Sparse Matrix Libraries in C++ for High Performance Architectures*

Jack Dongarra[§‡], Andrew Lumsdaine[◇], Xinhui Niu[◇]
Roldan Pozo[‡], Karin Remington[§]

[§]Oak Ridge National Laboratory, Mathematical Sciences Section
[‡]University of Tennessee, Dept. of Computer Science
[◇]University of Notre Dame, Dept. of Computer Science & Engineering

Abstract

We describe an object oriented sparse matrix library in C++ designed for portability and performance across a wide class of machine architectures. Besides simplifying the subroutine interface, the object oriented design allows the same driving code to be used for various sparse matrix formats, thus addressing many of the difficulties encountered with the typical approach to sparse matrix libraries. We also discuss the the design of a C++ library for implementing various iterative methods for solving linear systems of equations. Performance results indicate that the C++ codes are competitive with optimized Fortran.

1 Introduction

Sparse matrices are pervasive in scientific and engineering application codes. They often arise from finite difference, finite element, or finite volume discretizations of PDEs (e.g., in computational fluid dynamics) or from discrete, network-type problems (e.g., in circuit simulation). Over the past two decades, a number of research efforts have resulted in sparse matrix software. Our goal is to complement these efforts by developing a comprehensive sparse matrix package in C++. Several factors contribute to the difficulty of designing such a comprehensive library. Different computer architectures, as well as different applications, call for different sparse matrix data formats in order to best exploit registers, data locality, pipelining, and parallel processing. Furthermore, code involving sparse matrices tends to be complicated and less portable because assumptions of the underlying data formats are invariably entangled within the application code.

To address these difficulties, it is essential to develop codes which are "data format free", thus providing the greatest flexibility for using given algorithms (library routines) in various architecture/application combinations. In fact, the selection of an appropriate data structure can typically be deferred until link or run time. We describe **SparseLib++**, an object oriented C++ library for sparse matrix computations which provides a unified interface for sparse matrix computations across a variety of sparse data formats. We also describe **IML++**, an Iterative Methods Library in C++ for the iterative solution of linear systems of equations.

The design of these libraries is based on the following principles:

Clarity: Implementations of numerical algorithms should resemble the mathematical algorithms on which they are based. This is in contrast to Fortran, which can require complicated subroutine calls, often with parameter lists that stretch over several lines.

Reuse: A particular algorithm should only need to be coded once, with identical code used for all matrix representations.

Portability: Implementations of numerical algorithms should be directly portable across machine platforms.

High Performance: The object oriented library code should perform as well as optimized data-format-specific code written in C or Fortran.

*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, and NSF grant No. CCR-9209815.

The sparse matrix classes are all derived from an abstract matrix base class so that same driving algorithms can be used for various dense and sparse linear algebra computations across sequential and parallel architectures.

2 Sparse Matrix types

We have concentrated on the most commonly used data structures which occur in a large portion of application codes. The library can be arbitrarily extended to user-specific structures and will eventually grow. Matrix formats supported in the initial design of the library include:

Sparse Vector: List of nonzero elements with their index locations. It assumes no particular ordering of elements.

Coordinate Storage (COOR): List of nonzero elements with their respective row and column indices. This is the most general sparse matrix format, but it is not very space or computationally efficient. It assumes no ordering of nonzero matrix values.

Compressed Row Storage (CRS): Subsequent nonzeros of the matrix rows are stored in contiguous memory locations and additional integer arrays specify the column index of each nonzero and beginning offset of each row. It assumes no ordering among nonzero values within each row, but rows are stored in consecutive order.

Compressed Column Storage (CCS): Also commonly referred to as the Harwell-Boeing sparse matrix format [3]. Similar to CRS, except columns, rather than rows, are stored contiguously. Note that the CCS ordering of A is the same as the CRS of A^T.

Compressed Diagonal Storage (CDS): Designed primarily for matrices with relatively constant bandwidth, the sub- and super-diagonals are stored contiguously.

Jagged Diagonal Storage (JDS): . Also known as `ITPACK` storage. More space efficient than CDS at the cost of a gather/scatter operation.

