An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures

Jack J. Dongarra[§‡], Roldan Pozo[‡], and David W. Walker[§]

[§]Oak Ridge National Laboratory
Mathematical Sciences Section
P. O. Box 2008, Bldg. 6012
Oak Ridge, TN 37831-6367
Tel: (615) 574-7401
Fax: (615) 574-0680

[‡]University of Tennessee
Department of Computer Science
107 Ayres Hall
Knoxville, TN 37996-1301
Tel: (615) 974-8295
Fax: (615) 974-8296

Abstract

We describe the design of ScaLAPACK++, an object oriented C++ library for implementing linear algebra computations on distributed memory multicomputers. This package, when complete, will support distributed matrix operations for symmetric, positive-definite, and non-symmetric cases. In ScaLAPACK++ we have employed object oriented design methods to enchance scalability, portability, flexibility, and ease-of-use. We illustrate some of these points by describing the implementation of basic algorithms and comment on tradeoffs between elegance, generality, and performance.

1 Introduction

We describe an object oriented design for high performance linear algebra on distributed memory architectures based on extensions to the LAPACK [1] library. LAPACK includes state-of-the-art numerical algorithms for the more common linear algebra problems encountered in scientific and engineering applications. It is based on the widely used EISPACK [14] and LINPACK [8] libraries for solving linear equations, linear least squares, and eigenvalue problems for dense and banded systems.

ScaLAPACK++ is an object oriented extension designed to support distributed dense, banded, sparse matrix operations for symmetric, positive-definite, and non-symmetric cases. We have initially focused on the most common factorizations for dense systems: LU, QR, and Cholesky. The intent is that for large scale problems these ScaLAPACK++ routines should effectively exploit the computational hardware of medium grain-sized multicomputers with up to a few thousand processors, such as the Intel Paragon and Thinking Machines Corporation's CM-5.

Among the important design goals of ScaLAPACK++ are scalability, portability, flexibility, and ease-of-use. Such goals present serious challenges, as the layout of an application's data within the hierarchical memory of a concurrent computer is critical in determining the performance and scalability of the parallel code. To enhance the programmability of the library we would like details of the parallel implementation to be hidden as much as possible, but still provide the user with the capability to control the data distribution. The ScaLAPACK++ release will include a general two-dimensional matrix decomposition that supports the most common block/scattered schemes currently used. ScaLAPACK++ can be extended to support arbitrary matrix decompositions by providing the specific parallel BLAS library to operate on such matrices.

Decoupling the matrix operations from the details of the decomposition not only simplifies the encoding of an algorithm but also allows the possibility of postponing the decomposition until runtime. Often times the optimal matrix decomposition of is strongly dependent on how the matrix is utilized in other parts of the driving application code. Furthermore, it may be necessary to dynamically alter the matrix decomposition at runtime to accommodate special routines.

The currently supported decomposition scheme defines global matrix objects which are distributed across a $P \times Q$ logical grid of processors. Matrices are mapped to processors using a block scattered class of decompositions that allows a wide variety of matrix mappings while enhancing scalability and maintaining good load balance for various dense factorization algorithms. At each node of the multicomputer we have a sequential LAPACK++ library that provides the object oriented framework to describe block algorithms on conventional matrices in each individual processor.

Parallelism is exploited through the use of distributed memory versions of the Basic Linear Algebra Subprogram (BLAS)[9] that perform the basic computational units of the block algorithms. Thus, at a higher level, the block algorithms used look the same for the parallel or sequential versions, and only one version of each needs to be maintained.

The benefits of an object oriented design for ScaLAPACK++ include the ability to hide the implementation details of distributed matrices from the application programmer, and the ability to support a generic interface to basic computational kernels (BLAS), such as matrix multiply, without specifying the details of the matrix storage class.

