

# On the Reusability and Numeric Efficiency of C++ Packages in Scientific Computing

Ulisses Mello and Ildar Khabibrakhmanov

IBM T. J. Watson Research Center, Yorktown, NY, USA

**Abstract.** In this paper, we discuss the reusability and numerical efficiency of selected Object-Oriented numerical packages, serial and parallel, for developing high performance scientific computing applications. We selected packages that can be obtained freely on the internet and most of them are open source. Since the authors did not have extensive previous experience with all the packages, we believe that we approximately reproduced the typical process that an experienced OO programmer undergoes to apply the fundamental OO concepts to component reusability and general programming in new and existing OO scientific computing applications. We attempted to apply these concepts in the selection of numerical containers, defined as class with the main purpose of holding objects, such as vectors and matrices that could be both efficient and well tested for dense matrix operations. Our results indicates that in general serial OO packages still lags behind well-tuned non-OO libraries that supply BLAS type operations. Surprisingly, simple containers from the Standard Template Library (STL) outperformed most of OO libraries that used more sophisticated techniques such as expression templates. Careful wrapping of non-OO libraries seems to be the best way to obtain good performance and reusability. Most parallel OO packages use this approach rather than reimplementing BLAS operations natively. Thus they are more attractive in terms of performance and reusability than their serial counterparts.

## 1 Introduction

Object-Oriented Programming (OOP) has proven to be useful paradigm for programming complex models. Scientific applications are growing in size and complexity, making it more challenging to ensure software quality, robustness and performance. Because of the complexity associated with software development, the reuse of well designed and tested software components is highly desirable. OOP, by design, offers the technology to create reusable components that can be used for generic programming in scientific computing. An excellent overview of reusable software and algorithms can be found in [1].

Scientific computing has been traditionally expressed in the Fortran language and despite the recent interest in expressing OOP paradigms in languages such as Fortran90 [2,3,4], C++ is still the dominant OO language in scientific computing, despite its complexity. Barton & Nackman [5] advocated C++ as a replacement

for Fortran in engineering and scientific computing due to its availability, portability, efficiency, correctness, and generality. These authors used OOP for code reorganization of LAPACK (Linear Algebra PACKage) [6], and they were able to group and wrap over 250 Fortran routines into a much smaller set of classes, which expressed the common structure of LAPACK. In this reorganization, they took advantage of good, well-tested Fortran and C code without any reimplementation. They just created a new interface for these routines which improved their organization and usability and had minimal impact on their performance. The use of C++ for numerical linear algebra has been slow due to the difficulty in obtaining computational efficiency. However, in recent years various OO containers and linear algebra (LA) packages have been implemented in C++ using sophisticated techniques such as expression templates, static polymorphism, generative optimization, etc. In addition, compilers have improved so as to include new C++ optimization techniques. Some packages, such as DiffPack [7] and Blitz++ [8,9] claim that they have near Fortran performances under specific conditions (compilers and platforms). However, it is not clear in the literature how the numeric efficiency of OO C++ codes have evolved in Intel based platforms that dominate the Linux cluster market.

We are developing C++ applications in the area of Petroleum Exploration and Production for solving PDE's numerically using the Finite Element and Finite Volume methods. In these applications, we have implemented container classes for storing information about unstructured meshes. These classes use other basic STL containers such as vectors and lists. However, in these applications, a large fraction of the total computation time is spent on LA operations involving matrices and vectors, such as the solution of nonlinear systems resulting from implicit time discretization. Typical BLAS type operations such as inner product (DOT), vector update (AXPY), dense (GEMV) and sparse matrix-vector multiply are also performed outside of the context of the solution of the systems. Thus, it would be desirable to use vector and matrix containers that could exhibit good performance for linear algebra operations both in serial and in parallel. Recently, many OO software packages have been developed that provide matrix and vector containers that are designed for scientific computing applications. Some of them use non-OO external libraries to provide BLAS type operations and others have implemented the operations natively. To compare such packages is not an easy task because they vary largely in objective and implementations. To evaluate the performance of selected C++ OO packages, we decided to use some selected key BLAS type operations. Traditionally highly optimized Fortran, C or Assembly code is normally preferred for these critical parts of software, while OO techniques are more popular for organizing large-scale software components. This approach is based on the observation that performance conscious usage of OO features, for example restricting object orientation to the high-level administrative code, does not affect the overall efficiency. For serial programs, the direct benefits are code re-usability and extensibility. These benefits of OO techniques in serial codes are also extensible to the development of

parallel algorithms in distributed memory environments, where one has to deal with data partitioning and organizing message passing communications.

In this paper, we report the initial results of a comparative study of existing serial and parallel OO libraries performing a selected number of LA operations. This work was largely based on the interesting results reported by the BTL (Benchmark for Templated Libraries) framework, BTL [10], on serial numeric linear algebra libraries. The BTL is an open source project, which reported benchmarks for several open source serial OO libraries, such as Blitz++ [11], MTL [12], ublas [13], and tvmet [14], for basic BLAS type operations. The results for template libraries were compared with more traditional, non-OO libraries, such as Netlib’s BLAS, ATLAS, as well as with raw native C, C++ (STL) and Fortran 77 implementations. BTL has benchmarked Level 1 BLAS type Vector update operation (AXPY), level 2 matrix–vector, and level 3 matrix–matrix product operations on Intel processor-based hardware using the GNU compiler collection.

We extended the BTL methodology to include other compilers (Intel icc and ifc) on the Linux platforms for other serial packages. Furthermore, we also included OO parallel packages in the benchmark analysis. Note that we have not performed a comprehensive search to find the optimal flags for each compiler for each package. In most of the cases, we used the flags suggested by the authors of the packages, normally included in the configuration files of the package distributions.

We started the benchmarking process with packages containing serial code as will be described in detail later in this paper. We consider the results of the serial codes of interest to all Linux cluster communities because they capture the behavior of LA operations that are normally present in individual nodes of a cluster in parallel applications.

## 2 BTL Benchmark Methodology

In order to benchmark OO packages using the BTL methodology, the OO package must provide matrix and vector containers to store the data and a minimum set of methods operations. Normally, in the test drivers the data is stored first in STL containers and then converted to the native OO containers of the package to be tested. Subsequently, a set of general programming algorithms are called to benchmark such operations. Because the end user does not have to know how the data is stored nor how the operations are computed, this methodology stresses the most fundamental characteristics of OO programming, which are the code reusability and general programming paradigm. In theory, the end user could ”plug-and-play” different containers in the main application without rewriting any algorithm using the replaced container.

We provide some code fragments in the Appendix 1,2 and 3 to exemplify how the benchmark is performed and how to create an interface class for a particular package. BLT uses modern C++ template techniques to create a general benchmarking framework.

For this paper, we used packages that can be obtained freely on the internet, and most of them are open source. We used the documentation available at the appropriate sites to install and test the software. Since the authors did not have extensive previous experience with all the packages we believe that we approximately reproduced the typical process that an experienced OO programmer undergoes to apply the fundamental OO concepts of component reusability and general programming in new and existing OO scientific computing applications.

### 3 OO Techniques and Performance Issues

C++ is a very flexible and powerful OO language. Without doubt numerical packages can benefit from using its features. However, one has to be careful when developing high performance applications using C++, since some of the language features can lead to very poor performance. There are many good references [15,16] discussing the performance impact of operator overloads, dynamic polymorphism, etc.

A very well known example of dramatic loss of performance is the use of operator overload in LA operations, such as the matrix-vector product:

$$x = M * v, \tag{1}$$

where  $M$  is a matrix and  $x$  and  $v$  are vectors. Severe decrease in performance is caused by the creation of temporary copies of the objects in this operation. Most compilers are unable to avoid the creation of these temporary objects, implementing the following operations:

$$t = M * v; x = t, \tag{2}$$

where  $t$  is a temporary vector created by the compiler. Temporary object creation occurs even when an efficient external library is called and the matrix and vector objects are passed by reference to the external function. For example, a naive programmer could implement an operator with the form:

```
Vector operator * (Matrix& M, Vector& V) {
    call BLAS_GEMV;
}
```

This would still require the creation of a temporary vector to store the result of the matrix-vector product. For such objects, both large and small, the penalty for temporary object creation is unacceptable because of the time required to allocate and copy the object data.

