14

How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination

Jack Dongarra and Piotr Luszczek

The increasing availability of advanced-architecture computers, at affordable costs, has had a significant effect on all spheres of scientific computation. In this chapter, we'll show the need for designers of computing algorithms to make expeditious and substantial adaptations to algorithms, in reaction to architecture changes, by closely examining one simple but important algorithm in mathematical software: Gaussian elimination for the solution of linear systems of equations.

At the application level, science has to be captured in mathematical models, which in turn are expressed algorithmically and ultimately encoded as software. At the software level, there is a continuous tension between performance and portability on the one hand, and understandability of the underlying code. We'll examine these issues and look at trade-offs that have been made over time. Linear algebra—in particular, the solution of linear systems of equations—lies at the heart of most calculations in scientific computing. This chapter focuses on some of the recent developments in linear algebra software designed to exploit advanced-architecture computers over the decades.

There are two broad classes of algorithms: those for dense matrices and those for sparse matrices. A matrix is called *sparse* if it contains a substantial number of zero elements. For sparse matrices, radical savings in space and execution time can be achieved through specialized storage and algorithms. To narrow our discussion and keep it simple, we will look only at the *dense matrix problem* (a dense matrix is defined as one with few zero elements).

Much of the work in developing linear algebra software for advanced-architecture computers is motivated by the need to solve large problems on the fastest computers available. In this chapter, we'll discuss the development of standards for linear algebra software, the building blocks for software libraries, and aspects of algorithm design as

influenced by the opportunities for parallel implementation. We'll explain motivations for this work, and say a bit about future directions.

As representative examples of dense matrix routines, we will consider Gaussian elimination, or LU factorization. This examination, spanning hardware and software advances over the past 30 years, will highlight the most important factors that must be considered in designing linear algebra software for advanced-architecture computers. We use these factorization routines for illustrative purposes not only because they are relatively simple, but also because of their importance in several scientific and engineering applications that make use of boundary element methods. These applications include electromagnetic scattering and computational fluid dynamics problems.

The past 30 years have seen a great deal of activity in the area of algorithms and software for solving linear algebra problems. The goal of achieving high performance in code that is portable across platforms has largely been realized by the identification of linear algebra kernels, the Basic Linear Algebra Subprograms (BLAS). We will discuss the LINPACK, LAPACK, and ScaLAPACK libraries, which are expressed in successive levels of the BLAS. See "Further Reading" at the end of this chapter for discussions of these libraries.

The Effects of Computer Architectures on Matrix Algorithms

The key motivation in the design of efficient linear algebra algorithms for advanced-architecture computers involves the storage and retrieval of data. Designers wish to minimize the frequency with which data moves between different levels of the memory hierarchy. Once data is in registers or the fastest cache, all processing required for this data should be performed before it gets evicted back to the main memory. Thus, the main algorithmic approach for exploiting both vectorization and parallelism in our implementations uses *block-partitioned* algorithms, particularly in conjunction with highly tuned kernels for performing matrix-vector and matrix-matrix operations (the Level-2 and Level-3 BLAS). Block partitioning means that the data is divided into blocks, each of which should fit within a cache memory or a vector register file.

The computer architectures considered in this chapter are:

- Vector machines
- RISC computers with cache hierarchies
- Parallel systems with distributed memory
- Multi-core computers

Vector machines were introduced in the late 1970s and early 1980s. They were able in one step to perform a single operation on a relatively large number of operands stored in vector registers. Expressing matrix algorithms as vector-vector operations was a natural fit for this type of machines. However, some of the vector designs had a limited ability to load and store the vector registers in main memory. A technique called *chaining* allowed this limitation to be circumvented by moving data between the registers before accessing main memory. Chaining required recasting linear algebra in terms of matrix-vector operations.

RISC computers were introduced in the late 1980s and early 1990s. While their clock rates might have been comparable to those of the vector machines, the computing speed lagged behind due to their lack of vector registers. Another deficiency was their creation of a deep memory hierarchy with multiple levels of cache memory to alleviate the scarcity of bandwidth that was, in turn, caused mostly by a limited number of memory banks. The eventual success of this architecture is commonly attributed to the right price point and astonishing improvements in performance over time as predicted by Moore's Law. With RISC computers, the linear algebra algorithms had to be redone yet again. This time, the formulations had to expose as many matrix-matrix operations as possible, which guaranteed good cache reuse.

A natural way of achieving even greater performance levels with both vector and RISC processors is by connecting them together with a network and letting them cooperate to solve a problem bigger than would be feasible on just one processor. Many hardware configurations followed this path, so the matrix algorithms had to follow yet again as well. It was quickly discovered that good local performance has to be combined with good global partitioning of the matrices and vectors.

Any trivial divisions of matrix data quickly uncovered scalability problems dictated by so-called *Amdahl's Law*: the observation that the time taken by the sequential portion of a computation provides the minimum bound for the entire execution time, and therefore limits the gains achievable from parallel processing. In other words, unless most of computations can be done independently, the point of diminishing returns is reached, and adding more processors to the hardware mix will not result in faster processing.

For the sake of simplicity, the class of multi-core architectures includes both symmetric multiprocessing (SMP) and single-chip multi-core machines. This is probably an unfair simplification, as the SMP machines usually have better memory systems. But when applied to matrix algorithms, both yield good performance results with very similar algorithmic approaches: these combine local cache reuse and independent computation with explicit control of data dependences.

A Decompositional Approach

At the basis of solutions to dense linear systems lies a decompositional approach. The general idea is the following: given a problem involving a matrix A, one factors or decomposes A into a product of simpler matrices from which the problem can easily be solved. This divides the computational problem into two parts: first determine an appropriate decomposition, and then use it in solving the problem at hand.