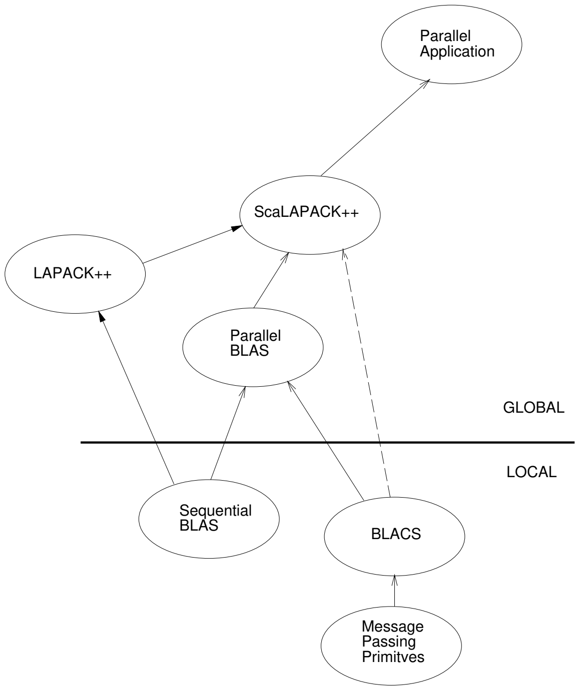

Figure 1: Design Hierarchy of ScaLAPACK++. In an SPMD environment, components above the horizontal reference line, represent a global viewpoint (a single distributed structure), while elements below represent a per-node local viewpoint of data.

2 Design Hierarchy

In figure 1 we illustrate the design hierarchy of ScaLAPACK++. A parallel SPMD application will utilize ScaLAPACK++ kernels to perform distributed linear algebra operations. Each node of a multicomputer runs a similar C++ program with calls to the ScaLAPACK++ interface. At this level distributed matrices are seen as a single object. The ScaLAPACK++ kernel, in turn, is built upon two important constituents: the basic algorithms of LAPACK++, and a parallel implementation of lower-computational kernels (BLAS). Since the code parallelism is imbedded in the low level BLAS kernels, the driving routines employing block matrix operations will look the same. Thus, the essential differences between LAPACK++ and ScaLAPACK++ codes are simply in the declarations of the matrix objects supported.

The parallel BLAS are modeled after their sequential counterparts and perform such tasks as matrix multiply, solving triangular systems, and performing rank-k updates. These operations constitute the basic level of parallelism in ScaLAPACK++ and typically require coordination between the various processors participating in the computation. At the local node level, interprocessor communication is accomplished via the Basic Linear Algebra Communication Subprogram (BLACS)[2] which utilize vectors and matrices as atomic message elements. This provides a portable message passing interface which hides the specific details of the underlying architecture-dependent message passing primitives. The BLACS, for example, have been implemented on top the Intel iPSC message passing interface , as well as the PVM [4] environment for heterogeneous networks. Ports to other platforms soon becoming available.

To illustrate the notion of how matrix algorithms are specified independent of their data decomposition, consider a block matrix algorithm to perform one of the most common matrix factorizations: a decomposition of a general non-symmetric matrix into upper and lower triangular factors, $A = LU$. This basic operation is performed as a subtask in solving systems of linear equations, $Ax = b$. Figure 2 describes the

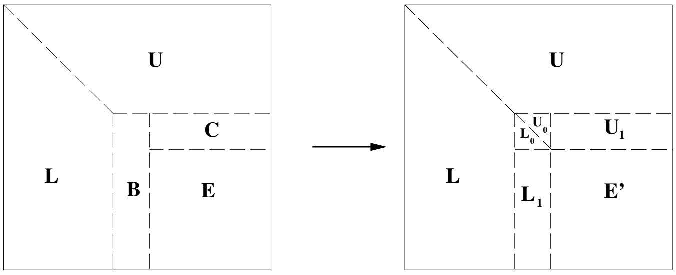

Figure 2: A schematic of the right-looking LU factorization algorithm using block matrix operations.

algorithm. The matrix A is broken up into panels and at each step we perform the following computation

1. LU factor the panel B into L_1, L_0, and U_0,
2. compute U_1 by solving the triangular system $L_0 U_1 = U_1$, and
3. update the trailing submatrix $E' = E - L_1 U_1$.

