ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance [*] (Technical Paper)

L. S. Blackford[†], J. Choi[‡], A. Cleary[§], J. Demmel[¶], I. Dhillon[¶], J. Dongarra[‖], S. Hammarling[**], G. Henry[††], A. Petitet[§], K. Stanley[¶], D. Walker[‡‡] and R. C. Whaley[§]

Abstract

This paper outlines the content and performance of ScaLAPACK, a collection of mathematical software for linear algebra computations on distributed memory computers. The importance of developing standards for computational and message passing interfaces is discussed. We present the different components and building blocks of ScaLAPACK, and indicate the difficulties inherent in producing correct codes for networks of heterogeneous processors. Finally, this paper briefly describes future directions for the ScaLAPACK library and concludes by suggesting alternative approaches to mathematical libraries, explaining how ScaLAPACK could be integrated into efficient and user-friendly distributed systems.

Keywords: parallel computing, numerical linear algebra, math libraries.

[*]This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400; and by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615.

[†](formerly L. S. Ostrouchov) Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301

[‡]Soongsil University, Seoul, Korea

[§]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301

[¶]Computer Science Division, University of California, Berkeley, Berkeley, CA 94720

[‖]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831

[**]Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, and NAG Ltd, England

[††]Intel SSPD, 15201 NW Greenbrier Pkwy., Bldg CO1-01, Beaverton OR 97006-5733

[‡‡]Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831

Contents

1 Overview and Motivation

ScaLAPACK is a library of high performance linear algebra routines for distributed memory MIMD machines. It is a continuation of the LAPACK project, which has designed and produced an efficient linear algebra library for workstations, vector supercomputers and shared memory parallel computers [1]. Both libraries contain routines for the solution of systems of linear equations, linear least squares problems and eigenvalue problems. The goals of the LAPACK project, which continue into the ScaLAPACK project, are efficiency so that the computationally intensive routines execute as fast as possible; scalability as the problem size and number of processors grow; reliability, including the return of error bounds; portability across machines; flexibility so that users may construct new routines from well designed components; and ease of use. Towards this last goal the ScaLAPACK software has been designed to look as much like the LAPACK software as possible.

Many of these goals have been attained by developing and promoting standards, especially specifications for basic computational and communication routines. Thus LAPACK relies on the BLAS [26, 15, 14], particularly the Level 2 and 3 BLAS for computational efficiency, and ScaLAPACK [32] relies upon the BLACS [18] for efficiency of communication and uses a set of parallel BLAS, the PBLAS [9], which themselves call the BLAS and the BLACS. LAPACK and ScaLAPACK will run on any machines for which the BLAS and the BLACS are available. A PVM [20] version of the BLACS has been available for some time and the portability of the BLACS has recently been further increased by the development of a version that uses MPI [33].

The first part of this paper presents the design of ScaLAPACK. After a brief discussion of the BLAS and LAPACK, the block cyclic data layout, the BLACS, the PBLAS, and the algorithms used are discussed. We also outline the difficulties encountered in producing correct code for networks of heterogeneous processors; difficulties that we believe are little recognized by other practitioners.

The paper then discusses the performance of ScaLAPACK. Extensive results on various platforms are presented. One of our goals is to model and predict the performance of each routine as a function of a few problem and machine parameters. One interesting result is that for some algorithms, speed is *not* a monotonic increasing function of the number of processors. In other words, it can sometimes be beneficial to let some processors remain idle. Finally, we look at possible future directions and give some concluding remarks.

2 Design of ScaLAPACK

2.1 Portability, Scalability and Standards

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 is the use of block-partitioned algorithms, particularly in conjunction with highly-tuned kernels for performing matrix-vector and matrix-matrix operations (BLAS). In general, block-partitioned algorithms require the movement of blocks, rather than vectors or scalars, resulting in a greatly reduced startup cost because fewer messages are exchanged.

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 grid.

In order to be truly portable, the building blocks underlying parallel software libraries must be *standardized*. The definition of computational and message-passing standards [26, 15, 14, 33] provides vendors with a clearly defined base set of routines that they can optimize. From the user's point of view, standards ensure portability. As new machines are developed, they may simply be added to the network, supplying cycles as appropriate.

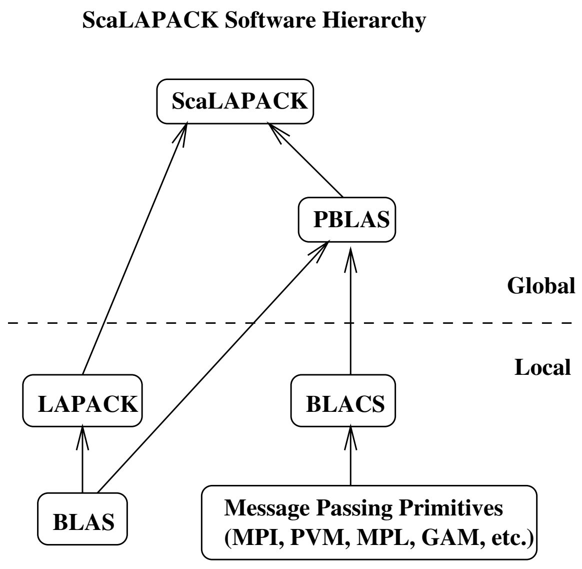

ScaLAPACK Software Hierarchy
ScaLAPACK
PBLAS
Global
Local
LAPACK
BLACS
BLAS
Message Passing Primitives
(MPI, PVM, MPL, GAM, etc.)

Figure 1: ScaLAPACK Software Hierarchy

From the mathematical software developer's point of view, portability may require significant effort. Standards permit the effort of developing and maintaining bodies of mathematical software to be leveraged over as many different computer systems as possible. Given the diversity of parallel architectures, portability is attainable to only a limited degree, but machine dependencies can at least be isolated.

Scalability demands that a program be reasonably effective over a wide range of numbers of processors. The scalability of parallel algorithms over a range of architectures and numbers of processors requires that the granularity of computation be adjustable. To accomplish this, we use block-partitioned algorithms with adjustable block sizes. Eventually, however, polyalgorithms (where the actual algorithm is selected at runtime depending on input data and machine parameters) may be required.

Scalable parallel architectures of the future are likely to use physically distributed memory. In the longer term, progress in hardware development, operating systems, languages, compilers, and communication systems may make it possible for users to view such distributed architectures (without significant loss of efficiency) as having a shared memory with 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.), are essential to the development of portable scalable libraries. 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.

2.2 ScaLAPACK Software Components

Figure 1 describes the ScaLAPACK software hierarchy. The components below the dashed line, labeled Local, are called on a single processor, with arguments stored on single processors only. The components above the line, labeled Global, are synchronous parallel routines, whose arguments include matrices and vectors distributed in a 2D block cyclic layout across multiple processors. We describe each component in turn.

2.3 Processes versus Processors

In ScaLAPACK, algorithms are presented in terms of *processes*, rather than physical processors. In general there may be several processes on a processor, in which case we assume that the runtime system handles the scheduling of processes. In the absence of such a runtime system, ScaLAPACK assumes one process per processor.

2.4 Local Components

The BLAS (Basic Linear Algebra Subprograms) [14, 15, 26] include subroutines for common linear algebra computations such as dot-products, matrix-vector multiplication, and matrix-matrix multiplication. As is well known, using matrix-matrix multiplication tuned for a particular architecture can effectively mask the effects of the memory hierarchy (cache misses, TLB misses, etc.), and permit floating point operations to be performed at the top speed of the machine.

As mentioned before, LAPACK, or Linear Algebra PACKage [1], is a collection of routines for linear system solving, linear least squares problems, and eigenproblems. High performance is attained by using algorithms that do most of their work in calls to the BLAS, with an emphasis on matrix-matrix multiplication. Each routine has one or more *performance tuning parameters*, such as the sizes of the blocks operated on by the BLAS. These parameters are machine dependent, and are obtained from a table at run-time.

The LAPACK routines are designed for single processors. LAPACK can also accommodate shared memory machines, provided parallel BLAS are available (in other words, the only parallelism is implicit in calls to BLAS). Extensive performance results for LAPACK can be found in the second edition of the users' guide [1].

