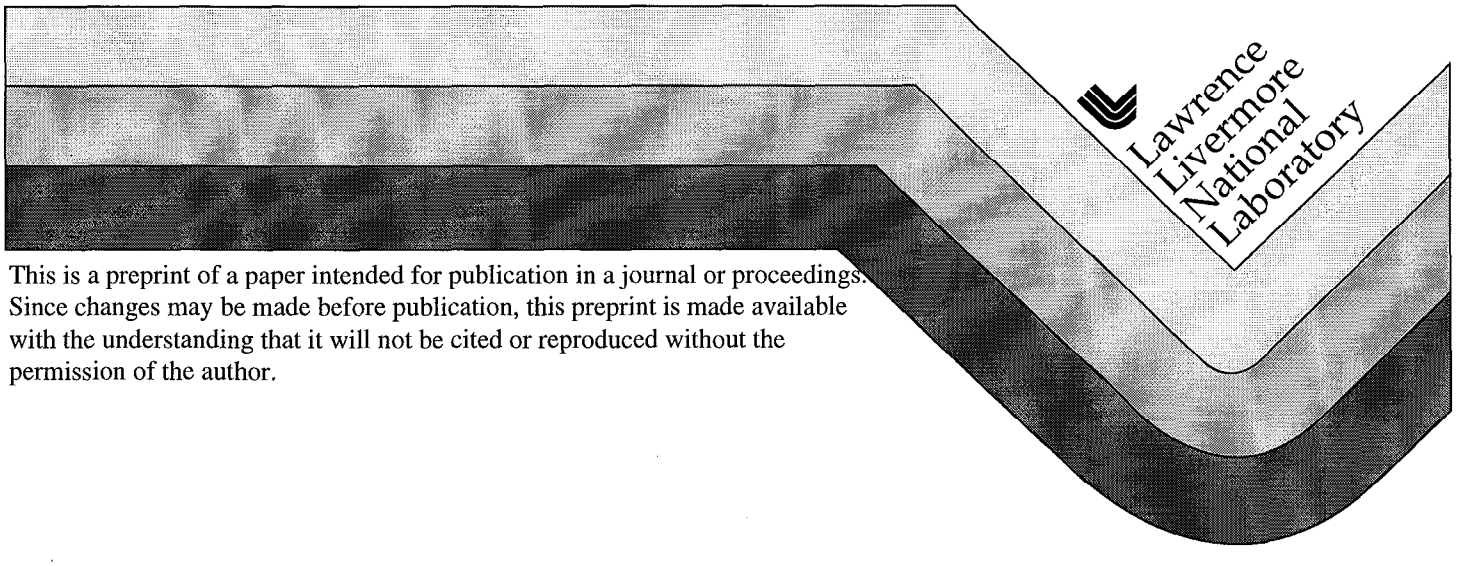
UCRL-JC-132017
Preprint

Overture: An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids

D. L. Brown,
W. D. Henshaw,
D. J. Quinlan

This paper was prepared for submittal to
Society for Industrial & Applied Mathematics
Workshop on Object Oriented Methods for
Interoperable Scientific and Engineering Computing
Yorktown Heights, NY
October 21-23, 1998

September 22, 1998



Overture : An Object-Oriented Framework for Solving Partial Differential Equations on Overlapping Grids*

David L. Brown† William D. Henshaw† Daniel J. Quinlan†

Abstract

The **Overture** framework is an object-oriented environment for solving partial differential equations in two and three space dimensions. It is a collection of C++ libraries that enables the use of finite difference and finite volume methods at a level that hides the details of the associated data structures. **Overture** can be used to solve problems in complicated, moving geometries using the method of overlapping grids. It merges geometry, grid generation, difference operators, boundary conditions, data-base access and graphics into an easy to use high level interface.

1 Introduction

The design of **Overture** has evolved over the past 15 years or so from the Fortran based CMPGRD[7] environment to the current C++ version [5, 6]. Although the Fortran implementation was used for complicated three-dimensional adaptive and moving grid computations, the programs were difficult to write and maintain. **Overture** was designed to have at least all the functionality of the Fortran code but to be as easy as possible to use; indeed, an entire PDE solver on an overlapping grid can be written on a single page (see section 2.7).

