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

J. Choi[†], J. Demmel[‡], I. Dhillon[‡], J. Dongarra[§],
S. Ostrouchov[†], 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. This paper outlines the difficulties inherent in producing correct codes for networks of heterogeneous processors. We define a theoretical model of parallel computers dedicated to linear algebra applications: the Distributed Linear Algebra Machine (DLAM). This model provides a convenient framework for developing parallel algorithms and investigating their scalability, performance and programmability. Extensive performance results on various platforms are presented and analyzed with the help of the DLAM. 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.

*This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, 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.

[†]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

[¶]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 computers. It is a continuation of the LAPACK project, which designed and produced analogous software for workstations, vector supercomputers, and shared memory parallel computers. Both libraries contain routines for solving systems of linear equations, least squares problems, and eigenvalue problems. The goals of both projects are efficiency (to run as fast as possible), scalability (as the problem size and number of processors grow), reliability (including error bounds), portability (across all important parallel machines), flexibility (so users can construct new routines from well-designed parts), and ease-of-use (by making LAPACK and ScaLAPACK look as similar as possible). Many of these goals, 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 [6, 7, 14, 16], and BLACS, or Basic Linear Algebra Communication Subroutines [8, 10]. LAPACK and ScaLAPACK will run on any machine where the BLAS and the BLACS are available.

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 (Parallel BLAS), and the algorithms used are discussed. We also outline the difficulties encountered in producing correct code for networks of heterogeneous processors; difficulties we believe are little recognized by other practitioners.

The second part of this paper presents a theoretical model of parallel computers dedicated to dense linear algebra: the Distributed Linear Algebra Machine (DLAM). This ideal model provides a convenient framework for developing parallel algorithms. Moreover, it can be applied to obtain theoretical performance bounds and to analyze the scalability and programmability of parallel algorithms.

Finally, the paper 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. We show how the DLAM can be used to express this function, identify performance bottlenecks during development, and help users to choose various implementation parameters (like the number of processors) to optimize performance. One interesting result is that for some algorithms, speed is *not* a monotonic increasing function of the number of processors. In other words, speed can be increased by letting some processors remain idle.

2 Design of ScaLAPACK

2.1 Portability, Scalability and Standards

In order to be truly portable, the building blocks underlying parallel software libraries must be *standardized.* The definition of computational and message-passing standards [12, 14] 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.

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 dependences can at least be isolated.

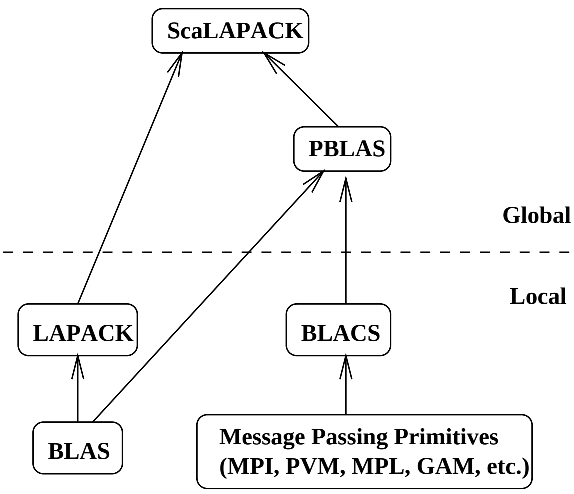
Scalability demands that a program be reasonably effective over a wide range of numbers of processors. The scalability of parallel algorithms over a range of architectures and numbers of processors requires that the granularity of computation be adjustable. To accomplish this, we use block algorithms 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

The following figure describes the ScaLAPACK software hierarchy. The components below the 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.

ScaLAPACK Software Hierarchy



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) [6, 7, 16] 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, least squares, 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 manual [2].

The **BLACS** (Basic Linear Algebra Communication Subprograms) [8, 10] 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 set up, 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 [12], also include this context concept. The BLACS provide facilities for safe interoperation of system contexts and BLACS contexts.

2.5 Block Cyclic Data Distribution

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 block cyclic distribution provides a simple, yet general-purpose way of distributing a block-partitioned matrix on distributed memory concurrent computers. It has been incorporated in the High Performance Fortran standard [11].

The block cyclic data distribution is parameterized by the four numbers P_r, P_c, r, and c, where $P_r \times P_c$ is the process template and $r \times c$ is the block size.