The BLACS (Basic Linear Algebra Communication Subprograms) [18] are a message passing library designed for linear algebra. The computational model consists of a one or two dimensional grid of processes, where each process stores matrices and vectors. The BLACS include synchronous send/receive routines to send a matrix or submatrix from one process to another, to broadcast submatrices to many processes, or to compute global reductions (sums, maxima and minima). There are also routines to construct, change, or query the process grid. Since several ScaLAPACK algorithms require broadcasts or reductions among different subsets of processes, the BLACS permit a processor to be a member of several overlapping or disjoint process grids, each one labeled by a *context*. Some message passing systems, such as MPI [27, 33], also include this context concept. (MPI calls this a communicator.) The BLACS provide facilities for safe interoperation of system contexts and BLACS contexts.

2.5 Block Cyclic Data Distribution

On a distributed memory computer the application programmer is responsible for decomposing the data over the processes of the computer. The way in which a matrix is distributed over the processes has a major impact on the load balance and communication characteristics of the concurrent algorithm, and hence largely determines its performance and scalability. The current implementation of ScaLAPACK assumes the matrices to be distributed according to the block-cyclic decomposition scheme. The block cyclic distribution provides a simple, yet general-purpose way of distributing a block-partitioned matrix on distributed memory concurrent computers. The High Performance Fortran standard [23, 25] provides a block cyclic data distribution as one of the basic data distributions.

Assuming a two-dimensional block cyclic data distribution, an `M` by `N` matrix is first decomposed into `MB` by `NB` blocks starting at its upper left corner. These blocks are then uniformly distributed across the process grid. Thus every process owns a collection of blocks, which are locally and contiguously stored in a two dimensional "column major" array. We present in Fig. 2 the partitioning of a 9×9 matrix into 2×2 blocks. Then in Fig. 3 we show how these 2×2 blocks are mapped onto a 2×3 process grid, i.e.,

$M = N = 9$ and $MB = NB = 2$. The local entries of every matrix column are contiguously stored in the processes' memories.

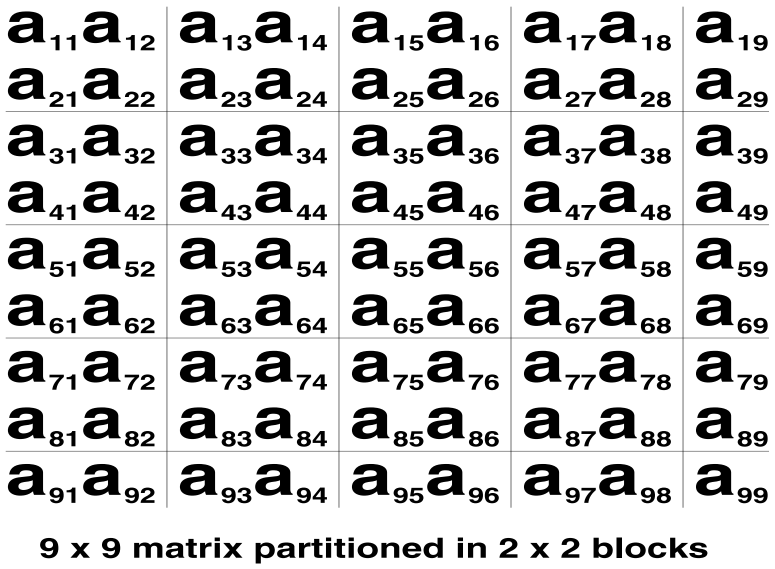

$a_{11}a_{12}$ $a_{13}a_{14}$ $a_{15}a_{16}$ $a_{17}a_{18}$ a_{19}
$a_{21}a_{22}$ $a_{23}a_{24}$ $a_{25}a_{26}$ $a_{27}a_{28}$ a_{29}
$a_{31}a_{32}$ $a_{33}a_{34}$ $a_{35}a_{36}$ $a_{37}a_{38}$ a_{39}
$a_{41}a_{42}$ $a_{43}a_{44}$ $a_{45}a_{46}$ $a_{47}a_{48}$ a_{49}
$a_{51}a_{52}$ $a_{53}a_{54}$ $a_{55}a_{56}$ $a_{57}a_{58}$ a_{59}
$a_{61}a_{62}$ $a_{63}a_{64}$ $a_{65}a_{66}$ $a_{67}a_{68}$ a_{69}
$a_{71}a_{72}$ $a_{73}a_{74}$ $a_{75}a_{76}$ $a_{77}a_{78}$ a_{79}
$a_{81}a_{82}$ $a_{83}a_{84}$ $a_{85}a_{86}$ $a_{87}a_{88}$ a_{89}
$a_{91}a_{92}$ $a_{93}a_{94}$ $a_{95}a_{96}$ $a_{97}a_{98}$ a_{99}
9 x 9 matrix partitioned in 2 x 2 blocks

Figure 2: Matrix partitioning

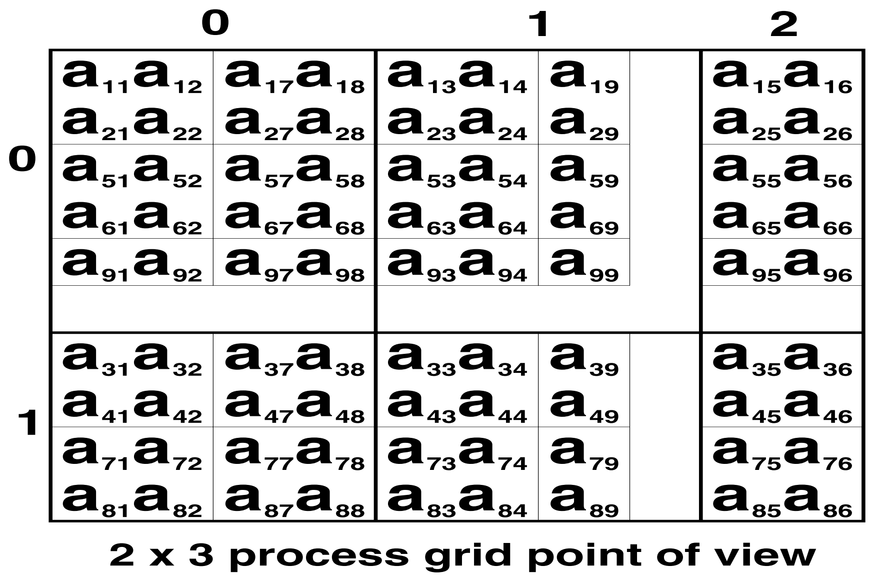

0
1
2
0
$a_{11}a_{12}$ $a_{17}a_{18}$ $a_{13}a_{14}$ a_{19} $a_{15}a_{16}$
$a_{21}a_{22}$ $a_{27}a_{28}$ $a_{23}a_{24}$ a_{29} $a_{25}a_{26}$
$a_{51}a_{52}$ $a_{57}a_{58}$ $a_{53}a_{54}$ a_{59} $a_{55}a_{56}$
$a_{61}a_{62}$ $a_{67}a_{68}$ $a_{63}a_{64}$ a_{69} $a_{65}a_{66}$
$a_{91}a_{92}$ $a_{97}a_{98}$ $a_{93}a_{94}$ a_{99} $a_{95}a_{96}$
1
$a_{31}a_{32}$ $a_{37}a_{38}$ $a_{33}a_{34}$ a_{39} $a_{35}a_{36}$
$a_{41}a_{42}$ $a_{47}a_{48}$ $a_{43}a_{44}$ a_{49} $a_{45}a_{46}$
$a_{71}a_{72}$ $a_{77}a_{78}$ $a_{73}a_{74}$ a_{79} $a_{75}a_{76}$
$a_{81}a_{82}$ $a_{87}a_{88}$ $a_{83}a_{84}$ a_{89} $a_{85}a_{86}$
2 x 3 process grid point of view

Figure 3: Mapping of matrix onto 2×3 process grid

For further details on data distributions, refer to [17].

2.6 PBLAS

In order to simplify the design of ScaLAPACK, and because the BLAS have proven to be very useful tools outside LAPACK, we chose to build a set of Parallel BLAS, or PBLAS, whose interface is as similar to the BLAS as possible. This decision has permitted the ScaLAPACK code to be quite similar, and sometimes nearly identical, to the analogous LAPACK code. Only one substantially new routine was added to the PBLAS, matrix transposition, since this is a complicated operation in a distributed memory environment [11].

We hope that the PBLAS will provide a distributed memory standard, just as the BLAS have provided a shared memory standard. This would simplify and encourage the development of high performance and portable parallel numerical software, as well as providing manufacturers with a small set of routines to be optimized. The acceptance of the PBLAS requires reasonable compromises among competing goals of functionality and simplicity. These issues are discussed below.

