
Overture Framework and General Optimizations of Object-Oriented Applications

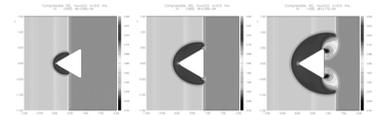
Object-Oriented Tools for Solving Partial Differential Equations

Dan Quinlan

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory



The Overture team at CASC



David Brown
*finite volume
methods, adaptive
mesh refinement*



Bill Henshaw
*grid generation,
CFD, combustion, multigrid,
Overture Framework*



Dan Quinlan
*P++, PADRE, ROSE,
AMR++*



Kyle Chand
hybrid grids



Brian Miller
*level set methods,
parallel methods*



Danny Thorne
*cache-based optimization,
ROSE*



Petri Fast
*Hele-Shaw flows;
fluid-elastic
interactions*



Anders Petersson
grid generation, CFD



Brian Gunney
ROSE, PADRE



Bobby Phillip
*AMR++, fluid-
elastic interactions, multigrid*



2

Overture project interacts extensively with university and Lab collaborators

- RPI
- CSU Fort Collins
- University of Colorado
- UC Davis
- Dartmouth
- University of Auckland, NZ
- Royal Institute of Technology, Sweden
- Dutch National Res. Ctr. For Math and Comp. Sci.

- Argonne National Laboratory : PETSc Interoperability
- Pacific Northwest National Laboratory: PADRE Interoperability



3

Outline of Talk

- OVERTURE Framework
 - Introduction (Overset Grids, Grid Generation, Solvers)
 - Example Applications
 - How Overture Abstractions Work Together
 - Interoperability through PADRE
 - Optimizations
 - ROSE (Performance Optimizations)
 - Support for User Defined Grammars (ROSETTA)
 - Support for User Defined Optimizations

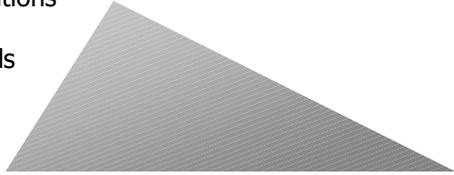


4

Software Development is a balancing act involving different types of complexity

Algorithm Complexity

- MultiGrid
- AMR
- Implicit/Explicit Equations
- Conservation
- Higher Order Methods



Application Complexity

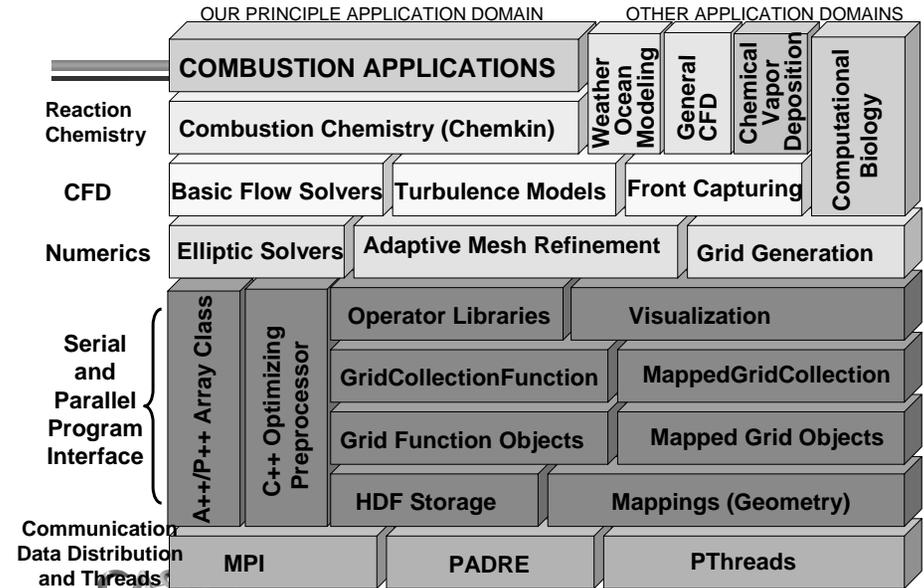
- Geometry
- Physics

Architecture Complexity

- Cache
- Vector
- Shared Memory Parallelism
- Distributed Memory Parallelism

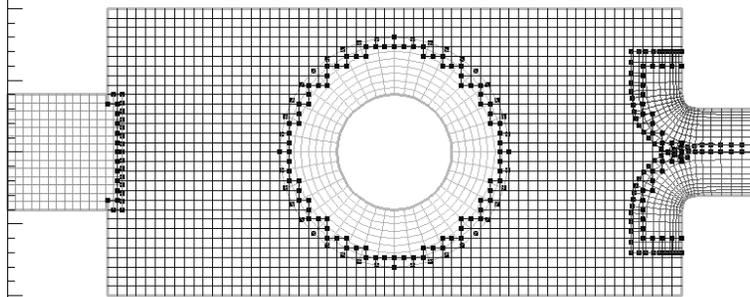


Design of the OVERTURE Framework

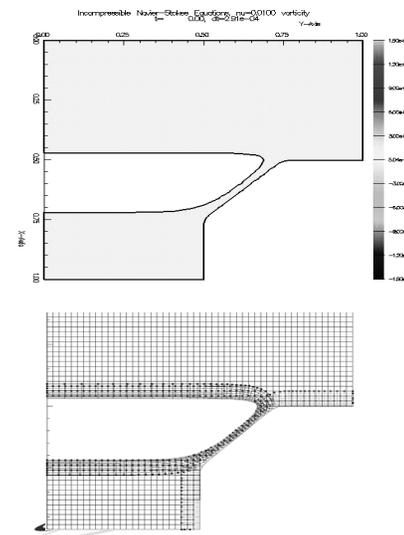


An *overset grid* consists of a set of logically rectangular curvilinear grids that overlap where they meet and which completely cover the computational domain.

The *Overture* grid generator, *Ogen*, automatically computes the overlap and connectivity information.

