Parallel and Distributed Computing Handbook

Albert Y. Zomaya, Editor

(4)

McGraw-Hill

New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

32

Constructing Numerical Software Libraries for High-Performance Computer Environments

Jack J. Dongarra and David W. Walker

This chapter discusses the design of linear algebra libraries for high-performance computers. Particular emphasis is placed on the development of scalable algorithms for MIMD distributed memory concurrent computers. A brief description of the EISPACK, LINPACK, and LAPACK libraries is given, followed by an outline of ScaLAPACK, which is a distributed memory version of LAPACK currently under development. The importance of block-partitioned algorithms in reducing the frequency of data movement between various levels of hierarchical memory is stressed. The use of such algorithms helps reduce the message start-up costs on distributed memory concurrent computers. Other key ideas in our approach are:

1. the use of distributed versions of the Level 3 Basic Linear Algebra Subprograms (BLAS) as computational building blocks
2. the use of Basic Linear Algebra Communication Subprograms (BLACS) as communication building blocks

Together, the distributed BLAS and the BLACS can be used to construct higher-level algorithms and to hide many details of the parallelism from the application developer. The block-cyclic data distribution is described and is adopted as a good way of distributing block partitioned matrices. Block-partitioned versions of the Cholesky and LU factorizations are presented, and optimization issues associated with the implementation of the LU factorization algorithm on distributed memory concurrent computers are discussed, together with its performance on the Intel Delta system. Finally, approaches to the design of library interfaces are reviewed.

32.1 Introduction

The increasing availability of advanced-architecture computers is having a very significant effect on all spheres of scientific computation, including algorithm research and soft-

ware development in numerical linear algebra. Linear algebra—in particular, the solution of linear systems of equations—lies at the heart of most calculations in scientific computing. This chapter discusses some of the recent developments in linear algebra designed to exploit these advanced-architecture computers. Particular attention will be paid to dense factorization routines such as the Cholesky and LU factorizations. These will be used as examples to 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, as discussed in more detail in Section 32.4.1.

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 focus on four basic issues:

1. the motivation for the work
2. the development of standards for use in linear algebra and the building blocks for a library
3. aspects of algorithm design and parallel implementation
4. future directions for research

For the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The linear algebra community has long recognized the need for help in developing algorithms into software libraries, and several years ago, as a community effort, put together a *de facto* standard for identifying basic operations required in linear algebra algorithms and software. The hope was that the routines making up this standard, known collectively as the Basic Linear Algebra Subprograms (BLAS), would be efficiently implemented on advanced-architecture computers by many manufacturers, making it possible to reap the portability benefits of having them efficiently implemented on a wide range of machines. This goal has been largely realized.

The key insight of our approach to designing linear algebra algorithms for advanced architecture computers is that the frequency with which data are moved between different levels of the memory hierarchy must be minimized in order to attain high performance. Thus, our main algorithmic approach for exploiting both vectorization and parallelism in our implementations is the use of block-partitioned algorithms, particularly in conjunction with highly-tuned kernels for performing matrix-vector and matrix-matrix operations (the Level 2 and 3 BLAS). In general, the use of block-partitioned algorithms requires data to be moved as blocks, rather than as vectors or scalars, so that although the total amount of data moved is unchanged, the latency (or startup cost) associated with the moment is greatly reduced because fewer messages are needed to move the data.

A second key idea is that the performance of an algorithm can be tuned by a user by varying the parameters that specify the data layout. On shared memory machines, this is controlled by the block size, while on distributed memory machines it is controlled by the block size and the configuration of the logical process mesh, as described in more detail in Section 32.5.

Section 32.1 gives an overview of some of the major software projects aimed at solving dense linear algebra problems. It then describes the types of machines that benefit most from the use of block-partitioned algorithms, and discusses what is meant by high-quality, reusable software for advanced-architecture computers. Section 32.2 discusses the role of

the BLAS in portability and performance on high-performance computers. The design of these building blocks, and their use in block-partitioned algorithms, are covered in Section 32.3. Section 32.4 focuses on the design of a block-partitioned algorithm for LU factorization, and Sections 32.5, 32.6, and 32.7 use this example to illustrate the most important factors in implementing dense linear algebra routines on MIMD distributed memory concurrent computers. Section 32.5 deals with the issue of mapping the data onto the hierarchical memory of a concurrent computer. The layout of an application's data is crucial in determining the performance and scalability of the parallel code. In Sections 32.6 and 32.7, details of the parallel implementation and optimization issues are discussed. Section 32.8 presents some future directions for investigation.

32.1.1 Dense linear algebra libraries

Over the past 25 years, the first author has been directly involved in the development of several important packages of dense linear algebra software: EISPACK, LINPACK, LAPACK, and the BLAS. In addition, both authors are currently involved in the development of ScaLAPACK, a scalable version of LAPACK for distributed memory concurrent computers. In this section, we give a brief review of these packages—their history, their advantages, and their limitations on high-performance computers.

