LA-UR--97-3888

CONF-971219--

A Comparison of Performance-Enhancing Strategies for Parallel Numerical Object-Oriented Frameworks

Federico Bassetti, Kei Davis, and Dan Quinlan

Scientific Computing Group CIC-19
Computing, Information, and Communications Division
Los Alamos NM, USA, 87545
{fede,kei,dquinlan}@lanl.gov

Abstract. Performance short of that of C or FORTRAN 77 is a significant obstacle to general acceptance of object-oriented C++ frameworks in high-performance parallel scientific computing, nonetheless, their value in simplifying complex computations is inarguable. Singular data points of good performance for object-oriented libraries/frameworks have been interesting, but a systematic analysis of the performance issues has not been done. This paper explores just a few of these issues and reports on the use of three mechanisms for enhancing the performance of object-oriented frameworks within numerical computation. The first is the commonly-used of binary overloaded operators (though implemented with substantial internal optimizations), the second is the use of expression templates, and the third is the use of an optimizing preprocessor. The first two have been completely implemented and are available within the A++/P++ array class library[1], the third, ROSE++[2], represents work in progress. This paper provides some perspective on the types of optimizations that we consider important within our own numerical applications using OVERTURE[3] involving complex geometry and AMR on parallel architectures.

1 Introduction

The use of object-oriented C++ frameworks has significantly simplified the development of numerous complex serial and parallel scientific applications at Los Alamos National Laboratory (LANL) and elsewhere. Examples from LANL include OVERTURE [BCHQ97] which supports complex geometries, adaptive mesh refinement (AMR), and overlapping grid computations in addition to more basic single rectangular grid computations, and POOMA [WL96] which has an emphasis on support of particle and molecular dynamics. In spite of considerable use of and commitment to these frameworks, concerns about performance

[1] A++/P++ is available from http://www.c3.lanl.gov/cic19/teams/napc/napc.shtml
[2] ROSE++ Web Site: http://www.c3.lanl.gov/ dquinlan/ROSE.html
[3] OVERTURE is available from http://www.c3.lanl.gov/cic19/teams/napc/napc.shtml

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible electronic image products. Images are produced from the best available original document.

are nonetheless a significant issue; performance very close to that of carefully hand-crafted C or FORTRAN 77 with message passing must be realized before the acceptance and use of such frameworks will be truly widespread. Our express long-term goal is to characterize those aspects of the use of C++ that give poorer performance than C and to identify or provide mechanisms to minimize or eliminate the performance penalties; we believe this goal to be fully realizable.

The potential performance of three execution models for implementing such frameworks is explored: implementation with *overloaded binary operators*, implementation with *expression templates (ETs)*, and *semantics-based source-to-source transformation* (a new approach) with a special-purpose preprocessor. Naive implementation of any of these three techniques will suffer putative disadvantages, however in the context of our comparison we have sought to focus on the practical limitations of each.

For numerical applications the array types may be regarded as the most basic and the performance of their implementation will have a fundamental impact on the performance of libraries and applications built on them. Our chosen target for optimization is A++/P++, the sophisticated 'array class' component of OVERTURE, which is the most complete parallel distributed dynamic array class of which we are aware, incorporating normal array operations, indirect addressing, and block 'where' statements. This paper reports our experiences with A++/P++ using each of the three techniques. A++/P++ has been implemented both using overloaded binary operators and ETs. In some cases of comparison the data for ET implementation is from hand-written 'idealized' output from ET expansion; we have verified this to be fair for determining an upper bound on ET performance while significantly simplifying the process of gathering performance data. Similarly, since the preprocessor is still in an early stage of development we compare directly to C code that the current preprocessor can or can be expected to produce; however, we make no specific claims about the preprocessor but instead concentrate on those issues to be addressed in achieving C performance from C++ applications using array-like constructs.

Relative to each technique two optimizations are evaluated, in turn intimately related to the two highest levels of the memory hierarchy: avoidance of register spillage and efficient use of cache. Other issues are scheduled for future work.

2 Execution Models

The reader is referred to the literature for in-depth description of the use of overloaded binary operators and ETs [Vel96,LQ92,PQ94,PQ93]. The third technique that we are actively developing is semantics-based source-to-source transformation. This technique seeks to perform optimizations much the same as does the use of ETs without their concomitant defects, as well optimizations such as loop fusion and rescheduling and aggregating communications which the authors suspect are not addressable using ETs. This technique is *semantics based* because its implementation is 'wired' or parameterized with exact knowledge of the semantics of the classes over which it optimizes; it is *source-to-source* because both

its input and output are C++ code. This built-in knowledge of class semantics makes possible optimizations that could not in general be performed automatically by a compiler. Further detail may be found elsewhere [QD97].