Ideally, the C++ code to implement this should be as close to the mathematical description as possible, only involving high-level matrix operations. In a sense, the above steps *define* the right-looking LU algorithm. No mention is made of *how* the matrix is stored or decomposed. This allows us to elegantly decouple *what* we are describing from *how* it is implemented. The result is that we can construct a common source code of this LU algorithm for any matrix decomposition.

The high-level programming style of the matrix classes may seem to imply that they are inefficient and incur a significant runtime performance overhead compared to similar computations using optimized Fortran. This section illustrates that this implication is not necessarily true. For single node performance, we have tested various prototype LAPACK++[11] modules on several architectures and found that they achieve competitive performance with similar optimized Fortran LAPACK routines. For example, figure 3 illustrates the performance of the LU factorization routine on an IBM RS/6000 Model 550 workstation. This particular implementation used GNU g++ v. 2.3.1 and utilized the BLAS-3 routines from the native ESSL library. The performance results are nearly identical with those of optimized Fortran calling the same library. This is accomplished by *inlining* the LAPACK++ kernels directly into the underlying Fortran or assembly-language. In this case, the C++ code is essentially expanded via macros to the underlying LAPACK Fortran routine `DGETRF()` and incurs no runtime overhead.

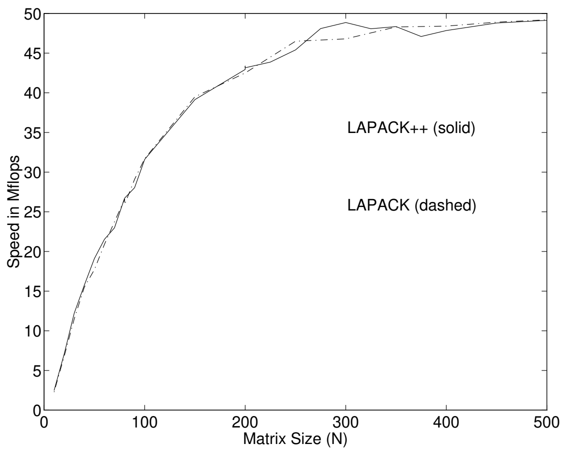

Figure 3: Performance of LAPACK++ LU factorization on the IBM RS/6000 Model 550 workstation, using GNU g++ v. 2.3.1 and BLAS routines from the IBM ESSL library. The results are nearly identical to the Fortran LAPACK performance.

3 Square Block Scattered Data Decomposition

In this section we consider the complications arising from distributed matrix structures and why it is important to decouple a matrix's decomposition from the driving algorithm. We present one of the more general matrix decomposition strategies that provides good performance on a variety of architectures.

Just as matrix algorithms in LAPACK seek to make efficient use of the hierarchical memory by maximizing data reuse, the block-partitioned algorithms in ScaLAPACK++ seek to reduce the frequency with which data must be transferred between processors. This reduces the fixed startup cost (or latency) incurred each time a message is communicated.

For matrix problems one can think of arranging the processors as a P by Q grid. Thus the grid consists of P rows of processors and Q columns of processors, and $N_p = PQ$. Each processor can be uniquely identified by its position, (p, q), on the processor grid. The decomposition of an $M \times N$ matrix can be regarded as the tensor product of two vector decompositions, μ and ν. The mapping μ decomposes the M rows of the matrix over the P rows of processors, and ν decomposes the N columns of the matrix over the Q columns of processors. Thus, if $\mu(m) = (p, i)$ and $\nu(n) = (q, j)$ then the matrix entry with global index (m, n) is assigned to the processor at position (p, q) on

