Preprint
UCRL-JC-146233

Treating a User-Defined Parallel Library as a Domain-Specific Language

D.J. Quinlan, B. Miller, M. Schordan, B. Philip

This article was submitted to
7th International Workshop on High-Level Programming Models and Supportive Environment, Ft. Lauderdale, FL, April 15-19, 2002

November 19, 2001

U.S. Department of Energy
Lawrence Livermore National Laboratory

Treating a User-Defined Parallel Library as a Domain-Specific Language

Daniel J. Quinlan[1], Brian Miller[1], Markus Schordan[1], and Bobby Philip[1]

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, Livermore, CA, USA

Abstract. An important purpose of a programming language is to insulate the programmer from low level details and provide a high enough level of abstraction to be productive and develop reasonabily portable application codes. For these reasons scientific programming is longer done using assembly language. But high performance of scientific applications often requires that critical sections of code be expressed at a particularly low level to avoid inefficencies introduced by the comiler (function call overhead, poor cache use, etc.). The use of high-level abstractions exaserbates this problem since the compiler is often unable to generate the equivalent low-level code required for good performance. The result is often significantly degraded performance.

Libraries provide a way for domain specific knowledge to be developed for large numbers of users. Libraries thus simplify the development of many application codes and the work spent building libraries can be amortized across large numbers of applications and application developers. Such a hierarchy puts langages and compilers at the root of tree of abstractsions developed within numerous libraries at one level and numerous applications at a second level. Libraries provide a way to define high-level abstractions.

We have developed specific libraries to simplify the development of serial and parallel scientific applications. The A++/P++ library provide an essential array abstraction for C++ scientific applications. The effect is to provide a single array abstraction that permits the development of serial code (using A++). The serial application code using the array abstractions need only be recompiled (using P++) to run on parallel distributed memory machines. The resulting abstractions are simple and powerful since it simplifies serial application code and even completely hides parallel details. But since it operates as a library the compiler is oblvious to its semantics and likewise the library is oblivious to the context of the use of its abstractons within the users application code. It is discouraging that the development of efficient code from high-level abstractions is blocked by compilers that are unable to use very specific high-level semantics essentially because it is user-defined.

In this paper we show how high-level serial and parallel libraries have been used to simplify the development of scientific applications and how with the specific semantics of such high-level abstractions we can develop

preprocessors that don't extend the C++ language but instead permit the user-defined semantics of the high-level abstractions to be leverages together with the context of the high-level abstractions within the user's application to optimize the performance of the final application code.

1 Introduction

The future of scientific computing depends upon the development of more sophisticated application codes. The original use of Fortran represented higher-level abstractions than the assembly instructions that preceded it, but exhibited performance problems that took years to overcome. However, the abstractions represented in Fortran were *standardized* within the language; today's much higher-level object-oriented abstractions are more difficult to optimize because they are *user-defined.*

The introduction of parallelism greatly exacerbates the compile-time optimization problem. While serial languages serve well for parallel programming, they know only the semantics of the serial language. As a result a serial compiler cannot introduce scalable parallel optimizations. Significant potential for optimization of parallel applications is lost as a result. There is a significant opportunity to capitalize upon the parallel semantics of the object-oriented framework and drive significant optimizations specific to both shared memory and distributed memory applications.

We present a preprocessor based mechanism, called *ROSE*, that optimizes parallel object-oriented scientific application codes that use high-level abstractions provided by object-oriented libraries. In contrast to compile-time optimization of basic language abstractions (loops, operators, etc.), the optimization of the *use* of library abstractions within applications has received far less attention. With ROSE, library developers define customized optimizations and build specialized preprocessors. Source-to-source transformations are then used to provide an efficient mechanism for introducing such custom optimizations into user applications. A significant advantage of our approach is that preprocessors can be built which are tailored to user-defined high-level abstractions, while vendor supplied C++ compilers know only the lower-level abstractions of the C++ lan-

guage they support. So far, our research has focused on applications and libraries written in C++.

