The Matrix Template Library:
A Unifying Framework for
Numerical Linear Algebra⋆

Jeremy G. Siek   Andrew Lumsdaine

Laboratory for Scientific Computing
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556
{jsiek,lums}@lsc.nd.edu
WWW home page: http://www.lsc.nd.edu/

Abstract. We present a unified approach for expressing high performance numerical linear algebra routines for a class of dense and sparse matrix formats and shapes. As with the Standard Template Library [7], we explicitly separate algorithms from data structures through the use of generic programming techniques. We conclude that such an approach does not hinder high performance. On the contrary, writing portable high performance codes is actually enabled with such an approach because the performance critical code sections can be isolated from the algorithms and the data structures.

1 Introduction

The traditional approach to writing basic linear algebra routines is a combinatorial affair. There are typically four precision types that need to be handled (single and double precision real, single and double precision complex), several dense storage types (general, banded, packed), a multitude of sparse storage types (the Sparse BLAS Standard Proposal includes 13 [1]), as well as row and column orientations for each matrix type. On top of that, if one wants to parallelize these codes, there are several data distributions to be supported. To provide a full implementation one might need to code literally hundreds of versions of the same routine! It is no wonder the NIST implementation of the Sparse BLAS [11] contains over 10,000 routines and an automatic code generation system.

This combinatorial explosion 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 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) [7].

In this paper we apply the fundamental generic programming approaches used by STL to the domain of numerical linear algebra. The resulting library, which we call 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 Matrix Template Library is in its second generation. The first version [9] focused on abstractions at the vector level. The current version of MTL has been completely rewritten using generic programming techniques, focusing on the use of iterators for a much higher degree of genericity and code reuse.

2   Generic Algorithms: The Standard Template Library

The principal idea behind the STL is that many algorithms can be abstracted away from the particular data structures on which they operate. Algorithms typically need the abstract functionality of *traversing* through a data structure and *accessing* its elements. If data structures provide a standard interface for traversal and access, generic algorithms can be freely mixed and matched with data structures (called *containers* in STL).

The main facilitator in the separation of algorithms and containers in STL is the *iterator* (sometimes called a "generalized pointer"). Iterators provide a mechanism for traversing containers and accessing their elements. The interface between an algorithm and a container is specified by the types of iterators exported by the container. Generic algorithms are written solely in terms of iterators and never rely upon specifics of a particular container. Iterators are classified into broad categories, some of which are: `InputIterator`, `ForwardIterator`, and `RandomAccessIterator`.

3   Generic Algorithms for Linear Algebra

Linear algebra routines operate on vectors and matrices. Vectors fit well into the STL container framework (there is in fact a `vector` container defined within STL). There are also many generic algorithms already defined in STL that provide linear algebra functionality, such as `inner_product()`. Extending these generic algorithms to encompass the rest of the common Level-1 BLAS [6] is a trivial matter.

There is no analogue of a matrix container in the STL, but we can construct 2-dimensional containers from the basic STL containers. For instance, one can create a dense matrix with `vector<vector<double>>`. The 2-dimensional containers can then be used in generic algorithms for Level-2 and Level-3 BLAS [3, 4].

The Level-2 and Level-3 generic algorithms are coded in terms of *iterators* and *two-dimensional iterators*. A 2-D iterator has the property that its `reference` type is a *container*. In other words, dereferencing a 2-D iterator produces a 1-D container. In this way, a `Row2DIter` can traverse the rows of a row-oriented matrix, and then can be dereferenced to perform operations on each row. The example in Fig. 1 shows how one can write a generic matrix-vector product.

```
template <class Row2DIter, class IterX, class IterY> void
matvec::mult(Row2DIter i, Row2DIter end_rows, IterX x, IterY y) {
  typename Row2DIter::value_type::const_iterator j;
  while (not_at(i, end_rows)) {
    j = (*i).begin();
    typename IterY::value_type tmp(0);
    while (not_at(j, (*i).end())) {
      tmp += *j * x[j.index()];
      ++j;
    }
    y[i.index()] = tmp;
    ++i;
  }
}
```

Fig. 1. Simplified example of a generic matrix-vector product.