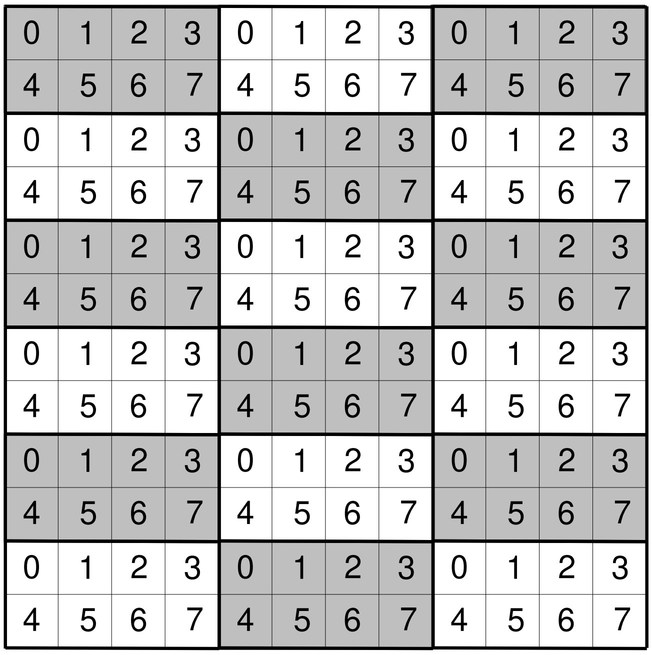

Figure 4: An example of block scattered decomposition over an 2x4 processor grid.

the processor grid, where it is stored in a local array with index (i, j).

Two common decompositions are the *block* and the *scattered* decompositions [7, 13]. The block decomposition, λ, assigns contiguous entries in the global vector to the processors in blocks.

$$\lambda(m) = (\lfloor m/L \rfloor, m \bmod L), \tag{1}$$

where $L = \lceil M/P \rceil$. The scattered decomposition, σ, assigns consecutive entries in the global vector to different processors,

$$\sigma(m) = (m \bmod P, \lfloor m/P \rfloor) \tag{2}$$

By applying the block and scattered decompositions over rows and columns a variety of matrix decompositions can be generated.

The *block scattered* decomposition scatters blocks of r elements over the processors instead of single elements, and if the blocks are rectangular, is able to reproduce the decompositions resulting from all possible block and scattered decompositions. Thus, by using the block scattered decomposition a large degree of decomposition independence can be attained. In the block scattered decomposition the mapping of the global index, m, can be expressed as a triplet of values, $\mu(m) = (p, t, i)$, where p is the processor position, t the block number, and i the local index within the block. For the block scattered decomposition we may write,

$$\zeta_r(m) = \left(\left\lfloor \frac{m \bmod T}{r} \right\rfloor, \left\lfloor \frac{m}{T} \right\rfloor, m \bmod r \right) \tag{3}$$

where $T = rP$. It should be noted that this reverts to the scattered decomposition when $r = 1$, with local block index $i = 0$. A block decomposition is recovered when $r = L$, with block number $t = 0$. The block scattered decomposition in one form or another has previously been used by various research groups (see [6] for references). It is also one of the decompositions provided in the Fortran D programming style [12].

We can view the block scattered decomposition as stamping a $P \times Q$ processor grid, or template, over the matrix, where each cell of the grid covers $r \times s$ data items, and is labeled by its position in the template (fig. 4). The block and scattered decompositions may be regarded as special cases of the block scattered decomposition. In general, the scattered blocks are rectangular, however, the use of nonsquare blocks can lead to complications, and additional concurrent overhead. We, therefore, propose to restrict ourselves to the square block scattered (SBS) class of decompositions. The column and row decompositions can still be recovered by setting $P = 1$ or $Q = 1$. However, more general decompositions for which $r \neq s$, and neither P nor Q is 1, cannot be reproduced by a SBS decomposition. These types of decomposition are not often used in matrix computations.

