# Generic Programming for High Performance Numerical Linear Algebra *

Jeremy G. Siek† Andrew Lumsdaine† Lie-Quan Lee†

**Abstract**

We present a generic programming methodology for expressing data structures and algorithms for high-performance numerical linear algebra. As with the Standard Template Library [14], our approach explicitly separates algorithms from data structures, allowing a single set of numerical routines to operate with a wide variety of matrix types, including sparse, dense, and banded. Through the use of C++ template programming, in conjunction with modern optimizing compilers, this generality does not come at the expense of performance. In fact, writing portable high-performance codes is actually enabled through the use of generic programming because performance critical code sections can be concentrated into a small number of basic kernels. Two libraries based on our approach are described. The Matrix Template Library (MTL) is a high-performance library providing comprehensive linear algebra functionality. The Iterative Template Library, based on MTL, extends the generic programming approach to iterative solvers and preconditioners.

## 1 Introduction

The traditional approach to developing numerical linear algebra libraries is a combinatorial affair. Individual subroutines must be written to support every desired combination of algorithm, basic numerical type, and matrix storage format. For a library to provide a rich set of functions and data types, one might need to code literally hundreds of versions of the same routine. As an example, to provide basic functionality for selected sparse matrix types, the NIST implementation of the Sparse BLAS [18] contains over 10,000 routines and a code generation system.

This combinatorial explosion in implementation effort arises because, with most programming languages, algorithms and data structures are more tightly coupled than is conceptually necessary. That is, one cannot express an algorithm as a subroutine independently from the type of data that is being operated on. Thus, although abstractly one might have only a single algorithm to be expressed, it must be realized separately for every data type that is to be supported. As a result, providing a comprehensive linear algebra library — much less one that also offers high-performance — would seem to be an overwhelming, if not impossible, task.

Fortunately, certain modern programming languages, such as Ada and C++, provide support for *generic programming*, a technique whereby an algorithm can be expressed independently of the data structure to which it is being applied. One of the most celebrated examples of generic programming is the C++ Standard Template Library (STL) [14].

In this paper we apply the fundamental generic programming approaches pioneered by STL to the domain of numerical linear algebra. The resulting library, which we call

*This work was supported by NSF grants ASC94-22380 and CCR95-02710.

†Laboratory for Scientific Computing, University of Notre Dame, Notre Dame, IN 46556, {jsiek,lums,llee1}@lsc.nd.edu.

the *Matrix Template Library* (MTL) provides comprehensive functionality with a small number of of fundamental algorithms, while at the same time achieving high performance. High performance generic algorithms are a new development made possible by the powerful template and object-oriented features of the C++ language and by advances in C and C++ compiler technology. The MTL harnesses these advances to achieve performance on par with vendor-tuned libraries.

The MTL provides a solid and easy to use foundation for constructing numeric codes. We demonstrate this with the construction of the Iterative Template Library (ITL). The ITL is a collection of sophisticated iterative solvers. The major components necessary for iterative solvers — preconditioner, linear transform (typically a matrix-vector product), and convergence tests — have been generalized in the ITL method interface to allow for maximum flexibility. The ITL provides many commonly used preconditioners and convergence test and uses MTL for its basic linear algebra operations.

## 2 Generic Algorithms for Linear Algebra

The principal idea behind generic programming is that many algorithms can be abstracted away from the particular data structures on which they operate. Algorithms typically need the abstract functionality of being able to *traverse* through a data structure and *access* its elements. If data structures provide a standard interface for traversal and access, generic algorithms can be mixed and matched with data structures (called *containers* in STL). This interface is realized through the *iterator* (sometimes called a generalized pointer).

