LAPACK++: A Design Overview of Object-Oriented Extensions for High Performance Linear Algebra

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

§Oak Ridge National Laboratory
Mathematical Sciences Section

‡University of Tennessee
Department of Computer Science

Abstract

LAPACK++ is an object-oriented C++ extension of the LAPACK (Linear Algebra PACKage) library for solving the common problems of numerical linear algebra: linear systems, linear least squares, and eigenvalue problems on high-performance computer architectures. The advantages of an object-oriented approach include the ability to encapsulate various matrix representations, hide their implementation details, reduce the number of subroutines, simplify their calling sequences, and provide an extendible software framework that can incorporate future extensions of LAPACK, such as ScaLAPACK++ for distributed memory architectures. We present an overview of the object-oriented design of the matrix and decomposition classes in C++ and discuss its impact on elegance, generality, and performance.

1 Introduction

LAPACK++ is an object-oriented C++ extension to the Fortran LAPACK [1] library for numerical linear algebra. This package 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 LINPACK [5] and EISPACK [13] libraries for solving linear equations, linear least squares, and eigenvalue problems for dense and banded systems. The current LAPACK software consists of over 1,000 routines and 600,000 lines of Fortran 77 source code.

The numerical algorithms in LAPACK utilize block-matrix operations, such as matrix-multiply, in the innermost loops to achieve high performance on cached and hierarchical memory architectures. These operations, standardized as a set of subroutines called the Level 3 BLAS (Basic Linear Algebra Subprograms [6]), improve performance by increasing the granularity of the computations and keeping the most frequently accessed subregions of a matrix in the fastest level of memory. The result is that these block matrix versions of the fundamental algorithms typically show performance improvements of a factor of three over non-blocked versions [1].

LAPACK++ provides a framework for describing general block matrix computations in C++. Without a proper design of fundamental matrix and factorization classes, the performance benefits of blocked codes can be easily lost due to unnecessary data copying, inefficient access of submatrices, and excessive run-time overhead in the dynamic-binding mechanisms of C++.

LAPACK++, however, is not a general purpose array package. There are no functions, for example, to compute trigonometric operations on matrices, or to deal with multi-dimensional arrays. There are several good public domain and commercial C++ packages for these problems [4], [9], [11], [12]. The classes in LAPACK++, however, can easily integrate with these or with any other C++ matrix interface. These objects have been explicitly designed with block matrix algorithms and make extensive use of the level 3 BLAS. Furthermore, LAPACK++ is more than just a shell to the FORTRAN library; some of the key routines, such as the matrix factorizations, are actually implemented in C++ so that the general algorithm can be applied to derived matrix classes, such as distributed memory matrix objects.

*This project was supported in part by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office, the Applied Mathematical Sciences subprogram of the Office of Energy Research, 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.

LAPACK++ provides speed and efficiency competitive with native Fortran codes (see Section 2.2), while allowing programmers to capitalize on the software engineering benefits of object oriented programming. Replacing the Fortran 77 interface of LAPACK with an object-oriented framework simplifies the coding style and allows for a more flexible and extendible software platform. In Section 6, for example, we discuss extensions to support distributed matrix computations on scalable architectures [2].

The motivation and design goals for LAPACK++ include

- Maintaining competitive performance with Fortran 77.
- Providing a simple interface that hides implementation details of various matrix storage schemes and their corresponding factorization structures.
- Providing a universal interface and open system design for integration into user-defined data structures and third-party matrix packages.
- Replacing static work array limitations of Fortran with more flexible and type-safe dynamic memory allocation schemes.
- Providing an efficient indexing scheme for matrix elements that has minimal overhead and can be optimized in most application code loops.
- Utilizing function and operator overloading in C++ to simplify and reduce the number of interface entry points to LAPACK.
- Providing the capability to access submatrices by **reference**, rather than by value, and perform factorizations "in place" – vital for implementing blocked algorithms efficiently.
- Providing more meaningful naming conventions for variables and functions (e.g. names no longer limited to six alphanumeric characters, and so on).

LAPACK++ also provides an object-oriented interface to the Basic Linear Algebra Subprograms (BLAS) [6], allowing programmers to utilize these optimized computational kernels in their own C++ applications.

2 Overview

The underlying philosophy of the LAPACK++ design is to provide an interface which is simple, yet powerful enough to express the sophisticated numerical algorithms within LAPACK, including those which optimize performance and/or storage. Programmers who wish to utilize LAPACK++ as a black box to solve $Ax = B$ need not be concerned with the intricate details.