The SBS decomposition scheme is practical and sufficiently general-purpose for most, if not all, dense linear algebra computations. Furthermore, in problems, such as LU factorization, in which rows and/or columns are eliminated in successive steps, the SBS decomposition enhances scalability by ensuring statistical load balance.

Preliminary experiments of an object-based LU factorization algorithm using an SBS decomposition [6] suggest these algorithms scale well on multicomputers. For example, on the Intel Touchtone Delta System, a 520 i860-based multicomputer, such algorithms can achieve nearly twelve Gflops (figure 5).

4 Object Oriented Representation

ScaLAPACK++ utilizes an abstract base class `LaDistGenMat`, for **La**pack **Dist**ributed **Gen**eral **Mat**rix, from which various specific matrix decomposition (such as SBS) and other user-defined mappings can be derived. Since this is an abstract base class

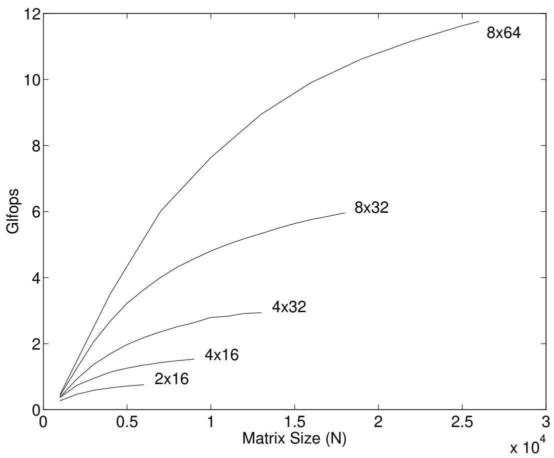

Figure 5: Performance of distributed LU factorization using an SBS matrix decomposition on the Intel Touchstone Delta system. Results are provided for various processor-grid configurations.

no objects of this type are explicitly created. Rather, they are used as placeholders in ScaLAPACK++ and parallel BLAS kernels.

Consider how a region of an distributed SBS matrix is viewed in each processor. Figure 6 depicts an SBS submatrix as from the global viewpoint and locally within each processor. Such a region might be selected, for example, to be used in a BLAS operation, or might be selected to receive the results of some previous calculation. In this example, processor 0 holds a complete block, while processors 2 and 6 do not contain any region of the submatrix. Thus, the local viewpoint of the SBS submatrix looks quite different in each processor and a nontrivial data structure is required to maintain this information.

Constructing SBS matrices can be accomplished by specifying a $P \times Q$ processor grid and blocksize r, as shown in figure 4. This is a piece of SPMD code that runs in each processor. Here `Procgrid` is an $P \times Q$ array of processor id's which describe the processor grid. The object, together with the blocksize can completely describe an SBS decomposition, `D`, which can in turn be used to initialize SBS matrices. For example, line `LaDistGenMat<double> A(M, N, D);` creates an SBS matrix of size $M \times N$ using the given decomposition characterization. At this point, the matrix `A` can be thought of as a special instance (derived class) of a more general (base class) distributed matrix class.

We can now call any matrix algorithm described in terms of the abstract distributed matrix class. The