32.1.1.1 EISPACK. EISPACK is a collection of Fortran subroutines that compute the eigenvalues and eigenvectors of nine classes of matrices: complex general, complex Hermitian, real general, real symmetric, real symmetric banded, real symmetric tridiagonal, special real tridiagonal, generalized real, and generalized real symmetric matrices. In addition, two routines are included that use singular value decomposition to solve certain least-squares problems.

EISPACK is primarily based on a collection of Algol procedures developed in the 1960s and collected by J. H. Wilkinson and C. Reinsch in a volume entitled in the *Handbook for Automatic Computation* [57] series. This volume was not designed to cover every possible method of solution; rather, algorithms were chosen on the basis of their generality, elegance, accuracy, speed, or economy of storage.

Since the release of EISPACK in 1972, over ten thousand copies of the collection have been distributed worldwide.

32.1.1.2 LINPACK. 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 is organized around four matrix factorizations: LU factorization, pivoted Cholesky factorization, QR factorization, and singular value decomposition. The term *LU factorization* is used here in a very general sense to mean the factorization of a square matrix into a lower triangular part and an upper triangular part, perhaps with pivoting. These factorizations will be treated at greater length later, when the actual LINPACK subroutines are discussed. But first a digression on organization and factors influencing LINPACK's efficiency is necessary.

LINPACK uses column-oriented algorithms to increase efficiency by preserving locality of reference. This means that if a program references an item in a particular block, the next

reference is likely to be in the same block. By column orientation we mean that the LINPACK codes always reference arrays down columns, not across rows. This works because Fortran stores arrays in column major order. Thus, as one proceeds down a column of an array, the memory references proceed sequentially in memory. On the other hand, as one proceeds across a row, the memory references jump across memory, the length of the jump being proportional to the length of a column. The effects of column orientation are quite dramatic: on systems with virtual or cache memories, the LINPACK codes will significantly outperform codes that are not column oriented. We note, however, that textbook examples of matrix algorithms are seldom column oriented.

Another important factor influencing the efficiency of LINPACK is the use of the Level 1 BLAS; there are three effects.

First, the overhead entailed in calling the BLAS reduces the efficiency of the code. This reduction is negligible for large matrices, but it can be quite significant for small matrices. The matrix size at which it becomes unimportant varies from system to system; for square matrices it is typically between $n = 25$ and $n = 100$. If this seems like an unacceptably large overhead, remember that on many modern systems the solution of a system of order 25 or less is itself a negligible calculation. Nonetheless, it cannot be denied that a person whose programs depend critically on solving small matrix problems in inner loops will be better off with BLAS-less versions of the LINPACK codes. Fortunately, the BLAS can be removed from the smaller, more frequently used program in a short editing session.

Second, the BLAS improve the efficiency of programs when they are run on nonoptomizing compilers. This is because doubly subscripted array references in the inner loop of the algorithm are replaced by singly subscripted array references in the appropriate BLAS. The effect can be seen for matrices of quite small order, and for large orders the savings are quite significant.

Finally, improved efficiency can be achieved by coding a set of BLAS [17] to take advantage of the special features of the computers on which LINPACK is being run. For most computers, this simply means producing machine-language versions. However, the code can also take advantage of more exotic architectural features, such as vector operations. Further details about the BLAS are presented in Section 32.2.

32.1.1.3 LAPACK. LAPACK [14] provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering the Schur factorizations and estimating condition numbers. Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

The original goal of the LAPACK project was to make the widely used EISPACK and LINPACK libraries run efficiently on shared-memory vector and parallel processors. On these machines, LINPACK and EISPACK are inefficient because their 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 [3, 14]. These block operations can be optimized for each architecture to account for the memory hierarchy [2], 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 already implemented 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 and EISPACK. It has virtually all the capabilities of these two packages and much more. LAPACK improves on LINPACK and EISPACK in four main respects: speed, accuracy, robustness and functionality. While LINPACK and EISPACK are based on the vector operation kernels of the Level 1 BLAS, LAPACK was designed at the outset to exploit the Level 3 BLAS—a set of specifications for Fortran subprograms that do various types of matrix multiplication and the solution of triangular systems with multiple right-hand sides. Because of the course granularity of the Level 3 BLAS operations, their use tends to promote high efficiency on many high-performance computers, particularly if specially coded implementations are provided by the manufacturer.

32.1.1.4 ScaLAPACK. The ScaLAPACK software library, scheduled for completion by the end of 1994, will extend the LAPACK library to run scalably on MIMD distributed memory concurrent computers [10, 11]. 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 among various 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) [16, 26] 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.

We envisage a number of user interfaces to ScaLAPACK. Initially, the interface will be similar to that of LAPACK, with some additional arguments passed to each routine to specify the data layout. Once this is in place, we intend to modify the interface so the arguments to each ScaLAPACK routine are the same as in LAPACK. This will require information about the data distribution of each matrix and vector to be hidden from the user. This may be done by means of a ScaLAPACK initialization routine. This interface will be fully compatible with LAPACK. Provided "dummy" versions of the ScaLAPACK initialization routine and the BLACS are added to LAPACK, there will be no distinction between LAPACK and ScaLAPACK at the application level, though each will link to various versions of the BLAS and BLACS. Following this, we will experiment with object-based interfaces for LAPACK and ScaLAPACK, with the goal of developing interfaces compatible with Fortran 90 [10] and C++ [24].