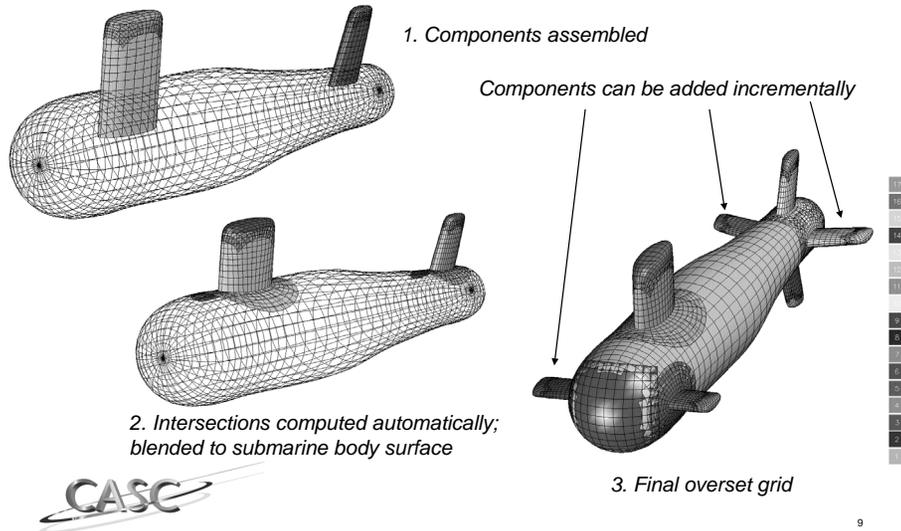


Overture supports overlapping grid technology for complex moving geometries

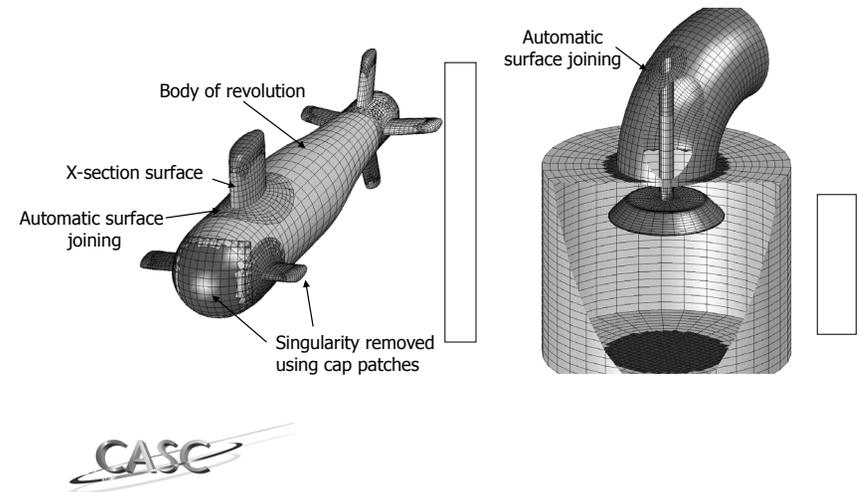


- Object-oriented tools for solving CFD problems in complex moving geometry
- Portable solution for serial and parallel environments using A++/P++ array class
- Adaptive mesh refinement capabilities
- Finite Difference and Finite Volume technology
- Incompressible, Nearly incompressible and Compressible Flow solvers

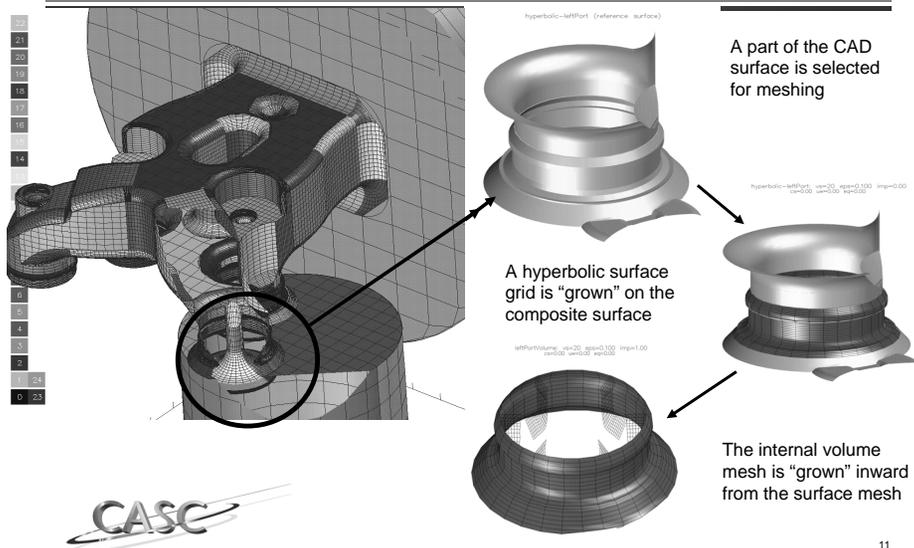
The overlapping grid construction approach is based on component assembly



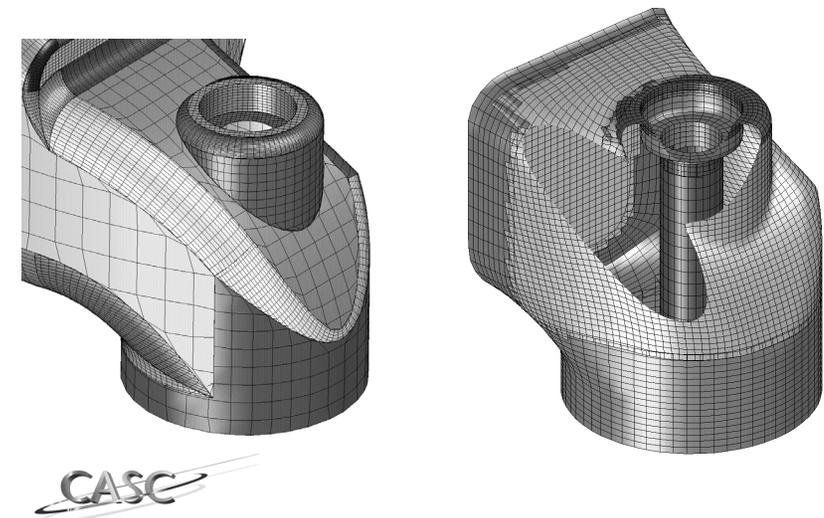
Overset Grid Generation Capabilities



For device-scale combustion simulations, grids must be constructed from CAD data



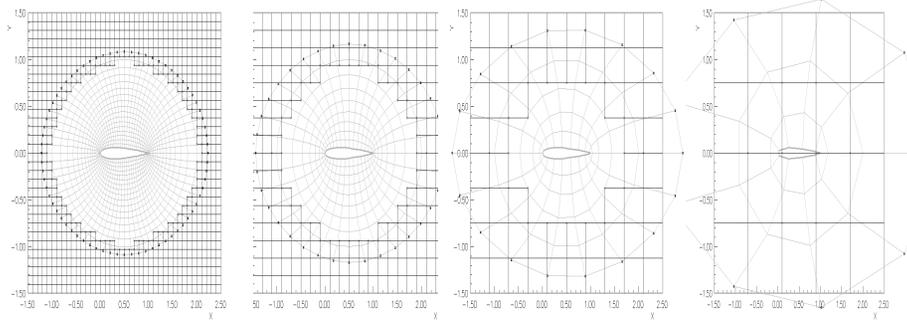
With Overture, CAD geometry is replaced with a surface mesh using hyperbolic mesh generation