Suppose first that we have M objects, indexed by an integer $0 \leq m < M$, to map onto P processes, using block size r. The m-th item will be stored in the i-th location of block b on process p, where

$$\langle p, b, i \rangle = \left\langle \left\lfloor \frac{m}{r} \right\rfloor \bmod P, \left\lfloor \frac{\lfloor \frac{m}{r} \rfloor}{P} \right\rfloor, m \bmod r \right\rangle .$$

In the special case where $r = 2^{\hat{r}}$ and $P = 2^{\hat{P}}$ are powers of two, this mapping is really just bit extraction, with i equal to the rightmost $\hat{r}$ bits of m, p equal to the next $\hat{P}$ bits of m, and b equal to the remaining leftmost bits of m. The distribution of a block-partitioned matrix can be regarded as the tensor product of two such mappings: one that distributes the rows of the matrix over P_r processes, and another that distributes the columns over P_c processes. That is, the matrix element indexed globally by (m, n) is stored in location

$$\langle (p, q), (b, d), (i, j) \rangle =$$

$$\left\langle \left(\left\lfloor \frac{m}{r} \right\rfloor \bmod P_r,\ \left\lfloor \frac{n}{c} \right\rfloor \bmod P_c \right),\ \left(\left\lfloor \frac{\lfloor \frac{m}{r} \rfloor}{P_r} \right\rfloor,\ \left\lfloor \frac{\lfloor \frac{n}{c} \rfloor}{P_c} \right\rfloor \right),\ (m \bmod r,\ n \bmod c) \right\rangle .$$

The nonscattered decomposition (or pure block distribution) is just the special case $r = \lceil M/P_r \rceil$ and $c = \lceil N/P_c \rceil$. Similarly a purely scattered decomposition (or two dimensional wrapped distribution) is the special case $r = c = 1$.

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 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 [3].

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*. An array descriptor includes

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

For example, here 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 the 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.

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 "intra-context" operations.

We have 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 ScaLAPACK – LU Decomposition

