SOFTWARE LIBRARIES FOR LINEAR ALGEBRA COMPUTATIONS ON HIGH PERFORMANCE COMPUTERS

Jack J. Dongarra
David W. Walker

Abstract

This paper 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 different levels of hierarchical memory is stressed. The use of such algorithms helps reduce the message startup costs on distributed memory concurrent computers. Other key ideas in our approach are the use of distributed versions of the Level 3 Basic Linear Algebra Subprograms (BLAS) as computational building blocks, and 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 hide many details of the parallelism from the application developer.

The block-cyclic data distribution is described, and 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.

Key words: Parallel computing, linear algebra software libraries, LAPACK, ScaLAPACK

AMS subject classifications: 65Y05, 65Y10, 65F05

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 software 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 paper 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, and 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.

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 paper, 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; and (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 movement 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 5.

In Section 1, we first give an overview of some of the major software projects aimed at solving dense linear algebra problems. Next, we describe the types of machine that benefit most from the use of block-partitioned algorithms, and discuss what is meant by high-quality, reusable software for advanced-architecture computers. Section 2 discusses the role of the BLAS in portability and performance on high-performance computers. We discuss the design of these building blocks, and their use in block-partitioned algorithms, in Section 3. Section 4 focuses on the design of a block-partitioned algorithm for LU factorization, and Sections 5, 6, and 7 use this example to illustrate the most important factors in implementing dense linear algebra routines on MIMD, distributed memory, concurrent computers. Section 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 6 and 7, details of the parallel implementation and optimization issues are discussed. Section 8 presents some future directions for investigation.

1.1. Dense Linear Algebra Libraries

Over the past twenty-five 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.

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 *Linear Algebra* in the *Handbook for Automatic Computation* [51] series. Since the release of EISPACK in 1972, over ten thousand copies of the collection have been distributed worldwide.

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.

Another important influence on the efficiency of LINPACK is the use of the Level 1 BLAS. These BLAS are a small set of routines that may be coded 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 2.

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 of 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 besides. 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 coarse 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.

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 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) [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. We are also experimenting with object-based interfaces for LAPACK and ScaLAPACK, with the goal of developing interfaces compatible with Fortran 90 [10] and C++ [24].

1.2. Target Architectures

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 [40]. 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 ScaLAPACK software library will extend the use of LAPACK to distributed memory concurrent supercomputers. The development of ScaLAPACK began in 1991 and is expected 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 different levels in hierarchical memory. Thus, the ideas discussed in this paper for developing a library for dense linear algebra computations are applicable to any computer with a hierarchical memory that (1) imposes a sufficiently large startup cost on the movement of data between different 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.

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 handled by separate threads, our approach will still be applicable on machines that exploit medium and coarse grain size multithreading.

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 different 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 different 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 movement 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 [30] 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 [41]: for vector operations, such as $y \leftarrow \alpha x + y$

Level 2 BLAS [18]: for matrix-vector operations, such as $y \leftarrow \alpha A x + \beta y$

Level 3 BLAS [17]: for matrix-matrix operations, such as $C \leftarrow \alpha A B + \beta C$.

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

Table 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	620	1240	2425
Level 2: $x \leftarrow U^{-1}x$	272	374	479	584
Level 3: $B \leftarrow U^{-1}B$	309	618	1235	2398
Peak	333	666	1332	2664

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 between different levels of memory.[1]

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 1 illustrates.

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 $A = U^T U$. We consider Cholesky factorization because the algorithm is simple, and no pivoting is required. In Section 4 we shall consider the slightly more complicated example of LU factorization.

Suppose that after $j-1$ steps the block A_{00} in the upper lefthand corner of A has been factored as $A_{00} = U_{00}^T U_{00}$. The next row and column of the factorization can then be computed by writing $A = U^T U$ as

$$\begin{pmatrix} A_{00} & b_j & A_{02} \\ . & a_{jj} & c_j^T \\ . & . & A_{22} \end{pmatrix} = \begin{pmatrix} U_{00}^T & 0 & 0 \\ v_j^T & u_{jj} & 0 \\ U_{02}^T & w_j & U_{22}^T \end{pmatrix} \begin{pmatrix} U_{00} & v_j & U_{02} \\ 0 & u_{jj} & w_j^T \\ 0 & 0 & U_{22} \end{pmatrix}$$

[1]Machines such as the CRAY Y-MP can perform two loads, a store, and a multiply-add operation all in one cycle, whereas the CRAY-2 and IBM 3090 VF cannot. For further details of how the performance of the BLAS are affected by such factors see [19].

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

$$\begin{aligned} b_j &= U_{00}^T v_j \\ a_{jj} &= v_j^T v_j + u_{jj}^2. \end{aligned}$$

Since U_{00} has already been computed, we can compute v_j and u_{jj} from the equations

$$\begin{aligned} U_{00}^T v_j &= b_j \\ u_{jj}^2 &= a_{jj} - v_j^T v_j. \end{aligned}$$

The body of the code of the LINPACK routine SPOFA that implements the above method is shown in Figure 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 Figure 2. The call to STRSV has replaced the loop over K which 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% of the peak speed of matrix-matrix multiplication on this processor, we cannot hope to do very much better by using Level 3 BLAS.

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.

3.1. Deriving a Block Algorithm

To derive a block form of Cholesky factorization, we partition the matrices as shown in Figure 4, 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

$$\begin{aligned} A_{01} &= U_{00}^T U_{01} \\ A_{11} &= U_{01}^T U_{01} + U_{11}^T U_{11}. \end{aligned}$$

Table 2: Speed (Megaflops) of Cholesky Factorization $A = U^TU$ for $n = 500$

	IBM 3090 VF, 1 proc.	CRAY Y-MP, 1 proc.	CRAY Y-MP, 8 proc.
j-variant: LINPACK	23	72	72
j-variant: using Level 2 BLAS	24	251	378
j-variant: using Level 3 BLAS	49	287	1225

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

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

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 77 does not allow recursion, a separate routine must be called (using Level 2 BLAS rather than Level 3), named SPOTF2 in Figure 3. In this way, successive blocks of columns of U are computed. The LAPACK-style code for the block algorithm is shown in Figure 3. 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 8 processors. Table 2 summarizes the results.

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 are required for either of these simple block algorithms. (See Gallivan et al. [32] and Dongarra et al. [19] for surveys of algorithms for dense linear algebra on high-performance computers.)

Table 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. In all cases, results are reported for the block size which is mostly nearly optimal over the range of problem sizes considered.

LAPACK is designed to give high efficiency on vector processors, high-performance "su-

```
do j = 0, n-1
  info = j + 1
  s = 0.0e0
  jm1 = j
  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
  s = a(j,j) - s
  if (s .le.  0.0e0) go to 40
  a(j,j) = sqrt(s)
  end do
```

