The Role of Abstraction in High-Performance Computing*

Brian C. McCandless and Andrew Lumsdaine

University of Notre Dame

Abstract. Although there is perpetual interest in using high-level languages such as C++ for high-performance computing, the conventional wisdom is that the very data abstractions that make these languages attractive from a software engineering perspective carry with them inherent performance penalties that make them unattractive from a performance perspective. The Matrix Template Library (MTL) is a C++ library specification that consists of a small number of composable template classes for defining sparse and dense matrix types as well as a comprehensive set of generic algorithms for numerical linear algebra. In this paper, we discuss our experiences with MTL and demonstrate that abstraction is not necessarily the enemy of performance and that, in fact, data abstraction can be an effective tool in enabling high performance.

1 Introduction

There is a common perception in scientific computing that abstraction is the enemy of performance. Although there is perpetual interest in using languages such as C or C++ and the powerful data abstractions that those languages provide, the conventional wisdom is that data abstractions inherently carry with them a (perhaps severe) performance penalty. Our thesis is that this is not necessarily the case and that, in fact, abstraction can be an effective tool in enabling high performance—but one must choose the right abstractions.

The misperception about abstraction springs from numerous examples of C++ libraries that provide a nice user interface through polymorphism, operator overloading and so forth, so that the user can implement an algorithm or a library in a "natural" way (SparseLib++ and IML++ [4], for example). Such an approach will (by design) hide computational costs from the user and degrade performance. One approach to providing performance and abstraction is through the use of lazy evaluation [1], but this approach can have other performance penalties as well as implementation difficulties.

One of the most important concerns in obtaining high performance on modern workstations is proper exploitation of the memory hierarchy. That is, a high-performance algorithm must be cognizant of the costs of memory accesses and must be structured to maximize use of registers and cache and to minimize

* This work was supported by NSF cooperative grant ASC94-22380.

cache misses and pipeline stalls. To properly complement high-performance algorithms, data abstractions should similarly account for hierarchical memory explicitly, and enable a programmer to readily exploit it.

The particular set of abstractions for bridging the performance-abstraction gap that we present here is the *Matrix Template Library* (MTL), written in C++ [8]. In the following sections, we describe the basic design of MTL and discuss our experiences in developing and using it. Experimental results are presented that show that MTL provides performance competitive with (or better than) traditional mathematical libraries.

We remark that this work is decidedly *not* an attempt to "prove" that a particular language (in our case, C++) offers higher performance than another language (Fortran, for example). Such arguments are, ultimately, pointless. Any language with a mature compiler can offer high performance (see PhiPAC [3]). Software development, even scientific software development, is about more than just performance and, except for academic situations, one must necessarily be concerned with the costs of software over its entire life-cycle. Thus, we contend that the only relevant discussion to have about languages is how particular languages enable the robust construction, maintenance, and evolution of complex software systems. In that light, modern high-level languages have a distinct advantage: most of them were designed specifically for the development of complex software systems, and the more widely-used ones have survived only because they are able to meet the needs of software developers.

2 The Matrix Template Library

MTL is by no means the first attempt to bring abstraction to scientific programming (see [2], for example), nor is it the first attempt at a mathematical library in C++ (see HPC++ [14], LAPACK++ [5], SparseLib++/IML++ [4], and the Template Numerical Toolkit [11]). MTL is unique, however, in its general underlying approach to separate algorithms from data structures, and in its particular commitment to self-contained high performance. Other libraries, if they are concerned about performance at all, attain high performance by making (mixed-language) calls to BLAS subroutines. The higher-level C++ code merely provides a syntactically pleasing means for gluing high-performance subroutines together, but does not provide flexible means for obtaining high performance (as MTL does).

The design of MTL is partly based on the idea that linear algebra objects (matrices and vectors) can be composed from simpler components. This can lead to good reuse of code, flexible storage formats, and easy-to-manage data distributions. The design is also based on the premise that storage format and algorithms can be efficiently separated and treated independently for a large class of algorithms. As a result, each algorithm needs to be implemented once and can automatically be called with matrices and vectors in any storage format. This drastically reduces the amount of code that needs to be written, debugged, and maintained.

3 Data Structures and Algorithms

One design goal of MTL is to separate algorithms from local data storage formats and from the distribution of matrices and vectors. This separation is a key ingredient in making the library extensible in both the number of linear algebra routines and the number of supported data storage formats.

