Treating a User-Defined Parallel Library as a Domain-Specific Language [*]

Daniel J. Quinlan Brian Miller Bobby Philip Markus Schordan
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory, Livermore, CA, USA

Abstract

The software crisis within scientific computing has been that application codes become larger and more complex. The only conceivable solution is to make application codes smaller and less complex. We know of no way to resolve this crisis, except to make each line of code mean *more; this* is *the process of defining high-level abstractions. Achieving high-performance from high-level abstractions represents an essential key to simplifying scientific software.*

This paper presents several high-level abstractions used within scientific computing. These abstractions are part of multiple object-oriented libraries and represent complex and precise semantics. In each case the semantics of the abstraction is user-defined and ignored by the compilation process at a significant performance penalty for the application code. Our research work presents a mechanism to analyze and optimize the use of high-level abstractions within scientific applications. In this paper, we show that the high-level abstractions are not just significantly easier to use in the development of application code but can be made to perform equivalently to hand-coded C and Fortran. Our research work shows how to effectively treat any object-oriented library and its abstractions as if it where a domain-specific language with equivalent builtin types and specialized compile-time analysis and optimizations. With acceptable performance of high-level abstractions within scientific software, we expect that application codes can be made smaller and less complex; allowing much more complex applications to be built in the future.

1. Introduction

The software crisis within scientific computing has been that application codes become larger and more complex. The only conceivable solution is to make application codes smaller and less complex. We know of no way to resolve this crisis, except to make each line of code *mean* more; this *is* the process of defining high-level abstractions. Achieving high-performance from high-level abstractions represents an essential key to simplifying scientific software.

The development of new languages requires significant time for the language to mature and requires a significant user base. The standards process for C++, for example, has taken many years; generally languages evolve at a slow pace. However, the required size of the user base tends to prevent the language being biased to any particular domain. In contrast, libraries are cheaper to develop, permit the relatively quick development of high-level abstractions, and don't require as large of a target user base to justify their development. As a result, libraries can more readily provide highly specialized abstractions which simplify application development. However, the abstractions within a language are generally better defined and their performance benefits from compile-time analysis and optimization using their known semantics.

The optimization of high-level abstractions within object-oriented libraries is essentially compromised by the library's inability to see the context of how they are used within the application program. Conversely, the compiler's inability to optimize the use of high-level abstractions is essentially due to its ignorance of the user-defined semantics of the library's high-level abstractions. The success of high-level abstractions within scientific computing is critically dependent upon the ability of users to get high performance from high-level abstractions that are otherwise compelling to use due to the significantly enhanced productivity that they represent.

Presently within scientific computing the successful abstractions are relatively coarse grain (elliptic equation solvers, etc.). Experience has found that finer granularity abstractions exhibit a performance penalty due to either higher function call overhead, poor cache behavior, or insufficient inter-procedural analysis/optimization. However, finer grain abstractions (e.g. multidimensional array abstractions, finite element abstractions, etc.) are particularly compelling because of how simply they permit the expression of numerical algorithms. Both coarse and fine granu-

[*]This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

larity abstractions have important roles in simplifying scientific software.

Scientific computing is unfortunately particularly performance sensitive. Research that we will present in this paper permits the compile-time analysis and optimization of even fine granularity abstractions using their user-defined semantics. Leveraging the user-defined semantics of high-level abstractions is a particularly critical part of making high-level abstractions useful within scientific computing. The C++ language's object-oriented mechanisms allow for the definition of high-level abstractions and the generation of user-defined types with overloaded operators. These mechanisms permit the definition of abstractions that are convenient and intuitive for users. We present two object-oriented libraries developed for scientific computation that provide numerous abstractions that are compelling to users but relatively fine grain and thus excellent targets for compile-time analysis and optimization.