What makes this algorithm generic is that it does not refer to the details of the matrix implementation. The only access into the matrix is through the iterators. Each matrix type has its own set of iterators that present a common interface to algorithms, but each has a very different implementation. For instance, this algorithm works with both dense and sparse, triangular and symmetric matrices. The differences are taken care of at the iterator level, which makes for a cleaner algorithm. The and end points of each row traversal are encapsulated in the iterator abstraction, as is the incrementing. The `not_at()` generic method hides the differences in how the end comparison may need to happen (the best comparison operator choice depends on the compiler and architecture). The `index()` method provides a uniform way to access vector x, whether the algorithms is traversing a dense row, or skipping through sparse elements.

4 The MTL Generic Algorithms for Linear Algebra

The Matrix Template Library provides a rich set of basic linear algebra operations, roughly equivalent to the Level-1, Level-2 and Level-3 BLAS. Table 1 lists the principle algorithms included in MTL. In the table, `alpha` and `s` are scalars, `x,y,z` are 1-D containers, `A,B,C,E` are `row` or `column` oriented matrices, `U, L` are upper and lower triangular matrices, and `i` is an iterator.

The unique feature of the Matrix Template Library is that, for the most part, each of the algorithms is implemented with just one template function. A single algorithm is used whether the matrix is sparse, dense, banded, is single precision, double, complex, etc.. From a software maintenance standpoint, this reuse of code gives MTL a huge advantage over the BLAS or even other object-oriented libraries like TNT [10] (which has different algorithms for different matrix formats).

The generic algorithm code reuse results in the MTL having a 6.5 times fewer lines of code than the netlib Fortran BLAS. The MTL has 8,399 lines of code for the algorithms and 15,411 lines of code for components, for a total of 23,810 lines of code. The Fortran BLAS total 154,495 lines of code, and do not handle sparse matrices, or provide as orthogonal a set of operations as the MTL.

Our implementation of these generic algorithms is built on top of two libraries. The first is a specifically modified version of STL and the second is BLAIS (both libraries are described in [13]). The use of these libraries allows the generic algorithms to be expressed in an elegant way, with very few lines of code and no loss in performance. More importantly, the BLAIS library allows the blocked MTL algorithms to be portable. With BLAIS, the blocking sizes can be modified at compile time through a few global constants, so that the algorithms can be customized for the memory hierarchy of a particular architecture.

Note that in Table 1 different operations are not defined for each permutation of transpose, scaling, and striding. Instead, only one algorithm is provided, but it can be combined with the use of `strided` and `scaled` vector adapters, or the `trans()` method to create the permutations. Fig. 2 shows how this is done with a matrix-vector multiply.

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

Fig. 2. Transpose, Scaled, and Strided Adapters

5 MTL Components

The Matrix Template Library defines a set of data structures and other components for representing linear algebra objects. These components work in con-

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)`	$s \leftarrow \sum_i \mid x_i \mid$	`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)`	$s \leftarrow \max \mid x_i \mid$	`s = dot(x,y)`	$s \leftarrow x^T \cdot y$
`i = find_max_abs(x)`	$i \leftarrow$ index of max $\mid x_i \mid$	`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)`	$s \leftarrow max_i(\sum_j \mid a_{ij} \mid)$	`rank_one(x,A)`	$A \leftarrow x \times y^T + A$
`s = inf_norm(A)`	$s \leftarrow max_j(\sum_i \mid a_{ij} \mid)$	`rank_two(x,y,A)`	$A \leftarrow x \times y^T +$
`transpose(A)`	$A \leftarrow A^T$		$y \times x^T + A$
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.

junction with the MTL generic linear algebra routines. An MTL matrix is constructed with layers of components. Each layer is a collection of classes that are templated on the lower layer. At the bottom most layer are the basic numeric types (float, double, etc). The next layers consists of the 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*. A complete MTL matrix type consists of a templated expression in the form `shape < orientation < twod < oned < elt_type > > > >`

5.1   1-D Containers

The 1-dimensional containers in the MTL have two purposes. They are used as vector objects as well as the building blocks for 2-dimensional containers. The `scaled` and `strided` adapters reduce the complexity of the generic algorithms by a factor of 2 or more.

dense1D Similar to the STL `vector` class, modified to work with the MTL algorithms and to ensure high performance.

sparse1D adapter This adapter class allows STL containers to be used as a sparse vector. The elements of the container must be index-value pairs. The `vector`, `list`, and `set` all work well with the `sparse1D` adapter.

compressed sparse vector The traditional *array of values* and *array of indices*, but packaged with the proper iterators to work seemlessly with the generic algorithms.

scaled adapter This adapter causes the values of the vector to be scaled as they are accessed. This is done through the use of a `scale_iterator` adapter.

strided adapter The `strided` adapter causes the algorithms to stride through the 1-D container. Again this is implemented through the use of a `strided_iterator` adapter which redefines the `operator++()`.