The same approach can be applied to linear algebra. Abstractly, linear algebra operations consist of traversing through vectors and matrices and accessing their elements. Vector operations fit neatly into the generic programming approach (in fact, the STL already defines several generic algorithms for vectors, such as `inner_product()`). Providing generic algorithms to encompass Level-1 BLAS functionality [13] is relatively straightforward.

```
template <class Matrix,class IterX,class IterY>
void matvec::mult(Matrix A, IterX x, IterY y) {
  typename Matrix::row_2Diterator i;
  typename Matrix::RowVector::iterator j;
  for (i = A.begin_rows(); not_at(i, A.end_rows()); ++i) {
    typename Matrix::PR tmp = y[i.index()];
    for(j = i->begin(); not_at(j,i->end()); ++j)
      tmp += *j * x[j.index()];
    y[i.index()] = tmp;
  }
}
```

FIG. 1. *Simplified example of a generic matrix-vector product.*

Matrix operations are slightly more complex, since the elements are arranged in a 2-dimensional format. The MTL algorithms process matrices *as if* they were containers of containers (note that the matrices are not necessarily implemented this way). The matrix algorithms are coded in terms of *iterators* and *two-dimensional iterators*. For example, a `row_2Diterator` can traverse the rows of a matrix, and produces a row vector when dereferenced. The iterator for the row vector can then be used to access the individual matrix elements. Fig. 1 shows how one can write a generic matrix-vector product.

| **Function Name** | **Operation** | **Function Name** | **Operation** |
|---|---|---|---|
| Vector Algorithms | | Vector Vector | |
| `set(x,alpha)` | $x_i \leftarrow \alpha$ | `copy(x,y)` | $y \leftarrow x$ |
| `scale(x,alpha)` | $x \leftarrow \alpha x$ | `swap(x,y)` | $y \leftrightarrow x$ |
| `s = sum(x)` | $s \leftarrow \sum_i x_i$ | `ele_mult(x,y,z)` | $z \leftarrow y \otimes x$ |
| `s = one_norm(x)` | $s \leftarrow \sum_i \lvert x_i \rvert$ | `ele_div(x,y,z)` | $z \leftarrow y \oslash x$ |
| `s = two_norm(x)` | $s \leftarrow (\sum_i x_i^2)^{\frac{1}{2}}$ | `add(x,y)` | $y \leftarrow x + y$ |
| `s = inf_norm(x)` | $s \leftarrow \max \lvert x_i \rvert$ | `s = dot(x,y)` | $s \leftarrow x^T \cdot y$ |
| `i = find_max_abs(x)` | $i \leftarrow$ index of $\max \lvert x_i \rvert$ | `s = dot_conj(x,y)` | $s \leftarrow x^T \cdot \bar{y}$ |
| `s = max(x)` | $s \leftarrow \max(x_i)$ | | |
| `s = min(x)` | $s \leftarrow \min(x_i)$ | | |
| Matrix Algorithms | | Matrix Vector | |
| `set(A, alpha)` | $A \leftarrow \alpha$ | `mult(A,x,y)` | $y \leftarrow A \times x$ |
| `scale(A,alpha)` | $A \leftarrow \alpha A$ | `mult(A,x,y,z)` | $z \leftarrow A \times x + y$ |
| `set_diag(A,alpha)` | $A_{ii} \leftarrow \alpha$ | `tri_solve(T,x,y)` | $y \leftarrow T^{-1} \times x$ |
| `s = one_norm(A)` | $s \leftarrow max_i(\sum_j \lvert a_{ij} \rvert)$ | `rank_one(x,A)` | $A \leftarrow x \times y^T + A$ |
| `s = inf_norm(A)` | $s \leftarrow max_j(\sum_i \lvert a_{ij} \rvert)$ | `rank_two(x,y,A)` | $A \leftarrow x \times y^T +$ $y \times x^T + A$ |
| `transpose(A)` | $A \leftarrow A^T$ | | |
| Matrix Matrix | | | |
| `copy(A,B)` | $B \leftarrow A$ | `swap(A,B)` | $B \leftrightarrow A$ |
| `add(A,C)` | $C \leftarrow A + C$ | `scale(A,alpha)` | $C \leftarrow \alpha A$ |
| `transpose(A,B)` | $B \leftarrow A^T$ | `ele_mult(A,B,C)` | $C \leftarrow B \otimes A$ |
| `mult(A,B,C)` | $C \leftarrow A \times B$ | `mult(A,B,C,E)` | $E \leftarrow A \times B + C$ |
| `tri_solve(T,B,C)` | $C \leftarrow T^{-1} \times B$ | | |

