
      implicit none
*
*     .. Scalar Arguments ..
      CHARACTER          UPLO
      INTEGER            INFO, LDA, N
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   A( 0:LDA±1,0:* )
*     ..
*
* Purpose
* =====
*
* RPOTF2 computes the Cholesky factorization of
* a real symmetric positive definite matrix A.
*
* The factorization has the form
*    A = U' * U ,  if UPLO = 'U', or
*    A = L  * L',  if UPLO = 'L',
* where U is an upper triangular matrix
* and L is lower triangular.
*
* This is a new recursive version of DPOTF2
* ( done in F77 ).
* The key idea is to produce a mostly
* level-3 component by introducing
* recursion. Recursion introduces variable
* blocking, which is more general than fixed
* blocking. Hence, this code will probably
* outperform the level-3 version DPOTRF.
*
*  ALGORITHM DESCRIPTION ( UPLO = 'U' case )
*
* IF ( N = 1 ) THEN
*
*   compute A = UT*U; i.e., compute sqrt
*   or issue non-P.D. message;
*
* ELSE
*
*   partition A into three block matrices A11,
*   A12, and A22, where A11 = A( 1:n,1:n ),
*   A12 = A( 1:n,n+1:N ), and
*   A22 = A( n+1:N,n+1:N ), and n is about N/2.
*   Perform four computations:
```

```
*
*   (1) Cholesky factor ( A11 ) = ( U11T \ U11 )
*       ( recursive call )
*
*   (2) compute U12 : A12 = U11T**(-1) *  A12
*       ( DTRSM )
*
*   (3) update  A22 : A22 = A22 - U12T *  U12
*       ( DSYRK )
*
*   (4) Cholesky factor ( A22 ) = ( U22T \ U22 )
*       ( recursive call )
*
* ENDIF
*
* Notes :
*
* i)   Recursion leads to automatic variable
*      blocking.
* ii)  Calls to DTRSM and DSYRK routines
*      automatically make this code level 3.
*      In essence, blocking is implicit.
*      DTRSM and DSYRK are each called
*      N - 1 times.
* iii) The recursion tree has N leaves.
*      Each leaf has size 1.
* iv)  The stack keeps track of the current col
*      dimension m of A and its position
*      J on the diagonal. Also needed is
*      isk(1,isp), which has values 0,1 to
*      signify that computation (1) is, or
*      computations (2), (3), (4) are
*      to be done next.
*      Note that computations (1) and (4) are
*      recursive calls, so that recursion
*      level isp needs to know where to
*      resume after the return from recursion
*      level isp + 1.
* v)   Most of the FLOPS are performed at the
*      top of the tree. For level 0,
*      N**3/4 FLOPS are done in one call each
*      to DTRSM and  DSYRK. At level i,
*      N**3/8**(i+1) FLOPS are performed
*      by each of the 2**i calls to
*      ( m = n = k = N/2**(i+1) ) DTRSM and
*      DSYRK.
*      The total FLOPS count at this level is
*      2*2**i*N/8**(i+1) = N/4**(i+1).
* vi)  At the lowest level, the matrix sizes
*      are 1 by 1. For each level up the tree
*      the matrix sizes double and the number
*      of calls is halved.
* vii) For small matrices the FLOP rate of
*      level-3 codes reduces drastically
*      ( reason for 'lite' BLAS ). Hence, at the
*      lower levels the FLOP rate starts to
*      drop off dramatically. To overcome
*      this, one should stop recursion when
*      N/2**(i+1) <= constant and replace the
*      subtree computation with a single call
*      to a factor kernel. This is what a
*      level-3 code would do.
*
* Arguments
* =======
* UPLO    (input) CHARACTER*1
*         Specifies whether the upper or lower
*         triangular part of the symmetric
*         matrix A is stored:
*         = 'U':  Upper triangular;
*         = 'L':  Lower triangular.
*
* N       (input) INTEGER
*         The number of rows and columns of the
*         matrix A.  N >= 0.
*
* A       (input/output) DOUBLE PRECISION array,
*         dimension( 0:LDA-1, 0:N-1). On entry,
*         the symmetric matrix A. If UPLO = 'U',
*         the upper triangular part of A
*         is used and the part of A below
*         the diagonal is not referenced;
*         if UPLO = 'L', the lower triangular
*         part of A is used and the part of A
*         above the diagonal is not referenced.
*         On exit, the factor U or L from the
*         Cholesky factorization.
*
* LDA     (input) INTEGER
*         The leading dimension of the array A.
*         LDA >= max(1,N).
*
* INFO    (output) INTEGER
*         = 0: successful exit;
*         < 0: if INFO = -k, the kth argument
*                            had an illegal
*                            value;
*         > 0: if INFO = k,  the leading minor
*                            of order K is
*                            not positive
*                            definite, and
*                            the factorization
*                            could not be
*                            completed.
*
*  =====================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0,
     $                     ZERO = 0.0D+0 )
*     ..
*     .. Local Arrays ..
      LOGICAL UPPER
      INTEGER   isk(3,0:20) ! handle matrices to
*               size 2**20
*     ..
*     .. Local Scalars ..
      INTEGER            J, m, m1, m2, J1, isp
*     ..
*     .. External Functions ..
      LOGICAL            LSAME
      EXTERNAL           LSAME
*     ..
*     .. External Subroutines ..
      EXTERNAL           DSYRK, DTRSM, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, DSQRT
*     ..
*     .. Executable Statements ..
*
*     Test the input parameters.
*
      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      IF( .NOT.UPPER  .AND.
     $    ( .NOT.LSAME( UPLO, 'L' ) ) ) THEN
          INFO = -1
      ELSE IF( N.LT.0 ) THEN
          INFO = -2
```

