Mathematical Modelling and Analysis. Ed. A.Buikis
Volume 12 Number 1, 2007, pages 71–79
© 2007 Technika ISSN 1392-6292 print, ISSN 1648-3510 online

APPLICATION OF TEMPLATE METAPROGRAMMING TECHNOLOGIES TO IMPROVE THE EFFICIENCY OF PARALLEL ARRAYS

A. JAKUŠEV

Vilnius Gediminas Technical University
Saulėtekio al. 11, LT-10223 Vilnius, Lithuania
E-mail: alexj@fm.vtu.lt

Received December 28, 2006; revised January 22, 2007; published online February 10, 2007

Abstract. Parallel array library ParSol is an easy way to parallelize data parallel algorithms implemented in C/C++. However, in order to use all the features provided by C++ and OOP in real life applications, the efficiency of C++ code that uses ParSol library must be similar to the one of C code. Template metaprogramming is one of the ways to achieve this goal. This paper describes the details of application of this technology to parallel arrays, and presents the efficiency tests.

Key words: ParSol, OOP, C/C++, template metaprogramming, expression templates

1. Introduction

ParSol is a parallel array library which brings to C++ programs functionality similar to the one that HPF brings to FORTRAN programs [4]. This allows for quick parallelization of data parallel algorithms. ParSol uses MPI standard [5] for interprocess communications, which makes it highly portable to various platforms. Initially designed with the purpose of parallelizing of flow in porous media problem solver [3], ParSol has developed into full featured library which is also successfully applied to such applications as image smoothing [1, 2], Schrodinger equation and others [6].

To simplify the parallelization of data parallel algorithms, the ParSol library introduces special array classes, which library user should use instead of native C/C++ arrays. In sequential version, the behaviour of these arrays is similar to any other arrays. However, in parallel case, the parallel versions

of these arrays are able to distribute contained data between processes and perform data exchange.

The next step in improving ParSol library is to increase the efficiency of its array classes. In this article, the results of applying of the *template metaprogramming* technology for the improvement of ParSol arrays efficiency are presented.

The template mechanism of C++ is very powerful due to such features of template programming as *specialization* (also partial specialization), *recursive instantiation* and others. It was shown that template programming of C++ is Turing complete [9], making it possible to solve various problems, such as computation of roots or prime number generation, during compile time. The name *metaprogramming* is better suited here due to the fact that by using templates a programmer writes a program that generates another program as a result.

By using the template metaprogramming, the following issues in ParSol arrays were addressed:

Efficient array expressions. Using the operator overloading, it is possible to write a program code that looks close to mathematical notation of various operations. It is possible to overload "+" operator in such a way that adding arrays will look the same as adding numbers:

```
Vector A(20), B(20), C(20);
...
A = B + C;
...
```

However, standard ways to implement such an overloading, while achieve the goal, are not computationally effective. In the example given above, if implemented in a standard way, the line `A = B + C;` would be equal to something like that:

```
{ Vector Temporary(B);
  Temporary += C;
  A = Temporary; }
```

Note the allocation and deallocation of a temporary array and several assignments, which means that a cycle through all array elements needs to be executed several times. In the more complex cases, such as `A = B + C + D + E;`, the inefficiency will be even more obvious, especially on large arrays. This problem was noticed by the creators of C++ language [8, §11.6].

In order to resolve this problem, *expression templates*, as one of the template metaprogramming techniques [11], were successfully applied. This paper presents the detailed description of the problems that had to be overcome, together with the results of efficiency tests.

Efficient code generalization. ParSol provides a hierarchy of various array and vector classes, and many of them have a great amount of similar functionality. From the one hand, this allows many classes to share common

code base, which would make code simpler and easier to maintain. But general algorithms, suitable for various cases, are less effective than its specific counterparts. Thus we have to choose between a better code structure and a faster performance.

By using the template metaprogramming, it became possible to optimize a general code for multidimensional cases. This paper describes a few cases for the ParSol arrays.