A composite overlapping grid consists of a set of logically rectangular curvilinear grids that overlap where they meet and together are used to describe a computational region of arbitrary complexity. This method has been used successfully over the last decade and a half, primarily to solve problems involving fluid flow in complex, often dynamically moving, geometries [2, 3, 10, 11, 23]. There are a number of reasons why the design and implementation of a flexible overlapping grid solver is quite complex. First of all the data structures required to hold geometry information (vertices, Jacobian, boundary normals, interpolation data, etc.) and the information for moving grids, adaptive grids and multigrid are extensive. Secondly, complicated partial differential equations are solved using sophisticated numerical algorithms; for example, a current application in our group involves low Mach-number combustion with many reacting species. In addition, techniques such as block structured adaptive-mesh-refinement [1, 2, 20] may be used to locally increase grid resolution and increase overall computational efficiency. If the simulation is to run on a parallel architecture, there are correspondingly more issues involved in writing the code. The net result of the data structures, advanced algorithms, and modern architectures is a PDE solver code that is an extremely complex system. Successfully writing, debugging, modifying and maintaining software that implements this system is a

*http://www.llnl.gov/casc/Overture

†Centre for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA.

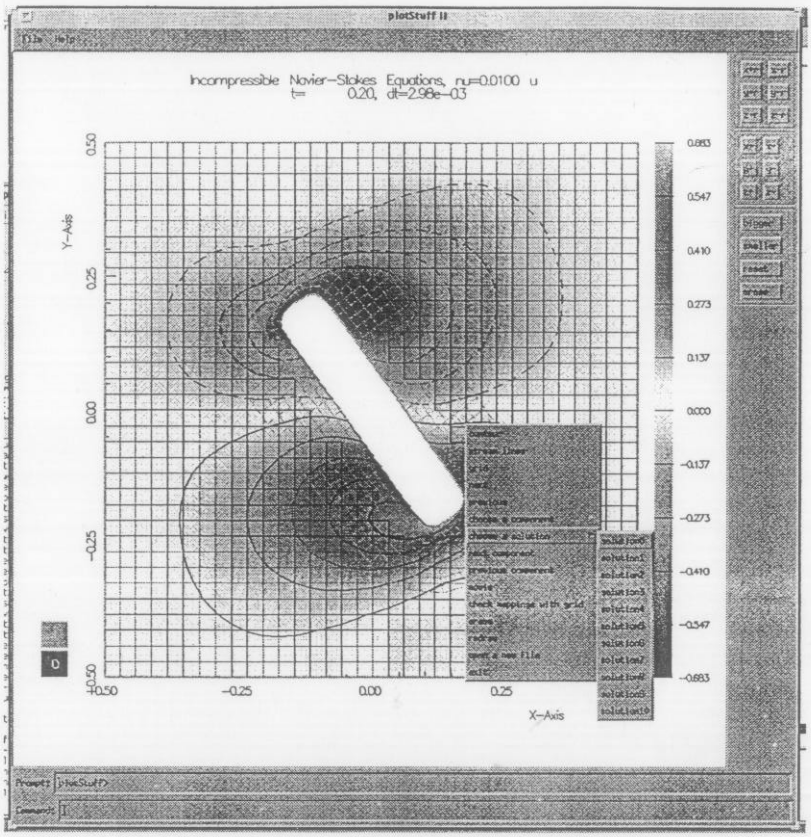


FIG. 1. *Displaying results from a moving grid computation using the* **Overture** *framework.*

daunting if not impossible task using a traditional structured programming approach and procedural languages such as Fortran or C. This has provided the primary motivation for the development of the **Overture** framework in C++.

There are a number of other very interesting projects developing scientific object-oriented frameworks. These include the SAMRAI framework for structured adaptive mesh refinement [22], PETSc (the Portable Extensible Toolkit for Scientific Computation) [18], POOMA (Parallel Object Oriented Methods and Applications)[19] and Diffpack[9].

2 The Overture framework

The main class categories that make up **Overture** are as follows:

- **Arrays** [21]: describe multidimensional arrays using A++/P++. A++ provides the serial array objects, and P++ provides the distribution and interpretation of communication required for their data parallel execution.