The Matrix Template Library was inspired to a large extent by the Standard Template Library (STL) for C++ [7]. STL has become extremely popular because of its elegance, richness, and versatility. The original motivation for STL, however, was not to provide yet another library of standard components, but rather to introduce a new programming paradigm [9, 10].

This new paradigm was based on the observation that many algorithms can be abstracted away from the particular representations of the data structures upon which they operate. As long as the data structures provide a standard interface for algorithms to use, algorithms and data structures can be freely mixed and matched. Moreover, this paradigm accommodates this process of abstraction without sacrificing performance.

To realize an implementation of an algorithm which is independent of data structure representation requires language support. In particular, a language must allow algorithms (and data) to be parameterized not only by the values of the formal parameters (the arguments), but also by the *type of the data.* Few languages offer this capability, and it has only (relatively) lately become part of C++. In C++, functions and object classes are parameterized through the use of *templates* [13], hence the realization of a generic algorithm library in C++ as the Standard Template Library.

The real contribution of STL, then, was to realize that algorithms and data structures can be separated, to classify types of standard components, to define a set of interfaces for standard component to use, and lastly, to provide a model implementation of generic algorithms and standard components in C++. Although it is this last (concrete) contribution that is most widely celebrated, the true value of STL is more profound. In developing MTL, we attempted to follow the complete theme of STL, rather than simply reconstructing STL in a linear algebra guise.

4 Components and Contracts

Component classes in MTL are designed to be interchangeable. Each component provides the same functionality and public interface as the other components of the same type. This interface is referred to as the component contract, and describes the basic requirements of each component. The semantics and syntax of each function is the same for each component, but is carried out in importantly different ways.

For example, matrices in MTL are described in terms of the local data storage format in conjunction with global mathematical properties. Distributed matrices are additionally described with a distribution. For example, the (sequential)

matrix class, `Matrix` and the distributed matrix class, `DMatrix`, are described as:

```
template <class Data, class Desc> class Matrix;
template <class Dist, class Data, class Desc> class DMatrix;
```

Here, the `Data` class stores all the data of a sequential matrix, or the local data of a distributed matrix. The `Data` class contains functions for constructing, destroying, setting, getting, and iterating over the matrix data. Examples of classes that fulfill the `Data` component contract are: compressed row, compressed column, coordinate, dense row major, dense column major, blocked compressed row, and others. This list can be extended by any class which conforms to the `Data` component contract.

Similarly, the `Desc` (description) class is used to control and manage global matrix properties such as symmetry and structure (e.g., triangular, banded, general).

The `Dist` (distribution) class provides methods for mapping global matrix indices to processors and local matrix indices to control how the matrix is distributed. The same distribution classes can also be used to distribute vectors (which is another example of component reuse). Examples include block cyclic, block, and cyclic.

The `Matrix` and `DMatrix` classes provide a uniform interface to the user for manipulating, querying, and operating on the matrix data through its public interface. Internally, the two matrix classes make calls into their component classes to carry out the matrix operations.

A large number of different types can be represented merely by combining different components together. The total number of matrix storage formats is the product of the number of components within each component type, whereas the number that need to be implemented is simply the sum. This is in contrast to some mathematical libraries, particularly those that use inheritance, which must implement a new class for each matrix type. There are also different levels at which composition can take place—some categories of data storage types can be composed from simpler components. For example, a two-dimensional dense data class component can be combined with a column-orientation component to represent a BLAS general matrix. Furthermore, some two-dimensional data classes can be constructed from arrays of one-dimensional containers. Such constructs are used in many sparse types, such as those based on balanced trees or linked lists. The idea of composition is clearly a powerful one. It allows code to be reused in a number of different situations while also making each individual component small and easy to manage.

5 Iterators

One approach to achieving independence of algorithms and data structures is through the use of iterators (the approach used by STL). Iterators are objects

that generalize access to other objects (iterators are sometimes called "generalized pointers"). The definition of the iterator classes in STL provide the uniform interface between algorithms and containers necessary to enable genericity. That is, each container class has certain iterators that can be used to access and perhaps manipulate its contents. STL algorithms are in turn written solely in terms of iterators

This scheme works well for one-dimensional containers, such as those found in STL. However, it has some serious drawbacks for two-dimensional matrix objects, since different storage formats have different performance characteristics depending on the order in which the elements are accessed. So, while it is possible to construct generic algorithms using one-dimensional container iterators, such an approach will tend to strongly bias one storage format over the other (or add additional inefficiencies).