Figure 1: The body of the LINPACK routine 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 2: The body of the "LAPACK-style" routine SPOFA for Cholesky factorization.

```
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 3: The body of the "LAPACK-style" routine SPOFA for block Cholesky factorization. In this code fragment, `nb` denotes the width of the blocks.

$$
\left[\begin{array}{c|c|c}
A_{00} & A_{01} & A_{02} \\ \hline
A_{01}^T & A_{11} & A_{12} \\ \hline
A_{02}^T & A_{12}^T & A_{22}
\end{array}\right]
=
\left[\begin{array}{c|c|c}
U_{00}^T & 0 & 0 \\ \hline
U_{01}^T & U_{11}^T & 0 \\ \hline
U_{02}^T & U_{12}^T & U_{22}^T
\end{array}\right]
*
\left[\begin{array}{c|c|c}
U_{00} & U_{01} & U_{02} \\ \hline
0 & U_{11} & U_{12} \\ \hline
0 & 0 & U_{22}
\end{array}\right]
$$

Figure 4: Partitioning of A, U^T, 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 next 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.

Table 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

perscalar" 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 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 4 gives similar results for Cholesky factorization, extending the results given in Table 2.

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.

Table 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

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.

Dense matrix computations, such as LU factorization, have important applications, as discussed in a recent survey by Edelman [29]. A major source of large dense linear systems is the solution of problems by the boundary element method. In this method integral equations defined on the boundary of a region of interest are used to compute some final desired quantity in three-dimensional space. The dense linear systems generated are commonly solved using LU factorization. Electromagnetic scattering studies make use of the boundary element method, which is usually referred to as the method of moments in this context [35, 50]. This approach is used in the design of aircraft with small radar cross-sections, and in the design of satellite antennae. Boundary element methods are also used in the study of fluid flows, and here the variant of the boundary element method used is called the panel method [37, 38].

4.1. Derivation of a Block Algorithm for LU Factorization

Suppose the $M \times N$ matrix A is partitioned as shown in Figure 5, and we seek a factorization $A = LU$, where the partitioning of L and U is also shown in Figure 5. Then we may write,

$$L_{00}U_{00} \quad = \quad A_{00} \tag{1}$$

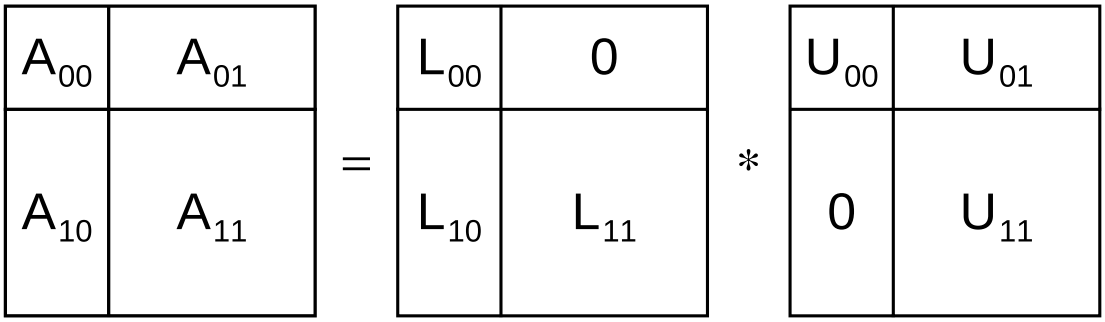


Figure 5: Block LU factorization of the partitioned matrix A. A_{00} is $r \times r$, A_{01} is $r \times (N-r)$, A_{10} is $(M-r) \times r$, and A_{11} is $(M-r) \times (N-r)$. L_{00} and L_{11} are lower triangular matrices with 1's on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

$$
\begin{aligned}
L_{10}U_{00} &= A_{10} && (2)\\
L_{00}U_{01} &= A_{01} && (3)\\
L_{10}U_{01} + L_{11}U_{11} &= A_{11} && (4)
\end{aligned}
$$

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 1s on the main diagonal, and U_{00} and U_{11} are upper triangular matrices.

Equations 1 and 2 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. 3 can be solved to give U_{01}. Finally, we rearrange Eq. 4 as,

$$A'_{11} = A_{11} - L_{10}U_{01} = L_{11}U_{11} \tag{5}$$

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

$$K = \min\left(\lceil M/r \rceil, \lceil N/r \rceil\right), \tag{6}$$

where $\lceil x \rceil$ denotes the least integer greater than or equal to x, 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 1s on the diagonal of L do not need to be stored explicitly. Similarly, when A is updated by Eq. 5 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

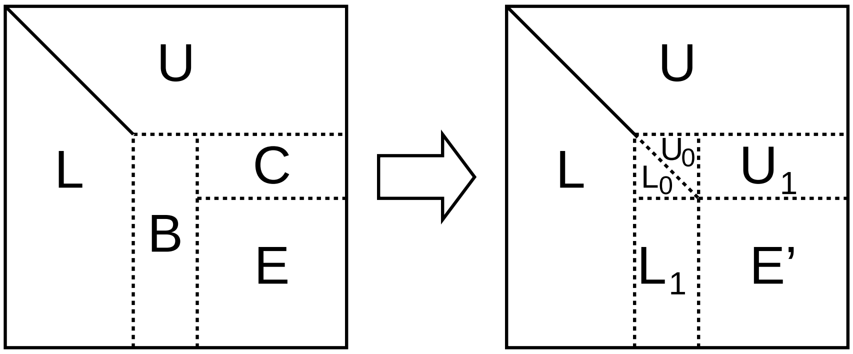


Figure 6: Stage $k+1$ of the block LU factorization algorithm showing how the panels B and C, and the trailing submatrix E are updated. The trapezoidal submatrices L and U have already been factored in previous steps. L has kr columns, and U has kr rows. In the step shown another r columns of L and r rows of U are evaluated.

evaluated, and matrix A has been updated to the form shown in Figure 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 Figure 14). This evaluates the matrices L_0, L_1, and U_0 in Figure 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 Figure 7.

5. Data Distribution

The fundamental data object in the LU factorization algorithm presented in Section 4.1 is a block-partitioned matrix. In this section, we describe 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 7.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	$A_{0,4}$	$A_{0,5}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$	$A_{1,4}$	$A_{1,5}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$	$A_{2,4}$	$A_{2,5}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,4}$	$A_{3,5}$
$A_{4,0}$	$A_{4,1}$	$A_{4,2}$	$A_{4,3}$	$A_{4,4}$	$A_{4,5}$
$A_{5,0}$	$A_{5,1}$	$A_{5,2}$	$A_{5,3}$	$A_{5,4}$	$A_{5,5}$

Figure 7: Block-partitioned matrix A. Each block $A_{i,j}$ consists of $r \times r$ matrix elements.

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, m, 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 $0 \leq m < M$ and $0 \leq p < P$.

Two common decompositions are the *block* and the *cyclic* decompositions [49, 31]. The block decomposition, which is often used when the computational load is distributed homogeneously over a regular data structure such as a Cartesian grid, assigns contiguous entries in the global vector to the processes in blocks.

$$m \mapsto (\lfloor m/L \rfloor, m \bmod L), \tag{7}$$

where $L = \lceil M/P \rceil$, and $\lfloor x \rfloor$ denotes the greatest integer less than or equal to x. The cyclic decomposition (also known as the wrapped or scattered decomposition) is commonly used to improve load balance when the computational load is distributed inhomogeneously over a regular data structure. The cyclic decomposition assigns consecutive entries in the global vector to successive different processes,

$$m \mapsto (m \bmod P, \lfloor m/P \rfloor) \tag{8}$$

Examples of the block and cyclic decompositions are shown in Figure 8.

m	0	1	2	3	4	5	6	7	8	9
p	0	0	0	0	1	1	1	1	2	2
i	0	1	2	3	0	1	2	3	0	1

(a) Block

m	0	1	2	3	4	5	6	7	8	9
p	0	1	2	0	1	2	0	1	2	0
i	0	0	0	1	1	1	2	2	2	3

(b) Cyclic

Figure 8: Examples of block and cyclic decompositions of $M = 10$ data objects over $P = 3$ processes.The global index m is mapped to the process number, p, and local index, i.

The block cyclic decomposition is a generalization of the block and cyclic decompositions in which blocks of consecutive data objects are distributed cyclically over the processes. In the block cyclic decomposition the mapping of the global index, m, can be expressed as $m \mapsto (p, b, i)$, where p is the process number, b is the block number in process p, and i is the index within block b to which m is mapped. Thus, if the number of data objects in a block is r, the block cyclic decomposition may be written,

$$m \mapsto \left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor, \left\lfloor \frac{m}{T} \right\rfloor, m \bmod r \right) \tag{9}$$

where $T = rP$. It should be noted that this reverts to the cyclic decomposition when $r = 1$, with local index $i = 0$ for all blocks. A block decomposition is recovered when $r = L$, in which case there is a single block in each process with block number $b = 0$. The inverse mapping of the triplet (p, b, i) to a global index is given by,

$$(p, b, i) \mapsto Br + i = pr + bT + i \tag{10}$$