Consider the problem of solving the linear system:

$$Ax = b$$

where A is a nonsingular matrix of order n. The decompositional approach begins with the observation that it is possible to factor A in the form:

$$A = LU$$

where L is a lower triangular matrix (a matrix that has only zeros above the diagonal) with ones on the diagonal, and U is upper triangular (with only zeros below the diagonal). During the decomposition process, diagonal elements of A (called pivots) are used to divide the elements below the diagonal. If matrix A has a zero pivot, the process will break with division-by-zero error. Also, small values of the pivots excessively amplify

the numerical errors of the process. So for numerical stability, the method needs to interchange rows of the matrix or make sure pivots are as large (in absolute value) as possible. This observation leads to a row permutation matrix P and modifies the factored form to:

$$P^{\mathrm{T}}A = LU$$

The solution can then be written in the form:

$$x = A^{-1}Pb$$

which then suggests the following algorithm for solving the system of equations:

1. Factor A
2. Solve the system $Ly = Pb$
3. Solve the system $Ux = y$

This approach to matrix computations through decomposition has proven very useful for several reasons. First, the approach separates the computation into two stages: the computation of a decomposition, followed by the use of the decomposition to solve the problem at hand. This can be important, for example, if different right hand sides are present and need to be solved at different points in the process. The matrix needs to be factored only once and reused for the different right hand sides. This is particularly important because the factorization of A, step 1, requires $O(n^3)$ operations, whereas the solutions, steps 2 and 3, require only $O(n^2)$ operations. Another aspect of the algorithm's strength is in storage: the L and U factors do not require extra storage, but can take over the space occupied initially by A.

For the discussion of coding this algorithm, we present only the computationally intensive part of the process, which is step 1, the factorization of the matrix.

A Simple Version

For the first version, we present a straightforward implementation of LU factorization. It consists of $n–1$ steps, where each step introduces more zeros below the diagonal, as shown in Figure 14-1.
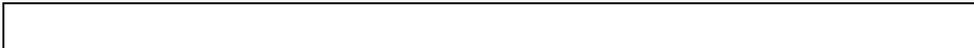
Figure 14-1. LU factorization

A tool often used to teach Gaussian elimination is MATLAB. It features a scripting language (also called MATLAB) that makes developing matrix algorithms very simple. The language might seem very unusual to people familiar with other scripting languages because it is oriented to process multidimensional arrays. The unique features of the language that we use in the example code are:

*   Transposition operator for vectors and matrices: ' (single quote)
*   Matrix indexing specified as:
    *   Simple integer values: `A(m, k)`
    *   Ranges: `A(k:n, k)`

- Other matrices: `A([k m], : )`
- Built-in matrix functions such as `size` (returns matrix dimensions), `tril` (returns the lower triangular portion of the matrix), `triu` (returns the upper triangular portion of the matrix), and `eye` (returns an identity matrix, which contains only zero entries, except for the diagonal, which is all ones)

Example 14-1 shows the simple implementation.

Example 14-1. Simple variant (MATLAB coding)

```
function [L,U,p] = lutx(A)
%LUTX  Triangular factorization, textbook version
%   [L,U,p] = lutx(A) produces a unit lower triangular matrix L,
%   an upper triangular matrix U, and a permutation vector p,
%   so that L*U = A(p,:)

[n,n] = size(A);
p = (1:n)';

for k = 1:n-1

   % Find index 'm' of largest element 'r' below diagonal in k-th column
   [r,m] = max(abs(A(k:n,k)));
   m = m+k-1; % adjust 'm' so it becomes a global index

   % Skip elimination if column is zero
   if (A(m,k) ~= 0)

      % Swap pivot row
      if (m ~= k)
         A([k m],:) = A([m k],:); % swap rows 'k' and 'm' of 'A'
         p([k m]) = p([m k]);     % swap entrix 'k' and 'm' of 'p'
      end

      % Compute multipliers
      i = k+1:n;
      A(i,k) = A(i,k)/A(k,k);

      % Update the remainder of the matrix
      j = k+1:n;
      A(i,j) = A(i,j) - A(i,k)*A(k,j);
   end
end

% Separate result
L = tril(A,-1) + eye(n,n);
U = triu(A);
```

The algorithm presented in Example 14-1 is row-oriented, in the sense that we are taking a scalar multiple of the "pivot" row and adding it to the rows below to introduce zeros below the diagonal. The beauty of the algorithm lies in its similarity to the mathematical notation. Hence, this is the preferred way of teaching the algorithm for the first time so that students can quickly turn formulas into running code.

This beauty, however, has its price. In the 1970s, Fortran was the language for scientific computations. Fortran stores two-dimensional arrays by column. Accessing the array in a row-wise fashion within the matrix could involve successive memory reference to

locations separated from each other by a large increment, depending on the size of the declared array. The situation was further complicated by the operating system's use of memory pages to effectively control memory usage. With a large matrix and a row-oriented algorithm in a Fortran environment, an excessive number of page swaps might be generated in the process of running the software. Cleve Moler pointed this out in the 1970s (see "Further Reading").

To avoid this situation, one needed simply to interchange the order of the innermost nested loops on i and j. This simple change resulted in more than 30 percent savings in wall-clock time to solve problems of size 200 on an IBM 360/67. Beauty was thus traded for efficiency by using a less obvious ordering of loops and a much more obscure (by today's standard) language.

LINPACK's DGEFA Subroutine

The performance issues with the MATLAB version of the code continued as, in the mid-1970s, vector architectures became available for scientific computations. Vector architectures exploit pipeline processing by running mathematical operations on arrays of data in a simultaneous or pipelined fashion. Most algorithms in linear algebra can be easily vectorized. Therefore, in the late 70s there was an effort to standardize vector operations for use in scientific computations. The idea was to define some simple, frequently used operations and implement them on various systems to achieve portability and efficiency. This package came to be known as the Level-1 Basic Linear Algebra Subprograms (BLAS) or Level-1 BLAS.

The term *Level-1* denotes vector-vector operations. As we will see, Level-2 (matrix-vector operations), and Level-3 (matrix-matrix operations) play important roles as well.

In the 1970s, the algorithms of dense linear algebra were implemented in a systematic way by the LINPACK project. LINPACK is a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problems. The package solves linear systems whose matrices are general, banded, symmetric indefinite, symmetric positive definite, triangular, and tridiagonal square. In addition, the package computes the QR and singular value decompositions of rectangular matrices and applies them to least-squares problems.

LINPACK uses column-oriented algorithms, which increase efficiency by preserving locality of reference. By column orientation, we mean that the LINPACK code always references arrays down columns, not across rows. This is important since Fortran stores arrays in column-major order. This means that as one proceeds down a column of an array, the memory references proceed sequentially through memory. Thus, if a program references an item in a particular block, the next reference is likely to be in the same block.

The software in LINPACK was 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.

Example 14-2 shows the LINPACK implementation of factorization.