We have developed specific libraries to simplify the development of serial and parallel scientific applications. The A++/P++[18] library provides an essential serial and parallel array abstraction for C++ scientific applications. The Overture[4] object-oriented framework permits even higher level abstractions built upon the A++/P++ array class library and specific to solving Partial Differential Equations (PDEs) on complex geometries. Since both A++/P++ and Overture are libraries the compiler is oblivious to their user-defined semantics and likewise the libraries cannot see the context of the use of their abstractions within the user's application codes. 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. Our optimization approach uses ROSE[3, 2] to implicitly define a higher-level grammar and build from this grammar a tool for the representation and modification of Abstract Syntax Trees (ASTs) of applications. Using ROSE, preprocessors can be built which introduce optimizations using source-to-source transformations for C++ applications. The resulting performance is equivalent to F77 and C code. In this specific case the high-level array abstractions are similar to those found in HPF (array abstractions). However, our approach is not limited to the specific HPF array abstractions and apply to any object-oriented abstraction (e.g. higher level abstractions within Overture). The result is a mechanism which can take a C++ object-oriented library and produce the compile-time optimizations previously available only within a compiler for a domain specific language with similar abstractions. This effectively permits any object-oriented library to appear indistinguishable from a domain specific language (even though its grammar is currently only implicitly represented). Further, our approach should be a particularly simple mechanism to define future domain specific libraries since they can be developed and evaluated easily as object-oriented libraries and optimized using an incrementally developed preprocessor approach using ROSE.

Related work on *telescoping languages* [8] shares some of the same goals as our research work. Other approaches we know of are based on the definition of library-specific *annotation languages* to guide optimizing source code transformations [13] and on the specification of both high-level languages and corresponding sets of axioms defining code optimizations [14]. Our appraoach is related to that of *semantic expansion* [9], but uses a source-to-source approach.

2. High-Level Parallel Abstractions

Application codes in scientific computing are becoming more and more complex. The use of object-oriented design techniques and programming languages represents a common approach towards overcoming many of the difficulties arising from the need for flexible and highly reusable software components. Usually large applications are based on high-level abstractions which are provided by underlying libraries. In our scientific applications, A++/P++ and Overture are important examples of object-oriented libraries providing high-level abstractions for numerical computations.

2.1. A++/P++ Library

A++/P++[17, 18] is a C++ class library implementing array operations for both serial and parallel environments.

The A++ *serial* array abstraction is similar to FORTRAN 90 in syntax. It provides multidimensional array objects which simplify the development of numerical software and provides a basis for developing parallel array abstractions. P++ provides a *parallel* array class abstraction which shares an identical interface to A++ abstractions by design. P++ provides a data parallel implementation of the array syntax represented by the abstractions within A++. As a result, A++ serial applications can be recompiled using P++ and thus run in parallel. This provides a simple and elegant mechanism for serial code to be reused in the parallel environment; simplifying the development of scientific software generally. While P++ shares a lot of commonality with FORTRAN 90 array syntax and the HPF programming model, P++ provides a programmable mechanism for the distribution of arrays and greater control as required for multiple grid applications represented by both the overlapping grid model and the adaptive mesh refinement (AMR) model present in some numerical computations.

Figure 1 shows a code fragment used to solve Poisson's equation in either a serial or parallel environment using the A++/P++ classes. Notice how the Jacobi iteration for the entire array can be written in one statement. The same code alternatively runs on distributed memory parallel computers. Alternative distributions can be specified if the defaults are inappropriate.

2.2. Overture

Overture[4] is an object-oriented C++ framework for solving partial differential equations associated with computational fluid dynamics applications within complex moving geometries. It uses the method of overlapping grids (also known as overset or Chimera grids; see figures 4, 5). Overture includes support for geometry, grid generation, difference operators, boundary conditions, database access and graphics. In Overture, the fundamental building blocks are objects such as grids and grid functions. These objects can be manipulated at a high level. Details of the implementation, such as how a grid is stored, are hidden from the user, but importantly all data is accessible to permit the optional use of the data directly as required for lower level optimizations. Within Overture, the A++/P++ array class library is used both internally and within the user interface.

The example shown in figure 3 demonstrates the power of the Overture framework by showing a basically complete code that solves the partial differential equation (PDE)

$$u_t + au_x + bu_y = nu(u_{xx} + u_{yy})$$

on any overlapping grid. It shows higher level abstractions represented within Overture (beyond that of the array abstractions).

2.3. The performance penalty of high-level abstractions

A common problem within object-oriented C++ scientific computing is that the high level semantics of abstractions introduced (e.g. parallel array objects) are ignored by the C++ compiler. Classes and overloaded operators are seen as unoptimizable structures and function calls. Such abstractions can provide for particularly simple development of large scale parallel scientific software, but the lack of optimization greatly effects performance and utility. Because C++ lacks a mechanism to interact with the compiler, elaborate mechanisms are often implemented within such parallel frameworks to introduce complex template-based and/or runtime optimizations (such as runtime dependence analysis, deferred evaluation, runtime code generation, etc.). These approaches are however not satisfactory since they either require long compile times (hours) or are not sufficiently robust.