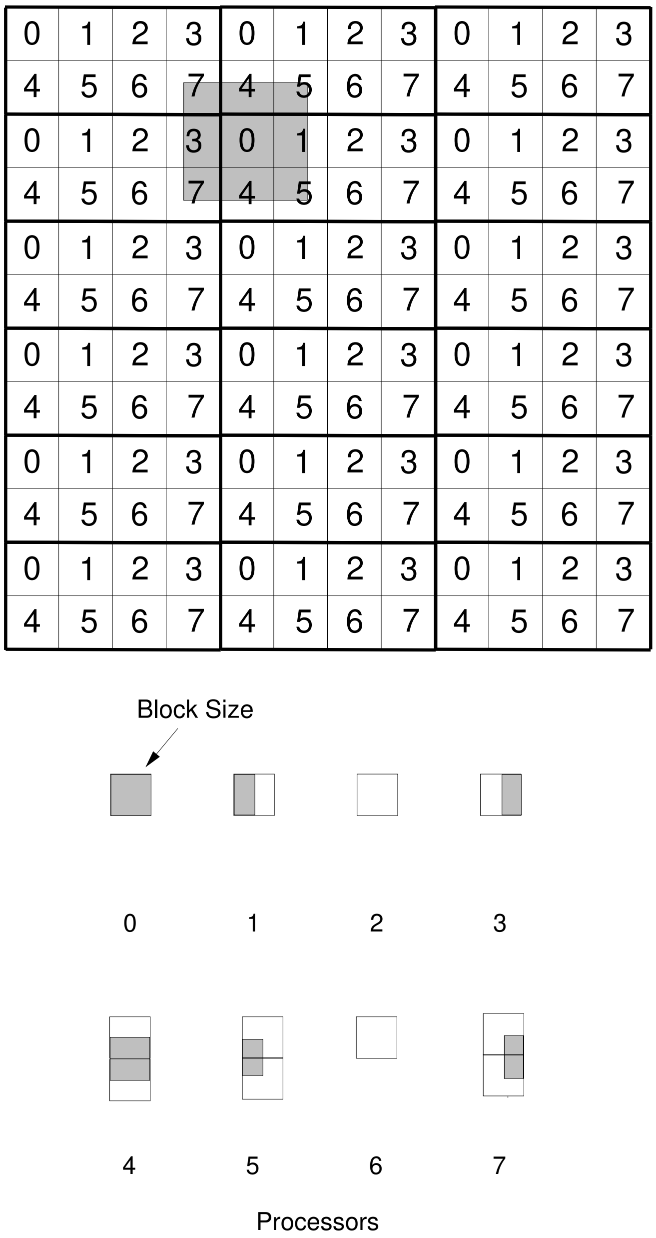

Figure 6: A submatrix as seen from the global viewpoint and locally within each processor. The internal details used to represent arbitrary rectangular regions of SBS matrices can become rather complicated.

```
LaMatrix<pid> Procgrid(P,Q);

LaSBSDecomp D(Procgrid, r);

LaSBSDistGenMat<double> A(M, N, D);
LaSBSDistGenMat<double> X(M, nb, D), B(M, nb, D);

Initialize(A,B);

LaSolve(A, X, B);
```

Figure 7: Constructing and utilizing distributed SBS matrices in ScaLAPACK++.

```
void Initialize(LaSBSDistGenMat<double> &A,
   LaSBSDistGenMat<double> &B)
{
    for (i=0; i<A.localsize(0); i++)
        for (j=0; j<A.localsize(1); j++)
        {
            I = A.globalindex(0,i);
            J = A.globalindex(1,j);

            A.local(i,j) =   .... f(I,J);
        }


    ...
}
```

Figure 8: A local viewpoint of an SBS matrix. Here we illustrate the mapping between global and local indices.

proper method will be called by using dynamic binding. All we need to supply with each new matrix decomposition is a matching parallel BLAS library that can perform the basic functions. The notion is that describing a new BLAS library is much simpler than specifying a new LAPACK library.

The code fragment in figure 4 illustrates how one can initialize the local processor-owned sections of the matrix using translation functions that map between local and global matrix indices.

5 LU Decomposition

To illustrate the ScaLAPACK++ code for the right-looking LU factorization, we first need to explain two other attributes of general block-matrix algorithms. The first is how to specify submatrix regions and the second is how to collect the pieces of an LU factorization into a single object.