3 Issues in the Optimization of Numerical Applications

For lack of space we address only two of the many issues in optimizing numerical applications in this context: minimization of register spillage and loop fusion. Other issues are under active investigation.

3.1 Register Spillage

Register spillage refers to the consequence of a code needing more registers than are available, resulting in register values being stored and subsequently reloaded. A high demand for registers may have an impact on performance even if the number of registers required doesn't exceed the number of registers available by preventing the compiler from performing such optimizations as pipelining.

In order to provide accurate and concrete results we show that the performance of a code that makes use of idealized ET code is directly related to the number of registers available, and compare this to C++ code. The data was collected on an SGI Origin 2000 system using the hardware performance counters of the MIPS R10000 microprocessor. Results using the KAI KCC compiler are presented; the SGI C++ compiler was used to verify the results and conclusions.

Test Code. Two versions of the test code implement a simple 3-point stencil using multidimensional arrays with subscript computation only along the innermost dimension. The core of the computation is embodied by a loop that traverses the array elements in memory order.

The main differences between the two codes is how the arrays are accessed. In C++ the array on the right-hand side is reused, thus resource requirements are a constant function of the number of operands. However, a code that makes use of ETs will transform the statement in the loop to use a different array pointer for each operand. Moreover, in an ET code each array carries an integer together with the pointer information on how to compute the proper offset (a more clear idea of the transformation may be had by inspecting the intermediate C code generated by a compiler).

Our benchmarking approach consisted of comparing the two different codes when the number of operands is increased, and also when the dimensionality of the arrays is increased. The simplicity of this benchmark enabled us to study the effect of register spillage without obscuring effects.

Register Spillage Measurements and Results. We capture the impact of register spillage by analyzing the number of loads and stores performed by examining assembly code and using hardware performance counters. This measure also reveals when the demand for registers inhibits other optimizations.

Figure 1 shows that the execution time measured in cycles is significantly different for the two codes. As the number of operands increases the C++ performance stays constant while the ET performance decreases inversely. Secondarily, increasing the dimensionality of the arrays has no effect on the C++ performance but degrades ET performance. In the multidimensional case we verified a 60% degradation in ET performance for six dimensions.

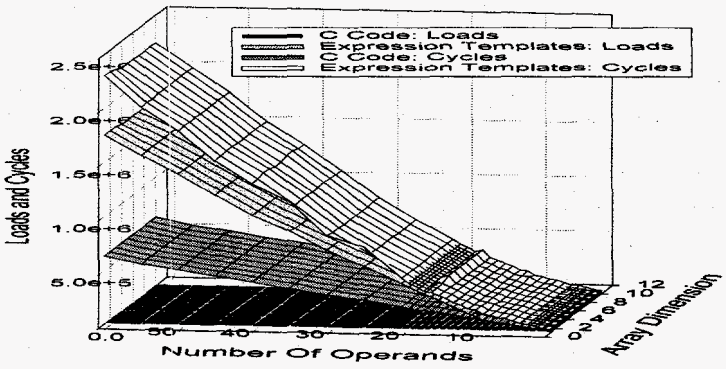

Fig. 1. Register spillage with expression templates

The impact of the number of operands increases dramatically starting at 21-26; the MIPS R10000 has 27 general-purpose integer registers and spilling of integer registers is directly related to the number of operands (there is no spillage of the floating point registers because the compiler schedules floating point operations to be performed pairwise). In the ET code two registers are needed for each array operand—one pointer and one for subscript computation. This is detailed in Table 1. For up to four operands ET performance is close to that of C++. Between five and thirteen operands loop unrolling is disabled because of demand for registers. Between fourteen and twenty-six pipelining is also turned off for the same reason. Beyond that register spillage exacts a further penalty.

Note that the figure shows a large number of operands such as appear in 3D codes using geometry, where stencils involve connecting cross derivative terms and variable coefficents; in this case a 3D code could easily use a 27 point stencil and require 27 coefficients, or 54 operands. Note too that the C code shows no register spillage. The effects are purely a result of the number of operands and the factor of degraded performance is about 4-5 for 54 operands.