Multigrid Solvers on Overlapping Grids

Automated construction of coarsenings

- an advantage of keeping structured grids in the solution process



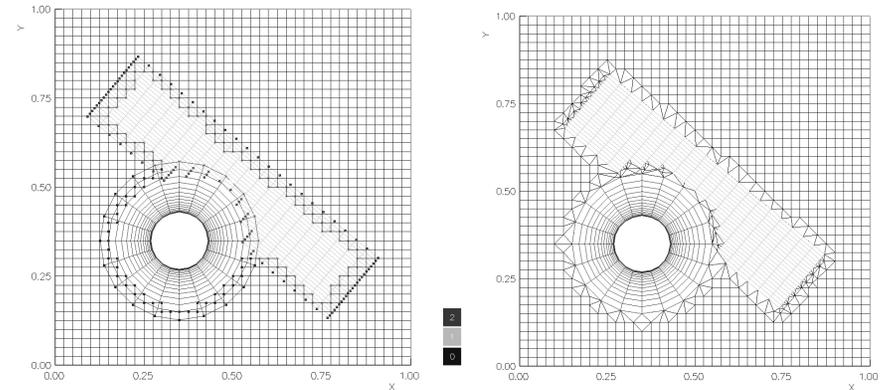
- Natural interpolation operators (AMG not a requirement)
- Structured Grids are CPU and memory efficient



Hybrid grid generation leverages *Ogen* hole-cutting algorithm

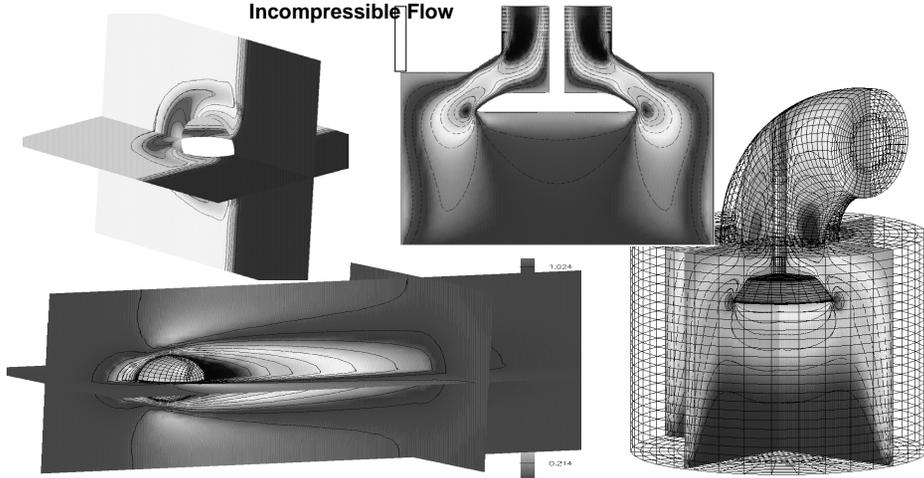
Overset mesh built with Overture

Hybrid Mesh

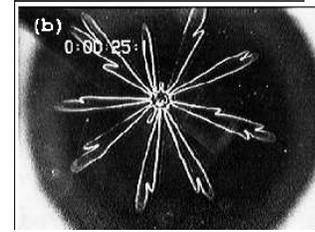


OVERBLOWN: CFD Solver

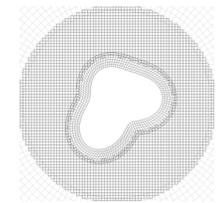
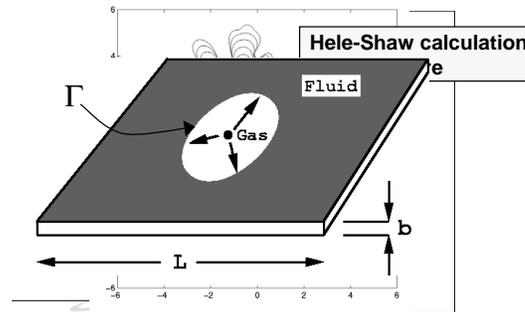
Geometry Independent Flow Solver using Overture Framework
Compressible Flow
Incompressible Flow



Moving Overlapping grids have been used to study free boundary problems



Non-Newtonian Hele-Shaw flow experiment

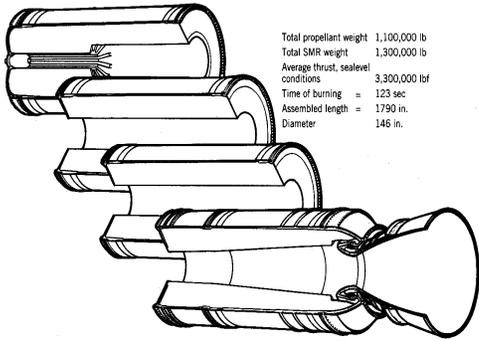


Computational Grid

Time = 0.00

Computational Model

SPACE SHUTTLE BOOSTER



Total propellant weight 1,100,000 lb
 Total SRM weight 1,300,000 lb
 Average thrust, sea-level conditions 3,300,000 lbf
 Time of burning = 123 sec
 Assembled length = 1790 in.
 Diameter = 146 in.

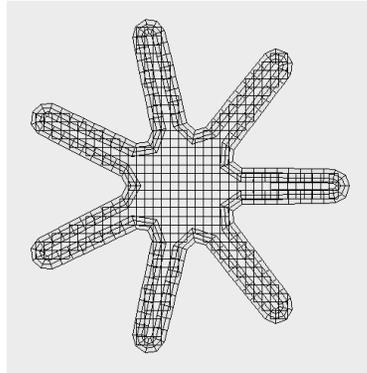
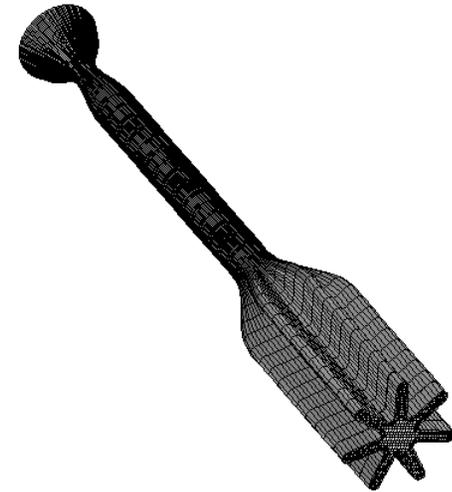


Fig. 14-2. Simplified diagram of the four segments of the Space Shuttle solid rocket motor. Details of the thrust vector actuating mechanisms or the ignition system are not shown. (Courtesy of NASA and Thiokol, Inc.)