where $B = p + bP$ is the global block number. The block cyclic decomposition is one of the data distributions supported by High Performance Fortran (HPF) [39], and has been previously used, in one form or another, by several researchers (see [1, 4, 5, 9, 23, 27, 45, 47, 48] for examples of its use). The block cyclic decomposition is illustrated with an example in Figure 9.

The form of the block cyclic decomposition given by Eq. 9 ensures that the block with global index 0 is placed in process 0, the next block is placed in process 1, and so on. However, it is sometimes necessary to offset the processes relative to the global block index so that, in general, the first block is placed in process p_0, the next in process $p_0 + 1$, and so on. We, therefore, generalize the block cyclic decomposition by replacing m on the righthand side of Eq. 9 by $m' = m + rp_0$ to give,

$$\begin{aligned} m &\mapsto \left(\left\lfloor \frac{m' \bmod T}{r} \right\rfloor, \left\lfloor \frac{m'}{T} \right\rfloor, m' \bmod r \right) \\ &= \left(\left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor + p_0 \right) \bmod P, \left\lfloor \frac{m + rp_0}{T} \right\rfloor, m \bmod r \right). \end{aligned} \tag{11}$$

m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
p	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2
b	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	3	3
i	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0

(a) $m \mapsto (p, b, i)$

p	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2
b	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3
i	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
m	0	1	6	7	12	13	18	19	2	3	8	9	14	15	20	21	4	5	10	11	16	17	22

(b) $(p, b, i) \mapsto m$

Figure 9: An example of the block cyclic decomposition of $M = 23$ data objects over $P = 3$ processes for a block size of $r = 2$. (a) shows the mapping from global index, m, to the triplet (p, b, i), and (b) shows the inverse mapping.

Equation 10 may also be generalized to,

$$(p, b, i) \mapsto Br + i = (p - p_0)r + bT + i \tag{12}$$

where now the global block number is given by $B = (p - p_0) + bP$. It should be noted that in processes with $p < p_0$, block 0 is not within the range of the block cyclic mapping and it is, therefore, an error to reference it in any way.

In decomposing an $M \times N$ matrix we apply independent block cyclic decompositions in the row and column directions. Thus, suppose the matrix rows are distributed with block size r and offset p_0 over P processes by the block cyclic mapping $\mu_{r,p_0,P}$, and the matrix columns are distributed with block size s and offset q_0 over Q processes by the block cyclic mapping $\nu_{s,q_0,Q}$. Then the matrix element indexed globally by (m, n) is mapped as follows,

$$\begin{aligned} m &\overset{\mu}{\longmapsto} (p, b, i) \\ n &\overset{\nu}{\longmapsto} (q, d, j). \end{aligned} \tag{13}$$

The decomposition of the matrix can be regarded as the tensor product of the row and column decompositions, and we can write,

$$(m, n) \mapsto \big((p, q),\ (b, d),\ (i, j) \big). \tag{14}$$

The block cyclic matrix decomposition given by Eqs. 13 and 14 distributes blocks of size $r \times s$ to a mesh of $P \times Q$ processes. We shall refer to this mesh as the *process template*, and refer to processes by their position in the template. Equation 14 says that global index (m, n) is mapped to process (p, q), where it is stored in the block at location (b, d) in a two-dimensional array of blocks. Within this block it is stored at location (i, j). The decomposition is completely specified by the parameters r, s, p_0, q_0, P, and Q. In Figure 10 an example is given of the block cyclic decomposition of a 48×80 matrix for block size 4×5, a process template 3×4, and a template offset $(p_0, q_0) = (0, 0)$. Figure 11 shows the same example but for a template offset of $(1, 2)$.

The block cyclic decomposition can reproduce most of the data distributions commonly used in linear algebra computations on parallel computers. For example, if $Q = 1$ and $r = \lceil M/P \rceil$ the block row decomposition is obtained. Similarly, $P = 1$ and $s = \lceil N/Q \rceil$ gives a block column decomposition. These decompositions, together with row and column cyclic decompositions, are shown in Figure 12. Other commonly used block cyclic matrix decompositions are shown in Figure 13.

6. Parallel Implementation

In this section we describe the parallel implementation of LU factorization, with partial pivoting over rows, for a block-partitioned matrix. The matrix, A, to be factored is assumed to have a block cyclic decomposition, and at the end of the computation is overwritten by the lower and upper triangular factors, L and U. This implicitly determines the decomposition of L and U. Quite a high-level description is given here since the details of the parallel implementation involve optimization issues that will be addressed in Section 7.

The sequential LU factorization algorithm described in Section 4.1 uses square blocks. Although in the parallel algorithm we could choose to decompose the matrix using nonsquare blocks, this would result in a more complicated code, and additional sources of concurrent overhead. For LU factorization we, therefore, restrict the decomposition to use only square blocks, so that the blocks used to decompose the matrix are the same as those used to partition the computation. If the block size is $r \times r$, then an $M \times N$ matrix consists of $M_b \times N_b$ blocks, where $M_b = \lceil M/r \rceil$ and $N_b = \lceil N/r \rceil$.

As discussed in Section 4.1, LU factorization proceeds in a series of sequential steps indexed by $k = 0, \min(M_b, N_b) - 1$, in each of which the following three tasks are performed,

1. factor the kth column of blocks, performing pivoting if necessary. This evaluates the matrices L_0, L_1, and U_0 in Figure 6.
2. evaluate the kth block row of U by solving the lower triangular system $L_0 U_1 = C$.
3. do a rank-r update on the trailing submatrix E, replacing it with $E' = E - L_1 U_1$.

We now consider the parallel implementation of each of these tasks. The computation in the factorization step involves a single column of blocks, and these lie in a single column of

									D								
	p,q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
	1	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
	2	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
	3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
	4	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
	5	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
B	6	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
	7	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
	8	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3
	9	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3
	10	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3
	11	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3

(a) Assignment of global block indices, (B, D), to processes, (p, q).

								q								
B,D	0				1				2				3			
	0,0	0,4	0,8	0,12	0,1	0,5	0,9	0,13	0,2	0,6	0,10	0,14	0,3	0,7	0,11	0,15
0	3,0	3,4	3,8	3,12	3,1	3,5	3,9	3,13	3,2	3,6	3,10	3,14	3,3	3,7	3,11	3,15
	6,0	6,4	6,8	6,12	6,1	6,5	6,9	6,13	6,2	6,6	6,10	6,14	6,3	6,7	6,11	6,15
	9,0	9,4	9,8	9,12	9,1	9,5	9,9	9,13	9,2	9,6	9,10	9,14	9,3	9,7	9,11	9,15
	1,0	1,4	1,8	1,12	1,1	1,5	1,9	1,13	1,2	1,6	1,10	1,14	1,3	1,7	1,11	1,15
	4,0	4,4	4,8	4,12	4,1	4,5	4,9	4,13	4,2	4,6	4,10	4,14	4,3	4,7	4,11	4,15
p 1	7,0	7,4	7,8	7,12	7,1	7,5	7,9	7,13	7,2	7,6	7,10	7,14	7,3	7,7	7,11	7,15
	10,0	10,4	10,8	10,12	10,1	10,5	10,9	10,13	10,2	10,6	10,10	10,14	10,3	10,7	10,11	10,15