This approach permits us to leverage existing vendor C++ compilers for architecture specific back-end optimizations. Significant improvements in performance associated with source-to-source transformations have already been demonstrated in recent work, underscoring the need for further research in this direction.

Other work exists which is related to our own research. Internally within ROSE a substantially modified version of the *SAGE II* [7] AST restructuring tool is used. *Nestor* [9] is a similar AST restructuring tool for Fortran 77, Fortran 90, and HPF2.0, which, however, does not attempt to recognize and optimize high-level user-defined abstractions. Work on *MPC++* [10, 11] has led to the development of a C++ tool similar to SAGE, but with some additional capabilities for optimization. However, it does not attempt to address the sophisticated scale of abstractions that we target or the transformations we are attempting to introduce.

Related work on *telescoping languages* [8] shares some of the same goals as our research work and we look forward to tracking its progress in the coming years. Other approaches we know of are based on the definition of library-specific *annotation languages* to guide optimizing source code transformations [12] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations [13].

Work at University of Tennessee has lead to the development of *Automatically Tuned Linear Algebra Software* (ATLAS) [5]. Within this approach numerous transformations are written to define a search space and the performance of a given architecture is evaluated. The parameters associated with the best performing transformation are thus identified. Our work is related to this in the sense that this is one possible mechanism for the identification of optimizing transformations that could be used within preprocessors built using ROSE to optimize application codes. Our approach to the specification of transformations in this paper is consistent with the source code generation techniques used to generate transformations within ATLAS.

The remainder of this paper is organized as follows. In section 2 we give a survey on the ROSE infrastructure; we describe the process of automatically generating library-specific preprocessors and

explain their source-to-source transformation mechanisms. The main focus of this paper is on the specification of these source-to-source transformations by the developer of the library. We will thus discuss two alternative specification approaches and an AST query mechanism in section ??. In section 4 we finally summarize our work.

2 ROSE Overview

We have developed ROSE as a preprocessor mechanism because our focus is on optimizing the use of user-defined high-level abstractions and not on lower-level optimizations associated with back-end code generation for specific platforms. Our approach permits ROSE to work as a preprocessor independent of any specific C++ compiler.

In the following we will briefly describe the internal structure of a preprocessor which has been automatically generated using ROSE; particularly the recognition of high-level abstractions (section 2.1), the overall preprocessor design (section 2.2), and finally the specification of the transformations (section ??), which is the main focus of this paper.

2.1 Recognition of Abstractions

We recognize abstractions within a user's application much the same way a compiler recognizes the syntax of its base language. To recognize high-level abstractions we build a hierarchy of *high-level abstract grammars* and the corresponding *high-level ASTs* using ROSE. This hierarchy is what provides for a relationship to telescoping languages [8].

These high-level abstract grammars are very similar to the base language abstract grammar — in our case an abstract C++ grammar. They are modified forms of the base language abstract grammar with added terminals and non-terminals associated with the abstractions we want to recognize. They cannot be modified in any way to introduce new keywords or new syntax, so clearly there are some restrictions. However, we can still leverage the lower-level compiler infrastructure; the parser that builds the base language AST. New terminals and nonterminals added to the base language abstract grammar might represent specific user-defined functions, data-

structures, user-defined types, etc. More detail about the recognition of high-level abstractions can be found in [3]