3. ROSE Architecture

ROSE is a tool that provides a connection between a library and a domain specific language. Based on an abstract C++ grammar and the abstractions used in a library, a higher-level grammar is generated that represents library specific constructs as additional elements in the C++ grammar. The C++ grammar plus these new elements is called a higher-level grammar.

The main purpose of higher-level grammars is to ease the task of the user to introduce a program transformation. It should allow him to focus on those constructs that are library specific and qualify for transformation. Using ROSE the qualification of the library specific constructs and the generation of the higher-level grammar is automated. Additional constraints on library specific constructs can be expressed using a query mechanism which allows interrogation of the abstract syntax tree for program information.

Our approach is to add additional optimization capabilities to the existing compiler optimizations. Therefore, we developed ROSE as a preprocessor mechanism that allows us to generate optimized C++ code which then has to be compiled using a platform dependent C++ compiler.

A single compiler that would generate object code from application source would not be practical since it would require us to address back-end code generation issues. This would lead us toward platform-specific details we wish to avoid.

It is important to mention that no modification of the base language is possible, since the use of the optimizing preprocessor is optional. This avoids any deviation from the C++ standard which would lead to portability problems for applications.

Our implementation is based on leveraging a standard base language front-end for the development of the preprocessor. We currently employ the Edison Design Group [6] (EDG) C++ front-end and the Sage III intermediate representation (the AST) which is based on SAGE [7]. The component of ROSE that generates the code implementing the ASTs (for the abstract C++ grammar and higher-level grammars) is called ROSETTA [1].

In order to simplify and focus the development of a library-specific optimizing preprocessor, we subdivide the introduction of optimizations into two phases:

1. Recognition Phase:
   The automatic recognition of high-level abstractions within the user's application code allows a preprocessor to treat user-defined abstractions similarly to builtin types in a domain specific language, see 3.1.

2. Transformation Specification:
   A transformation is specified by the semantic actions that are attached to language constructs that qualify for

transformation in the recognition phase. Library developers are expected to specify the transformations that will be used to optimize the user's application code, see 3.2.

3.1. Recognition of Abstractions

The automatic recognition of higher-level abstractions allows us to treat a library as a domain specific language. It is based on the type information stored with each expression in the AST and the signatures of functions. Additional predicates are used to qualify statements as domain specific statements.

In our present implementation we use a "qualifying-predicate" for library specific statements. If the given predicate holds on all statements within the scope of a given statement this statement is qualified as a library specific statement.

For example, given a code snippet of the program in fig. 8:

```
t = 0.0;
old_A = A;
```

Variable `t` is of type `double` and variables `old_A` and `A` are of type `doubleArray`. The class `doubleArray` is a user defined abstraction and defined in the library with an overloaded assignment operator. The qualifying predicate defines user abstractions in the library for which optimizations have been implemented to qualify for transformation. The class `doubleArray` is one of those. Therefore expressions of type `doubleArray` are qualified to be transformed but expressions of type `int` are not. Hence, the program fragment `old_A=A` is qualified to be a library specific assignment but not `x=1`. Qualification of program constructs is not limited to type information. Basically any information that can be computed by using attributes and semantic actions can be used to define the qualifying predicate.

A high-level abstract grammar is an extended form of the base language abstract grammar with added terminals, non-terminals, and rules 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. New terminals and nonterminals added to the base language abstract grammar 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]

3.2. Specification of Transformations

Transformations are specified by defining functions to be executed during a traversal at each node in the AST. We allow values to be passed downwards and upwards the AST. This gives a similar capability as with inherited and synthesized attributes in attributed grammars. In our current implementation, inherited and synthesized attributes cannot be combined with the same freedom as an attribute-evaluator generator would allow, but it has turned out that our mechanism is sufficient to implement complex queries needed for gathering program information.

For example, a query can be used to gather information about the types used in an expression, about the nesting level of scopes, or the source that is represented by a given subtree of the AST, etc.

A program transformation can be specified by a set of possibly nested queries on the AST. However, the final step is to compute a string value representing the new source code. This string is then used as input to the front-end and by re-invoking the front-end we create a new AST. The transformation phase may be performed iteratively, to deal with different transformations.