The PBLAS operate on matrices distributed in a 2D block cyclic layout. Since such a data layout requires many parameters to fully describe the distributed matrix, we have chosen a more object-oriented approach, and encapsulated these parameters in an integer array called an *array descriptor* which is passed to the PBLAS. An array descriptor includes

(1) the descriptor type,
(2) the BLACS context,
(3) the number of rows in the distributed matrix,
(4) the number of columns in the distributed matrix,
(5) the row block size (MB in section 2.5),
(6) the column block size (NB in section 2.5),
(7) the process row over which the first row of the matrix is distributed,
(8) the process column over which the first column of the matrix is distributed,
(9) the leading dimension of the local array storing the local blocks.

Below is an example of a call to the BLAS double precision matrix multiplication routine DGEMM, and the corresponding PBLAS routine PDGEMM; note how similar they are:

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

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

DGEMM computes $C = \beta * C + \alpha * op(A) * op(B)$, where $op(A)$ is either A or its transpose depending on $TRANSA$, $op(B)$ is similar, $op(A)$ is M-by-K, and $op(B)$ is K-by-N. PDGEMM is the same, with the exception of the way in which submatrices are specified. To pass the submatrix starting at $A(IA, JA)$ to DGEMM, for example, the actual argument corresponding to the formal argument A would simply be $A(IA, JA)$. PDGEMM, on the other hand, needs to understand the global storage scheme of A to extract the correct submatrix, so IA and JA must be passed in separately. $DESC\_A$ is the array descriptor for A. The parameters describing the matrix operands B and C are analogous to those describing A. In a truly object-oriented environment, matrices and $DESC\_A$ would be synonymous. However, this would require language support, and detract from portability.

Our implementation of the PBLAS emphasizes the mathematical view of a matrix over its storage. In fact, it is even possible to reuse our interface to implement the PBLAS for a different block data distribution that would not fit in the block-cyclic scheme (this is planned for future releases of ScaLAPACK).

The presence of a context associated with every distributed matrix provides the ability to have separate "universes" of message passing. The use of separate communication contexts by distinct libraries (or distinct library invocations) such as the PBLAS insulates communication internal to the library from external communication. When more than one descriptor array is present in the argument list of a routine in the PBLAS, it is required that the individual BLACS context entries must be equal. In other words, the PBLAS do not perform "inter-context" operations.

The PBLAS do not included specialized routines to take advantage of packed storage schemes for symmetric, Hermitian, or triangular matrices, nor of compact storage schemes for banded matrices.

2.7 LAPACK/ScaLAPACK code comparison

Given the infrastructure described above, the ScaLAPACK version (PDGETRF) of the LU decomposition is nearly identical to its LAPACK version (DGETRF), as illustrated in Figure 4.

3 ScaLAPACK – Content

The ScaLAPACK library includes routines for the solution of linear systems of equations, symmetric positive definite banded linear systems of equations, condition estimation and iterative refinement, for LU and Cholesky factorization, matrix inversion, full-rank linear least squares problems, orthogonal and generalized orthogonal factorizations, orthogonal transformation routines, reductions to upper Hessenberg, bidiagonal and tridiagonal form, reduction of a symmetric-definite generalized eigenproblem to standard form, the symmetric, generalized symmetric and the nonsymmetric eigenproblem. Software is available in single precision real, double precision real, single precision complex, and double precision complex.

The subroutines in ScaLAPACK are classified as follows:

- **driver** routines, each of which solves a complete problem, for example solving a system of linear equations, or computing the eigenvalues of a real symmetric matrix. Users are recommended to use a driver routine if there is one that meets their requirements. Global and local input error-checking are performed for these routines.
- **computational** routines, each of which performs a distinct computational task, for example an LU factorization, or the reduction of a real symmetric matrix to tridiagonal form. Each driver routine calls a sequence of computational routines. Users (especially software developers) may need to call computational routines directly to perform tasks, or sequences of tasks, that cannot conveniently be performed by the driver routines. Global and local input error-checking are performed for these routines.
- **auxiliary** routines, which in turn can be classified as follows:
  - routines that perform subtasks of block-partitioned algorithms — in particular, routines that implement unblocked versions of the algorithms;
  - routines that perform some commonly required low-level computations, for example scaling a matrix, computing a matrix-norm, or generating an elementary Householder matrix; some of these may be of interest to numerical analysts or software developers and could be considered for future additions to the PBLAS;
  - a few extensions to the PBLAS, such as routines for matrix-vector operations involving complex symmetric matrices (the PBLAS themselves are not strictly speaking part of ScaLAPACK).

A draft ScaLAPACK Users' Guide [32] and a comprehensive Installation Guide is provided, as well as test suites for all ScaLAPACK, PBLAS, and BLACS routines.

SEQUENTIAL LU FACTORIZATION CODE

```
      DO 20 J = 1, MIN( M, N ), NB
         JB = MIN( MIN( M, N )-J+1, NB )


         Factor diagonal and subdiagonal blocks and test for exact
         singularity.

         CALL DGETF2( M-J+1, JB, A( J, J ), LDA, IPIV( J ),
     $                IINFO )

         Adjust INFO and the pivot indices.

         IF( INFO.EQ.0 .AND. IINFO.GT.0 ) INFO = IINFO + J - 1
         DO 10 I = J, MIN( M, J+JB-1 )
            IPIV( I ) = J - 1 + IPIV( I )
   10    CONTINUE

         Apply interchanges to columns 1:J-1.

         CALL DLASWP( J-1, A, LDA, J, J+JB-1, IPIV, 1 )


         IF( J+JB.LE.N ) THEN

            Apply interchanges to columns J+JB:N.

            CALL DLASWP( N-J-JB+1, A( 1, J+JB ), LDA, J, J+JB-1,
     $                   IPIV, 1 )

            Compute block row of U.

            CALL DTRSM( 'Left', 'Lower', 'No transpose', 'Unit',
     $                  JB, N-J-JB+1, ONE, A( J, J ), LDA,
     $                  A( J, J+JB ), LDA )
            IF( J+JB.LE.M ) THEN

               Update trailing submatrix.

               CALL DGEMM( 'No transpose', 'No transpose',
     $                     M-J-JB+1, N-J-JB+1, JB, -ONE,
     $                     A( J+JB, J ), LDA, A( J, J+JB ), LDA,
     $                     ONE, A( J+JB, J+JB ), LDA )
            END IF
         END IF
   20 CONTINUE
```

PARALLEL LU FACTORIZATION CODE

```
      DO 10 J = JA, JA+MIN(M,N)-1, DESCA( 6 )
         JB = MIN( MIN(M,N)-J+JA, DESCA( 6 ) )
         I = IA + J - JA

         Factor diagonal and subdiagonal blocks and test for exact
         singularity.

         CALL PDGETF2( M-J+JA, JB, A, I, J, DESCA, IPIV, IINFO )


         Adjust INFO and the pivot indices.

         IF( INFO.EQ.0 .AND. IINFO.GT.0 )
     $      INFO = IINFO + J - JA



         Apply interchanges to columns JA:J-JA.

         CALL PDLASWP( 'Forward', 'Rows', J-JA, A, IA, JA, DESCA,
     $                 J, J+JB-1, IPIV )

         IF( J-JA+JB+1.LE.N ) THEN

            Apply interchanges to columns J+JB:JA+N-1.

            CALL PDLASWP( 'Forward', 'Rows', N-J-JB+JA, A, IA,
     $                    J+JB, DESCA, J, J+JB-1, IPIV )

            Compute block row of U.

            CALL PDTRSM( 'Left', 'Lower', 'No transpose', 'Unit',
     $                   JB, N-J-JB+JA, ONE, A, I, J, DESCA, A, I,
     $                   J+JB, DESCA )
            IF( J-JA+JB+1.LE.M ) THEN

               Update trailing submatrix.

               CALL PDGEMM( 'No transpose', 'No transpose',
     $                      M-J-JB+JA, N-J-JB+JA, JB, -ONE, A,
     $                      I+JB, J, DESCA, A, I, J+JB, DESCA,
     $                      ONE, A, I+JB, J+JB, DESCA )
            END IF
         END IF
   10 CONTINUE
```

Figure 4: Comparison of LU factorization in LAPACK and ScaLAPACK

3.1 Linear Equations

We use the standard notation for a system of simultaneous linear equations:

$$Ax = b \tag{1}$$

where A is the **coefficient matrix**, b is the **right hand side**, and x is the **solution**. In (1) A is assumed to be a square matrix of order n, but some of the individual routines allow A to be rectangular. If there are several right hand sides, we write

$$AX = B \tag{2}$$

where the columns of B are the individual right hand sides, and the columns of X are the corresponding solutions. The basic task is to compute X, given A and B.

