



THE MATRIX TEMPLATE LIBRARY: GENERIC COMPONENTS FOR HIGH-PERFORMANCE SCIENTIFIC COMPUTING

By Jeremy G. Siek and Andrew Lumsdaine

THE STANDARD TEMPLATE LIBRARY WAS RELEASED IN 1995 AND ADOPTED INTO THE ANSI C++ STANDARD SHORTLY THEREAFTER.¹ WHEN WE FIRST DISCOVERED THE STL, IT BECAME APPARENT TO US THAT *GENERIC PROGRAMMING*, THE FUNDAMEN-

tal paradigm underlying the STL, was an important and powerful new software development methodology—and this has been borne out by the tremendous success of the STL for general-purpose programming. Not so obvious then, however, was how (or even if) generic programming could apply to other problem domains. To investigate the merit of this approach for scientific computing, we embarked on a research project to apply generic programming to high-performance numerical linear algebra.

That effort brought forth the Matrix Template Library, a generic component library for scientific computing.² Although MTL consists of a relatively small number of software components, its power and utility arise from the composability of the components and the generic nature of the algorithms. That is, the components can be composed arbitrarily to produce an extremely wide variety of matrix formats. Similarly, each algorithm can operate on any matrix type defined in this fashion. The resulting linear algebra functionality far exceeds that of libraries that are not based on component technology, while requiring orders of mag-

nitude fewer lines of code.

Composability and genericity are all well and good, but to many scientific computing users the advantages of an elegant programming interface are secondary to issues of performance. Given MTL's heavy use (and many layers) of abstraction, we might naturally assume that there is a corresponding performance penalty. It turns out that generic programming is a powerful tool with respect to performance as well—in two regards:

- Properly designed abstractions (in conjunction with modern compilers) incur no performance penalty per se. That is, generic components are as efficient as their handwritten counterparts.
- High-level performance-tuning mechanisms (such as cache or register-blocking schemes) can be described in a generic fashion, giving users vendor-optimized performance levels in a portable and easy-to-tune fashion. This has been borne out in our own experiments—MTL can match the performance of vendor-tuned libraries on a number of platforms.

Generic programming

As Alexander Stepanov has described, the generic-programming process applied to a particular problem domain takes the following basic steps:

1. Identify useful and efficient algorithms.
2. Find their generic representation (parameterize each algorithm such that it makes the fewest possible requirements of the data on which it operates).
3. Derive a set of (minimal) requirements that allow these algorithms to run and to run efficiently.
4. Construct a framework based on classifications of requirements.

In applying this process to numerical linear algebra, the first step can be directly motivated by the mathematical definition of *linear algebra*. That is, we need algorithms that implement the basic axiomatic operations of linear algebras: multiplying a vector by a scalar, adding two vectors, or applying a linear transformation to a vector. In addition to these operations, we can include operations that give our linear space extra structure, such as inner products and norms as well as algorithms for working with dual spaces (that is, a transpose operation). Thus, our set of useful and efficient algorithms consists of just these six (respectively performing the desired operations described above): `scale()`, `sum()`, `mult()`, `dot()`, `norm()`, and `transpose()`.

Cafe Dubois

Lapack, Version 3.0 is available

Lapack is a library of numerical linear algebra subroutines designed for high performance on workstations, vector computers, and shared-memory multiprocessors. Jack Dongarra (dongarra@cs.utk.edu) recently announced the release of Version 3.0, which introduces new routines, as well as extending the functionality of existing routines.

According to the announcement, the most significant new routines and functions are singular value decomposition computed by the divide-and-conquer method; SVD-based divide-and-conquer least-squares solver; new simple and expert drivers for the generalized nonsymmetric eigenproblem; new generalized symmetric eigenproblem drivers; symmetric eigenproblem drivers using fast but relatively robust eigenvector computations; a faster QR decomposition with column pivoting; a faster solver for the rank-deficient least squares problem; a blocked version of xTZROF (xTZRF) and associated xORMRZ/xUNMRZ solver for the generalized Sylvester equation; and computational routines (xTGEXC, xTGEN, xTGSNA).

For complete information, please refer to the release notes file in the Lapack directory on netlib (<http://www.netlib.org/lapack/releasenotes>). There are Lapack bindings available for Fortran 90, C, and Java (<http://www.netlib.org/lapack90>, <http://www.netlib.org/clapack>, and <http://www.netlib.org/java/f2j>).

The third edition of the *Lapack Users' Guide* is in preparation and will be available soon at <http://www.netlib.org/lapack/lug/lapacklug.html>.

Little languages

I continually find myself needing to write a parser for a small language. Often this is useful when translating something in one format to another, or as an input language for a small tool. There is a neat little tool to help do this written by John Aycocock of the University of Victoria. John calls it "a framework for implementing little languages, in 100% pure Python." Since John didn't name it, I'll call it LLP. A paper about LLP and download information is at <http://csr.uvic.ca/aycock/python>. I used LLP when I wrote Pyfort (Sept./Oct. issue). The entire LLP package has some facilities that aren't needed for a lot of simple applications. As you read about it, be aware that you're not locked into using the whole thing if you don't want to.

One thing to like about LLP is that it uses an algorithm called the Earley algorithm; it requires less experience to write a grammar for this tool than for LR(1) parsers such as yacc. Beginners at yacc often get as far as their first "shift/reduce error" and give up.