We provide mechanisms to only locally modify the AST if only local optimizations are performed. For a more detailed description of the transformation specification see [2].

3.3. Preprocessor Execution Phases

Figure 6 shows how the individual phases of compilation 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 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 mod-

ified. 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.

4. Performance Measurements

To show the value in using such an automated source-to-source preprocessing tool, we compare the parallel performance of a ROSE-transformed C++ code to an HPF implementation solving the same problem. The C++ code is written using the P++ parallel array class library and shown in figure 8. Additional versions of this code have been written in HPF, C, and using the A++ abstractions locally on each processor. Using the abstractions available in the P++ library greatly simplifies code development with the resulting source code being quite compact and very easy to understand. We have used a preprocessor built using ROSE to automatically transform the high-level abstractions to code that can be highly optimized by the compiler, thereby achieving high performance[1]. Through a performance comparison of these different versions of the same code (implemented with different levels of abstractions) we will show that these techniques enable users to write C++ code using elegant high-level abstractions, yet still see runtime performance rivaling FORTRAN 77 in a serial environment and HPF in parallel.

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 $u_e = (1+t)(2+x+y)$ used 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 ROSE transformed 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++.

[1]Minor manual corrections were required to fix current bugs in the automated code generation

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 2 shows the compiler options used to compile each version of the loop test. This table also shows the total computation time in seconds for the loop test, 100 repetitions of applying a five point stencil operation and copying one 1000x1000 array to another. These results confirm that under the right conditions, namely using restricted pointers and aggressive optimization, C and C++ code can achieve FORTRAN like performance.

F77	A++	Transformed A++ Code
22.2 s.	115.0 s.	24.1 s.

Table 1. Performance of identical five point stencil using Fortran, A++, and automatically generated code using a ROSE preprocessor.

We next investigate whether the code output by a ROSE preprocessor is optimal. In Table 1 we present a comparison of the performance of a typical finite difference operation implemented using, FORTRAN 77, A++, and an automatically transformed version of that A++ code (with a preprocessor built using ROSE). These codes were compiled using the compiler options detailed in Table 2. We see that the ROSE transformed version achieves essentially the same performance as the F77 implementation.

We finally turn to our intended target, a performance comparison of the numerical solution of the linear PDE (1), (2), and (3). Separate codes solving these convection equations have been developed using different languages and abstractions: **Con-HPF**, **Con-P++**, **Con-A++**, **Con-ROSE**. **Con-HPF** is the convection code implemented in HPF. **Con-P++** uses the highest level of abstraction available in the P++ library, and is the type of code that users of the P++ library are expected to write. This code looks much like HPF, but performs poorly. **Con-A++** uses A++ abstractions locally on each processor and P++ for communication between processors. **Con-ROSE** is the code resulting from the replacement of the serial array objects in the **Con-A++** with preprocessor-generated transformations similar to those tested in Table 1. This code has at its core loops over C arrays and as seen in Tables 3 and 4 achieves HPF like performance.

Two scaling studies are presented. The first shown in Table 3 keeps the array size fixed as the number of processors grows from 1 to 64 while the second test (Table 4) fixes the array size per processor for all numbers of processors. We see that all versions of the code scale similarly, but only the code containing the ROSE preprocessor generated transformations performs at the same level as its FORTRAN counterparts.

5. Conclusions

Within this paper we have presented some example high-level abstractions used within our research work on numerical methods together with an evaluation of their performance on a relatively simple convection code. Clearly the unoptimized use of the abstractions results in slower performance than the equivalent HPF or F77 implementation. However, using a preprocessor built using ROSE and leveraging the semantics of the A++/P++ abstractions allows the same code to obtain performance equivalent to Fortran on both serial and parallel computers. Within this paper we have demonstrated an approach to use the semantic information about user-defined abstractions to drive optimizations which could not otherwise be done by the compiler and which complement those done by the compiler. With the compile-time analysis and optimization of high-level used-defined abstractions our work permits object-oriented libraries to have compile-time support equivalent to domain-specific languages.

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., Kowarschik, M., Philip, B., Schordan, M. ”The Specification of Source-To-Source Transformations for the Compile-Time Optimization of Parallel Object-Oriented Scientific Applications”, Submitted to Parallel Processing Letters, also in Proceedings of 14th Workshop on Languages and Compilers for Parallel Computing (LCPC2001), Cumberland Falls, KY, August 1-3 2001.