Example 14-2. LINPACK variant (Fortran 66 coding)

```
subroutine dgefa(a,lda,n,ipvt,info)
integer lda,n,ipvt(1),info
```

```
      double precision a(lda,1)
      double precision t
      integer idamax,j,k,kp1,l,nm1
c
c
c     gaussian elimination with partial pivoting
c
      info = 0
      nm1 = n - 1
      if (nm1 .lt. 1) go to 70
      do 60 k = 1, nm1
         kp1 = k + 1
c
c        find l = pivot index
c
         l = idamax(n-k+1,a(k,k),1) + k - 1
         ipvt(k) = l
c
c        zero pivot implies this column already triangularized
c
         if (a(l,k) .eq. 0.0d0) go to 40
c
c           interchange if necessary
c
            if (l .eq. k) go to 10
               t = a(l,k)
               a(l,k) = a(k,k)
               a(k,k) = t
   10       continue
c
c           compute multipliers
c
            t = -1.0d0/a(k,k)
            call dscal(n-k,t,a(k+1,k),1)
c
c           row elimination with column indexing
c
            do 30 j = kp1, n
               t = a(l,j)
               if (l .eq. k) go to 20
                  a(l,j) = a(k,j)
                  a(k,j) = t
   20          continue
               call daxpy(n-k,t,a(k+1,k),1,a(k+1,j),1)
   30       continue
         go to 50
   40    continue
            info = k
   50    continue
   60 continue
   70 continue
      ipvt(n) = n
      if (a(n,n) .eq. 0.0d0) info = n
      return
      end
```

The Level-1 BLAS subroutines DAXPY, DSCAL, and IDAMAX are used in the routine DGEFA. The main difference between Example 14-1 and Example 14-2 (other than the programming language and the interchange of loop indexes) is the use of routine DAXPY to encode the inner loop of the method.

It was presumed that the BLAS operations would be implemented in an efficient, machine-specific way suitable for the computer on which the subroutines were executed. On a vector computer, this could translate into a simple, single vector operation. This avoided leaving the optimization up to the compiler and explicitly exposing a performance-critical operation.

In a sense, then, the beauty of the original code was regained with the use of a new vocabulary to describe the algorithms: the BLAS. Over time, the BLAS became a widely adopted standard and were most likely the first to enforce two key aspects of software: modularity and portability. Again, these are taken for granted today, but at the time they were not. One could have the cake of compact algorithm representation and eat it too, because the resulting Fortran code was portable.

Most algorithms in linear algebra can be easily vectorized. However, to gain the most out of such architectures, simple vectorization is usually not enough. Some vector computers are limited by having only one path between memory and the vector registers. This creates a bottleneck if a program loads a vector from memory, performs some arithmetic operations, and then stores the results. In order to achieve top performance, the scope of the vectorization must be expanded to facilitate chaining operations together and to minimize data movement, in addition to using vector operations. Recasting the algorithms in terms of matrix-vector operations makes it easy for a vectorizing compiler to achieve these goals.

Thus, as computer architectures 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 Level-2 and Level-3.

LAPACK DGETRF

As mentioned before, the introduction in the late 1970s and early 1980s of vector machines brought about the development of another variant of algorithms for dense linear algebra. This variant was centered on the multiplication of a matrix by a vector. These subroutines were meant to give improved performance over the dense linear algebra subroutines in LINPACK, which were based on Level-1 BLAS. Later on, in the late 1980s and early 1990s, with the introduction of RISC-type microprocessors (the "killer micros") 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, in this case, is matrix multiplication.

The original goal of the LAPACK project was to make the widely used LINPACK library run efficiently on vector and shared-memory parallel processors. On these machines, LINPACK is inefficient because its memory access patterns disregard the multilayered memory hierarchies of the machines, thereby spending too much time moving data instead of doing useful floating-point operations. LAPACK addresses this problem by reorganizing the algorithms to use block matrix operations, such as matrix multiplication, in the innermost loops (see the paper by E. Anderson and J. Dongarra under "Further Reading"). These block operations can be optimized for each architecture to account for

its memory hierarchy, and so provide a transportable way to achieve high efficiency on diverse modern machines.

Here we use the term "transportable" instead of "portable" because, for fastest possible performance, LAPACK requires that highly optimized block matrix operations be implemented already on each machine. In other words, the correctness of the code is portable, but high performance is not—if we limit ourselves to a single Fortran source code.

LAPACK can be regarded as a successor to LINPACK in terms of functionality, although it doesn't always use the same function-calling sequences. As such a successor, LAPACK was a win for the scientific community because it could keep LINPACK's functionality while getting improved use out of new hardware.

Example 14-3 shows the LAPACK solution to LU factorization.

Example 14-3. LAPACK solution factorization

```
SUBROUTINE DGETRF( M, N, A, LDA, IPIV, INFO )
      INTEGER           INFO, LDA, M, N
      INTEGER           IPIV( * )
      DOUBLE PRECISION  A( LDA, * )
      DOUBLE PRECISION  ONE
      PARAMETER         ( ONE = 1.0D+0 )
      INTEGER           I, IINFO, J, JB, NB
      EXTERNAL          DGEMM, DGETF2, DLASWP, DTRSM, XERBLA
      INTEGER           ILAENV
      EXTERNAL          ILAENV
      INTRINSIC         MAX, MIN
      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( 'DGETRF', -INFO )
         RETURN
      END IF
      IF( M.EQ.0 .OR. N.EQ.0 ) RETURN
      NB = ILAENV( 1, 'DGETRF', ' ', M, N, -1, -1 )
      IF( NB.LE.1 .OR. NB.GE.MIN( M, N ) ) THEN
         CALL DGETF2( M, N, A, LDA, IPIV, INFO )
      ELSE
         DO 20 J = 1, MIN( M, N ), NB
            JB = MIN( MIN( M, N )-J+1, NB )
*           Factor diagonal and subdiagonal blocks and test for exact
*           singularity.
            CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ), IINFO )
*           Adjust INFO and the pivot indices.
            IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
            DO 10 I = J, MIN( M, J+JB-1 )
               IPIV( I ) = J - 1 + IPIV( I )
   10       CONTINUE
*           Apply interchanges to columns 1:J-1.
            CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )
```

```
*
            IF( J+JB.LE.N ) THEN
*               Apply interchanges to columns J+JB:N.
                CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1, IPIV, 1 )
*               Compute block row of U.
                CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
     $                     N-J-JB+1, ONE, A( J, J ), LDA, A( J, J+JB ), LDA )
                IF( J+JB.LE.M ) THEN
*                  Update trailing submatrix.
                   CALL DGEMM( 'No transpose', 'No transpose', 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
   20    CONTINUE
      END IF
      RETURN
      end
```

Most of the computational work in the algorithm from Example 14-3 is contained in three routines:

DGEMM

Matrix-matrix multiplication

DTRSM

Triangular solve with multiple right hand sides

DGETF2

Unblocked LU factorization for operations within a block column

One of the key parameters in the algorithm is the block size, called NB here. If NB is too small or too large, poor performance can result—hence the importance of the ILAENV function, whose standard implementation was meant to be replaced by a vendor implementation encapsulating machine-specific parameters upon installation of the LAPACK library. At any given point of the algorithm, NB columns or rows are exposed to a well-optimized Level-3 BLAS. If NB is 1, the algorithm is equivalent in performance and memory access patterns to the LINPACK's version.