If A is upper or lower triangular, (1) can be solved by a straightforward process of backward or forward substitution. Otherwise, the solution is obtained after first factorizing A as a product of triangular matrices (and possibly also a diagonal matrix or permutation matrix).

The form of the factorization depends on the properties of the matrix A. ScaLAPACK provides routines for the following types of matrices, based on the stated factorizations:

- **general** matrices (LU factorization with partial pivoting):

  $$A = PLU$$

  where P is a permutation matrix, L is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$), and U is upper triangular (upper trapezoidal if $m < n$).
- **general band** matrices including **tridiagonal** matrices
  - (LU factorization with partial pivoting): If A is m-by-n with kl subdiagonals and ku superdiagonals, the factorization is

    $$A = LU$$

    where L is a product of permutation and unit lower triangular matrices with kl subdiagonals, and U is upper triangular with $kl + ku$ superdiagonals.
  - (LU factorization without pivoting) for matrices where it is known *a priori* that pivoting is not necessary for numerical stability, such as diagonally dominant matrices.
- **symmetric and Hermitian positive definite** matrices including **band** matrices (Cholesky factorization):

  $$A = U^T U \text{ or } A = LL^T \text{ (in the symmetric case)}$$

  $$A = U^H U \text{ or } A = LL^H \text{ (in the Hermitian case)}$$

  where U is an upper triangular matrix and L is lower triangular.
- **symmetric and Hermitian positive definite tridiagonal** matrices (LDL^T factorization):

  $$A = UDU^T \text{ or } A = LDL^T \text{ (in the symmetric case)}$$

  $$A = UDU^H \text{ or } A = LDL^H \text{ (in the Hermitian case)}$$

  where U is a unit upper bidiagonal matrix, L is unit lower bidiagonal, and D is diagonal.

The factorization for a general tridiagonal matrix is like that for a general band matrix with $kl = 1$ and $ku = 1$. The factorization for a symmetric positive definite band matrix with k superdiagonals (or subdiagonals) has the same form as for a symmetric positive definite matrix, but the factor U (or L) is a band matrix with k superdiagonals (subdiagonals). Band matrices use the band storage scheme described in section 3.2.

While the primary use of a matrix factorization is to solve a system of equations, other related tasks are provided as well.

3.2 Band Decomposition

Band matrix storage in ScaLAPACK follows the conventions of LAPACK. In LAPACK, an m-by-n band matrix with kl subdiagonals and ku superdiagonals may be stored compactly in a two-dimensional array with at least $kl + ku + 1$ rows and n columns. Columns of the matrix are stored in corresponding columns of the array, and diagonals of the matrix are stored in rows of the array. The same array layout is specified in ScaLAPACK except the array is divided across the processes.

The optimal algorithm for solving banded (and as a special case, tridiagonal) linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case $N/P \gg kl, ku$ are implemented. The general family of algorithms of this sort is variously known as partitioning methods, domain decomposition methods, or divide and conquer methods. A prototypical algorithm of this form is described in [16].

Partitioning methods utilize large-grained task parallelism, in contrast with the algorithms used in dense ScaLAPACK which are fine-grained. As such, the appropriate decomposition for these algorithms differs from the dense square matrix case, and a one-dimensional blocked decomposition is used.

The bottleneck that has traditionally limited the effectiveness of partitioning methods is the solution of the reduced system that represents the interaction of the systems stored on each process. Many implementations have adopted what are essentially sequential algorithms for the reduced system, but this detracts from scalability. For ScaLAPACK we have implemented a block odd-even reduction algorithm whose running time scales optimally.

3.3 Orthogonal Factorizations and Linear Least Squares Problems

ScaLAPACK provides a number of routines for factorizing a general rectangular m-by-n matrix A, as the product of an **orthogonal** matrix (**unitary** if complex) and a **triangular** (or possibly trapezoidal) matrix.

A real matrix Q is **orthogonal** if $Q^T Q = I$; a complex matrix Q is **unitary** if $Q^H Q = I$. Orthogonal or unitary matrices have the important property that they leave the two-norm of a vector invariant:

$$||x||_2 = ||Qx||_2, \quad \text{if } Q \text{ is orthogonal or unitary.}$$

As a result, they help to maintain numerical stability because they do not amplify rounding errors.

Orthogonal factorizations are used in the solution of linear least squares problems. They may also be used to perform preliminary steps in the solution of eigenvalue or singular value problems.

3.4 Generalized Orthogonal Factorizations and Linear Least Squares Problems

ScaLAPACK includes routines for generalized QR and RQ factorizations. Future releases of ScaLAPACK will include routines for solving generalized linear least squares problems.

3.5 Symmetric Eigenproblems

Let A be a real symmetric or complex Hermitian n-by-n matrix. A scalar λ is called an **eigenvalue** and a nonzero column vector z the corresponding **eigenvector** if $Az = \lambda z$. λ is always real when A is real symmetric or complex Hermitian.

The basic task of the symmetric eigenproblem routines is to compute values of λ and, optionally, corresponding vectors z for a given matrix A.

This computation proceeds in the following stages:

1. The real symmetric or complex Hermitian matrix A is reduced to **real tridiagonal form** T. If A is real symmetric this decomposition is $A = QTQ^T$ with Q orthogonal and T symmetric tridiagonal. If A is complex Hermitian, the decomposition is $A = QTQ^H$ with Q unitary and T, as before, *real* symmetric tridiagonal.

2. Eigenvalues and eigenvectors of the real symmetric tridiagonal matrix T are computed. If all eigenvalues and eigenvectors are computed, this is equivalent to factorizing T as $T = S\Lambda S^T$, where S is orthogonal and Λ is diagonal. The diagonal entries of Λ are the eigenvalues of T, which are also the eigenvalues of A, and the columns of S are the eigenvectors of T; the eigenvectors of A are the columns of $Z = QS$, so that $A = Z\Lambda Z^T$ ($Z\Lambda Z^H$ when A is complex Hermitian).

The solution of the symmetric eigenproblem implemented in PDSYEVX consists of three phases: (1) reduce the original matrix A to tridiagonal form $A = QTQ^T$ where Q is orthogonal and T is tridiagonal, (2) find the eigenvalues $\Lambda = \text{diag}(\lambda_1, ..., \lambda_n)$ and eigenvectors $U = [u_1, ..., u_n]$ of T so that $T = U\Lambda U^T$, and (3) form the eigenvector matrix V of A so $A = Q(U\Lambda U^T)Q^T = (QU)\Lambda(QU)^T = V\Lambda V^T$. Phases 1 and 3 are analogous to their LAPACK counterparts. However, our current design for phase 2 differs from the serial (or shared memory) design. We have chosen to do bisection followed by inverse iteration (like the LAPACK expert driver DSYEVX), but with the reorthogonalization phase of inverse iteration limited to the eigenvectors stored in a single process. A straightforward parallelization of DSYEVX would have led to a serial bottleneck and significant slowdowns in the rare situation of matrices with eigenvalues tightly clustered together. The current design guarantees that phase (2) is inexpensive compared to the other phases once problems are reasonably large. An alternative algorithm which eliminates the need for reorthogonalization is currently being developed by Dhillon and Parlett using Fernando's double factorization method [31], and we expect to use this new algorithm in the near future. This new routine should guarantee high accuracy and high speed independent of the eigenvalue distribution.

3.6 Nonsymmetric Eigenproblem and Schur Factorization

Let A be a square n-by-n matrix. A scalar λ is called an **eigenvalue** and a non-zero column vector v the corresponding **right eigenvector** if $Av = \lambda v$. A nonzero column vector u satisfying $u^H A = \lambda u^H$ is called the **left eigenvector**. The first basic task of the routines described in this section is to compute, for a given matrix A, all n values of λ and, if desired, their associated right eigenvectors v and/or left eigenvectors u.