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

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( 4 )
         JB = MIN( MIN(M,N)-J+JA, DESCA( 4 ) )
         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
```

The Cholesky decompositions (PDPOTRF and DPOTRF) and QR decompositions (PDGEQRF and DGEQRF) are analogous.

2.8 ScaLAPACK - Symmetric Eigenproblem

The solution of the symmetric eigenproblem 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 $\Lambda = \mathrm{diag}(\lambda_1, ..., \lambda_n)$ 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, similarly to LU. 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 completely eliminates the need for reorthogonalization has recently been discovered by Parlett, Fernando, and Dhillon [17], and we expect to use this version of the routine in the near future. This new routine will guarantee high accuracy and high speed independent of the eigenvalue distribution.

2.9 Heterogeneous Networks

There are special challenges associated with writing reliable numerical software on networks containing heterogeneous processors, i.e., 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. The basic problem lies in making *data dependent branches* on different processors, which may branch differently than expected on different processors, leading to different processors executing a completely different section of code than the other processors expect. We give three examples of this below.

The simplest example is an iteration where the stopping criterion depends on the machine precision. If the precision varies from processor to processor, different processors will have significantly different stopping criteria than others. In particular, the criterion for the most accurate processor may never be satisfied if it depends on data computed less accurately by other processors. Many problems like this can be eliminated by using the *largest* machine epilson among all participating processes. Routine PDLAMCH returns this largest value, replacing the uniprocessor DLAMCH. Similarly, one would use the smallest overflow threshold and largest underflow threshold for other calculations. But this is not a panacea, as subsequent examples show.

Next, consider the situation where processors sharing a distributed vector v compute its two-norm, and depending on that either scale v by a constant much different from 1, or do not. This happens in the inner loop of the QR decomposition, for example. The two-norm is computed by the ScaLAPACK routine PDNRM2, which computes two-norms locally and does a reduction. If the participating processors have different floating point formats, they may receive different values of the two-norm on return, just because the same floating point numbers cannot be represented on all machines. This two-norm is then compared to a threshold, and if it exceeds the threshold scaling takes place. Since the two-norm may be different, and the threshold may be different, the result of the comparison could differ on different processors, so that one process would scale the sub-vector it owns, and another would not. This would very likely lead to erroneous results. This could in principle be corrected by extending the reduction operation PDNRM2 to broadcast a discrete value (like the boolean value of a comparison); then all participating processors would be able to agree with the processor at the root of the reduction tree.