2.2 Preprocessor Design

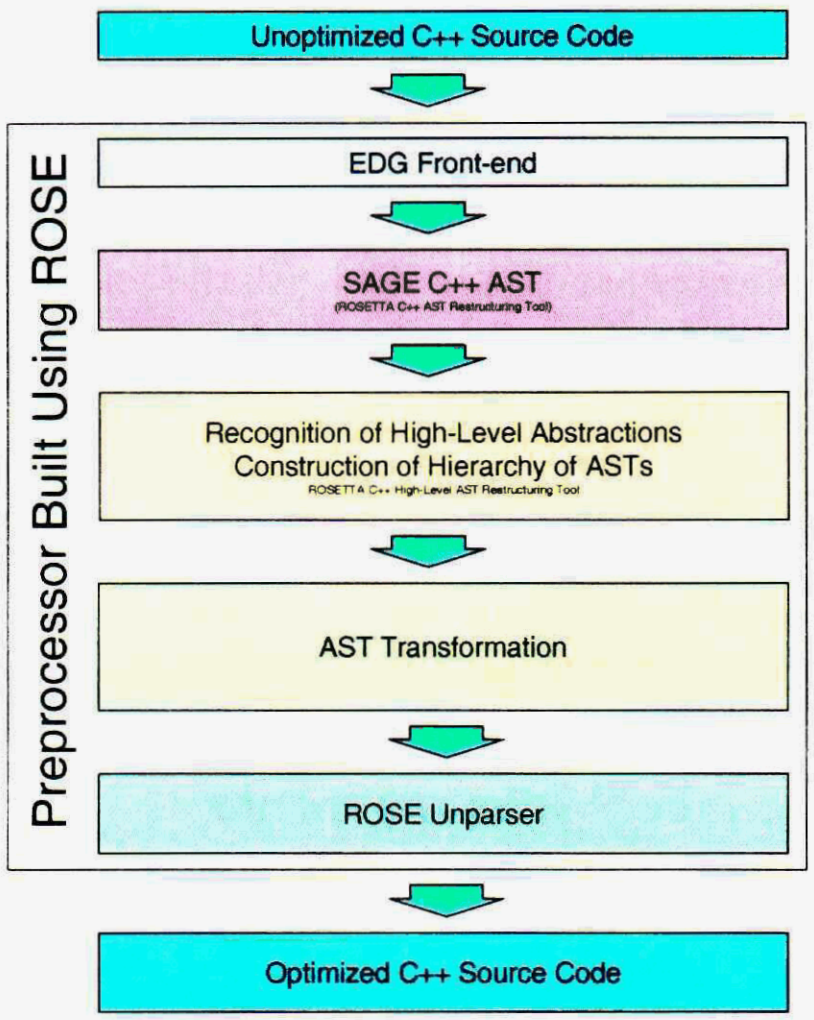

Fig. 1. Source-to-source C++ transformation with preprocessors using the ROSE infrastructure.

Figure 1 shows how the individual ASTs are connected in a sequence of steps; automatically generated translators generate higher level ASTs from lower level ASTs. The following describes these steps:

1. The first step generates the Edison Design Group (EDG) AST. This AST has a proprietary interface and is translated in the second step to form the abstract C++ grammar's AST.
2. The C++ AST restructuring tool is generated by ROSETTA [1] and is essentially comformant with the SAGE II implementation. This second step is representative of what SAGE II provides and

presents the AST in a form where it can be modified with a non-proprietary public interface. At this second step the original EDG AST is deleted and afterwards is unavailable.

3. The third step is the most interesting since at this step the abstract C++ Grammar's AST is translated into higher level ASTs. Each parent AST (associated with a lower level abstract grammar) is translated into all of its child ASTs so that the hierarchy of abstract grammars is represented by a corresponding hierarchy of ASTs (one for each abstract grammar). Transformations can be applied at any stage of this third step and modify the parent AST recursively until the AST associated with the original abstract C++ grammar is modified. At the end of this third step all transformations have been applied.
4. The fourth step is to traverse the C++ AST and generate optimized C++ source code (unparsing). This completes the source-to-source preprocessing.

An obvious next and final step is to compile the resulting optimized C++ source code using a vendor's C++ compiler.