Block Compressed Row Storage (BCRS): Useful when the sparse matrix is comprised of square dense blocks of nonzeros in some regular pattern. The savings in storage and reduced indirect addressing over CRS can be significant for matrices with large block sizes.

Skyline Storage (SKS): Also for variable band or profile matrices. Mainly used in direct solvers, but can also be used for handling the diagonal blocks in block matrix factorizations.

In addition, symmetric and Hermitian versions of most of these sparse formats will be supported. In such cases only an upper (or lower) triangular portion of the matrix is stored. The trade-off is slightly more complicated kernel operations with a somewhat different pattern of data access. Details of each data storage format are given in [1] and [5].

2.1 Sparse Matrix Operations

Our library contains the common computational kernels required for solving linear systems with many direct and iterative methods. The internal data structures of these kernels are compatible with the proposed Level 3 Sparse BLAS, thus providing the user with a large software base of Fortran module and application libraries. Just as the dense Level 3 BLAS [2] have allowed for higher performance kernels on hierarchical memory architectures, the Sparse BLAS allow vendors to provide optimized routines taking advantage of indirect addressing hardware, registers, pipelining, caches, memory management, and parallelism on their particular architecture. Standardizing the Sparse BLAS will not only provide efficient codes, but will also ensure portable computational kernels with a common interface.

There are two types of C++ interfaces to basic kernels. The first uses simple binary operators for multiplication and addition, and the second uses functional interfaces which can group triad and more complex operations (e.g. `Blas_Mat_Mult()`) . The binary operators provide a simpler interface, (e.g. $y = A * x$ denotes a sparse matrix-vector multiply) but may produce less efficient code (since the destination is not known at the `*` operator phase). On distributed memory architectures, the alignment of the temporary result of `A*x` with `y` could cause extra data movement.

The computational kernels include:

- sparse matrix products,
  $C \leftarrow \alpha \; op(A) \; B + \beta C$
- solution of triangular systems,
  $C \leftarrow \alpha D \; op(A)^{-1} \; B + \beta C$
- reordering of a sparse matrix (permutations),
  $A \leftarrow A \; op(P)$
- conversion of one data format to another,
  $A' \leftarrow A$

Initial $r^{(0)} = b - Ax^{(0)}$
for $i = 1, 2, \ldots$
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
  if $i = 1$
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1}p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence;
end

```
r = b - A*x;
for (int i = 1; i < maxiter; i++) {
    z = M.solve(r);
    rho = dot(r, z);
    if (i == 1)
       p = z;
    else {
       beta = rho1/ rho0;
       p = z + p * beta;
    }
    q = A*p;
    alpha = rho1 / dot(p, q);
    x += alpha * p;
    r -= alpha * q;
    if (norm(r)/norm(b) < tol) break;
}
```

Figure 1: Comparison of an algorithm for the preconditioned conjugate gradient method and the corresponding IML++ routine.

where α and β are scalars, B and C are rectangular matrices, D is a (block) diagonal matrix, A and A' are sparse matrices, and $op(A)$ is either A or A^T.

2.2 Matrix Construction and I/O

In dealing with issues of I/O, the C++ library is presently designed to support reading and writing to Harwell-Boeing format sparse matrix files [3]. These files are inherently in compressed column storage; however, since sparse matrices in the library can be transformed between various data formats, this is not a severe limitation. File input is embedded as another form of a sparse matrix constructor. A file can also be read and transformed into another format using conversions and the `iostream` operators. In the future, the library will also support other matrix file formats, such as MATLAB™ compatible format, and IEEE binary formats. Sparse matrices can also be initialized from conventional data and index vectors, thus allowing a universal interface to import data from C or Fortran modules.

3 Iterative Solvers

One motivation for this work is that high level matrix algorithms, such as those found in [1], can be easily implemented in C++. For example, consider the preconditioned conjugate gradient algorithm, used to solve $Ax = b$, with preconditioner M. The comparison between the pseudo-code and the C++ listing appears in Figure 1. Here the operators such as `*` and `+=` have been overloaded to work with matrix and vectors formats. This code fragment works for all of the supported sparse storage classes and makes use of data and architecture specific computational kernels (such as the proposed Level 3 Sparse BLAS [4]).