[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, `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] Wu, P., Midkiff,S., Moreira, J., Gupta, M., ”Efficient Support for Complex Numbers in Java”, Preceedings of Java Grande Conference, 1999.

[10] Silber, G.-A., `http://www.ens-lyon.fr/~gsilber/nestor`.

[11] 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`.

[12] 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.

[13] Guyer, S.Z., Lin, C., ”An Annotation Language for Optimizing Software Libraries”, Proceedings of the Second Conference on Domain-Specific Languages, October 1999.

[14] Menon, V., Pingali, K., ”High-Level Semantic Optimization of Numerical Codes”, Proceedings of the ACM/IEEE Supercomputing 1999 Conference (SC99), Portland, OR, 1999.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

```
// Solve u_xx + u_yy = f by a Jacobi Iteration
   Range R(0,n)               //indices: 0,1,2,...,n
   floatArray u(R,R), f(R,R)  // declare 2 2-d arrays
   f = 1.; u = 0.; h = 1./n;  // initialize
   Range I(1,n-1), J(1,n-1);  // define interior

// data parallel statement
   for( int iteration=0; iteration<100; iteration++ )
      u(I,J)=.25*(u(I+1,J)+u(I-1,J)+u(I,J+1)+
                  u(I,J-1)-f(I,J)*(h*h));
```

Figure 1. Example A++/P++ code showing use of array abstraction for Poisson’s equation.

```
int main()
{
  CompositeGrid cg;                      // create grid
  getFromADataBaseFile(cg,"myGrid.hdf"); // read grid
  floatCompositeGridFunction u(cg);//create grid func
  u=1.;                            // assign data
  CompositeGridOperators op(cg);   // create operators
  u.setOperators(cg);
  PlotStuff ps;                    // make a plot object
  // --- solve a PDE ----
  float t=0, dt=.005, a=1., b=1., nu=.1;
  for (int step=0; step < 100; step++)
  {
    u+=dt*( -a*u.x()-b*u.y()+nu*(u.xx()+u.yy()) );
    t+=dt;
    u.interpolate();       // interpolate overlap bdys
    // apply the BC u=0 on all boundaries
    u.applyBoundaryCondition(0,dirichlet,allBndries,0);
    u.finishBoundaryConditions();
    ps.contour(u);                // plot contours
  }
  return 0;
}
```

Figure 3. Overture code demonstrating use of *floatCompositeGridFunction* abstraction. In this example differential operators associated with u are combined to form a simple discretization of the convective diffusion equation $u_t + au_x + bu_y = nu(u_{xx} + u_{yy})$ for an overlapping grid.

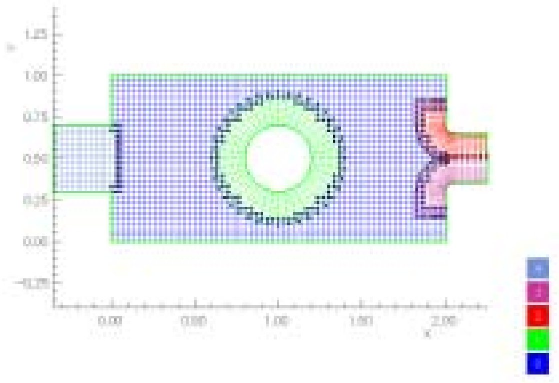
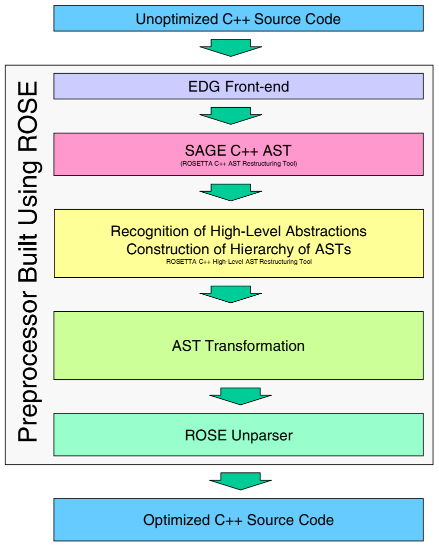


Figure 4. Example overlapping grid used in Overture code.