5.2   2-D Containers

The Matrix Template Library defines a set of requirements for 2-D containers, and provides implementations of many of the common matrix formats conformant to this set of requirements. This allows the user to mix and match algorithms with 2-D containers. Note that both sparse and dense matrices are unified under this abstraction. The 2-D container requirements are formulated with respect to *major* and *minor* instead of row and column. This allows the flexibility of using the same 2-D container for both row and column oriented matrices.

array2D This containers allows 1-D containers to be composed to make a two dimensional container. Both sparse and dense matrix types can be constructed from `array2d`.

dense2D This container allocates a contiguous chunk of memory for the whole dense matrix.

sparse2D Vectors of index-value pairs of the sparse matrix are stored contiguously.

compressed2D Index and value arrays are stored contiguously. Essentially the traditional compressed row or column storage.

scaled2D adapter Similar to the `scaled1D` adapter.

5.3   Matrix Orientation

The `row` and `column` adapters map the *major* and *minor* aspects of a matrix to the corresponding *row* or *column*. 2-D containers must be wrapped up with one of these adapters to be used in the MTL algorithms. An excerpt from the `row` orientation class is listed in Fig. 3.

5.4   Matrix Shapes

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. Using a

```
template <class TwoD>
class row : public TwoD {
public:
  typedef typename TwoD::major_iterator row_2Diterator;
  typedef typename TwoD::minor_iterator column_2Diterator;
  typedef typename TwoD::MajorVector RowVector;
  typedef typename TwoD::MinorVector ColumnVector;

  row_2Diterator begin_rows() { return begin_major(); }
  row_2Diterator end_rows() { return end_major(); }
  column_2Diterator begin_columns() { return begin_minor(); }
  column_2Diterator end_columns() { return end_minor(); }

  typename RowVector::reference
  operator()(int row, int col) { return TwoD::operator()(row,col); }
};
```

Fig. 3. Exerpt of row orientation class.

generic approach, we can collapse these cases into a single algorithm, handling the differences at the iterator level.

For example, the difference between the general and the triangular matrix-vector product algorithm is only in the start and end points for each vector. Consider our example of a generic matrix-vector product in Fig. 1. The start and end points of the inner loop are obtained through the `begin()` and `end()` methods of the vector. This means the difference between a `_GEMV` and a `_TRMV` can be abstracted away with the proper vector types.

This is where the MTL shape adapters do their work. MTL provides the `triangle`, `banded`, and `symmetric` adapters which handle the differences in vector traversal. In addition, the shape classes take a template argument to denote whether the matrix is packed or unpacked (e.g. a symmetric matrix stored in packed form).

5.5   Iterators and Iterator Adapters

The STL iterators provide two of the necessary services to unify dense and sparse matrix algorithms, traversal and access. However, they do not provide a way to obtain the index corresponding to the iterator's position. For sparse matrices, the index is found in a structure or array. For dense, the index can be inferred. We unify dense and sparse algorithms with the use of iterator adapters that define the `int index()` method. This provides a common interface for accessing the row (or column) index corresponding to a matrix element. All of the MTL iterators and iterator adapters provide this method.

sparse iterators Iterators for compressed and pair vectors.
dense iterators Adds the `int index()` method to the iterator.

strided iterator adapter The `operator++` moves the iterator a stride.

scale iterator The dereference operation also scales the value.

block iterator Similar to `strided_iterator`, but dereferencing gives a `block`, instead of a single value.

