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

Approved for public release; further dissemination unlimited
DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes 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 the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This report has been reproduced directly from the best available copy.

Available electronically at http://www.doc.gov/bridge

Available for a processing fee to U.S. Department of Energy
And its contractors in paper from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831-0062
Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-mail: reports@adonis.osti.gov

Available for the sale to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/ordering.htm

OR

Lawrence Livermore National Laboratory
Technical Information Department's Digital Library
http://www.llnl.gov/tid/Library.html
Treating a User-Defined Parallel Library as a Domain-Specific Language

Daniel J. Quinlan¹, Brian Miller¹, Markus Schordan¹, and Bobby Philip¹

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 reasonably 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 inefficiencies introduced by the compiler (function call overhead, poor cache use, etc.). The use of high-level abstractions exacerbates 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 languages and compilers at the root of tree of abstractions 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++) and the use of 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 par-
allel programming, they know only the semantics of the serial lan-
guage. As a result a serial compiler cannot introduce scalable parallel
optimizations. Significant potential for optimization of parallel ap-
plications is lost as a result. There is a significant opportunity to cap-
italize 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 ab-
stractions 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 cus-
tom 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

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 conformant 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 ab-
stract C++ Grammar's AST is translated into higher level ASTs.
Each parent AST (associated with a lower level abstract gram-
mar) 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 ab-
stract 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 opti-
mized C++ source code (unparsing). This completes the source-
to-source preprocessing.

An obvious next and final step is to compile the resulting opti-
mized 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)

\[ u_t + u_x + u_y = f(x, y, t) \quad (x, y) \in \Omega, t > 0 \quad (1) \]
\[ u(x, y, 0) = u_0(x, y) \quad (x, y) \in \Omega \quad (2) \]
\[ u(x, y, t) = u_c(x, y, t) \quad (x, y) \in \partial\Omega, t > 0. \quad (3) \]

Where we fix an exact solution \( u_c = (1 + t)(2 + x + y) \) which we
use to determine the forcing \( f(x, y, t) \) and boundary conditions for
the PDE. The domain \( \Omega \) 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.

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Options</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>xlf</td>
<td>-qarch=auto -O4 -qhot</td>
<td>.169 s.</td>
</tr>
<tr>
<td>xlc</td>
<td>-O5 -qarch=auto -qtune=auto -qcache=auto -qalias=allp -qunroll=6</td>
<td>.159 s.</td>
</tr>
<tr>
<td>KCC</td>
<td>-O3 +K3 -qmaxmem=8192 -backend &quot;-O5 -qalias=allp -qunroll=6&quot; restrict -abstract_pointer</td>
<td>.158 s.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>np</th>
<th>HPF</th>
<th>P++ High</th>
<th>P++ Med</th>
<th>P++ Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39.5</td>
<td>133.9</td>
<td>38.8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>23.3</td>
<td>72.4</td>
<td>20.7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14.0</td>
<td>44.3</td>
<td>14.0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>7.2</td>
<td>22.9</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>3.9</td>
<td>12.3</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>7.0</td>
<td>7.0</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td>3.65</td>
<td>1.4</td>
<td></td>
</tr>
</tbody>
</table>

Table 2. Scaling for constant size problem

\[ \text{P++ High} \] represents using the highest level of abstractions available in P++, with the resulting code looking very much like HPF. \[ \text{P++ Med} \] uses a lower level API to access C++ objects local to each processor. \[ \text{P++ Low} \] 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.

<table>
<thead>
<tr>
<th>np</th>
<th>HPF</th>
<th>P++ High</th>
<th>P++ Med</th>
<th>P++ Low</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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