```
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'RPOTF2', -INFO )
         RETURN
      END IF
*
*     Quick return if possible.
*
      IF( N.EQ.0 )
     $   RETURN
*
*     Initialize variables for recursion.
*
      isp = -1
      J = 0
      m = N ! Recursion matrix is input matrix A.
*
*     CALL RPOTF2( UPLO, m, A( J, J ),
*                  lda, INFO )
*     Push the stack isk.
*
    1 isp=isp+1      ! Make recursive call by
                       pushing down stack.
      isk(1,isp)=0
      isk(2,isp)=J
      isk(3,isp)=m
*
*     A branch to label 2 happens ONLY
*     during intermediate recursion
*     with isk(1,isp) > 0 and m > 1. Hence,
*     we will continue execution of this
*     intermediate recursion.
*
    2 continue

      IF( m.EQ.1 ) then ! lowest recursion level.
*
*       Compute pivot and test for non-positive-
*       definiteness.
*
        IF( A( J, J ).LE.ZERO )then
          INFO = J + 1 ! origin 1
          isp = 0 ! force immediate return
        ELSE
          A( J, J ) = DSQRT( A( J, J ) )
        END IF
      ELSE ! here m > 1 and the recursion is
*     intermediate.
*
*       Set recursion variables J1, m1, and m2.
*       At level isp, four computations will be
*       done as isk(1,isp) takes the values 0,1.
*       For each of these values one makes a
*       recursive call by branching out.
*       Exit from this clause occurs only when
*       isk(1,isp) = 2.
*
        m1=m/2
        m2=m-m1
        J1=J+m1

        IF( isk(1,isp).EQ.0 )then
*
*         Set up RPOTF2( UPLO, m1, A( J, J ),
*    $                   lda, INFO )
*         call by setting return value
*         and new values for J and m.
*         This is computation (1).
*
          isk(1,isp)=1
          m=m1 ! J already set
          goto 1