Following the framework of LAPACK, the C++ extension contains **driver routines** for solving standard types of problems, **computational routines** to perform distinct computational tasks, and **auxiliary routines** to perform certain subtasks or common low-level computations. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines.

Utilizing function overloading and object inheritance in C++, the procedural interface to LAPACK has been simplified: two fundamental drivers and their variants, `LaLinSolve()` and `LaEigenSolve()`, replace several hundred subroutines in the original Fortran version.

LAPACK++ supports various algorithms for solving linear equations and eigenvalue problems:

- Algorithms
  - LU Factorization
  - Cholesky (LL^T) Factorization
  - QR Factorization (linear least squares)
  - Singular Value Decomposition (SVD)
  - Eigenvalue problems (as included in LAPACK)
- Storage Classes
  - rectangular matrices
  - symmetric and symmetric positive definite (SPD)
  - banded matrices
  - tri/bidiagonal matrices
- Element Data Types
  - float, double, single and double precision complex

In this paper we focus on matrix factorizations for linear equations and linear least squares.

2.1 A simple code example

To illustrate how LAPACK++ simplifies the user interface, we present a small code fragment to solve linear systems. The examples are incomplete and are meant to merely illustrate the interface style. The next few sections discuss the details of matrix classes and their operations.

Consider solving the linear system $Ax = b$ using LU factorization in LAPACK++:

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

LaGenMatDouble A(N,N);
LaVectorDouble x(N), b(N);

// ...

LaLinSolve(A,x,b);
```

The first line includes the LAPACK++ object and function declarations. The second line declares A to be a square $N \times N$ coefficient matrix, while the third line declares the right-hand-side and solution vectors. Finally, the `LaLinSolve()` function in the last line calls the underlying LAPACK driver routine for solving general linear equations.

Consider now solving a similar system with a tridiagonal coefficient matrix:

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

LaTridiagMatDouble A(N,N);
LaVectorDouble x(N), b(N);

// ...

LaLinSolve(A,x,b);
```

The only code modification is in the declaration of A. In this case `LaLinSolve()` calls the driver routine `DGTSV()` for tridiagonal linear systems. The `LaLinSolve()` function has been **overloaded** to perform different tasks depending on the type of the input matrix A. There is no runtime overhead associated with this; it is resolved by C++ at compile time.

2.2 Performance

The elegance of the LAPACK++ matrix classes may seem to imply that they incur a significant runtime performance overhead compared to similar computations using optimized Fortran. This is not true. The design of LAPACK++ has been optimized for performance and can utilize the BLAS kernels as efficiently as Fortran. Figure 1, for example, illustrates the Megaflop rating of the simple code

```
C = A*B;
```

for square matrices of various sizes on the IBM RS/6000 Model 550 workstation. This particular implementation used GNU g++ v. 2.3.1 and utilized the Level 3 BLAS 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++ BLAS kernels. That is, these functions are expanded at the point of their call by C++ compiler, saving the runtime overhead of an explicit function call.

In this case the above expression calls the underlying DGEMM BLAS 3 routine. This occurs at *compile* time, without any runtime overhead. The performance numbers are very near the machine peak and illustrate that using C++ with optimized computational kernels provides an elegant high-level interface without sacrificing performance.

Figure 2 illustrates performance characteristics of the LU factorization of various matrices on the same architecture using the LAPACK++

```
LaLUFactorIP(A,F)
```

routine, which overwrites `A` with its LU factors. This routine essentially inlines to the underlying LAPACK routine `DGETRF()` and incurs no runtime overhead. (The `IP` suffix stands for "In Place" factorization. See Section 4.1 for details.)

3 LAPACK++ Matrix Objects

The fundamental objects in LAPACK++ are numerical vectors and matrices; however, LAPACK++ is *not* a general-purpose array package. Rather, LAPACK++ is a self-contained interface consisting of only the minimal number of classes to support the functionality of the LAPACK algorithms and data structures.