3 Performance Measurements

We wish to compare the parallel performance of a ROSE-transformed C++ code to an HPF implementation solving the same problem. We choose to solve the simple partial differential equation (PDE)

$$u_t + u_x + u_y = f(x,y,t) \qquad (x,y) \in \Omega, t > 0 \qquad (1)$$

$$u(x,y,0) = u_0(x,y) \qquad (x,y) \in \Omega \qquad (2)$$

$$u(x,y,t) = u_e(x,y,t) \qquad (x,y) \in \partial\Omega, t > 0. \qquad (3)$$

Where we fix an exact solution $u_e = (1+t)(2+x+y)$ which we use to determine the forcing $f(x,y,t)$ and boundary conditions for the PDE. The domain Ω is the unit square $(x,y) \in [0,1] \times [0,1]$. We use centered finite differences to discretize the x and y derivatives, and the leap frog method to advance in time. This numerical method is formally second order accurate and thus solves the PDE exactly. We use this fact to ensure the correctness of our implementation and to detect any errors introduced by the optimizing compiler.

Our C++ implementation takes advantage of restricted pointers. That is, pointers are guaranteed to have no aliases. With this assumption, the code should perform as well as a FORTRAN 77 implementation. To test this for the platform of interest, we construct three smaller test codes that simply apply a five point stencil operation and then copy one array to another. This loop test was written in FORTRAN 77, ANSI C, and ANSI C++.

Our test machine is ASCI Blue Pacific at LLNL. This IBM machine consists of 256 compute nodes, each node containing 4 332MHz. PowerPC 604e CPUs with 1.5 GB of RAM. Our initial test was to confirm that our loop test codes written in C and C++ could indeed achieve F77 performance levels when run on a single processor. Table 3 shows the compiler options used to compile each version of the loop test. This table also shows the total computation time for the loop test, 100 repetitions of applying a five point stencil operation and copying one 1000x1000 array to another.

xlf	-qarch=auto -O4 -qhot	.169 s.
xlc	-O5 -qarch=auto -qtune=auto -qcache=auto -qalias=allp -qunroll=6	.159 s.
KCC	-O3 +K3 -qmaxmem=8192 -backend "-O5 -qalias=allp -qunroll=6" -restrict –abstract_pointer	.158 s.

Table 1. Compiler Options

These results confirm that under the right conditions, namely using restricted pointers and aggressive optimization, C and C++ code can achieve FORTRAN like performance. We next turn to our intended target, a performance comparison of the numerical solution of the linear PDE (1), (2), and (3).

Each code partitions the computational domain into strips perpendicular to the x-axis. The HPF code represents the solution values using its intrinsic distributed arrays. The C++ code uses the P++ parallel array class library to do the same. We have tested three P++ based codes using various levels of abstraction available in P++. Two scaling studies are presented. The first keeps the array size fixed as the number of processors grows from 1 to 64 while

the second test fixes the array size per processor for all numbers of processors.

np	HPF	P++ High	P++ Med	P++ Low
1	39.5		133.9	38.8
2	23.3		72.4	20.7
4	14.0		44.3	14.0
8	7.2		22.9	7.5
16	3.9		12.3	3.8
32			7.0	2.5
64			3.65	1.4

Table 2. Scaling for constant size problem

P++ High represents using the highest level of abstractions available in P++, with the resulting code looking very much like HPF. **P++ Med** uses a lower level API to access C++ objects local to each processor. **P++ Low** is the lowest level API available in P++ using pointers to data local to each processor. This code has at its core loops over C arrays, but also achieves HPF like performance. The ROSE-preprocessed code will use this level of abstraction to meet our performance requirements.

Table 3 indicates that although all versions of the code scale equally, only the version of the code using the lowest level API achieves the performance of HPF. In Table 3 we see as before, that all versions of the code scale similarly, but only the C++ version using the lowest level P++ API achieves HPF performance.

np	HPF	P++ High	P++ Med	P++ Low
1				
2				
4				
8				
16				
32				
64				

Table 3. Scaling for constant size per processor problem

4 Conclusions

ROSE is a library to simplify the construction of optimizing preprocessors. The specification of the transformation is done within the program that is compiled to be the preprocessor. This program leverages both the ROSE library for internal infrastructure and the source code generated by ROSETTA (part of ROSE). Source code generated by ROSETTA implements AST restructuring tools corresponding to abstract grammars and higher-level abstractions, this source code is compiled to build the preprocessor. Infrastructure within ROSE permits the specification of transformations, either directly modifying the AST or indirectly through the specification of source-strings which are processed to form AST fragments which are used to modify the AST.