Various iterative methods, as described in Barrett *et al.* [1], have been incorporated into the design of IML++, an Iterative Methods Library in C++. The methods supported by the design of IML++, together with their preconditioned counterparts, include:

- Jacobi SOR (SOR)
- Conjugate Gradient (CG)
- Conjugate Gradient on Normal Equations (CGNE, CGNR)
- Generalized Minimal Residual (GMRES)
- Minimum Residual (MINRES)
- Quasi-Minimal Residual (QMR)
- Chebyshev Iteration (Cheb)
- Conjugate Gradient Squared (CGS)
- Biconjugate Gradient (BiCG)
- Biconjugate Gradient Stabilized (Bi-CGSTAB)

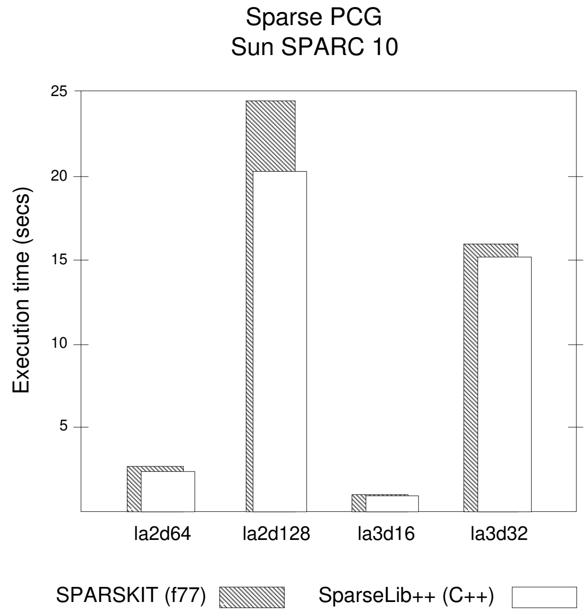

Figure 2: Performance comparison of C++ (g++) vs. optimized Fortran (f77 -O) on a Sun SPARC 10.

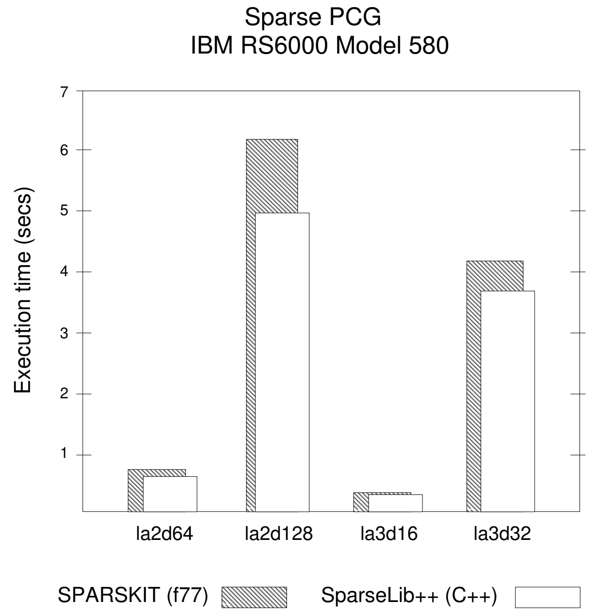

Figure 3: Performance comparison of C++ (xlC) vs. optimized Fortran (xlf -O) on an IBM RS/6000 Model 580.

Although iterative methods have provided much of the motivation for SparseLib++, many of the same operations and design issues apply to direct methods as well. In particular, some of the most popular preconditioners, such as Incomplete LU Factorization (ILU) [6], have components quite similar to direct methods.

