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

Federico Bassetti, Kei Davis, and Dan Quinlan

Los Alamos National Laboratory

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. Examples of good performance for object-oriented libraries/frameworks are interesting, but a systematic analysis of performance issues has not been done. This paper explores a few of these issues and reports on three mechanisms for enhancing the performance of object-oriented frameworks for numerical computation. The first is binary operator overloading implemented with substantial internal optimizations, the second is expression templates, and the third is an optimizing preprocessor. The first two have been completely implemented and are available in the A++/P++ array class library[1], the third, ROSE++[2], is work in progress. This paper provides some perspective on the types of optimizations that we consider important in our numerical applications using OVERTURE[3] involving complex geometry and AMR on parallel architectures.

1 Introduction

The use of object-oriented C++ frameworks has greately simplified the development of complex serial and parallel scientific applications at Los Alamos National Laboratory (LANL) and elsewhere. Examples from LANL include OVERTURE [1] which supports complex geometries, adaptive mesh refinement (AMR), and overlapping grid computations as well as simpler single rectangular grid computations, and POOMA [8] which has an emphasis on support for particle and molecular dynamics. In spite of considerable use of and commitment to these frameworks, concerns about performance are a significant issue; performance very close to that of carefully hand-crafted C or FORTRAN 77 with message passing must be achieved before acceptance and use of such frameworks will be widespread. Our goal is to characterize aspects of C++ usage that give poorer performance than C and to provide mechanisms to minimize or eliminate the performance penalties; we believe this goal to be fully realizable.

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

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 techniques will suffer putative disadvantages, so in the context of our comparison we have focused on the practical limitations of each.

For numerical applications array types are basic, and the performance of their implementation has a fundamental impact on the performance of libraries and applications built on them. Our target for optimization is the parallel distributed dynamic array class A++/P++, a component of OVERTURE, which incorporates normal array operations, indirect addressing, and block "where" statements. We report 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 the data for ETs is from hand-written idealizations of ET expansion; this is fair for determining an upper bound on ET performance while simplifying the gathering of performance data. Similarly, since the preprocessor is still in 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. Instead, we concentrate on issues in achieving C performance from C++ applications using array-like constructs.

For each technique two optimizations are evaluated in terms of performance at the top of the memory hierarchy: avoidance of register spillage and efficient use of cache. Other issues are scheduled for future work.

2 Execution Models

Detailed discussions of operator overloading and ETs may be found in the literature [7, 4, 6, 5]. The third technique that we are developing is semantics-based source-to-source transformation. This technique performs optimizations much like ETs do without their concomitant defects, and also applies optimizations such as loop fusion and rescheduling and aggregating communications. This technique is *semantics based* because its implementation has 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 exists elsewhere [2].

3 Issues in the Optimization of Numerical Applications

We address only two of the many issues in optimizing numerical applications here: minimization of register spillage, and loop fusion. Other issues are under investigation.

3.1 Register Spillage

Register spillage occurs when a program needs more registers than are available, resulting in register values being stored and subsequently reloaded. High register demand may impact performance even in the absence of spillage because it may prevent other optimizations such as pipelining.

To provide accurate and concrete results we show that the performance of a program that makes use of idealized ET code is directly related to the number of registers available; this is compared to code without ETs. Data was collected using the KAI C++ compiler[4] on an SGI Origin 2000 system using the hardware performance counters of the MIPS R10000 microprocessor.

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, so resource requirements are constant in the number of operands. However, a code using ETs will transform the statement into a loop that uses a different array pointer for each operand. Further, 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). We compared the performance of the two codes while independently increasing the number of operands and the dimensionality of the arrays.

Register Spillage Measurements and Results. We capture the impact of register spillage by analyzing the number of loads and stores performed from assembly code and using hardware performance counters. This measure also shows when register demand 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. Secondarily, increasing the dimensionality of the arrays has no effect on the C++ performance but degrades ET performance. In the multidimensional case we see a 60% degradation in ET performance for six dimensions.

The impact of the number of operands increases rapidly 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++.

[4] The SGI C++ compiler was used to verify the results and conclusions.

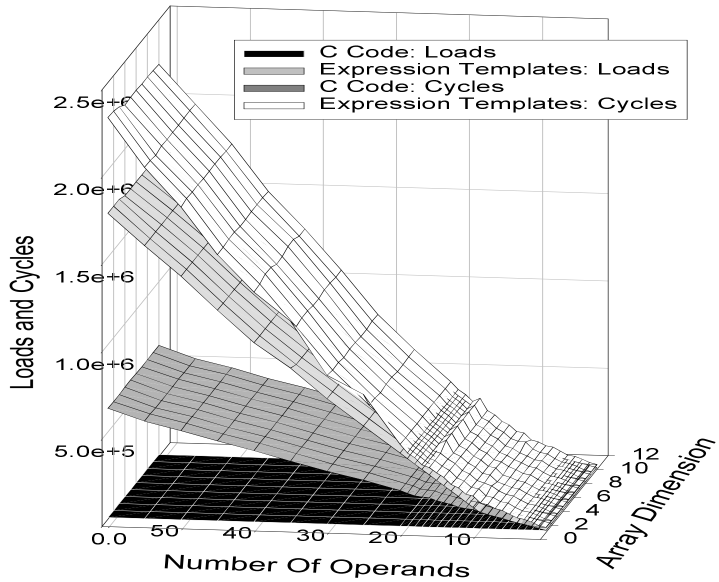

Fig. 1. Register Spillage with Expression Templates

Between five and thirteen operands loop unrolling is inhibited because of register demand. Between 14 and 26 pipelining is inhibited for the same reason. Beyond that register spillage exacts a further penalty.

Table 1. Impact of Register Demand on Pipelining and Unrolling

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

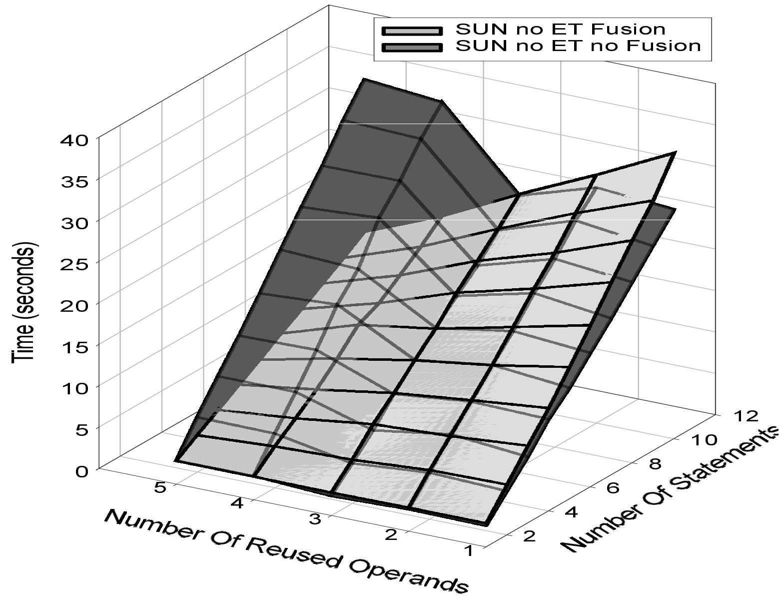

Fig. 2. Loop Fusion

Figure 1 shows the effect of a large number of operands such as appears in 3D codes using geometry where stencils involve connecting cross-derivative terms and variable coefficents. A 3D code could easily use a 27-point stencil and require 27 coefficients, or 54 operands. The C code shows no register spillage, with performance depending only on the number of operands; performance degrades by about 4–5 for 54 operands.

Additional data not presented here shows that the dimensionality of the arrays increases the number of cycles. The difference is a result of the KAI C++ compiler lifting loop-invariant code from an innermost loop to only the next outer loop. The performance degradation is about 60% for 6D arrays, but 6D and higher arrays are important for 3D applications that handle geometry (for example in OVERTURE).

3.2 Loop Fusion as an Optimization Mechanism

Figure 2 shows the effects of loop fusion over a range of 1–12 array statements, parameterized by the number of operands reused within the statements. Where reuse is high loop fusion doubles performance. In C the array statements were written as explicit loops to remove any effects of the array class overhead. We have not shown results in Fig. 2 for 13 and 14 statements for scaling reasons; at this point register spillage causes an order-of-magnitude increase in execution time.

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. From those tests we expect that loop fusion can yield even greater performance gains than presented here.

Similar data for ETs is also available but not presented here. The poor results can be inferred from previous graphs on register spillage with ETs. Loop fusion amplifies this problem by adding even more operands in the inner loop.

Loop fusion is an important optimization that cannot be assured by any combination of C++ compiler or ET technology. This is because the ET implementation must introduce run-time checking for loop dependence so that parallelism (and general array semantics) can be supported. This introduces conditionals between statements (along with hundreds of lines of other ET code, parallel message passing, and so forth), which inhibits compiler-driven loop fusion. This is part of the justification for research on preprocessor mechanisms for loop fusion.

3.3 Stencil and Non-Stencil Applications

A++ implements both binary operator overloading and ETs, so it is a simple setting in which to evaluate the performance of the two mechanisms using the identical application codes. Figure 3 shows the performance of A++ with and without ETs and compares it to C for a 1D shock tube application. While this program does not characterizes all codes (it has no stencil operations, for example), numerical codes are composed of more than stencils. The equivalent performance of the two mechanisms is an unexpected result warranting further investigation.

The results shown in Fig. 4 show that performance is not uniform across all numerical codes. For stencil-like operations the performance of ETs is better than for binary operators, and in fact comes close to that of C code. For stencils additional improvements are possible with blocking schemes, but using them would obscure comparisions with the unblocked C code. The existing implementation is as efficient as the other results for ET implementations, yet for a non-stencil code its performance is no better than the binary operator implementation. We suspect that the computational costs are similar because the number of loads and cache misses are similar and there is little operand reuse on the right-hand sides of the statements.

4 Conclusions

The more traditional execution model of overloaded binary operators has performance problems on cache-based architectures with stencil-type expressions; this is less of an issue on other (e.g. vector) architectures. However, it appears to make little difference if stencils are not an important part of the computation. The ET implementations have extremely poor compile times (factors of 1000 or more), but yield a factor of only 3–4 improvement in operations involving stencils. Using the SGI C++ compiler, where the compile times are only a factor of

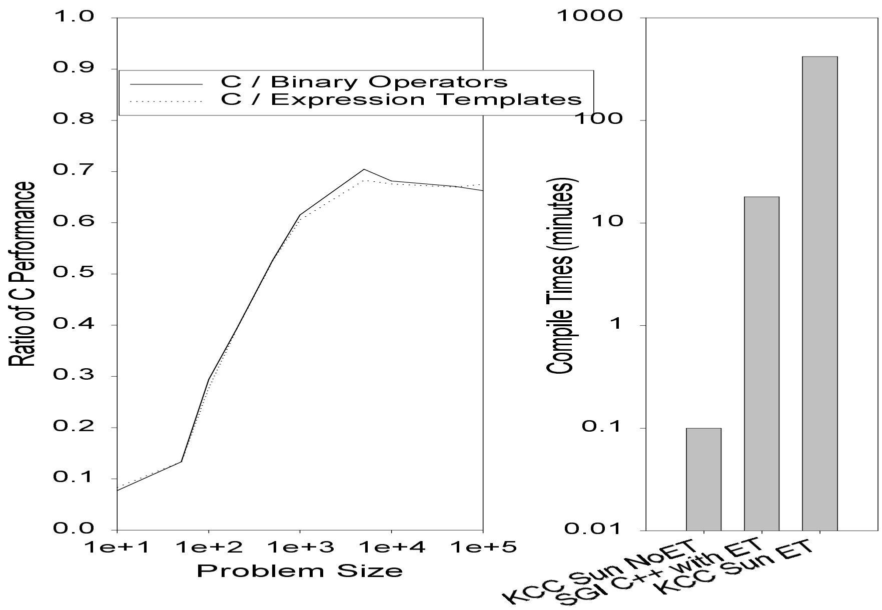

Fig. 3. 1-D Shock Tube Application with Compile Times

20–40 from that of the non-ET code, the performance is far worse than that for binary overloaded operators, for example 1/20 of C performance for 3-D arrays.

ETs also show poor performance for large stencils that appear in OVERTURE applications with complex geometry, though this would not be so much the case in 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 array expressions it can still only be compiled by a few C++ compilers (the KAI C++ compiler is the only one which achieves high performance). However, the performance of ETs is sensitive to the quality of the underlying C compiler, something that programmers cannot control.

Finally, we believe that attempts to link the future of object-oriented scientific computing to ET technology are flawed. With some effort better performance can be obtained with a special-purpose optimizing preprocessor. Tools such as SAGE++[3] greatly simplify the development of such preprocessors. More sophisticated transformations are possible with a preprocessor because it can perform semantics-based program analysis impossible with an ET approach. 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.

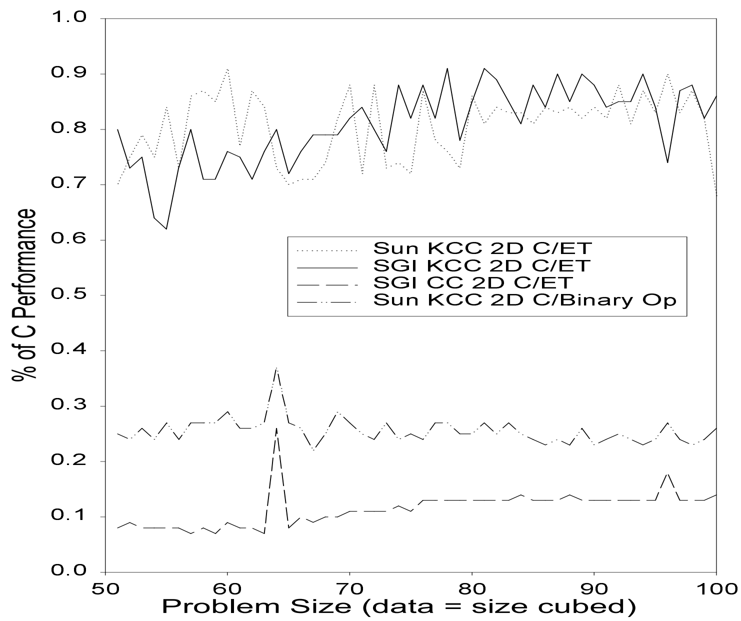

Fig. 4. 2-D Stencil

References

1. David Brown, Geoff Chesshire, William Henshaw, and Dan Quinlan. Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments. In *Proceedings of the SIAM Parallel Conference*, Minneapolis, MN, March 1997.
2. Kei Davis and Dan Quinlan. Rose: An optimizing preprocessor for the object-oriented overture framework. http://www.c3.lanl.gov/~dquinlan/ROSE.html.
3. B. Francois et. al. Sage++: An object-oriented toolkit and class library for building fortran and c++ restructuring tools. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, 1994.
4. M. Lemke and Dan Quinlan. P++, a c++ virtual shared grids based programming environment for architecture-independent development of structured grid applications. In *CONPAR/VAPP V*, LNCS. Springer-Verlag, September 1992.
5. Dan Quinlan and Rebecca Parsons. Run-time recognition of task parallelism within the p++ parallel array class library. In *Proceedings of the Conference on Parallel Scalable Libraries*, 1993.
6. Dan Quinlan and Rebecca Parsons. A++/p++ array classes for architecture independent finite difference computations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, April 1994.
7. Todd Veldhuizen. Expression templates. In S.B. Lippmann, editor, *C++ Gems*. Prentice-Hall, 1996.
8. O.V. Wilson and P. Lu. *Parallel Programming Using C++*, chapter 14, pages 547–587. MIT Press, 1997.