32.1.2 Target architecture

The EISPACK and LINPACK software libraries were designed for supercomputers used in the 1970s and early 1980s, such as the CDC-7600, Cyber 205, and Cray-1. These machines featured multiple functional units pipelined for good performance [43]. The CDC-7600 was basically a high-performance scalar computer, while the Cyber 205 and Cray-1 were early vector computers.

The development of LAPACK in the late 1980s was intended to make the EISPACK and LINPACK libraries run efficiently on shared memory vector supercomputers. The ScaLA-

PACK software library will extend the use of LAPACK to distributed memory concurrent supercomputers. The development of ScaLAPACK began in 1991 and was scheduled to be completed by the end of 1994.

The underlying concept of both the LAPACK and ScaLAPACK libraries is the use of block partitioned algorithms to minimize data movement between various levels in hierarchical memory. Thus, the ideas discussed in this chapter for developing a library for dense linear algebra computations are applicable to any computer with a hierarchical memory that (1) imposes a sufficiently large start-up cost on the movement of data among various levels in the hierarchy, and for which (2) the cost of a context switch is too great to make fine grain size multithreading worthwhile. Our target machines are, therefore, medium and large grain size advanced-architecture computers. These include "traditional" shared memory vector supercomputers, such as the Cray Y-MP and C90, and MIMD distributed memory concurrent supercomputers, such as the Intel Paragon and Thinking Machines' CM-5, and the more recently announced IBM SP1 and Cray T3D concurrent systems. Since these machines have only very recently become available, most of the ongoing development of the ScaLAPACK library is being done on a 128-node Intel iPSC/860 hypercube and on the 520-node Intel Delta system.

The Intel Paragon supercomputer can have up to 2000 nodes, each consisting of an i860 processor and a communications processor. The nodes each have at least 16 MB of memory, and are connected by a high-speed network with the topology of a two-dimensional mesh. The CM-5 from Thinking Machines Corporation [53] supports both SIMD and MIMD programming models, and may have up to 16k processors, though the largest CM-5 currently installed has 1024 processors. Each CM-5 node is a Sparc processor and up to 4 associated vector processors. Point-to-point communication between nodes is supported by a data network with the topology of a "fat tree" [46]. Global communication operations, such as synchronization and reduction, are supported by a separate control network. The IBM SP1 system is based on the same RISC chip used in the IBM RS/6000 workstations and uses a multistage switch to connect processors. The Cray T3D uses the Alpha chip from Digital Equipment Corporation, and connects the processors in a three-dimensional torus.

Future advances in compiler and hardware technologies in the mid to late 1990s are expected to make multithreading a viable approach for masking communication costs. Since the blocks in a block-partitioned algorithm can be regarded as separate threads, our approach will still be applicable on machines that exploit medium and coarse grain size multithreading.

32.1.3 High-quality, reusable, mathematical software

In developing a library of high-quality subroutines for dense linear algebra computations the design goals fall into three broad classes:

- performance
- ease-of-use
- range-of-use

32.1.3.1 Performance. Two important performance metrics are concurrent efficiency and scalability. We seek good performance characteristics in our algorithms by eliminating, as much as possible, overhead that is due to load imbalance, data movement, and algorithm restructuring. The way the data are distributed (or decomposed) over the memory hierar-

chy of a computer is of fundamental importance to these factors. Concurrent efficiency, ε, defined as the concurrent speedup per processor [32], where the concurrent speedup is the execution time, T_{seq}, for the best sequential algorithm running on one processor of the concurrent computer, divided by the execution time, T, of the parallel algorithm running on N_p processors. When direct methods are used, as in LU factorization, the concurrent efficiency depends on the problem size and the number of processors, so on a given parallel computer and for a fixed number of processors, the running time should not vary greatly for problems of the same size. Thus, we may write

$$\varepsilon(N, N_p) = \frac{1}{N_p} \times \frac{T_{seq}(N)}{N_p T(N, N_p)} \tag{32.1}$$

where N represents the problem size. In dense linear algebra computations, the execution time is usually dominated by the floating-point operation count, so the concurrent efficiency is related to performance, G, measured in floating-point operations per second by

$$G(N, N_p) = \frac{N_p}{t_{calc}} \times \varepsilon(N, N_p) \tag{32.2}$$

where t_{calc} is the time for one floating-point operation. For iterative routines, such as eigensolvers, the number of iterations, and hence the execution time, depends not only on the problem size, but also on other characteristics of the input data, such as condition number. A parallel algorithm is said to be scalable [37] if the concurrent efficiency depends on the problem size and number of processors only through their ratio. This ratio is simply the problem size per processor, often referred to as the *granularity*. Thus, for a scalable algorithm, the concurrent efficiency is constant as the number of processors increases while keeping the granularity fixed. Alternatively, Eq. 32.2 shows that this is equivalent to saying that, for a scalable algorithm, the performance depends linearly on the number of processors for fixed granularity.

