LA-UR 96-2931

CONF-9608134--1

TITLE: # OVERTURE: AN ADVANCED OBJECT-ORIENTED SOFTWARE SYSTEM FOR MOVING OVERLAPPING GRID COMPUTATIONS

AUTHOR(S): David L. Brown and William D. Henshaw

SUBMITTED TO: Proceedings of Computational Aerosciences Workshop
NASA Ames Research Center
August 13–15, 1996

OVERTURE: AN ADVANCED OBJECT-ORIENTED SOFTWARE SYSTEM FOR MOVING OVERLAPPING GRID COMPUTATIONS

David L. Brown and William D. Henshaw
Scientific Computing Group CIC-19, MS B256
Los Alamos National Laboratory
Los Alamos, NM 87545
dlb@lanl.gov, henshaw@lanl.gov *Web site:* http://www.c3.lanl.gov/cic19/teams/napc/
(505) 667-0120

1. Introduction. While the development of high-level, easy-to-use software libraries for numerical computations has been successful in some areas (e.g. linear system solvers, ODE solvers, grid generation), this has been an elusive goal for developers of partial differential equation (PDE) solvers. The advent of new high level languages such as C++ has begun to make this an achievable goal. This report discusses an object-oriented environment that we are developing for solving problems on overlapping (Chimera) grids. The goal of this effort is to support flexible PDE solvers on adaptive, moving, overlapping grids. An overlapping grid, as we define it, consists of a set of logically rectangular grids that cover a domain and overlap where they meet. Solutions values at the overlap are determined by interpolation. The overlapping grid approach is particularly efficient for rapidly generating high-quality grids for moving geometries since as the component grids move, only the list of interpolation points changes, and the component grids do not have to be regenerated. We use structured component grids so that efficient, fast finite-difference algorithms can be used. Oliger-Berger-Colella type mesh refinement is used to efficiently resolve fine features of the flow.[1]

Our PDE solvers are written in an object-oriented fashion in C++. We are currently developing solvers for compressible, incompressible and "all-speed" flows in two and three space dimensions. The solvers are written using the *Overture* library, which consists of C++ classes that represent domain mappings, grids, grid functions and difference operators. The operator classes, for example, define discrete approximations to differential operators and their matrix representations as well as a library of elementary boundary conditions . Both vertex-centered (finite difference) and cell-centered (finite volume) approaches are supported. For problems with moving component grids, a C++ version of the CMPGRD[2] automatic overlap algorithm is used. By using a moving grid class, the interface between flow solver and grid generator has been made extremely clean. It is significantly easier to program PDE solvers in C++ using these classes than in Fortran (as we have done in the past), since the details of the overlapping grid data structures are effectively hidden from the programmer. The class libraries and solvers extensively use the A++ array class library[3]. A++ is a serial/parallel array language with a Fortran-90-like syntax. Codes written in A++ will, with little or no changes, run in parallel using the P++ class library.

The remaining sections of this report present some examples demonstrating the *Overture* library functions. These examples clearly demonstrate the power of using a high-level language like C++.

---

[1] K. Brislawn, D. L. Brown, G. Chesshire and J. Saltzman, *Adaptively-refined overlapping grids for the numerical solution of hyperbolic systems of conservation laws*, report LA-UR-95-257, Los Alamos National Laboratory, 1995
[2] G. Chesshire and W. D. Henshaw, *Composite overlapping meshes for the numerical solution of partial differential equations*, J. Comp. Phys., 90, (1990), pp. 1-64.
[3] Quinlan, Daniel, *A++/P++ Manual (version 0.6.5)*, report LA-UR-95-3273, Los Alamos National Laboratory, 1995

2. Overview of the Overture Classes. The main class categories that make up *Overture* are as follows:

1. *Arrays:* multi-dimensional arrays based on A++.
2. *Mappings:* define transformations such as curves, surfaces, areas, and volumes. These are used to represent the geometry of the computational domain.
3. *Grids:* define a discrete representation of a mapping or mappings.
4. *Grid functions:* solution values, such as density, velocity, pressure, defined at each point on a grid.
5. *Operators:* (discrete) differential operators and boundary conditions.
6. *Plotting:* high-level plotting interface based on OpenGL.
7. *Grid Constructors:* classes to construct overlapping grids and handle moving component grids.