TABLE 1
*MTL linear algebra operations.*

## 3 MTL Algorithms

Table 1 lists the principal algorithms provided by MTL. This list seems sparse, but a large number of functions are available by combining the above algorithms with the `strided()`, `scaled()`, and `trans()` iterator adapters. Fig. 2 shows how the matrix-vector multiply algorithm (generically written to compute $y \leftarrow A \times x$) can be used to compute $y \leftarrow A^T \times \alpha x$ with the use of iterator adapters to transpose `A` and to scale `x` by `alpha`. Note that the adapters cause the appropriate changes to occur within the algorithm — they are not evaluated before the call to `matvec::mult()` (which would hurt performance). This adapter technique drastically reduces the amount of code that must be written for each algorithm.

```
//   y <- A' * alpha x
matvec::mult(trans(A), scale(x, alpha), y);
```

FIG. 2. *Example use of the transpose and scaled iterator adapters.*

The Matrix Template Library is unique in that, for the most part, each algorithm is implemented with just one template function. That is, a single algorithm is used whether the matrix is sparse, dense, banded, single precision, double, complex, etc. From a software maintenance standpoint, the reuse of code gives MTL a significant advantage over the

BLAS [6, 7, 13] or even other object-oriented libraries like TNT [16] (which has different algorithms for different matrix formats).

Because of the code reuse provided by generic programming, MTL has an order of magnitude fewer lines of code than the Netlib Fortran BLAS [20] — while providing much greater functionality and achieving significantly better performance. The MTL has 8,284 lines of code for its algorithms and 6,900 lines of code for its dense containers, while the Fortran BLAS total 154,495 lines of code. High performance versions of the BLAS (with which MTL is competitive) are even more verbose.

## 4 MTL Components

The Matrix Template Library defines a set of data structures and other components for representing linear algebra objects. An MTL matrix is constructed with layers of components. Each layer is a collection of classes that are templated on the lower layer. The bottom most layer consists of the numerical types (`float`, `double`, etc). The next layers consist of 1-D containers followed by 2-D containers. The 2-D containers are wrapped up with an *orientation*, which in turn is wrapped with a *shape*. Fig. 3 shows the general form of the layering, as well as a couple concrete examples. Note that some 2-D containers also subsume the 1-D type, such as the contiguous `dense2D` container.

```
// general form
shape < orientation < twod < oned < num_type > > > >
// triangular matrix
triangle< column< array2D< dense1D< double > > >, upper, packed>
// dense contiguous matrix, extended precision
row< dense2D< doubledouble > >
```

FIG. 3. *MTL component layering examples.*

**Matrix Orientation** The `row` and `column` adapters map the *major* and *minor* aspects of a matrix to the corresponding *row* or *column*. This technique allows the same code for data structures to provide both row and column orientations of the matrix. 2-D containers must be wrapped up with one of these adapters to be used in the MTL algorithms.

**Matrix Shape** Matrices can be categorized into several shapes: general, upper triangular, lower triangular, symmetric, Hermitian, etc. The traditional approach to handling the algorithmic differences due to shape is to have a separate function for each type. For instance, in the BLAS we have a `_GEMV`, `_SYMV`, `_TRMV`, etc. The MTL instead uses different data structures for each shape, with the `banded`, `triangle`, `symmetric`, and `hermitian` matrix adapters. It is the responsibility of these adapters to make sure that they work with all of the MTL generic algorithms. The MTL philosophy is to use *smarter* data structures to allow for fewer and simpler algorithms.

## 5 High Performance

We have presented many levels of abstraction, and a comprehensive set algorithms for a variety of matrices, but this elegance matters little if high performance can not be achieved. Generic programming coupled with modern compilers provide several mechanisms for high performance.

**Static Polymorphism** The template facilities in C++ allow functions to be selected at compile-time based on data type, as compared to virtual functions which allow for function selection at run-time (dynamic polymorphism). Static polymorphism provides a mechanism for abstraction which preserves high performance, since the template functions can be inlined just as regular functions. This ensures that the numerous small function calls in the MTL (such as iterator increment operators) introduce no extra overhead.