There are at least two solutions to the problem of temporary object creation in C++. The first and more traditional solution, popular in the C++ community, uses the so-called "composition closure objects" technique to defer the operation evaluation. This technique is discussed in detail by Bjarne Stroustrup [15] in connection with numerical calculations in C++. In order to defer the evaluation of the expression above, the binary operator `Vector operator * (Matrix& M,`

`Vector& V`) should store the references from the operands along with other necessary information to perform the operation and return a small temporary object `MVmul`, instead of performing the operation right away. Consequently, the actual operation is delayed and it is only done when this small temporary object is assigned to the vector storing the result. At that point, references to all objects, including, the one that stores the result of the computation, are available and no copy of any object is necessary. Because all intermediate functions are very small, they are inlined by the compiler and the overall result is just one function call per expression, something in the form of:

```
X.operator=(MVmul(M,V))
```

The problem with this approach is that intermediate objects, like `MVmul` must be coded by hand for every type of the expression in which it occurs. This limits the complexity of expressions allowed in the code. Although this approach can be implemented easily for a limited set of BLAS type operations, it requires more maintenance because new operation classes have to be developed when new expressions are created.

This process can be automated by using C++ templates in a technique called "expression templates," which inlines the necessary calls for arbitrarily complex expressions at compile time. This second solution is generic in terms of the ability to evaluate sufficiently complex expressions. In practice, it is limited only by the compiler's ability to handle recursive template evaluations. Packages such as Blitz++, POOMA, ublas, and MTL, make extensive use of the expression template technique, which is very promising as compilers become more sophisticated. In this paper, we make comparisons between these packages and our simple implementation of vectors and matrices, BSM, which is based on the `STL::valarray`, a container recently provided by the STL for numerical calculations.

Another important technique that has been used by several C++ packages to boost performance of small objects is the template metaprogramming to generate specialized algorithms, and for example, unrolling loops automatically. The following example is a template metaprogram from the Blitz++ documentation. Here, the code:

```
TinyVector<double,4> a, b;
double r = dot(a,b);
```

At compile time this code expands to:

```
= meta_dot<3>(a,b);
= a[3]*b[3] + meta_dot<2>(a,b);
= a[3]*b[3] + a[2]*b[2] + meta_dot<1>(a,b);
= a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + meta_dot<0>(a,b);
= a[3]*b[3] + a[2]*b[2] + a[1]*b[1] + a[0]*b[0];
```

Effectively unrolling the dot product loop. Note that the template function `meta_dot` is inlined recursively at compile time.

## 4 Serial OO Packages

In this section, we compare the performance of LA operations for several serial packages including: A++, ATLAS, Blitz++, GOTO, MTL, Ublas as well as raw C, f77, and STL\_algo code. For example, the level 1 BLAS type AXPY operation using the STL vector container is implemented as:

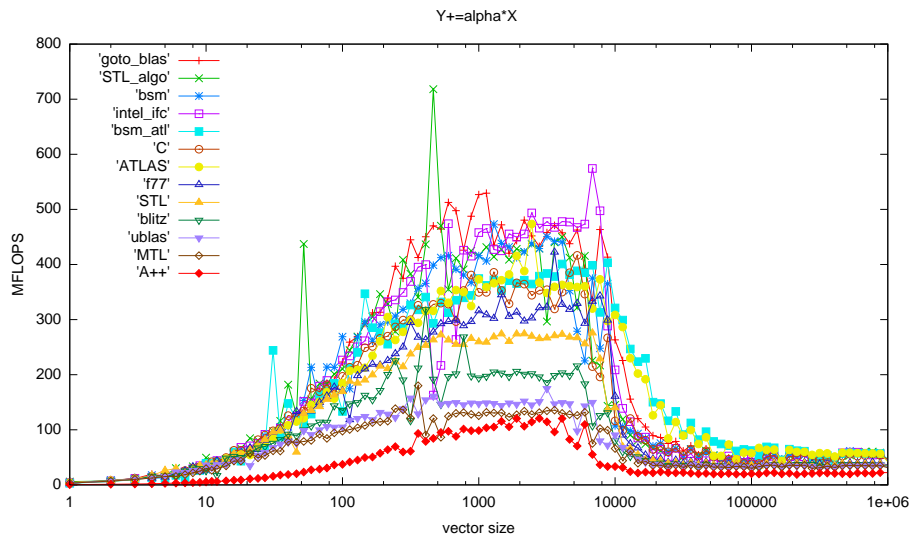
```
inline void axpy(real coef, const stl::vector<real> & X,
stl::vector<real> & Y) {
    for (int i=0;i<X.size();i++)
        Y[i]+=coef*X[i];
}
```

This simple operations was also implemented by the BTL creator, Laurent Plagne, using the STL::transform algorithm (see Appendix 3). As mentioned before, we tested an STL valarray implementation using a set of container we implemented called BSM, which also uses the "composition closure objects" technique described above to avoid the creation of temporary objects in algebraic expressions. We also used the BSM interface to wrap other packages in our scientific applications. For example, BSM.ATL is the BSM package compiled with a USE.ATLAS flag, enabling the call of corresponding ATLAS function instead of the stl::valarray's inner\_product() method.

Some general performance characteristics of these packages can be extracted from Figures 1 to 3. These figures represent typical dense-matrix LA operations for BLAS level 1 to 3, respectively. They were generated using the GNU Compiler Collection (version 3.2.2) on RedHat 7.3, on an 8-node IBM Linux cluster 1300 with a Myrinet switch. For the serial tests, we used a single node that has two 700MHz Pentium III processors, each with 256KB of L2 cache and 32KB L1 cache.

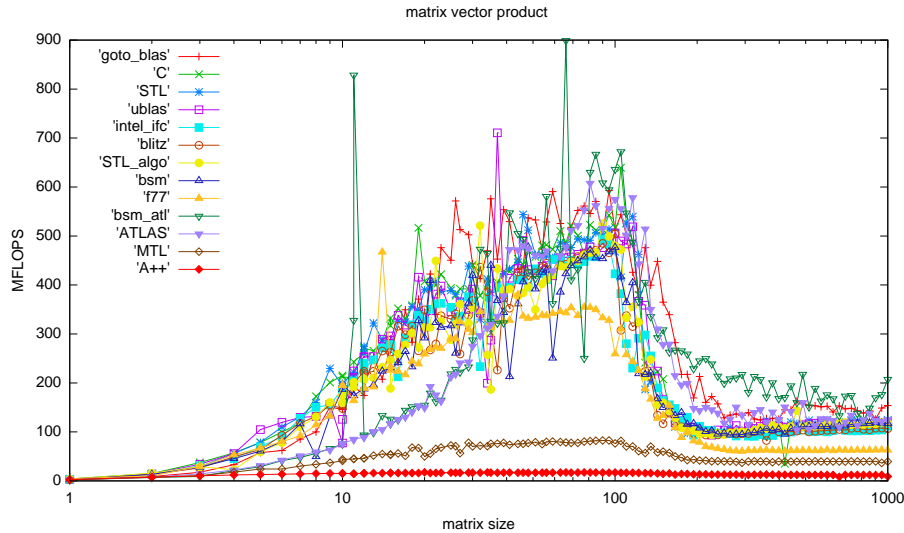
We used the GNU Compiler with -O3 optimization, and no gcc specific extensions for case of vector update (AXPY) operation. This operation is frequently used as a measure of memory performance because the MFLOP rate is limited by the storage access rate. Figure 1 shows clearly that the performance drops significantly when vector sizes ( $\approx 16K$ ) exceed the L2 cache size. For vector sizes smaller than 300, the low performance is caused mainly by compulsory cache misses. The best performance is obtained for intermediate vector sizes that are L2 resident after the data was loaded.

Note that the memory access time also controls the copy operation between objects. OO packages normally use raw C arrays to store the data internally in vector and matrix classes. Access value\_type operator[](size\_t i) operators normally are inlined and thus no impact on performance should be expected when using access operators. Unfortunately, if the raw C arrays are replaced by OO containers, not all compilers perform the common code optimization that would occur with raw C arrays. We noticed a significant drop in performance when we copied objects using no aggregate operations. Most likely the compiler is not generating code that pre-fetches the data or it is not inlining the code optimally.