	2,0	2,4	2,8	2,12	2,1	2,5	2,9	2,13	2,2	2,6	2,10	2,14	2,3	2,7	2,11	2,15
	5,0	5,4	5,8	5,12	5,1	5,5	5,9	5,13	5,2	5,6	5,10	5,14	5,3	5,7	5,11	5,15
2	8,0	8,4	8,8	8,12	8,1	8,5	8,9	8,13	8,2	8,6	8,10	8,14	8,3	8,7	8,11	8,15
	11,0	11,4	11,8	11,12	11,1	11,5	11,9	11,13	11,2	11,6	11,10	11,14	11,3	11,7	11,11	11,15

(b) Global blocks, (B, D), in each process, (p, q).

Figure 10: Block cyclic decomposition of a 48×80 matrix with a block size of 4×5, onto a 3×4 process template. Each small rectangle represents one matrix block – individual matrix elements are not shown. In (a), shading is used to emphasize the process template that is periodically stamped over the matrix, and each block is labeled with the process to which it is assigned. In (b), each shaded region shows the blocks in one process, and is labeled with the corresponding global block indices. In both figures, the black rectangles indicate the blocks assigned to process $(0, 0)$.

p,q	D 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
B 0	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
2	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
3	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
4	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
5	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
6	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
7	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
8	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
9	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
10	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
11	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1

(a) Assignment of global block indices, (B, D), to processes, (p, q).

B,D	q 0					1					2					3				
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	2,2	2,6	2,10	2,14	—	2,3	2,7	2,11	2,15	2,0	2,4	2,8	2,12	—	2,1	2,5	2,9	2,13	—
p 0	—	5,2	5,6	5,10	5,14	—	5,3	5,7	5,11	5,15	5,0	5,4	5,8	5,12	—	5,1	5,5	5,9	5,13	—
	—	8,2	8,6	8,10	8,14	—	8,3	8,7	8,11	8,15	8,0	8,4	8,8	8,12	—	8,1	8,5	8,9	8,13	—
	—	11,2	11,6	11,10	11,14	—	11,3	11,7	11,11	11,15	11,0	11,4	11,8	11,12	—	11,1	11,5	11,9	11,13	—
	—	0,2	0,6	0,10	0,14	—	0,3	0,7	0,11	0,15	0,0	0,4	0,8	0,12	—	0,1	0,5	0,9	0,13	—
	—	3,2	3,6	3,10	3,14	—	3,3	3,7	3,11	3,15	3,0	3,4	3,8	3,12	—	3,1	3,5	3,9	3,13	—
1	—	6,2	6,6	6,10	6,14	—	6,3	6,7	6,11	6,15	6,0	6,4	6,8	6,12	—	6,1	6,5	6,9	6,13	—
	—	9,2	9,6	9,10	9,14	—	9,3	9,7	9,11	9,15	9,0	9,4	9,8	9,12	—	9,1	9,5	9,9	9,13	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	—	1,2	1,6	1,10	1,14	—	1,3	1,7	1,11	1,15	1,0	1,4	1,8	1,12	—	1,1	1,5	1,9	1,13	—
	—	4,2	4,6	4,10	4,14	—	4,3	4,7	4,11	4,15	4,0	4,4	4,8	4,12	—	4,1	4,5	4,9	4,13	—
2	—	7,2	7,6	7,10	7,14	—	7,3	7,7	7,11	7,15	7,0	7,4	7,8	7,12	—	7,1	7,5	7,9	7,13	—
	—	10,2	10,6	10,10	10,14	—	10,3	10,7	10,11	10,15	10,0	10,4	10,8	10,12	—	10,1	10,5	10,9	10,13	—
	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

(b) Global blocks, (B, D), in each process, (p, q).

Figure 11: The same matrix decomposition as shown in Figure 10, but for a template offset of $(p_0, q_0) = (1, 2)$. Dashed entries in (b) indicate that the block does not contain any data. In both figures, the black rectangles indicate the blocks assigned to process $(0, 0)$.

0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0

(a) $r = 3$, $s = 10$, $P = 4$, $Q = 1$

0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0
2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0	2,0
3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0	3,0
0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0	1,0

(b) $r = 1$, $s = 10$, $P = 4$, $Q = 1$

0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3

(c) $r = 10$, $s = 3$, $P = 1$, $Q = 4$

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1

(d) $r = 10$, $s = 1$, $P = 1$, $Q = 4$

Figure 12: These 4 figures show different ways of decomposing a 10×10 matrix. Each cell represents a matrix element, and is labeled by the position, (p, q), in the template of the process to which it is assigned. To emphasize the pattern of decomposition, the matrix entries assigned to the process in the first row and column of the template are shown shaded, and each separate shaded region represents a matrix block. Figures (a) and (b) show block and cyclic row-oriented decompositions, respectively, for 4 nodes. In figures (c) and (d) the corresponding column-oriented decompositions are shown. Below each figure we give the values of r, s, P, and Q corresponding to the decomposition. In all cases $p_0 = q_0 = 0$.

0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
3,0	3,0	3,0	3,1	3,1	3,1	3,2	3,2	3,2	3,3

(a) $r = 3,\ s = 3,\ P = 4,\ Q = 4$

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1

(b) $r = 3,\ s = 1,\ P = 4,\ Q = 4$

0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
3,0	3,0	3,0	3,1	3,1	3,1	3,2	3,2	3,2	3,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3
2,0	2,0	2,0	2,1	2,1	2,1	2,2	2,2	2,2	2,3
3,0	3,0	3,0	3,1	3,1	3,1	3,2	3,2	3,2	3,3
0,0	0,0	0,0	0,1	0,1	0,1	0,2	0,2	0,2	0,3
1,0	1,0	1,0	1,1	1,1	1,1	1,2	1,2	1,2	1,3

(c) $r = 1,\ s = 3,\ P = 4,\ Q = 4$

0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1
2,0	2,1	2,2	2,3	2,0	2,1	2,2	2,3	2,0	2,1
3,0	3,1	3,2	3,3	3,0	3,1	3,2	3,3	3,0	3,1
0,0	0,1	0,2	0,3	0,0	0,1	0,2	0,3	0,0	0,1
1,0	1,1	1,2	1,3	1,0	1,1	1,2	1,3	1,0	1,1

(d) $r = 1,\ s = 1,\ P = 4,\ Q = 4$

Figure 13: These 4 figures show different ways of decomposing a 10×10 matrix over 16 processes arranged as a 4×4 template. Below each figure we give the values of r, s, P, and Q corresponding to the decomposition. In all cases $p_0 = q_0 = 0$.

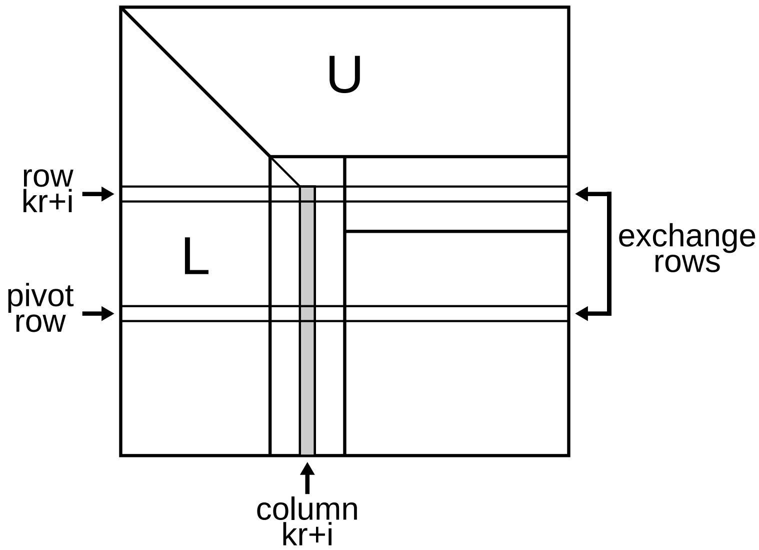


Figure 14: This figure shows pivoting for step i of the kth stage of LU factorization. The element with largest absolute value in the gray shaded part of column $kr + i$ is found, and the row containing it is exchanged with row $kr + i$. If the rows exchanged lie in different processes, communication may be necessary.

the process template. In the kth factorization step, each of the r columns in block column k is processed in turn. Consider the ith column in block column k. The pivot is selected by finding the element with largest absolute value in this column between row $kr + i$ and the last row, inclusive. The elements involved in the pivot search at this stage are shown shaded in Figure 14. Having selected the pivot, the value of the pivot and its row are broadcast to all other processors. Next, pivoting is performed by exchanging the entire row $kr + i$ with the row containing the pivot. We exchange entire rows, rather than just the part to the right of the columns already factored, in order to simplify the application of the pivots to the righthand side in any subsequent solve phase. Finally, each value in the column below the pivot is divided by the pivot. If a cyclic column decomposition is used, like that shown in Figure 12(d), only one processor is involved in the factorization of the block column, and no communication is necessary between the processes. However, in general P processes are involved, and communication is necessary in selecting the pivot, and exchanging the pivot rows.

The solution of the lower triangular system $L_0U_1 = C$ to evaluate the kth block row of U involves a single row of blocks, and these lie in a single row of the process template. If a cyclic row decomposition is used, like that shown in Figure 12(b), only one processor is involved in the triangular solve, and no communication is necessary between the processes. However, in general Q processes are involved, and communication is necessary to broadcast the lower triangular matrix, L_0, to all processes in the row. Once this has been done, each process in the