However, there are still harder problems. Consider bisection for finding eigenvalues of symmetric matrices. In this algorithm, the real axis is broken into disjoint intervals to be searched by different processors for the eigenvalues contained in each. Disjoint intervals are searched in parallel. The algorithm depends on a function, call it `count(a,b)`, that counts the number of eigenvalues in the half open interval [`a`, `b`). Using `count`, intervals can be subdivided into smaller intervals containing eigenvalues until the intervals are narrow enough to declare the eigenvalues they contain as "found". One problem is that two processors with different floating point formats cannot even agree on the boundary between their intervals, because they cannot store the same floating point number. This could result in multiple copies of eigenvalues if intervals overlap, or missing eigenvalues if there are gaps between intervals. Furthermore, the `count` function may count differently on different processors, so an interval [`a`, `b`) may be considered to contain 1 eigenvalue by processor A, but 0 eigenvalues by processor B, which has been given the interval by processor A during load balancing. This can happen even if processors A and B are identical, but if their compilers generate slightly different code sequences for `count`. We have not yet decided what to do about all of these problems, so we currently only guarantee correctness of `PDSYEVX` for networks of processors with identical floating point formats (slightly different floating point operations are acceptable). See [4] for details.

3 The Distributed Linear Algebra Machine (DLAM)

In this section, we present a theoretical model of a parallel computer dedicated to dense linear algebra. This model is from an abstraction of physical models. This ideal model provides a convenient framework for developing parallel algorithms without worrying about the implementation details or physical constraints. However, we defined this restricted model such that actual code should be easily produced from it.

The model can be applied to obtain theoretical performance bounds on parallel computers or to estimate the execution time before or after the algorithm has been implemented. The abstract model is also useful in scalability and programmability analysis.

A P-process DLAM is constructed out of P "BLAS-processes" interconnected by a logical "BLACS-network". This network is a $P_r \times P_c$ logical mesh such that $P_r.P_c \leq P$. Data are exchanged between BLAS processes through the BLACS network by calling BLACS primitives. The processes can only perform BLAS and BLACS operations.

The DLAM presented here could be very easily extended by adding a host process. This host process could act like a server acting upon a user request, creating the BLACS-network, distributing the data, starting the BLAS-processes and collecting the results. This host process could also be used for fault-tolerant applications. In this case, it would take the appropriate course of action in the case of a BLAS-process failure. In the following sections, however, we describe only the hostless DLAM.

3.1 The BLAS Process

As mentioned before, an efficient implementation of the BLAS masks the effects of the processor memory hierarchy and frees the programmer from local tuning of this basic kernel. The performance of the BLAS heavily depends on the number of memory references per floating point operation. This ratio naturally sorts the BLAS in three levels, where routines belonging to the same level usually reach similar execution rates. Consequently, the BLAS processes are, as far as performance analysis is concerned, able to perform only three instructions, corresponding to the three BLAS levels. The execution times per floating point operation of each of these instructions are then denoted by γ_i, with $i = 1, 2, 3$.

3.2 The BLACS Network

The BLAS processes communicate with each other via calls to the BLACS. For the sake of simplicity, we model a restricted subset of the possible BLACS operations, namely point-to-point communication and broadcast/combine operations along a row or column of the mesh. It is customary to model the time for sending a message of n items between two processes by

$$T_s(n, \alpha, \beta) = \alpha + n\beta$$

where α denotes the latency, and β the inverse of the bandwidth. The broadcast/combine operations are more complicated since the BLACS allow the user to specify a topology argument [8, 10]. We estimate the cost of broadcasting n items using a split-ring topology to $p-1$ processes by