        ELSE IF( isk(1,isp).EQ.1 )then

          IF( UPPER ) THEN
*
*       Solve for A( J:J1-1,J1:J1+m2-1 ).
*       This is computation (2).
*
            CALL DTRSM( 'Left', 'Upper',
     $                  'Transpose', 'Non-unit',
     $                  m1, m2, ONE, A( J, J ),
     $                  LDA, A( J, J1 ), LDA )
*
*       Update for A( J1:J1+m2-1,J1:J1+m2-1 ).
*       This is computation (3).
*
            CALL DSYRK( 'Upper', 'Transpose',
     $                  m2, m1, -ONE, A( J, J1 ),
     $                  LDA, ONE, A( J1, J1 ), LDA )

          ELSE
*
*       Solve for A( J1:J1+m2-1,J:J1-1 ).
*       This is computation (2).
*
            CALL DTRSM( 'Right', 'Lower',
     $                  'Transpose', 'Non-unit',
     $                  m2, m1, ONE, A( J, J ),
     $                  LDA, A( J1, J ), LDA )
*
*       Update for A( J1:J1+m2-1,J1:J1+m2-1 ).
*       This is computation (3).
*
            CALL DSYRK( 'Lower', 'No transpose',
     $                  m2, m1, -ONE,
     $                  A( J1, J ), LDA, ONE,
     $                  A( J1, J1 ), LDA )

          END IF
*
*         Set up RPOTF2( UPLO, m2, A( J1, J1 ),
     $                   lda, INFO )
*         call by setting return value
*         and new values for J and m.
*         This is computation (4).
*
          isk(1,isp)=2
          m = m2
          J = J1
          goto 1

        END IF

*     END IF
*
*     Rtn from call RPOTF2( UPLO, m, A( J, J ),
*                           LDA, INFO ).
*     Pop the stack isk and return to the next
*     recursion level.
*
      isp=isp-1
      IF(  isp.GE.0 )then
        J=isk(2,isp)
        m=isk(3,isp)
        goto 2
      END IF
*     if(isp.GE.0)goto 2
```