**Lightweight Object Optimization** The generic programming style introduces a large number of small objects into the code. This can incur a performance penalty because the presence of a structure can interfere with other optimizations, including the mapping of the individual data items to registers. This problem is solved with lightweight object optimization, also know as scalar replacement of aggregates [15].

**Automatic Unrolling and Instruction Scheduling** Modern compilers can do a great job of unrolling loops and scheduling instructions, but typically only for specific (recognizable) cases. There are many ways, especially in C and C++ to interfere with the optimization process. The MTL containers and algorithms are designed to result in code that is easy for the compiler to optimize. Furthermore, the *iterator* abstraction makes inter-compiler portability possible, since it encapsulates how looping is performed.

**Algorithmic Blocking** To obtain high performance on a modern microprocessor, an algorithm must properly exploit the associated memory hierarchy and pipeline architecture (typically through careful loop blocking and structuring). Ideally, one would like to express high performance algorithms in a portable fashion, but there is not enough expressiveness in languages such as C or Fortran to do so. Recent efforts (PHiPAC [2], ATLAS [17]) have resorted to going outside the language, i.e., to code generation systems, in order to gain this kind of flexibility. We have shown that with the use of meta-programming techniques in C++, it is possible to create flexible and elegant high-performance kernels that automate the generation of blocked and unrolled code. We present such a collection of kernels, the Basic Linear Algebra Instruction Set (BLAIS), in [19].

## 6 Performance Experiments

In the following figures we present some performance results comparing MTL with other available libraries (both public domain and vendor-supplied). Fig. 4 shows the dense matrix-matrix product performance for MTL, Fortran BLAS, the Sun Performance Library, TNT [16], and ATLAS [17], all obtained on a Sun UltraSPARC 170E. The MTL and TNT executables were compiled using Kuck and Associates C++ (KCC) [11], in conjunction with the Solaris C compiler. ATLAS was compiled with the Solaris C compiler and the Fortran BLAS (obtained from Netlib) were compiled with the Solaris Fortran 77 compiler. All possible compiler optimization flags were used in all cases.

To demonstrate portability across different architectures and compilers, Fig. 4 also compares the performance of MTL with ESSL [9] on an IBM RS/6000 590. In this case, the MTL executable was compiled with the KCC and IBM xlc compilers. To demonstrate genericity across different data structures and data types, Fig. 5 shows performance results obtained using the same generic matrix-vector multiplication algorithm for dense and for sparse matrices, and compares the performance to that obtained with non-generic libraries.

The presence (and absence) of different optimization techniques in the various programs can readily be seen in Fig. 4 (left) and manifest themselves quite strongly as a function of matrix size. In the region from $N = 10$ to $N = 256$, performance is dominated by register usage and pipeline performance. "Unroll and jam" techniques [5, 21] are used to attain high levels of performance in this region. In the region from 256 to approximately 1024, performance is dominated by data locality. Loop blocking for cache usage is used to attain high levels of performance here. Finally, for matrix sizes larger than approximately $N = 1024$, performance can be affected by conflict misses in the cache — notice how the results for ATLAS and Fortran BLAS fall precipitously at this point. To attain good performance in the face of conflict misses (in low associativity caches) block-copy techniques as described in [12] are used. Note that performance effects are cumulative. For instance, the Fortran BLAS do not use any of the techniques listed above for performance enhancement. As a result, performance is poor initially and continues to degrade as different effects come into play.