$$T_b('S-ring, p, n, \alpha, \beta) = K('bcast',' S-ring', p, n)\ T_s(n, \alpha, \beta).$$

Similarly, the cost of a 1-tree combine operation of n items involving p processes is estimated by

$$T_c('S-ring, p, n, \alpha, \beta) = K('combine',' S-ring', p, n)\ T_s(n, \alpha, \beta).$$

At this level of the model, it is not possible to determine the values of K because no assumption has been made so far on the physical network to model. This justifies the introduction of these functions $K()$.

3.3 Accuracy and Refinement of the DLAM

When applying numerically the results obtained by the DLAM, we choose $\gamma_1 = \gamma_2 = 0$, assuming that the cost of these instructions will always be negligible compared to BLACS operations or a Level 3 instruction. We determined γ_3 as being the achieved peak performance of the BLAS matrix-multiply GEMM. This approximation is incorrect for small block sizes, in which case Level 2 operations are performed and γ_2, γ_3 should be set respectively to the achieved peak performance of the BLAS matrix-vector multiply GEMV and zero. Obviously, these coarse approximations could be refined by computing a piece-wise linear approximation of the γ_i's with respect to the problem size. This model smoothes the influence of the physical memory hierarchy and could be adapted to out-of-core BLAS operations.

Modeling the performance of the DLAM network is tightly coupled to the physical network. Experimental values of α and β can easily be determined for a given machine. If the logical mesh can be embedded into the physical network and the message collisions ignored, $2\ \log_2(p)$ is a good approximation of $K('combine',' 1-tree', p, n)$ assuming the result has to be left on the p processes and neglecting the cost of the local computations; similarly, $K('bcast',' 1-tree', p, n) \approx \log_2(p)$. When the communications can be pipelined, it is reasonable to estimate $K('bcast',' S-ring', p, n)$ by 2. Because this model ignores the probable collision of messages or possible network contention problems, its accuracy depends on the number of physical links. For instance, when comparing the performance obtained on an ideal DLAM with those obtained on an ethernet based network of workstations sharing one physical link, it is important to use appropriate values for K. Indeed, an upper bound for $K('combine',' 1-tree', p, n)$ is given by $2\ (p-1)$. However, for a given value of p, it is possible to experimentally determine constants which take into account the cost due to network contention and message collisions. More accurate models taking into account the collisions of messages could be used, but this is beyond the scope of this paper. Finally, the described model could obviously be refined by computing a piece-wise linear approximation of the time for sending a message with respect to the message length.

3.4 The LU factorization on the DLAM

We present in this section the model corresponding to the parallel right-looking LU factorization implemented in ScaLAPACK [9]. We restrict ourselves to the case where the matrix is distributed on the processes using a square ($r = c$) block cyclic decomposition scheme. We ignore the possible collision of messages on the network. It can be briefly described as follows: Assume the LU factorization of the $k \times r$ first columns has proceeded with $k \in \{0, 1, \ldots \frac{n-1}{r}\}$. During the next step, the algorithm factors the next panel of r columns, pivoting if necessary. Next the pivots are applied to the remainder of the matrix. The lower trapezoid factor just computed is broadcast to the other process columns of the grid using a split-ring topology [8, 10], so that the the upper trapezoid factor can be updated via a triangular solve. This factor is then broadcast to the other process rows using a 1-tree topology [8, 10], so that the remainder of the matrix can be updated by a rank-r update. This process continues recursively with the updated matrix. The total execution time $T_{LU}(n^2, P)$ can be estimated by