```
pcol= q_0
prow= p_0
do k= 0, min(M_b, N_b) - 1
   do i= 0, r - 1
      if (q =pcol) find pivot value and location
      broadcast pivot value and location to all processes
      exchange pivot rows
      if (q =pcol) divide column r below diagonal by pivot
   end do

   if (p =prow) then
      broadcast L_0 to all process in same template row
      solve L_0 U_1 = C
   end if

   broadcast L_1 to all processes in same template row
   broadcast U_1 to all processes in same template column
   update E <- E - L_1 U_1

   pcol= (pcol + 1) mod Q
   prow= (prow + 1) mod P
end do
```

Figure 15: Pseudocode for the basic parallel block-partitioned LU factorization algorithm. This code is executed by each process. The first box inside the k loop factors the kth column of blocks. The second box solves a lower triangular system to evaluate the kth row of blocks of U, and the third box updates the trailing submatrix. The template offset is given by (p_0, q_0), and (p, q) is position of a process in the template.

row independently performs a lower triangular solve for the blocks of C that it holds.

The communication necessary to update the trailing submatrix at step k takes place in two steps. First, each process holding part of L_1 broadcasts these blocks to the other processes in the same row of the template. This may be done in conjunction with the broadcast of L_0, mentioned in the preceding paragraph, so that all of the factored panel is broadcast together. Next, each process holding part of U_1 broadcasts these blocks to the other processes in the same column of the template. Each process can then complete the update of the blocks that it holds with no further communication.

A pseudocode outline of the parallel LU factorization algorithm is given in Figure 15. There are two points worth noting in Figure 15. First, the triangular solve and update phases operate on matrix blocks and may, therefore, be done with parallel versions of the Level 3 BLAS (specifically, xTRSM and xGEMM, respectively). The factorization of the column of blocks, however, involves a loop over matrix columns. Hence, is it not a block-oriented computation, and cannot be performed using the Level 3 BLAS. The second point to note is that most of the parallelism in the code comes from updating the trailing submatrix since this is the only phase in which all the processes are busy.

Figure 15 also shows quite clearly where communication is required; namely, in finding the pivot, exchanging pivot rows, and performing various types of broadcast. The exact way in which these communications are done and interleaved with computation generally has an important effect on performance, and will be discussed in more detail in Section 7.

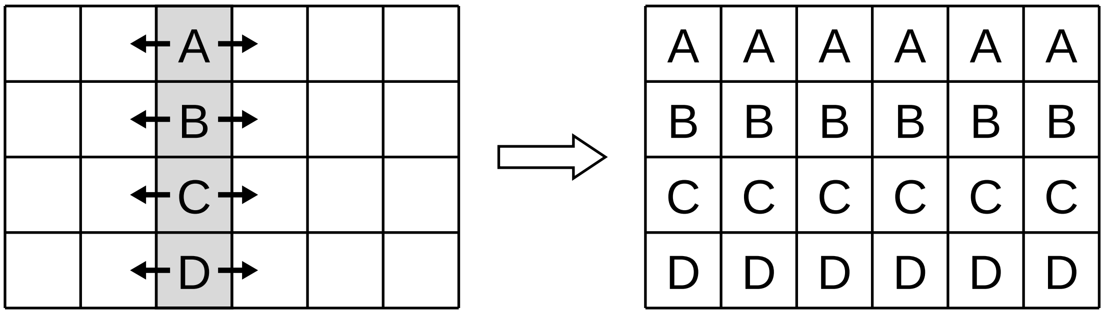
Figure 15 refers to broadcasting data to all processes in the same row or column of the template. This is a common operation in parallel linear algebra algorithms, so the idea will be described here in a little more detail. Consider, for example, the task of broadcasting the lower triangular block, L_0, to all processes in the same row of the template, as required before solving $L_0U_1 = C$. If L_0 is in process (p,q), then it will be broadcast to all processes in row p of the process template. As a second example, consider the broadcast of L_1 to all processes in the same template row, as required before updating the trailing submatrix. This type of "rowcast" is shown schematically in Figure 16(a). If L_1 is in column q of the template, then each process (p,q) broadcasts its blocks of L_1 to the other processes in row p of the template. Loosely speaking, we can say that L_0 and L_1 are broadcast along the rows of the template. This type of data movement is the same as that performed by the Fortran 90 routine SPREAD [7]. The broadcast of U_1 to all processes in the same template column is very similar. This type of communication is sometimes referred to as a "colcast", and is shown in Figure 16(b).

7. Optimization, Tuning, and Trade-offs

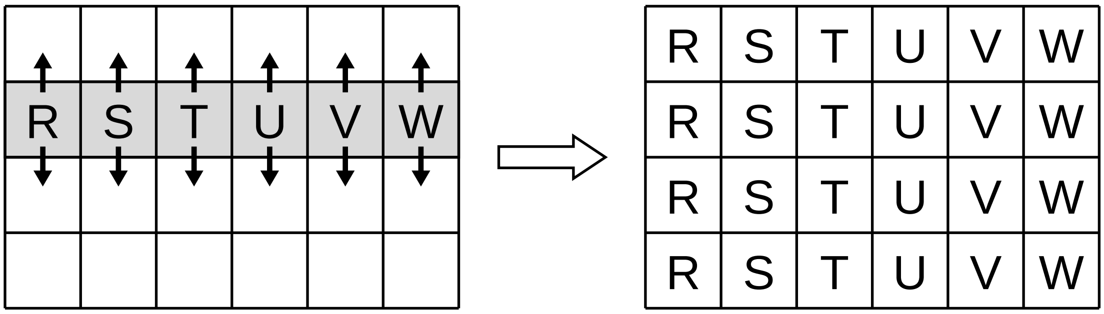
In this section, we shall examine techniques for optimizing the basic LU factorization code presented in Section 4.1. Among the issues to be considered are the assignment of processes to physical processors, the arrangement of the data in the local memory of each process, the trade-off between load imbalance and communication latency, the potential for overlapping communication and calculation, and the type of algorithm used to broadcast data. Many of these issues are interdependent, and in addition the portability and ease of code maintenance and use must be considered. For further details of the optimization of parallel LU factorization algorithms for specific concurrent machines, together with timing results, the reader is referred to the work of Chu and George [12], Geist and Heath [33], Geist and Romine [34], Van de Velde [49], Brent [8], Hendrickson and Womble [36], Lichtenstein and Johnsson [42], and Dongarra and co-workers [10, 25].

7.1. Mapping Logical Memory to Physical Memory

In Section 5, a logical (or virtual) matrix decomposition was described in which the global index (m,n) is mapped to a position, (p,q), in a logical process template, a position, (b,d), in



(a) Broadcast along rows.



(b) Broadcast along columns.

Figure 16: Schematic representation of broadcast along rows and columns of a 4×6 process template. In (a), each shaded process broadcasts to the processes in the same row of the process template. In (b), each shaded process broadcasts to the processes in the same column of the process template.

a logical array of blocks local to the process, and a position, (i, j), in a logical array of matrix elements local to the block. Thus, the block cyclic decomposition is hierarchical, and attempts to represent the hierarchical memory of advanced-architecture computers. Although the parallel LU factorization algorithm can be specified solely in terms of this logical hierarchical memory, its performance depends on how the logical memory is mapped to physical memory.

7.1.1. Assignment of Processes to Processors

Consider, first, the assignment of processes, (p, q), to physical processors. In general, more than one process may be assigned to a processor, so the problem may be overdecomposed. To avoid load imbalance the same number of processes should be assigned to each processor as nearly as possible. If this condition is satisfied, the assignment of processes to processors can still affect performance by influencing the communication overhead. On recent distributed memory machines, such as the Intel Delta and CM-5, the time to send a single message between two processors is largely independent of their physical location [28, 43, 44], and hence the assignment of processes to processors does not have much direct effect on performance. However, when a collective communication task, such as a broadcast, is being done, contention for physical resources can degrade performance. Thus, the way in which processes are assigned to processors can affect performance if some assignments result in differing amounts of contention. Logarithmic contention-free broadcast algorithms have been developed for processors connected as a two-dimensional mesh [6, 46], so on such machines process (p, q) is usually mapped to the processor at position (p, q) in the mesh of processors. Such an assignment also ensures that the multiple one-dimensional broadcasts of L_1 and U_1 along the rows and columns of the template, respectively, do not give rise to contention.

7.1.2. Layout of Local Process Memory

The layout of matrix blocks in the local memory of a process, and the arrangement of matrix elements within each block, can also affect performance. Here, tradeoffs among several factors need to be taken into account. When communicating matrix blocks, for example in the broadcasts of L_1 and U_1, we would like the data in each block to be contiguous in physical memory so there is no need to pack them into a communication buffer before sending them. On the other hand, when updating the trailing submatrix, E, each process multiplies a column of blocks by a row of blocks, to do a rank-r update on the part of E that it contains. If this were done as a series of separate block-block matrix multiplications, as shown in Figure 18(a), the performance would be poor except for sufficiently large block sizes, r, since the vector and/or pipeline units on most processors would not be fully utilized, as may be seen in Figure 17 for the i860 processor. Instead, we arrange the loops of the computation as shown in Figure 18(b). Now, if the data are laid out in physical memory first by running over the i index and then over the d index the inner two loops can be merged, so that the length of the inner loop is now $rd_{\max}$. This generally results in much better vector/pipeline performance. The b and j loops in Figure 18(b) can also be merged, giving the algorithm shown in Figure 18(c). This is just the outer product form of the multiplication of an $rd_{\max} \times r$ by an $r \times rb_{\max}$ matrix, and would usually be done by a call to the Level 3 BLAS routine xGEMM of which an assembly coded sequential version is available on most machines. Note that in Figure 18(c) the order of the inner two loops is appropriate for a Fortran implementation – for the C language this order should be reversed, and the data should be stored in each process by rows instead of by columns.

We have found in our work on the Intel iPSC/860 hypercube and the Delta system that it is better to optimize for the sequential matrix multiplication with an (i, d, j, b) ordering of

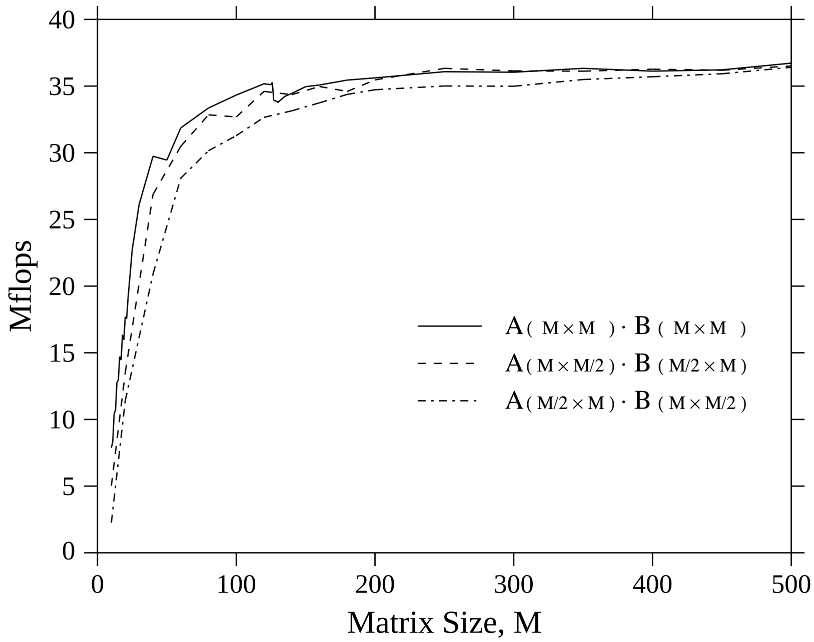


Figure 17: Performance of the assembly-coded Level 3 BLAS matrix multiplication routine DGEMM on one i860 processor of the Intel Delta system. Results for square and rectangular matrices are shown. Note that the peak performance of about 35 Mflops is attained only for matrices whose smallest dimension exceeds 100. Thus, performance is improved if a few large matrices are multiplied by each process, rather than many small ones.

memory in each process, rather than adopting an (i, j, d, b) ordering to avoid buffer copies when communicating blocks. However, there is another reason for doing this. On most distributed memory computers the message startup cost is sufficiently large that it is preferable wherever possible to send data as one large message rather than as several smaller messages. Thus, when communicating L_1 and U_1 the blocks to be broadcast would be amalgamated into a single message, which requires a buffer copy. The emerging Message Passing Interface (MPI) standard [21] provides support for noncontiguous messages, so in the future the need to avoid buffer copies will not be of such concern to the application developer.

7.2. Tradeoffs between Load Balance and Communication Latency

We have discussed the mapping of the logical hierarchical memory to physical memory. In addition, we have pointed out the importance of maintaining long inner loops to get good sequential performance for each process, and the desirability of sending a few large messages rather than many smaller ones. We next consider load balance issues. Assuming that equal numbers of processes have been assigned to each processor, load imbalance arises in two phases of the parallel LU factorization algorithm; namely, in factoring each column block, which involves only P processes, and in solving the lower triangular system to evaluate each row block of U, which