3. Using the A++ array class. A++ is an array class library for performing array operations in C++. It will become the array class for HPC++ (High-Performance C++). Here is an example code segment that solves Poisson's equation with the Jacobi method:

```
// Solve u_xx + u_yy = f by a Jacobi iteration
int n = 10;
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));
```

Notice how the Jacobi iteration for the entire array can be written in one statement. When linked with the P++ library, this will be a parallel code.

4. Mappings and Grids. The geometry of the computational domain is defined by a set of mappings, one mapping for each grid. Mappings have been designed so that an object can be easily moved by composing it with a transformation such as a translation, rotation or scaling. 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. 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 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. 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.

5. 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. 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, we make a grid function and assign it values at all points on the grid.

```
SphereMapping sphere;              // ... create a mapping
MappedGrid mg (sphere);            // ... the sphere  mapping has been used to define a grid
mg.update();                       // ... this function computes all the geometry arrays

GridFunctionParameters defaultCentering;          //...other grid function centerings can be
                                                  //...specified through this class
// ... create a grid fn with  default centering and 2 components defined at all grid points
floatMappedGridFunction  u(mg,defaultCentering,2);
Index I1,I2,I3;
getIndex(mg.dimension,I1,I2,I3);                  // ... get Index'es for all grid points

// ... set x-component to sin(x)*cos(y)
const int xComp = 0, yComp = 1;
u(I1,I2,I3,xComp) = sin(mg.vertex(I1,I2,I3,xComp))*cos(mg.vertex(I1,I2,I3,yComp));
```

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

6. 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` defines finite-difference style operators, while the class `MappedGridFiniteVolumeOperators` defines finite-volume style operators . We have also implemented an operator class for incompressible flow Godunov methods. 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);
MappedGridFiniteVolumeOperators op(mg);          //define operators for a MappedGrid
floatMappedGridFunction u(mg), v(mg);
u.setOperators (op);                             //associate operators with grid fn.
u = ...                                          //assign u some values
v = u.grad();                                    //compute gradient of u
v = u.laplacian();                               //compute Laplacian(u)
v = op.laplacianCoefficients();                  //compute matrix for the discrete Laplacian
```

The result of the statement `u.grad()` is a grid function containing the gradient of `u`. An equivalent statement is `op.grad(u)`. The matrix for the discrete Laplacian holds the stencil at each grid point for the Laplacian, and so is a grid function itself. This grid function can be passed to a sparse solver, for example.

6.1 Boundary Conditions. The programming model for boundary conditions is to use ghost points (instead of one-sided difference approximations). We have defined a library of elementary boundary conditions such as Dirichlet, Neumann, extrapolation, etc. Solvers define more complicated boundary conditions in terms of these elementary ones. The interface is quite simple, as can be seen in the following routine.

```
// ... composite grid boundary  types
const int wall      = 1;
const int inflow    = 3;
const int outflow   = 4;
const int slip      = 5;
```

```
void applyVelocityBoundaryConditions (floatCompositeGridFunction & v)
{
  Index allVelocityComponents;
  allVelocityComponents = Range (0,1);
  float ZERO = 0., INFLOW_VELOCITY = 1.0;
  int uComponent = 0, vComponent = 1;

               // ... set velocity to zero on walls
  v.applyBoundaryCondition (allVelocityComponents, BCTypes::dirichlet, wall, ZERO);
               // ... set v=0, du/dn=0 on slip walls (assumed horizontal)
  v.applyBoundaryCondition (uComponent, BCTypes::neumann,   slip, ZERO);
  v.applyBoundaryCondition (vComponent, BCTypes::dirichlet, slip, ZERO);
               // ... set velocity to inflow velocity on inflow boundaries (assumed vertical)
  v.applyBoundaryCondition (uComponent, BCTypes::dirichlet, inflow, INFLOW_VELOCITY);
  v.applyBoundaryCondition (vComponent, BCTypes::dirichlet, inflow, ZERO);
               // ... extrapolate velocities at outflow
  v.applyBoundaryCondition (allVelocityComponents, BCTypes::extrapolate, outflow);
               // ... extrapolate corners, enforce  periodic conditions, interpolate, etc.
  v.finishBoundaryConditions();
}
```

7. An *Overture* code to solve the incompressible Navier-Stokes equations. This example shows a working code that solves the incompressible Navier-Stokes equations in any number of space dimensions on an overlapping grid. It is based on a cell-centered Projection method with a two-stage Runge-Kutta time integrator. A routine to initialize the velocity, `initializeVelocity`, and to initialize the Projection boundary conditions, `setProjectionBoundaryConditions`, must also be supplied to complete the code. `PlotStuff` is the graphics package associated with *Overture*.