Another fun thing about it is that it uses introspection. Believe it or not, you input your grammar as comments in the functions that will carry out your actions. Likewise, you enter your lexical expressions as comments in the functions that will create tokens after the recognizer spots them. The tool examines the functions you added to it and extracts the grammar. This turns out to be strange but very effective.



Floating-point exceptions and the Borland compiler

Dan Kerner (kerner@civilized.com) is a senior programmer for Civilized Software. Civilized Software's flagship product is MLAB, a program for curve fitting, solving systems of ordinary differential equations, statistics, and graphics. For more information, see the Web site at <http://www.civilized.com>. Dan uses the Borland compiler for Windows. Dan writes, "I recently finished a paper describing how to handle floating-point exceptions in 32-bit computer programs running on Windows 95/98/NT PCs." Unfortunately, this same method does not work for Microsoft Visual C++, so the audience for the paper is too small for this column, but Dan will be happy to e-mail you the paper if you are interested.

Conferences past and present

The Open Source Software Conference put on by O'Reilly and Associates (<http://www.oreilly.com>) in August was a really great affair. They combined conferences on six major open-source software components (Linux, Sendmail, Perl, Python, Apache, and Tcl/Tk) with an open-source business track. This was just a great idea; you could attend sessions in any of the conferences. Every conference always has some dead spots where you are not interested in the subject, so rather than suffer through a boring session, you can go learn something you wouldn't ordinarily have time for. This is a very cool idea.

Bill Joy from Sun Microsystems described a new "Community Source" license scheme that you might call "Sorta-Open Software." He said Sun was committed to releasing its sources under this license, although not all at once.

The next great conference is the 8th International Python Conference, 24-27 January, in Alexandria, Virginia. Sign up at <http://www.python.org>. Be there or be square.

—Paul Dubois, pauldubois@home.com

More about MTL

As with everything these days, to find out more about MTL you can refer to the official MTL Web site: <http://lsc.nd.edu/research/mtl>. This site contains complete documentation of the MTL as well as the freely available source code. In addition, links to some of the other generic programming efforts mentioned in this article (such as the Standard Template Library, Iterative Template Library, and Generic Graph Component Library) are available there.

Let's look at the most interesting of these operations (`mult()`) and consider how we can make it generic. To be generic, we would like to realize the operation, say, $z \leftarrow Ax$ for any concrete representation of vectors x and z as well for any matrix (linear operator) A . We can implement type independence (at least syntactically) in C++ by using templates. Thus, we could prototype a generic `mult()` as follows:

```
template <class Matrix, class VecX, class VecZ>
void mult (const Matrix& A, const VecX& x, VecZ z);
```

The next step is to define the body of `mult()` such that it can work with arbitrary types (having a suitably defined interface). At first, this might seem impossible. After all, there are myriad matrix types: sparse, dense, rectangular, banded, column oriented, row oriented, and so on.

Interestingly, the mathematical description of a matrix-vector product points us in the right direction. We can write down what we mean by matrix-vector product in an N -dimensional space this way:

$$z_i = \sum_j a_{ij} x_j.$$

One textual interpretation of this mathematical statement is the following. Let the matrix A be such that each element in A has a corresponding row and column index, i and j , respectively. For each element in A , sum the product of that element with the j th element of x into the i th element of z .

This leads to the next step of our generic-programming process: deriving a set of minimal requirements for the algorithm.

In this case, our textual description of the algorithm provides those requirements. To realize a generic `mult()`, we must be able to

- visit each element of A (optionally skipping zeroes),
- access the value of (de-reference) each element of A ,
- access the row and column indices of each element of A , and
- randomly access values (that is, given some integer k , access the k th value) of the vectors x and z .

Note what is not required. We need not visit the elements of A in any particular order. Nor must A be of any particular shape, nor must zero values be stored, nor must we visit every

element of x and z (or visit them in any particular order).

We can implement the requirements that we do face using an interface similar to STLs. As with the STL, *iterators* form the principal interface to data types (for using `operator++()` and `operator*` for traversal and access). We extend the STL notion of iterator with respect to a matrix in two ways. First, MTL has a two-level hierarchy of iterators to traverse matrices (to reflect the objects' 2D nature). Second, iterators over matrix elements can provide row and column index information as well as be de-referenced for the matrix element value.

The body of the `mult()` algorithm is then:

```
{
    typename Matrix::const_iterator i;
    typename Matrix::OneD::const_iterator j;

    for (i = A.begin(); i != A.end(); ++i)
        for (j = (*i).begin(); j != (*i).end(); ++j)
            z[row(j)] += *j * x[column(j)];
}
```

For those who might be unfamiliar with C++, or with the particulars of STL-style C++ generic programs, this algorithm simply embodies the textual description of the `mult()` algorithm. The two nested loops serve to iterate over all the elements of A in the order they are stored in memory—whether that be by row, column, or diagonal. Thus, for row-oriented matrices, the inner loop performs a dot-product operation on rows of A with vector x ; for column-oriented matrices, the inner loop takes linear combinations of columns of A scaled by elements of x (in an `axpy()`-like manner).

With this basic set of requirements in place, we can move to the last step of the generic-programming process—framework construction.