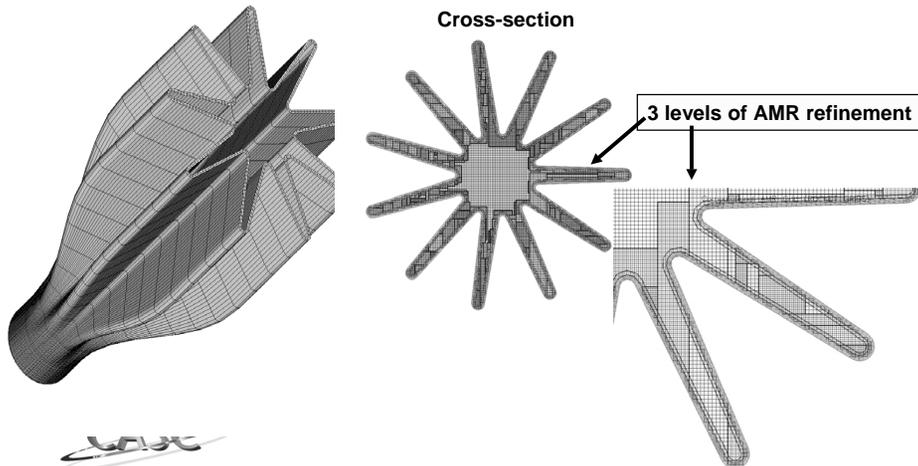


3D Rocket Booster Model



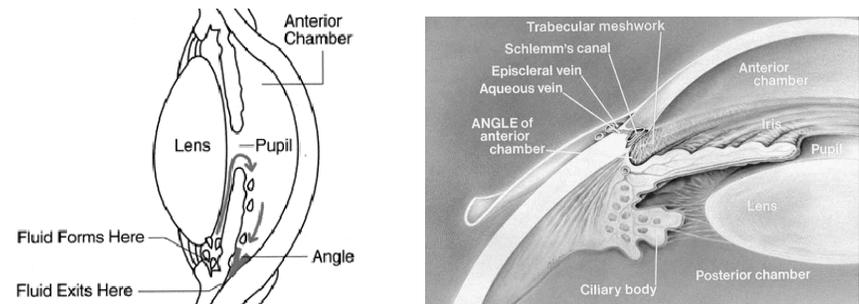
Solid Rocket Booster (SRB) Fuel Grain

3D Boundary Grid for Interior of SRB

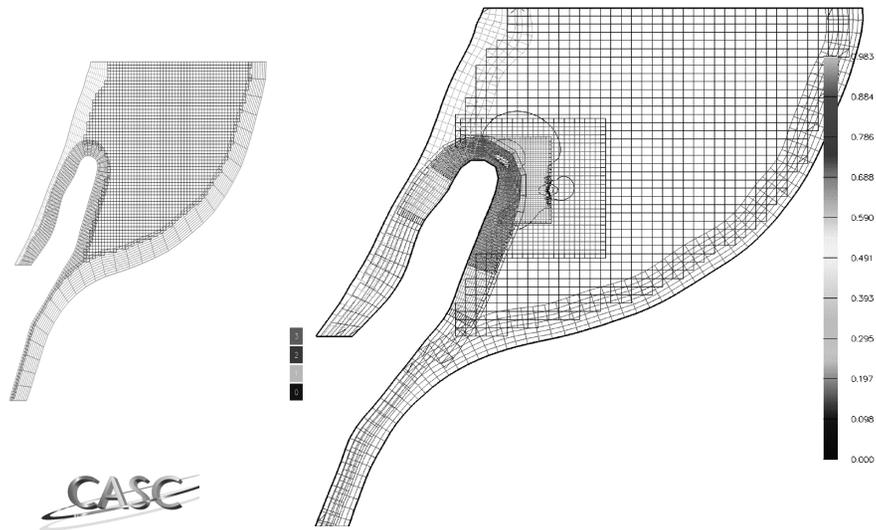


APPLICATION PROBLEM

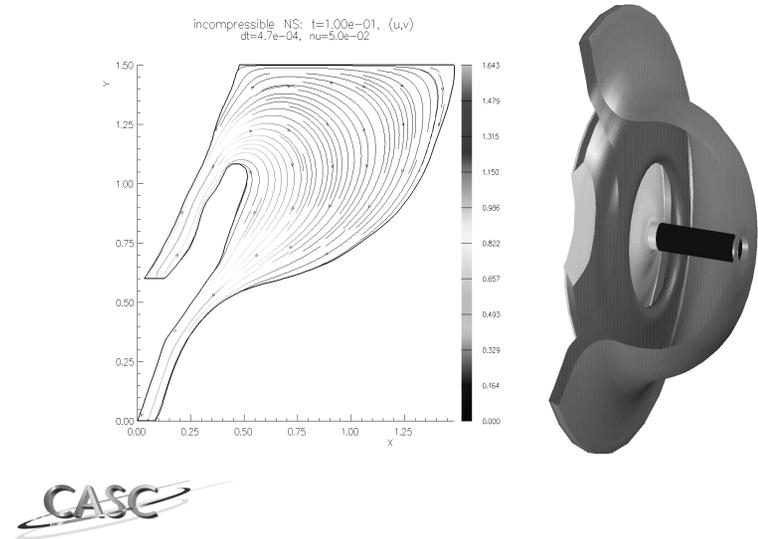
Fluid Flow in The Eye



Computational grid with different levels of refinement



Preliminary Computations and Computational Grids

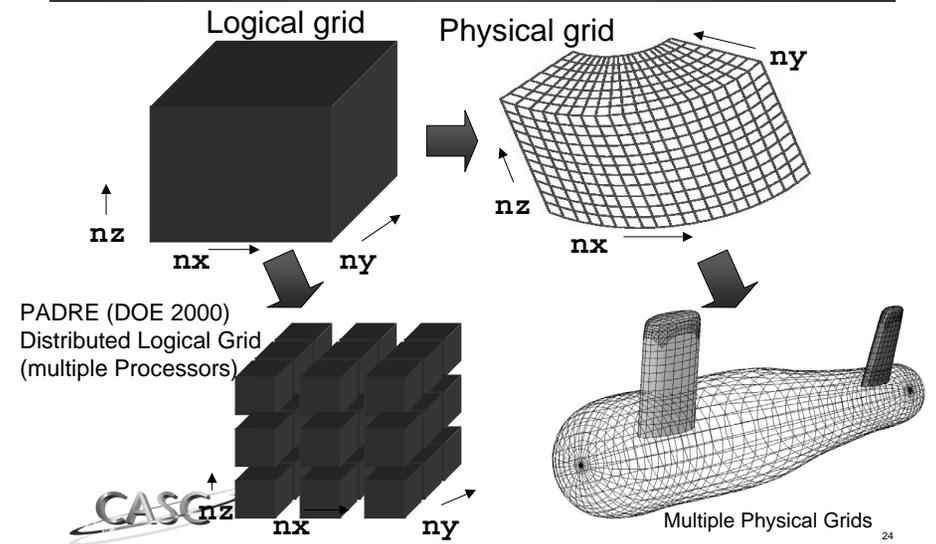


Overture

Program Interfaces



Grids and Data



The fundamental building block for the *Overture* framework is the P++ array class

Stencil operations on structured grids are naturally expressed in terms of array operations

Details of parallel implementation can be hidden from the user by the array class

Like F90 Index Triplet

Parallel communication occurs at the =

```

Index I(1,98),J(1,98);
FloatArray u(100,100);

// update stencil and communicate between processors
u(I,J) += .25*(u(I-1,J) + u(I+1,J) + u(I,J-1) + u(I,J+1));
    
```

Like F90 Arrays



25

In the *Overture* Framework, complex objects behave like built-in types...

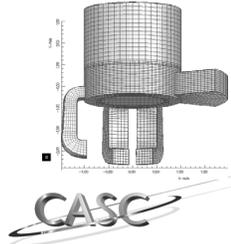
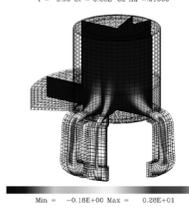
```

int i, j, k;
float a, b[10];

j = i + 10;
b[i] = b[i+1];

Index I,J;
CompositeGrid cg;
floatCompositeGridFunction u,v,w;
int grid;

w = u + v;
u[grid](I,J) = w[grid](I+1,J-1)
    
```

26

But in addition, more complex operations can be defined ...

```

Index I,J;
CompositeGrid cg;
floatCompositeGridFunction u,v,w;

w = u + v;
u[k](I,J) = w[k](I+1,J-1);

w = u.x(); // w = u_x
v = u.y(); // v = u_y
w = u.xx() + u.yy();
w = u.laplacian();
v = u.div();
    
```



27

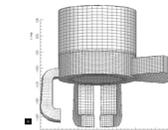
... and complex operations can be expressed with concise syntax

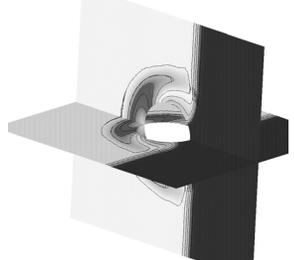
```

CompositeGrid grid;
// Database tools
readFromDatabaseFile (grid, filename);

// Make a grid function from a grid
floatCompositeGridFunction w(grid, cellCentered, 2);

// visualization tools
PlotStuff ps;
ps.plot (grid);
ps.contour (w);
    
```






28

At the highest level, *Overture* code looks like the underlying mathematics.

Mathematical expressions involving differential operators such as

$$\mathbf{u}_{new} = \mathbf{u} - \delta t \left((\mathbf{u} \cdot \nabla) \mathbf{u} - \nu \Delta \mathbf{u} \right)$$

are expressed concisely using the *Overture* operator classes.

This example advances a convection-diffusion equation on an overlapping grid. All grid-dependent and parallel details are hidden at this level.

```
uNew = u - dt*( u.convectiveDerivative() - nu*u.laplacian());
```



29

3D Incompressible Navier-Stokes

High Level Operations on Overset Grids

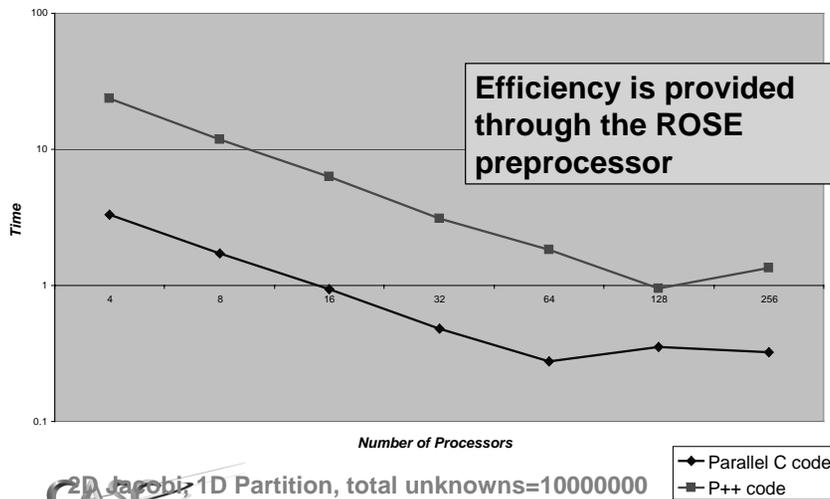
```
float dt=.0005, viscosity=.05;

for (int step=0; step < 100; step++)
{
  // ... forward Euler
  u += dt*( -u.convectiveDerivative() + viscosity*u.laplacian());
  u.applyBoundaryCondition (allVelocityComponents, dirichlet, wall);
  u.interpolate();
  // ... correct by enforcing incompressibility constraint
  u = projection.project (u);
  // ... visualize
  if (step % 10 == 0) ps.streamLines (u);
}
```



30

P++ Stencil Operations Scale Well on ASCI Blue Pacific



31

Compilers cannot optimize object interactions

Problem Statement

Object-oriented applications for scientific computing introduce abstractions customized to application domains but the compiler is unable to optimize these abstractions on parallel computer architectures

Solution

Semantics-Driven Optimization

Use what we know about the objects to drive the optimization



32

ROSE transforms high-level *Overture* statements into low-level code the compiler can optimize

P++ Code

```
Index I (1,n,1);
doubleArray Solution(n+1);
doubleArray RHS(n+1);
Solution(I) =
  ((h*h)*RHS(I) + Solution(I+1) + Solution(I-1)) / 0.5;
```