Additional data not presented here demonstrates the dependence on the dimensionality of the arrays on the number of cycles. The difference is due to the KAI C++ compiler and the way it lifts loop invariant code from the inner most loop to only the next outer loops. The performance degradation is only about 60% for a 6D array, but 6D arrays, and higher, form an important part of the 3D applications codes that handle geometry (such as within OVERTURE).

number of operands	fixed point registers used	floating point registers used	software pipelining	iterations unrolled
1	15	1	ON	4
2	12	2	ON	2
3	16	4	ON	2
4	12	3	ON	0
5	14	3	ON	0
6	16	4	ON	0
7	18	3	ON	0
8	20	4	ON	0
9	22	4	ON	0
10	24	4	ON	0
11	24	4	ON	0
12	27	5	ON	0
13	27	3	ON	0
14			OFF	0

Table 1. Impact of register demand on pipelining and unrolling

3.2 Loop fusion as an optimization mechanism

Figure 2 shows the timings of using loop fusion on a range of 1-12 array statements parameterized by the number of operands reused within the statements. Where reuse is high the loop fusion results in a factor of two performance gain. In all cases the array statements were written as explicit loops in C to remove any effects of the array class overhead. So that the scales of the data in Fig. 2 could be properly represented we have not shown that when 13 and 14 statements are fused the time increases by an order of magnitude or more because of register spillage.

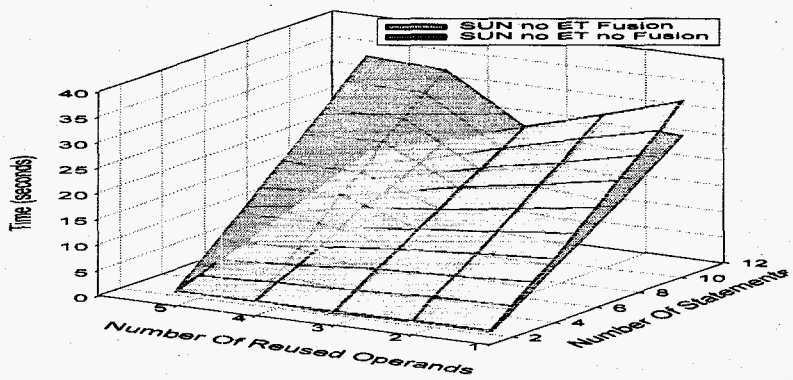

Fig. 2. Loop fusion

Separate tests of the potential for loop fusion within AMR using red-black relaxation as implemented within an array class have shown factors of 4-6 in improved performance. Similar tests on interpolation operators implemented in the array class yields performance increases of 3-4 through loop fusion. From this data we expect that a more accurate characterization of the potential for loop

fusion is possible which would show improved results over the graphs presented here.

Similar data for ETs is also available but not presented here since the rather poor results can be inferred from the previous graphs that characterize the register spillage problems with ETs; loop fusion amplifies this problem by introducing even more operands into the inner loop body.

Loop fusion is an important optimization and one which we can not rely on any combination of C++ compiler or ET technology to provide. This is fundamentally because of the way that the ET implementation must introduce runtime checking to identify loop dependence so that parallelism (and general array semantics) can be supported. The effect is the introduction of conditionals between statements (along with hundreds of lines of other ET code, parallel message passing, etc.) which would negate compiler-driven loop fusion. We believe that this important optimization is largely a justification for research on preprocessor type mechanisms.

3.3 Stencil and Non-Stencil Applications

A++ presently implements both binary operators and ETs and in doing so it provides a simple mechanism to evaluate the performance of the two mechanisms in an unbiased way using the identical application codes. Figure 3 shows the performance of A++ with and without ETs and compares it to C for the 1D shock tube application using the second order PPM Godonov method (the code forms a core of split scheme multidimensional hyperbolic solvers). We don't suggest that this code characterizes all numerical codes, for example it has no stencil operations, but numerical codes are not only composed of stencils. That the two mechanisms give equivalent performance is an unexpected result warranting further investigation.

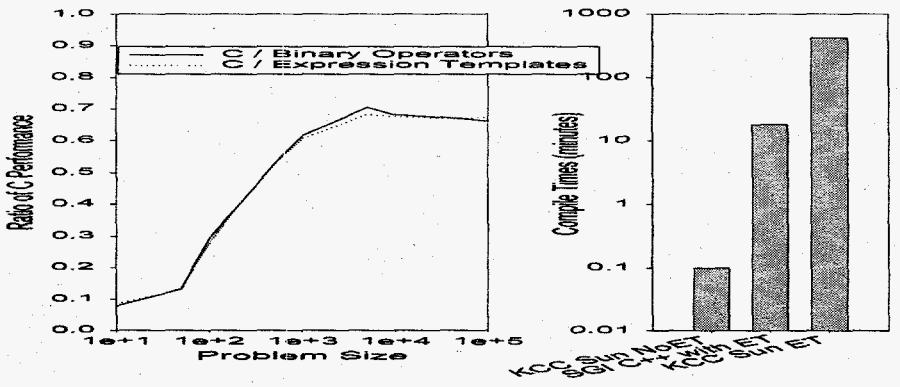

Fig. 3. 1-D shock tube application with compile times

The results shown in Fig. 3 show that performance is not uniform across all numerical codes. For stencil type operations the performance of the ETs is

significantly improved over that of binary operators and in fact close to that of C code. It is known that for stencils additional improvements are possible with blocking mechanisms, but these would not provide to clear comparisions of the data with the unblocked C code. From the near-C performance it is clear that the existing implementation is as efficient as the much-popularized results for ET implementations, yet for a non-stencil based code its performance no better than the binary operator array class implementation. We suspect that the computational costs are similar because the number of loads and cache misses are similar because of the minimal reuse of operands on the right-hand sides of the statements.