Data types in MTL

Although our tour thus far through MTL's generic-programming process has described what the interfaces to matrices and vectors should look like, it leaves open the issue of the underlying implementation of those objects—allowing, in some sense, almost infinite flexibility, provided the interface conditions are satisfied. Within MTL, we sought to provide the largest possible variety of concrete matrix types (particularly those that are commonly used elsewhere) while also requiring only a small number of actual components. To accomplish these simultaneous goals, we

developed a compositional model for matrix types consisting of five basically orthogonal categories:

- **Element type:** The type of the individual matrix element—for example, a real or complex floating-point number.
- **1D storage:** Containers of elements, similar to STL containers—for example, dense, compressed.
- **2D storage:** Abstractly, a container of 1D storage containers (not necessarily concretely so).
- **Orientation:** The mapping of indices between the 2D storage structure and the matrix—for example, row-major, column-major.
- **Shape:** The outer envelope describing the structure of the matrix—for example, rectangular, banded, triangular.

MTL matrices. Combining the appropriate components to specify particular matrix types in MTL can be a somewhat complex process, so we have created a *matrix type specification interface* that simplifies this process for the user. The matrix interface can be described with a simple grammar.³

```
Matrix:      matrix<EltType, Shape, Storage,
              Orientation>::type
EltType:    float | double | doubledouble | complex
            <float> | complex<double> | complex
            <doubledouble> | (any other field type)
Shape:      rectangle | diagonal | banded | triangle
            | symmetric | hermitian |
TwoD Storage: dense | packed | banded | bandedview |
              compressed | array of OneD | envelope
              | (other TwoD)
OneD:       dense vector | compressed vector | pair
            vector | fortran-style list | linked
            list | red-black tree |
Orientation: rowmajor | columnmajor
```

This specification is open-ended. A new component type can be used with any component group simply by meeting the (deliberately minimal) interface specification for that group. For example, we could use an interval class for *Elt-Type*. A combinatorial number of matrices can be constructed from the basic components already defined in MTL. We can use the current collection of just 21 MTL matrix components with the standard numerical types of instance to construct literally *thousands* of different matrix types!

The following code examples show how to specify MTL matrices corresponding to some of the more commonly used matrix types.

Jeremy G. Siek is a PhD candidate in computer science at the University of Notre Dame. His interests include generic programming, high-performance libraries for C++, and language/compiler support for active libraries. He received a BS in mathematics and an MS in computer science at Notre Dame. Contact him at SGI, 1600 Amphitheatre Pkwy MS 7U-178, Mountain View, CA 94043; jsiek@lsc.nd.edu.



Andrew Lumsdaine is an associate professor in the Department of Computer Science and Engineering at the University of Notre Dame and is presently enjoying a sabbatical at Lawrence Berkeley National Laboratory. His research interests include generic programming, high-performance scientific computing, and parallel and distributed processing. He received his undergraduate and graduate education (along with a PhD) from MIT. He is a member of the IEEE, SIAM, and ACM. Through the summer of 2000, he can be contacted at M/S 50B-2239, One Cyclotron Rd., Berkeley, CA 94720; lums@lsc.nd.edu.



```
typedef matrix<double, rectangle<>,
              compressed<>, row_major>::type CompressedRow;
```

```
typedef matrix<double, rectangle<>,
              dense<>, column_major>::type FortranMatrix;
```

```
typedef matrix<complex<double>, banded<>,
              packed<>, column_major>::type BLASComplexPacked;
```

```
typedef matrix<double, banded<>,
              banded<>, column_major>::type BLASBanded;
```

To construct a matrix, we need only invoke the constructor with the matrix dimensions. We can create a matrix from data in a file (for example, in Matrix Market or Harwell Boeing format) by using the appropriate matrix stream. For interfacing to other libraries, we can also create MTL matrices from existing data by specifying that the matrix has “external” storage and passing the appropriate data pointers into the matrix constructor. Here are several examples.

```
// Create an empty 1000 x 1000 matrix
FortranMatrix A(1000, 1000);
```

```
// Create a matrix from a file
matrix_market_stream<double> mms (filename);
CompressedRow B(mms);
```

```
// Create a matrix from existing data
typedef matrix<double, rectangle<>,
              compressed<int, external>, column_major>::type
              ExtCompRow;
ExtCompRow C(m, n, nnz, values, indices, row_ptrs);
```

For the most part, MTL users do not need to access individual elements of a matrix once it is constructed because the MTL algorithms provide most operations. However, users may want to construct their own matrix algorithms. With this in mind, there are several ways to access the elements of an MTL matrix. The main access method is through iterators, as we showed in the previous matrix-vector multiply example. Users can also access an MTL matrix in more traditional ways. For example, a single matrix element can be accessed with the operator `(i, j)`, 1D slices can be accessed with the operator `[i]`, and a submatrix can be obtained with the `sub_matrix(i, j, m, n)` method. The type of a 1D slice depends on the matrix type (the slice might be a row, a column, or a diagonal). The `rows()` and `columns()` helper functions provide an interface for creating different views of the same matrix regardless of the underlying storage matrix layout.

```
// Get an Element
FortranMatrix::value_type w = A(4, 5);
// Get a slice, in this case a column
FortranMatrix::Column c = A[4];
// Get a row explicitly
FortranMatrix::Row r = rows(A)[3];
```

MTL matrices (and vectors) are reference counted, so the user need not be concerned with memory management.