Matrix-matrix operations offer the proper level of modularity for performance and transportability across a wide range of computer architectures, including parallel systems with memory hierarchy. This enhanced performance is primarily due to a greater opportunity for reusing data. There are numerous ways to accomplish this reuse of data to reduce memory traffic and to increase the ratio of floating-point operations to data movement through the memory hierarchy. This improvement can bring a three- to ten-fold improvement in performance on modern computer architectures.

The jury is still out concerning the productivity of writing and reading the LAPACK code: how hard is it to generate the code from its mathematical description? The use of vector notation in LINPACK is arguably more natural than LAPACK's matrix formulation. The mathematical formulas that describe algorithms are usually more complex if only matrices are used, as opposed to mixed vector-matrix notation.

Recursive LU

Setting the block size parameter for the LAPACK's LU might seem like a trivial matter at first. But in practice, it requires a lot of tuning for various precisions and matrix sizes. Many users end up leaving the setting unchanged, even if the tuning has to be done only once at installation. This problem is exacerbated by the fact that not just one but many LAPACK routines use a blocking parameter.

Another issue with LAPACK's formulation of LU is the factorization of tall and narrow panels of columns performed by the DGETF2 routine. It uses Level-1 BLAS and was found to become a bottleneck as the processors became faster throughout the 90s without corresponding increases in memory bandwidth.

A solution came from a rather unlikely direction: divide-and-conquer recursion. In place of LAPACK's looping constructs, the newer recursive LU algorithm splits the work in half, factorizes the left part of the matrix, updates the rest of the matrix, and factorizes the right part. The use of Level-1 BLAS is reduced to an acceptable minimum, and most of the calls to Level-3 BLAS operate on larger portions of the matrix than LAPACK's algorithm. And, of course, the block size does not have to be tuned anymore.

Recursive LU required the use of Fortran 90, which was the first Fortran standard to allow recursive subroutines. A side effect of using Fortran 90 was the increased importance of the LDA parameter, the leading dimension of A. It allows more flexible use of the subroutine, as well as performance tuning for cases when matrix dimension m would cause memory bank conflicts that could significantly reduce available memory bandwidth.

The Fortran 90 compilers use the LDA parameter to avoid copying the data into a contiguous buffer when calling external routines, such as one of the BLAS. Without LDA, the compiler has to assume the worst-case scenario when input matrix a is not contiguous and needs to be copied to a temporary contiguous buffer so the call to BLAS does not end up with an out-of-bands memory access. With LDA, the compiler passes array pointers to BLAS without any copies.

Example 14-4 shows recursive LU factorization.

Example 14-4. Recursive variant (Fortran 90 coding)

```fortran
recursive subroutine rdgetrf(m, n, a, lda, ipiv, info)
implicit none

integer, intent(in) :: m, n, lda
double precision, intent(inout) :: a(lda,*)
integer, intent(out) :: ipiv(*)
integer, intent(out) :: info

integer :: mn, nleft, nright, i
double precision :: tmp

double precision :: pone, negone, zero
parameter (pone=1.0d0)
parameter (negone=-1.0d0)
parameter (zero=0.0d0)

intrinsic min
```

```
      integer idamax
      external dgemm, dtrsm, dlaswp, idamax, dscal

      mn = min(m, n)

      if (mn .gt. 1) then
         nleft = mn / 2
         nright = n - nleft

         call rdgetrf(m, nleft, a, lda, ipiv, info)

         if (info .ne. 0) return
         call dlaswp(nright, a(1, nleft+1), lda, 1, nleft, ipiv, 1)

         call dtrsm('L', 'L', 'N', 'U', nleft, nright, pone, a, lda,
     $        a(1, nleft+1), lda)

         call dgemm('N', 'N', m-nleft, nright, nleft, negone,
     $        a(nleft+1,1) , lda, a(1, nleft+1), lda, pone,
     $        a(nleft+1, nleft+1), lda)

         call rdgetrf(m - nleft, nright, a(nleft+1, nleft+1), lda,
     $        ipiv(nleft+1), info)
         if (info .ne. 0) then
            info = info + nleft
            return
         end if

         do i = nleft+1, m
            ipiv(i) = ipiv(i) + nleft
         end do

         call dlaswp(nleft, a, lda, nleft+1, mn, ipiv, 1)

      else if (mn .eq. 1) then
         i = idamax(m, a, 1)
         ipiv(1) = i
         tmp = a(i, 1)

         if (tmp .ne. zero .and. tmp .ne. -zero) then
            call dscal(m, pone/tmp, a, 1)

            a(i,1) = a(1,1)
            a(1,1) = tmp
         else
            info = 1
         end if

      end if

      return
      end
```

There is a certain degree of elegance in the recursive variant. No loops are exposed in the routine. Instead, the algorithm is driven by the recursive nature of the method (see the paper by F. G. Gustavson under "Further Reading").

The Recursive LU Algorithm consists of four basic steps, illustrated in Figure 14-2:

1. Split the matrix into two rectangles ($m * n/2$); if the left part ends up being only a single column, scale it by the reciprocal of the pivot and return.

2. Apply the *LU* algorithm to the left part. Apply transformations to the right part (perform the triangular solve $A_{12} = L^{-1}A_{12}$ and matrix multiplication $A_{22}=A_{22} -A_{21}*A_{12}$).

3. Apply the *LU* algorithm to the right part.

Figure 14-2. Recursive LU factorization
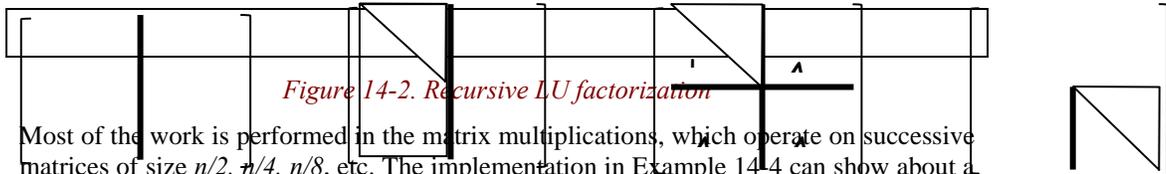
Most of the work is performed in the matrix multiplications, which operate on successive matrices of size *n/2, n/4, n/8*, etc. The implementation in Example 14-4 can show about a 10 percent improvement in performance over the LAPACK implementation given in Example 14-3.

In a sense, any of the previous renditions of the LU algorithm could be considered a step backwards in terms of code elegance. But divide-and-conquer recursion was a tremendous leap forward (even dismissing the modest performance gains). The recursive algorithm for matrix factorization can now be taught to students alongside other recursive algorithms, such as various kinds of sorting methods.

By changing just the size of matrix parts, it is possible to achieve the same memory access pattern as in LINPACK or LAPACK. Setting `nleft` to 1 makes the code operate on vectors, just as in LINPACK, whereas setting `nleft` to NB>1 makes it behave like LAPACK's blocked code. In both cases, the original recursion deteriorates from divide-and-conquer to the tail kind. The behavior of such variations of the recursive algorithm can be studied alongside a quicksort algorithm with various partitioning schemes of the sorted array.

Finally, we leave as an exercise to the reader to try to mimic the recursive code without using recursion and without explicitly handling the recursive call stack—an important problem to solve if the Fortran compiler cannot handle recursive functions or subroutines.

ScaLAPACK PDGETRF

LAPACK is designed to be highly efficient on vector processors, high-performance "superscalar" workstations, and shared-memory multiprocessors. LAPACK can also be used satisfactorily on all types of scalar machines (PCs, workstations, and mainframes). However, LAPACK in its present form is less likely to give good performance on other types of parallel architectures—for example, massively parallel Single Instruction Multiple Data (SIMD) machines, or Multiple Instruction Multiple Data (MIMD) distributed-memory machines. The ScaLAPACK effort was intended to adapt LAPACK to these new architectures.

By creating the ScaLAPACK software library, we extended the LAPACK library to scalable MIMD, distributed-memory, concurrent computers. For such machines, the memory hierarchy includes the off-processor memory of other processors, in addition to the hierarchy of registers, cache, and local memory on each processor.

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms in order to minimize the frequency of data movement between different levels of the memory hierarchy. The fundamental building blocks of the ScaLAPACK library are distributed-memory versions of the Level-2 and Level-3 BLAS, and a set of Basic Linear Algebra Communication Subprograms (BLACS) for communication tasks that arise frequently in parallel linear algebra computations. In the ScaLAPACK routines, all interprocessor communication occurs within the distributed BLAS and the BLACS, so the source code of the top software layer of ScaLAPACK looks very similar to that of LAPACK.

The ScaLAPACK solution to LU factorization is shown in Example 14-5.

Example 14-5. ScaLAPACK variant (Fortran 90 coding)

```fortran
      SUBROUTINE PDGETRF( M, N, A, IA, JA, DESCA, IPIV, INFO )
       INTEGER           BLOCK_CYCLIC_2D, CSRC_, CTXT_, DLEN_, DTYPE_,
      $                  LLD_, MB_, M_, NB_, N_, RSRC_
       PARAMETER         ( BLOCK_CYCLIC_2D = 1, DLEN_ = 9, DTYPE_ = 1,
      $                    CTXT_ = 2, M_ = 3, N_ = 4, MB_ = 5, NB_ = 6,
      $                    RSRC_ = 7, CSRC_ = 8, LLD_ = 9 )
       DOUBLE PRECISION  ONE
       PARAMETER         ( ONE = 1.0D+0 )
       CHARACTER         COLBTOP, COLCTOP, ROWBTOP
       INTEGER           I, ICOFF, ICTXT, IINFO, IN, IROFF, J, JB, JN,
      $                  MN, MYCOL, MYROW, NPCOL, NPROW
       INTEGER           IDUM1( 1 ), IDUM2( 1 )
       EXTERNAL          BLACS_GRIDINFO, CHK1MAT, IGAMN2D, PCHK1MAT, PB_TOPGET,
      $                  PB_TOPSET, PDGEMM, PDGETF2, PDLASWP, PDTRSM, PXERBLA
       INTEGER           ICEIL
       EXTERNAL          ICEIL
       INTRINSIC         MIN, MOD
*     Get grid parameters
      ICTXT = DESCA( CTXT_ )
      CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
*     Test the input parameters
      INFO = 0
      IF( NPROW.EQ.-1 ) THEN
         INFO = -(600+CTXT_)
      ELSE
         CALL CHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, INFO )
         IF( INFO.EQ.0 ) THEN
            IROFF = MOD( IA-1, DESCA( MB_ ) )
            ICOFF = MOD( JA-1, DESCA( NB_ ) )
            IF( IROFF.NE.0 ) THEN
               INFO = -4
            ELSE IF( ICOFF.NE.0 ) THEN
               INFO = -5
            ELSE IF( DESCA( MB_ ).NE.DESCA( NB_ ) ) THEN
               INFO = -(600+NB_)
            END IF
         END IF
         CALL PCHK1MAT( M, 1, N, 2, IA, JA, DESCA, 6, 0, IDUM1, IDUM2, INFO )
      END IF
      IF( INFO.NE.0 ) THEN
         CALL PXERBLA( ICTXT, 'PDGETRF', -INFO )
         RETURN
      END IF
```

```
      IF( DESCA( M_ ).EQ.1 ) THEN
         IPIV( 1 ) = 1
         RETURN
      ELSE IF( M.EQ.0 .OR. N.EQ.0 ) THEN
         RETURN
      END IF
*     Split-ring topology for the communication along process rows
      CALL PB_TOPGET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
      CALL PB_TOPGET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
      CALL PB_TOPGET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', 'S-ring' )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', ' ' )
      CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', ' ' )
*     Handle the first block of columns separately
      MN = MIN( M, N )
      IN = MIN( ICEIL( IA, DESCA( MB_ ) )*DESCA( MB_ ), IA+M-1 )
      JN = MIN( ICEIL( JA, DESCA( NB_ ) )*DESCA( NB_ ), JA+MN-1 )
      JB = JN - JA + 1
*     Factor diagonal and subdiagonal blocks and test for exact
*     singularity.
      CALL PDGETF2( M, JB, A, IA, JA, DESCA, IPIV, INFO )
      IF( JB+1.LE.N ) THEN
*        Apply interchanges to columns JN+1:JA+N-1.
         CALL PDLASWP('Forward', 'Rows', N-JB, A, IA, JN+1, DESCA, IA, IN, IPIV )
*        Compute block row of U.
         CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
     $                N-JB, ONE, A, IA, JA, DESCA, A, IA, JN+1, DESCA )
*
         IF( JB+1.LE.M ) THEN
*           Update trailing submatrix.
            CALL PDGEMM( 'No transpose', 'No transpose', M-JB, N-JB, JB,
     $                   -ONE, A, IN+1, JA, DESCA, A, IA, JN+1, DESCA,
     $                   ONE, A, IN+1, JN+1, DESCA )
         END IF
      END IF
*     Loop over the remaining blocks of columns.
      DO 10 J = JN+1, JA+MN-1, DESCA( NB_ )
         JB = MIN( MN-J+JA, DESCA( NB_ ) )
         I = IA + J - JA
*
*        Factor diagonal and subdiagonal blocks and test for exact
*        singularity.
*
         CALL PDGETF2( M-J+JA, JB, A, I, J, DESCA, IPIV, IINFO )
*
         IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - JA
*
*        Apply interchanges to columns JA:J-JA.
*
         CALL PDLASWP('Forward', 'Rowwise', J-JA, A, IA, JA, DESCA, I,I+JB-1, IPIV)
         IF( J-JA+JB+1.LE.N ) THEN
*           Apply interchanges to columns J+JB:JA+N-1.
            CALL PDLASWP( 'Forward', 'Rowwise', N-J-JB+JA, A, IA, J+JB,
     $                    DESCA, I, I+JB-1, IPIV )
*           Compute block row of U.
            CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit', JB,
     $                   N-J-JB+JA, ONE, A, I, J, DESCA, A, I, J+JB,
```

```
     $                       DESCA )
             IF( J-JA+JB+1.LE.M ) THEN
*                Update trailing submatrix.
                 CALL PDGEMM( 'No transpose', 'No transpose', M-J-JB+JA,
     $                        N-J-JB+JA, JB, -ONE, A, I+JB, J, DESCA, A,
     $                        I, J+JB, DESCA, ONE, A, I+JB, J+JB, DESCA )
             END IF
          END IF
   10 CONTINUE
      IF( INFO.EQ.0 ) INFO = MN + 1
      CALL IGAMN2D(ICTXT, 'Rowwise', ' ', 1, 1, INFO, 1, IDUM1,IDUM2, -1,-1, MYCOL)
      IF( INFO.EQ.MN+1 ) INFO = 0
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Rowwise', ROWBTOP )
      CALL PB_TOPSET( ICTXT, 'Broadcast', 'Columnwise', COLBTOP )
      CALL PB_TOPSET( ICTXT, 'Combine', 'Columnwise', COLCTOP )
      RETURN
      END
```

In order to simplify the design of ScaLAPACK, and because the BLAS have proven to be very useful tools outside LAPACK, we chose to build a Parallel BLAS, or PBLAS (described in the paper by Choi et al; see "Further Reading"), whose interface is as similar to the BLAS as possible. This decision has permitted the ScaLAPACK code to be quite similar, and sometimes nearly identical, to the analogous LAPACK code.

It was our aim that the PBLAS would provide a distributed memory standard, just as the BLAS provided a shared memory standard. This would simplify and encourage the development of high-performance and portable parallel numerical software, as well as providing manufacturers with just a small set of routines to be optimized. The acceptance of the PBLAS requires reasonable compromises between competing goals of functionality and simplicity.

The PBLAS operate on matrices distributed in a two-dimensional block cyclic layout. Because such a data layout requires many parameters to fully describe the distributed matrix, we have chosen a more object-oriented approach and encapsulated these parameters in an integer array called an *array descriptor*. An array descriptor includes:

- The descriptor type
- The BLACS context (a virtual space for messages that is created to avoid collisions between logically distinct messages)
- The number of rows in the distributed matrix
- The number of columns in the distributed matrix
- The row block size
- The column block size
- The process row over which the first row of the matrix is distributed
- The process column over which the first column of the matrix is distributed
- The leading dimension of the local array storing the local blocks

By using this descriptor, a call to a PBLAS routine is very similar to a call to the corresponding BLAS routine:

```
      CALL DGEMM ( TRANSA, TRANSB, M, N, K, ALPHA,
                   A( IA, JA ), LDA,
                   B( IB, JB ), LDB, BETA,
```

```
                    C( IC, JC ), LDC )

        CALL PDGEMM( TRANSA, TRANSB, M, N, K, ALPHA,
                     A, IA, JA, DESC_A,
                     B, JB, DESC_B, BETA,
                     C, IC, JC, DESC_C )
```

DGEMM computes *C = BETA * C + ALPHA * op(A) * op(B),* where *op(A)* is either *A* or its transpose depending on *TRANSA*, *op(B)* is similar, *op(A)* is *M-by-K*, and *op(B)* is *K-by-N*. *PDGEMM* is the same, with the exception of the way submatrices are specified. To pass the submatrix starting at *A(IA,JA)* to *DGEMM*, for example, the actual argument corresponding to the formal argument *A* is simply *A(IA,JA)*. *PDGEMM*, on the other hand, needs to understand the global storage scheme of *A* to extract the correct submatrix, so *IA* and *JA* must be passed in separately.

DESC_A is the array descriptor for *A*. The parameters describing the matrix operands *B* and *C* are analogous to those describing *A*. In a truly object-oriented environment, matrices and *DESC_A* would be synonymous. However, this would require language support and detract from portability.

Using message passing and scalable algorithms from the ScaLAPACK library makes it possible to factor matrices of arbitrarily increasing size, given machines with more processors. By design, the library computes more than it communicates, so for the most part, data stays locally for processing and travels only occasionally across the interconnect network.

But the number and types of messages exchanged between processors can sometimes be hard to manage. The context associated with every distributed matrix lets implementations use separate "universes" for message passing. The use of separate communication contexts by distinct libraries (or distinct library invocations) such as the PBLAS insulates communication internal to the library from external communication. When more than one descriptor array is present in the argument list of a routine in the PBLAS, the individual BLACS context entries must be equal. In other words, the PBLAS do not perform "inter-context" operations.

In the performance sense, ScaLAPACK did to LAPACK what LAPACK did to LINPACK: it broadened the range of hardware where LU factorization (and other codes) could run efficiently. In terms of code elegance, the ScaLAPACK's changes were much more drastic: the same mathematical operation now required large amounts of tedious work. Both the users and the library writers were now forced into explicitly controlling data storage intricacies, because data locality became paramount for performance. The victim was the readability of the code, despite efforts to modularize the code according to the best software engineering practices of the day.

Multithreading for Multi-core Systems

The advent of multi-core chips brought about a fundamental shift in the way software is produced. Dense linear algebra is no exception. The good news is that LAPACK's LU factorization runs on a multi-core system and can even deliver a modest increase of performance if multithreaded BLAS are used. In technical terms, this is the fork-join model of computation: each call to BLAS (from a single main thread) forks a suitable number of threads, which perform the work on each core and then join the main thread of computation. The fork-join model implies a synchronization point at each join operation.

The bad news is that the LAPACK's fork-join algorithm gravely impairs scalability even on small multi-core computers that do not have the memory systems available in SMP systems. The inherent scalability flaw is the heavy synchronization in the fork-join model (only a single thread is allowed to perform the significant computation that occupies the critical section of the code, leaving other threads idle) that results in lock-step execution and prevents hiding of inherently sequential portions of the code behind parallel ones. In other words, the threads are forced to perform the same operation on different data. If there is not enough data for some threads, they will have to stay idle and wait for the rest of the threads that perform useful work on their data. Clearly, another version of the LU algorithm is needed such that would allow threads to stay busy all the time by possibly making them perform different operations during some portion of the execution.

The multithreaded version of the algorithm recognizes the existence of a so-called *critical path* in the algorithm: a portion of the code whose execution depends on previous calculations and can block the progress of the algorithm. The LAPACK's LU does not treat this critical portion of the code in any special way: the DGETF2 subroutine is called by a single thread and doesn't allow much parallelization even at the BLAS level. While one thread calls this routine, the other ones wait idly. And since the performance of DGETF2 is bound by memory bandwidth (rather than processor speed), this bottleneck will exacerbate scalability problems as systems with more cores are introduced.

The multithreaded version of the algorithm attacks this problem head-on by introducing the notion of look-ahead: calculating things ahead of time to avoid potential stagnation in the progress of the computations. This of course requires additional synchronization and bookkeeping not present in the previous versions—a trade-off between code complexity and performance. Another aspect of the multithreaded code is the use of recursion in the panel factorization. It turns out that the use of recursion can give even greater performance benefits for tall panel matrices than it does for the square ones

Example 14-6 shows a factorization suitable for multithreaded execution.

Example 14-6. Factorization for multithreaded execution (C code)

```
void SMP_dgetrf(int n, double *a, int lda, int *ipiv, int pw,
                         int tid, int tsize, int *pready,ptm *mtx, ptc *cnd) {
  int pcnt, pfctr, ufrom, uto, ifrom, p;
  double *pa = a, *pl, *pf, *lp;

  pcnt = n / pw; /* number of panels */

  pfctr = tid + (tid ? 0 : tsize); /* first panel that should be factored by this
                       thread after the very first panel (number 0) gets factored
*/

  /* this is a pointer to the last panel */
  lp = a + (size_t)(n - pw) * (size_t)lda;

  /* for each panel (that is used as source of updates) */
  for (ufrom = 0; ufrom < pcnt; ufrom++, pa += (size_t)pw * (size_t)(lda + 1)){
    p = ufrom * pw; /* column number */

    /* if the panel to be used for updates has not been factored yet; 'ipiv'
       does not be consulted, but it is to possibly avoid accesses to 'pready'*/
    if (! ipiv[p + pw - 1] || ! pready[ufrom]) {

      if (ufrom % tsize == tid) { /* if this is this thread's panel */
```

```
        pfactor( n – p, pw, pa, lda, ipiv + p, pready, ufrom, mtx, cnd );
      } else if (ufrom < pcnt – 1) { /* if this is not the last panel */
        LOCK( mtx );
        while (! pready[ufrom]) { WAIT( cnd, mtx ); }
        UNLOCK( mtx );
      }
    }
    /* for each panel to be updated */
    for (uto = first_panel_to_update( ufrom, tid, tsize ); uto < pcnt;
         uto += tsize) {
      /* if there are still panels to factor by this thread and preceding panel
         has been factored; test to 'ipiv' could be skipped but is in there to
         decrease number of accesses to 'pready' */
      if (pfctr < pcnt && ipiv[pfctr * pw – 1] && pready[pfctr – 1]) {
        /* for each panel that has to (still) update panel 'pfctr' */
        for (ifrom = ufrom + (uto > pfctr ? 1 : 0); ifrom < pfctr; ifrom++) {
          p = ifrom * pw;
          pl = a + (size_t)p * (size_t)(lda + 1);
          pf = pl + (size_t)(pfctr – ifrom) * (size_t)pw * (size_t)lda;
          pupdate( n – p, pw, pl, pf, lda, p, ipiv, lp );
        }
        p = pfctr * pw;
        pl = a + (size_t)p * (size_t)(lda + 1);
        pfactor( n – p, pw, pl, lda, ipiv + p, pready, pfctr, mtx, cnd );
        pfctr += tsize; /* move to this thread's next panel */
      }

      /* if panel 'uto' hasn't been factored (if it was, it certainly has been
         updated, so no update is necessary) */
      if (uto > pfctr || ! ipiv[uto * pw]) {
        p = ufrom * pw;
        pf = pa + (size_t)(uto – ufrom) * (size_t)pw * (size_t)lda;
        pupdate( n – p, pw, pa, pf, lda, p, ipiv, lp );
      }
    }
  }
}
```

The algorithm is the same for each thread (the SIMD paradigm), and the matrix data is partitioned among threads in a cyclic manner using panels with pw columns in each panel (except maybe the last). The pw parameter corresponds to the blocking parameter NB of LAPACK. The difference is the logical assignment of panels (blocks of columns) to threads. (Physically, all panels are equally accessible, because the code operates in a shared memory regimen.) The benefits of blocking in a thread are the same as they were in LAPACK: better cache reuse and less stress on the memory bus. Assigning a portion of the matrix to a thread seems an artificial requirement at first, but it simplifies the code and the bookkeeping data structures; most importantly, it provides better memory affinity. It turns out that multi-core chips are not symmetric in terms of memory access bandwidth, so minimizing the number of reassignments of memory pages to cores directly benefits performance.

The standard components of LU factorization are represented by the pfactor() and pupdate() functions. As one might expect, the former factors a panel, whereas the latter updates a panel using one of the previously factored panels.

The main loop makes each thread iterate over each panel in turn. If necessary, the panel is factored by the owner thread while other threads wait (if they happen to need this panel for their updates).

The look-ahead logic is inside the nested loop (prefaced by the comment `for each panel to be updated`) that replaces DGEMM or PDGEMM from previous algorithms. Before each thread updates one of its panels, it checks whether it's already feasible to factor its first unfactored panel. This minimizes the number of times the threads have to wait because each thread constantly attempts to eliminate the potential bottleneck.

As was the case for ScaLAPACK, the multithreaded version detracts from the inherent elegance of the LAPACK's version. Also in the same spirit, performance is the main culprit: LAPACK's code will not run efficiently on machines with ever-increasing numbers of cores. Explicit control of execution threads at the LAPACK level rather than the BLAS level is critical: parallelism cannot be encapsulated in a library call. The only good news is that the code is not as complicated as ScaLAPACK's, and efficient BLAS can still be put to a good use.

A Word About the Error Analysis and Operation Count

The key aspect of all of the implementations presented in this chapter is their numerical properties.

It is acceptable to forgo elegance in order to gain performance. But numerical stability is of vital importance and cannot be sacrificed, because it is an inherent part of the algorithm's correctness. While these are serious considerations, there is some consolation to follow. It may be surprising to some readers that all of the algorithms presented are the same, even though it's virtually impossible to make each excerpt of code produce exactly the same output for exactly the same inputs.

When it comes to repeatability of results, the vagaries of floating-point representation may be captured in a rigorous way by *error bounds*. One way of expressing the numerical robustness of the previous algorithms is with the following formula:

$$||r|| / ||A|| \le ||e|| \le ||A^{-1}|| \; ||r||$$

where error $e = x - y$ is the difference between the computed solution y and the correct solution x, and $r = Ay - b$ is a so-called "residual." The previous formula basically says that the size of the error (the parallel bars surrounding a value indicate a norm—a measure of absolute size) is as small as warranted by the quality of the matrix A. Therefore, if the matrix is close to being singular in numerical sense (some entries are so small that they might as well be considered to be zero) the algorithms will not give an accurate answer. But otherwise, a relatively good quality of result can be expected.

Another feature that is common to all the versions presented is the operation count: they all perform $2/3n^3$ floating-point multiplications and/or additions. The order of these operations is what differentiates them. There exist algorithms that increase the amount of floating-point work to save on memory traffic or network transfers (especially for distribute-memory parallel algorithms.) But because the algorithms shown in this chapter have the same operation count, it is valid to compare them for performance. The computational rate (number of floating-point operations per second) may be used instead

of the time taken to solve the problem, provided that the matrix size is the same. But comparing computational rates is sometimes better because it allows a comparison of algorithms when the matrix sizes differ. For example, a sequential algorithm on a single processor can be directly compared with a parallel one working on a large cluster on a much bigger matrix.

Future Directions for Research

In this chapter we have looked at the evolution of the design of a simple but important algorithm in computational science. The changes over the past 30 years have been necessary to follow the lead of the advances in computer architectures. In some cases these changes have been simple, such as interchanging loops. In other cases, they have been as complex as the introduction of recursion and look-ahead computations. In each case, however, the code's ability to efficiently utilize the memory hierarchy is the key to high performance on a single processor as well as on shared and distributed memory systems.

The essence of the problem is the dramatic increase in complexity that software developers have had to confront, and still do. Dual-core machines are already common, and the number of cores is expected to roughly double with each processor generation. But contrary to the assumptions of the old model, programmers will not be able to consider these cores independently (i.e., multi-core is *not* "the new SMP") because they share on-chip resources in ways that separate processors do not. This situation is made even more complicated by the other nonstandard components that future architectures are expected to deploy, including the mixing of different types of cores, hardware accelerators, and memory systems.

Finally, the proliferation of widely divergent design ideas shows that the question of how to best combine all these new resources and components is largely unsettled. When combined, these changes produce a picture of a future in which programmers will have to overcome software design problems vastly more complex and challenging than those in the past in order to take advantage of the much higher degrees of concurrency and greater computing power that new architectures will offer.

So the bad news is that none of the presented code will work efficiently someday. The good news is that we have learned various ways to mold the original simple rendition of the algorithm to meet the ever-increasing challenges of hardware designs.

Further Reading

- *LINPACK User's Guide*, J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, SIAM: Philadelphia, 1979, ISBN 0-89871-172-X.

- *LAPACK Users' Guide, 3rd Edition*, E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammaring, A. McKenney, and D. Sorensen, SIAM: Philadelphia, 1999, ISBN 0-89871-447-8.

- *ScaLAPACK Users' Guide*, L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, SIAM Publications, Philadelphia, 1997, ISBN 0-89871-397-8.

- "Basic Linear Algebra Subprograms for FORTRAN usage," C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh, ACM Trans. Math. Soft., Vol. 5, pp. 308—323, 1979.

- "An extended set of FORTRAN Basic Linear Algebra Subprograms," J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, ACM Trans. Math. Soft., Vol. 14,, pp. 1-17, 1988.

- "A set of Level 3 Basic Linear Algebra Subprograms," J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, ACM Trans. Math. Soft., Vol. 16, pp. 1—17, 1990.

- *Implementation Guide for LAPACK*, E. Anderson and J. Dongarra, UT-CS-90-101, April 1990.

- *A Proposal for a Set of Parallel Basic Linear Algebra Subprograms*, J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley, UT-CS-95-292, May 1995.

- *LAPACK Working Note 37: Two Dimensional Basic Linear Algebra Communication Subprograms, J. Dongarra and R. A. van de Geijn,* University of Tennessee Computer Science Technical Report, UT-CS-91-138, October 1991.

- "Matrix computations with Fortran and paging," Cleve B. Moler, *Communications of the ACM*, 15(4), 1972, pp. 268-270.

- *LAPACK Working Note 19: Evaluating Block Algorithm Variants in LAPACK*, E. Anderson and J. Dongarra, University of Tennessee Computer Science Technical Report, UT-CS-90-103, April 1990.

- "Recursion leads to automatic variable blocking for dense linear-algebra algorithms," Gustavson, F. G. *IBM J. Res. Dev.* Vol. 41, No. 6 (Nov. 1997), pp. 737-756.