**Fig. 1.** Rates for the level 1 BLAS type AXPY operation  $X+ = \alpha Y$  using different libraries.

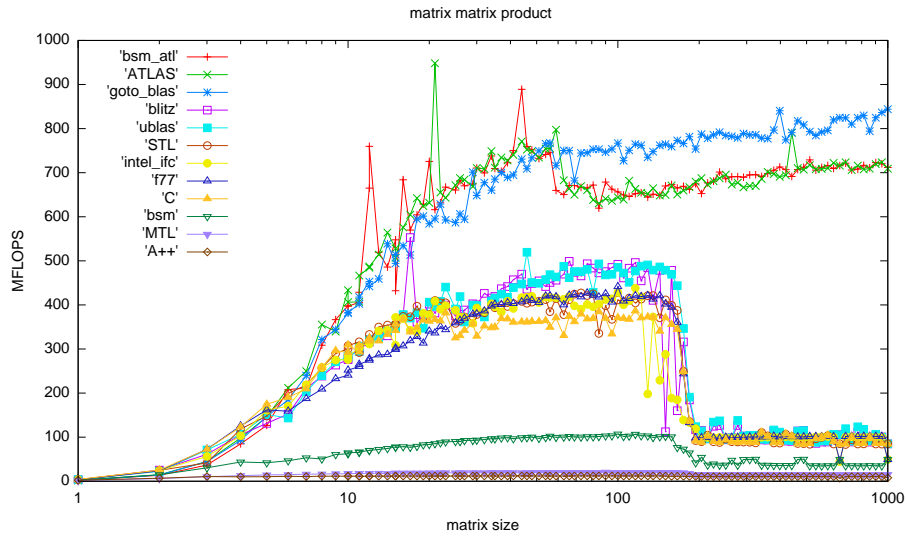
We can observe that the performance of the tested packages varied by a factor of 4. The slowest at  $\approx 100$  MFLOPS and the fastest at  $\approx 400$  MFLOPS. The worst performers are the C++ OO packages. All of them performed worse than the Fortran 77 BLAS package compiled with g77. The best performer was the Fortran 77 BLAS package compiled with Intel ifc compiler. Interestingly, the `STL::transform` algorithm performed quite well on the `STL::vector` update. Although the `STL::valarray` has been designed to provide performance superior to that of `STL::vector` for numerical operations, the BSM package using the `stl::valarray`'s `inner_product` method yields performance similar to that of the regular `STL::vector` container. For very large vectors, ATLAS has the best performance while GOTO BLAS [17] and Pentium III optimized BLAS library shows very good performance for intermediate sizes. Rather unexpectedly, expression templates based libraries show poor performance. In general their performance is worse than the straightforward `stl::vector` implementation shown above. However, OO packages that use template metaprogram techniques (e.g., Blitz++, uBLAS, and POOMA) to effectively unroll loops automatically, have good performance for very small objects. For example, in Figure 3, uBLAS was a top performer along with raw C in matrix-matrix products for matrix sizes smaller than 8. In addition, expression templates seem to be very efficient in removing the function call overhead for small objects. This is very clear for matrix-vector product (Figure 2) and matrix-matrix products (Figure 3). Again, it is noteworthy that the straightforward implementation using `stl::vector` and `stl::vector<stl::vector >` is also competitive for small sizes.



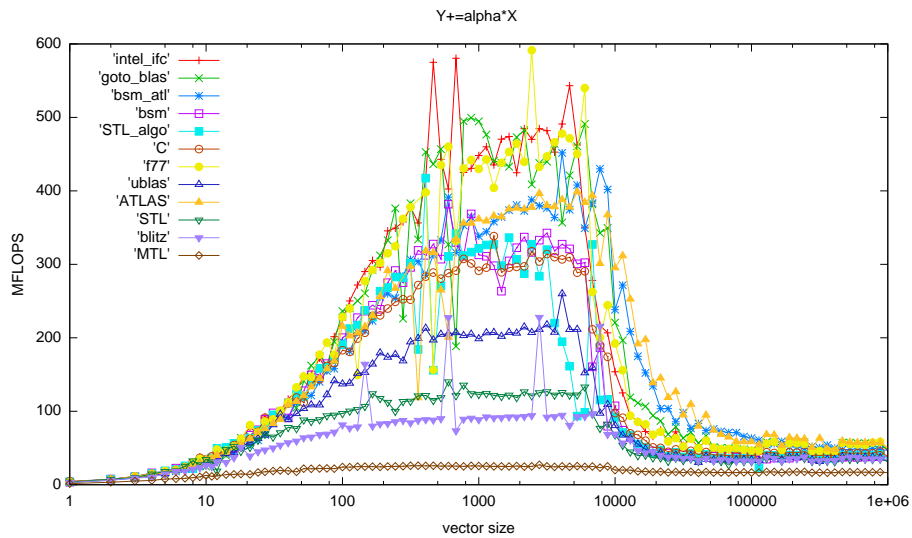
**Fig. 2.** Rates for the level 2 BLAS type gemv operation  $X = MY$  using different libraries.

For matrix-matrix products (Figure 3) the advantage of cache size tuned ATLAS and vendor specific GOTO BLAS for relatively larger matrices is clear, especially for sizes ( $> 120$ ) exceeding L2 cache size. This shows the value of generative optimization techniques that ATLAS utilizes. In ATLAS, the system-specific routines are isolated and the code necessary for these routines are automatically created using timings to determine the correct blocking and loop-unrolling factors to optimize the performance. Our BSM\_ATL that wraps around ATLAS, allows the reutilization of non-OO packages, such as ATLAS, as our containers. We can see that `stl::valarray` based implementation of vectors can be very efficiently mapped to this external BLAS implementation. This allows an expression, such as  $x = M * v$  in a C++ program, to be calculated using the most efficient BLAS implementation. In fact, this implementation is even capable of switching from one BLAS library to another for different container sizes.

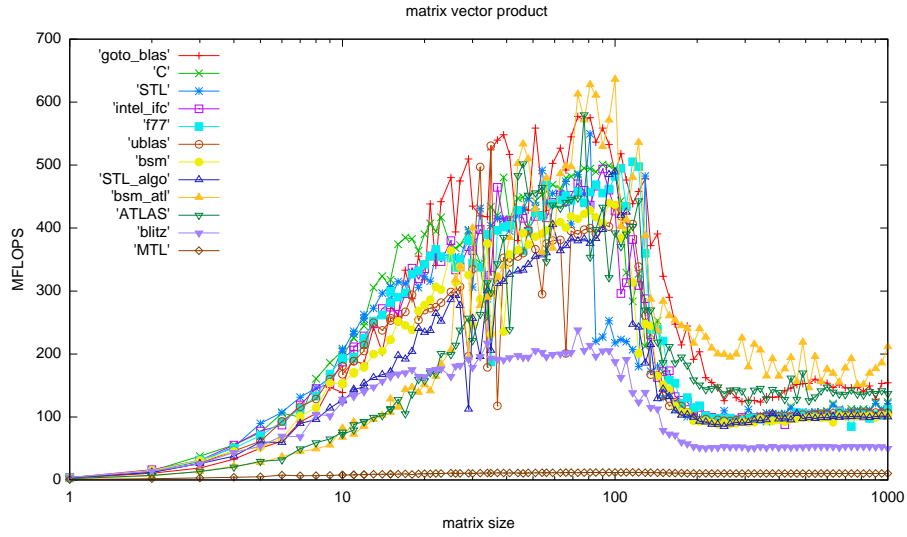
Figures 4 to 6 show the same operations as those illustrated in Figures 1 to 3 but using the Intel compiler rather than GNU compilers. In general, the performance of the selected operations exhibits similar behavior to those compiled with the GNU compilers. However, non-OO packages presented an increase in performance from 10 to 20% while the opposite happened with some OO packages such as Blitz++. Unfortunately, the STL implementation in the Intel compiler is not as mature as the one present in the g++ compiler. This is especially true for the matrix-matrix product (Figure 3). We did not investigate the exact cause for the poor performance in this operation, but based on some of our experience with icc, it seems that there is still room for optimization in its STL implementation.



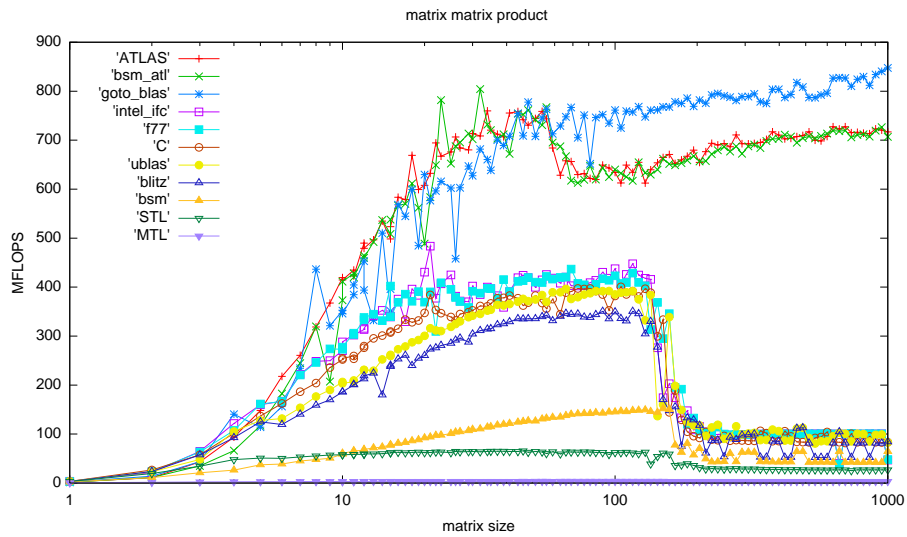
**Fig. 3.** Rates for the level 3 BLAS type gemm operation  $X = MY$  using different libraries.