Automated ROSE Transformation

```
Index I (1,n,1);
doubleArray RHS(n+1);
doubleArray Solution(n+1);
double* restrict RHS_data = RHS.getDataPointer();
double* restrict Solution_data = Solution.getDataPointer();

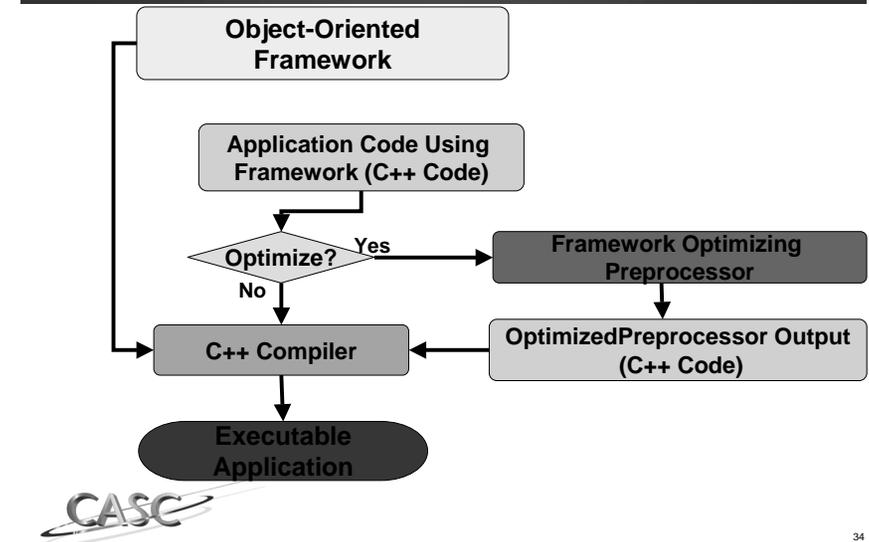
int I_index = 0;
int I_base = I.getBase();
int I_bound = I.getBound();

for (I_index = I_base; I_index < I_bound; I_index++)
  Solution_data[I_index] = ((h*h)*RHS_data[I_index] +
    Solution_data[I_index+1] +
    Solution_data[I_index-1]) / 0.5;
```

CASC

33

Preprocessor is Optional



CASC

34

ROSE: Optimizing preprocessor

- **Optional Optimizing Preprocessor**
- **Automates sophisticated transformations**
 - **Translates array syntax to optimized C-loops**
 - Fusion of expressions
 - Fusion of statements
 - **Cache based transformations for Super-FORTRAN Performance**
 - **Message Passing Latency Hiding**
 - **Eliminates Array Class Overhead**
- **Uses Edison Design Group (EDG) C++ Front-End**
- **Uses SAGE II C++ source code restructuring tool internally**

CASC

35

ROSE Preprocessors

ROSE Project Objective: Better performance for OO applications

How ROSE Works: C++ source → C++ source w/transformations

- 1) Parse C++ application to form a C++ program tree
(nodes in the tree represent elements of C++ grammar)
- 2) Define grammars (*the elements of a computer language*) specific to an OO application or library
- 3) Parse C++ program tree to build program trees for each grammar
- 4) Use grammars to recognize high level objects and expressions
- 5) Edit the C++ program tree to build in transformations
- 6) Unparse the C++ program tree to generate C++ code

CASC

36

Performance of Object-Oriented Abstractions

- Use of high-level abstractions simplifies development
- Parallel and Single CPU Performance is also important
- ROSE: tool for building optimizing preprocessors
 - Uses modified version of SAGE II (based on EDG)
 - ROSETTA: builds grammars
 - Source code generator for modified SAGE II (C++ Grammar)
 - Generates additional higher-level grammars based on SAGE II
 - Preprocessor source code is automatically generated
 - User specifies:
 - 1) Targets to optimize (user defined objects)
 - 2) Transformations
 - Optimized application code (C and C++) is generated
 - output code is formatted the same as input code



37

Purpose of Grammars

Recognize the use of Abstractions within an Application's Program Tree

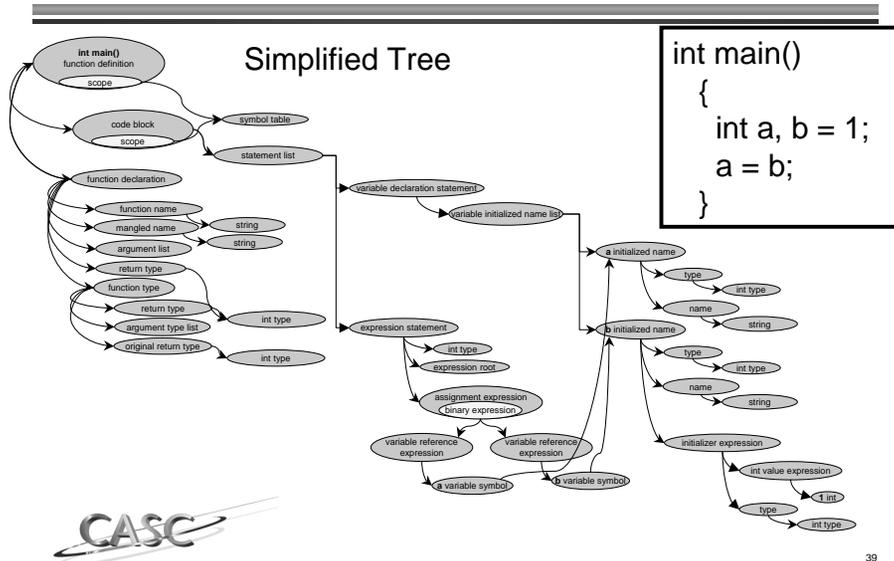
Reduce the Complexity of the Program Tree

Grammars Identify Statements, Expressions, and Types associated with a target abstraction and those not associated with abstractions (defined to be other Statements, other Expressions, other Types)



38

Application Program Trees



39

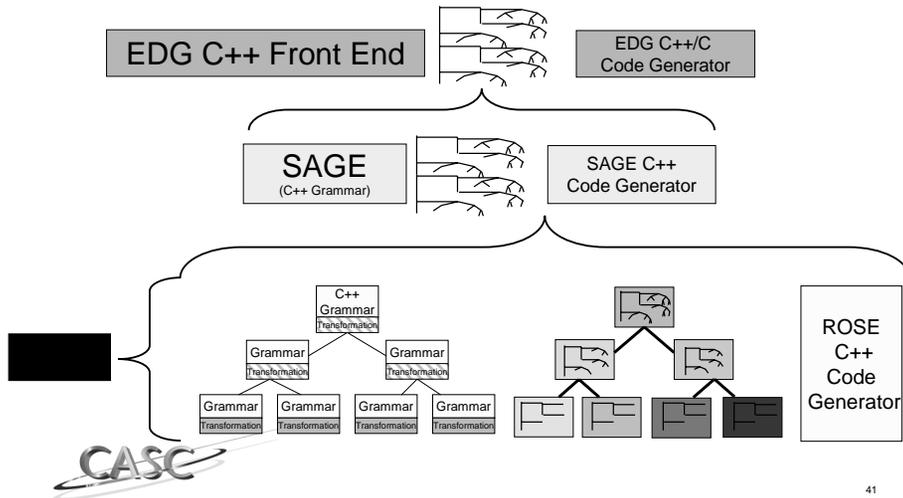
Where to apply optimizations (finding our way in the forest)