6   Performance

We have presented many levels of abstraction, and a set of unified algorithms for a variety of matrices, but this matters little if high performance can not be achieved. The most exciting aspect of the Matrix Template Library is that we can provide a very high level of performance. Here we present the performance results for dense column-oriented matrix-vector product and sparse row-oriented matrix vector product. The compilers used were Kuck and Associates C++ [5], and the Sun Solaris C compiler with maximum available optimizations. The experiments were run on a Sun UltraSPARC 170E.

Fig. 4 shows the dense matrix-vector performance for MTL, Fortran BLAS, the Sun Performance Library, and TNT [10].
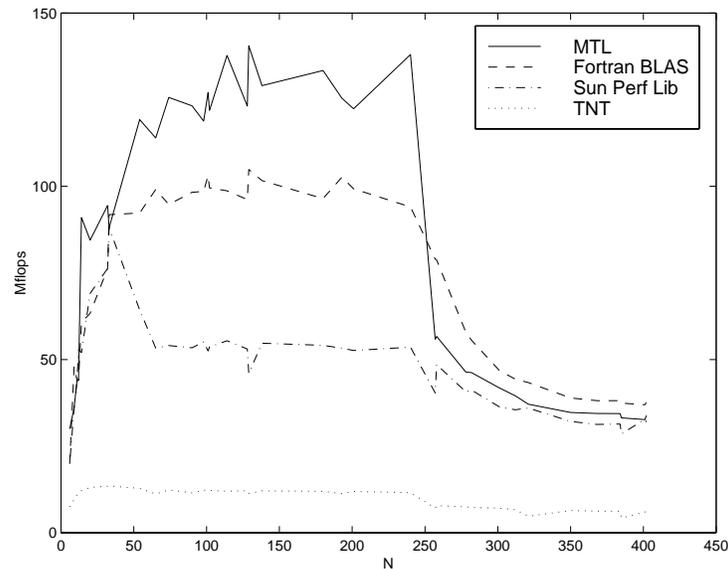

Fig. 4. Dense Matrix-Vector Multiply (Column Oriented)

Fig. 5 shows the sparse matrix-vector performance for MTL, SPARSKIT [12] (Fortran), NIST [11](C), and TNT (C++). The sparse matrices used are from the MatrixMarket collection.

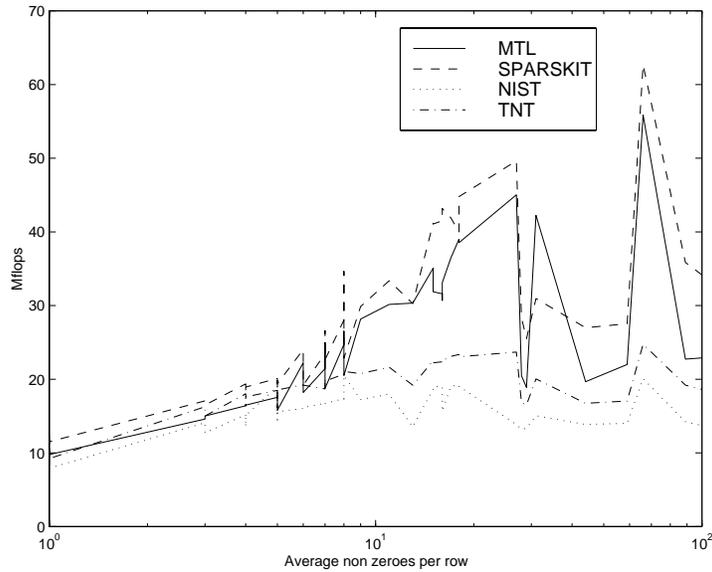Fig. 6 shows the dense matrix-matrix performance for MTL, Fortran BLAS, the Sun Performance Library, and TNT [10].

Fig. 5. Sparse Matrix-Vector Multiply (Row Oriented)

6.1 Analysis

Achieving high performance with a high level language involves making the compiler work as hard as possible. 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 (often inadvertently) interfere with the optimization process. The abstractions of the MTL are deliberately designed so that the resulting implementation enables aggressive compiler optimizations. One particular example is the `iterator` abstration which (among other things) encapsulates how looping is performed. Note that this encapsulation also has advantages from a software engineering point of view. For example, portability is also enhanced, since performance critical code is located in one place.