MTL vectors. MTL provides both dense-vector and several sparse-vector types, implemented using standard STL components (the MTL layer adds reference counting and handle-based semantics). The MTL vectors export the same interface as STL containers, including the `begin()` and `end()` iterator accessors and the usual operator `[i]`. In addition, MTL vectors provide a convenient way to access subvector views.

```
// subrange vector s refers to elements [10, 30)
dense1D<double>::subrange_type s = x(10, 30);
```

MTL algorithms and adaptors

The MTL provides basic (abstract) linear algebra functionality and also provides a number of utility functions. MTL thus provides functionality basically equivalent to that available with the BLAS Levels-1, 2 and 3.⁴⁻⁶ However, in contrast to the BLAS, MTL algorithms work with a larger number of matrix types (any matrix type that can be constructed within MTL), such as sparse matrices, and also with any element type, not just single, double, and complex.

Algorithms. Table 1 lists the principle algorithms included in MTL. In the table, `alpha` and `s` are scalars, `x`, `y`, and `z` are 1D containers, `A`, `B`, `C`, and `E` are matrices, and `T` is a triangular matrix. MTL does not define different operations for each permutation of transpose, scaling, and striding as is typically necessary in traditional libraries. Instead, only one algorithm is provided, but it can be combined with the use of `strided` and `scaled` vector adaptors, or the `trans()` modifier, to create the permutations as described later.

Adaptors. One novel aspect of the MTL algorithm interface is the way we use adaptors to provide algorithm flexibility at a small constant implementation cost and with little or no extra runtime cost. Algorithm flexibility improves performance by allowing a single function to carry out entire families of operations. For example, you might wish to scale a vector while adding it to another ($y \leftarrow \alpha \times x + y$). This is the operation carried out by the `daxpy()` BLAS function. By using adaptors, the MTL `add()` operation can handle any combination of scaling or striding without loss of performance. The adaptor modifies the behavior of the vector inside of the algorithm. In the case of scaling, this causes the elements to be multiplied as they are accessed. The call to `scaled()` does not perform the multiplications before the call to `add()`, as this would hurt performance, but instead the multiplications happen during the `add()`. With a good optimizing C++ compiler, there is no extra overhead induced by the adaptors.

```
//  $y \leftarrow \alpha x + y$ 
add(scale(x, alpha), x, y);

// equivalent operation using BLAS
daxpy(n, alpha, xptr, 1, yptr, 1);
```

The transpose of a matrix can be used in an algorithm with the `trans()` adaptor. This adaptor performs a type conversion (swapping matrix orientations) at compile time—there is zero runtime cost. The next example shows how the matrix-vector multiply algorithm (generically written to compute $z \leftarrow A \times x + y$) can also compute $y \leftarrow A^T \times (\alpha x) + \beta y$.

```
//  $y \leftarrow A^T \times (\alpha x) + \beta y$ 
mult(trans(A), scaled(x, alpha), scaled(y, beta),
      y);

// equivalent operation using BLAS
dgemv('T', M, N, alpha, A_ptr, A_ld, x_ptr, 1,
      beta, y_ptr, 1);
```

Example: preconditioned GMRES(m)

One important use for MTL is to rapidly construct high-quality, high-performance numerical libraries. Although not necessary, a generic approach can be used when developing these libraries as well, resulting in reusable scientific software at a higher level. To demonstrate how you might use MTL for a nontrivial high-level library, we show the complete implementation of the preconditioned GMRES(m) algorithm⁷ in Figure 1 (taken from our Iterative Template Library collection).

The basic algorithmic steps (corresponding to the GMRES algorithm⁷) are given in the comments, and the calls to MTL in the body of the algorithm should be fairly clear. Some of the other code might seem somewhat impenetrable at first glance, so we'll quickly walk through the more difficult statements.

The algorithm parameterizes GMRES in some important ways, as the template statement shows on lines 1 and 2. The matrix and vector types are parameterized, so that any matrix type can be used. In particular, you can use matrices having any *element type*—real or complex. In fact, matrices without explicit elements at all (matrix-free operators⁸) can be used. For MTL matrices, the generic MTL `mult()` will generally suffice. For non-MTL matrices, or matrix-free operators, a suitably overloaded `mult()` must be provided.

There are also two other type parameterizations of interest, the `Preconditioner` and the `Iteration`. Similar to the matrix parameterization, the preconditioner type is parameterized so that arbitrary preconditioners can be used. The preconditioner simply must be callable with the `solve()` algorithm. The `Iteration` type parameter lets users control the stopping criterion for the algorithm. (ITL includes predefined stopping criteria.)

The `using namespace mt1` statement on line 6 lets us access MTL functions (which are all declared within the `mt1` namespace) without explicitly using the `mt1::` scope operator. The use of a namespace helps prevent name clashes with other libraries and user code.

On line 7, we use the internally defined `typedef` for the `value_type` to determine the type of the individual elements of the matrix. All MTL matrix and vector classes have an accessible type member called `value_type` that specifies the type of the element data. By using this internal type, rather than a fixed type, we can make the algorithm generic with respect to element type.

Although the entire algorithm fits on a single page, it is not a toy implementation—it is both high quality and high performance. This is where the power of reusable software components becomes apparent. Now that a generic GMRES(m)

Table 1. MTL linear algebra operations.