32.1.3.2 Ease-of-use. Ease-of-use is concerned with factors such as portability and the user interface to the library. Portability, in its most inclusive sense, means that the code is written in a standard language, such as Fortran, and that the source code can be compiled on an arbitrary machine to produce a program that will run correctly. We call this the "mail-order software" model of portability, since it reflects the model used by software servers such as *netlib* [20]. This notion of portability is quite demanding. It requires that all relevant properties of the computer's arithmetic and architecture be discovered at run time within the confines of a Fortran code. For example, if it is important to know the overflow threshold for scaling purposes, it must be determined at run time *without overflowing*, since overflow is generally fatal. Such demands have resulted in quite large and sophisticated programs [28, 44] that must be modified frequently to deal with new architectures and software releases. This "mail-order" notion of software portability also means that codes generally must be written for the worst possible machine expected to be used, thereby often degrading performance on all others. Ease-of-use is also enhanced if implementation details are largely hidden from the user, for example, through the use of an object-based interface to the library [24].

32.1.3.3 Range-of-use. Range-of-use may be gauged by how numerically stable the algorithms are over a range of input problems, and the range of data structures the library will support. For example, LINPACK and EISPACK deal with dense matrices stored in a rectangular array, packed matrices where only the upper or lower half of a symmetric matrix is stored, and banded matrices where only the nonzero bands are stored. In addition, some special formats such as Householder vectors are used internally to represent orthogonal matrices. There are also sparse matrices, which may be stored in many different ways; but in this chapter we focus on dense and banded matrices, the mathematical types addressed by LINPACK, EISPACK, and LAPACK.

32.2 The BLAS as the Key to Portability

At least three factors affect the performance of portable Fortran code.

1. *Vectorization.* Designing vectorizable algorithms in linear algebra is usually straightforward. Indeed, for many computations there are several variants, all vectorizable, but with varied characteristics in performance. (See, for example, [15].) Linear algebra algorithms can approach the peak performance of many machines—principally because peak performance depends on some form of chaining of vector addition and multiplication operations, and this is just what the algorithms require. However, when the algorithms are realized in straightforward Fortran 77 code, the performance may fall well short of the expected level, usually because vectorizing Fortran compilers fail to minimize the number of memory references—that is, the number of vector load and store operations.
2. *Data movement.* What often limits the actual performance of a vector or scalar floating-point unit is the rate of transfer of data between various levels of memory in the machine. Examples include the transfer of vector operands in and out of vector registers, the transfer of scalar operands in and out of a high-speed scalar processor, the movements of data between main memory and a high-speed cache or local memory, paging between actual memory and disk storage in a virtual memory system, and interprocessor communication on a distributed memory concurrent computer.
3. *Parallelism.* The nested loop structure of most linear algebra algorithms offers considerable scope for loop-based parallelism. This is the principal type of parallelism that LAPACK and ScaLAPACK presently aim to exploit. On shared memory concurrent computers, this type of parallelism can sometimes be generated automatically by a compiler, but often requires the insertion of compiler directives. On distributed memory concurrent computers, data must be moved between processors. This is usually done by explicit calls to message passing routines, although parallel language extensions such as Coherent Parallel C [31] and Split-C [13] do the message passing implicitly.

The question arises, "How can we achieve sufficient control over these three factors to obtain the levels of performance that machines can offer?" The answer is through use of the BLAS.

There are now three levels of BLAS:

Level 1 BLAS [45]: for vector operations, such as $y \leftarrow \alpha x + y$
Level 2 BLAS [18]: for matrix-vector operations, such as $y \leftarrow \alpha Ax + \beta y$
Level 3 BLAS [17]: for matrix-matrix $\leftarrow$operations, such as $C \leftarrow \alpha AB + \beta C$.

Here, A, B and C are matrices, x and y are vectors, and α and β are scalars.

The Level 1 BLAS are used in LAPACK, but for convenience rather than for performance. They perform an insignificant fraction of the computation, and they cannot achieve high efficiency on most modern supercomputers.

The Level 2 BLAS can achieve near-peak performance on many vector processors, such as a single processor of a CRAY X-MP or Y-MP, or Convex C-2 machine. However, on other vector processors such as a CRAY-2 or an IBM 3090 VF, the performance of the Level 2 BLAS is limited by the rate of data movement among various levels of memory.

The Level 3 BLAS overcome this limitation. This third level of BLAS performs $O(n^3)$ floating point operations on $O(n^2)$ data, whereas the Level 2 BLAS perform only $O(n^2)$ operations on $O(n^2)$ data. The Level 3 BLAS also allow us to exploit parallelism in a way that is transparent to the software that calls them. While the Level 2 BLAS offer some scope for exploiting parallelism, greater scope is provided by the Level 3 BLAS, as Table 32.1 illustrates.

TABLE 32.1 Speed (Megaflops) of Level 2 and Level 3 BLAS Operations on a Cray Y-MP. (All matrices are of order 500; U is upper triangular.)