$$(n-1)T_c('1-tree', P_r, 2, \alpha, \beta)\ + \qquad \text{(Determine pivot row)}$$

$$(n-1)2T_s(r, \alpha, \beta)\ + \qquad \text{(Swap rows in current panel)}$$

$$\tfrac{n-1}{r}T_b('S-ring', P_c, r, \alpha, \beta)\ + \qquad \text{(Broadcast pivot information)}$$

$$\sum_{k=0}^{\frac{n-1}{r}} 2r(T_s(\frac{kr}{P_c}, \alpha, \beta)\ + T_s(\frac{n-(k+1)r}{P_c}, \alpha, \beta))\ + \qquad \text{(Swap remaining rows)}$$

$$\sum_{k=0}^{\frac{n-1}{r}} T_b('S-ring', P_c, \frac{n-kr}{P_r}, \alpha, \beta)\ + \qquad \text{(Broadcast lower trapezoid factor)}$$

$$\sum_{k=0}^{\frac{n-1}{r}} \frac{n-(k+1)r}{P_c} r^2 \gamma_3\ + \qquad \text{(Triangular solve: BLAS 3 TRSM)}$$

$$\sum_{k=0}^{\frac{n-1}{r}} T_b('1-tree', P_r, \frac{n-(k+1)r}{P_c}, \alpha, \beta)\ + \qquad \text{(Broadcast upper trapezoid factor)}$$

$$\sum_{k=0}^{\frac{n-1}{r}} 2\frac{n-(k+1)r}{P_r}\frac{n-(k+1)r}{P_c} r\gamma_3 \qquad \text{(Rank-}r\text{ update: BLAS 3 GEMM)}$$

Notice that we neglected the BLAS 1 computations performed during the factorization of the current panel of columns, considering that the contribution of this operations to the execution time is mostly due to communication. In addition, when the logical mesh can be embedded into the physical network and the message collisions neglected, the previous formula can be simplified to:

$$T_{LU}(n^2, P) = 2n\log_2(P_r)\alpha + \frac{n^2}{2P}(2P_c + P_r\log_2(P_r))\beta + \frac{2n^3}{3P}\gamma_3. \qquad (1)$$

4 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 [13] as

$$E(N,P) = \frac{1}{P}\frac{T_{\mathrm{seq}}(N)}{T(N,P)} \tag{2}$$

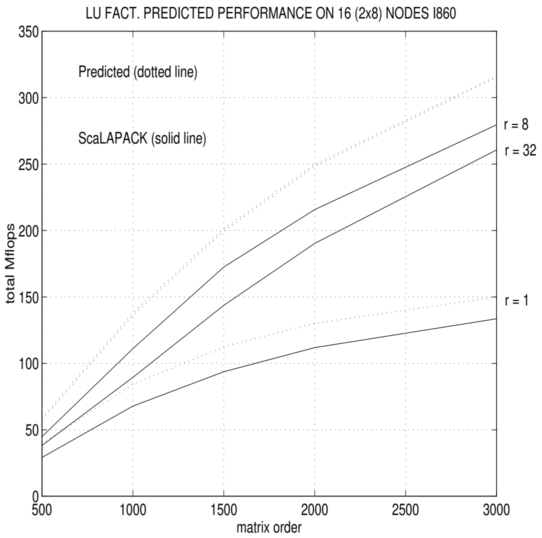
where $T(N,P)$ is the runtime of the parallel algorithm, and $T_{\mathrm{seq}}(N)$ is the runtime of the best sequential algorithm. An implementation is said to be *scalable* if the efficiency is an increasing function of N/P, the problem size per processor (in the case of dense matrix computations,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

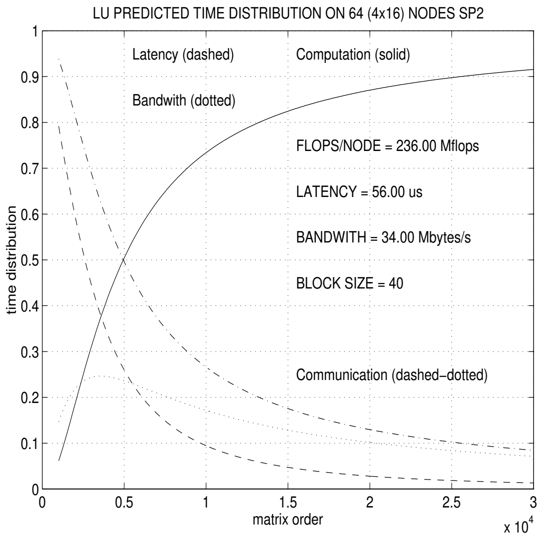
We will also measure the *performance* of our algorithm in Megaflops/sec (or Gigaflops/sec). This is appropriate for large dense linear algebra computations, since floating point dominates communication. For a scalable algorithm with N/P held fixed, we expect the performance to be proportional to P.

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. Performance data for the symmetric eigensolver (PDSYEVX) are presented in [5].

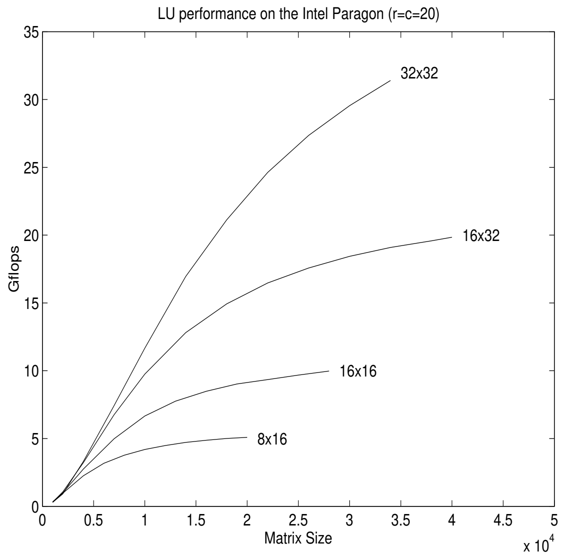
4.1 Choice of Block Size

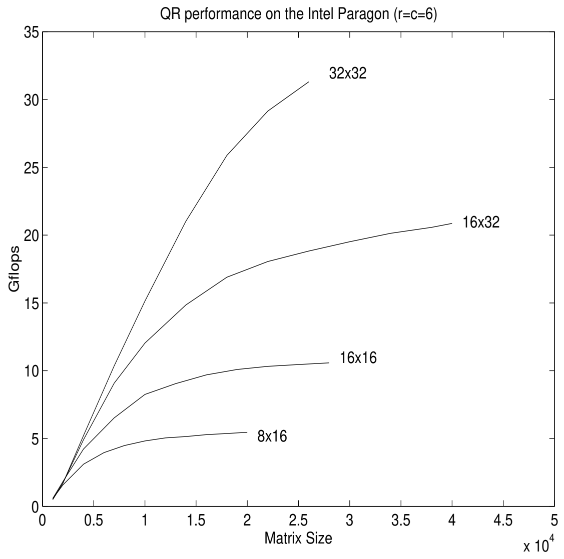
In the factorization or reduction routines, the work distribution becomes uneven as the computation progresses. A larger block size results in greater load imbalance, but reduces the frequency of communication between processes. There is, therefore, a tradeoff between load imbalance and communication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



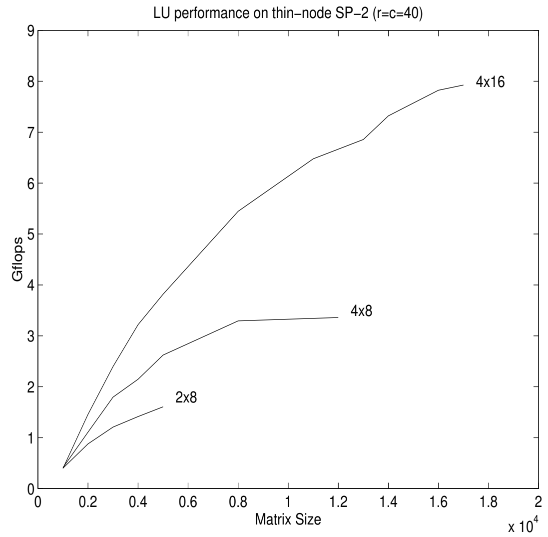


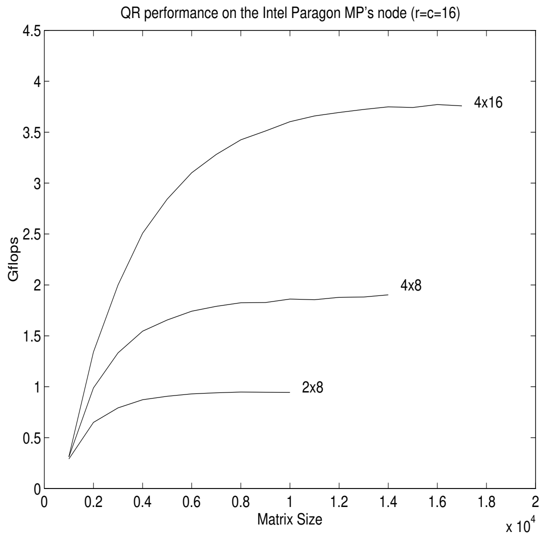
Most of the computation of the ScaLAPACK routines is performed in a blocked fashion using Level 3 BLAS, as is done in LAPACK. The computational blocking factor is chosen to be the same as the distribution block size. Therefore, smaller distribution block sizes increase the loop and index computation overhead. However, because the computation cost ultimately



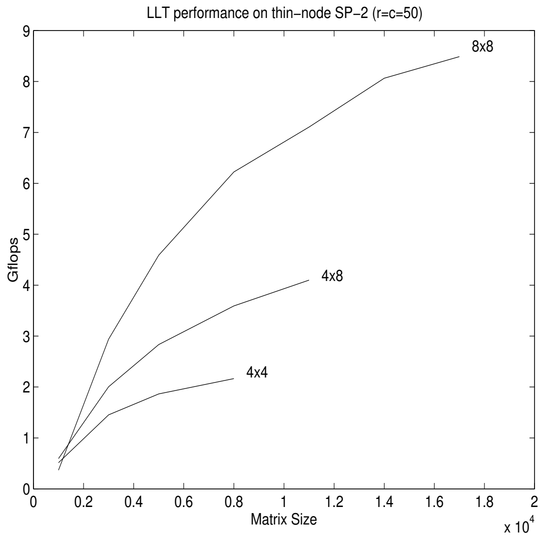


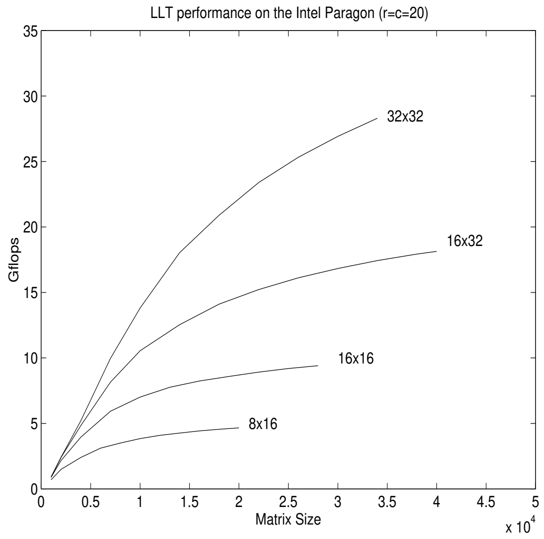
dominates, the influence of the block size on the overall communication startup cost and loop and index computation overhead decreases very rapidly with the problem size for a given grid of processes. Consequently, the performance of the ScaLAPACK library is not very sensitive to the block size, as long as the extreme cases are avoided. A very small block size leads to BLAS 2 operations and poorer performance (see section 3.3). A very large block size leads to computational imbalance.



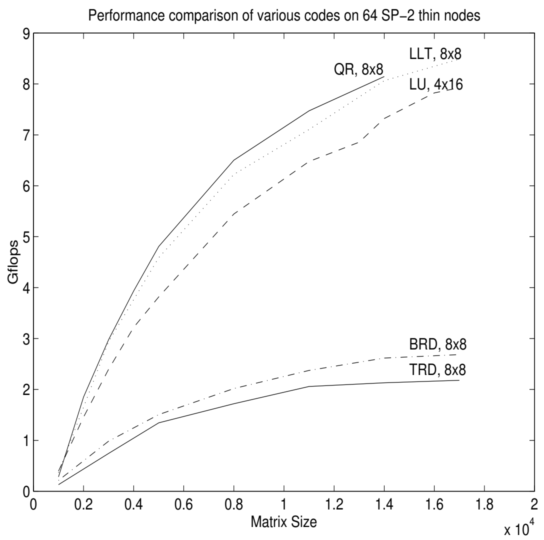


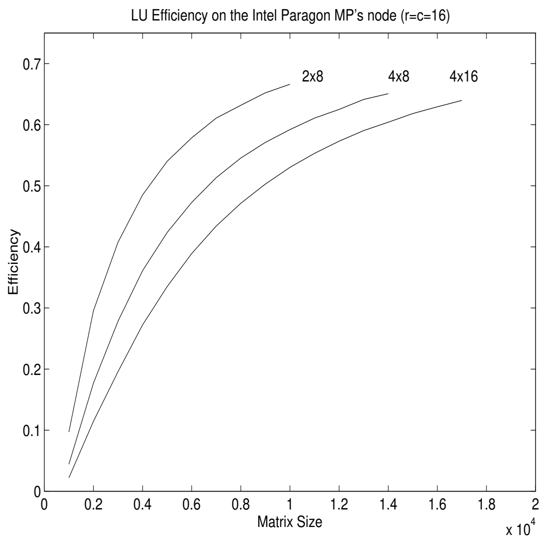