Function name	Operation
<code>scale(x, alpha)</code>	$x \leftarrow \alpha x$
<code>scale(A, alpha)</code>	$A \leftarrow \alpha A$
<code>add(x, y)</code>	$y \leftarrow x + y$
<code>add(x, y, z)</code>	$z \leftarrow x + y$
<code>add(x, y, z, w)</code>	$w \leftarrow x + y + z$
<code>add(A, C)</code>	$C \leftarrow A + C$
<code>add(A, B, C)</code>	$C \leftarrow A + B$
<code>mult(A, x, y)</code>	$y \leftarrow A \times x$
<code>mult(A, x, y, z)</code>	$z \leftarrow A \times x + y$
<code>mult(A, B, C)</code>	$C \leftarrow A \times B$
<code>mult(A, B, C, E)</code>	$E \leftarrow A \times B + C$
<code>tri_solve(T, x, y)</code>	$y \leftarrow T^{-1} \times x$
<code>tri_solve(T, B, C)</code>	$C \leftarrow T^{-1} \times B$
<code>rank_one(x, A)</code>	$A \leftarrow x \times y^T + A$
<code>rank_two(x, y, A)</code>	$A \leftarrow x \times y^T + y \times x^T + A$
<code>s = dot(x, y)</code>	$s \leftarrow x^T \cdot y$
<code>s = dot_conj(x, y)</code>	$s \leftarrow x^T \cdot \bar{y}$
<code>transpose(A)</code>	$A \leftarrow A^T$
<code>transpose(A, B)</code>	$B \leftarrow A^T$
<code>s = one_norm(x)</code>	$s \leftarrow \sum_i x_i $
<code>s = one_norm(A)</code>	$s \leftarrow \max_j (\sum_i a_{ij})$
<code>s = two_norm(x)</code>	$s \leftarrow (\sum_i x_i^2)^{1/2}$
<code>s = inf_norm(x)</code>	$s \leftarrow \max x_i $
<code>s = inf_norm(A)</code>	$s \leftarrow \max_j (\sum_i a_{ij})$
<code>s = sum(x)</code>	$s \leftarrow \sum_i x_i$
<code>s = max(x)</code>	$s \leftarrow \max(x_i)$
<code>s = min(x)</code>	$s \leftarrow \min(x_i)$
<code>i = max_index(x)</code>	$i \leftarrow \text{index of max } x_i $
<code>ele_mult(x, y, z)</code>	$z \leftarrow y \otimes x$
<code>ele_div(x, y, z)</code>	$z \leftarrow y \oslash x$
<code>set(x, alpha)</code>	$x_i \leftarrow \alpha$
<code>set(A, alpha)</code>	$A \leftarrow \alpha$
<code>set_diag(A_{ij}, alpha)</code>	$A_{ij} \leftarrow \alpha$
<code>copy(x, y)</code>	$y \leftarrow x$
<code>copy(A, B)</code>	$B \leftarrow A$
<code>swap(x, y)</code>	$y \leftrightarrow x$
<code>swap(A, B)</code>	$B \leftrightarrow A$

has been implemented, tested, and debugged, programmers wanting to use GMRES are forever spared the work of implementation, testing, and debugging GMRES themselves, saving time and reliability.

Performance

To compare MTL with other available libraries (both public domain and vendor-supplied), we performed a set of experiments involving dense matrix-matrix multiplication, dense matrix-vector multiplication, and sparse matrix-vector multiplication.

Dense matrix-matrix multiplication. Figure 2a shows

```

1  template <class Matrix, class Vector, class VectorB,
2      class Preconditioner, class Iteration>
3  int gmres (const Matrix &A, Vector &x, const Vector B &b;
4      const Preconditioner &M, int m, Iteration& outer)
5  {
6      using namespace mtl;
7      typedef typename Matrix::value_type T;
8      typedef matrix<T, rectangle<>, dense<>,
9          column_major>::type InternalMatrix;
10     InternalMatrix H(m+1, m), V(x.size(), m+1);
11     Vector s(m+1), w(x.size()), r(x.size()), u(x.size());
12     std::vector< givens_rotation<T> > rotations (m+1);
13
14     mult(A, scaled(x, -1.0), b, w);
15     solve(M, w, r); //  $r_0 = b - Ax_0$ 
16     typename Iteration::real beta = abs(two_norm(r));
17
18     while (! outer.finished(beta)) { // outer iteration
19         copy (scaled(r, 1./beta), v[0]); //  $v_1 = r_0 / \|r_0\|$ 
20         set (s, 0.0);
21         s[0] = beta;
22         int j = 0;
23         Iteration inner (outer.normb(), m, outer.tol());
24
25         do { // Inner iteration
26             mult(A, V[j], u);
27             solve(M, u, w);
28             for (int i = 0; i <= j; i++) {
29                 H(i, j) = dot_conj(w, V[i]); //  $h_{ij} = \langle Av_j, v_i \rangle$ 
30                 add(w, scaled(V[i], -h(i, j)), w); //  $\hat{v}_{k+1} = Av_j - \sum_{i=1}^j b_{ij}v_i$ 
31             }
32             H(j + 1, j) = two_norm(w); //  $b_{j+1,j} = \|\hat{v}_{j+1}\|$ 
33             copy(scaled(w, 1./H(j + 1)), V[j+1]); //  $v_{j+1} = \hat{v}_{j+1}/b_{j+1,j}$ 
34
35             // QR triangularization of H
36             for (int i = 0; i < j; i++)
37                 rotations [i].apply(H(i, j), H(i+1, j));
38
39             rotations [j] = givens_rotation<T>(H(j, j), H(j+1, j));
40             rotations [j].apply(H(j, j), H(j+1, j));
41             rotations [j].apply(s[i], s[i+1]);
42
43             ++inner, ++outer, ++j;
44         } while (! Inner.finished(abs(s[j])));
45
46         // Form the approximate solution
47         tri_solve(tri_view<upper>() (H.sub_matrix(0, j, 0, j)), s);
48         mult(V.sub_matrix(0, x.size(), 0, j), s, x, x);
49
50         // Restart
51         mult(A, scaled(x, -1.0), b, w);
52         solve(M, w, r);
53         beta = abs(two_norm(r));
54     }
55     return outer.error_code();
56 }