Distributed submatrices in ScaLAPACK++ are denoted by specifying a subscript range via the `LaIndex()` function. For example,

```
A( LaIndex(0,2), LaIndex(3,6) )
```

denotes the elements $a_{ij}, i = 0,1,2;\ j = 3,4,5,6$. This is equivalent to the `A(0:2, 3:6)` notation of Fortran 90. By specifying an optional third argument to `LaIndex()` we can specify a non-unit stride, so that the expression `LaIndex(s,e,i)` generates the index sequence

$$s,\ \ s+i,\ \ s+2i,\ \ \ldots\ \ s+\lfloor\frac{e-s}{i}\rfloor i$$

Factorization classes are used to describe the various types of matrix decompositions: LU, Cholesky (LL^T), QR, and singular-value decompositions (SVD). The driver routines of LAPACK++ typically choose an appropriate factorization, but the advanced user can express specific factorization algorithms and their variants for finer control of their application or to meet specific memory storage and performance requirements.

In an object-oriented paradigm it is natural to encapsulate the factored representation of a matrix in a single object. A successful LU factorization, for example, will return the upper and unit-lower triangular factors, L and U, as well the pivoting information that describes how the rows were permuted during the factorization process. The representation of the L and U factors is meaningless without this information. Rather than store and manage these components separately, we couple the L and U matrices together with the pivot vector into one object. For example, to solve $AX = B$ we could write

```
LaGenMat<double> A, B, X;
LaGenFact<double> F;

LaLUFactor(F,A);
LaSolve(F, X ,B);
```

The ScaLAPACK++ code for the right-looking LU decomposition is shown in figure 9. The main point of this example code is that it independent of matrix decomposition. The code also includes a pivoting phase where rows of the original matrix are exchanged to enhance numerical stability of the algorithm.

6 Conclusion

We have presented a design of an object oriented scalable linear algebra library for distributed and hierarchical memory architectures. The design treats

```
#include <scalapack++.h>

int LaLUFactorDouble(LaGenMatDouble &A,
              LaGenFactDouble &F, int nb)
{
    // if blocksize is unacceptable
    // use the unblocked version of the code

    int j, jb, M = A.size(0), N = A.size(1);

    if (nb < 1 || nb > min(M,N))
        // use BLAS-2 level
       return LaLUFactorDoubleUnblocked(A,F);
    else
    {
       LaGenFactDouble F1;

       // use blocked code
       for (j=0; j<min(M,N); j+=nb)
       {
         jb = min(min(M,N)-j+1,nb);

         // factor current panel, unblocked version
         LaLUFactorDoubleUnblocked(
            A(Index(j,M-1), Index(j,j+jb-1)), F1);

         // apply interchanges to rows 0:j
         LaSwap(A(Index(), Index(0,j)),
            F1.pivot()(Index(0,jb-1)));

         if (j+jb < N)
         {
        // apply interchanges to rows j + jb-1:N
        LaSwap(A(Index(), Index(j+jb-1,N)),
            F1.pivot()(Index(0,jb-1)));

        // update global pivot vector
        F.pivot()(Index(j,min(M,j+jb-1))) =
            F1.pivot()(Index(0,jb-1)+(j-1));

        // compute block row of U
        LaSolve(F1.L(), A(Index(j+jb,M),
            Index(j,j+jb-1)));

        // update trailing submatrix E = E - LU
        LaMatMult(A(Index(j+jb,M), Index(j+jb,N)), -1.0,
              A(Index(j+jb,M), Index(j,j+jb-1)), 1.0,
              A(Index(j+jb,M), Index(j+jb,N)));
         }
       }
    }
}
```

Figure 9: ScaLAPACK++ example listing for right-looking LU decomposition.

both conventional and distributed matrices as fundamental objects. The ScaLAPACK++ library will include the basic matrix factorizations: LU, LL^T and QR decompositions.

In short, the goal of the ScaLAPACK++ software design is to denote numerical liner algebra algorithms in terms of high level primitives which are independent of a matrix's decomposition over the physical nodes of a multicomputer. By describing these matrix algorithms in terms of abstract base classes, we can share a common source code for both parallel and sequential versions. This is based on the observation that the parallelism available from block matrix algorithms, such as the LU decomposition is embedded deep inside in the block-level computational kernels, such as matrix multiply. These matrices may reside on a single processor, on a shared memory machine, distributed across a multicomputer, or over a cluster of workstations.