The chosen block size impacts the amount of workspace needed on every process. This amount of workspace is typically large enough to contain a block of columns or a block of rows of the matrix operands. Therefore, the larger the block size, the greater the necessary workspace, i.e the smaller the largest solvable problem on a given grid of processes. For Level 3 BLAS blocked algorithms, the smallest possible block operands are of size $r \times c$. 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.





4.2 Choice of Grid Size

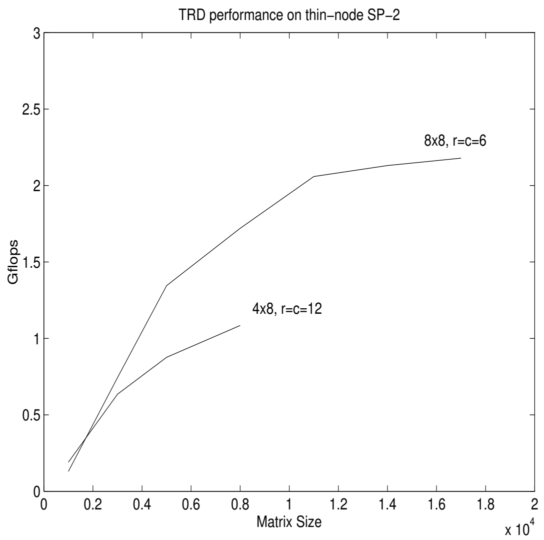
The best grid shape is determined by the algorithm implemented in the library and the underlying physical network. A one link physical network will favor $P_r = 1$ or $P_c = 1$. This affects the scalabilty of the algorithm, but reduces the overhead due to message collisions. It is possible to predict 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.

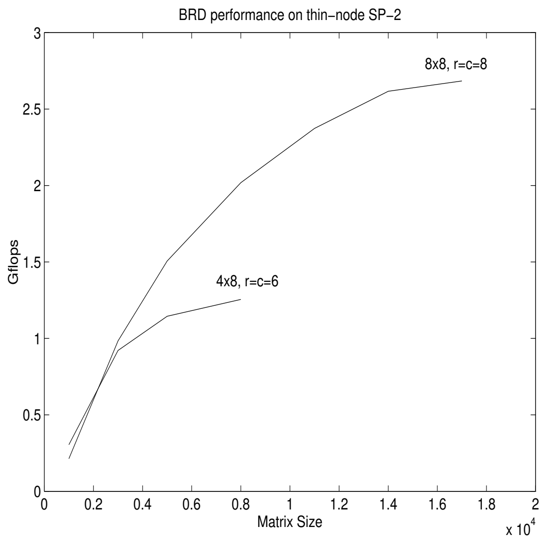
If at every step of the algorithm a block of columns and/or rows needs to be broadcast, as in the LU or QR factorizations, it is possible to pipeline this communication phase and overlap it with some computation. The direction of the pipeline determines the shape of the grid. For example, the LU, QR 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 (see (1), (2)):

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

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 [9]. Moreover, let 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 sufficient 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 the plot above showing the efficiency of the LU factorization on the Intel Paragon.





The second group of routines physically transpose a block of columns and/or rows at every step of the algorithm. 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) and tridiagonal form (TRD). 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 [15]. 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.

5 Future Directions

5.1 Future addition to ScaLAPACK

Basic building blocks like the BLAS, the BLACS and the PBLAS have been made publically available. At the time this paper was written, the current version of the PBLAS was being extended by removing alignment assumptions made on the operands. Moreover, the PBLAS package is being internally restructured to facilitate its maintenance and reinforce its robustness. Concurrently, many of the LAPACK functions missing in ScaLAPACK are being assembled and integrated. These include condition estimation, iterative refinement of linear solutions and linear least square solvers. We are planning improved versions of the symmetric eigenvalue routine. SVD and nonsymmetric eigenvalue routines are also in preparation. More elaborate testing and timing programs are being developed to ensure the robustness and the efficiency of the library. Finally, banded, general sparse, and out-of-core prototype routines are being investigated.

5.2 Alternative Approaches to Libraries

Traditionally, large, general-purpose mathematical software libraries on uniprocessors and shared memory machines have tried to hide much of the complexity of data structures and performance issues from the user. For example, the LAPACK project incorporates parallelism in the Level 3 BLAS, where it is not directly visible to the user. Unfortunately, it is not possible to hide these details as neatly on distributed memory machines. Currently, the data structures and data decomposition must be specified by the user, and it may be necessary to explicitly transform these structures in between calls to different library routines. These deficiencies in the conventional user interface have prompted extensive discussion of alternative approaches for scalable parallel software libraries of the future. Here are some possibilities.