2. Application of Template Metaprogramming

2.1. ParSol class structure

The ParSol arrays class hierarchy is shown in Figure 1. The general idea is that common functionality must reside in base classes, and children classes must provide an easier user interface and specific optimization algorithms. Generally, different children are derived for every number of dimensions. For example, a general array functionality resides in the template class `CmArray<class,int>`, and there are specific children for 2D, 3D and other cases.

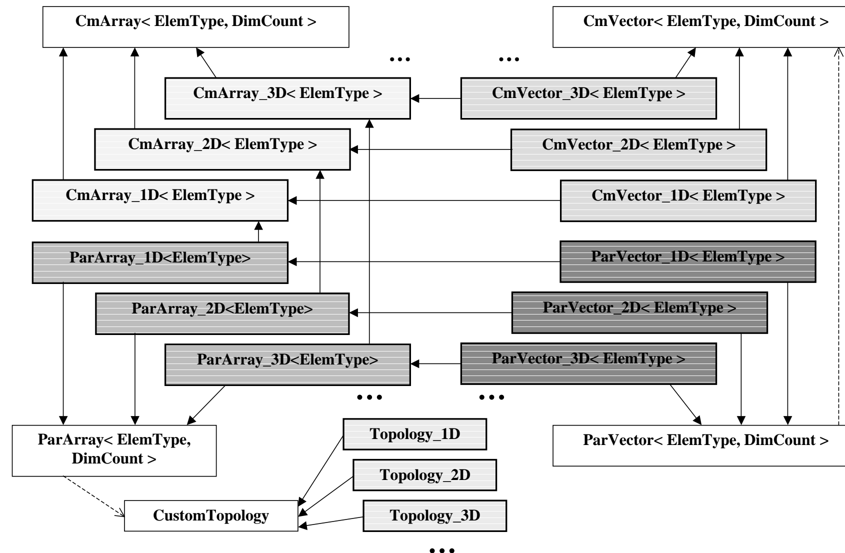

Figure 1. ParSol class diagram.

The part of ParSol library that has been optimized consists of arrays and vectors, both parallel and sequential versions. All of the classes are template classes, where base classes usually have two template parameters, array element type and number of dimensions, and children classes for every number

of dimensions have one template parameter, data type. The structure of *sequential array* classes was described above. *Sequential vectors* are basically the arrays with additional mathematical functions provided, thus they are descendants of appropriate sequential arrays and class `CmVector`, which contains all the additional mathematical functionality. *Parallel arrays*, which are used in parallel versions of user programs, are also children of appropriate sequential arrays, with common parallel functionality inherited additionally from `ParArray` class. And, finally, *parallel vectors*, similar to sequential ones, are children of parallel array classes, with mathematical functionality inherited from `ParVector`.

2.2. Expression templates

With the help of the expression templates, the following goals were achieved:

- Fast and efficient vector expressions without memory and processor time overhead;
- Possibility to mix vectors and scalars in expressions, for example `A=(B+C)/2;`
- Compile-time vector type compatibility check, so that it is impossible to mix vectors with different element types or number of dimensions in expressions.

Originally, expression templates (ET) were designed to be used with non-template vectors [11]. However, without modifications, this technology would not allow to perform strict array type checking due to its template nature. Moreover, initially ET were formed using array iterators, in order to have generalised code, as in STL. In ParSol, it was chosen to pass references to arrays themselves, the approach is also taken in [7]. While this approach makes it very difficult to reuse ET implementation outside ParSol library, it allows us for an additional code optimization and run-time checks.

Since expressions, as other mathematical operations, should be available for vectors only, the core ET functionality is programmed around the `CmVector` class. ET are implemented mainly by two kinds of helper classes: `OpWrapper` and *operation classes*.

`OpWrapper` is a very simple class that behaves as a template container for various data types and provides access to contained data via two main functions:

elem — access to the array elements of expression it contains;
SamePhysically — required to check if data it contains is compatible with the vector where result is to be put.