4 Efficiency

4.1 Performance

To get some measure of the efficiency of our C++ class designs, we tested the performance of our library modules against the public-domain Fortran sparse matrix package SPARSKIT [7]. The SPARSKIT package was designed as a "tool kit", with one of its basic goals being to facilitate the transfer of data among researchers in sparse matrix computations, and peak efficiency across machines was not of primary concern. As such, we recognize that it cannot be expected to provide ultimate performance on any particular architecture. We use it as a basis for comparison only to demonstrate that the performance of our library is at least comparable to good Fortran codes.

Figures 2 and 3 illustrate the performance of the PCG method with diagonal preconditioning coded with both our C++ library and with SPARSKIT routines. The test matrices correspond to 2D and 3D finite difference Laplacian operators, and we solve $\nabla^2 u = 0$ on grids of sizes 64×64, 128×128, $16 \times 16 \times 16$, and $32 \times 32 \times 32$. In all cases we utilized full optimization of each compiler.

Table 1 shows a more detailed comparison of the CPU times for Sparskit, for the C package, for IML++/SparseLib++ with an operator overloading interface, and for IML++/SparseLib++ with a functional interface. In these tests, the C++ modules run slightly more efficiently than their Fortran counterparts. This is true because any overhead in the C++ sparse matrix classes has been minimized, and because we exploit the low-level Sparse BLAS kernels. We expect that a Fortran package which also exploited these kernels would achieve virtually the same performance as our C++ library. The main point is that it is possible to have an elegant coding interface (as shown in Figure 1) and still maintain performance that is competitive with conventional Fortran modules.

4.2 Memory Resources

The memory usage requirements of the C++ sparse matrix objects are nearly identical to conventional storage requirements in C or Fortran. For example, an $M \times N$ compressed column matrix with nz nonzeros, requires nz floating point numbers and $nz + N$ integer indices. The sparse matrix classes require only two or three additional words for bookkeeping.

Problem	RS/6000 CPU Time Fortran	C	C++/fun	C++/op	SPARC 10 CPU Time Fortran	C	C++/fun	C++/op
la2d64	0.74	0.55	0.65	0.95	3.09	2.55	2.583	3.27
la2d128	6.26	4.68	5.37	7.65	25.41	20.38	20.95	27.53
la3d16	0.27	0.21	0.24	0.32	1.15	0.95	1.00	1.28
la3d32	4.41	3.49	3.89	5.14	18.01	14.93	15.25	19.48

Table 1: Comparison between Fortran, C, and C++ codes for solving Poisson equations on RS/6000 Model 580 and SPARC 10. C++/op and C++/fun respectively indicate SparseLib using the operator and functional interfaces.

When computing `y = A*x`, a temporary vector is allocated, but the assignment into `y` is performed by shallow assignment, rather than an $O(N)$ memory copy. Dense vectors and matrices utilize reference-count schemes for automatic garbage collection.

5 Conclusion

We have demonstrated with our sparse matrix library of C++ classes, SparseLib++, that one can abstract the underlying storage format details without sacrificing performance. The Sparse BLAS provide a framework that can optimized for a given architecture while maintaining a consistent interface.

Our iterative methods library, IML++, can work with any C++ matrix class employing basic operations, including distributed sparse and dense matrices. In effect, we have separated the details of the underlying data structure from the mathematical algorithm. The result is a library of high level mathematical denotations which can run on distributed networks, multicomputers, and single node workstations without modification. These libraries are essentially "data format free", providing increased portability, readability, and reliability.

References

[1] R. Barrett et al. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Press, Philadelphia, 1994.

[2] J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16:1–17, 1990.

[3] I. Duff, R. Grimes, and J. Lewis. Sparse matrix test problems. *ACM Trans. Math. Soft.*, 15:1–14, 1989.

[4] I. Duff, M. Marrone, and G. Radicati. A proposal for user level sparse BLAS. Technical report, CERFACS TR/PA/92/85, 1992.

[5] M. A. Heroux. A proposal for a sparse BLAS toolkit. Technical report, CERFACS TR/PA/92/90, 1992.

[6] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148–162, 1977.

[7] Y. Saad. Sparskit: A basic toolkit for sparse matrix computations. Technical report, NASA Ames Research Center TR 90-20, 1990.