```
main ()
{
  CompositeGrid cg;                         //...create composite grid
  getFromADataBase (cg, "grid.hdf");        //...read in from database (HDF) file
  cg.update ();
  Interpolant interp (cg);                  // ... initialize interpolant

  PlotStuff ps (TRUE);                      // ... initialize plotting
  ps.plot (cg);                             // ... plot the grid

  int numberOfVelocityComponents = 2;       // ... velocities stored in q,qMid
  GridFunctionParameters::cellCentered = GridFunctionParameters::cellCentered;
  floatCompositeGridFunction q    (cg, cellCentered, numberOfVelocityComponents);
  floatCompositeGridFunction qMid (cg, cellCentered, numberOfVelocityComponents);
  initializeVelocity (vortexInBox, q, cg);

  CompositeGridFiniteVolumeOperators op (cg);
  q.setOperators (op);
  qMid.setOperators (op);

  Projection projection (cg);               // ... initialize Projection operator
  setProjectionBoundaryConditions (projection);

// ... solve Incompressible Navier-Stokes equations

  float t=0., dt=.0005, viscosity=.05; int numberOfSteps=100;
  int frequencyOfOutput = 10;

  for (int step=0; step < numberOfSteps; step++)
{
                  // ... predict velocity at midpoint using forward Euler
  qMid = q + 0.5*dt*( -1.0*q.convectiveDerivative() + viscosity*q.laplacian());
  applyVelocityBoundaryConditions (qMid);
                  // ... correct by enforcing incompressibility constraint
  qMid = projection.project (qMid);
                  // ... predict velocity at new time using midpoint rule
  q = q + dt*( -1.0*qMid.convectiveDerivative() + viscosity*qMid.laplacian() );
  applyVelocityBoundaryConditions (q);
                  // ... correct again with projection
  q = projection.project (q);
                   // ... plot every so many timesteps
  if (step % frequencyOfOutput == 0) ps.streamLines (q);
}
}
```

8. An *Overture* code that uses moving grids. When a component grid changes during a moving grid computation, the overlapping grid generator must be called at each time step to update the interpolation points. The component grids themselves do not have to be recomputed unless they deform in shape. The grid generator is told which component grids have moved. By default it assumes that the grids have not very far, and can therefore use a much more efficient algorithm to update the interpolation information than is used in the initial grid generation step. In the event that the more efficient algorithm fails, the generator reverts to the standard algorithm. The details of the grid movement and recalculation have been encapsulated in the `MovingGrids` class. The code example below demonstrates some of the functionality of this class.

```
... (initializations) ...

// ... initialize moving grids

  int whichComponentGridToMove = 1;
  real gridRotationRate    = 180.;
  MovingGrids movingGrids(ps, cg, whichComponentGridToMove, gridRotationRate);

  floatCompositeGridFunction gridVelocity   (cg, cellCentered, numberOfVelocityComponents);

  dt = .001;                              // ... set the timestep
  for (int step=0; step < numberOfSteps; step++)
  {
    movingGrids.moveTheGrids (dt);         // ... move the grids

    // ... get the "old" and "mid" level grids
    CompositeGrid & cgOld = movingGrids.getCompositeGrid(0);
    CompositeGrid & cgMid = movingGrids.getCompositeGrid(1);
    //... the grid velocity will be needed by the advection algorithm:
    gridVelocity          = movingGrids.getTransformedGridVelocity (0);