Unoptimized C++ Source Code
Preprocessor Built Using ROSE
EDG Front-end
SAGE C++ AST
(ROSETTA C++ AST Restructuring Tool)
Recognition of High-Level Abstractions
Construction of Hierarchy of ASTs
ROSETTA C++ High-Level AST Restructuring Tool
AST Transformation
ROSE Unparser
Optimized C++ Source Code

Figure 6. Source-to-source C++ transformation with preprocessors using the ROSE infrastructure.

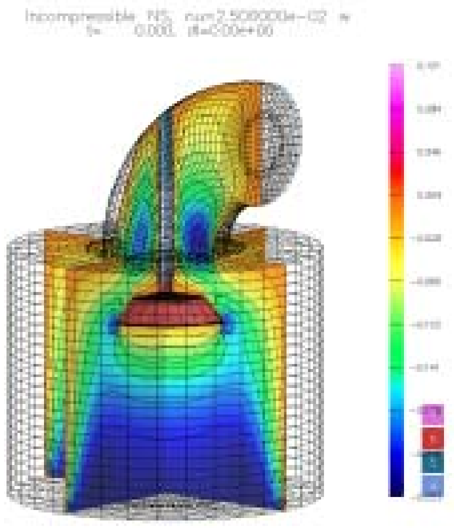


Figure 5. Example overlapping grid for complex combustion cylinder geometry.

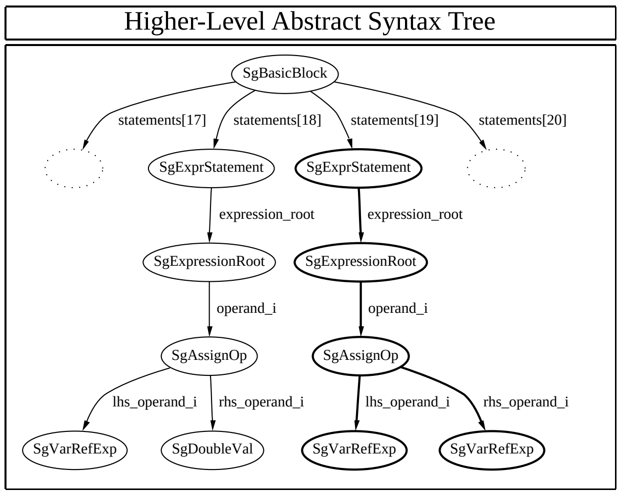


Figure 7. AST fragment representing t=0.0; old_A=A; in figure 8 (recognized as library specific (in bold))

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 2. Compiler Options used for different versions of convection code and related tests.

np	Con-HPF	Con-P++	Con-A++	Con-ROSE
1	39.7	359.6	133.9	38.8
2	23.4	192.4	72.4	20.7
4	14.1	113.0	44.3	14.0
8	6.9	58.4	22.9	7.5
16	3.9	30.2	12.3	3.8
32	2.2	16.2	7.0	2.5
64	1.2	8.8	3.6	1.4

Table 3. Scaling for constant size problem, time reported in seconds.

np	Con-HPF	Con-P++	Con-A++	Con-ROSE
1	39.5	359.6	135.5	37.4
2	45.1	381.0	143.8	40.4
4	54.7	446.2	172.0	50.9
8	54.8	449.4	172.0	50.7
16	55.7	454.0	172.7	50.9
32	55.3	450.2	174.6	51.8
64	54.9	450.5	173.9	52.9