```
      RETURN
*
*     End of RPOTF2
*
      END
```

Appendix B: RGETF2

```
      SUBROUTINE RGETF2( M, N, A, LDA, IPIV,
     $                   INFO )
      implicit none
*
*     .. Scalar Arguments ..
      INTEGER            INFO, LDA, M, N
*     ..
*     .. Array Arguments ..
      INTEGER            IPIV( 0:* ) ! origin 0
      DOUBLE PRECISION   A( 0:LDA-1,0:* ) !
*                        origin 0
*     ..
*
*  Purpose
*  =====
*
*  RGETF2 computes an LU factorization of a
*  general M-by-N matrix A using partial
*  pivoting with row interchanges.
*
*  The factorization has the form
*    A = P * L * U,
*  where P is a permutation matrix, L is lower
*  triangular with unit diagonal elements
*  (lower trapezoidal if m > n), and U is upper
*  triangular (upper trapezoidal if m < n).
*
*  This is a new recursive version of DGETF2
*  ( done in F77 ).
*  The key idea is to produce a mostly level-3
*  component by introducing recursion.
*  Recursion introduces variable blocking,
*  which is more general than fixed blocking.
*  Hence, this code will probably outperform
*  the level-3 version DGETRF.
*
*              ALGORITHM DESCRIPTION
*
*  Wlog assume M >= N . If N > M,
*  apply algorithm to A11 = A( 1:M,1:M ).
*  It returns P * A11 = L11 * U11.
*  Let A12 = A( 1:M,M+1:N ). Now compute
*  U12 = L11**(-1) * P * A12.
*
*  IF ( N <= 1 ) THEN
*
*    compute P*A = L*U; i.e., find the pivot,
*    interchange it, and scale;
*
*  ELSE
*
*    partition A into four block matrices A11,
*    A12, A21, and A22, where
*    A11 = A( 1:n,1:n ), A12 = A( 1:n,n+1,N ),
*    A21 = A( n+1:M,1:N ), and
*    A22 = A( n+1:M,n+1:N ),
*    and n is about N/2.
*    Perform six computations:
*
*    (1) factor P1 * ( A11 ) = ( L11 \ U11 )
*                    ( A21 )   ( L21       )
*                    ( recursive call )
*
*    (2) forward
*        pivot ( A12 ) : ( A12 ) = P1 * ( A12 )
*              ( A22 )   ( A22 )        ( A22 )
*                    ( DLASWP )
*
*    (3) compute U12 : A12 = L11**(-1)  *  A12
*                    ( DTRSM )
*
*    (4) update  A22 : A22 = A22 - L11  * U12
*                    ( DGEMM )
*
*    (5) factor P2 * A22 = ( L22 \ U22 )
*                    ( recursive call )
*
*    (6) back pivot A21  :  A21 = P2 * A21
*                    ( DLASWP )
*
*  ENDIF
*
*  Notes :
*
*  i)   Recursion leads to automatic variable
*       blocking.
*  ii)  Calls to DTRSM and DGEMM routines
*       automatically make this code level 3.
*       In essence, blocking is implicit.
*  iii) The recursion tree has ceil( min(m,n) )
*       leaves. At each leaf ns = 1.
*  iv)  The stack keeps track of the current
*       col dimension ns of A and its position
*       J on the diagonal. Also needed is
*       isk(1,isp), which has values 0,1,2 to
*       signify that computation (1) is,
*       computations (2), (3), (4), (5) are,
*       or computation (6) is to be done
*       next. Note that computations (1) and (5)
*       are recursive calls, so that recursion
*       level isp needs to know where to
*       resume after the return from recursion
*       level isp + 1.
*
*  Arguments
*  =======
*
*  M       (input) INTEGER
*          The number of rows of the matrix A.
*          M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the
*          matrix A.  N >= 0.
*
*  A       (input/output) DOUBLE PRECISION
*          array, dim. (0:LDA-1,0:N-1).
*          On entry, the m × n matrix to be
*          factored. On exit, the factors
*          L and U; the unit diagonal
*          elements of L are not stored.
*
*  LDA     (input) INTEGER
*          The leading dimension of the
*          array A.  LDA >= max(1,M).
*
*  IPIV    (output) INTEGER array, dimension
*          (0:min(M,N)-1).
*          The pivot indices. Row i of the
*          matrix was interchanged with row
*          IPIV(i).
*
*  INFO    (output) INTEGER
*          = 0: successful exit;
```

```
*          < 0: if INFO = -k, the kth argument
*                             had an illegal
*                             value;
*          > 0: if INFO = k,  U(k,k) is exactly
*                             zero. The
*                             factorization
*                             has been completed,
*                             but the factor
*                             U is exactly
*                             singular, and
*                             division by zero
*                             will occur if it
*                             is used to solve
*                             a system of
*                             equations or to
*                             compute the
*                             inverse of A.
*
*  =====================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0 ,
     $                     ZERO = 0.0D+0 )
*     ..
*     .. Local Arrays ..
      INTEGER            isk(3,0:20) !
*                        min(M,N) <= 2**20
*     ..
*     .. Local Scalars ..
      INTEGER            J, JP, isp, J1, ns,
     $                   n1s, n2s
      DOUBLE PRECISION   t
*     ..
*     .. EXTERNAL FUNCTIONS ..
      INTEGER            IDAMAX
      EXTERNAL           IDAMAX
*     .. External Subroutines ..
      EXTERNAL           DGEMM, DLASWP, DTRSM,
     $                   DSCAL, XERBLA
      EXTERNAL           push, pop
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          MAX, MIN
*     ..
*     .. Executable Statements
*
*     Test the input parameters.
*
      INFO = 0
      IF( M.LT.0 ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, M ) ) THEN
         INFO = -4
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'RGETF2', -INFO )
         RETURN
      END IF
*
*     Quick return if possible.
*
      IF( M.EQ.0 .OR. N.EQ.0 )
     $   RETURN
*
*     Initialize variables for recursion.
*
      isp = -1
      J = 0
      ns = MIN( M ,N ) ! recursion matrix is
*     M rows by ns cols
*
      CALL RGETF2( M-J, ns, A( J, J ),
*                  LDA, IPIV( J ), INFO )
*     Push the stack isk.