For example, a relatively efficient way to implement the matrix-vector product routine using iterators is shown below. The increment operator function retrieves the element from the matrix which is closest in memory. In a compressed row storage format, the elements on the same row are closest to each other; the reverse is true for compressed column.

```
template <class MAT, class VEC1, class VEC2>
void matvec_product(const MAT &A, const VEC1 &x, VEC2 &y)
{
  for (MAT::iterator i = A.begin(); i != A.end(); ++i)
    y[i->row] += i->value * x[i->col];
}
```

However, this approach does not lend itself to compiler optimizations such as loop unrolling, nor does it efficiently handle dense and block sparse matrices. To obtain high performance while separating data representations from algorithmic implementation, new approaches are needed.

One mechanism for achieving this interface is as follows. Each storage format type implements a function, called `Iterate()`, which is templated on a linear algebra operation object. The `Iterate()` function traverses through the elements of the matrix in the most efficient way. For each new element it calls an inlined function on the linear algebra operation that performs the inner loop of the operation. The object associated with the matrix-vector product operation might look like Fig. 1. The `template<class OP> Iterate()` method is now implemented by each of the data storage formats. It efficiently iterates over the matrix, calling the `inner()` method on the linear algebra object `OP` for each element. Loop unrolling, register blocking, and other optimizations can be used to improve performance. It has been our experience that some compilers optimize this code very well with no performance penalty over specifically tuned code. Unfortunately, other compilers do not optimize this quite as well.

A second approach adopted by MTL is to provide interfaces to the data storage classes that are of appropriate granularity for high performance. The stored data is thus not necessarily traversed element by element, but in larger, multi-element, chunks. The specifics of the dimensions of the traversal can be

```
template <class VEC1, class VEC2, class PR>
class matvec_product_op {
public:
  matvec_product_op(VEC1& _x, VEC2& _y) : x(_x), y(_y) {}
  inline void inner(PR& val, int row, int col) {
    y[row] += val * x[col];
  }
protected:
  VEC1 &x;
  VEC2 &y;
};
```

Fig. 1. Object associated with matrix-vector product.

varied according to the algorithm requiring the traversal. For example, matrix-matrix multiplication benefits greatly from using multiple rows of a matrix.

The results in Fig. 2 show how this second approach compares with the element-wise approach in other linear algebra libraries (the NIST Sparse BLAS [12] and the Template Numerical Toolkit [11]). The test matrices come from standard collections and are each on the order of 10,000 rows and columns. For these test cases, MTL significantly outperforms the other libraries, while still providing a powerful and flexible set of abstractions.

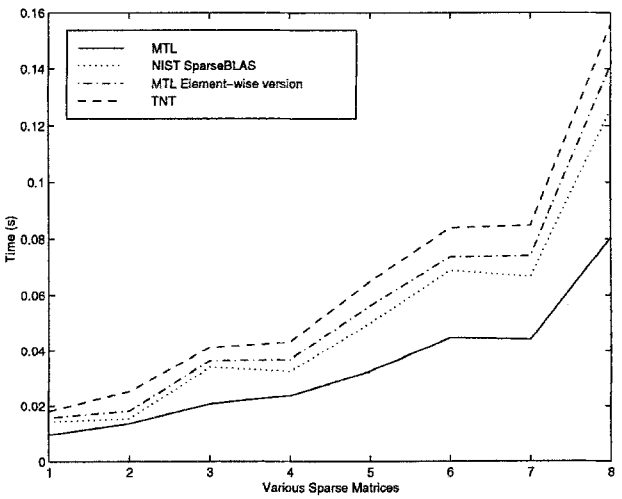

Fig. 2. Execution times for sparse matrix-vector multiplication on Sun UltraSPARC 170E.