**Fig. 4.** Rates for the level 1 BLAS type AXPY operation  $X += \alpha Y$  using different libraries and Intel compiler.



**Fig. 5.** Rates for the level 2 BLAS type gemv operation  $X = MY$  using different libraries and Intel compiler.



**Fig. 6.** Rates for the level 3 BLAS type gemm operation  $X = MY$  using different libraries and Intel compiler.

In summary, the performance-tuned BLAS libraries, such as ATLAS and GOTO, are clearly superior for the level 3 type operations on large matrices when system-specific optimization is critical. For matrices with size larger than 100, ATLAS has sustained performance 4 times better than any other implementation. OO packages such as STL and ublas had performance close to Netlib's Fortran 77 BLAS while MTL ranked last. For the level 2 and level 1 type operations OO libraries have performed better than raw Fortran 77. However, the BLAS package available from Netlib is general and it not tuned for system-specific features. Vendor BLAS implementations normally outperform Netlib's version.

It is important to note that, despite the fact the BLAS design is not object-oriented it provides the highest software reusability possible. BLAS specification became the unofficial standard for vector and matrix numerical computations largely due to the support from vendors. These hand-tuned subroutines for a number of years provided the best possible performance on many platforms. As a result numerical algorithms were developed to effectively utilize BLAS type operations and any LA package should provide efficient access interfaces to these operations. Fortunately, the development of processor and computer architecture seems to increasingly unify the optimization techniques involved in the creation of highly optimized numerical software libraries. As a result one can attempt to automatically tune numerical procedures to the best performance by trying, comparing and choosing the optimal one from the finite pool of available optimization techniques. In this manner one can generate code automatically, providing very efficient cross platform implementation of BLAS. Efforts such as ublas from Boost [18] may result in similar results, especially if these packages are incorporated using the STL. It is important to note that the C++ STL implementation has performed surprisingly well, on par with native C, and sometimes better than the performance-tuned ATLAS implementation. This result is very encouraging because it suggests that careful OO implementations based on STL can be as efficient as raw C and Fortran 77, producing highly efficient code. This is especially true for LA algorithms that rely only in Level 1 and Level 2 operations.

## 5 Parallel OO Libraries

In this section we discuss the results of using the BTL framework to evaluate parallel OO libraries in distributed memory environments such as Linux clusters. The packages that we selected have been developed using the Message Passing Interface (MPI). These packages make use of OOP techniques to manage the complexity of numerical message-passing codes and to partially conceal parallelism issues from the application developer.

Operations with parallel containers is much more complex than in the serial case because other issues, such as the amount of communication, network topology, bandwidth and latency are critical. Thus an efficient parallel implementation requires careful design to maximize the locality of the operations. Here,

we extended the BTL framework to benchmark several parallel packages for the same selected operations we described in the previous section. We performed the benchmarks on an IBM-1300 Linux cluster, and all software was compiled using the GNU compiler collection version 3.2.2 and MPI/PRO version 1.6.4. When the parallel package was required to be linked to an external serial BLAS library, we used the GOTO BLAS version for the Pentium III.

In the parallel case, it is inherently much more difficult to compare libraries, not only because of the increase in complexity but also because each of these packages were designed to achieve distinct objectives. We selected parallel OO packages that have been used for the solution of PDE's. In addition, we are aware that we are not conducting an extensive set of benchmark operations, and our results are only representative of the reduced set of operations we tested. However, this reduced set of operations was sufficient to evaluate the reusability of the parallel container available in these packages. In sequence that follows, we discuss the results for the packages: PETSc, PLAPACK, ScaLAPACK, POOMA, and P++.

## 5.1 PETSc

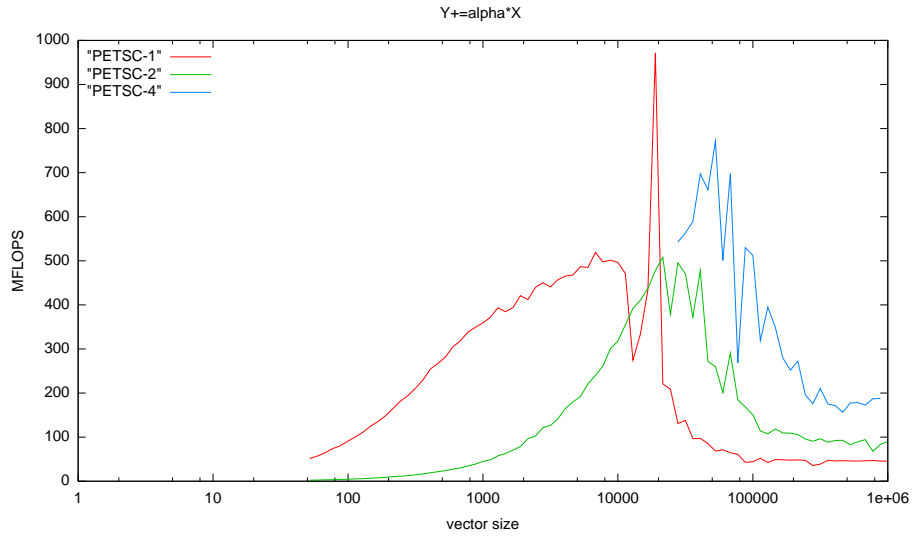
PETSc is implemented in object oriented C and has been designed to provide scalable, robust solutions of linear and nonlinear systems. It provides a high-level mathematics interface as well as low level computational kernels. The BLAS type operations, such as vector update and matrix-vector product, for distributed vectors and matrices are provided but there is no aggregate matrix-matrix product operation available.

It was very simple to integrate PETSc containers into the BTL framework. For proper initialization in the BLT framework, it is was necessary to wrap PETSc objects into a reference counted container. An example of the vector update operation using PETSc is:

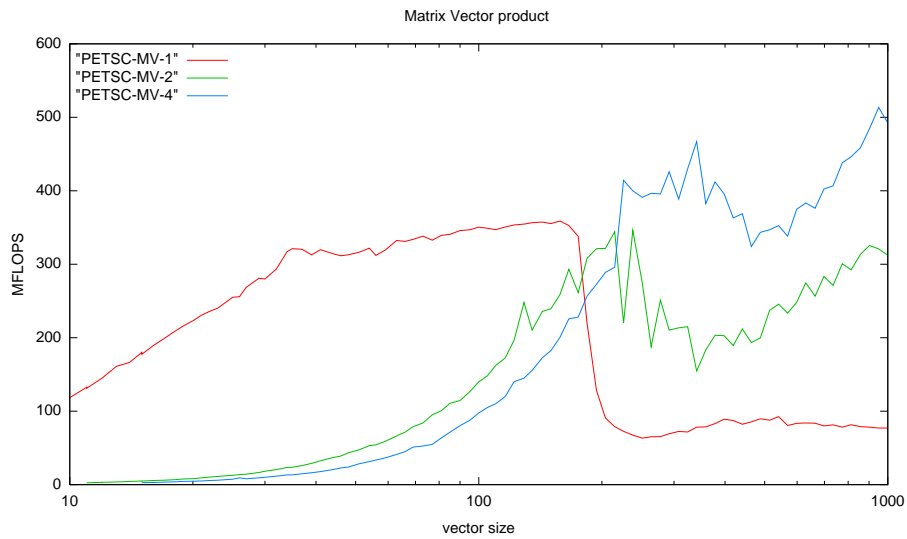
```
static inline void axpy(const real coef, const gene_vector & X,
gene_vector & Y, int N)
{
    VecXPY(&coef,X.object,Y.object);
}
```

The PETSc's API is very well designed but one of the authors had difficulties with `xxxCopy` method family because it expects the source object first and the destination object second. This is opposite to standard C library conventions (e.g., `strcpy(char* dest, char* source)`), and it took a while to get used to this during the debbuging operations.