Number of processors:	1	2	4	8
Level 2: $y \leftarrow \alpha Ax + \beta y$	311	611	1197	2285
Level 3: $C \leftarrow \alpha AB + \beta C$	312	623	1247	2425
Level 2: $x \leftarrow Ux$	293	544	898	1613
Level 3: $B \leftarrow UB$	310	374	479	584
Level 2: $x \leftarrow U^{-1}x$	272	374	479	584
Level 3: $B \leftarrow U^{-1}B$	309	618	1235	2398

32.3 Block Algorithms and Their Derivation

It is comparatively straightforward to recode many of the algorithms in LINPACK and EISPACK so that they call Level 2 BLAS. Indeed, in the simplest cases the same floating-point operations are done, possibly even in the same order. It is just a matter of reorganizing the software. To illustrate this point, we consider the Cholesky factorization algorithm used in the LINPACK routine SPOFA, which factorizes a symmetric positive definite matrix as A = UTU. We consider Cholesky factorization because the algorithm is simple, and no pivoting is required. In Section 32.4 we shall consider the slightly more complicated example of LU factorization.

Suppose that after j–1 steps the block A_{oo} in the upper lefthand corner of A has been factored as $A_{oo} = U^T_{oo} U_{oo}$. The next row and column of the factorization can then be computed by writing $A = U^T U$ as

$$\begin{bmatrix} A_{oo} & b_j & A_{02} \\ \cdot & a_{jj} & c_j^T \\ \cdot & \cdot & A_{22} \end{bmatrix} = \begin{bmatrix} U_{oo}^T & 0 & 0 \\ v_j^T & u_{jj} & 0 \\ U_{02}^T & w_j & U_{22}^T \end{bmatrix} \begin{bmatrix} U_{oo} & v_j & U_{02} \\ 0 & u_{jj} & w_j^T \\ 0 & 0 & U_{22} \end{bmatrix}$$

where b_j, c_j, v_j, and w_j are column vectors of length j–1, and a_{jj} and u_{jj} are scalars. Equating coefficients of the j^{th} column, we obtain

$$b_j = U_{oo}^T v_j$$

$$a_{jj} = v_j^T v_j + u_{jj}^2$$

Since U_{oo} has already been computed, we can compute v_j and u_{jj} from the equations

$$U_{oo}^T v_j = b_j$$

$$u_{jj}^2 = a_{jj} - v_j^T v_j$$

The body of the code of the LINPACK routine SPOFA that implements the above method is shown in Fig. 32.1. The same computation recoded in "LAPACK-style" to use the Level 2 BLAS routine STRSV (which solves a triangular system of equations) is shown in Fig. 32.2. The call to STRSV has replaced the loop over K that made several calls to the Level 1 BLAS routine SDOT. (For reasons given below, this is not the actual code used in LAPACK—hence the term "LAPACK-style.")

This change by itself is sufficient to result in large gains in performance on a number of machines—for example, from 72 to 251 megaflops for a matrix of order 500 on one processor of a CRAY Y-MP. Since this is 81 percent of the peak speed of matrix-matrix multiplication on this processor, we cannot hope to do very much better by using Level 3 BLAS.

```
do j = 0, n-1
  info = j + 1
  s = 0.0e0
  jm1 = j
  if (jm1 .ge.  1) then
    do k = 0, jm1 - 1
      t = a(k,j) - sdot(k,a(0,k),1,a(0,j),1)
      t = t/a(k,k)
      a(k,j) = t
      s = s + t*t
    end do
  end if
  s = a(j,j) - s
  if (s .le.  0.0e0) go to 40
  a(j,j) = sqrt(s)
  end do
```

Figure 32.1 Body of the LINPACK routing SPOFA for Cholesky factorization

```
do j = 0, n - 1
  call strsv( 'upper', 'transpose', 'non-unit', j, a, lda,
a(0,j), 1 )
  s = a(j,j) - sdot( j, a(0,j), 1, a(0,j), 1 )
  if ( s .le.  zero ) go to 20
  a(j,j) = sqrt( s )
end do
```

Figure 32.2 Body of the "LINPACK-style" routing SPOFA for Cholesky factorization

We can, however, restructure the algorithm at a deeper level to exploit the faster speed of the Level 3 BLAS. This restructuring involves recasting the algorithm as a block algorithm—that is, an algorithm that operates on blocks or submatrices of the original matrix.

32.3.1 Deriving a block algorithm

To derive a block form of Cholesky factorization, we partition the matrices as shown in Fig. 32.3, in which the diagonal blocks of A and U are square, but of differing sizes. We assume that the first block has already been factored as $A_{00} = U_{00}^T U_{00}$, and that we now want to determine the second block column of U consisting of the blocks U_{01} and U_{11}. Equating submatrices in the second block of columns, we obtain

$$A_{01} = U_{00}^T U_{01}$$

$$A_{11} = U_{01}^T U_{01} = U_{11}^T U_{11}$$

Hence, since U_{00} has already been computed, we can compute U_{01} as the solution to the equation