This class is a glue to the whole ET mechanism. All vector operators return the answer of type `OpWrapper` (the data it contains may differ, though), and a value of this type is expected for assignment to vector. This allows `C++` compiler to make correct template instantiation decisions, as well as restricts the number of data types which are allowed to participate in expressions.

There are several implementations of this class, using a template partial specialization. The main difference among them is the way how they store their data. The wrapper for vectors holds a pointer to its vector, which greatly reduces amount of data that needs to be copied. Since no temporary vectors participate in expressions, such behaviour is correct. Other wrappers hold copies of their data. Since their data is usually some kind of temporary wrapper, such behaviour is necessary. The overhead produced by copying may be neglected since, if only vectors and numerical constants participate in expressions, the size of temporary wrappers is small.

The *operation classes* are also template containers, but they contain not only some data, but also an information about what should be done with them. Currently, there are two kinds of operations: one-operand (`OpArr1`) and two-operand (`OpArr2`). Operation classes provide the same interface as `OpWrapper`, however, it gives the access to the result of operation with the contained data.

For example, the definition of operator for adding two vectors looks as follows (it is a part of `CmVector` class definition):

```
...
/// Helper type to make some expressions smaller
typedef Internal::OpWrapper<
    ElemType,DimCount,CmArray<ElemType,DimCount>
  > ArrWrapper;

Internal::OpWrapper<
  ElemType,
  DimCount,
  Internal::OpArr2<
    ArrWrapper, ArrWrapper, ElemType, DimCount,
    Internal::et_summ
  >
> operator+( const CmVector<ElemType,DimCount>& v )
...
```

Plus operator actually constructs and returns new data type, which is an `OpWrapper` of 2-operand operation class `OpArr2`, which in turn contains `OpWrapper`'s of two vectors and operation to be performed on them (`et_summ` in this case).

Finally, the "=" template operator for `OpWrapper<...>` needs to be implemented in vector classes. The structure of `C++` language requires that assignment operators may not be inherited, thus this is the only part of ET implementation that needs to be copied in every descendant of `CmVector`. However, assignment operator in children consists of only one-line invocation of general `OpWrapper` assignment code provided by `CmVector`.

ParSol ET implementation provides strong both compile and run time checking of vectors in expressions. First of all, all template operators and main ET helper classes have additional template parameters of type `class` (ElemType) and `int` (DimCount). While adding very little additional functionality,

these parameters ensure that only expressions of vectors with the same element type and number of dimensions will match and therefore compile. That granted, the default assignment operator in `CmVector` performs one-time check of other parameters of vectors in expression. This check consists of comparison of several integer values and does not affect the performance, but ensures, for example, that vectors of different sizes won't be accidentally added.

The implementation of ET for parallel vectors is similar to the sequential version. As before, most functionality is implemented around `ParVector` class, with only assignment operator being reimplemented in `ParVector`'s children. ET for parallel vectors uses codebase from sequential implementation where possible.

2.3. General code optimizations

A template metaprogramming was also used to optimize the code in general ancestors such as `CmArray`, `CmVector` and others. The main type of optimization performed was a loop unrolling [10]. The loop unrolling is especially useful for loops with small number of iterations. The problem is that compilers usually optimize loops for many iterations, which is counterproductive in this case. These kinds of loops are common in general code, where number of iterations depends on the number of array dimensions. This technique is employed for the same purposes in such libraries as Blitz++, MTL and FreePOOMA.

The key to the loop unrolling is template classes with some integer template parameter. If this class uses the same class, only with a different value of that template parameter (usually one less than the original), as a part of its declaration, then instantiation of such class will trigger recursive instantiation of the same class with different integer parameters, possibly producing loop unrolling effect. Examples of such code may be found in [10, §17.7].

In ParSol, there are several such classes, which are in internal name-space and therefore not accessible by default. They are just holders for unrolling of usually unrelated various loops.