Figures 7 and 8 display the rates for parallel AXPY and GEMV type operations, respectively. The parallel AXPY operation presents characteristic similar to the serial case. It is interesting to note that as the number of the nodes increases, the curve shape shifts towards larger vector sizes, when all the nodes data sizes exceed the size of their L2 caches. The scaling depends on the vector



**Fig. 7.** Rates for the level 1 BLAS type AXPY operation  $X+ = \alpha Y$  using PETSc.

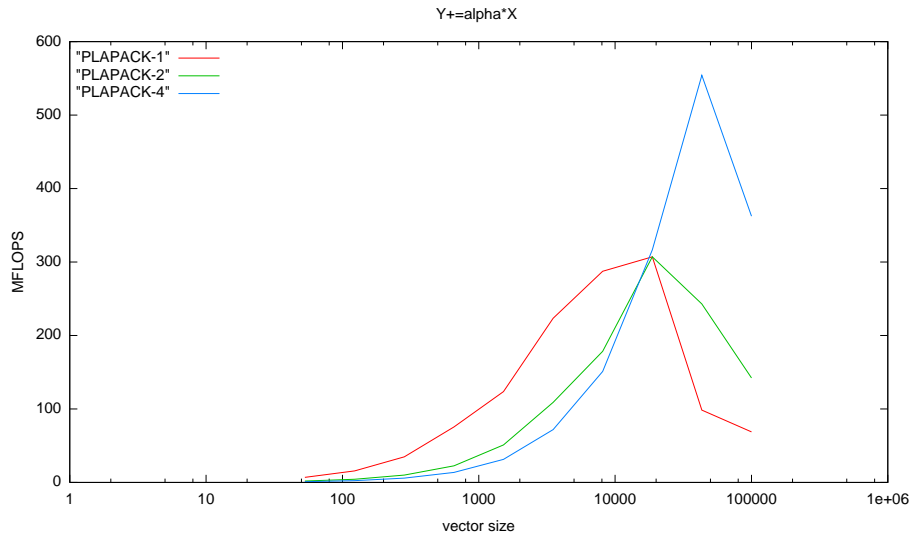


**Fig. 8.** Rates for the level 2 BLAS type gemv operation  $X = MY$  using PETSc.

size. Small vectors present a double performance penalty due to the communication and compulsory cache hit. For vector sizes larger than 20K the operation scales well for peak performance. However, 2 nodes have practically the same peak performance as a single node, showing the impact of the communication cost in this operation. The matrix-vector multiply (Figure 8) does not show the typical drop in performance for larger sizes observed in the serial case for this operation. However, this analysis is limited because of the small number of nodes used in the test.

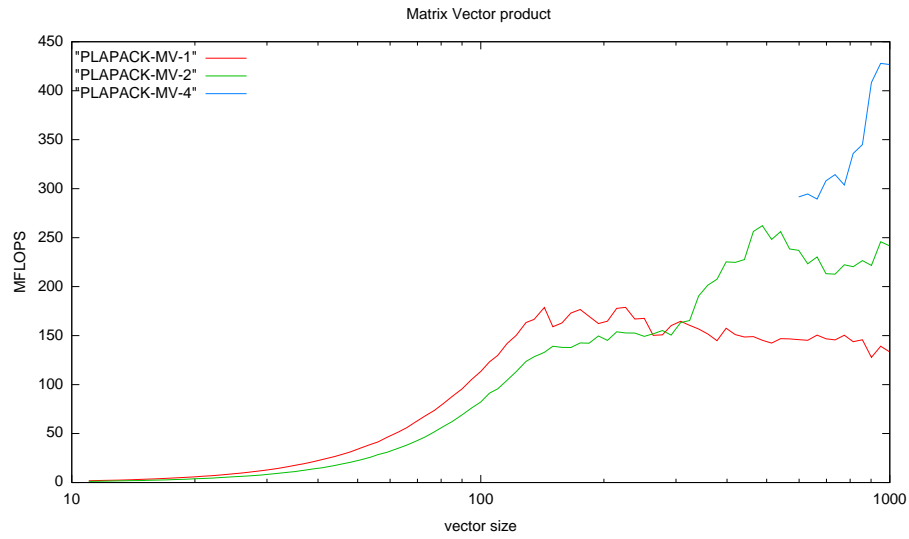
## 5.2 PLAPACK

PLAPACK represents an example of good design, well thought interface, implemented using object-based approach in C and it depends on an external serial BLAS library. It is well documented and the user manual was published in 1997 [19]. This library needed more work to be tested in the BTL framework, due to the fact that version 3.0 (downloaded from the web site), does not provide the fully implementation for all objects in some important functions (e.g., `PLA_Copy`) documented in the manual.

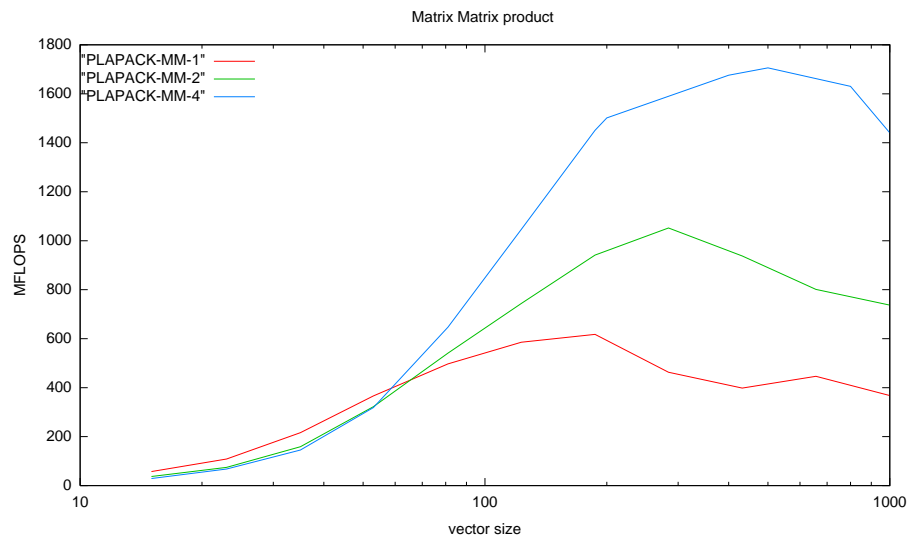


**Fig. 9.** Rates for the level 1 BLAS type AXPY operation  $X+ = \alpha Y$  using PLAPACK.

PLAPACK provides an aggregate version of the matrix-matrix multiply and the performance of the other selected operations are slightly better than PETSc. However, this version proved to be unstable for some matrix sizes. We are investigating the causes of this behaviour.



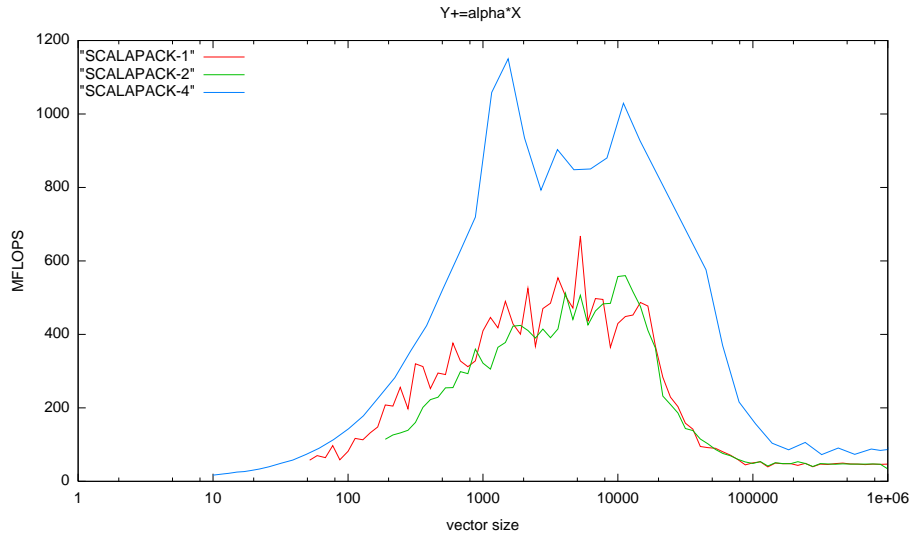
**Fig. 10.** Rates for the level 2 BLAS type gemv operation  $X = MY$  using PLAPACK.



**Fig. 11.** Rates for the level 3 BLAS type gemm operation  $X = MY$  using PLAPACK.

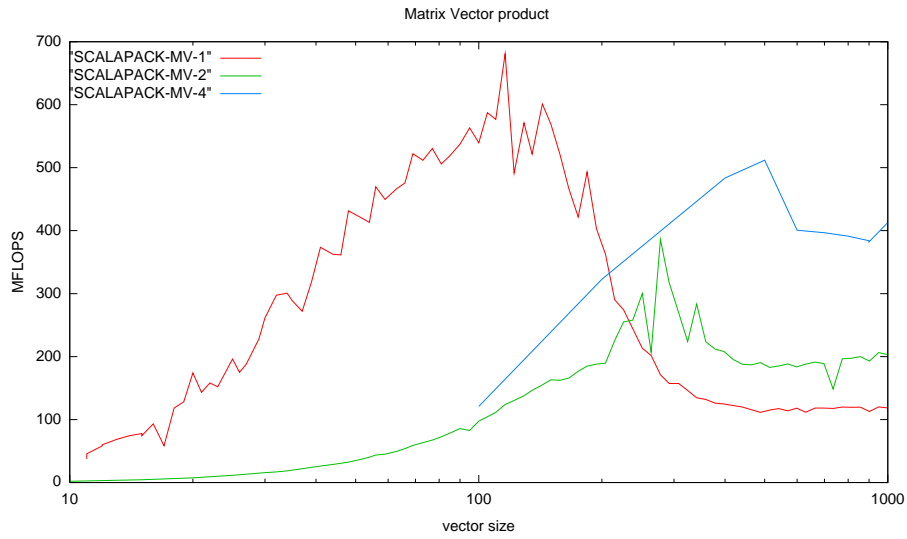
### 5.3 ScaLAPACK

ScaLAPACK is not a parallel, non-OO package but it extends the LAPACK library to parallel environments. It uses a more traditional multilayered type of software design pattern. It uses PBLAS, a parallel BLAS implementation and the BLACS communication library.