LAPACK++ matrices can be referenced, assigned, and used in mathematical expressions as naturally as if they were an integral part of C++; the matrix element a_{ij}, for example, is referenced as `A(i,j)`. By default, matrix subscripts begin at zero, keeping with the indexing convention of C++; however, they can be set to any user-defined value. (Fortran programmers typically prefer 1.) Internally, LAPACK++ matrices

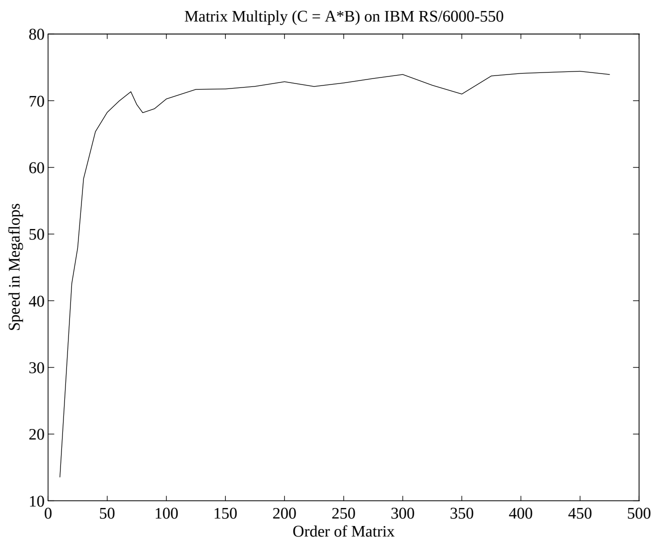

Figure 1: Performance of matrix multiply in LAPACK++ on the IBM RS/6000 Model 550 workstation. GNU g++ v. 2.3.1 was used together with the ESSL Level 3 routine **dgemm**.

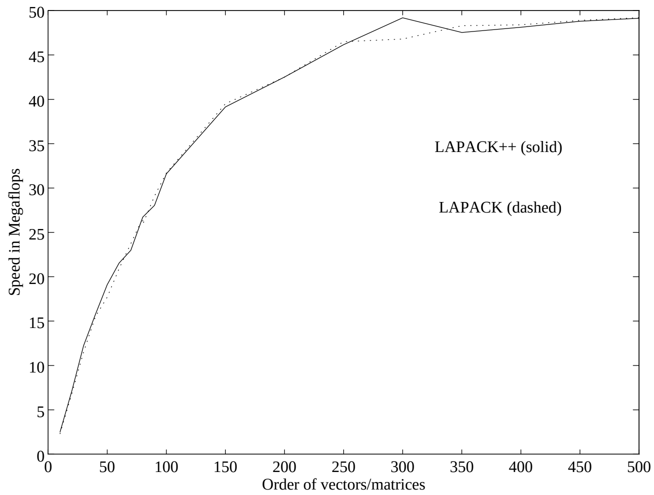

Figure 2: Performance of LAPACK++ LU factorization on the IBM RS/6000 Model 550 workstation, using GNU g++ v. 2.3.1 and BLAS routines from the IBM ESSL library. The results are indistinguishable between the Fortran and C++ interfaces.

are typically stored in column-order for compatibility with Fortran subroutines and libraries.

Various types of matrices are supported: banded, symmetric, Hermitian, packed, triangular, tridiagonal, bidiagonal, and non-symmetric. Rather than have an unstructured collection of matrix classes, LAPACK++ maintains a class hierarchy (Figure 3) to exploit commonality in the derivation of the fundamental matrix types. This limits much of the code redundancy and allows for an open-ended design which can be extended as new matrix storage structures are introduced.

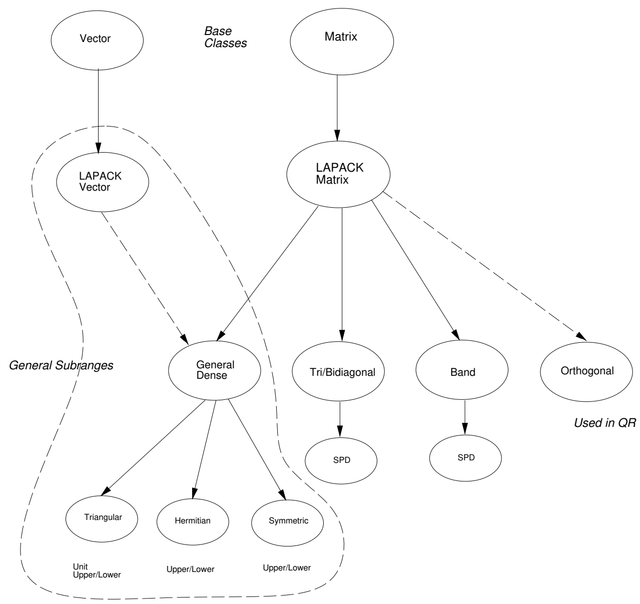

Figure 3: The Matrix Hierarchy of LAPACK++. Rectangular and Vector classes allow indexing of any rectangular submatrix by reference.