A second basic task is to compute the **Schur factorization** of a matrix A. If A is complex, then its Schur factorization is $A = ZTZ^H$, where Z is unitary and T is upper triangular. If A is real, its Schur factorization is $A = ZTZ^T$, where Z is orthogonal. and T is upper quasi-triangular (1-by-1 and 2-by-2 blocks on its diagonal). The columns of Z are called the **Schur vectors** of A. The eigenvalues of A appear on the diagonal of T; complex conjugate eigenvalues of a real A correspond to 2-by-2 blocks on the diagonal of T.

We are implementing two parallel algorithms for computing the Schur factorization. The first is a parallelization of the conventional sequential algorithm, and the second is a novel divide-and-conquer algorithm. We discuss each briefly in turn.

The ScaLAPACK solution has elements in common with LAPACK. For example, both reduce to upper Hessenberg form initially. Both use an implicit QR algorithm based approach. The QR algorithm involves bulge chasing, and LAPACK uses a single bulge. Parallelism is obtained in ScaLAPACK by chasing multiple bulges that are staggered far enough apart so that all nodes have computation to perform (except during pipeline start-up and wind-down.) The broadcast of the Householder reflections which is critical to any QR algorithm is blocked independently of the distributed block size. The application of the Householder reflections are also done in a block fashion (independent of the above broadcast blocking) to increase data re-use.

Our second algorithm is based on the *sign function* of a matrix [3, 4, 5]. This algorithm offers the flexibility of computing just part of the Schur form (corresponding, say, to the eigenvalues with positive real parts), uses more highly parallelized building blocks from ScaLAPACK than the HQR algorithm in the last paragraph (matrix inversion, matrix multiply and QR factorization), but also requires more floating point operations; it remains to be seen for which problems and on which machines which algorithm is faster. The sign function also entails a dynamic load balancing scheme [7] to implement its divide-and-conquer approach most efficiently.

3.7 Singular Value Decomposition

Let A be a general real m-by-n matrix. The **singular value decomposition (SVD)** of A is the factorization $A = U\Sigma V^T$, where U and V are orthogonal, and $\Sigma = \text{diag}(\sigma_1, \ldots \sigma_r)$, $r = \min(m, n)$, with $\sigma_1 \geq \cdots \geq \sigma_r \geq 0$. If A is complex, then its SVD is $A = U\Sigma V^H$ where U and V are unitary, and Σ is as before with real diagonal elements. The σ_i are called the **singular values**, the first r columns of V the **right singular vectors** and the first r columns of U the **left singular vectors**.

The SVD and symmetric eigendecompositions are entirely analogous, so that any algorithm for one has a counterpart for the other. As soon as the final version of the symmetric eigenvalue algorithm has been developed, we will produce an SVD version. In the meantime, we plan to release an SVD code based on serial QR iteration, where each processor redundantly runs QR iteration on a bidiagonal matrix, but updates a subset of the rows of the U and V in an embarrassingly parallel fashion.

3.8 Generalized Symmetric Definite Eigenproblems

The generalized eigenvalue problems are defined as $Az = \lambda Bz$, or $ABz = \lambda z$, or $BAz = \lambda z$, where A and B are real symmetric or complex Hermitian and B is positive definite. Each of these problems can be reduced to a standard symmetric eigenvalue problem, using a Cholesky factorization of B as either $B = LL^T$ or $B = U^TU$ (LL^H or U^HU in the Hermitian case).

With $B = LL^T$, we have

$$Az = \lambda Bz \quad \Rightarrow \quad (L^{-1}AL^{-T})(L^Tz) = \lambda(L^Tz).$$

Hence the eigenvalues of $Az = \lambda Bz$ are those of $Cy = \lambda y$, where C is the symmetric matrix $C = L^{-1}AL^{-T}$ and $y = L^Tz$. In the complex case C is Hermitian with $C = L^{-1}AL^{-H}$ and $y = L^Hz$.

Table 1 summarizes how each of the three types of problem may be reduced to standard form $Cy = \lambda y$, and how the eigenvectors z of the original problem may be recovered from the eigenvectors y of the reduced problem. The table applies to real problems; for complex problems, transposed matrices must be replaced by conjugate-transposes.

Table 1: Reduction of generalized symmetric definite eigenproblems to standard problems

	Type of problem	Factorization of B	Reduction	Recovery of eigenvectors
1.	$Az = \lambda Bz$	$B = LL^T$	$C = L^{-1}AL^{-T}$	$z = L^{-T}y$
		$B = U^TU$	$C = U^{-T}AU^{-1}$	$z = U^{-1}y$
2.	$ABz = \lambda z$	$B = LL^T$	$C = L^TAL$	$z = L^{-T}y$
		$B = U^TU$	$C = UAU^T$	$z = U^{-1}y$
3.	$BAz = \lambda z$	$B = LL^T$	$C = L^TAL$	$z = Ly$
		$B = U^TU$	$C = UAU^T$	$z = U^Ty$

4 Heterogeneous Networks

There are special challenges associated with writing reliable numerical software on networks containing heterogeneous processors. That is, processors which may do floating point arithmetic differently. This includes not just machines with completely different floating point formats and semantics (e.g. Cray versus workstations running IEEE standard floating point arithmetic), but even supposedly identical machines running with different compilers or even just different compiler options or runtime environment. The basic problem occurs when making *data dependent branches* on different processors. The flow of an algorithm is usually data dependent and so slight variations in the data may lead to different processors executing completely different sections of code.

A simple example of where an algorithm might not work correctly is an iteration where the stopping criterion depends on the value of the machine precision. If the precision varies from process to process, different processes may have significantly different stopping criteria. In particular, the stopping criterion used by the most accurate process may never be satisfied if it depends on data computed less accurately by other processes.

Many such problems can be eliminated by using the *largest* machine precision among all participating processes. In LAPACK routine `DLAMCH` returns the (double precision) machine precision (as well as other machine parameters). In ScaLAPACK this is replaced by `PDLAMCH` which returns the largest value over all the processes, replacing the uniprocessor value returned by `DLAMCH`. Similarly, one should use the smallest overflow threshold and largest underflow threshold over the processes being used. In a non-homogeneous environment the ScaLAPACK routine `PDLAMCH` runs the LAPACK routine `DLAMCH` on each process and computes the relevant maximum or minimum value. We refer to these machine parameters as the *multiprocessor machine parameters*.

It should be noted that if the code contains communication between processes within an iteration, it will not complete if one process converges before the others. In a heterogeneous environment, the only way to guarantee termination is to have one process make the convergence decision and broadcast that decision. Further problems and suggested solutions are discussed in [12, 6].

5 Performance

An important performance metric is *parallel efficiency*. Parallel efficiency, $E(N,P)$, for a problem of size N on P processors is defined in the usual way [19] as

$$E(N,P) = \frac{1}{P}\frac{T_{\text{seq}}(N)}{T(N,P)} \tag{3}$$

where $T(N,P)$ is the runtime of the parallel algorithm, and $T_{\text{seq}}(N)$ is the runtime of the best sequential algorithm. An implementation is said to be *scalable* if the efficiency is an increasing function of N/P, the problem size per processor (in the case of dense matrix computations, $N = n^2$, the number of words in the input).

We will also measure the *performance* of our algorithm in Megaflops/sec (or Gigaflops/sec). This is appropriate for large dense linear algebra computations, since floating point dominates communication. For a scalable algorithm with N/P held fixed, we expect the performance to be proportional to P.

We seek to increase the performance of our algorithms by reducing overhead due to load imbalance, data movement, and algorithm restructuring. The way the data are distributed over the memory hierarchy of a computer is of fundamental importance to these factors. We present in this section extensive performance results on various platforms for the ScaLAPACK factorization and reductions routines. Figures 5 and 6 show the performance of LU, Cholesky and QR factorizations for various sizes of matrices and number of processes on an Intel Paragon and IBM SP-2. The number of processes used in the experiments is specified by a process grid dimension cited next to each curve in the graph. As can be seen the algorithms show scalability as the size of the problem and the number of processes are increased.

Performance data for the symmetric positive definite banded solvers is presented in Figure 7. The execution time reflects three parameters: the execution time of sequential LAPACK banded routines on each process, the redundancy in operation count of a factor of four caused by fill-in, and the logarithmic execution time of the reduced system. In Figure 7, we graph execution rate as a function of problem size while holding the work per process constant. There are two curves in Figure 7: the top curve plots the actual computational speed of the algorithm using the operation count from the parallel band algorithm, while the bottom curve plots speed relative to the sequential operation count. The latter curve actually shows slower execution time as the number of processes increases from 1 to 2 and from 2 to 4 due to the increased operation count incurred by band partition algorithms due to fill-in. Note the almost linear scalability beyond P=4, with the slight drop-off due to the logarithmic complexity of the block odd-even reduction. The first curve drops mildly going from P=1 to P=2 and P=4 due to the introduction of communication versus the P=1 case.