    ... (do stuff with the grids and gridVelocity) ...
```

9. Efficiency. In the current implementation of the A++ array class, array operations typically run at 50% the speed of Fortran. This is because array operations in A++ are performed as a sequence of binary operations. Since the *Overture* classes use A++ for all array operations, our solvers show similar performance; typically we find that they run at about 40% the speed of similar Fortran code. To address this problem, we are in the process of redesigning A++ using C++ “expression templates”. We have demonstrated with simple array classes that we can obtain closer to 90% the efficiency of Fortran using the new approach. and are optimistic that we will achieve this performance enhancement in the new version of A++ as well.

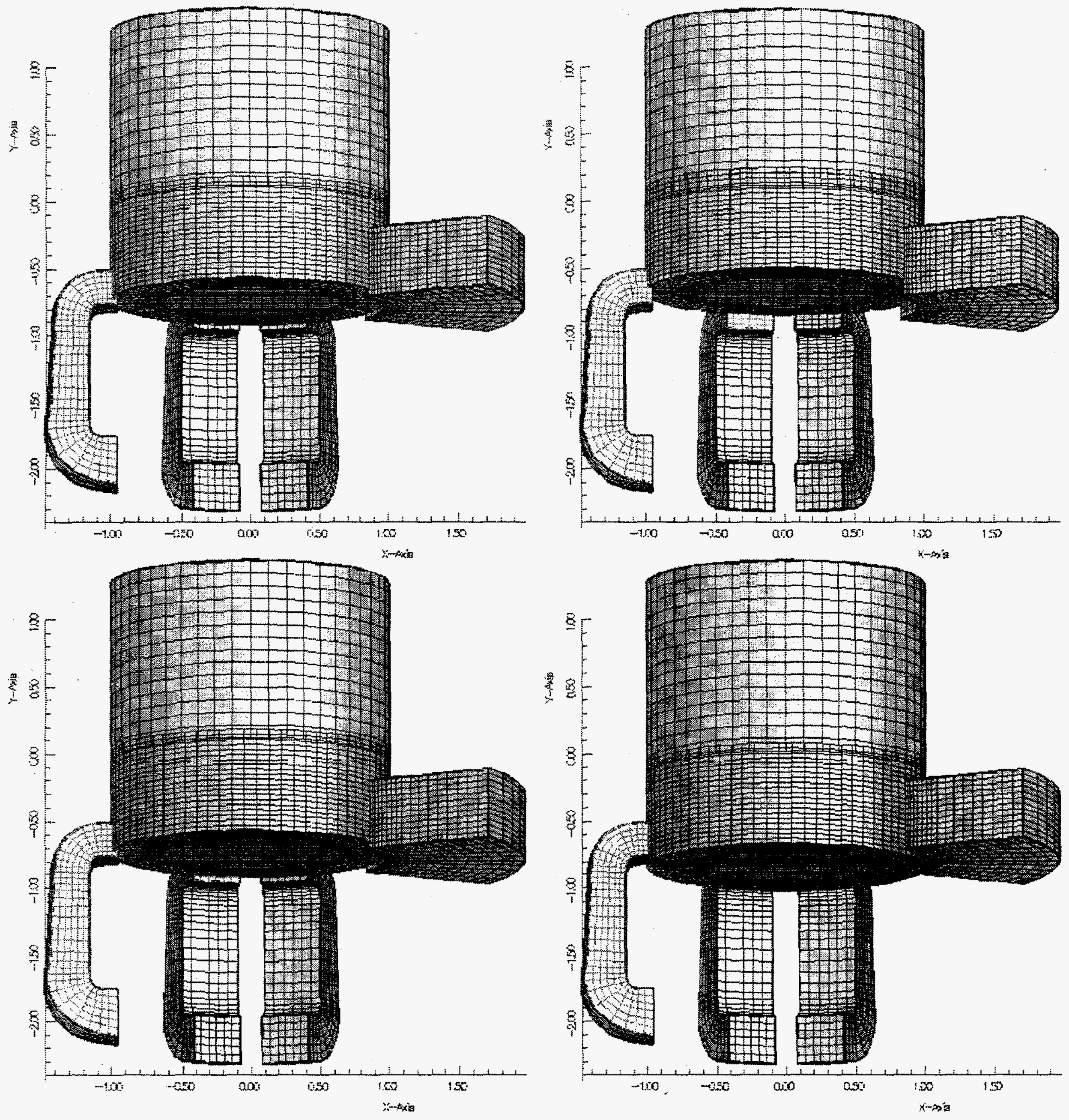

Figure: Overlapping grids for a two-stroke engine; the bottom section of the cylinder moves up and down.

10. Moving grid examples. In the figure we show a moving grid for a two-stroke engine. As the bottom section of the cylinder moves up and down, it closes off the inlet and outlet ports. Further moving grid examples, including flow solutions are available via the Web page listed above.