Matrix classes and other related data types specific to LAPACK++ begin with the prefix "`La`" to avoid naming conflicts with user-defined or third-party matrix packages. The list of valid names is a subset of the nomenclature shown in Figure 4.

3.1 General Matrices

One of the fundamental matrix types in LAPACK++ is a general (nonsymmetric) rectangular matrix. The possible data element types of this matrix include single and double precision of real and complex numbers. The corresponding LAPACK++ names are given as

`LaGenMat`*type*

$$\mathtt{La}\left\{\begin{array}{c}\mathtt{Gen}\\ \mathtt{Orthog}\\ \left[\begin{array}{l}\mathtt{Symm}\\ \mathtt{Hermitian}\\ \mathtt{SPD}\end{array}\right]\left\{\begin{array}{l}\mathtt{Band}\\ \mathtt{Tridiag}\\ \mathtt{Bidiag}\\ \mathtt{Packed}\end{array}\right\}\\ {[}\mathtt{Unit}{]}\left\{\begin{array}{c}\mathtt{Upper}\\ \mathtt{Lower}\end{array}\right\}\{\mathtt{Triang}\}\end{array}\right\}\mathtt{Mat}\left\{\begin{array}{l}\mathtt{Float}\\ \mathtt{Double}\\ \mathtt{Fcomplex}\\ \mathtt{Dcomplex}\end{array}\right\}$$

Figure 4: LAPACK++ matrix nomenclature. Items in square brackets are optional.

for `La`pack `Gen`eral `Mat`rix. The *type* suffix can be **`float`**, **`double`**, **`fcomplex`**, or **`dcomplex`**. Matrices in this category have the added property that submatrices can be efficiently accessed and referenced in matrix expressions. This is a necessity for describing block-structured algorithms.

3.1.1 Declarations

General LAPACK++ matrices may be declared (constructed) in various ways:

```
#include <lapack++.h>
float d[4] = {1.0, 2.0, 3.0, 4.0};

LaGenMatDouble     A(200,100) = 0.0; // 1
LaGenMatDComplex   B;                // 2
LaGenMatDouble     C.ref(A);         // 3
LaGenMatDouble     D(A);             // 4
LaGenMatFloat      E(d, 2, 2)        // 5
```

Line (1) declares `A` to be a rectangular 200x100 matrix, with all of its elements initialized to 0.0. Line (2) declares `B` to be an empty (uninitialized) matrix. Until B becomes initialized, any attempt to reference its elements will result in a run time error. Line (3) declares `C` to share the same elements of `A`. Line (4) illustrates an equivalent way of specifying this at the time of a new object construction. Finally, line (5) demonstrates how one can initialize a 2x2 matrix with the data from a standard C++ vector. The values are initalized in column-major form, so that the first column of `E` contains $\{1.0, 2.0\}^T$, and the second column contains $\{3.0, 4.0\}^T$.

3.1.2 Submatrices

Blocked linear algebra algorithms utilize submatrices as their basic unit of computation. It is crucial that submatrix operations be highly optimized. Because of this, LAPACK++ provides mechanisms for accessing rectangular subregions of a general matrix. These regions are accessed by *reference*, that is, without copying data, and can be used in any matrix expression.

Ideally, one would like to use familiar colon notation of Fortran 90 or Matlab for expressing submatrices. However, this modification of the C++ syntax is not possible without redefining the language specifications. As a reasonable compromise, LAPACK++ denotes submatrices by specifying a subscript range through the `LaIndex()` function. For example, the 3x3 matrix in the upper left corner of `A` is denoted as

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

This references $A_{ij}, i = 0, 1, 2\ j = 0, 1, 2$, and is equivalent to the `A(0:2,0:2)` colon notation used elsewhere. Submatrix expressions may be also be used as a destination for assignment, as in

```
A( LaIndex(0,2), LaIndex(0,2) ) = 0.0;
```

which sets the 3x3 submatrix of A to zero. Following the Fortran 90 conventions, the index notation has an optional third argument denoting the stride value,

$$\boxed{\mathtt{LaIndex(start,\ end,\ increment)}}$$

If the `increment` value is not specified it is assumed to be one. The expression `LaIndex(s, e, i)` is equivalent to the index sequence