$$U_{00}^T U_{01} = A_{01}$$

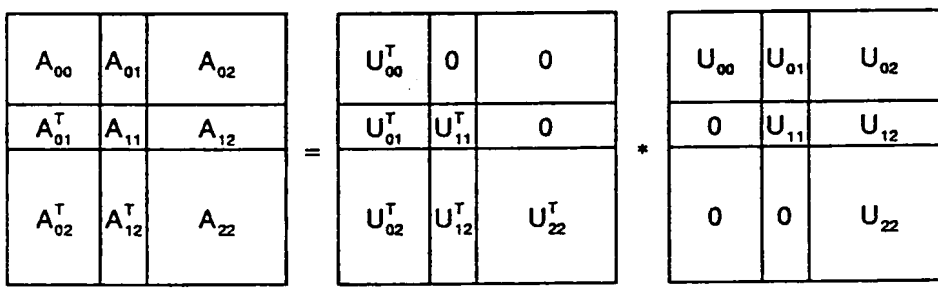

Figure 32.3 Partitioning of A, UT, and U into blocks. It is assumed that the first block has already been factored as $A_{00} = U_{00}^T U_{00}$, and we want to determine the block column consisting of U_{01} and U_{11}. Note that the diagonal blocks of A and U are square matrices.

by a call to the Level 3 BLAS routine STRSM; and then we can compute U_{11} from

$$U_{11}^T U_{11} = A_{11} - U_{01}^T U_{01}$$

This involves first updating the symmetric submatrix A_{11} by a call to the Level 3 BLAS routine SSYRK, and then computing its Cholesky factorization. Since Fortran does not allow recursion, a separate routine must be called (using Level 2 BLAS rather than Level 3), named SPOTF2 in Fig. 32.4. In this way, successive blocks of columns of U are computed. The LAPACK-style code for the block algorithm is shown in Fig. 32.4. This code runs at 49 megaflops on an IBM 3090, more than double the speed of the LINPACK code. On a CRAY Y-MP, the use of Level 3 BLAS squeezes a little more performance out of one processor, but makes a large improvement when using all eight processors.

But that is not the end of the story, and the code given above is not the code actually used in the LAPACK routine SPOTRF. We mentioned earlier that for many linear algebra computations there are several algorithmic variants, often referred to as *i*-, *j*-, and *k*-variants, according to a convention introduced in [15] and used in [36]. The same is true of the corresponding block algorithms.

It turns out that the j-variant chosen for LINPACK, and used in the above examples, is not the fastest on many machines, because it performs most of the work in solving triangular systems of equations, which can be significantly slower than matrix-matrix multiplication. The variant actually used in LAPACK is the *i*-variant, which relies on matrix-matrix multiplication for most of the work. Table 32.2 summarizes the results.

TABLE 32.2 Speed (Megaflops) of Cholesky Factorization $A = U^T U$ for $n = 500$

	IBM 3090 VF, 1 processor	Cray Y-MP, 1 processor	Cray Y-MP, 8 processors
j-variant: LINPACK	23	72	72
j-variant: using level 2 BLAS	24	251	378
j-variant: using level 3BLAS	49	287	1225
i-variant: using level 3BLAS	50	290	1414

```
do j = 0, n-1, nb
  jb = min( nb, n-j )
  call strsm( 'left', 'upper', 'transpose', 'non-unit', j, jb,
one,
              a, lda, a(0,j), lda )
  call ssyrk( 'upper', 'transpose', jb, j, -one, a(0,j), lda,
one,
              a(j,j), lda )
  call spotf2( 'upper', jb, a(j,j), lda, info )
  if( info .ne.  0 ) go to 20
end do
```

Figure 32.4 The body of the "LAPACK-style" routine SPOFA for block Cholesky factorization. In this code segment, nb denotes the width of the blocks.

32.3.2 Examples of block algorithms in LAPACK

Having discussed in detail the derivation of one particular block algorithm, we now describe examples of the performance achieved with two well known block algorithms: LU and Cholesky factorizations. No extra floating-point operations nor extra working storage is required for either of these simple block algorithms. (See Gallivan et al. [33] and Dongarra et al. [19] for surveys of algorithms for dense linear algebra on high-performance computers.)

Table 32.3 illustrates the speed of the LAPACK routine for LU factorization of a real matrix, SGETRF in single precision on CRAY machines, and DGETRF in double precision on all other machines. Thus, 64-bit floating-point arithmetic is used on all machines tested. A block size of 1 means that the unblocked algorithm is used, since it is faster than—or at least as fast as—a block algorithm.

TABLE 32.3 Speed (Megaflops) of SGETRF/DGETRF for Square Matrices of Order *n*

Machine	No. of processors	Block size	Values of *n*				
			100	200	300	400	500
IBM RISC/6000-530	1	32	19	25	29	31	33
Alliant FX/8	8	16	9	26	32	46	57
IBM 3090J VF	1	64	23	41	52	58	63
Convex C-240	4	64	31	60	82	100	112
Cray Y-MP	1	1	132	219	254	272	283
Cray-2	1	64	110	211	292	318	358
Siemens/Fujitsu VP 400-EX	1	64	46	132	222	309	397
NEC SX2	1	1	118	274	412	504	577
Cray Y-MP	8	64	195	556	920	1188	1408

LAPACK is designed to give high efficiency on vector processors, high-performance "superscalar" workstations, and shared memory multiprocessors. LAPACK in its present form is less likely to give good performance on other types of parallel architectures (for example, massively parallel SIMD machines or MIMD distributed memory machines), but the ScaLAPACK project, described in Section 32.1.1.4, is intended to adapt LAPACK to these new architectures. LAPACK can also be used satisfactorily on all types of scalar machines (PCs, workstations, mainframes). Table 32.4 gives similar results for Cholesky factorization, extending the results given in Table 32.2.