Performance data for the symmetric eigensolver (PDSYEVX) are presented in [13].

The timings in Figure 8 represent a single super iteration (an iteration with multiple bulges) of the implicit multiple double shift nonsymmetric QR eigenvalue problem (PDLAHQR.) PDLAHQR maintains similiar accuracy as DLAHQR. Efficiencies are reported as compared to PDLAHQR run on a single node. Because PDLAHQR uses applies Householder transforms in a block fashion, it is faster than DLAHQR on a single node. We have seen speed-ups compared to DLAHQR on 144 Paragon nodes actually surpass 144 times the maximum performance of DLAHQR. In each case, a distribution block size between 50 and 125 was chosen, but these were blocked into smaller sets (12-30) for communication, and further blocked in sets of three for computation. The largest problems run were problems chosen to fill the same amount of memory, in this case, the Hessenberg matrix and Schur vectors and work buffers all fit into around 52 Mbytes. The work on the Schur vectors was also included. The number of processes used in the experiments is specified by a process grid dimension cited next to each curve in the graph. Actual experiments indicate that overall performance of the code, using standard flop count heuristics such as found in [21], is sharply underestimated by looking at a single super iteration.

5.1 Choice of Block Size

In the factorization or reduction routines, the work distribution becomes uneven as the computation progresses. A larger block size results in greater load imbalance, but reduces the frequency of communication between processes. There is, therefore, a tradeoff between load imbalance and communication startup cost, which can be controlled by varying the block size. This is illustrated in Figure 9 with a model of the computational time.

Most of the computation of the ScaLAPACK routines is performed in a blocked fashion using Level 3 BLAS, as is done in LAPACK. The computational blocking factor is chosen to be the same as the distribution block size. Therefore, smaller distribution block sizes increase the loop and index computation overhead. However, because the computation cost ultimately dominates, the influence of the block size on the overall communication startup cost and loop and index computation overhead decreases very rapidly with the problem size for a given grid of processes. Consequently, the performance of the ScaLAPACK library is not very sensitive to the block size, as long as the extreme cases are avoided. A very small block size leads to BLAS 2 operations and poorer performance. A very large block size leads to computational imbalance.

One exception is PDLAHQR (the nonsymmetric QR eigenvalue algorithm.) Here, the computational blocking factor has an upper bound given by the distribution block size. But the distribution block size corresponds to border communications, and so the algorithm desires block sizes larger than one might typically require for Level 3 based algorithms. A wise choice is therefore critical to performance and ultimately redistribution might be necessary for unwise choices.

The chosen block size impacts the amount of workspace needed on every process. This amount of workspace is typically large enough to contain a block of columns or a block of rows of the matrix operands. Therefore, the larger the block size, the greater the necessary workspace, i.e the smaller the largest solvable problem on a given grid of processes. For Level 3 BLAS blocked algorithms, the smallest possible block operands are of size $MB \times NB$. Therefore, it is good practice to choose the block size to be the problem size for which the BLAS matrix-multiply GEMM routine achieves 90 % of its reachable peak.

Determining optimal, or near optimal block sizes for different environments is a difficult task because it depends on many factors including the machine architecture, speeds of the different BLAS levels, the latency and bandwidth of message passing, the number of process available, the dimensions of the process grid, the dimension of the problem, and so on. However, there is enough evidence and expertise for automatically and accurately determining optimal, or near optimal block sizes via an enquiry routine. Furthermore, for small problem sizes it is also possible to determine if redistributing n^2 data items is an acceptable cost in terms of performance as well as memory usage. In the future, we hope to calculate the optimal block size via an enquiry routine.

5.2 Choice of Grid Size

The best grid shape is determined by the algorithm implemented in the library and the underlying physical network. A one link physical network will favor $P_r = 1$ or $P_c = 1$, where $P_r \times P_c$ is the process grid. This affects the scalability of the algorithm, but reduces the overhead due to message collisions. It is possible to predict the best grid shape given the number of processes available. The current algorithms for the factorization or reduction routines can be split into two categories.

If at every step of the algorithm a block of columns and/or rows needs to be broadcast, as in the LU or QR factorizations, it is possible to pipeline this communication phase and overlap it with some computation. The direction of the pipeline determines the shape of the grid. For example, the LU, QR and QL factorizations perform better for "flat" process grids ($P_r < P_c$). These factorizations share a common bottleneck of performing a reduction operation along each column (for pivoting in LU, and for computing a norm in QR and QL). The first implication of this observation is that large latency message passing perform better on a "flat" grid than on a square grid. Secondly, after this reduction has been performed, it is important to update the next block of columns as fast as possible. This is done by broadcasting the current block of columns using a ring topology, i.e, feeding the ongoing communication pipe. Similarly, the performance of the LQ and RQ factorizations take advantage of "tall" grids ($P_r > P_c$) for the same reasons, but transposed.

The theoretical efficiency of the LU factorization can be estimated by (3):

$$E_{LU}(N,P) = \frac{1}{1 + \frac{3P\log P_r}{n^2}\frac{\alpha}{\gamma_3} + \frac{3}{4n}(2P_c + P_r\log P_r)\frac{\beta}{\gamma_3}}$$

For large n, the last term in the denominator dominates, and it is minimized by choosing a P_r slightly smaller than P_c. $P_c = 2P_r$ works well on Intel machines. For smaller n, the middle term dominates, and it becomes more important to choose a small P_r. Suppose that we keep the ratio P_r/P_c constant as P increases, thus we have $P_r = u\sqrt{P}$ and $P_c = v\sqrt{P}$, where u and v are constant [10]. Moreover, let us ignore the $\log_2(P_r)$ factor for a moment. In this case, P_r/n and P_c/n are proportional to $\sqrt{P}/n$ and n^2 must grow with P to maintain efficiency. For sufficiently large P_r, the $\log_2(P_r)$ factor cannot be ignored, and the performance will slowly degrade with the number of processors P. This phenomenon is observed in practice as shown in Figure 10 for the efficiency of the LU factorization on the Intel Paragon.

The second group of routines are two-sided algorithms as opposed to one-sided algorithms. In these cases, it is not usually possible to maintain a communication pipeline, and thus square or near square grids are more optimal. This is the case for the algorithms used for implementing the Cholesky factorization, the matrix inversion and the reduction to bidiagonal form (BRD), Hessenberg form (HRD), tridiagonal form (TRD) and the nonsymmetric QR eigenvalue algorithm (HQR). For example, the update phase of the Cholesky factorization of a lower symmetric matrix physically transposes the current block of columns of the lower triangular factor.

Assume now that at most P processes are available. A natural question arising is: could we decide what process grid $P_r \times P_c \leq P$ should be used? Similarly, depending on P, it is not always possible to factor $P = P_r.P_c$ to create the appropriate grid. For example, if P is prime, the only possible grids are $1 \times P$ and $P \times 1$. If such grids are particularly bad for performance, it may be beneficial to let some processors remain idle, so the remainder can be formed into a "squarer" grid [24]. These problems can be analyzed by a complicated function of the machine and problem parameters. It is possible to develop models depending on the machine and problem parameters which accurately estimate the impact of modifying the shape of the grid on the total execution time, as well as predicting the necessary amount of extra memory required for each routine.