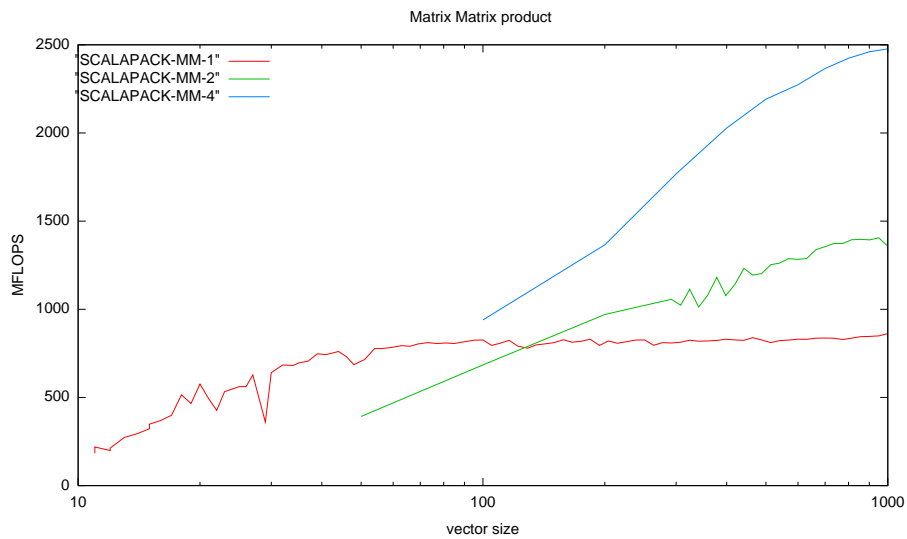


**Fig. 12.** Rates for the level 1 BLAS type AXPY operation  $X+ = \alpha Y$  using ScaLAPACK.

As can be seen in Figures 12 to 14, ScaLAPACK achieves good performance. There is no speed gain in AXPY operation in Figure 12 when moving from one to two processors because of the default vector distribution among processors. For  $1 \times 2$  2D topology the vector is distributed on the first node only. This approach is oriented toward matrix-vector multiplication operations. When 4 processors are used,  $2 \times 2$  2D topology is default, vectors are still distributed on the first column of the processor array, which now has 2 processors and the vector update is done approximately twice as fast. If the application is using only BLAS level 1 operations, a change of the default vector distribution policy is necessary. Otherwise, for BLAS level 2 and level 3 operations default provides good scalability, as can be seen from Figures 13 and 14. In term of usability ScaLAPACK is relatively easy to install on Linux. As is the case with most scientific software, it does not provide an automated installation procedure. However, its installation instructions have sufficient details and the makefiles are provided for about twenty different systems.



**Fig. 13.** Rates for the level 2 BLAS type gemv operation  $X = MY$  using ScaLAPACK.

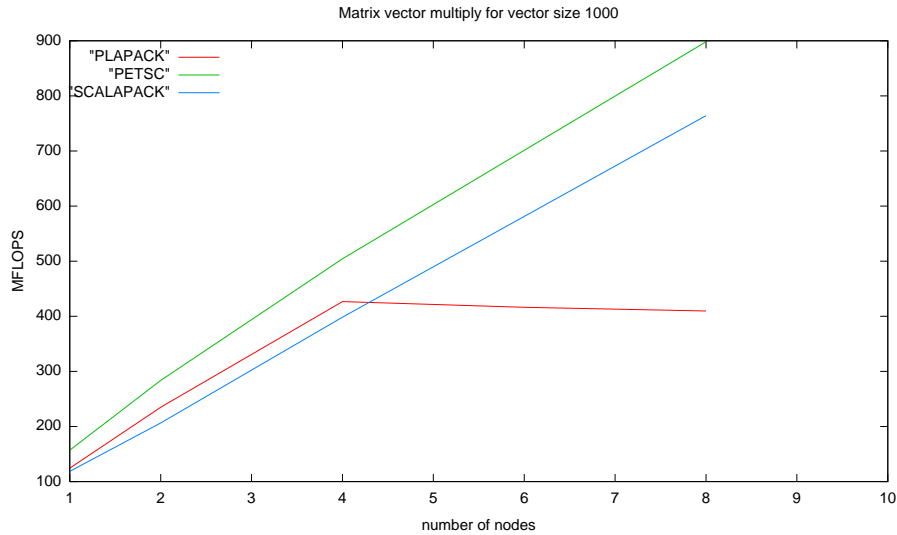


**Fig. 14.** Rates for the level 3 BLAS type gemm operation  $X = MY$  using ScaLAPACK.

## 5.4 P++ and POOMA

P++ is the parallel counterpart of the A++ array library discussed previously. It is used by the Overture package [20]. Implemented in C++ provides very good user interface. The Parallel Object-Oriented Methods and Applications (POOMA) package has been designed to handle multiple large-scale parallel applications, providing physics-based objects that represents arrays, particles, fields, meshes and solvers. Unfortunately, the OO containers provided by P++ and POOMA have no built-in support for aggregate BLAS type operations. Of course, an indexed based, element by element access, implementation results in very poor performance, which is almost 10 times slower than the other packages. These packages seem appropriate for time explicit discretization, and not for problems that have to solve linear and nonlinear systems of equations in parallel.

## 5.5 Scalability



**Fig. 15.** Rates for the level 2 BLAS type gemv operation  $X = MY$  as a function of number of nodes for fixed matrix size 1000.

Figure 15 demonstrates scalability of SCALAPACK, PETSc, and PLAPACK on matrix vector operation for matrix sizes 1000. Due to some yet unidentified reasons PLAPACK has demonstrated a significant drop of performance when the number of nodes increased from 4 to 8. PETSc and SCALAPACK scale well and SCALAPACK has slightly better performance than the other packages in this benchmark.

## 6 Discussion and Conclusions

We found that BTL provides an effective framework for benchmarking BLAS type operations for serial and parallel numerical containers as well as for OO and non-OO packages. Because of its modern design, it can accommodate distinct policies for timings and operations. We hope that this open source effort evolves to allow benchmarks with various compilers in various systems.

Based on our results, it is clear that serial non-OO packages are more mature than the serial OO packages. Despite the redundancy and complexity of the LAPACK and BLAS APIs, it seems that it is worthwhile to leverage this software in high performance OO scientific computing. The wrapping approach suggested by Barton & Nackman [5] is still preferable to reimplementing these operations using C++. The lack of maturity of some OO packages should also be an important consideration. Many of these packages are not actively maintained and some of them still do not provide all the basic operations with all the containers. Paradoxically, more complex, parallel OO packages are in general better documented and maintained than the serial ones. PETSc and POOMA seem to be the most mature of the lot. However, for the solution of linear systems PESTc is the most robust and provides more functionality. In order to reach Fortran like performance using C++, serial OO packages tend to use the latest features of the language. Unfortunately, not all compilers possess the optimization necessary to reach their peak performance. This creates a very tight dependence on compilers. Traditional optimization techniques need to be modified to work with nested aggregates and temporary objects, which are very common in C++. For example, most of the results reported in the literature of very good performance of OO packages such as Blitz++ were obtained using the KCC compiler which was a top-notch compiler for optimizing C++. KCC was acquired by Intel and has been discontinued. At the present time the Intel C++ compiler (icc) does not provide the same level of optimization in C++ that KCC provided.

In terms of reusability, packages that do not allow at least one-time dynamic allocation of containers cannot be easily incorporated in pre-existing codes. This is especially true for packages that use template metaprogramming techniques to unroll recursively loops. For example, consider a pre-existing mesh container class that uses other basic containers internally to store node positions and cell connectivity:

```
class mesh {
protected:
    matrix<int> element_connectivity;
    matrix<double> node_positions;
public:
    ...
};
```

The size of the containers `element_connectivity` and `node_positions` are usually defined after reading the input files and thus they are not known at

compile time. If we use the PETSc containers, it is simple to replace the declaration of these basic containers and create/resize them at run time. Using good design, it is possible to allocate the containers just once and reuse the container to avoid performance penalties associated with reallocation. In this particular case, PETSc offers such flexibility in that it is possible to define if the container will be serial or parallel at run time. However, if one wants to use the array containers from Blitz++, the dimension of the arrays has to be defined at compile time, like Fortran 77, otherwise template metaprogramming recursion operations cannot be used. If we do not want to hardwire a particular size for the arrays, this requires making the mesh class a templated class:

```
template<int NDIM>
class mesh {
protected:
    Array<int,NDIM+1> element_connectivity;
    Array<double,NDIM> node_positions;
public:
    ...
};
```

Even making this change, at some point the dimension in the templated mesh class has to be defined at compile time:

```
mesh<2> mesh2D;
mesh<3> mesh3D;
```

This would reduce the flexibility in retro-fitting of pre-existing code that use the same container independently of the mesh dimension.