    1 call push( J, ns, isp, isk)
*
*     A branch to label 2 happens ONLY
*     during intermediate recursion with
*     isk(1,isp) = 1 or 2 and ns > 1.
*     Hence, we will continue execution
*     of this intermediate recursion.

    2 CONTINUE

      IF( ns.LE.1 ) THEN ! lowest recursion level
*
*       Find pivot, check for singularity,
*       interchange and scale.
*
        JP = J  + IDAMAX( M-J, A( J, J ), 1 )
     $                    - 1  ! origin 0
        IPIV( J ) = JP + 1 ! origin 1

        IF( A( JP, J ).NE.ZERO ) THEN
          IF( JP.NE.J ) THEN
            t = A( J, J )
            A( J, J ) = A(JP, J )
            A(JP, J ) = t
          END IF
          IF( M-1-J.GT.0 )
             CALL DSCAL( M-1-J,ONE/A( J, J ),
     $                   A( J+1, J ), 1 )
        ELSE
          INFO = J + 1 ! origin 1
        END IF

      ELSE ! Here ns > 1 and the recursion
         is intermediate.
*
*       Set recursion variables J1, n1s, and n2s.
*       At level isp, six computations will
*       be done as isk(1,isp) takes the
*       values 0,1,2. For values 0 and 1 one
*       makes a recursive call by branching
*       out. Exit from this clause occurs
*       only when isk(1,isp) = 2.
*
        n1s = ns/2 ! fixed blocking
        n2s = ns - n1s
        J1 = J + n1s

        IF( isk(1,isp).EQ.0 ) THEN ! RGETF2
*
*         Set up RGETF2( M-J, n1s, A( J, J ),
*                        LDA, IPIV( J ), INFO )
*         call by setting return value and
*         new values for J and ns.
*         This is computation (1).
*
          isk(1,isp) = 1 ! Set return value
*         to 1 on the current stack.
          ns = n1s ! J is already set.
          GOTO 1 ! Call is made there.

        ELSEIF( isk(1,isp).EQ.1 ) THEN
*       Do computations DLASWP, DTRSM,
*                       DGEMM, RGETF2
*
```

```
*       Forward pivot cols J1:J1+n2s-1 of
*       A( J:M-1, J1:J1+n2s-1 ).
*       This is computation (2).
*
        CALL DLASWP( n2s, A( 0, J1 ), LDA,
     $               J+1, J1, IPIV, 1 )
*
*       Compute u( J:J+n1s-1,  J1:J1+n2s-1)
*                  = 1**-1*a.
*       This is computation (3).
*
        CALL DTRSM( 'Left', 'Lower',
     $              'No transpose', 'Unit',
     $              n1s, n2s, ONE, A( J, J ),
     $              LDA, A( J, J1 ), LDA )
*
*       Update  A( J1: J1+M-1, J1: J1+n2s-1)
*                    = a - 1*u.
*       This is computation (4).
*
        CALL DGEMM( 'No transpose',
     $              'No transpose',
     $              M- J1, n2s, n1s,
     $              -ONE, A( J1, J ), LDA,
     $              A( J, J1 ), LDA, ONE,
     $              A( J1, J1 ), LDA)
*
*       Set up  RGETF2( M-J1, n2s, A( J1, J1 ),
*                       LDA, IPIV( J1 ), INFO )
*       call by setting return value and
*       new values for J and ns.
*       This is computation (5).
*
        isk(1,isp) = 2 ! Set return value
*       to 2 on the current stack.
        ns = n2s
        J = J1
        GOTO 1 ! Call is made there.