Table 4. Scaling for constant size per processor problem, time reported in seconds

```
#include "A++.h"

int main(int argc, char** argv)
{
  int numProcs=0,i,j;
  Optimization_Manager::
   Initialize_Virtual_Machine("",numProcs,argc,argv);

  const int Xsize=1003, Ysize=1003*numProcs;
  const Range ix(-1,Xsize-2), iy(-1,Ysize-2), all;
  const Range ix1(0,Xsize-3), iy1(0,Ysize-3);
  const Range iU(Xsize-2,Xsize-2),iL(-1,-1);
  const Range jU(Ysize-2,Ysize-2),jL(-1,-1);

  doubleArray A(ix,iy),old_A(ix,iy),x(ix,iy),y(ix,iy);
  double dx=1./(Xsize-3), dy=1./(Ysize-3);
  double t=0.0,maxError, dt=0.1/(Xsize+Ysize);

  for(i=x.getLocalBase(0);i<=x.getLocalBound(0);i++)
    x(i,all) = i*dx;
  for(j=y.getLocalBase(1);j<=y.getLocalBound(1);j++)
    y(all,j) = j*dy;

  x.updateGhostBoundaries();
  y.updateGhostBoundaries();

  A = (1.0 + t)*(2.0 + x + y);
  A.updateGhostBoundaries();

  t = 0.0;
  old_A = A;
  old_A.updateGhostBoundaries();
  doubleArray temp(ix,iy);

  temp(ix1,iy1) = A(ix1,iy1) -
      dt*((A(ix1+1,iy1)-A(ix1-1,iy1))/(2.0*dx)+
          (A(ix1,iy1+1)-A(ix1,iy1-1))/(2.0*dy)-
          (4.0+2.0*t+x(ix1,iy1)+y(ix1,iy1)) );
  A(ix1,iy1) = temp(ix1,iy1);
  A(all,jL)=(1+(t+dt))*(2+x(all,jL)+y(all,jL));
  A(all,jU)=(1+(t+dt))*(2+x(all,jU)+y(all,jU));
  A(iL,iy1)=(1+(t+dt))*(2+x(iL,iy1)+y(iL,iy1));
  A(iU,iy1)=(1+(t+dt))*(2+x(iU,iy1)+y(iU,iy1));
  A.updateGhostBoundaries();
  t += dt;

  for (int k = 0; k<100; k++)
  {
    temp(ix1,iy1) = old_A(ix1,iy1) -
      2.*dt*((A(ix1+1,iy1)-A(ix1-1,iy1))/(2*dx)+
             (A(ix1,iy1+1)-A(ix1,iy1-1))/(2*dy)-
              (4+2*t+x(ix1,iy1)+y(ix1,iy1)) );
    old_A = A;
    A(ix1,iy1)= temp(ix1,iy1);
    A(all,jL)=(1+(t+dt))*(2+x(all,jL)+y(all,jL));
    A(all,jU)=(1+(t+dt))*(2+x(all,jU)+y(all,jU));
    A(iL,iy1)=(1+(t+dt))*(2+x(iL,iy1)+y(iL,iy1));
    A(iU,iy1)=(1+(t+dt))*(2+x(iU,iy1)+y(iU,iy1));
    A.updateGhostBoundaries();
    t +=dt;
  }
  Optimization_Manager::Exit_Virtual_Machine();
  return 0;
}
```

Figure 8. Convection code used for performance benchmarks of high-level abstractions (shown here is the P++ version of the code, not shown are other versions of the same code in HPF, and various levels of automated and non-automated optimizations).

```
// Automatically Introduced Transformation
{
  LOOP_INDEX_DATA_VARIABLES_MACRO();
  // Data variables declaration macro to support
  // transformations (once per transformation)
  Extra3 = Index(0,1); Extra4 = Index(0,1);Extra5 = Index(0,1); Extra6 = Index(0,1);

  ARRAY_STATEMENT_TRANSFORMATION_DATA_MACRO(uLocal,ILocInterior,JLocInterior,Extra3,Extra4,Extra5,Extra6);

  // Data variables declaration macro to support transformations (once per operand)

  ARRAY_STATEMENT_OPERAND_TRANSFORMATION_DATA_MACRO(uLocal);
  double* APP_RESTRICT uLocal_pointer = uLocal.Array_Descriptor.Array_View_Pointer5;

  ARRAY_STATEMENT_OPERAND_TRANSFORMATION_DATA_MACRO(fLocal);
  double* APP_RESTRICT fLocal_pointer = fLocal.Array_Descriptor.Array_View_Pointer5;

  // for loop for 2D array operations with stride
  for (index2 = base2; index2 <= bound2; index2++)
    for (index1 = base1; index1 <= bound1; index1++)
    {
      uLocal_pointer[SUBSCRIPT(0,0)] = .25*(uLocal_pointer[SUBSCRIPT(+1,0)]+
         uLocal_pointer[SUBSCRIPT(-1,0)] + uLocal_pointer[SUBSCRIPT(0,+1)]+
         uLocal_pointer[SUBSCRIPT(0,-1)] - fLocal_pointer[SUBSCRIPT(0,0)]*(h*h));
    }
}//end of tranformation block
```

Figure 2. Example of tranformed A++/P++ code from figure 1. This code uses macros for array subscript computation and variable initialization. Replacing the high level array abstractions with direct access to the underlying pointers results in improved performance.