We have presented the ROSE infrastructure to automatically generate library-specific source-to-source compilers (preprocessors). These preprocessors can be used to optimize the use of high-level abstractions in parallel object-oriented applications.

We have presented two basic approaches for specifying transformations. While our first approach of direct AST construction turned out to be tedious (especially for complex cache-based transformations), our second approach, which leverages the compiler front-end instead, provides an elegant and comfortable alternative.

References

1. Quinlan, D., Philip, B., "ROSETTA: The Compile-Time Recognition Of Object-Oriented Library Abstractions And Their Use Within Applications", Proceedings of the PDPTA'2001 Conference, Las Vegas, Nevada, June 24-27 2001
2. Quinlan, D., "ROSE: Compiler Support for Object-Oriented Frameworks", Parallel Processing Letters, Vol. 10, also Proceedings of Conference on Parallel Compilers (CPC2000), Aussois, France, January 2000.
3. Quinlan, D. Schordan, M. Philip, B. Kowarschik, M. "Parallel Object-Oriented Framework Optimization", (submitted to) Special Issue of Concurrency: Practice and Experience, also in Proceedings of Conference on Parallel Compilers (CPC2001), Edinburgh, Scotland, June 2001.
4. Brown, D., Henshaw, W., Quinlan, D., "OVERTURE: A Framework for Complex Geometries", Proceedings of the ISCOPE'99 Conference, San Francisco, CA, Dec 7-10 1999.
5. ATLAS homepage, `http://www.netlib.org/atlas`.
6. Edison Design Group, `http://www.edg.com`.

7. Bodin, F. 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.
8. Broom, B., Cooper, K., Dongarra, J., Fowler, R., Gannon, D., Johnsson, L., Kennedy, K., Mellor-Crummey, J., Torczon, L., "Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries", Journal of Parallel and Distributed Computing, 2000.
9. Silber, G.-A., `http://www.ens-lyon.fr/~gsilber/nestor`.
10. Ishikawa, Y., et. al., "Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —", Proceedings of Reflection'96 Conference, April 1996, more info available at: `http://pdswww.rwcp.or.jp/mpc++/mpc++.html`.
11. Chiba, S., "Macro Processing in Object-Oriented Languages", Proc. of Technology of Object-Oriented Languages and Systems (TOOLS Pacific '98), Australia, November, IEEE Press, 1998, more info available at: `http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html`.
12. Guyer, S.Z., Lin, C., "An Annotation Language for Optimizing Software Libraries", Proceedings of the Second Conference on Domain-Specific Languages, October 1999.
13. Menon, V., Pingali, K., "High-Level Semantic Optimization of Numerical Codes", Proceedings of the ACM/IEEE Supercomputing 1999 Conference (SC99), Portland, OR, 1999.
14. Bassetti, F., Davis, K., Quinlan, D., "Optimizing Transformations of Stencil Operations for Parallel Object-Oriented Scientific Frameworks on Cache-Based Architectures" Proceedings of the ISCOPE'98 Conference, Santa Fe, NM, 1998.
15. Weiß, C., Karl, W., Kowarschik, M., Rüde, U., "Memory Characteristics of Iterative Methods", Proceedings of the ACM/IEEE Supercomputing 1999 Conference (SC99), Portland, OR, 1999.
16. Lemke, M., Quinlan, D., "P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications", published as part of CONPAR/VAPP V, September 1992, Lyon, France; also published in Lecture Notes in Computer Science, Springer Verlag, September 1992.
17. Parsons, R., Quinlan, D., "A++/P++ Array Classes for Architecture Independent Finite Difference Computations", Proceedings of the Second Annual Object-Oriented Numerics Conference, pages 408-418, Sunriver, OR, April 1994.