- Looking at the C++ Grammar is like reading a map with an electron microscope (everything is there but you still can't see where you are)
- Abstractions help condense the program tree
- The semantics of the abstractions help us read the condensed program tree



40

Relationship of ROSE to Other work



41

Meta Program Level

Program Level:

variables are *objects with a defined type*

variables *define* locations in memory

```
int i;
floatArray X;
list<X> listOfX;
```

Meta Program Level:

variables are *Grammars, Terminals, and Non-Terminals*
 Grammars, Terminals, and Non-Terminals *define* types

variables *define* types

```
Grammar CxxGrammar;
Grammar XGrammar;
Terminal ClassType = XGrammar.getTerminal("ClassType");
Nonterminal X = ClassType.buildNewType("X");
XGrammar.addTerminal(X);
```



42

Output Must Resemble Input

Can't expect users to trust a tool for which the output does not resemble the input

Machine generated source code can be:

- Ugly
- Impossible to debug
- Difficult to trust

ROSE Unparser generates friendly output:

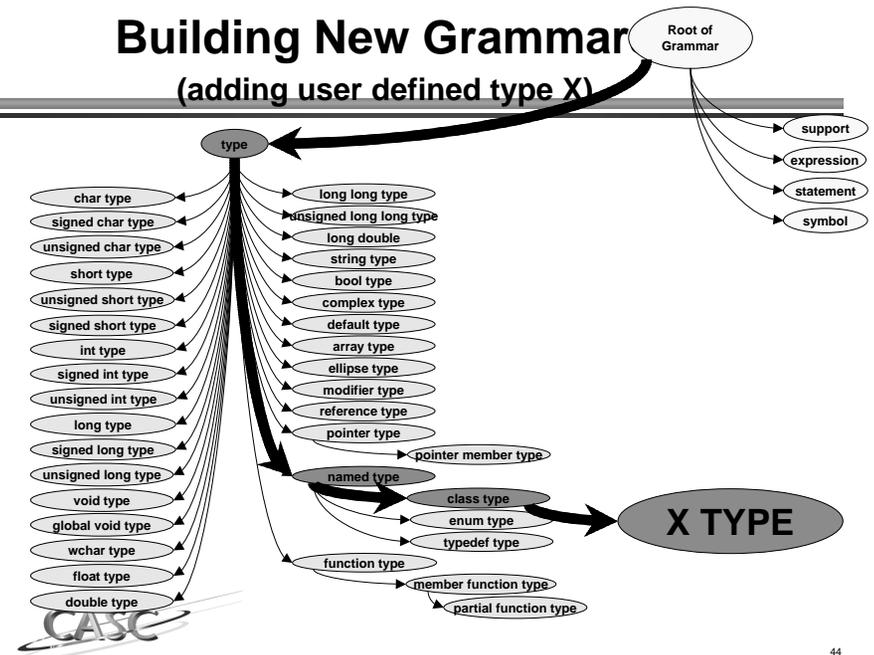
- Preserves variable names
- Inserts references to line numbers in original source (for debuggers)
- Preserves all preprocessor control structure (#include, #ifdefs, etc.)
- Preserves users comments
- Lots of options to control unparsing



43

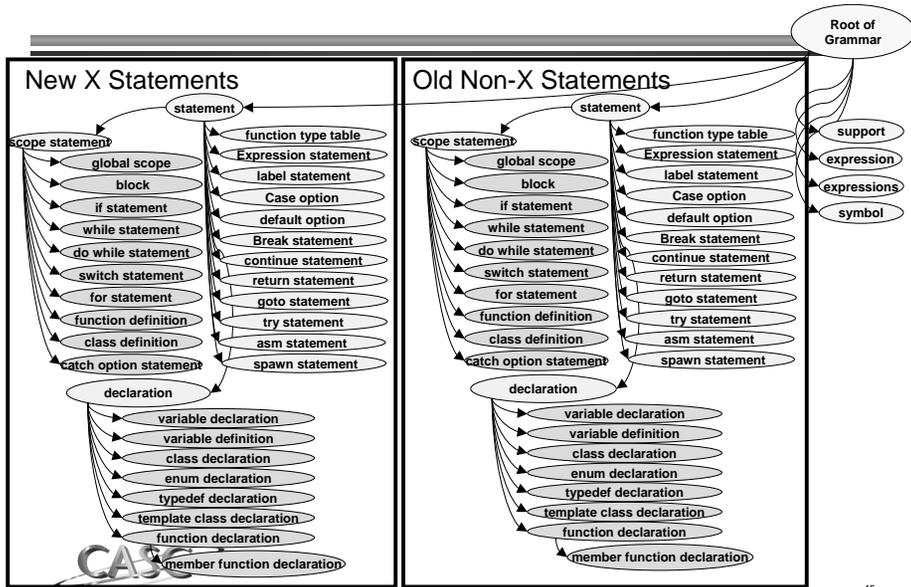
Building New Grammar

(adding user defined type X)

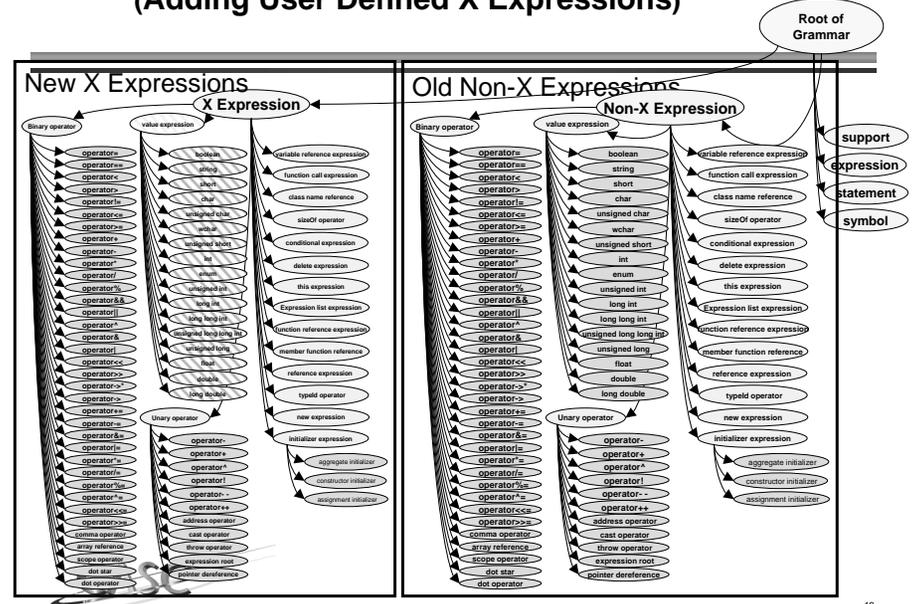


44

C++ Grammar (statements)