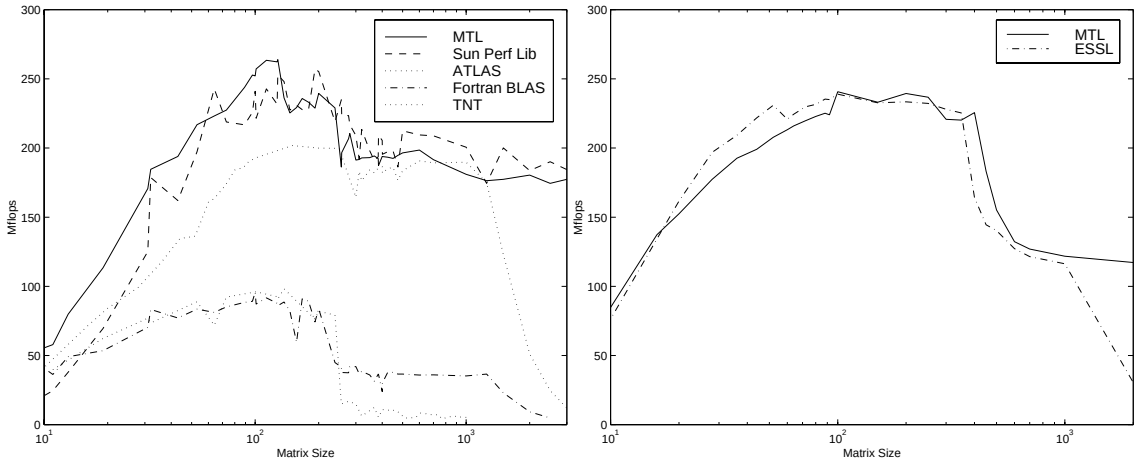


FIG. 4. *Performance comparison of generic dense matrix-matrix product with other libraries on Sun UltraSPARC (left) and IBM RS6000 (right).*

## 7 Iterative Template Library (ITL)

The Iterative Template Library is collection of sophisticated iterative methods written in C++ (similar to the IML++ Library [10]). It contains methods for solving both symmetric and nonsymmetric linear systems of equations, many of which are described in [1]. The ITL methods were constructed in a generic style, allowing for maximum flexibility and separation of concerns about matrix data structure, performance optimization, and algorithms.

Presently, ITL contains routines for conjugate gradient (CG), conjugate gradient squared (CGS), biconjugate gradient (BiCG), biconjugate gradient stabilized (BiCGStab), generalized minimal residual (GMRES), quasi-minimal residual (QMR) without look-ahead, transpose-free QMR, and Chebyshev and Richardson iterations. In addition, ITL provides the following preconditioners: SSOR, incomplete Cholesky, incomplete LU($n$), and incomplete LU with thresholding.

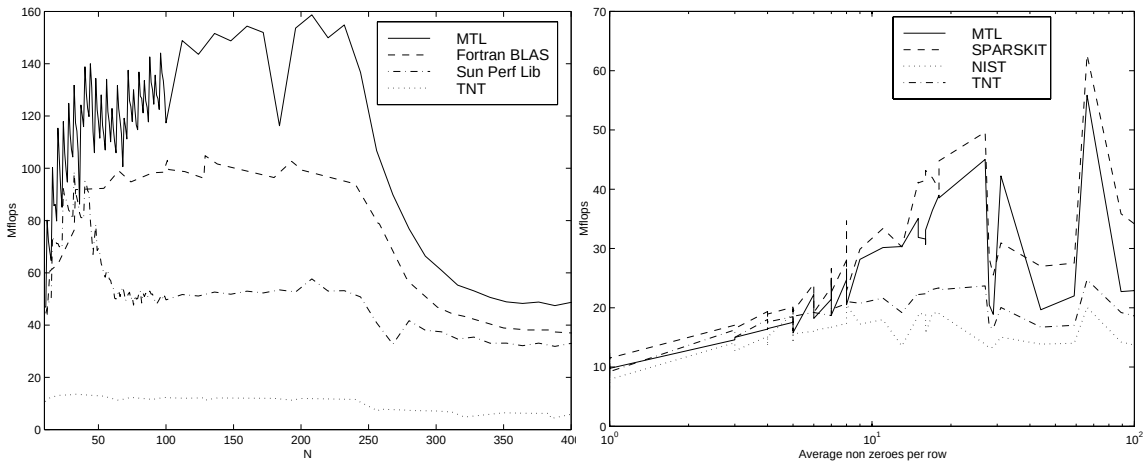


FIG. 5. *Performance of generic matrix-vector product applied to column-oriented dense (left) and row-oriented sparse (right) data structures compared with other libraries on Sun UltraSPARC.*

## 7.1 Generic Interface

The generic construction of the ITL revolves around the four major interface components.

**Vector** An array class with an STL-like iterator interface.

**Matrix** Either a MTL matrix, a *multiplier* (for matrix-free methods), or a custom matrix with a specialized `matvec::mult(A,x,y)` function defined.

**Preconditioner** An object with `solve(x,z)` and `trans_solve(x,z)` methods defined.

**Iteration** An object that defines the test for convergence, and the maximum number of iterations.

Fig. 6 shows the generic interface for the QMR iterative method. The function is templated for each interface component, which allows for the ability to mix and match concrete components. This particular method uses a split preconditioner, hence the `M1`, and `M2` arguments.

```
template <class Matrix, class Vector, class VectorB,
          class Precond1, class Precond2, class Iteration>
int qmr(const Matrix &A, Vector &x, const VectorB &b,
        const Precond1 &M1,const Precond2 &M2, Iteration& iter);
```

FIG. 6. *An ITL method interface example.*

Fig. 7 gives an example of how one might call the `qmr()` routine. The interface presented to the user of ITL has been made as simple as possible. This example uses the `compressed2D` matrix type from MTL and the SSOR preconditioner.

As listed above, there are certain requirements for each interface component, but there is a significant amount of flexibility in the concrete implementation of a particular component. For instance, any of the ITL supplied preconditioners (Cholesky, ILU, ILUT, SSOR) can be used with any method (though some are for symmetric matrices only), and custom

```
typedef row< compressed2D<double> >  matrix;
int max_iter = 50;
matrix A(5, 5);
dense1D<double> x(A.nrows(), 0.0);
dense1D<double> b(A.ncols(), 1.0);
// fill A ...
SSOR<matrix> precond(A);
basic_iteration<double> iter(b, max_iter, 1e-6);
qmr(A, x, b, precond.left(), precond.right(), iter);
```

FIG. 7. *Example use of the ITL QMR iterative method.*

preconditioners can be added to the list with little extra effort. Likewise, the control over the test for convergence has been encapsulated in the `Iteration` interface, so that variations in this regard can be made independent of the main algorithms.

Similarly, the `Matrix` and `Vector` interfaces allow for flexibility in matrix storage implementation, or even in how the matrix-vector multiplication is carried out. There are several levels of flexibility available. The ITL uses the MTL interface for its basic linear algebra operations. Since the MTL linear algebra operations are generic algorithms, a wide range of matrix types can be used including all of the dense, sparse, and banded types provided in the MTL. Additionally, the MTL generic algorithms can work with custom matrix types, assuming the matrix exports the required interface.

A second level of flexibility is available in that the user may specialize the `matvec::mult(A,x,y)` function for a custom matrix type, and bypass the MTL generic algorithms. An example use of this would be to perform matrix-free computations [3, 4]. Another possibility would be in using a distributed matrix with parallel versions of the linear algebra operations.

### 7.2 Ease of Implementation

The most significant benefit of layering the ITL on top of the MTL interface is the ease of implementation. The ITL algorithms can be expressed in a concise fashion, very close to the pseudo-code from a text book. Fig. 8 compares the preconditioned conjugate gradient algorithm from [1] with the code from the ITL.

The generic component construction of the ITL also aids in testing and verifiction of the software, since it enables an incremental approach. The basic linear algebra operations are tested thoroughly in the MTL test suite, and the preconditioners are tested individually before they are used in conjunction with the iterative method. In addition, there is the identity preconditioner, so that the iterative methods can be tested in isolation (without a real preconditioner).

The abstraction level over the linear algebra also makes performance optimization much easier. Since the ITL iterative methods do not enforce the use of a particular matrix type, or a particular matrix-vector multiply, optimizations at these levels can happen with no change to the iterative method code.

### 7.3 ITL Performance

Here we present a performance comparison with the IML++ [10], one of the few other comprehensize iterative method packages (which uses SparseLib++ [8] for the sparse basic

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**

```
while (! iter.finished(r)) {
  M.solve(r, z);
  rho = vecvec::dot_conj(r, z);
  if (iter.first())
    vecvec::copy(z, p);
  else {
    beta = rho / rho_1;
    vecvec::add(z, scaled(p, beta), p);
  }
  matvec::mult(A, p, q);
  alpha = rho / vecvec::dot_conj(p, q);
  vecvec::add(x, scaled(p, alpha), x);
  vecvec::add(r, scaled(q, -alpha), r);
  rho_1 = rho;
  ++iter;
}
```

FIG. 8. *Comparison of an algorithm for the preconditioned conjugate gradient method and the corresponding ITL algorithm.*

linear algebra). Six matrices from the Harwell Boeing collection are used. Computation time (in seconds) per iteration is plotted in Fig. 9 for each of the following methods: CGS, BICG, BICGSTAB, QMR and TFQMR (which only exists in ITL). The ILU (without fill-in) preconditioner was used in all of the experiments. All timings were run an a Sun UltraSPARC 30. The ITL methods were roughly twice as fast as the corresponding IML++ methods.

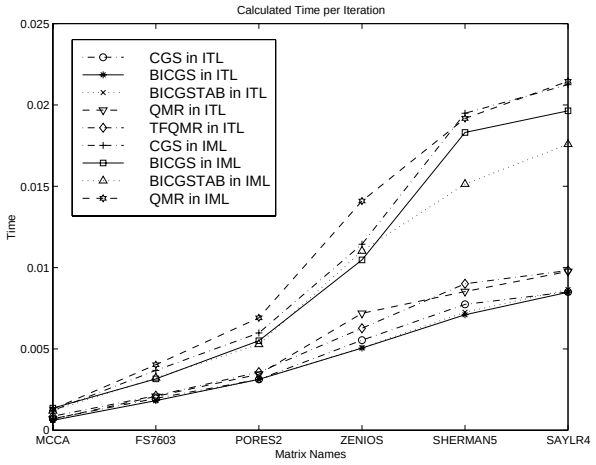


FIG. 9. *Comparison of ITL and IML++ performance over six matrices.*

## 8 MTL Availability

The Matrix Template Library and Iterative Template Library can be downloaded from the MTL web page at `http://www.lsc.nd.edu/research/mtl/`

## Acknowledgments

This work was supported by NSF grants ASC94-22380 and CCR95-02710. The authors would like to express their appreciation to Tony Skjellum and Puri Bangalore for numerous helpful discussions. Jeff Squyres and Kinis Meyer also deserve thanks for their help on parts of the MTL.

## References

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

[2] J. Bilmes, K. Asanovic, J. Demmel, D. Lam, and C.-W. Chin. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. Technical Report CS-96-326, University of Tennessee, May 1996. Also available as LAPACK working note 111.

[3] Peter N. Brown and Alan C. Hindmarsh. Matrix-free methods for stiff systems of ODE's. *SIAM J. Numer. Anal.*, 23(3):610–638, June 1986.

[4] P.N. Brown and Y. Saad. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Statist. Comput.*, 11(3):450–481, May 1990.

[5] S. Carr and Y. Guan. Unroll-and-jam using uniformly generated sets. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 349–357, Los Alamitos, December1–3 1997. IEEE Computer Society.

[6] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended set of basic linear algebra subprograms: Model implementations and test programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.

[8] J. J. Dongarra, A. Lumsdaine, R. Pozo, and K. A. Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 214–218, 1994.

[9] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference*, 2 edition, 1992.

[10] Roldan Pozo Jack Dongarra, Andrew Lumsdaine and Karin A. Remington. *Iterative Methods Library Reference Guide*, v. 1.2 edition, April 1997.

[11] Kuck and Associates. *Kuck and Associates C++ User's Guide*.

[12] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS IV*, April 1991.

[13] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[14] Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.

[15] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[16] Roldan Pozo. *Template Numerical Toolkit (TNT) for Linear Algebra*. National Insitute of Standards and Technology.

[17] Jack J. Dongarra R. Clint Whaley. Automatically tuned linear algebra software (ATLAS). Technical report, University of Tennessee and Oak Ridge National Laboratory, 1997.

[18] Karen A. Remington and Roldan Pozo. *NIST Sparse BLAS User's Guide*. National Institute of Standards and Technology.

[19] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[20] University of Tennessee, Knoxville and Oak Ridge National Laboratory. *Netlib Software Repository*. http://www.netlib.org.

[21] Michael E. Wolf and Monica S. Lam. A data locality optimising algorithm. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, ON, Canada, June 1991. ACM Press.