```

the performance of dense matrix-matrix product for MTL, Fortran BLAS, and the Sun Performance Library, all obtained on a Sun Ultra 30. The experiment shows that the MTL can compete with vendor-tuned libraries (on an algorithm that tends to

get extra attention due to benchmarking). We compiled the MTL executables using Kuck and Associates C++, in conjunction with v.5.0 of the Solaris C compiler. We compiled the Fortran BLAS (obtained from Netlib) with v.5.0 of the Solaris Fortran 77 com-

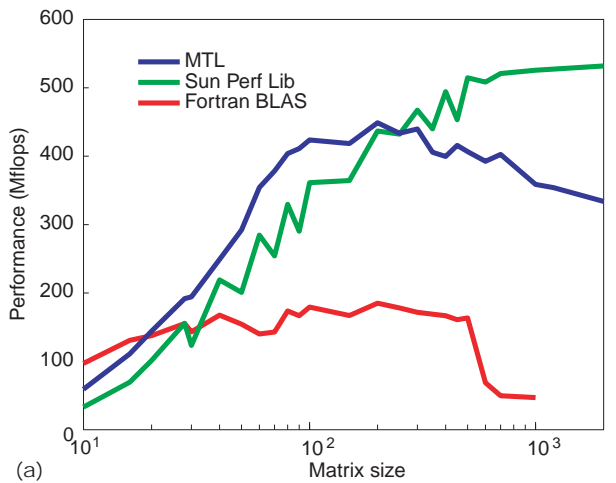
Figure 1. The Iterative Template Library (ITL) implementation of the preconditioned GMRES(m) algorithm. This algorithm computes an approximate solution to $Ax = b$ preconditioned with M . The restart value is specified by the parameter m .

piler. We used all possible compiler-optimization flags in all cases and cleared the cache between each trial. To demonstrate portability across different architectures and compilers, Figure 2b compares the performance of MTL with the Engineering and Scientific Subroutine Library (ESSL) on an IBM RS/6000 590. In this case, we compiled the MTL executable with the KCC and IBM xlc compilers.

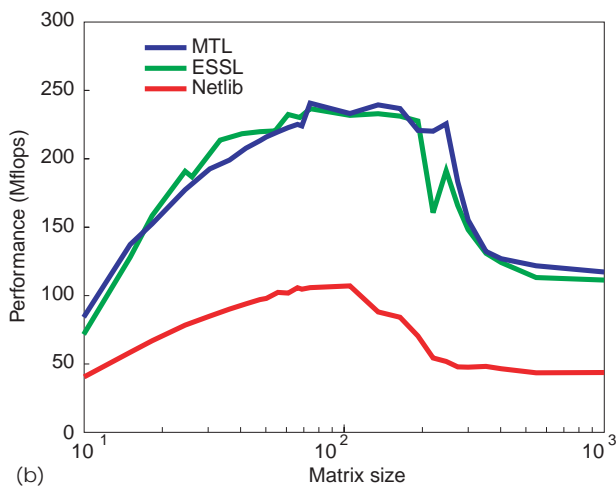
Dense and sparse matrix-vector multiplication. Figure 3 shows performance results obtained using the matrix-vector multiplication algorithm for dense and for sparse matrices, and compares the performance to that obtained with nongeneric libraries. Figure 3a compares MTL's dense matrix-vector performance to the Netlib BLAS (Fortran) and the Sun Performance Library. Figure 3b compares MTL's sparse matrix-vector performance to Sparskit⁹ (Fortran), and the NIST Sparse BLAS (C).¹⁰ We ran the experiments on a Sun Ultra 30, used sparse matrices from the MatrixMarket collection, and did not clear the cache between each matrix-vector timing trial. This experiment focused on the algorithm's pipeline behavior. If we had cleared the cache, the bottleneck would have become memory bandwidth, and we could not have seen differences in pipeline behavior. Blocking for cache is not as important for matrix-vector multiplication because there is no reuse of matrix data.