3. Numerical Tests

In order to evaluate the efficiency of expression templates, computational tests were performed. The tests were performed on a computer with AMD Athlon 64 3200+ processor and 512 MB of RAM. Operating system was 32-bit GNU/Linux with kernel version 2.6.8-12-amd64-k8. The tests were compiled using `g++` version 3.3.5, with maximum (`-O3`) optimization.

Since ParSol arrays have only ET implementation of binary operations, the comparison of ET to classical method was tested on specially created template arrays, which had various kinds of children, implementing different techniques of addition. The results of the test are presented in Figure 2. In the test, we have addition of three vectors of different sizes of double. We may see that ET realization is from 2.5 to 4 times faster than a classical realization,

and its efficiency is almost the same as C-style realization (their graphs closely overlap).

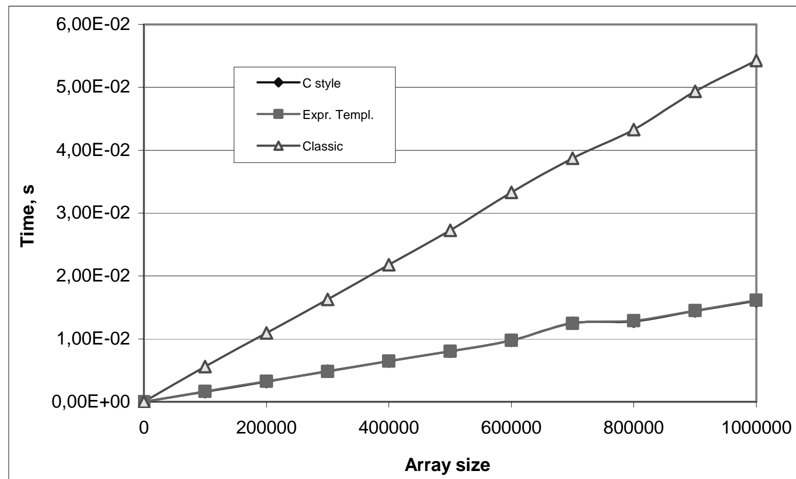

Figure 2. Comparison of the efficiency of ET to other types of array addition.

In another experiment, the performance of ParSol vector additions was tested. The same vectors were added either by using ET (`A=B+C+...;`) or using *C-style* `for` loop:

```
for( int i = 0; i < VecSize; i++ )
  A(i)=B(i)+C(i)+...;
```

Vectors of type `CmVector_2D` were used, and the element data type was `double`. The tests were performed with varying vector sizes and number of added vectors.

During the experiments, computation time of operations was measured. The time spent on computations was measured using *user time* returned by `times` function. Each test was repeated 10000 times, and average computation time was calculated.

The results of the tests are shown in Figure 3. The computation time dependency on the number of added vectors is presented for a number of different vector sizes and types of addition. While the increase of vector size makes clear impact on the computation time, the difference between C-style and ET programming is often negligible. In the worst case, the ET performance is decreased 9.5% comparing to C-style programming, and average decrease of performance was $\approx 2.6\%$.

Assuming that performance ratio between C-style and classic addition implementation would be approximately the same here as in previous experiment, it is possible to state that ET implementation of binary operations in ParSol is $\approx 2.5 - 4$ times faster than if it were implemented in a classical way. These results are comparable to [11].