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

The internal representation of an index is not expanded to a full vector, but kept in its compact triplet format. The increment values may be negative and allow one to traverse a subscript range in the opposite direction, such as in `(10,7,-1)` to denote the sequence $\{10, 9, 8, 7\}$. Indices can be named and used in expressions, as in the following submatrix assignments,

```
LaGenMat<double> A(10,10), B, C;  // 1
LaIndex I(1,9,2),                 // 2
LaIndex J(1,3,2);                 // 3

B.ref(A(I,I));                    // 4
B(2,3) = 3.1;                     // 5
C = B(LaIndex(2,4,2), J);         // 6
```

In lines (2) and (3) we declare indices `I` $= \{1, 3, 5, 7, 9\}$, and `J` $=\{1, 3\}$. Line (4) sets `B` to the specified 5x5 submatrix of `A`. The matrix B can used in any matrix expression, including accessing its individual elements, as in line (5). Note that `B(2,3)` is the same memory location as `A(5,7)`, so that a change to B will also modify the contents of A. Line (6) assigns the 2x2 submatrix of B to C. Note that C can also be referenced as `A( LaIndex(5,9,2), LaIndex(3,7,2))`.

Although LAPACK++ submatrix expressions allow one to access non-contiguous rows or columns, many of the LAPACK routines only allow submatrices with unit stride in the column direction. Calling an LAPACK++ routine with a non-contiguous submatrix columns *may* cause data to be copied into contiguous submatrix and can optionally generate a runtime warning to advise the programmer that data copying has taken place. (In Fortran, the user would need to need to do this by hand.)

3.1.3 Submatrices

Blocked linear algebra algorithms utilize submatrices as their basic unit of computation. It is crucial that submatrix operations be highly optimized. Because of this, LAPACK++ provides mechanisms for accessing rectangular subregions of a general matrix. These regions are accessed by *reference*, that is, without copying data, and can be used in any matrix expression.

Ideally, one would like to use familiar colon notation of Fortran 90 or Matlab for expressing submatrices. However, this modification of the C++ syntax is not possible without redefining the language specifications. As a reasonable compromise, LAPACK++ denotes submatrices by specifying a subscript range through the `LaIndex()` function. For example, the 3x3 matrix in the upper left corner of `A` is denoted as

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

This references $A_{ij}, i = 0, 1, 2\ j = 0, 1, 2$, and is equivalent to the `A(0:2,0:2)` colon notation used elsewhere. Submatrix expressions may be also be used as a destination for assignment, as in

```
A( LaIndex(0,2), LaIndex(0,2) ) = 0.0;
```

which sets the 3x3 submatrix of A to zero. Following the Fortran 90 conventions, the index notation has an optional third argument denoting the stride value,

```
LaIndex(start, end, increment)
```

If the `increment` value is not specified it is assumed to be one. The expression `LaIndex(s, e, i)` is equivalent to the index sequence

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

The internal representation of an index is not expanded to a full vector, but kept in its compact triplet format. The increment values may be negative and allow one to traverse a subscript range in the opposite direction, such as in `(10,7,-1)` to denote the sequence $\{10, 9, 8, 7\}$. Indices can be named and used in expressions, as in the following submatrix assignments,

```
LaGenMat<double> A(10,10), B, C;  // 1
LaIndex I(1,9,2),                 // 2
LaIndex J(1,3,2);                 // 3

B.ref(A(I,I));                    // 4
B(2,3) = 3.1;                     // 5
C = B(LaIndex(2,4,2), J);         // 6
```

In lines (2) and (3) we declare indices `I` $= \{1, 3, 5, 7, 9\}$, and `J` $=\{1, 3\}$. Line (4) sets `B` to the specified 5x5 submatrix of `A`. The matrix B can used in any matrix expression, including accessing its individual elements, as in line (5). Note that `B(2,3)` is the same memory location as `A(5,7)`, so that a change to B will also modify the contents of A. Line (6) assigns the 2x2 submatrix of B to C. Note that C can also be referenced as `A( LaIndex(5,9,2), LaIndex(3,7,2))`.