The STL standard does not include matrix and vector algebra specifications, the standard only provides the lowest level building blocks for numerical linear algebra in the form of the `stl::valarray`. The overall performance of the C++ STL based solution using modern compilers opens avenues for OO methods into high performance computing. We believe this can play an important role in the implementation of OO scalable parallel software libraries in the future. STL and Boost can make it possible the provision of a common simplified API and at the same time hide efficient implementations of parallel algorithms.

Our results suggest that careful implementation could take advantage of highly tuned low level libraries and provide a logical and easy to use interfaces, hiding all implementation details from the user, all without compromising the resulting speed. The BTL suite is itself a wrapper around the objects and methods of the particular library and thus the implementation of this wrapper interface itself gives a very good sense of the reusability of the library.

Overall results of our benchmark show that the increasing complexity of processors and computer architectures introduces new opportunities for optimization. Even mature Fortran code may require additional tuning to perform well in new architectures. Thus, generative optimization techniques associated

with the isolation of system-specific routines seems to be an effective approach to generate cross platform portable HPC applications.

Programming for distributed environments is inherently difficult, error prone, and difficult to debug. Our benchmark put a moderately heavy stress on the message passing layer. On some occasions, some operations failed in some of the nodes, for particular data sizes leading to stalled processes on all other nodes waiting for response from the dead process. Most of these failures are reproducible but we did not have time to investigate the cause for such problems. In a OO context, there is a strong need for the development of efficient “distributed exceptions”, capable of detecting runtime problems during the execution of parallel applications, informing the user of the library about the problems and providing ways to gracefully exit all processes.

## 7 Acknowledgements

The authors would like to thank Anshul Gupta for suggestions to improve an early version of this manuscript. We also would like to thank John Gunnels and Andrew Conn for reviewing critically this manuscript.

## References

1. Jack Dongarra, Ian Foster, and Ken Kennedy. Reusable software and algorithms. In Jack Dongarra, Ian Foster, Geoffrey Fox, William Fox, Ken Kennedy, Linda Torczon, and Andy White, editors, *Source book of parallel computing*, pages 483–490. Morgan Kaufmann Publishers, 2003.
2. V. K. Decyk, C. D. Norton, and B. K. Szymanski. How to express C++ concepts in FORTRAN 90. *Scientific Programming*, 6(4):363, 1997.
3. V. K. Decyk, C. D. Norton, and B. K. Szymanski. Expressing object-oriented concepts in FORTRAN 90. *ACM FORTRAN Forum*, 16(1), 1997.
4. V. K. Decyk, C. D. Norton, and B. K. Szymanski. How to support inheritance and run-time polymorphism in FORTRAN 90. *Computer Physics Communications*, 115(9), 1998.
5. John J. Barton and Lee R. Nackman. *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples*. Addison-Wesley, Reading, MA, 1994.
6. Z. Bai, C. Bischo, J. Demmel, and J. Dongarra. *LAPACK Users’ Guide*, page 325. Society for Industrial & Applied Mathematics, 2 edition, 1995.
7. E. Arge, A. M. Bruaset, P. B. Clavin, J. F. Kanney, H. P. Langtangen, and C. T. Miller. On the numerical efficiency of C++ in scientific computing. In Morten Dhlen and Aslak Tveito, editors, *Numerical Methods and Software Tools in Industrial Mathematics*, page 91. Birkhäuser, 1997.
8. T. L. Veldhuizen and M. E. Jernigan. Will C++ be faster than FORTRAN? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE’97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
9. Todd L. Veldhuizen. Scientific computing: C++ versus FORTRAN: C++ has more than caught up. *Dr. Dobb’s Journal of Software Tools*, 22(11):34, 36–38, 91, November 1997.

10. Laurent Plagne. Bench templated libraries  
<http://www.opencascade.org/upload/87>.
11. <http://www.oonumerics.org/blitz>.
12. <http://osl.iu.edu/research/mtl>.
13. <http://www.genesys-e.org/ublas>.
14. <http://tvmet.sourceforge.net>.
15. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 2000.
16. Todd L. Veldhuizen. Techniques for scientific computing C++. technical report # 542. Technical report, Indiana University Computer Science, 2000.
17. <http://www.cs.utexas.edu/users/flame/goto>.
18. <http://www.boost.org>.
19. Robert A. van de Geijn. *Using LAPACK*. The MIT Press, Cambridge, MA, 1997.
20. <http://www.llnl.gov/CASC/Overture/>.

## 8 Appendix 1

Implementation of the generic performance analyzer.

```
template <template<class> class Perf_Analyzer, class Action>
void bench( int size_min, int size_max, int nb_point ){
    ...
    std::vector<double> tab_mflops(nb_point);
    std::vector<int> tab_sizes(nb_point);
    // matrices and vector size calculations
    size_lin_log(nb_point,size_min,size_max,tab_sizes);
    // loop on matrix size
    Perf_Analyzer<Action> perf_action;
    for (int i=nb_point-1;i>=0;i--){
        tab_mflops[i]= perf_action.eval_mflops(tab_sizes[i]);
    }
    ...
}

template <class Action>
class Portable_Perf_Analyzer{
public:
    Portable_Perf_Analyzer( void ):_nb_calc(1),_nb_init(1),_chronos({});
    inline double eval_mflops(int size);
    inline double time_init(Action & action);
    inline double time_calculate(Action & action);
    unsigned long long get_nb_calc( void ) { return _nb_calc; }
private:
    unsigned long long _nb_calc;
    unsigned long long _nb_init;
    Portable_Timer _chronos;
};
```

Implementation of the BTL interface for uBLAS library.

```
template <class real>
class ublas_interface{
public :
    typedef real real_type ;
    typedef std::vector<real> stl_vector;
    typedef std::vector<stl_vector > stl_matrix;
    typedef typename boost::numeric::ublas::matrix<real> gene_matrix;
    typedef typename boost::numeric::ublas::vector<real> gene_vector;
```

```

static void free_matrix(gene_matrix & A, int N){}
static void free_vector(gene_vector & B){}
static inline void matrix_from_stl(gene_matrix & A, stl_matrix & A_stl){
    A.resize(A_stl.size(),A_stl[0].size());
    for (int i=0; i<A_stl.size(); i++)
        for (int j=0; j<A_stl[i].size(); j++) A(i,j)=A_stl[i][j];
}

static inline void vector_from_stl(gene_vector & B, stl_vector & B_stl){
    B.resize(B_stl.size());
    for (int i=0; i<B_stl.size(); i++) B(i)=B_stl[i];
}

static inline void vector_to_stl(gene_vector & B, stl_vector & B_stl){
    for (int i=0; i<B_stl.size(); i++) B_stl[i]=B(i);
}

static inline void matrix_to_stl(gene_matrix & A, stl_matrix & A_stl){
    int N=A_stl.size();
    for (int i=0;i<N;i++){
        A_stl[i].resize(N);
        for (int j=0;j<N;j++) A_stl[i][j]=A(i,j);
    }
}

static inline void copy_vector(const gene_vector & source,
gene_vector & cible, int N){
    for (int i=0;i<N;i++) cible(i)=source(i);
}

static inline void copy_matrix(const gene_matrix & source,
gene_matrix & cible, int N){
    for (int i=0;i<N;i++)
        for (int j=0;j<N;j++) cible(i,j)=source(i,j);
}

static inline void matrix_vector_product_slow(gene_matrix & A,
gene_vector & B,
gene_vector & X, int N) {
    X = prod(A,B);
}

static inline void matrix_matrix_product_slow(gene_matrix & A,
gene_matrix & B,
gene_matrix & X, int N) {
    X = prod(A,B);
}

static inline void axpy_slow(const real coef, const gene_vector & X,
gene_vector & Y,
int N) {
    Y+=coef*X;
}

static inline void matrix_vector_product(gene_matrix & A, gene_vector &
B, gene_vector & X, int N) {
    X.assign(prod(A,B));
}

static inline void matrix_matrix_product(gene_matrix & A, gene_matrix &
B, gene_matrix & X, int N) {
    X.assign(prod(A,B));
}