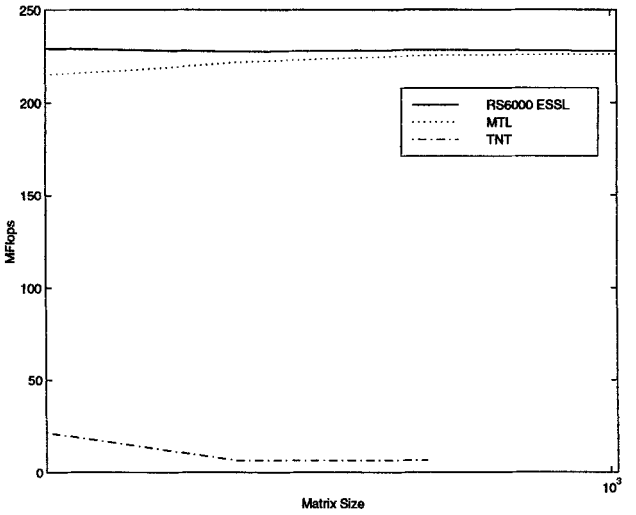

Fig. 3. MFLOPS for dense matrix-matrix multiplication on IBM RS6000.

6 Poly-Algorithms

The approaches described above for high performance are appropriate when there is a single "dominant" storage type. For example, in a matrix-vector product, the layout of the matrix dictates the structure of memory accesses. For matrix-matrix multiplication, the situation is somewhat more complicated, because two stored entities need to be traversed in high-performance fashion, and the traversals must cooperate to properly perform the matrix-matrix multiplication. MTL addresses this issue through the use of poly-algorithms, whereby different implementations of the same functional routine are invoked depending on some compile-time or run-time information.

The sequential case is addressed by using a blocking algorithm, which converts or copies portions of the matrices into a buffer for repeated reuse [6]. The blocked algorithm then dispatches a matrix multiplication kernel based on the type of the matrices. For example, MTL contains kernels for multiplying dense matrices, sparse matrices, and blocked sparse matrices. Dispatching the appropriate kernel is a compile-time decision, so there is no performance penalty. However, even in the parallel case, where dispatching may need to be performed at run time (depending on the parallel run-time environment), the dispatch is only done once per algorithm invocation, and thus has inconsequential overhead.

Figure 3 compares execution times for dense matrix-matrix multiplication for MTL, the Template Numerical Toolkit (TNT), and the vendor supplied BLAS (ESSL) on an IBM RS6000 Model 590. Although MTL is completely written in C++, it achieves nearly the performance of ESSL (recognized as being one of the best vendor libraries available).

7 Conclusions

Our initial experiences with MTL are encouraging and have borne out our conjecture that abstraction and high performance are not necessarily mutually exclusive. However, our studies with MTL thus far have been of an abstract benchmark nature. Although such studies have some value, one of the goals of MTL is to ease the development of large-scale scientific software. Future work will therefore focus on incorporating MTL into real application codes.

References

1. Susan Atlas et al. POOMA: A high performance distributed simulation environment for scientific applications. In *Proceedings Supercomputing '95*, 1995.
2. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*. Birkhauser, 1997.
3. 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.
4. J. Dongarra, Andrew Lumsdaine, Xinhui Niu, Roldan Pozo, and Karin Remington. A sparse matrix library in C++ for high performance architectures. In *Proceedings Object Oriented Numerics Conference*, Sun River, OR, 1994.
5. J. Dongarra, R. Pozo, and D. Walker. LAPACK++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing '93*, pages 162–171. IEEE Press, 1993.
6. Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV Proceedings - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM Press, 1991.
7. Meng Lee and Alexander Stepanov. The standard template library. Technical report, HP Laboratories, February 1995.
8. Andrew Lumsdaine and Brian McCandless. Parallel extensions to the matrix template library. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1997.
9. David R. Musser and Alexander A. Stepanov. Generic programming. In *Lecture Notes in Computer Science 358*, pages 13–25. Springer-Verlag, 1989.
10. David R. Musser and Alexander A. Stepanov. Algorithm-oriented generic libraries. *Software–Practice and Experience*, 24(7):623–642, July 1994.
11. Roldan Pozo. Template numerical toolkit for linear algebra: high performance programming with C++ and the standard template library. In *Proceedings ETPSC III*, August 1996.
12. Karen A. Remington and Roldan Pozo. *NIST Sparse BLAS User's Guide*. National Institute of Standards and Technology.
13. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.
14. The HPC++ Working Group. HPC++ white papers. Technical report, Center for Research on Parallel Computation, 1995.