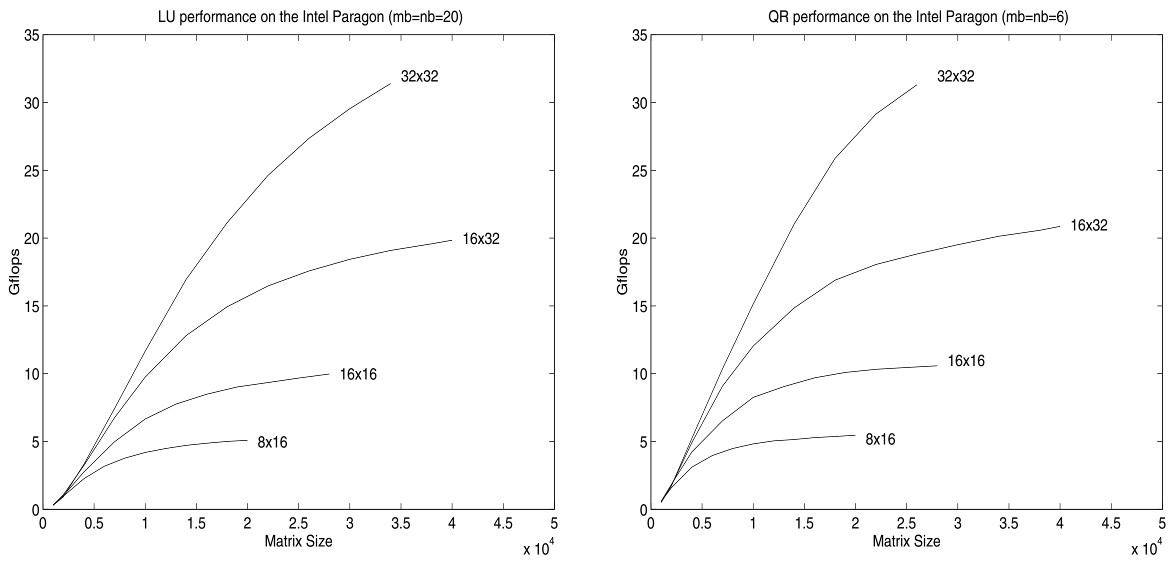


Figure 5: Intel Paragon LU and QR Performance

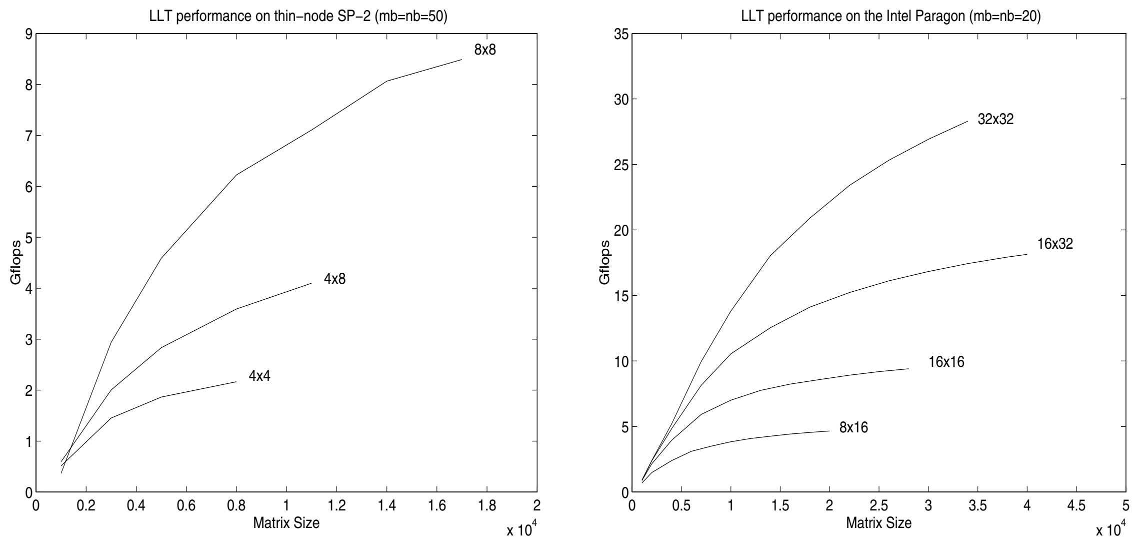


Figure 6: IBM SP-2 and Intel Paragon Cholesky Factorization

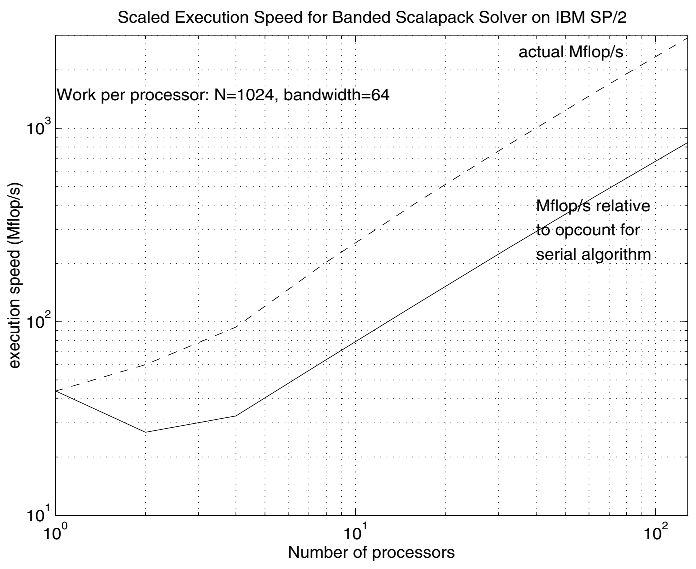


Figure 7: Execution speed in Mflop/s for the band algorithm on the IBM SP-2.

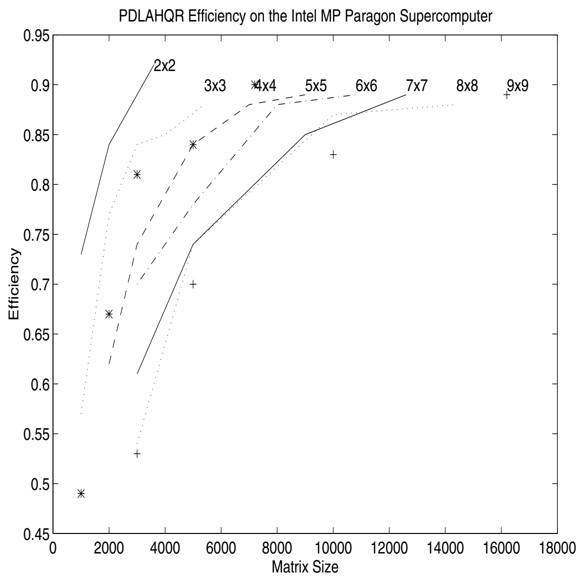
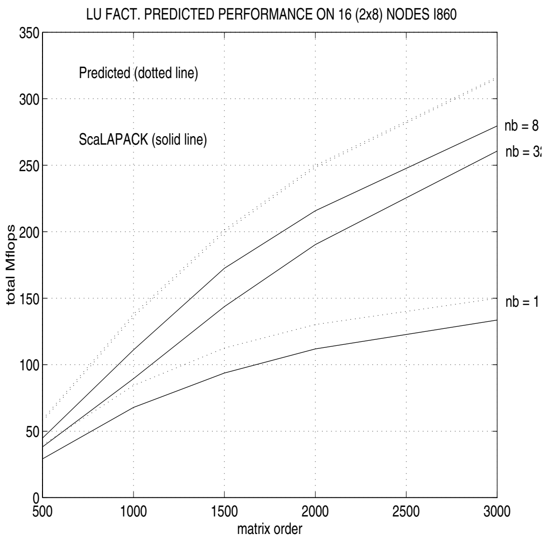


Figure 8: Performance of the Parallel QR Algorithm on the Intel MP Paragon for a 2x2 up to a 9x9 grid of processes.



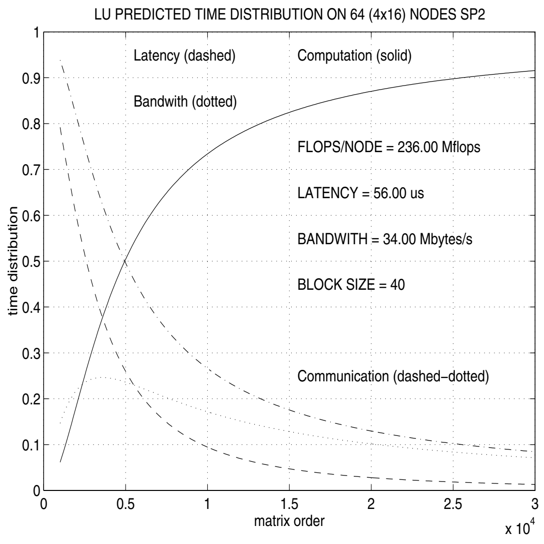
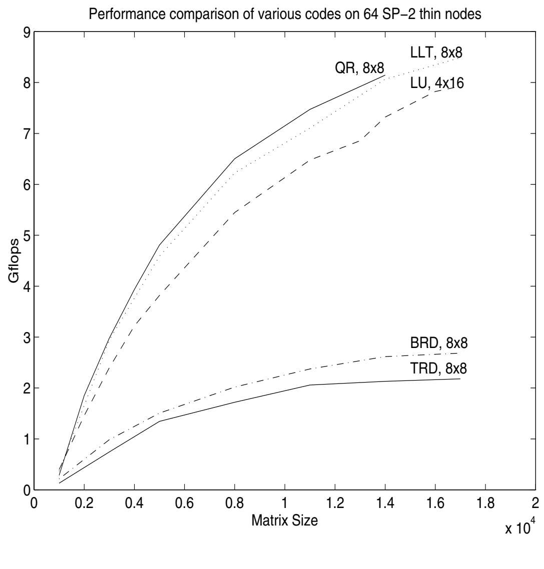


Figure 9: Performance with different blocksize NB ($MB = NB$) and prediction of percentage of time to performance operations.