```
do b = 0, b_max − 1
  do d = 0, d_max − 1
    do i = 0, r − 1
      do j = 0, r − 1
        do k = 0, r − 1
          E(b, d; i, j) = E(b, d; i, j) − L_1(b, d; i, k) U_1(b, d; k, j)
end all do loops
```

(a) Block-block multiplication

```
do k = 0, r − 1
  do b = 0, b_max − 1
    do j = 0, r − 1
      do d = 0, d_max − 1
        do i = 0, r − 1
          E(b, d; i, j) = E(b, d; i, j) − L_1(b, d; i, k) U_1(b, d; k, j)
end all do loops
```

(b) Intermediate form of algorithm

```
do k = 0, r − 1
  do x = 0, r b_max − 1
    do y = 0, r d_max − 1
      E(x, y) = E(x, y) − L_1(x, k) U_1(k, y)
end all do loops
```

(c) Outer product form of algorithm

Figure 18: Pseudocode for different versions of the rank-r update, $E \leftarrow E - L_1 U_1$, for one process. The number of row and column blocks per process is given by $b_{\max}$ and $d_{\max}$, respectively; r is the block size. Blocks are indexed by (b, d), and elements within a block by (i, j). In version (a) the $r \times r$ blocks are multiplied one at a time, giving an inner loop of length r. (b) shows the loops rearranged before merging the i and d loops, and the j and b loops. This leads to the outer product form of the algorithm shown in (c) in which the inner loop is now of length $rd_{\max}$.

involves only Q processes. If the time for data movement is negligible, the aspect ratio of the template that minimizes load imbalance in step k of the algorithm is,

$$\begin{aligned} \frac{P}{Q} &= \frac{\text{Sequential time to factor column block}}{\text{Sequential time for triangular solve}} \\ &= \frac{M_b - k - 1/3 + \mathrm{O}(1/r^2)}{N_b - k - 1 + \mathrm{O}(1/r^2)} \end{aligned} \tag{15}$$

where $M_b \times N_b$ is the matrix size in blocks, and r the block size. Thus, the optimal aspect ratio of the template should be the same as the aspect ratio of the matrix, i.e., M_b/N_b in blocks, or M/N in elements. If the effect of communication time is included then we must take into account the relative times taken to locate and broadcast the pivot information, and

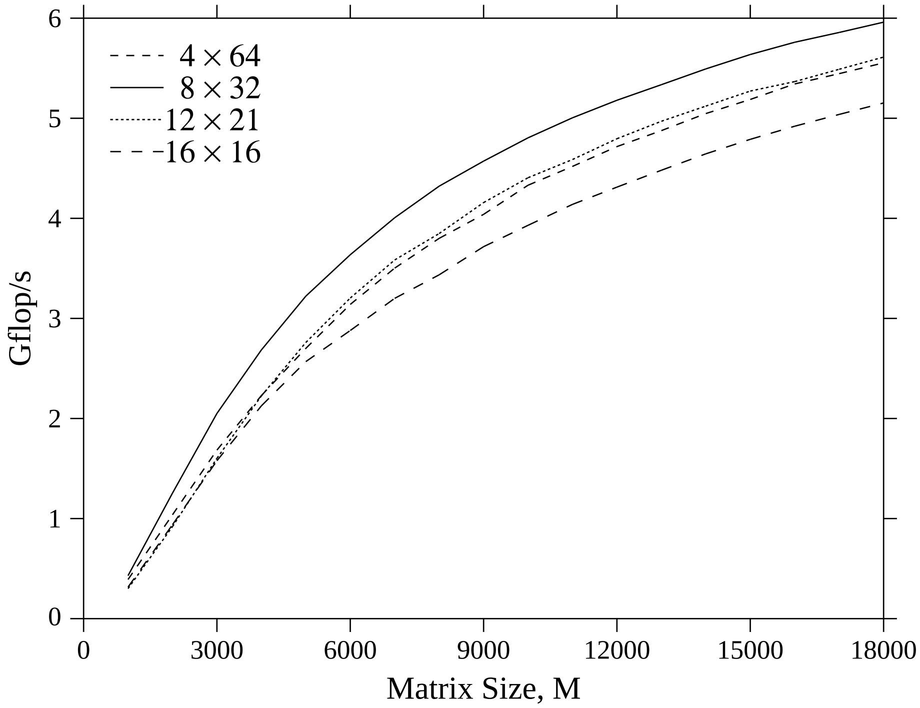


Figure 19: Performance of LU factorization on the Intel Delta as a function of square matrix size for different processor templates containing approximately 256 processors. The best performance is for an aspect ratio of 1/4, though the dependence on aspect ratio is rather weak.

the time to broadcast the lower triangular matrix, L_0, along a row of the template. For both tasks the communication time increases with the number of processes involved, and since the communication time associated with the pivoting is greater than that associated with the triangular solve, we would expect the optimum aspect ratio of the template to be less than M/N. In fact, for our runs on the Intel Delta system we found an aspect ratio, P/Q, of between 1/4 and 1/8 to be optimal for most problems with square matrices, and that performance depends rather weakly on the aspect ratio, particularly for large grain sizes. Some typical results are shown in Figure 19 for 256 processors, which show a variation of less than 20% in performance as P/Q varies between 1/16 and 1 for the largest problem.

The block size, r, also affects load balance. Here the tradeoff is between the load imbalance that arises as rows and columns of the matrix are eliminated as the algorithm progresses, and communication startup costs. The block cyclic decomposition seeks to maintain good load balance by cyclically assigning blocks to processes, and the load balance is best if the blocks are small. On the other hand, cumulative communication startup costs are less if the block size is large since, in this case, fewer messages must be sent (although the total volume of data sent is independent of the block size). Thus, there is a block size that optimally balances the load imbalance and communication startup costs.

7.3. Optimality and Pipelining Tradeoffs

The communication algorithms used also influence performance. In the LU factorization algorithm, all the communication can be done by moving data along rows and/or columns of the process template. This type of communication can be done by passing from one process to the next along the row or column. We shall call this a "ring" algorithm, although the ring may, or may not, be closed. An alternative is to use a spanning tree algorithm, of which there are several varieties. The complexity of the ring algorithm is linear in the number of processes involved, whereas that of spanning tree algorithms is logarithmic (for example, see [6]). Thus, considered in isolation, the spanning tree algorithms are preferable to a ring algorithm. However, in a spanning tree algorithm, a process may take part in several of the logarithmic steps, and in some implementations these algorithms act as a barrier. In a ring algorithm, each process needs to communicate only once, and can then continue to compute, in effect overlapping the communication with computation. An algorithm that interleaves communication and calculation in this way is often referred to as a pipelined algorithm. In a pipelined LU factorization algorithm with no pivoting, communication and calculation would flow in waves across the matrix. Pivoting tends to inhibit this advantage of pipelining.

In the pseudocode in Figure 15, we do not specify how the pivot information should be broadcast. In an optimized implementation, we need to finish with the pivot phase, and the triangular solve phase, as soon as possible in order to begin the update phase which is richest in parallelism. Thus, it is not a good idea to broadcast the pivot information from a single source process using a spanning tree algorithm, since this may occupy some of the processes involved in the panel factorization for too long. It is important to get the pivot information to the other processes in this template column as soon as possible, so the pivot information is first sent to these processes which subsequently broadcast it along the template rows to the other processes not involved in the panel factorization. In addition, the exchange of the parts of the pivot rows lying within the panel is done separately from that of the parts outside the pivot panel. Another factor to consider here is when the pivot information should be broadcast along the template columns. In Figure 15, the information is broadcast, and rows exchanged, immediately after the pivot is found. An alternative is to store up the sequence of r pivots for a panel and to broadcast them along the template rows when panel factorization is complete. This defers the exchange of pivot rows for the parts outside the panel until the panel factorization has been done, as shown in the pseudocode fragment in Figure 20. An advantage of this second approach is that only one message is used to send the pivot information for the panel along the template

```
if (q =pcol) then
   do i= 0, r − 1
      find pivot value and location
      exchange pivot rows lying within panel
      divide column r below diagonal by pivot
   end do
end if
broadcast pivot information for r pivots along template rows
exchange pivot rows lying outside the panel for each of r pivots
```

Figure 20: Pseudocode fragment for partial pivoting over rows. This may be regarded as replacing the first box inside the k loop in Figure 15. In the above code pivot information is first disseminated within the template column doing the panel factorization. The pivoting of the parts of the rows lying outside the panel is deferred until the panel factorization has been completed.

rows, instead of r messages.

In our implementation of LU factorization on the Intel Delta system, we used a spanning tree algorithm to locate the pivot and to broadcast it within the column of the process template performing the panel factorization. This ensures that pivoting, which involves only P processes, is completed as quickly as possible. A ring broadcast is used to pipeline the pivot information and the factored panel along the template rows. Finally, after the triangular solve phase has completed, a spanning tree broadcast is used to send the newly-formed block row of U along the template columns. Results for square matrices from runs on the Intel Delta system are shown in Figure 21. For each curve the results for the best process template configuration are shown. For a memory-constrained scalable algorithm the performance should depend linearly on the number of processors for fixed granularity, and so scalability may be assessed by the extent to which isogranularity curves differ from linearity. An isogranularity curve is a plot of performance against number of processors for a fixed granularity. The results in Figure 21 can be used to generate the isogranularity curves shown in Figure 22 which show that on the Delta system the LU factorization routine starts to lose scalability when the granularity falls below about 0.2×10^6. This corresponds to a matrix size of about $M = 10000$ on 512 processors, or about 13% of the memory available to applications on the Delta, indicating that LU factorization scales rather well on the Intel Delta system.

8. Conclusions and Future Research Directions

Portability of programs has always been an important consideration. Portability was easy to achieve when there was a single architectural paradigm (the serial von Neumann machine) and a single programming language for scientific programming (Fortran) embodying that common

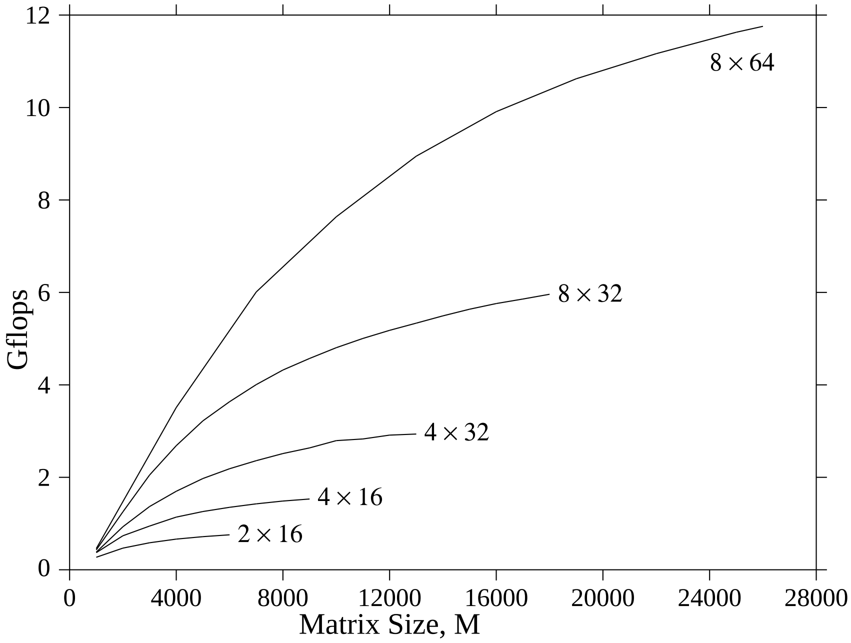


Figure 21: Performance of LU factorization on the Intel Delta as a function of square matrix size for different numbers of processors. For each curve, results are shown for the process template configuration that gave the best performance for that number of processors.

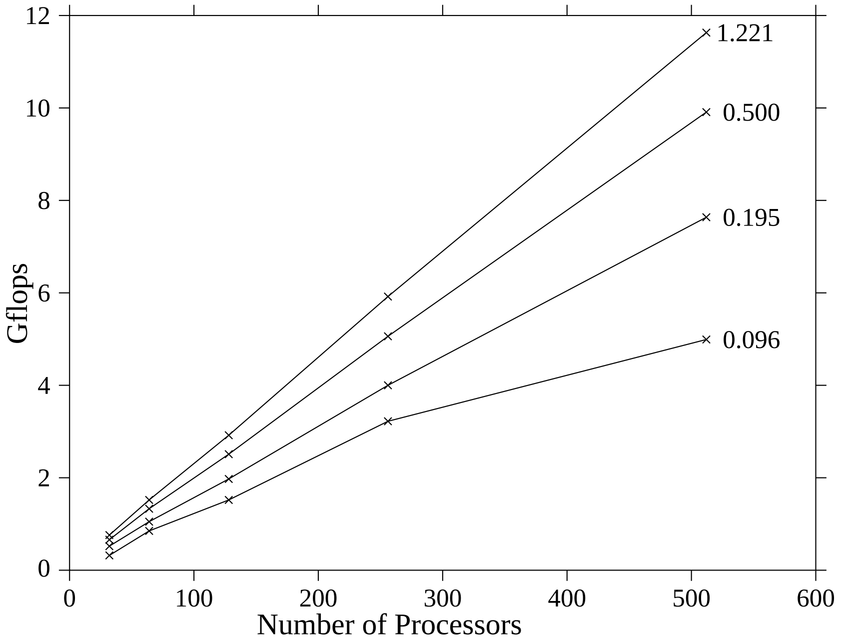


Figure 22: Isogranularity curves in the (N_p, G) plane for the LU factorization of square matrices on the Intel Delta system. The curves are labeled by the granularity in units of 10^6 matrix elements per processor. The linearity of the plots for granularities exceeding about 0.2×10^6 indicates that the LU factorization algorithm scales well on the Delta.

model of computation. Architectural and linguistic diversity have made portability much more difficult, but no less important, to attain. Users simply do not wish to invest significant amounts of time to create large-scale application codes for each new machine. Our answer is to develop portable software libraries that hide machine-specific details.

In order to be truly portable, parallel software libraries must be *standardized*. In a parallel computing environment in which the higher-level routines and/or abstractions are built upon lower-level computation and message-passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of computational and message-passing standards provides vendors with a clearly defined base set of routines that they can implement efficiently.

From the user's point of view, portability means that, as new machines are developed, they are simply added to the network, supplying cycles where they are most appropriate.

From the mathematical software developer's point of view, portability may require significant effort. Economy in development and maintenance of mathematical software demands that such development effort be leveraged over as many different computer systems as possible. Given the great diversity of parallel architectures, this type of portability is attainable to only a limited degree, but machine dependences can at least be isolated.

LAPACK is an example of a mathematical software package whose highest-level components are portable, while machine dependences are hidden in lower-level modules. Such a hierarchical approach is probably the closest one can come to software portability across diverse parallel architectures. And the BLAS that are used so heavily in LAPACK provide a portable, efficient, and flexible standard for applications programmers.

Like portability, *scalability* demands that a program be reasonably effective over a wide range of number of processors. The scalability of parallel algorithms, and software libraries based on them, over a wide range of architectural designs and numbers of processors will likely require that the fundamental granularity of computation be adjustable to suit the particular circumstances in which the software may happen to execute. Our approach to this problem is block algorithms with adjustable block size. In many cases, however, polyalgorithms[2] may be required to deal with the full range of architectures and processor multiplicity likely to be available in the future.

Scalable parallel architectures of the future are likely to be based on a distributed memory architectural paradigm. In the longer term, progress in hardware development, operating systems, languages, compilers, and communications may make it possible for users to view such distributed architectures (without significant loss of efficiency) as having a shared memory with

[2]In a polyalgorithm the actual algorithm used depends on the computing environment and the input data. The optimal algorithm in a particular instance is automatically selected at runtime.

a global address space. For the near term, however, the distributed nature of the underlying hardware will continue to be visible at the programming level; therefore, efficient procedures for explicit communication will continue to be necessary. Given this fact, standards for basic message passing (send/receive), as well as higher-level communication constructs (global summation, broadcast, etc.), become essential to the development of scalable libraries that have any degree of portability. In addition to standardizing general communication primitives, it may also be advantageous to establish standards for problem-specific constructs in commonly occurring areas such as linear algebra.

The BLACS (Basic Linear Algebra Communication Subprograms) [16, 26] is a package that provides the same ease of use and portability for MIMD message-passing linear algebra communication that the BLAS [17, 18, 41] provide for linear algebra computation. Therefore, we recommend that future software for dense linear algebra on MIMD platforms consist of calls to the BLAS for computation and calls to the BLACS for communication. Since both packages will have been optimized for a particular platform, good performance should be achieved with relatively little effort. Also, since both packages will be available on a wide variety of machines, code modifications required to change platforms should be minimal. This is borne out by prototype implementations of ScaLAPACK on the IBM SP-1, the CRAY T3D, and Thinking Machines Corporation's CM-5. Preliminary results for the IBM SP-1 and CRAY T3D indicate that the ScaLAPACK code runs efficiently on these machines – a 12000×12000 LU factorization runs at 21.4 Gflop/s on a 256-node CRAY T3D. An 8000×8000 LU factorization runs at 5.3 Gflop/s on a 64-node CRAY T3D, compared with 2.6 and 2.0 Gflop/s for 64-node IBM SP-1 and Intel Paragon machines, respectively. Initial results on the CM-5, however, have been disappointing because of the difficulty of using the vector units in message passing programs.

The EISPACK, LINPACK, and LAPACK linear algebra libraries are in the public domain, and are available from *netlib*. For example, for more information on how to obtain LAPACK, send the following one-line email message to `netlib@ornl.gov`:

```
send index from lapack
```

Information for EISPACK and LINPACK can be similarly obtained. A preliminary version of the ScaLAPACK library is also available from *netlib*.

Acknowledgments

The authors are grateful for the comments and suggestions of the anonymous referees. This research was performed in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided through the Center for Research on Parallel Computing.

9. References

[1] E. Anderson, A. Benzoni, J. J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. LAPACK for distributed memory architectures: Progress report. In *Parallel Processing for Scientific Computing, Fifth SIAM Conference*. SIAM, 1991.

[2] E. Anderson and J. Dongarra. Results from the initial release of LAPACK. Technical Report LAPACK working note 16, Computer Science Department, University of Tennessee, Knoxville, TN, 1989.

[3] E. Anderson and J. Dongarra. Evaluating block algorithm variants in LAPACK. Technical Report LAPACK working note 19, Computer Science Department, University of Tennessee, Knoxville, TN, 1990.

[4] C. C. Ashcraft. The distributed solution of linear systems using the torus wrap data mapping. Engineering Computing and Analysis Technical Report ECA-TR-147, Boeing Computer Services, 1990.

[5] C. C. Ashcraft. A taxonamy of distributed dense LU factorization methods. Engineering Computing and Analysis Technical Report ECA-TR-161, Boeing Computer Services, 1991.

[6] M. Barnett, D. G. Payne, and R. van de Geijn. Broadcasting on meshes with worm-hole routing. Technical report, Department of Computer Science, University of Texas at Austin, April 1993. Submitted to Supercomputing '93.

[7] W. S. Brainerd, C. H. Goldbergs, and J. C. Adams. *Programmers Guide to Fortran 90*. McGraw-Hill, New York, 1990.

[8] R. P. Brent. The LINPACK benchmark for the Fujitsu AP 1000. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 128–135. IEEE Computer Society Press, 1992.

[9] R. P. Brent. The LINPACK benchmark on the AP 1000: Preliminary report. In *Proceedings of the 2nd CAP Workshop*, NOV 1991.

[10] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Computer Society Press, 1992.

[11] J. Choi, J. J. Dongarra, and D. W. Walker. The design of scalable software libraries for distributed memory concurrent computers. In J. J. Dongarra and B. Tourancheau, editors,

Environments and Tools for Parallel Scientific Computing. Elsevier Science Publishers, 1993.

[12] E. Chu and A. George. Gaussian elimination with partial pivoting and load balancing on a multiprocessor. *Parallel Computing*, 5:65–74, 1987.

[13] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Introduction to Split-C: Version 0.9. Technical report, Computer Science Division – EECS, University of California, Berkeley, CA 94720, February 1993.

[14] J. Demmel. LAPACK: A portable linear algebra library for supercomputers. In *Proceedings of the 1989 IEEE Control Systems Society Workshop on Computer-Aided Control System Design*, December 1989.

[15] J. J. Dongarra. Increasing the performance of mathematical software through high-level modularity. In *Proc. Sixth Int. Symp. Comp. Methods in Eng. & Applied Sciences, Versailles, France*, pages 239–248. North-Holland, 1984.

[16] J. J. Dongarra. LAPACK Working Note 34: Workshop on the BLACS. Computer Science Dept. Technical Report CS-91-134, University of Tennessee, Knoxville, TN, May 1991. (LAPACK Working Note #34).

[17] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[18] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An extended set of Fortran basic linear algebra subroutines. *ACM Transactions on Mathematical Software*, 14(1):1–17, March 1988.

[19] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM Publications, Philadelphia, PA, 1991.

[20] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, July 1987.

[21] J. J. Dongarra, R. Hempel, A. J. G. Hey, and D. W. Walker. A proposal for a user-level message passing interface in a distributed memory environment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[22] J. J. Dongarra, Peter Mayes, and Giuseppe Radicati di Brozolo. The IBM RISC System/6000 and linear algebra operations. *Supercomputer*, 44(VIII-4):15–30, 1991.

[23] J. J. Dongarra and S. Ostrouchov. LAPACK block factorization algorithms on the Intel iPSC/860. Technical Report CS-90-115, University of Tennessee at Knoxville, Computer Science Department, October 1990.

[24] J. J. Dongarra, R. Pozo, and D. W. Walker. An object oriented design for high performance linear algebra on distributed memory architectures. In *Proceedings of the Object Oriented Numerics Conference*, 1993.

[25] J. J. Dongarra, R. van de Geijn, and D. W. Walker. A look at scalable dense linear algebra libraries. In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference*, pages 372–379. IEEE Publishers, 1992.

[26] J. J. Dongarra and R. A. van de Geijn. Two-dimensional basic linear algebra communication subprograms. Technical Report LAPACK working note 37, Computer Science Department, University of Tennessee, Knoxville, TN, October 1991.

[27] J. J. Dongarra and R. A. van de Geijn. Reduction to condensed form for the eigenvalue problem on distributed memory architectures. *Parallel Computing*, 18:973–982, 1992.

[28] T. H. Dunigan. Communication performance of the Intel Touchstone Delta mesh. Technical Report TM-11983, Oak Ridge National Laboratory, January 1992.

[29] A. Edelman. Large dense numerical linear algebra in 1993: The parallel computing influence. *International Journal Supercomputer Applications*, 1993. Accepted for publication.

[30] E. W. Felten and S. W. Otto. Coherent parallel C. In G. C. Fox, editor, *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 440–450. ACM Press, 1988.

[31] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J., 1988.

[32] K. Gallivan, R. Plemmons, and A. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, 1990.

[33] A. Geist and M. Heath. Matrix factorization on a hypercube multiprocessor. In M. Heath, editor, *Hypercube Multiprocessors, 1986*, pages 161–180, Philadelphia, PA, 1986. Society for Industrial and Applied Mathematics.

[34] A. Geist and C. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM J. Sci. Statist. Comput.*, 9(4):639–649, July 1988.

[35] R. Harrington. Origin and development of the method of moments for field computation. *IEEE Antennas and Propagation Magazine*, June 1990.

[36] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix computations on massively parallel computers. Technical Report SAND92-0792, Sandia National Laboratories, April 1992.

[37] J. L. Hess. Panel methods in computational fluid dynamics. *Annual Reviews of Fluid Mechanics*, 22:255–274, 1990.

[38] J. L. Hess and M. O. Smith. Calculation of potential flows about arbitrary bodies. In D. Küchemann, editor, *Progress in Aeronautical Sciences, Volume 8*. Pergamon Press, 1967.

[39] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, January 1993.

[40] R. W. Hockney and C. R. Jesshope. *Parallel Computers*. Adam Hilger Ltd., Bristol, UK, 1981.

[41] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5:308–323, 1979.

[42] W. Lichtenstein and S. L. Johnsson. Block-cyclic dense linear algebra. Technical Report TR-04-92, Harvard University, Center for Research in Computing Technology, January 1992.

[43] M. Lin, D. Du, A. E. Klietz, and S. Saroff. Performance evaluation of the CM-5 interconnection network. Technical report, Department of Computer Science, University of Minnesota, 1992.

[44] R. Ponnusamy, A. Choudhary, and G. Fox. Communication overhead on CM-5: An experimental performance evaluation. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 108–115. IEEE Computer Society Press, 1992.

[45] Y. Saad and M. H. Schultz. Parallel direct methods for solving banded linear systems. Technical Report YALEU/DCS/RR-387, Department of Computer Science, Yale University, 1985.

[46] S. R. Seidel. Broadcasting on linear arrays and meshes. Technical Report TM-12356, Oak Ridge National Laboratory, April 1993.

[47] A. Skjellum and A. Leung. LU factorization of sparse, unsymmetric, Jacobian matrices on multicomputers. In D. W. Walker and Q. F. Stout, editors, *Proceedings of the Fifth Distributed Memory Concurrent Computing Conference*, pages 328–337. IEEE Press, 1990.

[48] R. A. van de Geijn. Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems. Computer Science report TR-91-28, Univ. of Texas, 1991.

[49] E. F. Van de Velde. Data redistribution and concurrency. *Parallel Computing*, 16, December 1990.

[50] J. J. H. Wang. *Generalized Moment Methods in Electromagnetics.* John Wiley & Sons, New York, 1991.

[51] J. Wilkinson and C. Reinsch. *Handbook for Automatic Computation: Volume II - Linear Algebra.* Springer-Verlag, New York, 1971.