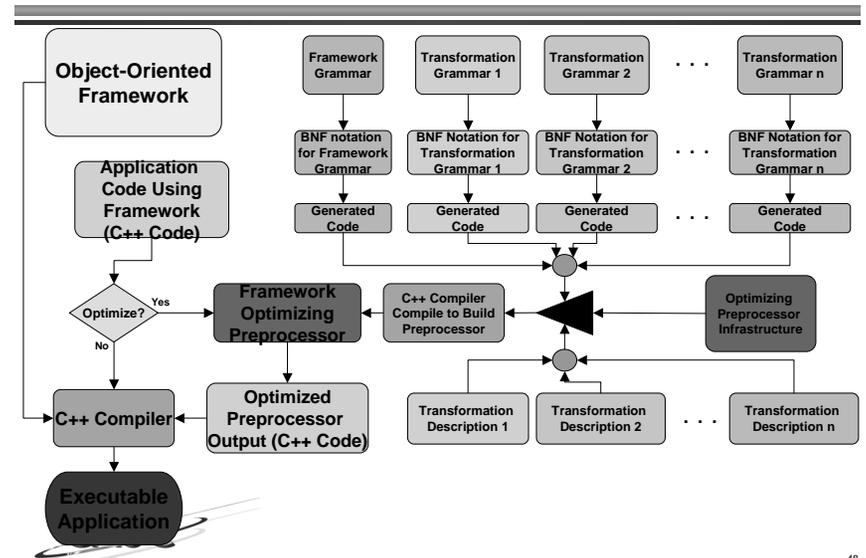
Building New Grammars (Adding User Defined X Expressions)



Unparsed Example

Original Input C++ Source code	Unparsed Output C++ Source code
<pre>#include "A++.h" #include "../include/ROSE_TRANSFORMATION_SOURCE.h" #include <iostream.h> int main() { int x = 4; //these comments are difficult for (int i = 0; i < 10; i++) { while (x) { x = x + 1; if (false) { x++; x = 7+x; } else { x = x - 1; x--; } // comments! x++; x += 1; } } return 0; }</pre>	<pre>#include "A++.h" #include "../include/ROSE_TRANSFORMATION_SOURCE.h" #include <iostream.h> int main(){ int x=4; //these comments are difficult for (int i=0; i < 10; i++){ while(x){ x = x + 1; if (FALSE){ x++; x = 7 + x; } else { x = x - 1; x--; } // comments! x++; x += 1; } } return 0; }</pre>

Research Approach: Preprocessor Design Flowchart



ROSE Transformation: A++ Code

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

int main()
{
    int size = 10;
    doubleArray A(size);
    doubleArray B(size);
    Range I(1, size-2);

    A(I) = ( B(I+1) + B(I-1) ) * 2.0;

    printf ("Program Terminated Normally! \n");
    return 0;
}
```



49

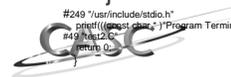
ROSE Transformation: After ROSE

```
#include <A++.h>

#4 "test2.C"
int main()
{
    auto int size=10;
    auto double gamma=2;
    auto doubleArray A(size);
    #9 "test2.C"
    auto doubleArray B(size);
    #10 "test2.C"
    auto Range I(1,size-2);
    #11 "test2.C"
    auto Range J(1,size-2);

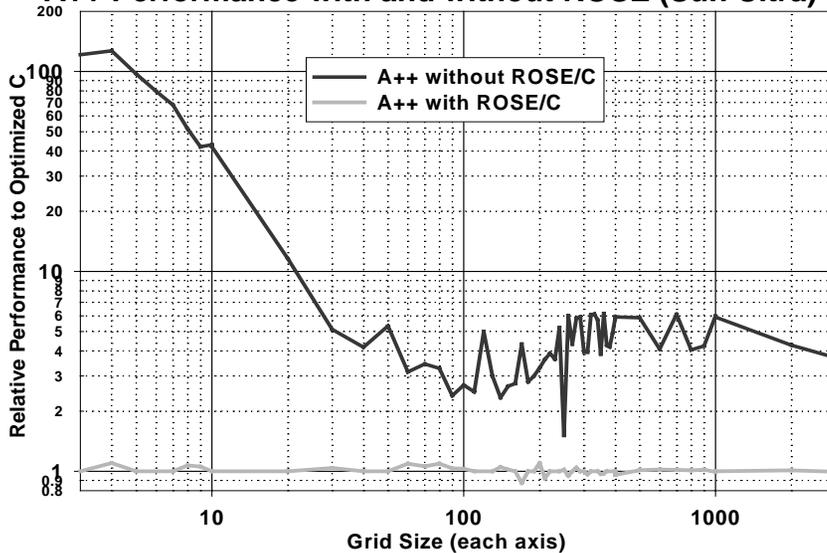
    #13 "test2.C"
    {
        // Transformation for: A(I) = B(I-1) + B(I+1);
        int rose_index=0;
        double * restrict A_rose_pointer = (A . getDataPointer());
        double * restrict B_rose_pointer = (B . getDataPointer());
        const int base_1D_0 = (I . getBase());
        const int bound_1D_0 = (I . getBound());
        for(rose_index = base_1D_0; rose_index <= bound_1D_0; rose_index++)
        {
            A_rose_pointer[rose_index] =
                (B_rose_pointer[(rose_index + 1)] + B_rose_pointer[(rose_index - 1)]) * 2;
        }
    }

    #249 "usr/include/stdo.h"
    printf(((const char*) "Program Terminated Normally! \n"));
    #47 "test2.C"
    return 0;
}
```



50

A++ Performance with and without ROSE (Sun Ultra)



Future Work

- Improved Performance (ROSE)
- Parallelism for Overlapping Grid Problems
- More Solvers
 - Access to more external solver libraries (elliptic methods)
 - Improved integration with Petsc



52