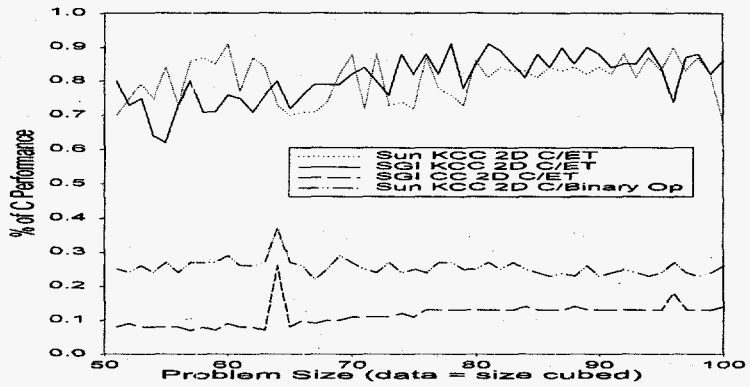

Fig. 4. 2-D stencil

4 Conclusions

The more traditional execution model of overloaded binary operations has performance problems on cache-based architectures with stencil-type expressions; this is less of an issue on other architectures (e.g. vector architectures). However, it appears to make little difference where stencils are not an important part of the computation. The ET implementations have extremely poor compile times (factors of 1000 or more), given the mere factor of 3-4 improvement only for operations involving stencils (which may or may not be a significant portion of the application code). On at least one compiler (the SGI C++ compiler) where the compile times are only a factor of 20-40 from that of the non-ET code the performance is far worse that that for the binary overloaded operators, for example 1/20 of C performance for 3-D arrays.

ETs exhibit additional poor performance for large stencils as such regularly appear within OVERTURE applications with complex geometry, though this would not be so much the case if restricted to single square grids with no geometry where most of the ET experience is based. In contrast the binary overloaded operators generate no similar register spillage.

Where the use of ETs significantly improves the performance of the most problematic array expressions, it can still only be compiled by a small number of C++ compilers with the KAI C++ compiler being the only one which achieves any significant performance. However, the performance of ETs is sensitive to the quality of the underlying C compiler, something that few people have any significant control over. Finally, we feel that the attempts made to link the future of object-oriented scientific computing to ET technology are seriously flawed. With some effort better performance can be obtained with a special-purpose optimizing preprocessor. Existing tools such as SAGE++[BBG97] greatly simplify the development of such preprocessors. The more sophisticated transformations possible with the preprocessor derive primarily from the ability to perform semantics-based program analysis such as dependence analysis and alias analysis.

We suspect that preprocessing is a superior and ultimately more powerful alternative to addressing these problems, but enhancement and refinement of the ET mechanism (possibly in conjuction with additional annotation by the user) may also be possible.

References

[BCHQ97] Brown, D., Chesshire, G., Henshaw, W., and Quinlan, D., OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments, Proceedings of the SIAM Parallel Conference, Minneapolis, MN, March 1997.

[WL96] Wilson, G.V. and Lu, P., *Parallel Programming Using C++*, Ch. 14, pp. 547-587, MIT Press, 1996.

[Vel96] Veldhuizen, T., Expression templates. In *C++ Gems*, S.B. Lippman, ed., Prentice-Hall 1996.

[QD97] Quinlan, D., Davis, K., ROSE: An Optimizing Preprocessor for the Object-Oriented OVERTURE Framework, http://www.c3.lanl.gov/ dquinlan/ROSE.html.

[PQ94] Parsons, R., Quinlan, D., A++/P++ Array Classes for Architecture Independent Finite Difference Computations. Proceedings of the Second Annual Object-Oriented Numerics Conference, Sunriver, Oregon, April 1994.

[PQ93] Parsons, R., Quinlan, D., Run-time Recognition of Task Parallelism within the P++ Parallel Array Class Library, Proceedings of the Conference on Parallel Scalable Libraries, Mississippi State, 1993.

[LQ92] Lemke, M., Quinlan, D., P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications. CONPAR/VAPP V, September 1992, Lyon, France, LNCS, Springer Verlag, September 1992.

[BBG97] Francois, B., et. al., Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools. Proceedings of the Second Annual Object-Oriented Numerics Conference, 1994.