In early implementations of C++ compilers, the use of small objects such as iterators added significant overhead to execution time, and interfered with any attempts by the compiler to perform unrolling. Modern optimizing compilers such as Kuck and Associates C++ performs lightweight object optimization, inlining, and several other steps that remove this extra overhead.

7 Parallel MTL (PMTL)

Parallel extensions to MTL are currently being developed to extend the MTL framework to operations on distributed matrices and vectors. The code to handle the communication will be incorporated into container adapters and special
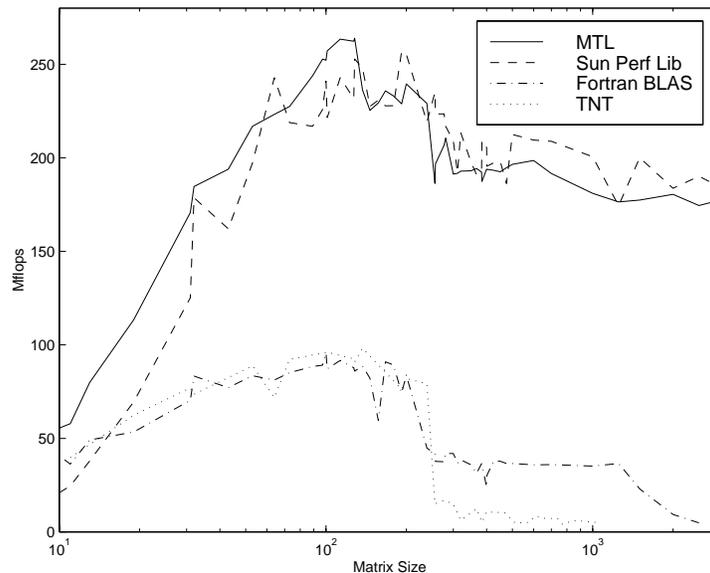
Fig. 6. Dense Matrix-Matrix Multiply

iterators. In this way the MTL algorithms can be converted from sequential to parallel through the wrapping of matrices with such adapters, and not with changes in the generic algorithms. Part of the container adapters will be a distribution object, which will define a flexible way to specify the distribution of the matrix over the network/parallel machine. PMTL will use MPI for implementing communication.

Current parallel linear algebra libraries, such as BlockSolve [8], ScaLAPACK [2], and PLAPACK [14] offer high perforance, but do not necessarily provide flexibility. This is especially problematic for sparse matrix codes, were the optimal matrix format is very problem dependent. The Bernoulli Compiler from Cornell [15] attempts to address this problem with a special purpose compiler that takes dense code and sparse matrix descriptions as input, and creates parallel code. With the Parallel Matrix Template Library we hope to demonstrate that the expressiveness of the C++ language makes the creation of such special purpose compilers unnecessary.

8   MTL Availability

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

Acknowledgments

References

1. Blas standard draft chapter 3: Sparse blas. Technical report, Basic Linear Algebra Subprograms Technical Forum, December 1997.
2. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, May 1997.
3. 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.
4. 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.
5. Kuck and Associates. *Kuck and Associates C++ User's Guide*.
6. 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.
7. Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.
8. Paul E. Plassmann Mark T. Jones. *BlockSolve95 Users Manual: Scalable Library for the Parallel Solution of Sparse Linear Systems*. Argonne National Laboratory, December 1995.
9. Brian C. McCandless and Andrew Lumsdaine. The role of abstraction in high-performance computing. In *Scientific Computing in Object-Oriented Parallel Environments*. ISCOPE, December 1997.
10. Roldan Pozo. *Template Numerical Toolkit (TNT) for Linear Algebra*. National Insitute of Standards and Technology.
11. Karen A. Remington and Roldan Pozo. *NIST Sparse BLAS User's Guide*. National Institute of Standards and Technology.
12. Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations. Technical report, NASA Ames Research Center, 1990.
13. Jeremy G. Siek and Andrew Lumsdaine. A rational approach to portable high performance: The basic linear algebra instruction set (blais) and the fixed algorithm size template (fast) library. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.
14. Robert A. van de Geijn. *Using PLAPACK*. MIT Press, April 1997.
15. Paul Stodghill Vladimir Kotlyar, Keshav Pingali. Compiling parallel code for sparse matrix applications. Technical report, Cornell University, 1997.