Although LAPACK++ submatrix expressions allow one to access non-contiguous rows or columns, many of the LAPACK routines only allow submatrices with unit stride in the column direction. Calling an LAPACK++ routine with a non-contiguous submatrix columns *may* cause data to be copied into contiguous submatrix and can optionally generate a runtime warning to advise the programmer that data copying has taken place. (In Fortran, the user would need to need to do this by hand.)

4 Driver Routines

This section discusses LAPACK++ routines for solving linear a system of linear equations

$$Ax = b,$$

where A is the **coefficient matrix**, b is the **right hand side**, and x is the **solution**. A is assumed to be a square matrix of order n, although underlying computational routines allow for A to be rectangular. For several right hand sides, we write

$$AX = B,$$

where the columns of B are individual right hand sides, and the columns of X are the corresponding solutions. The task is to find X, given A and B. The coefficient matrix A can be of the types show in Figure 4.

The basic syntax for a linear equation driver in LAPACK++ is given by

```
LaLinSolve(op(A), X, B);
```

The matrices `A` and `B` are input, and `X` is the output. `A` is an $M \times N$ matrix of one of the above types. Letting $nrhs$ denote the number of right hand sides in eq. 4, `X` and `B` are both rectangular matrices of size $N \times nrhs$. The syntax op(`A`) can denote either `A` or the transpose of `A`, expressed as `transp(A)`.

This version requires intermediate storage of approximately $M * (N + nrhs)$ elements.

In cases where no additional information is supplied, the LAPACK++ routines will attempt to follow an intelligent course of action. For example, if `LaLinSolve(A,X,B)` is called with a non-square $M \times N$ matrix, the solution returned will be the linear least square that minimizes $||Ax - b||_2$ using a QR factorization. Or, if A is declared as SPD, then a Cholesky factorization will be used. Alternatively, one can directly specify the exact factorization method, such as `LaLUFactor(F, A)`. In this case, if A is non-square, the factors return only a partial factorization of the upper square portion of A.

Error conditions in performing the `LaLinSolve()` operations can be retrieved via the `LaLinSolveInfo()` function, which returns information about the last called `LaLinSolve()`. A zero value denotes a successful completion. A value of $-i$ denotes that the ith argument was somehow invalid or inappropriate. A positive value of i denotes that in the LU decomposition, $U(i,i) = 0$; the factorization has been completed but the factor U is exactly singular, so the solution could not be computed. In this case, the value returned by `LaLinSolve()` is a null (0x0) matrix.

4.1 Memory Optimizations: Factorizing in place

When using large matrices that consume a significant portion of available memory, it may be beneficial to remove the requirement of storing intermediate factorization representations at the expense of destroying the contents of the input matrix A. For most matrix factorizations we require temporary data structures roughly equal to the size of the original input matrix. (For general banded matrices, one needs slightly more storage due to pivoting, which causes fill in additional bands.) For example, the temporary memory requirement of a square $N \times N$ dense non-symmetric factorization can be reduced from $N \times (N + nrhs + 1)$ elements to $N \times 1$. Such memory-efficient factorizations are performed with the `LaLinSolveIP()` routine:

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

Here the contents of A are overwritten (with the respective factorization). These "in-place" functions are intended for advanced programmers and are not recommended for general use. They assume the programmer's responsibility to recognize that the contents of `A` have been destroyed; however, they can allow a large numerical problem to be solved on a machine with limited memory.

5 Programming Examples

This code example solves the linear least squares problem of fitting N data points (x_i, y_i) with a dth degree polynomial equation

$$p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_d x^d$$

using QR factorization. It is intended for illustrative purposes only; there are more effective methods to solve this problem.

Given the two vectors x and y it returns the vector of coefficients $\mathbf{a} = \{a_0, a_1, a_2, \ldots, a_{d-1}\}$. It is assumed that $N \gg d$. The solution arises from solving the overdetermined Vandermonde system $Xa = y$:

$$\begin{bmatrix} 1 & x_0^1 & x_0^2 & \ldots & x_0^d \\ 1 & x_1^1 & x_1^2 & \ldots & x_1^d \\ \vdots & & & & \vdots \\ 1 & x_{N-1}^1 & x_{N-1}^2 & \ldots & x_{N-1}^d \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{bmatrix}$$

in the least squares sense, i.e., minimizing $||Xa - y||_2$. The resulting code is shown in figure 5.