TABLE 32.4 Speed (Megaflops) of SPOTRF/DPOTRF for Matrices of Order *n*. Here, UPLO = "U," so the factorization is of the form $A = U^T U$.

Machine	No. of processors	Block size	Values of *n*				
			100	200	300	400	500
IBM RISC/6000-530	1	32	21	29	34	36	38
Alliant FX/8	8	16	10	27	40	49	52
IBM 3090J VF	1	48	26	43	56	62	67
Convex C-240	4	64	32	63	82	96	103
Cray Y-MP	1	1	126	219	257	275	285
Cray-2	1	64	109	213	294	318	362
Siemens/Fujitsu VP 400-EX	1	64	53	145	237	312	369
NEC SX2	1	1	155	387	589	719	819
Cray Y-MP	8	32	146	479	845	1164	1393

LAPACK, like LINPACK, provides LU and Cholesky factorizations of band matrices. The LINPACK, algorithms can easily be restructured to use Level 2 BLAS, though restructuring has little effect on performance for matrices of very narrow bandwidth. It is also possible to use Level 3 BLAS, at the price of doing some extra work with zero elements outside the band [22]. This process becomes worthwhile for large matrices and semi-bandwidth greater than 100 or so.

32.4 LU Factorization

In this section, we first discuss the uses of dense LU factorization in several fields. We next develop a block-partitioned version of the *k*, or right-looking, variant of the LU factorization algorithm. In subsequent sections, the parallelization of this algorithm is described in detail in order to highlight the issues and considerations that must be taken into account in developing an efficient, scalable, and transportable dense linear algebra library for MIMD distributed memory concurrent computers.

32.4.1 Uses of LU factorization in science and engineering

A major source of large dense linear systems is that of problems involving the solution of boundary integral equations. These are integral equations defined on the boundary of a region of interest. All examples of practical interest compute some intermediate quantity on a two-dimensional boundary and then use this information to compute the final desired quantity in three-dimensional space. The price one pays for replacing three dimensions with two is that what started as a sparse problem in $O(n^3)$ variables is replaced by a dense problem in $O(n^2)$.

Dense systems of linear equations are found in numerous applications, including:

- airplane wing design
- radar cross-section studies
- flow around ships and other off-shore constructions
- diffusion of solid bodies in a liquid
- noise reduction
- diffusion of light through small particles

The electromagnetics community is a major user of dense linear systems solvers. Of particular interest to this community is the solution of the so-called radar cross-section problem. In this problem, a signal of fixed frequency bounces off an object. The goal is to determine the intensity of the reflected signal in all possible directions. The underlying differential equation may vary, depending on the specific problem. In the design of stealth aircraft, the principal equation is the Helmholtz equation. To solve this equation, researchers use the *method of moments* [38, 56]. In the case of fluid flow, the problem often involves solving the Laplace or Poisson equation. Here, the boundary integral solution is known as the *panel method* [40, 41], so named from the quadrilaterals that discretize and approximate a structure such as an airplane. Generally, these methods are called *boundary element methods.*

Use of these methods produces a dense linear system of size $O(N)$ by $O(N)$, where N is the number of boundary points (or panels) being used. It is not unusual to see size $3N$ by $3N$, because of three physical quantities of interest at every boundary element.

A typical approach to solving such systems is to use LU factorization. Each entry of the matrix is computed as an interaction of two boundary elements. Often, many integrals

must be computed. In many instances, the time required to compute the matrix is considerably larger than the time for solution.

Only the builders of stealth technology who are interested in radar cross sections are considering using direct Gaussian elimination methods for solving dense linear systems. These systems are always symmetric and complex, but not Hermitian.

For further information on various methods for solving large dense linear algebra problems that arise in computational fluid dynamics, see the report by Edelman [30].

32.4.2 Derivation of a block algorithm for LU factorization

Suppose the $M \times N$ matrix A is partitioned as shown in Fig. 32.5, and we seek a factorization $A = LU$, where the partitioning of L and U is also shown in Fig. 32.5. Then we may write,

$$L_{00}U_{00} = A_{00} \tag{38.3}$$

$$L_{10}U_{00} = A_{10} \tag{38.4}$$

$$L_{00}U_{01} = A_{01} \tag{38.5}$$

$$L_{10}U_{01} + L_{11}U_{11} = A_{11} \tag{38.6}$$

where A_{00} is $r \times r$, A_{01} is $r \times (N-r)$, A_{10} is $(M-r) \times r$, and A_{11} is $(M-r) \times (N-r)$. L_{00} and L_{11} are lower triangular matrices with ones on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

Equations (32.3) and (32.4), taken together, perform an LU factorization on the first $M \times r$ panel of A (i.e., A_{00} and A_{10}). Once this is completed, the matrices, L_{00}, L_{10}, and U_{00}, are known, and the lower triangular system in Eq. (32.5) can be solved to give U_{01}. Finally, we rearrange Eq. (32.6) as