The future of MTL

Although MTL's core functionality is complete, in many ways our work has only begun. Using MTL as a foundation, we plan to develop several libraries with higher levels of function-



(a)

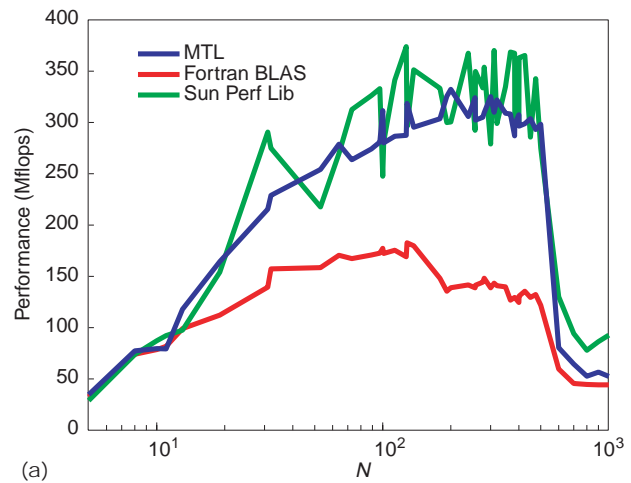


(b)

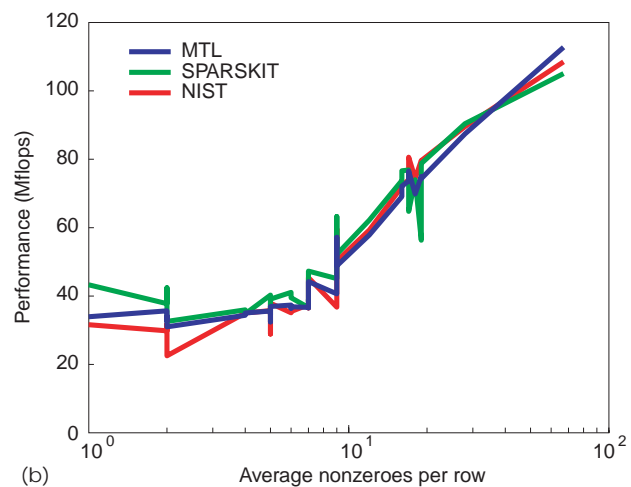
Figure 2. Performance comparison of the MTL dense matrix-matrix product with other libraries on (a) Sun Ultra 30 and (b) IBM RS6000.

ality (similar to how Lapack¹¹ uses the BLAS). As mentioned earlier, we have already developed the first such library, a collection of iterative solvers called the Iterative Template Library. Current work focuses on sparse and dense linear solvers, eigenproblem routines, and SVD computations. In the meantime, we provide wrappers to give users a convenient MTL-style interface to Lapack. The growing MTL user group is actively building on top of MTL and contributing algorithms.

We are often asked about using overloaded operators for MTL. Although an operator-based syntax can be very handy for rapid prototyping, it is in some sense “syntactic sugar” that we felt to be orthogonal to MTL’s original goals. We don’t feel the present MTL syntax to be a significant drawback in terms of its original goals—to apply generic programming to the domain of numerical linear algebra. Similarly, in terms of software-engineering practice (for



(a)



(b)


Figure 3. Performance of the MTL matrix-vector product applied to (a) column-oriented dense and (b) row-oriented sparse data structures compared with other libraries on Sun Ultra 30.

example, for library development) the existing syntax is perfectly suitable. Nevertheless, such a syntax can have value, so we are investigating an operator-based interface based on the expression template technology found in Blitz++¹² and PETE.¹³

Using an interpretive front end offers an alternative approach to rapid prototyping (and an operator-based syntax) with MTL. We have recently developed one such system (having a Matlab-like syntax) and are also investigating the use of other interpreted scripting languages (such as Python).

We continue to refine MTL and work on porting it to new compilers. Our (perhaps immodest) hope is that it will ultimately become suitable as a standard. We are currently working closely with the SGI STL team to better integrate MTL with the STL.

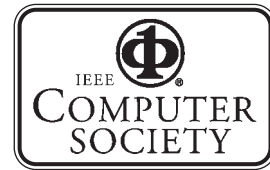
Beyond MTL, we are also investigating the application of

generic programming techniques to other problem domains such as graph algorithms, image and video processing, and parallel processing. We have made considerable progress in the former effort, which produced the Generic Graph Component Library.¹⁴ We report on GGCL elsewhere, but remark that it offers the same benefits as MTL—abstraction with high levels of performance. A GGCL implementation of minimum degree sparse matrix ordering matched the performance of the very best Fortran routines.¹⁵ 

References

1. A. Stepanov and M. Lee, *The Standard Template Library*, Tech. Report HPL-95-11, HP Laboratories, Menlo Park, Calif., 1995.
2. J.G. Siek and A. Lumsdaine, "The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra," *Proc. Int'l Symp. Computing in Object-Oriented Parallel Environments*, Springer Lecture Notes in Computer Science, pp. 59-70
3. K. Czarnecki, *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*, PhD thesis, Technische Universität, Ilmenau, Germany, 1998.
4. J. Dongarra et al., "Algorithm 656: An Extended Set of Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software*, Vol. 14, No. 1, 1998, pp. 18-32.
5. J. Dongarra et al., "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software*, Vol. 16, No. 1, 1990, pp. 1-17.
6. C. Lawson et al., "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Mathematical Software*, Vol. 5, No. 3, 1979, pp. 308-323.
7. Y. Saad and M. Shultz, "GMRES: A Generalized Minimum Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM J. Scientific Statistical Computing*, Vol. 7, No. 3, July 1986, pp. 856-869.
8. P.N. Brown and A.C. Hindmarsh, "Matrix-Free Methods for Stiff Systems of ODEs," *SIAM J. Numerical Analysis*, Vol. 23, No. 3, June 1986, pp. 610-638.
9. Y. Saad, *SPARSKIT: A Basic Tool Kit for Sparse Matrix Computations*, tech. report, NASA Ames Research Center, Moffitt Field, Calif., 1990.
10. K.A. Remington and R. Pozo, *NIST Sparse BLAS User's Guide*, National Institute of Standards and Technology, Washington DC.
11. E. Anderson et al., "Lapack: A Portable Linear Algebra Package for High-Performance Computers," *Proc. Supercomputing '90*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1990, pp. 1-10.
12. T.L. Veldhuizen, "Arrays in Blitz++," *Proc. Second Int'l Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1998, pp. 223-230.
13. S. Haney et al., *Easy Expression Templates Using PETE*, the Portable Expression Template Engine, Tech. Report LA-UR-99-777, Advanced Computing Laboratory, LANL, Los Alamos, N.M., 1999.
14. L.-Q. Lee, J.G. Siek, and A. Lumsdaine, "The Generic Graph Component Library," *OOPSLA '99*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1999, to be published.
15. L.-Q. Lee, J. G. Siek, and A. Lumsdaine, "Generic Graph Component Library," *ISCOPE '99*, 1999, to be published.