Although we have used the square scattered sub-block (SBS) matrix decompositions to illustrate some of the issues in building distributed linear algebra subroutines, the flexibility of ScaLAPACK++ allows for arbitrary user-defined matrix decompositions. All that is needed is to provide a parallel BLAS kernel that supports the fundamental matrix operations (such as matrix multiply, triangular solve, etc.)

Decoupling the matrix algorithm from a specific data decomposition provides three important attributes: (1) it results in simpler code which more closely matches the underlying mathematical formulation, (2) it allows for one "universal" algorithm, rather than supporting one version for each data decomposition needed, and (3) it allows one to postpone the data decomposition decision until runtime.

The third point is perhaps the most important in establishing truly portable software platforms for distributed memory architectures. The ideal data layout is not only dependent on the BLAS operations but also on how the matrix is used in other parts of the driving application. Moreover, the application any require various data decompositions during a single execution.

We have used the **inheritance** mechanism of the object oriented design to provide a common source code for both parallel and sequential versions of the code. Because the parallelism in embedded in the parallel BLAS library, the sequential and parallel high level matrix algorithms in ScaLAPACK++ look the same. This tremendously simplifies the complexity of the parallel libraries.

We have used **polymorphism**, or late binding, mechanisms to achieve a truly portable distributed matrix library in which the data decomposition may be dynamically changed at runtime.

We have utilized operator and function **overloading** capabilities of C++ to simplify the syntax and user interface into the LAPACK and BLAS functions.

We have utilized the **function inlinig** capabilities of C++ to reduce the function-call overhead usually associated with interfacing Fortran or assembly kernels.

Finally we have also used the **template** facility of C++ to reduce the amount of code redundancy by not having to separately maintain single precision, double precision, single complex precision, and double complex precision versions. By making the matrix data element type a parameter, we can acheive this goal and also provide the extendability to integrate other user-defined data types.

In short, we have used various important aspects of object oriented mechanisms and C++ in the design of ScaLAPACK++. These attributes were utilized not because of novelty, but out of necessity to incorporate a design which provides scalibility, portability, flexibility, and ease-of-use.

References

[1] E. Anderson and Z. Bai and J. Demmel and J. Dongarra and J. DuCroz and A. Greenbaum and S. Hammarling and A. McKenney and S. Ostrouchov and D. Sorensen, *LAPACK Users' Guide*, SIAM Press, Philadelphia, PA, 1992.

[2] E. Anderson, A. Benzoni, J. Dongarra, S. Moulton, S. Ostrouchov, B. Tourancheau, and R. van de Geijn. Basic Linear Algebra Communication Subprograms. In *Sixth Distributed Memory Computing Conference Proceedings*, pages 287–290. IEEE Computer Society Press, 1991.

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

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, V. Sunderam, *User's Guide to PVM: Parallel Virtual Machine*, ORNL/TM-11826, Mathematical Sciences Section, Oak Ridge National Laboratory, Sept. 1991.

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

[6] J. Choi, J. J. Dongarra, R. Pozo, D. W. Walker Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Press, Oct. 1992.

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

[8] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, *LINPACK Users' Guide*, SIAM, Philadelphia, PA, 1979.

[9] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, *A set of Level 3 Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 16 (1990), pp. 1–17.

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

[11] J. J. Dongarra, R. Pozo, D. W. Walker, *LAPACK++: A design overview of Object Oriented Extensions for High Peformance Linear Algebra*, in preparation.

[12] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C-W. Tseng, and M-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, Rice University, December 1990.

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

[14] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *Matrix Eigensystem Routines – EISPACK Guide*, vol. 6 of Lecture Notes in Computer Science, Springer-Verlag, Berlin, 2 ed., 1976.