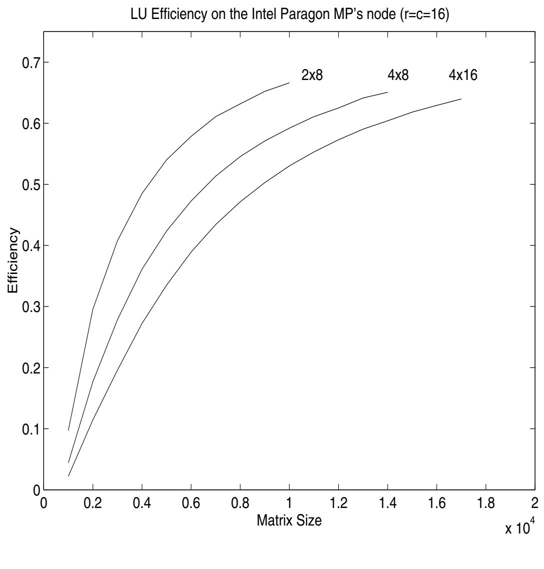


Figure 10: IBM SP-2 and Intel Paragon Performance

6 Future Directions

Future releases of the ScaLAPACK library will extend the flexibility of the PBLAS and increase the functionality of the library to include routines for the solution of general banded linear systems, general and symmetric positive definite tridiagonal systems, rank-deficient linear least squares problems, generalized linear least squares problems, and the singular value decomposition. Finally, general sparse and out-of-core prototype routines are being investigated, as well as an HPF [23, 25] interface for the ScaLAPACK library. An extensive discussion of alternative approaches for scalable parallel software libraries of the future can be found in [8].

7 Conclusions

ScaLAPACK is portable across a wide range of distributed-memory environments such as the IBM SP series, Intel series (Gamma, Delta, Paragon), Cray T3 series, TM CM-5, clusters of workstations, and any system for which PVM [20] or MPI [33] is available. Similar to the BLAS and LAPACK, many of the goals of the ScaLAPACK project, particularly portability, are aided by developing and promoting *standards*, especially for low-level communication and computation routines. We have been successful in attaining these goals, limiting most machine dependencies to two standard libraries called the BLAS, or Basic Linear Algebra Subroutines, and BLACS, or Basic Linear Algebra Communication Subroutines. ScaLAPACK will run on any machine where both the BLAS and the BLACS are available.

All ScaLAPACK-related software is publically available on *netlib* via the URL:

http://www.netlib.org/scalapack/index.html

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *"LAPACK Users' Guide, Second Edition"*. SIAM, Philadelphia, PA, 1995.

[2] E. Anderson, Z. Bai, and J. J. Dongarra. Generalized QR Factorization and its Applications. *Linear Algebra and Its Applications*, pages 162–164:243–273, 1992.

[3] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox, Part I. In *Proceedings of the Sixth SIAM Conference on Parallel Proceesing for Scientific Computing*. SIAM, 1993. Long version available as UC Berkeley Computer Science report all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-92-718.

[4] Z. Bai, J. Demmel, J. Dongarra, A. Petitet, H. Robinson, and K. Stanley. The spectral decomposition of nonsymmetric matrices on distributed memory computers. LAPACK Working Note #91 Technical report UT CS–95–273, University of Tennessee, 1995.

[5] Z. Bai, J. Demmel, and M. Gu. Inverse free parallel spectral divide and conquer algorithms for nonsymmetric eigenproblems. *Num. Math.*, 1996. to appear.

[6] L. S. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, R. C. Whaley. Practical experience in the dangers of heterogeneous computing. LAPACK Working Note #112 Technical report UT CS–96–330, University of Tennessee, 1996.

[7] S. Chakrabarti, J. Demmel, and K. Yelick. Modeling the benefits of mixed data and task parallelism. In Symposium on Parallel Algorithms and Architectures (SPAA), July 1995.

[8] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers – design issues and performance. LAPACK Working Note #95 Technical report UT CS–95–283, University of Tennessee, 1995.

[9] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. LAPACK Working Note #100 Technical report UT CS–95–292, University of Tennessee, 1995.

[10] J. Choi, J. Dongarra, R. Pozo, and D. Walker. "ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers". Technical Report UT CS-92-181, LAPACK Working Note #55, University of Tennessee, 1992.

[11] J. Choi, J. Dongarra, and D. Walker. "Parallel Matrix Transpose Algorithms on Distributed Concurrent Computers". Technical Report UT CS-93-215, LAPACK Working Note #65, University of Tennessee, 1993.

[12] J. Demmel, J. J. Dongarra, S. Hammarling, S. Ostrouchov, and K. Stanley. The dangers of heterogeneous network computing: Heterogenous networks considered harmful. In *Proceedings Heterogeneous Computing Workshop '96*, pages 64–71. IEEE, 1996.

[13] J. Demmel and K. Stanley. "The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers". In *Proceedings of the Seventh SIAM Conference on Parallel Proceesing for Scientific Computing*. SIAM, 1994.

[14] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. "A Set of Level 3 Basic Linear Algebra Subprograms". *ACM Transactions on Mathematical Software*, 16(1):1–28, 1990.

[15] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. "An Extended Set of Fortran Basic Linear Algebra Subprograms". *ACM Transactions on Mathematical Software*, 14(1):1–32, 1988.

[16] J. Dongarra and L. Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5:219–246, 1987.

[17] J. Dongarra and D. W. Walker. The Design of Linear Algebra Libraries for High Performance Computers. Technical Report UT CS-93-192, LAPACK Working Note #58, University of Tennessee, 1993.

[18] J. Dongarra and R. C. Whaley. "A User's Guide to the BLACS v1.0". Technical Report UT CS-95-281, LAPACK Working Note #94, University of Tennessee, 1995.

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

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM : Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. The MIT Press Cambridge, Massachusetts, 1994.

[21] G. Golub and C. F. Van Loan. *"Matrix Computations - 2nd Edition"*. Johns Hopkins University Press, Baltimore, MD, 1989.

[22] S. Hammarling. The numerical solution of the general Gauss-Markov linear model. In T. S. Durrani et al., editor, *Mathematics in Signal Processing*. Clarendon Press, Oxford, 1986.

[23] High Performance Forum. "High Performance Fortran Language Specification". Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, May 1993.

[24] W. Hsu, G. Thanh Nguyen, and X. Jiang. "Going Beyond Binary". http://www.cs.berkeley.edu/ xjiang/cs258/project_1.html, 1995. CS 258 Class project.

[25] Koebel, C., Loveman, D., Schreiber, R., Steele, G., Zosel, M.: The High Performance Fortran Handbook. The MIT Press, Cambridge, Massachusetts, 1994.

[26] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[27] Message Passing Interface Forum. "MPI: A Message-Passing Interface standard". *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[28] B. De Moor and P. Van Dooren. Generalization of the singular value and QR decompositions. *SIAM J. Matrix Anal. Appl.*, 13:993–1014, 1992.

[29] C. Paige. Fast numerically stable computations for generalized linear least squares problems. *SIAM J. Num. Anal.*, 16:165–179, 1979.

[30] C. Paige. Some aspects of generalized QR factorization. In M. Cox and S. Hammarling, editors, *Reliable Numerical Computations*. Clarendon Press, Oxford, 1990.

[31] B. Parlett, I. Dhillon, and V. Fernando. Private Communication, 1995.

[32] ScaLAPACK Users' Guide. http://www.netlib.org/scalapack/scalapack_ug.ps. To be published by SIAM.

[33] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, Massachusetts, 1996.