- **Mappings** [14]: define transformations such as curves, surfaces, areas, and volumes. These are used to represent the geometry of the computational domain.

- **Grids** [8, 13]: define a discrete representation of a mapping or mappings. These include single grids, and collections of grids; in particular composite overlapping grids.

- **Grid functions** [13]: storage of solution values, such as density, velocity, pressure, defined at each point on the grid(s).

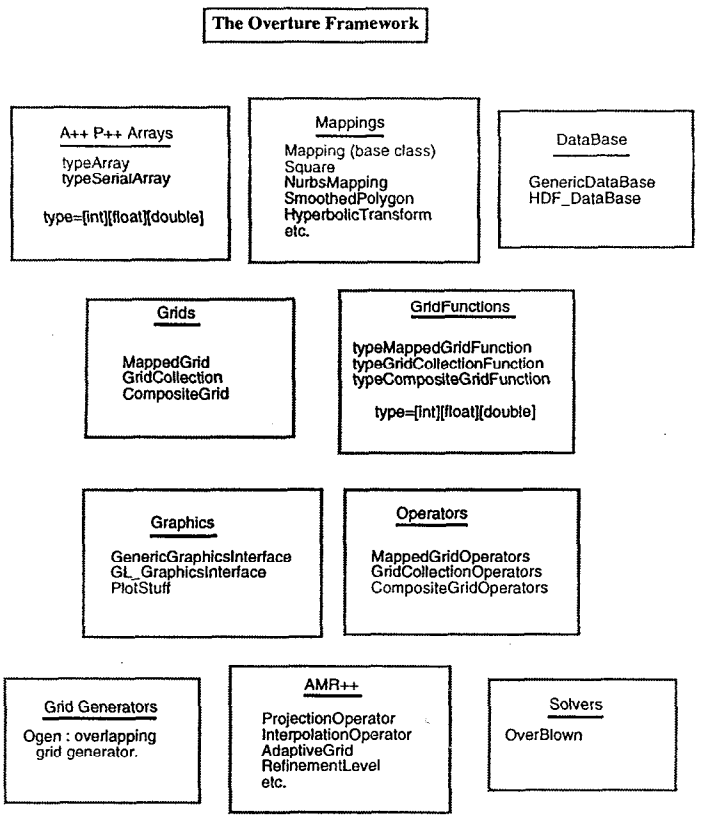


FIG. 2. *An overview of the* **Overture** *classes*

- **Operators** [4, 12]: provide discrete representations of differential operators and boundary conditions
- **Plotting** [17]: a high-level interface based on OpenGL allows for plotting **Overture** objects.
- **Adaptive Mesh Refinement:** The AMR++ library for patch based refinement is described in section 2.6.

Solvers for partial differential equations are written using the above classes.

2.1 Array operations

A++ and P++ [21] are array class libraries for performing array operations in C++ in serial and parallel environments, respectively. The use of P++ is the principle mechanism by which the **Overture** framework operates in parallel, there is little code in **Overture** outside of P++ which is specific to parallel execution.

A++ is a *serial* array class library similar to FORTRAN 90 in syntax, but not requiring any modification to the C++ compiler or language. A++ provides an object-oriented array abstraction specifically well suited to large scale numerical computation. It provides efficient use of multidimensional array objects which serves to both simplify the development of numerical software and provide a basis for the development of parallel array abstractions. P++ is the *parallel* array class library and shares an identical interface to A++, effectively allowing A++ serial applications to be recompiled using P++ and thus run in parallel. This provides a simple and elegant mechanism that allows serial code to be reused in the parallel environment.