6 ScaLAPACK++: an extension for distributed architectures

There are various ways to extend LAPACK++. Here we discuss one such extension, ScaLAPACK++ [2], for linear algebra on distributed memory architectures. The intent is that for large scale problems ScaLAPACK++ 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.

Achieving these goals while maintaining portability, flexibility, and ease-of-use can present serious challenges, since the layout of an application's data within the hierarchical memory of a concurrent computer is

```
void poly_fit(LaVector<double> &x,
      LaVector<double> &y, LaVector<double> &p)
{
    int N = min(x.size(), y.size());
    int d = p.size();

    LaGenMatDouble P(N,d);
    LaVectorDouble a(d);
    double x_to_the_j;

    // construct Vandermonde matrix
    for (i=0; i<N; i++)
    {
        x_to_the_j = 1;
        for (j=0; j<d; j++)
        {
            P(i,j) = x_to_the_j;
            x_to_the_j *= x(i);
        }
    }
    // solve Pa = y using linear least squares
    LaLinSolveIP(P, p, y);

}
```

Figure 5: LAPACK++ code example: polynomial data fitting.

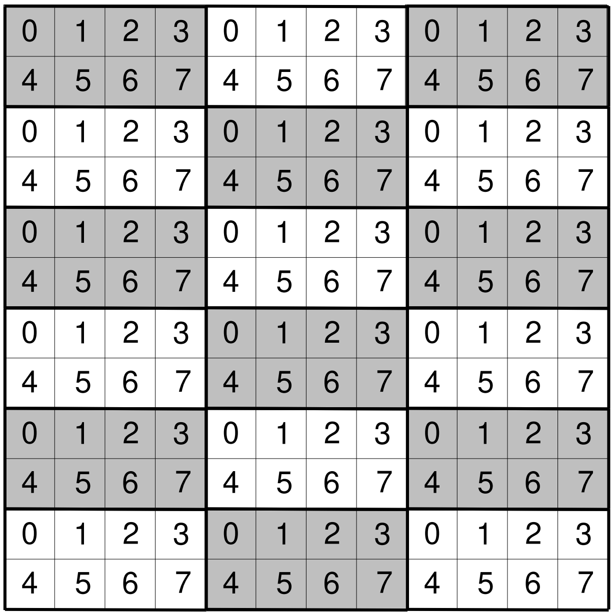

Figure 6: An example of block scattered decomposition over an 2x4 processor grid.

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 design of ScaLAPACK++ includes a general two-dimensional matrix decomposition that supports the most common block scattered encountered in the current litertature. 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 scheme until runtime. This is only possible with object-oriented programming languages, like C++, that support *dynamic-binding*, or *polymorphism* – the ability to examine an object's type and dynamically select the appropriate action [10]. In many applications the optimal matrix decomposition is strongly dependent on how the matrix is utilized in other parts of the program. Furthermore, it is often necessary to dynamically alter the matrix decomposition at runtime to accommodate special routines. The ability to support dynamic run-time decomposition strategies is one of the key features of ScaLAPACK++ that makes it integrable with scalable applications.

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 (Figure 6) that allows a wide variety of matrix mappings while enhancing scalability and maintaining good load balance for various factorization algorithms. Each node of the multicomputer runs a sequential LAPACK++ library that provides the object oriented framework to describe block algorithms on conventional matrices in each individual processor.

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 (Figure 6). The block and scattered decompositions may be regarded as special cases of the block scattered (SBS) decomposition. This 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

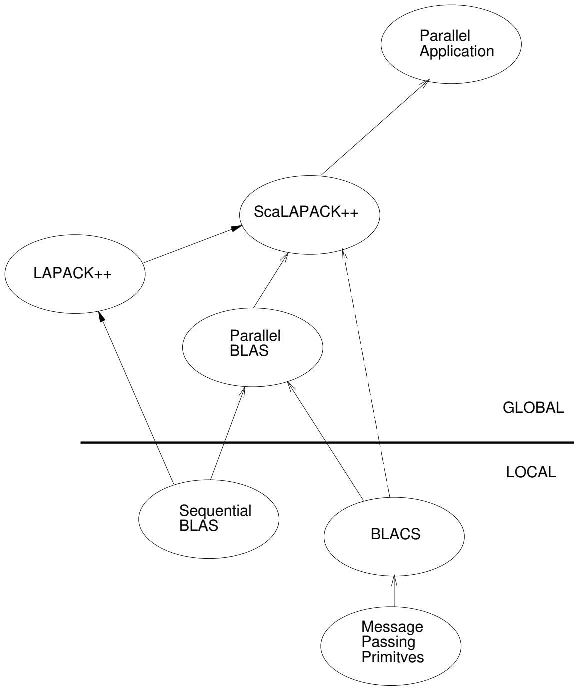

Figure 7: Design Hierarchy of ScaLAPACK++. In an SPMD environment, components above the horizontal reference line represent a global viewpoint (a single distributed structure), while elements below represent a per-node local viewpoint of data.

