

Analysis of Object-Oriented Numerical Libraries

Kostas Zotos

Dept. of Applied Informatics,
University of Macedonia,
54006 Thessaloniki, GREECE

George Stephanides

Dept. of Applied Informatics,
University of Macedonia,
54006 Thessaloniki, GREECE

***Abstract** - In this paper, we examine numerical efficiency of selected Object-Oriented numerical libraries which can be obtained freely on the internet and are open source. This work was largely based on the interesting results reported by the BTL(Benchmark for Templated Libraries) framework on serial numeric linear algebra libraries.*

***Keywords:** Numerical libraries efficiency; Object-Oriented libraries.*

1 Introduction

Today 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. The use of C++ for numerical linear algebra has been slow due to the difficulty in obtaining computational efficiency. However, in recent years various Object Oriented(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. In this paper, we report the results of a comparative study of existing OO libraries in C++ performing a selected number of LA operations.

2 Basics of C++ programming language

C++ was created as a bridge between Object-Oriented programming and C, the world's most popular programming language for commercial software development. The goal was to provide Object-Oriented design to a fast, commercial software development platform. C was developed as a middle ground between high-level business applications languages such as COBOL and the pedal-to-the-metal, high-performance, but difficult-to-use Assembler language. C was to enforce "structured" programming, in which problems were "decomposed" into smaller units of repeatable activities called procedures.

C++ programming language was initially created by Bjarne Stroustrup and it was an attempt to extend and change the C programming language to overcome some of the common and big problems with the language. C++ was first standardized in 1998. C++ is still the dominant OO language in scientific computing, despite its complexity[3].

3 Object-Oriented design

The programs we're writing are far more complex than those written at the beginning of the decade. Programs created in procedural languages tend to be difficult to manage, hard to maintain, and impossible to extend. Graphical user interfaces, the Internet, digital telephony, and a host of new technologies have dramatically increased the complexity of our projects at the very same time that consumer expectations for the quality of the user interface are rising. In the face of this increasing complexity, developers took a long hard look at the state of the industry. What they found was disheartening, at best. Software was late, broken, defective, bug ridden, unreliable, and expensive. Projects routinely ran over budget and were delivered late to market. The cost of maintaining and building on these projects was prohibitive, and a tremendous amount of money was being wasted.

Object-Oriented software development offers a path out of the abyss. Object-Oriented programming languages build a strong link between the data structures and the methods that manipulate that data. More important, in Object-Oriented programming, you no longer think about data structures and manipulating functions; you think instead about objects[2].

The world is populated by things: cars, dogs, trees, clouds, flowers. Things. Each thing has characteristics (fast, friendly, brown, puffy, pretty). Most things have behaviour (move, bark, grow, rain, wilt). We don't think about a dog's data and how we might manipulate it — we think about a dog as a thing in the world, what it is like and what it does[3].

4 BTL benchmark methodology

In this paper, we report the initial results of a comparative study of existing serial OO libraries performing a selected number of Linear Algebra operations. This work was largely based on the interesting results reported by the BTL(Benchmark for Templated Libraries) framework on serial numeric linear algebra libraries. Using the BTL[1] we are reported benchmarks for several open source serial OO libraries, such as Blitz++ [4], MTL[5], ublas[6], and tvmet[7], for some basic operations. The results for template libraries were compared with more traditional, non-OO libraries, such as BLAS, ATLAS, as well as with raw native C, C++ (STL).

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. 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 are open source. We used the documentation available at the appropriate sites to install and test the software.

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[8,9] 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: $x = M * u$, where M is a matrix and j and u 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: $t = M * u$; $x = t$, 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 naïve 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 or temporary object creation is unacceptable because of the time required to locate 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 [8].

5 Results

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 code. For example, the 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 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. We can observe that the performance of the tested packages varied by a factor of 4. The slowest at ≈ 100 MFLOPS and the fastest at ≈ 400 MFLOPS.

The performance of ATLAS and GOTO are clearly superior for 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 BLAS while MTL ranked last. However, the BLAS package available from Netlib is general and it not tuned for system specific features.

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[10] 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, producing highly efficient code.

[7] <http://tvmet.sourceforge.net/>

[8] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, Reading, MA, 2000

[9] Todd L. Veldhuizen. Techniques for scientific computing C++. technical report #542. Technical report, Indiana University Computer Science, 2000.

[10] <http://www.boost.org/>

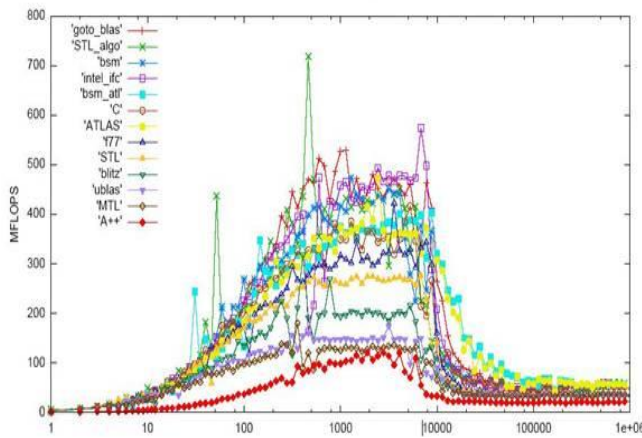


Fig.1 Rates from BTL using different libraries for AXPY operation

6 Conclusions

Object Oriented packages still lags behind well-tuned non-Object Oriented libraries. Based on our results, it is clear that serial non-OO packages are more mature than the serial OO packages. Surprisingly, simple containers from the Standard Template Library (STL) outperformed most of Object Oriented libraries that used more sophisticated techniques such as expression templates. Careful wrapping of non-Object Oriented libraries seems to be the best way to obtain good performance and reusability. However, further research is required in order to examine some other techniques such as parallel OO libraries.

7 References

[1] <http://projects.openscade.org/btl>

[2] <http://www.zib.de/Visual/people/mueller/Course/Tutorial/tutorial.html>

[3] <http://java.sun.com/docs/books/tutorial/java/concepts/>

[4] <http://www.oonumerics.org/blitz>

[5] <http://osl.iu.edu/research/mtl>

[6] <http://www.genesys-e.org/ublas>