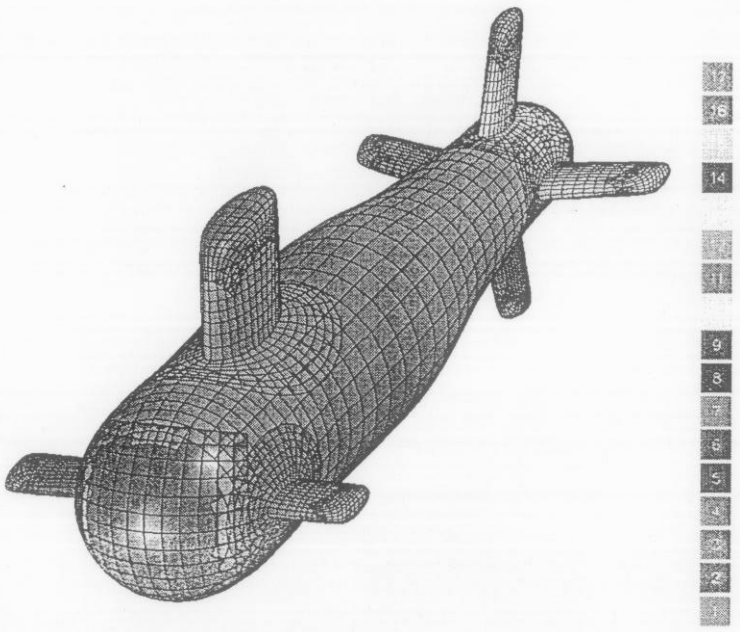


FIG. 3. *Overlapping grid for a submarine.*

P++ provides a data parallel implementation of the array syntax represented by the A++ array class library. To this extent it shares a lot of commonality with FORTRAN 90 array syntax and the HPF programming model. However, in contrast to HPF, P++ provides a more general mechanism for the distribution of arrays and greater control as required for the multiple grid applications represented by both the Overlapping Grid model and the Adaptive Mesh Refinement (AMR) model. Additionally, current work is addressing the addition of task parallelism as required for parallel adaptive mesh refinement.

Here is a simple example code segment that solves 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.

```
  // Solve u_xx + u_yy = f by a Jacobi Iteration
Range R(0,n)                          // ... define a range of indices: 0,1,2,...,n
floatArray u(R,R), f(R,R)             // ... declare two two-dimensional arrays
f = 1.; u = 0.; h = 1./n;             // ... initialize arrays and parameters
Range I(1,n-1), J(1,n-1);             // ... define ranges for the interior

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)); // ... data parallel
```

2.2 Mappings, grids and grid generation

The geometry of the computational domain is defined by a set of mappings, one mapping for each grid. In general, a mapping defines a transformation from R^n to R^m. In particular, mappings can define lines, curves, surfaces, volumes, rotations, coordinate stretchings, etc.[14]. The base class `Mapping` contains the data and functions that apply to all mappings. Specific types of mappings are derived from this base class. Mappings contain a variety of

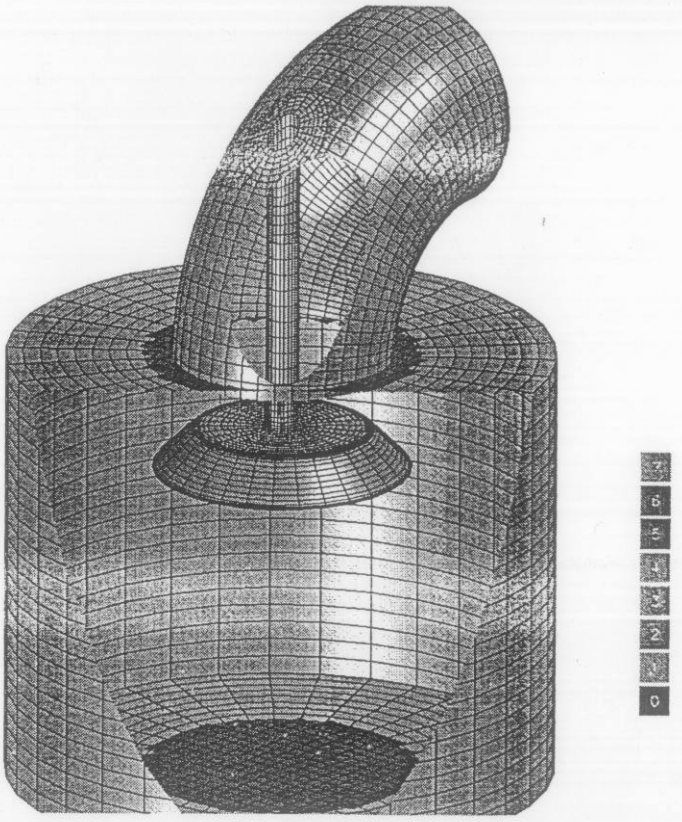


FIG. 4. *Overlapping grid for a valve, port and cylinder.*

information and functions that can be useful for grid generators and solvers. For example, mappings contain information about their domain space, range space, boundary conditions and singularities. Mappings are easily composed, allowing coordinate stretching, rotations, translations, bodies of revolution, etc. The inverse of a mapping is always defined, either analytically or by discrete approximation.