and/or columns are eliminated in successive steps, the SBS decomposition enhances scalability by ensuring processor load balance. Preliminary experiments of an object-based LU factorization algorithm using an SBS decomposition [2] suggest these algorithms scale well on multicomputers. For example, on the Intel Touchstone Delta System, a 520 node i860-based multicomputer, such algorithms can achieve nearly twelve Gflops (figure 8).

Parallelism is exploited through the use of distributed memory versions of the Basic Linear Algebra Subprogram (BLAS) [6] [3] that perform the basic computational units of the block algorithms. Thus, at a higher level, the block algorithms look the same for the parallel and sequential versions, and only one version of each needs to be maintained.

The benefits of an object oriented design (Figure 7) include the ability to hide the implementation details

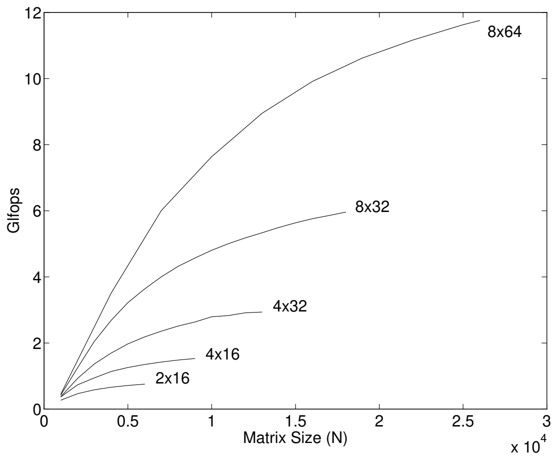

Figure 8: Performance of distributed LU factorization using an Square Block Scattered (SBS) matrix decomposition on the Intel Touchstone Delta system. Results are provided for various processor-grid configurations.

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.

7 Conclusion

We have presented a design overview for object oriented linear algebra on high performance architectures. We have also described extensions for distributed memory architectures. These designs treat both conventional and distributed matrices as fundamental objects.

In both the parallel and sequential aspects of LAPACK++, 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.

We have used the **inheritance** mechanism of the object oriented design to provide a common source code for both the parallel and sequential versions. 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 dynmanic binding mechanisms to achieve a truly portable 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 inlining** capabilities of C++ to reduce the function-call overhead usually associated with interfacing Fortran or assembly kernels.

In short, we have used various important aspects of object oriented mechanisms and C++ in the design of LAPACK++. 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, Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK User's Guide*, SIAM, Philadelphia, 1992.

[2] J. Choi,J. J. Dongarra, R. Pozo, D. W. Walker, *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, Frontiers of Massively Parallel Computing, McLean, Virginia, October 19-21, 1992.

[3] J. Choi and J. J. Dongarra and D. W. Walker, *PB-BLAS : Parallel Block Basic Linear Algebra Subroutines on Distributed Memory Concurrent Computers*, Oak Ridge National Laboratory, Mathematical Sciences Section, in preparation, 1993.

[4] R. Davies, *newmat07, an experimental matrix package in C++*, robertd@kauri.vuw.ac.nz, 1993.

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

[6] 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.

[7] J. J. Dongarra, R. Pozo, D. W. Walker, *LAPACK++ Users' Manual*, in preparation.

[8] J. J. Dongarra, R. Pozo, D. W. Walker, *An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures*, Object Orriented Numerics Conference (OONSKI), Sunriver, Oregon, May 26-27, 1993.

[9] Dyad Software Corporation, *M++ Class Library*, Bellevue, Washington, 1991.

[10] M. A. Ellis, B. Stroustrup, *The Annotated c++ Reference Manual*, Addison-Wesley, 1990.

[11] K. E. Gorlen, S. M. Orlow, P. S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley & Sons, Chichester, England, 1990.

[12] Rogue Wave Software, *Math.h++*, Corvallis, Oregon, 1992.

[13] 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.