1. Traditional function library (i.e., minimum possible change to the status quo in going from serial to parallel environment). This will allow one to protect the programming investment that has been made. More aggressive use of performance models may permit us to choose the best layout and redistribute the input data structure automatically. This is attractive for dense linear algebra since for large problems the $O(n^3)$ floating point operations will dominate the $O(n^2)$ cost of redistribution.
2. Reactive servers on the network. A user would be able to send a computational problem to a server that was specialized in dealing with the problem. This fits well with the concepts of a networked, heterogeneous computing environment with various specialized hardware

resources (or even the heterogeneous partitioning of a single homogeneous parallel machine). Again, this is attractive for dense linear algebra since $O(n^3)$ flops are performed on a data structure of size $O(n^2)$.

3. Interactive environments like Matlab or Mathematica, perhaps with "expert" drivers (i.e. knowledge-based systems) for special domains, such as structural analysis. Such environments have proven to be especially attractive for rapid prototyping of new algorithms and systems that may subsequently be implemented in a more customized manner for higher performance. With the growing popularity of the many integrated packages based on this idea, this approach would provide an interactive, graphical interface for specifying and solving scientific problems. Both the algorithms and data structures are hidden from the user, because the package itself is responsible for storing and retrieving the problem data in an efficient, distributed manner. In a heterogeneous networked environment, such interfaces could provide seamless access to computational engines that would be invoked selectively for different parts of the user's computation according to which machine is most appropriate for a particular subproblem.
4. Reusable templates (i.e., users adapt "source code" to their particular applications). A template is a description of a general algorithm rather than the executable object code or the source code more commonly found in a conventional software library. Nevertheless, although templates use generic versions of key data structures, they offer whatever degree of customization the user may desire. We have constructed such a set of template for interactive linear system solvers, and are currently constructing one for eigenvalue problems.

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"*. SIAM, Philadelphia, PA, 1992.

[2] 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.

[3] 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.

[4] J. Demmel, I. Dhillon, and H. Ren. "On the correctness of parallel bisection in floating point". Technical Report UCB//CSD-94-805, University of California, Berkeley Computer Science Division, 1994. available via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/csd/csd-94-805, file all.ps.

[5] 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.

[6] 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):1–17, 1990.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. "Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs". *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[8] J. Dongarra and R. van de Geijn. "Two dimensional Basic Linear Algebra Communication Subprograms". Technical Report UT CS-91-138, LAPACK Working Note #37, University of Tennessee, 1991.

[9] J. Dongarra, R. van de Geijn, and D. Walker. "A Look at Scalable Dense Linear Algebra Librairies". Technical Report UT CS-92-155, LAPACK Working Note #43, University of Tennessee, 1992.

[10] 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.

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

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

[13] 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.

[14] R. Hanson, F. Krogh, and C. Lawson. "A Proposal for Standard Linear Algebra Subprograms". *ACM SIGNUM Newsl.*, 8(16), 1973.

[15] 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.

[16] 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.

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