Grids define a discrete representation of a mapping. There are several main grid classes [8, 13]. The `MappedGrid` class defines a grid for a single mapping that contains, among other things, a mapping and a mask array for cut-out regions. The `GridCollection` class defines a collection of `MappedGrid`'s. The `CompositeGrid` class defines a valid overlapping grid, which is essentially a `GridCollection` plus interpolation information. Grids contain many geometry arrays such as grid points, Jacobians , normal vectors, face areas and cell volumes.

Overture has support for the creation of overlapping grids for complicated geometries. The overlapping grids shown in figures (3) and (4), for example, were created entirely with **Overture** tools. Current work involves the creation of grids for geometries imported from computer-aided-design packages. **Overture** has a sophisticated overlapping grid generator, Ogen [15], that automatically determines how component grids overlap one another, computing which points to interpolate and removing points that lie outside the computational domain.

2.3 Grid functions

Grid functions represent solution values at each point on a grid or grid-collection. There is a grid function class (of `float`'s, `int`'s or `double`'s) corresponding to each type

of grid [13]. So, for example, a MappedGridFunction lives on a MappedGrid and a CompositeGridFunction lives on a CompositeGrid. Grid functions are defined with up to three coordinate indices (i.e. up to three space dimensions) and up to five component indices (i.e. they can be scalars, vectors, matrices, 3-tensors,...). Since they are derived from A++ arrays, all of the array operations are defined. In the following example, a grid function is made and assigned values at all points on the grid.

```
SphereMapping sphere;                          // mapping for a spherical shell
MappedGrid mg(sphere);                         // grid for a spherical shell
mg.update();                                   // compute some geometry arrays
Range all;                                     // used as a place marker for coordinates
floatMappedGridFunction u(mg,all,all,all,2); // define a 3D grid function with 2 components
Index I1,I2,I3;
getIndex(mg.dimension,I1,I2,I3);               // define indices for all grid points
// first component of u = sin(x)*cos(y) :
u(I1,I2,I3,0)=sin(mg.vertex(I1,I2,I3,0))*cos(mg.vertex(I1,I2,I3,1));
```

Notice that when the floatMappedGridFunction is declared, the number of grid points does not have to be specified since this information is contained in the MappedGrid.

2.4 Differential operators

Operators define discrete approximations to differential operators and boundary conditions for grid functions. Many different types of approximations can be used. For example, the class MappedGridOperators [12] defines finite-difference style operators, while the class MappedGridFiniteVolumeOperators [4] defines finite-volume style operators. An operator class for incompressible flow Godunov methods has also been implemented. The Projection class computes the divergence-free part of a velocity function and is used in some of our incompressible flow codes. Here is an example using one of the operator classes:

```
...
 MappedGrid mg(sphere);
 MappedGridOperators op(mg);                   // Define operators for a MappedGrid
 floatMappedGridFunction u(mg), v(mg), coeff;
 u.setOperators(op);                           // associate operators with a grid function
 ... assign u here ...
 v=u.x();                                      // take the x derivative of u
 // form the matrix for the discrete Laplacian:
 coeff=u.xxCoefficients()+u.yyCoefficients()+u.zzCoefficients();
 // another way to form the matrix for the Laplacian:
 coeff=u.laplacianCoefficients();
```

The result of the statement u.x() is a grid function containing the x-derivative of u (on a curvilinear grid). The last line in the above example generates a discrete representation of the Laplacian operator (stored as a 27 point stencil in the second-order accurate case). This coeff grid function can be passed to a sparse matrix solver, using for example the Oges class in **Overture** [16].