      ELSE ! Back pivot and return.
*
*       Back pivot cols J to J1-1 of
*       A( J1:M-1, J:J1-1).
*       This is computation (6).
*
        CALL DLASWP( n1s, A(0, J ),
     $               LDA, J1+1, J1+n2s,
     $               IPIV, 1 )
      END IF

    END IF
*
*   Rtn from call RGETF2( M-J, ns, A( J, J ),
*                         LDA, IPIV( J ),
*                         INFO ).
*   Pop the stack isk and return to the
*   next recursion level.
*
    Call pop( isp, isk, J, ns )
    IF( isp.GE.0 )GOTO 2

    IF( N.GT.M  ) THEN
*
*     Forward pivot cols M :N - 1 of a.
*
      CALL DLASWP( N-M , A(0,M ), LDA,
     $             1, M, IPIV, 1 )
*
*     Compute u(0:M -1,M :N - 1) = 1**-1*a
*
      CALL DTRSM( 'Left', 'Lower',
     $            'No transpose', 'Unit',
     $            M, N-M, ONE, A, LDA,
     $            A(0,M ), LDA)
    END IF
*
    return
*
*   End of recursive RGETF2
*
    end
    subroutine push(js,ns,sp,stack)
    implicit none
    integer*4 sp,js,ns,stack(3,0:*)
    sp=sp+1
    stack(1,sp)=0
    stack(2,sp)=js
    stack(3,sp)=ns
    return
    end
    subroutine pop(sp,stack,js,ns)
*   implicit none
    integer*4 sp,js,ns,stack(3,0:*)
    sp=sp-1 ! recursive return
    if(sp.ge.0)then
      js=stack(2,sp)
      ns=stack(3,sp)
    endif
    return
    end
```

References

1. *Engineering and Scientific Subroutine Library Version 2 Release 2, Guide and Reference*, Volumes 1–3, Order No. SC23-0526-01, 1994; available through IBM branch offices.
2. *IMSL*, IMSL Incorporated, 2500 Park West Tower 1, 2500 City Blvd., Houston, TX 77042.
3. E. Anderson, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, "LAPACK: A Portable Linear Algebra Library for High-Performance Computers" (LAPACK Working Note 20), *Computer Science Department Technical Report CS-90-105*, University of Tennessee, Knoxville, 1990.
4. The MathWorks, Incorporated, 20 North Main Street, Sherborn, MA 01770.
5. NAG Ltd., Wilkinson House, Jordon Hill Road, Oxford OX8 YDE, England.
6. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Rev.* **26,** No. 1, 91–112 (January 1984).
7. Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM Publications, Philadelphia, 1991.
8. S. Toledo, "Locality of Reference in LU Decomposition with Partial Pivoting," *Research Report RC-20344*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, January 1996. Also to appear in *SIAM J. Matrix Anal. & Appl.*
9. J. J. Dongarra, J. Bunch, C. Moler, and G. Stewart, *LINPACK User's Guide*, SIAM Publications, Philadelphia, 1979.
10. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Exploiting Functional Parallelism of POWER2 to Design High-Performance Numerical Algorithms," *IBM J. Res. Develop.* **38,** 563–576 (1994).
11. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 2nd ed., The Johns Hopkins Press, Baltimore, 1989.

12. Jack J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software" (Linpack Benchmark Report), *Computer Science Department Technical Report CS-89-85*, University of Tennessee, Knoxville, 1997. *http:/www.netlib.org/benchmark/performance.ps*

Received June 6, 1997; accepted for publication August 8, 1997

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (GUSTAV at YKTVMV, gustav@watson.ibm.com).*
Dr. Gustavson manages the Algorithms and Architectures group in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute, joining IBM Research in 1963. One of Dr. Gustavson's primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. He has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2 for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Outstanding Invention Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a Fellow of the IEEE.