PURPOSE THE IEEE Computer Society is the world's largest association of computing professionals, and is the leading provider of technical information in the field.



MEMBERSHIP Members receive the monthly magazine **COMPUTER**, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

EXECUTIVE COMMITTEE

- President:* LEONARD L. TRIPP
Boeing Commercial Airplane Group
P.O. Box 3707
M/S 19-RF
Seattle, WA 98124
- VP, Standards Activities:*
 STEVEN L. DIAMOND *
VP, Technical Activities:
 JAMES D. ISAAK *
Secretary:
 DEBORAH K. SCHERRER *
Treasurer:
 MICHEL ISRAEL *
IEEE Division V Director:
 MARIO R. BARBACCI †
IEEE Division VIII Director:
 BARRY W. JOHNSON †
Executive Director & Chief Executive Officer:
 T. MICHAEL ELLIOTT †
- President-Elect:*
 GUYLAINE M. POLLOCK *
Past President:
 DORIS L. CARVER *
VP, Press Activities:
 CARL K. CHANG †
VP, Educational Activities:
 JAMES H. CROSS II †
VP, Conferences and Tutorials:
 WILLIS K. KING (2ND VP) *
VP, Chapters Activities:
 FRANCIS C.M. LAU *
VP, Publications:
 BENJAMIN W. WAH (1ST VP) *

*voting member of the Board of Governors †nonvoting member of the Board of Governors

BOARD OF GOVERNORS

- Term Expiring 1999: *Steven L. Diamond, Richard A. Eckhouse, Gene F. Hoffnagle, Tadao Ichikawa, James D. Isaak, Karl Reed, Deborah K. Scherrer*
- Term Expiring 2000: *Fiorenza C. Albert-Howard, Paul L. Borrell, Carl K. Chang, Deborah M. Cooper, James H. Cross, II, Ming T. Liu, Christina M. Schober*
- Term Expiring 2001: *Kenneth R. Anderson, Wolfgang K. Giloi, Haruhisa Ichikawa, Lowell G. Johnson, David G. McKendry, Anneliese von Mayrhauser, Thomas W. Williams*
- Next Board Meeting: *15 November 1999, Portland, Ore.*

COMPUTER SOCIETY OFFICES

- | | |
|---|---|
| Headquarters Office
1730 Massachusetts Ave. NW,
Washington, DC 20036-1992
Phone: (202) 371-0101
Fax: (202) 728-9614
E-mail: hq.ofc@computer.org | European Office
13, Ave. de L'Aquilon
B-1200 Brussels, Belgium
Phone: 32 (2) 770-21-98
Fax: 32 (2) 770-85-05
E-mail: euro.ofc@computer.org |
| Publications Office
10662 Los Vaqueros Cir.,
PO Box 3014
Los Alamitos, CA 90720-1314
General Information:
Phone: (714) 821-8380
membership@computer.org
Membership and
Publication Orders: (800) 272-6657
Fax: (714) 821-4641
E-mail: cs.books@computer.org | Asia/Pacific Office
Watanabe Building
1-4-2 Minami-Aoyama,
Minato-ku, Tokyo 107-0062,
Japan
Phone: 81 (3) 3408-3118
Fax: 81 (3) 3408-3553
E-mail: tokyo.ofc@computer.org |

EXECUTIVE STAFF


- | | |
|---|---|
| <i>Executive Director & Chief Executive Officer:</i>
T. MICHAEL ELLIOTT

<i>Director, Volunteer Services:</i>
ANNE MARIE KELLY

<i>Chief Financial Officer:</i>
VIOLET S. DOAN | <i>Chief Information Officer:</i>
ROBERT G. CARE

<i>Manager, Research & Planning:</i>
JOHN C. KEATON |
|---|---|

IEEE OFFICERS

- 
- President:* KENNETH R. LAKER
President-Elect: BRUCE A. EISENSTEIN
Executive Director: DANIEL J. SENESE
Secretary: MAURICE PAPO
Treasurer: DAVID A. CONNOR
VP, Educational Activities: ARTHUR W. WINSTON
VP, Publications Activities: LLOYD A. "PETE" MORLEY
VP, Regional Activities: DANIEL R. BENIGNI
VP, Standards Association: DONALD C. LOUGHRY
VP, Technical Activities: MICHAEL S. ADLER
President, IEEE-USA: PAUL J. KOSTEK