2.5 Boundary Conditions

The programming model for boundary conditions is to use ghost points (instead of one-sided difference approximations). A library of elementary boundary conditions such as Dirichlet, Neumann, mixed, extrapolation, etc. has been defined. Solvers define more complicated boundary conditions in terms of these elementary ones. The interface is quite simple, as can be seen in the following code fragment that sets boundary conditions for the incompressible Navier-Stokes equations on a no-slip wall.

```
// no-slip wall:
//  (1)  (u,v,w) = 0.
//  (2)  extrapolate ghostline values
//  (3)  set the divergence of (u,v,w) equal to zero

const int wall=5;  // This value is associated with a boundary condition
                   // value that is given to each side of each grid

Range V(0,numberOfDimensions-1);  // velocity components

u.applyBoundaryCondition(V,dirichlet,            wall,0.);
u.applyBoundaryCondition(V,extrapolate,          wall);
u.applyBoundaryCondition(V,generalizedDivergence,wall);
u.finishBoundaryConditions();
}
```

2.6 Adaptive mesh refinement

Adaptive mesh refinement is the process of permitting local grids to be added to the computational domain and thus adaptively tailoring the resolution of the computational grid. The block-structured AMR algorithm implemented in **Overture** provides such support for both simple problems with a single underlying grid, and problems that use the composite overlapping grid method. The AMR algorithm itself uses the multiple grid functionality provided by the basic **Overture** classes in an essential way. AMR results is greater computational efficiency but is difficult to support. AMR++ is a library within the **Overture** framework which builds on top of the previously mentioned components and provides support for **Overture** applications requiring adaptive mesh refinement. AMR++ is current work being developed and supports the adaptive regridding, transfer of data between adaptive refinement levels, parent/child/sibling operations between local refinement levels, and includes parallel AMR support. AMR++ is a parallel adaptive mesh refinement library because it is uses classes which derive their parallel support from the A++/P++ array class library.

2.7 Writing PDE solvers

This example 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 an overlapping grid.

```
int main()
{
  CompositeGrid cg;                          // create a composite grid
  getFromADataBaseFile(cg,"myGrid.hdf");     // read the grid in
  floatCompositeGridFunction u(cg);          // create a grid function
  u=1.;                                      // assign initial conditions
  CompositeGridOperators op(cg);             // create operators
  u.setOperators(cg);
  PlotStuff ps;                              // make an object for plotting
  // --- 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()) );
```

```
* Choose the overlapping
grid: stir.hdf
stir.show
incompressibleNavierStokes
turn off twilight zone
project initial conditions
turn on moving grids
specify grids to move
  stir
  rotate
  0. 0. 0.
  .5
done
choose grids for implicit
  all=explicit
  stir=implicit
done
pde parameters
  nu
  .01
done
boundary conditions
  all=noSlipWall
done
initial conditions
  uniform flow
  p=1.
final time (tf=)
  .5
times to plot (tp=)
  .025
```