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \tag{32.7}$$

From this equation, we see that the problem of finding L_{11} and U_{11} reduces to finding the LU factorization of the $(M-r) \times (N-r)$ matrix A'_{11}. This can be done by applying the steps outlined above to A_{11} instead of to A. Repeating these steps K times, where

$$K = \min(\lceil M/r \rceil, \lceil N/r \rceil) \tag{32.8}$$

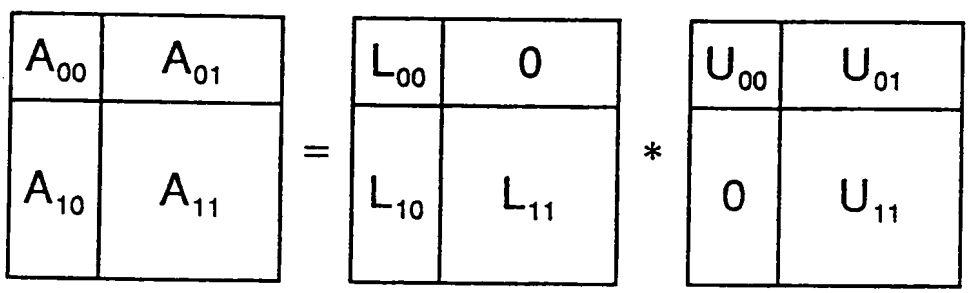

Figure 32.5 Block LU factorization of the partitioned matrix A. A_{00} is $r \times r$, A_{01} is $r \times (N-r)$, A_{10} is $(M-r) \times r$, and A_{11} is $(M-r) \times (N-r)$. L_{00} and L_{11} are lower triangular matrices with ones on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

we obtain the LU factorization of the original $M \times N$ matrix A. For an in-place algorithm, A is overwritten by L and U—the ones on the diagonal of L do not need to be stored explicitly. Similarly, when A is updated by Eq. (32.7), this may also be done in place.

After k of these K steps, the first kr columns of L and the first kr rows of U have been evaluated, and matrix A has been updated to the form shown in Fig. 32.6, in which panel B is $(M - kr) \times r$ and C is $r \times (N - (k-1)\, r)$. Step $k + 1$ then proceeds as follows:

1. Factor B to form the next panel of L, performing partial pivoting over rows if necessary. (See Fig. 32.14.) This evaluates the matrices L_0, L_1, and U_0 in Fig. 32.6.
2. Solve the triangular system $L_0 U_1 = C$ to get the next row of blocks of U.
3. Do a rank-r update on the trailing submatrix E, replacing it with $E' = E - L_1 U_1$.

The LAPACK implementation of this form of LU factorization uses the Level 3 BLAS routines xTRSM and xGEMM to perform the triangular solve and rank-r update. We can regard the algorithm as acting on matrices that have been partitioned into blocks of $r \times r$ elements, as shown in Fig. 32.7.

32.5 Data Distribution

The fundamental data object in the LU factorization algorithm presented in Section 32.4.2 is a block-partitioned matrix. In this section, we described the block-cyclic method for distributing such a matrix over a two-dimensional mesh of processes, or template. In general, each process has an independent thread of control, and with each process is associated some local memory directly accessible only by that process. The assignment of these processes to physical processors is a machine-dependent optimization issue, and will be considered later in Section 32.7.

An important property of the class of data distribution we shall use is that independent decompositions are applied over rows and columns. We shall, therefore, begin by considering the distribution of a vector of M data objects over P processes. This can be described by a mapping of the global index, of a data object to an index pair, (p,i), where p specifies the process to which the data object is assigned, and i specifies the location in the local memory of p at which it is stored. We shall assume $0 \le m < M$ and $0 \le p < P$.

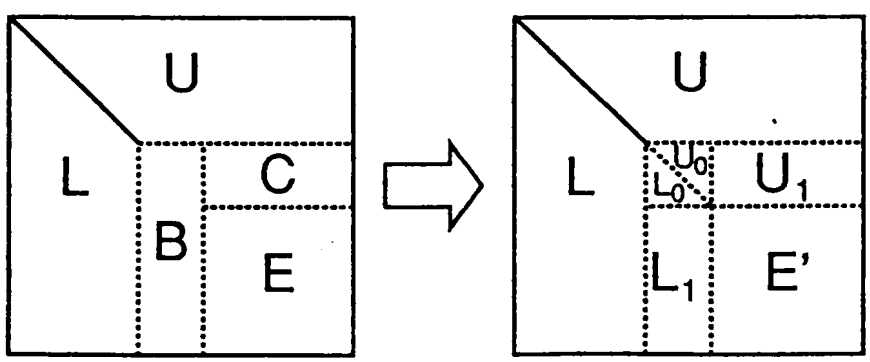

Figure 32.6 Stage $k + 1$ of the block LU factorization algorithm showing how the panels B and C, and the trailing submatrix E are updated. The trapezoidal submatrices L and U have already been factored in previous steps. L has kr columns, and U has kr rows. In the step shown another r columns of L and r rows of U are evaluated.