```

```

    }

    static inline void axpy(const real coef, const gene_vector & X,
gene_vector & Y, int N) {
        Y.plus_assign(coef*X);
    }

    static inline void ata_product(gene_matrix & A, gene_matrix & X, int N) {
        X.assign(prod(trans(A),A));
    }

    static inline void aat_product(gene_matrix & A, gene_matrix & X, int N) {
        X.assign(prod(A,trans(A)));
    }
};

```

Implementation of the BTL main driver for uBLAS library.

```

int main()
{
    // vector update
    bench<Action_axpy<ublas_interface<REAL_TYPE> >
>(MIN_AXPY,MAX_AXPY,NB_POINT);
    // matrix-vector multiply
    bench<Action_matrix_vector_product<ublas_interface<REAL_TYPE> >
>(MIN_MV,MAX_MV,NB_POINT);
    // matrix-matrix product
    bench<Action_matrix_matrix_product<ublas_interface<REAL_TYPE> >
>(MIN_MM,MAX_MM,NB_POINT);
    // matrix-transpose matrix product
    bench<Action_ata_product<ublas_interface<REAL_TYPE> >
>(MIN_MM,MAX_MM,NB_POINT);

    return 0;
}

```

## 9 Appendix 2

Implementation of the BTL interface for PETSc library.

```

template<class real>
class petsc_interface {
public :
    typedef real real_type ;
    typedef std::vector<real> stl_vector;
    typedef std::vector<stl_vector > stl_matrix;
    typedef PetscVec gene_vector;
    typedef PetscMat gene_matrix;
    static void init() { MPI_Barrier(PETSC_COMM_WORLD); }
    static void close() { MPI_Barrier(PETSC_COMM_WORLD); }
    static inline std::string name( void ) {return "PetSc"; }

    static void free_matrix(gene_matrix & A, int N){
        int ierr;
        if (A.object) ierr = MatDestroy(A.object);
        A.object = NULL;
    }

    static void free_vector(gene_vector & B){
        int ierr;
        if (B.object) ierr = VecDestroy(B.object);
        B.object = NULL;
    }
}

```

```

static inline void matrix_from_stl(gene_matrix & A, stl_matrix & A_stl){
    int ierr, N = A_stl.size();
    if (A.object) MatDestroy(A.object);
    ierr =
MatCreateMPI Dense(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,N,N,PETSC_NULL
,&A.object);
    if (rank == 0) {
        std::vector<int> indi(N);
        std::vector<int> indj(N);
        for (int i=0; i<N; i++) indi[i] = indj[i] = i;
        for (int i=0; i<N; ++i) {
for (int j=0; j<N; ++j) {
            int ii = i, jj = j;
            PetscScalar x = A_stl[i][j];
            ierr =
MatSetValues(A.object,1,&i,1,&j,&x,INSERT_VALUES);
        }
    }
    MatAssemblyBegin(A.object,MAT_FINAL_ASSEMBLY);
    MatAssemblyEnd(A.object,MAT_FINAL_ASSEMBLY);
    return;
}

static inline void vector_from_stl(gene_vector & B, stl_vector & B_stl){
    int ierr;
    if (B.object) VecDestroy(B.object);
    ierr =
VecCreateMPI(PETSC_COMM_WORLD,PETSC_DECIDE,B_stl.size(),&B.object);
    ierr = VecSetFromOptions(B.object);///CHKERRQ(ierr);
    if (rank == 0) {
        std::vector<int> ind(B_stl.size());
        for (int i=0; i<B_stl.size(); ++i) ind[i] = i;
        ierr =
VecSetValues(B.object,B_stl.size(),&ind[0],&B_stl[0],INSERT_VALUES);
    }
    VecAssemblyBegin(B.object);
    VecAssemblyEnd(B.object);

    return;
}

static inline void vector_to_stl(gene_vector & B, stl_vector & B_stl){
    int size;
    VecGetSize(B.object,&size);

    Vec x;
    VecScatter scatter;
    IS from,to;
    PetscScalar *values;
    int *idx_from, *idx_to;
    idx_from = new int[size];
    idx_to = new int[size];
    for (int i=0; i<size; ++i) idx_from[i] = idx_to[i] = i;
    MPI_Barrier(PETSC_COMM_WORLD);
    VecCreateSeq(PETSC_COMM_SELF, size, &x);
    ISCreateGeneral(PETSC_COMM_SELF, size, idx_from, &from);
    ISCreateGeneral(PETSC_COMM_SELF, size, idx_to, &to);
    VecScatterCreate(B.object, from, x, to, &scatter);
    VecScatterBegin(B.object, x, INSERT_VALUES, SCATTER_FORWARD, scatter);
    VecScatterEnd(B.object, x, INSERT_VALUES, SCATTER_FORWARD, scatter);
    VecGetArray(x, &values);
    ISDestroy(from);
    ISDestroy(to);
    VecScatterDestroy(scatter);
    MPI_Barrier(PETSC_COMM_WORLD);
    if (rank == 0) {

```

```

        for (int i=0;i<size;++i) B_stl[i] = values[i];
        delete [] idx_from;
        delete [] idx_to;
    }
}

static inline void matrix_to_stl(gene_matrix & A, stl_matrix & A_stl){
    int low, high, ncols, *cols;
    PetscScalar *values;
    MPI_Barrier(PETSC_COMM_WORLD);
    MatGetOwnershipRange(A.object,&low,&high);
    for (int i=low;i<high;++i) {
        MatGetRow(A.object,i,&ncols,&cols,&values);
        for (int j=0;j<A_stl.size();++j) {
            if (rank == 0) INFOS("starting " << "i = " << i << " j = " << j);
            A_stl[i][j] = values[j]; // this only works for dense
Matrix
        }
        MatRestoreRow(A.object,i,&ncols,&cols,&values);
    }
    MPI_Barrier(PETSC_COMM_WORLD);
}

static inline void copy_matrix(const gene_matrix & source, gene_matrix &
cible, int N) {
    MatCopy(source.object,cible.object,SAME_NONZERO_PATTERN);
}

static inline void copy_vector(const gene_vector & source, gene_vector &
cible, int N) {
    VecCopy(source.object,cible.object);
}

static inline void matrix_vector_product(gene_matrix & A, gene_vector &
B, gene_vector & X, int N) {
    MatMult(A.object,B.object,X.object);
}

static inline void axpy(const real coef, const gene_vector & X,
gene_vector & Y, int N) {
    VecAXPY(&coef,X.object,Y.object);
}
};

```

## 10 Appendix 3

Implementation of the BTL interface for C++ STL library using algorithms for computations.

```

template<class real>
class STL_algo_interface{
public :
    typedef real real_type ;
    typedef std::vector<real> stl_vector;
    typedef std::vector<stl_vector > stl_matrix;
    typedef stl_matrix gene_matrix;
    typedef stl_vector gene_vector;

    static void free_matrix(gene_matrix & A, int N){}
    static void free_vector(gene_vector & B){}

    static inline void matrix_from_stl(gene_matrix & A, stl_matrix & A_stl){
        A=A_stl ;
    }
};

```

```

}

static inline void vector_from_stl(gene_vector & B, stl_vector & B_stl){
    B=B_stl ;
}

static inline void vector_to_stl(gene_vector & B, stl_vector & B_stl){
    B_stl=B ;
}

static inline void matrix_to_stl(gene_matrix & A, stl_matrix & A_stl){
    A_stl=A ;
}

static inline void copy_vector(const gene_vector & source, gene_vector &
cible, int N) {
    for (int i=0;i<N;i++) cible[i]=source[i];
}

static inline void copy_matrix(const gene_matrix & source, gene_matrix &
cible, int N) {
    for (int i=0;i<N;i++)
        for (int j=0;j<N;j++)
cible[i][j]=source[i][j];
}

class somme {
public:
    somme(real coef):_coef(coef){};
    real operator()(const real & val1, const real & val2) { return
_coef*val1+val2; }
private:
    real _coef;
};

class vector_generator {
public:
    vector_generator(const gene_matrix & a_matrix, const gene_vector &
a_vector):
        _matrice(a_matrix),
        _vecteur(a_vector),
        _index(0)
    {};
    real operator()( void ) {
        const gene_vector & ai=_matrice[_index];
        int N=ai.size();
        _index++;
        return std::inner_product(&ai[0],&ai[N],&_vecteur[0],0.0);
    }
private:
    int _index;
    const gene_matrix & _matrice;
    const gene_vector & _vecteur;
};

static inline void matrix_vector_product(const gene_matrix & A, const
gene_vector & B, gene_vector & X, int N) {
    std::generate(&X[0],&X[N],vector_generator(A,B));
}

static inline void axpy(real coef, const gene_vector & X, gene_vector &
Y, int N) {
std::transform(&X[0],&X[N],&Y[0],&Y[0],somme(coef));
}
};

```