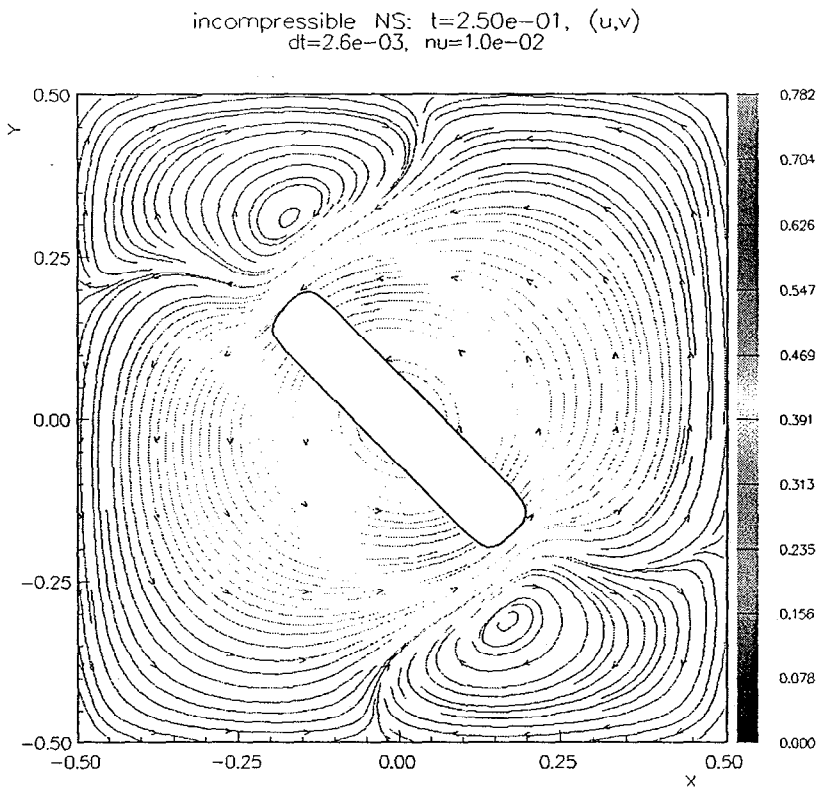


FIG. 5. *On the left is a sample script command file for running the flow solver OverBlown. On the right is the result from the computation of incompressible flow around a rotating stirring stick.*

```
    t+=dt;
    u.interpolate();                          // interpolate overlapping boundaries
    // apply the BC u=0 on all boundaries
    u.applyBoundaryCondition(0,dirichlet,allBoundaries,0.);
    u.finishBoundaryConditions();
    ps.contour(u);                            // plot contours of the solution
  }
  return 0;
}
```

The `PlotStuff` object is used to interactively plot contours of the solution at each time step[17].

Overture can be easily used by the numerical analyst to develop PDE solvers for overlapping grids. Within our research group we are also developing sophisticated PDE solvers for various applications that we hope to distribute in the near future. For example, the **OverBlown** Navier-Stokes solver is being developed for solving the Navier-Stokes equations at different Mach numbers. Figure (5) shows streamlines of the solution to the incompressible Navier-Stokes equations around a rotating stirring stick and the script file that was used to run OverBlown. Figure (6) shows results of a three-dimensional computation from OverBlown.

3 Performance

Overture makes extensive use of the A++/P++ array class library that supports array operations on both serial and parallel machines. In the current implementation, the array

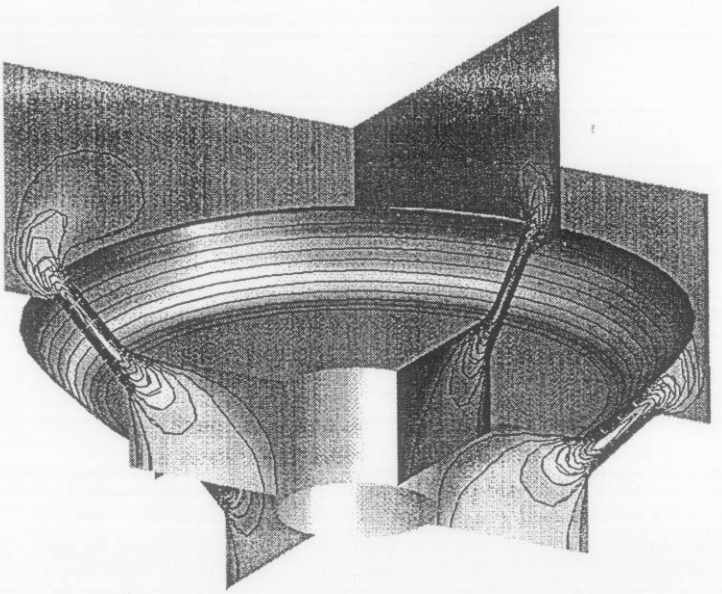


FIG. 6. *Incompressible flow through a three-dimensional valve.*

operations can be 2-8 times slower that Fortran 77 on some cache based architectures. However, preliminary results from the ROSE preprocessor project indicate that full Fortran 77 performance can be recovered from the array statements since the preprocessor can replace the high level operations by multi-dimensional loops. It has also been shown that a common array operation of an array statement inside a loop (such as a Jacobi iteration) can actually be made to run over twice as fast as the standard Fortran 77 implementation. These preliminary results show that high level implementations may actually have a greater practical potential for performance gains than a lower level approach. It is much easier to have a preprocessor that understands the semantics of a class interpret high-level statements and produce (perhaps lengthly) optimized code that to write this same code in many places through-out a long program. For further details see the article on *Optimizations for Parallel Object-Oriented Frameworks* in this proceedings.

4 Software availability

The **Overture** framework and documentation is available for public distribution from the web site, `http://www.llnl.gov/casc/Overture`.

References

[1] M. J. Berger and P. Colella, *Local adaptive mesh refinement for shock hydrodynamics*, J. Comp. Phys., 82 (1989), pp. 64–84.

[2] K. D. Brislawn, D. L. Brown, G. Chesshire, and J. S. Saltzman, *Adaptive composite overlapping grids for hyperbolic conservation laws*, LANL Unclassified Report 95-257, Los Alamos National Laboratory, 1995.

[3] D. L. Brown, *An unsplit Godunov method for systems of conservation laws on curvilinear overlapping grids*, Math. Comput. Modelling, 20 (1994), pp. 29–48.

[4] ——, *Classes for finite volume operators and projection operators*, LANL unclassified report 96-3470, Los Alamos National Laboratory, 1996.

[5] D. L. Brown, Geoffrey S. Chesshire, William D. Henshaw and Daniel J. Quinlan, **Overture** : *An Object Oriented Software System for Solving Partial Differential Equations in Serial and Parallel Environments*, Proceedings of the Eight SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[6] , D. L. Brown, William D. Henshaw and Daniel J. Quinlan, **Overture** : *An Object Oriented Framework for Solving Partial Differential Equations*, Scientific Computing in Object-Oriented Parallel Environments, Springer Lecture Notes in Computer Science, 1343, 1997.

[7] G. Chesshire and W. D. Henshaw, *Composite overlapping meshes for the solution of partial differential equations*, J. Comp. Phys., 90 (1990), pp. 1–64.

[8] G. S. Chesshire, **Overture** : *the grid classes*, LANL unclassified report 96-3708, Los Alamos National Laboratory, 1996.

[9] Diffpack homepage, `http://www.nobjects.com/diffpack`.

[10] F. C. Dougherty and J. Kuan, *Transonic store separation using a three-dimensional Chimera grid scheme*, AIAA paper 89-0637, AIAA, 1989.

[11] W. D. Henshaw, *A fourth-order accurate method for the incompressible Navier-Stokes equations on overlapping grids*, J. Comp. Phys., 113 (1994), pp. 13–25.

[12] ——, *Finite difference operators and boundary conditions for Overture, user guide, version 1.00*, LANL unclassified report 96-3467, Los Alamos National Laboratory, 1996.

[13] ——, *Grid, GridFunction and Interpolant classes for* **Overture** , *AMR++ and CMPGRD, user guide, version 1.00*, LANL unclassified report 96-3464, Los Alamos National Laboratory, 1996.

[14] ——, *Mappings for* **Overture** : *A description of the mapping class and documentation for many useful mappings*, LANL unclassified report 96-3469, Los Alamos National Laboratory, 1996.

[15] ——, *Ogen: an overlapping grid generator for Overture*, LANL unclassified report 96-3466, Los Alamos National Laboratory, 1996.

[16] ——, *Oges user guide, version 1.00, a solver for steady state boundary value problems on overlapping grids*, LANL unclassified report 96-3468, Los Alamos National Laboratory, 1996.

[17] ——, *PlotStuff: a class for plotting stuff from* **Overture** , LANL unclassified report 96-3893, Los Alamos National Laboratory, 1996.

[18] Satish Balay, William Gropp, Lois Curfman McInnes and Barry Smith, *The Portable Extensible Toolkit for Scientific Computation*, `http://www.mcs.anl.gov/petsc/petsc.html`.

[19] Steve Karmesin et.al, *Parallel Object Oriented Methods and Applications*, `http://www.acl.lanl.gov/PoomaFramework`.

[20] D. Quinlan, *Adaptive Mesh Refinement for Distributed Parallel Processors*, PhD thesis, University of Colorado, Denver, June 1993.

[21] ——, *A++/P++ manual*, LANL Unclassified Report 95-3273, Los Alamos National Laboratory, 1995.

[22] Xabier Garaizar, Richard Hornung and Scott Kohn, Structured Adaptive Mesh Refinement Applications Infracture, `http://www.llnl.gov/casc/SAMRAI`.

[23] J. L. Steger and J. A. Benek, *On the use of composite grid schemes in computational aerodynamics*, Computer Methods in Applied Mechanics and Engineering, 64 (1987), pp. 301–320.