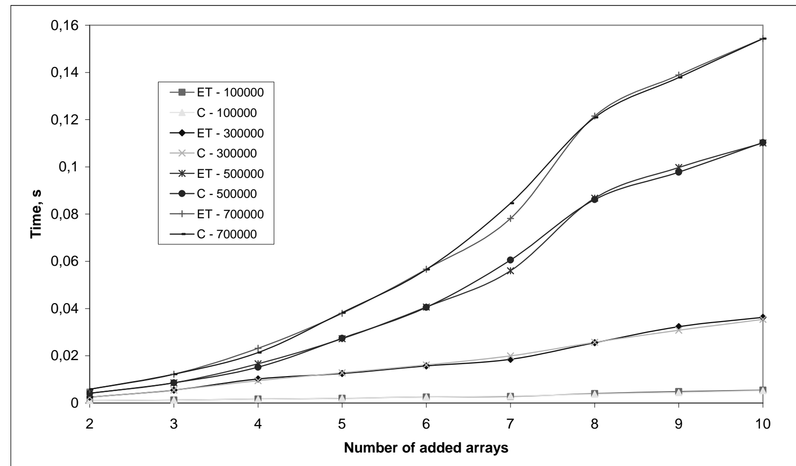

Figure 3. Comparison of C-style and ET performance of ParSol vectors.

While template metaprogramming allows to achieve high computational efficiency, it brings some problems as well. One of the problems is much longer program compilation time. This may slow down the development of the product, especially if compiler does not support precompiled headers. The size of resulting executables is also considerably bigger in case of template metaprogramming, however, with modern storage capacities, this is not an issue in most cases.

Another problem is that template metaprogramming, although C++ standard compliant, requires such compiler features that may not be supported or thoroughly tested on all compilers. For example, on Ubuntu Linux system with g++ version 4.0.2 (prerelease), the compiler was unable to produce working executable for the same test suite, when compiled with maximal optimization. However, with the rapid improvement of C/C++ compilers, this issue will become less important with time.

4. Conclusions

Template metaprogramming technology is not trivial to implement, however it may provide performance to C++ programs comparable to C or FORTRAN languages. That's why it is ideal to use in libraries, so that library user may use highly optimized code without much of additional knowledge. The existence of such numerical packages as Blitz++, MTL or FreePOOMA proves this fact.

Array and vector classes of ParSol library were successfully optimized using such template metaprogramming techniques as expression templates, loop unrolling and others. Initial ET technology had to be modified to fit into the structure of ParSol template classes. This resulted in the efficiency of ParSol vector operators that is only $\approx 2.6\%$ lower than its C-style hand coded equivalents.

While template metaprogramming provides a significant improvement of run-time performance, it also results in bigger size of executables, longer compilation time and compatibility problems with some compilers. However, these problems should lose their importance in the future.

Acknowledgement

This work was also supported by the Lithuanian State Science and Studies Foundation within the framework of the Eureka Project EUREKA E!3691 OPTCABLES.

References

[1] R. Čiegis and A. Jakušev. Parallel algorithms in image filtering. *Lithuanian Mathematical Journal*, **45**, 411–416, 2005. (in Lithuanian)

[2] R. Čiegis, A. Jakušev, A. Krylovas and O. Suboč. Parallel algorithms for solution of nonlinear diffusion problems in image smoothing. *Mathematical Modelling and Analysis*, **10**(2), 155–172, 2005.

[3] R. Čiegis, A. Jakušev and V. Starikovičius. Parallel tool for solution of multiphase flow problems. *Lecture Notes in Computer Science, PPAM-2005 Revised Selected Papers*, 312–319, 2006. Springer

[4] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Electronic version available on Internet, 1997. (Version 2.0)

[5] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard.* Electronic version available on Internet, 1995. (Version 1.1)

[6] A. Jakušev and V. Starikovičius. Application of parallel arrays for parallelization of data parallel algorithms. *Computer Aided Methods in Optimal Design and Operations, Series on Computers and Operations research*, **7**, 109–118, 2006. Springer

[7] C. Pflaum. Expression templates for partial differential equations. *Computing and Visualization in Science*, **4**(1), 1–8, November 2001. Springer-Verlag

[8] B. Stroustrup. *The C++ Programming Language, 3-rd ed.* Addison-Wesley, 1997.

[9] E. Unruh. Prime number computation. *ANSI X3J16-94-0075/ISO WG21-462*, 1994.

[10] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide.* Addison-Wesley, 2002.

[11] Todd L. Veldhuizen. Expression templates. *C++ Report*, **7**(5), 26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman