Object-Oriented Implementation of Adaptive Mesh Refinement Algorithms

WILLIAM Y. CRUTCHFIELD AND MICHAEL L. WELCOME

Applied Mathematics Group, Lawrence Livermore National Laboratory, Livermore, CA 94550

ABSTRACT

We describe C++ classes that simplify development of adaptive mesh refinement (AMR) algorithms. The classes divide into two groups, generic classes that are broadly useful in adaptive algorithms, and application-specific classes that are the basis for our AMR algorithm. We employ two languages, with C++ responsible for the high-level data structures, and Fortran responsible for low-level numerics. The C++ implementation is as fast as the original Fortran implementation. Use of inheritance has allowed us to extend the original AMR algorithm to other problems with greatly reduced development time. © 1994 by John Wiley & Sons, Inc.

1 INTRODUCTION

Advanced finite difference methods, by themselves, are unable to provide adequate resolution of three-dimensional (3D) phenomena without overwhelming currently available computer resources. High-resolution 3D modeling requires algorithms that focus computational effort where it is needed. Adaptive mesh refinement (AMR) algorithms for hyperbolic systems of conservation laws have been shown to be effective for concentrating the computational effort [1–3]. AMR is based on a sequence of nested grids with finer and finer mesh spacing in both time and space. These fine grids are recursively embedded in coarser grids until the solution is sufficiently resolved. The accuracy of the solution is automatically estimated and rectangular fine grid patches are dynamically created or removed to achieve a desired accuracy. Special difference equations are used at the interface between coarse and fine grids to enforce conservation. This is all handled dynamically without user intervention. On large-scale calculations in shock physics, this AMR algorithm has been shown to be one to two orders of magnitude more efficient than comparable uniform grid or exponentially stretched grid algorithms [3].

AMR is a complex algorithm, requiring approximately 10,000 lines in a Fortran implementation of the core algorithm, exclusive of user interface. Only 20% of the lines are the finite difference integrator. The rest of the lines are devoted to maintaining the hierarchy of grids. The following operations are typical of the grid maintenance issues.

1. The hierarchy of grids must be properly nested, with each grid of level n + 1 separated by NPROPER cells of level n from a boundary with grids of level n − 1.
2. Grids are advanced recursively. Advancing the grids at level n for one of their time-steps

Received April 1993
Revised June 1993

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Partial support was provided by the Applied Mathematical Sciences Program of the Office of Energy Research under Contract No. W-7405-Eng-48.

Scientific Programming, Vol. 2, pp. 145–156 (1993)
CCC 1058-9244/94/040145-12

causes the grids of level n + 1 to be advanced several of their time-steps until both levels of refinement are at the same time.
3. When different levels of refinement reach the same point in time, consistency must be enforced. The conservation laws are enforced for all coarse grid cells overlain by fine grid cells, and at boundaries between fine and coarse grids.
4. In time advancing a grid, boundary information may be taken from adjoining grids of the same resolution, adjoining grids of coarser resolution, or from the boundary conditions.
5. When new grid locations are being computed, cells with high truncation error are tagged, then buffering cells tagged, then an efficient partition of tagged cells into properly nested rectangular grid patches is computed.

All of these operations require manipulation of complex data structures, which is difficult in Fortran. The basic ideas of AMR are easy to understand because they are primarily geometric in nature. However geometric concepts are not naturally expressible in Fortran because that language only directly supports algebraic imperative statements. The basic data representation of the solution changes its structure in time in order to represent the changing solution. Grid locations change, the number of grids in a refinement level change, even the number of levels of refinement changes. The dynamic character of the algorithm's data structures is difficult to express in Fortran, which does not even have the concept of a data pointer built into the language. AMR is naturally recursive in its time-stepping, whereas standard Fortran 77 does not support recursion. All these characteristics of AMR make it difficult to express in Fortran.

AMR has been used for 3D calculations in gas dynamics that strain the capacity of the largest currently available vector supercomputers. In typical applications, the time required to manage the grid hierarchy is too small to be reliably measured. Nearly all the time is spent in routines that act upon regular arrays of numbers, e.g., finite difference time-step integration, interpolation, Richardson extrapolation. For the routines that perform these actions, Fortran is an excellent choice of language. First, the only data structure required by such routines is the rectangular array of floating point numbers, which Fortran is adequate to handle. Second, Fortran compilers are still more efficient at producing vectorizing code for these routines than any other language known to the authors. Because these routines dominate the computational cost of the AMR algorithm, use of a less optimized language would be unacceptable.

AMR for systems of hyperbolic conservation laws is a generic numerical technology that is applicable to any set of hyperbolic conservation laws. However, our experience with the Fortran implementation was that the AMR implementation became intertwined with the underlying set of partial differential equations (PDEs). For example, even the upper levels of the Fortran implementation were hard-wired with the number of variables per cell required by the PDE. This intertwining made it nontrivial to replace one PDE package with another, even if both packages modeled hyperbolic systems of conservation laws. By way of comparison, the standard method for utilizing mathematical software is to call a subroutine from a precompiled library. Obviously if a hypothetical mathematical subroutine had to be recompiled with modifications for every use, errors would be much more common. If the AMR logic could be decoupled from the underlying PDE, it would become much more accessible to users.

The AMR algorithm for systems of hyperbolic conservation laws has been undergoing continual development and enhancement. Between 1986 and January 1991, the Applied Math Group at Lawrence Livermore National Laboratory developed nine separate implementations of the AMR algorithm. Several other similar implementations were developed by collaborators of the Applied Math Group. These implementations represented major new applications of the AMR methodology, e.g., AMR on quadrilateral grids [4], or AMR with front tracking [5]. At the same time, algorithmic improvements to the AMR internals took place, along with numerous bug-fixes. With so many versions to maintain, it became difficult to keep all implementations of the AMR algorithm up-to-date with the latest fixes and improvements.

Our experience with Fortran implementations of the hyperbolic AMR methodology was that writing and debugging an implementation took about two thirds of a man-year for an experienced implementer. As we began to look toward using AMR in new physical regimes, i.e., introducing elliptic and parabolic terms to the system of hyperbolic conservation laws, we felt the need to look for methods of reducing the development time.

In early 1991, we began to address these problems by reimplementing our AMR algorithms in the object-oriented language C++. We began with an AMR algorithm for the hyperbolic equations of inviscid gas dynamics (Euler equations) in conservative form. Our objective was to produce an efficient implementation that would be easier to develop and maintain. The ability to define abstract data types (ADT) in C++ made it easy to define objects with an intuitive geometric basis, which made the AMR algorithms easier to understand. The initial AMR implementation would then provide a basis for further AMR algorithms. Subsequent AMR algorithms would draw upon the object-oriented libraries developed for AMR inviscid gas dynamics, and also achieve reuse of code via inheritance. For reasons of efficiency, we chose not to totally reimplement AMR in C++. Low level numerical routines remain in Fortran. Only the high-level, organizational levels of the AMR algorithm were written in C++ (approximately 80% of total lines of code). In brief, each language was given a job commensurate to its abilities.

As of March 1993, we have three major implementations of AMR for three different classes of PDEs: hyperbolic systems of conservation laws, hyperbolic systems of conservation laws with a Cartesian grid representation of boundaries, and coupled hyperbolic and parabolic systems of conservation laws. The implementations draw upon the same object libraries, and are coupled through the use of inheritance. At this time, we have three more implementations in various stages of completion: coupled hyperbolic, parabolic, and elliptic systems of PDEs, hyperbolic systems with front tracking, and multifluid hyperbolic systems.

In the following sections, we will describe our object-oriented implementation of AMR algorithms. In Section 2 we will describe the AMR algorithm for systems of hyperbolic conservation laws in abstract. In Section 3 we describe the object-oriented component libraries we base our implementation upon. In Section 4 we will describe classes that are specific to our implementation of hyperbolic AMR. In Section 5 we describe how we have reused the hyperbolic AMR-specific classes in the derivation of related AMR algorithms. In conclusion, we describe the best and worst points of our experience with the object-oriented implementation of AMR.

2 THE AMR ALGORITHM

Because the requirements of hyperbolic AMR motivate our object-oriented classes, we will discuss in this section the AMR algorithm for hyperbolic systems. For a less terse description, the reader is referred to Berger and Colella [2]. The AMR algorithm for hyperbolic systems of conservation laws is exemplified by our original AMR for inviscid gas dynamics.

The AMR algorithm for hyperbolic systems of conservation laws calculates its solution on a sequence of adaptively refined rectangular grid patches. The class of solutions of these equations consists of regions of smooth solution separated by discontinuities or steep gradients in the solution variables. To compute an accurate solution on a finite difference grid, the mesh spacing must be sufficiently refined to match the length scales of the important features of the solution. In the smooth regions a coarse grid solution is satisfactory but a refined grid is necessary to accurately capture the discontinuities as steep gradients. The AMR algorithm begins with an underlying (level 0) coarse grid (or collection of grids) that covers the entire computational domain. An error estimation procedure identifies cells where the solution is not resolved to a given error tolerance. These tagged cells are grouped into rectangular patches that are spatially and temporally refined to form the level 1 grids. The refinement ratio between the levels is an even constant specified at runtime. This procedure is recursive: error estimation is performed on the level L grids, and tagged cells are grouped and refined to form the level L + 1 grid patches. The process is repeated until either the error tolerance is satisfied or a specified maximum level is reached. The grid patches at each level must be properly nested in the underlying coarse level. Proper nesting requires that the union of level L + 1 grids be properly contained in the interior of the region covered by the level L grids (except at the boundary of the physical domain where all levels can be refined up to the edge).

An example of the AMR grid structure can be seen in Figure 1, which shows the density contour map for the interaction of a shock with an inclined plane in inviscid gas dynamics. The inclined plane is modeled with a Cartesian Grid representation of the body. The Cartesian Grid AMR code is discussed briefly in Section 5. Note that the finest level boxes track the large gradients in the solution. The grid placement is calculated without human intervention.

For the inviscid gas dynamics AMR algorithm, we use an explicit second order Godunov method to advance the solution on each grid patch. The stability requirement for this method is that a signal not be able to pass entirely through a finite

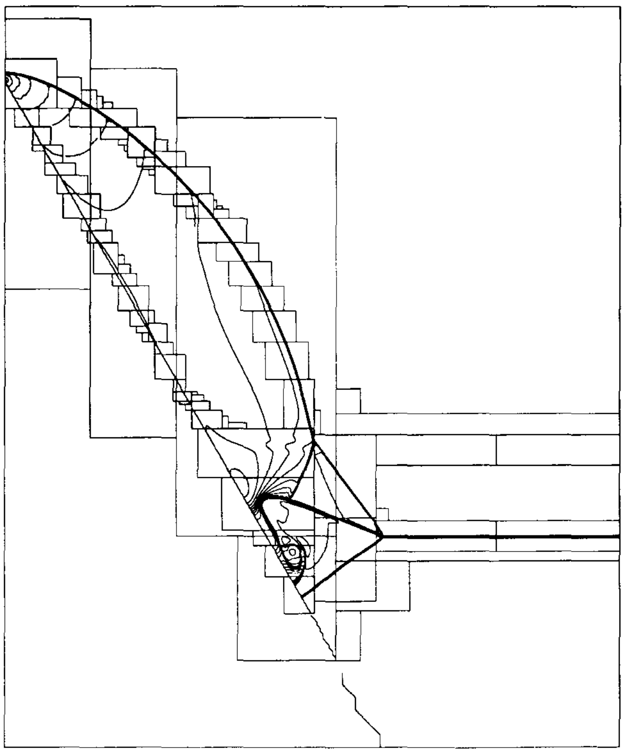

FIGURE 1 Density contours for shock-ramp interactions. Note that grids are automatically placed on regions of large gradients in the solution.

difference cell in any given time-step. This requirement is enforced by restricting the time-step such that S * $dt/dx < CFL < 1$ where S is the speed of the fastest wave in the problem and CFL is the Courant number specified by the user. From this we can see that as we move from one level to the next finest with a refinement ratio of R the time-steps taken on the finer grids must be reduced by a factor of R. Further, for the fine grids to be advanced to the same point in time as the coarse level we must advance them R times for each coarse grid advancement. The time stepping algorithm is recursive: The grids at level L are advanced with a time-step dt(L). The grids at level L + 1 are advanced R times with time-step dt (L + 1) = dt(L)/R. Finally, a synchronization step is performed between level L and L + 1. Note the costs involved in refining a region in 3D with a refinement ratio of R. The storage increases by a factor of R^3 and the computational effort increases by a factor of R^4. R is a small even integer, typically 2 or 4.

The same integration module is used to advance both coarse and fine grids. The stencil for this integrator requires that a certain number of boundary values be supplied for each grid. The boundary values are supplied by either (1) copying from adjacent grids, (2) calling user supplied physical boundary condition functions or (3) interpolating (both in space and time) from grids at a coarser level. When interpolating data from coarser grids, we obtain the data on the required sub-patch by a recursive call to this fill algorithm. Because a time interpolation must be used to supply boundary data on the coarse/fine grid interface we must keep two copies of the data on the coarse levels.

The synchronization step is used to ensure that the solutions on the coarse and fine levels remain consistent. It consists of an "average-down" step followed by a "refluxing" step. Because the numerical method discretely respects the conservation laws, it is necessary that the amount of the conserved quantities contained in a fine grid be the same as that contained in the underlying coarse grid region. The average-down step is implemented by averaging the fine grid data (in a volume weighted fashion) down to the coarse grid region covered by the fine grids.

A conservative scheme updates the solution by computing fluxes across the faces of each finite difference cell. The state of the solution in a cell at the new time is the state at the old time plus the net flux across each of its faces. The fluxes computed on the coarse grid are in general not equal to the cumulative fluxes over the same physical region on the fine grids. The "refluxing" step effectively replaces the coarse grid fluxes with the cumulative fine grid fluxes. The reflux step updates the coarse grid cells adjacent to but not covered by the fine grids with a correction term that represents the difference between the coarse grid and fine grid fluxes.

As the dynamics evolve in time the important features of the solution move through the computational domain in a way that cannot be predicted a priori. The hierarchical grid structure must be periodically regenerated to capture these features. Grid generation is performed from the finest level down to the coarsest. Error estimation is performed on the old level L grids to determine where the new level L + 1 grids will be placed. The new level L grids are determined by an error estimation procedure on the level L − 1 grids with the added restriction that the new level L + 1 grids be properly nested within the new level L grids. Once the grid locations are determined for all levels, data space is allocated and state variables are defined by either (1) copying on overlap from the old grid structure, or (2) spatial interpolation from the old grid structure. Care is taken to release the memory from the old grid structure as soon as possible and to delay memory allocation for the new grid structure to minimize memory consumption.

The error estimation procedure consists of both a Richardson error estimation to detect errors in the solution as well as a physics-dependent error estimator that can detect slowly moving or stationery gradients missed by the Richardson estimator. High error cells are tagged, then buffered by tagging neighbor cells. All the tagged cells are organized into a "cluster," which consists of a list of tagged cells and the minimal rectangular box containing them. We use a cutting algorithm based on a combination of signatures and edge detection used in computer vision and pattern recognition [6] to find the best place to chop the cluster. Chopping a cluster consists of sorting the tagged cells into two lists depending on which side of the cutting plane they lie. Minimal boxes are computed for each cluster after the chop. This process is performed until the ratio of tagged cells to total number of cells in a cluster reaches a given efficiency value. These clusters may be further chopped if necessary so that they lie within the proper nesting domain for the level in which they reside. The proper nesting domain for a level is the largest subregion of that level that can be refined to the next level without violating the proper nesting requirement. Finally, the cluster boxes at level L are refined to level L + 1 to become the new level L + 1 grid structure.

3 GENERIC OBJECT LIBRARIES FOR ADAPTIVE ALGORITHMS

In the process of designing the C++ version of AMR we identified several basic classes that would serve as the building blocks for many of the larger classes used in AMR. Considerable effort was spent in designing, implementing, optimizing, and testing these basic classes because their use was widespread. One of our goals was to implement AMR in a dimension-independent manner, with the dimension specified at compile time. The base classes implement this goal with a macro for the class name that expands to the actual name of a class with the correct dimensionality. For example, the "box" class is actually three classes, "box1d", "box2d" and "box3d", but the macro

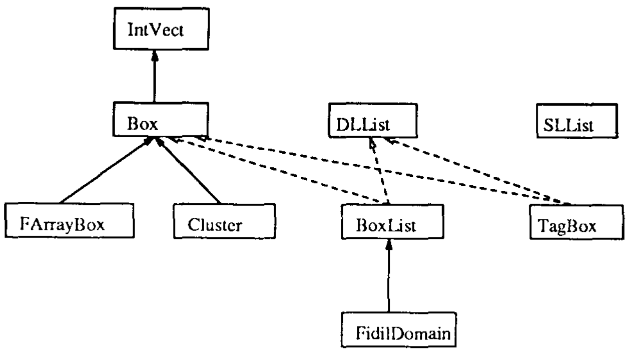

FIGURE 2 Derivation/composition graph for some of the generic library classes. Dashed lines show class derivation and solid lines indicate class composition. Arrows point from larger, more composite class to smaller, more fundamental class.

name "BOX" can be used and will be translated to the appropriate name at compile time based on the spatial dimension. In general, the only exceptions to the dimension-independent interface are in certain dimension-specific constructors.

Figure 2 shows the more important of the generic classes we have derived. In addition to the classes shown in Figure 2, we have implemented classes for X-Window or postscript graphics and character strings, such as are found in many class libraries. In Figure 2, lines indicate relationships between classes. Solid lines indicate that one class includes objects of the second class (composition). Dashed lines indicate that one class inherits from the second. The lines indicate a direction, going from the larger, more composite to the smaller, more fundamental.

The most basic class is INTVECT, which implements points in N-dimensional integer space. Operations include partial ordering relations, I/O operations, selection of individual components, and basic arithmetic operations.

One very useful class is the BOX class, which describes rectangular subregions of N-dimensional integer space. A BOX consists of two INTVECTs, specifying the diagonal corners of the region. Boxes have a large number of operations including intersection, translation, refinement, and coarsening by a given ratio. BOX expansion, shrinking, and chopping operations are available as well as a full complement of access functions. The BOX class has 59 different member functions, exclusive of the constructors, destructors, and access functions. In addition, a BOX can be defined to represent a rectangular region in either cell centered or node centered coordinates in each index direction. The usefulness of defining a BOX class for adaptive algorithms can be illustrated with a code fragment. Very often in the AMR algorithm it is necessary to ask if two rectangular regions have a non-null intersection. In Fortran, such a test would be expressed as:

```
ixlo = max0(iplo,imlo)
ixhi = min0(iphi,imhi)
jxlo = max0(jplo,jmlo)
jxhi = min0(jphi,jmhi)
kxlo = max0(kplo,kmlo)
kxhi = min0(kphi,kmhi)
if ((ixlo.le.ixhi) .and.
(jxlo.le.jxhi) .and.
(kxlo.le.kxhi)) then
      .
      .
      .
endif
```

In comparison, with the use of the BOX class, the same test can be written as:

```
  if (pbox.intersects(mbox)) {
        .
        .
        .
  }
```

Clearly the second case is much easier to read and write, as well as much more likely to be entered correctly. By actual count, the hyperbolic AMR algorithm calls the BOX intersection test 19 times explicitly. In addition, the AMR algorithm calls many library functions that implicitly call the BOX intersection test.

We have also implemented generic singly and doubly linked list classes. Through derivation from the BOX class and the doubly linked list class we have implemented a BOXLIST class. We have extended this class with functions that, for example, determine if a given box is contained in the union of the boxes in the list. We have used this class as the private representation of a FIDIL_DOMAIN [7] class which is used to represent arbitrary subregions of N-dimensional integer space as a list of mutually disjoint boxes. Some interesting operations of the FIDIL_DOMAIN class are union, intersection, complement and accrete. We use FIDIL_DOMAINS to represent the proper nesting regions of each level during the regridding process of the AMR algorithm.

A CLUSTER class consists of a list of INTVECTS and the minimal BOX containing them as described in the AMR algorithm above. A TAGBOX class is also defined. It consists of a BOX along with an integer tag array used in marking cells of a grid for masking operations.

One of the most useful classes in the base library is the FARRAYBOX class. This class is a C++ implementation of a Fortran array. It consists of a box defining the index extent of the array and a pointer into heap allocated storage where the array data are stored. The data are stored in Fortran column major order so that it can easily be passed to Fortran worker routines that manipulate the data. The majority of data manipulated in the AMR code, such as the state arrays for a grid, are implemented as FARRAYBOX's. FARRAYBOX operations include copying on intersect, selection of values, access functions, dynamic resizing operations, index shifting, computing min and max values, along with a variety of simple arithmetic operations such as the addition of two FARRAYBOX's or multiplication by a scalar value. We stopped short of implementing a general array language because it is difficult to implement a general array language efficiently. Our general philosophy in this area is that if efficiency is important, it should be written in Fortran. FARRAYBOX operations are used frequently so we took care to optimize the operations to take advantage of the target architecture. When compiling for a vector processor we unroll loops and vectorize over the longest index direction. When compiling for a cache architecture, the loops respect the cache. Further, because all the FARRAYBOX memory is allocated by C++ and the majority of the dynamic memory used by AMR is contained in FARRAYBOX objects, we have taken over memory management for this class. We have implemented a compacting memory manager that eliminates fragmentation and have recently written a swapping memory manager that efficiently uses the solid state disks available on the CRAY YMP series as secondary memory. With this feature, we are able to run problems that would ordinarily require over 100 megawords of main storage in only 10–20 megawords of main memory. The overhead realized in the SSD version is less than 5% in the tests we have made to date.

4 AMR IMPLEMENTATION CLASSES

Each of the AMR-specific classes corresponds to a major component of the AMR algorithm. The Integrator class corresponds to the finite difference

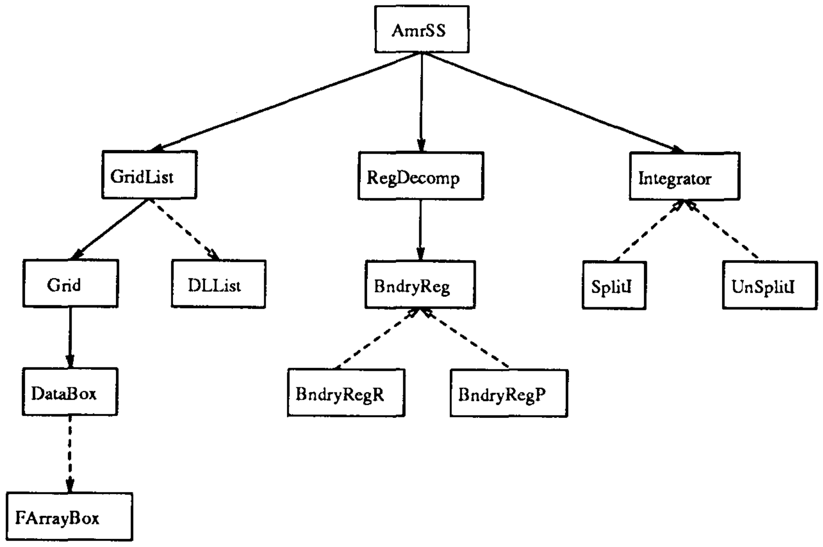

FIGURE 3 Derivation/composition graph for the major AMR implementation classes. Dashed lines show class derivation and solid lines indicate class composition. Arrows point from larger, more composite class to smaller, more fundamental class.

method that advances a grid by the time-step. The DataBox class contains the state variables for a single rectangular region of the index space at a specified time. Objects of the Grid class contain two DataBox's, representing the state of a rectangular region in index space at two successive times. All the Grid objects at a given level of refinement are collected into objects of class GridList. A GridList object contains all the information describing the problem solution on a given level of refinement at two successive times. At the highest level of abstraction are objects of class AmrSS. An object of this class contains all the data and member functions necessary to implement AMR for a hyperbolic system of conservation laws. Each AmrSS object contains one GridList for each level of refinement it uses. The set of GridList's represents all the state data contained in the system of hierarchically refined grids. Figure 3 shows the derivation and composition relationships between the major AMR implementation classes. Dashed lines show class derivation and solid lines indicate class composition.

The Integrator class is a virtual base class [8, 9]. No objects of class Integrator should ever exist. Useful integrator classes are derived from the base class Integrator. Class Integrator merely defines a uniform interface to actual integrator classes. We have many subclasses derived from class Integrator, corresponding to different finite difference methods and different PDEs, e.g., split Godunov integrator for inviscid gas dynamics [10–12], unsplit Godunov integrators for inviscid gas dynamics [13], fourth-order accurate Lax-Wendroff scheme for acoustics, etc. Objects derived from class Integrator implement finite difference schemes and often contain no data of their own. Such data-less objects are employed for their abilities (to apply a finite difference method) not for their data. In our AMR implementations, an object of base class Integrator is accessed through a pointer and the class's two virtual functions. When the upper levels of the AMR algorithm are compiled, the AMR algorithm has no way to know which class derived from class Integrator it will be used with. In fact, our implementation makes the choice of Integrator object a run-time decision. Class Integrator defines two virtual functions. One function applies a finite difference scheme to a rectangular array of numbers. A second function returns a set of integers that characterize the finite difference method and the physical system. The numbers include the number of variables required to define the state of the physical system in each grid cell, the maximum allowable size of grid-patches, the stencil width or number of extra zones of data required to apply the finite difference method, the coordinate system, etc. The higher levels of AMR access this function during initialization and take appropriate actions during the computation based on these numbers.

An object of class DataBox contains the information required to describe the state of the system on a rectangular region of index space at a single point in time. In the simplest cases, the state can be described with a rectangular array with a fixed number of variables per each cell in the array, but more complex configurations are possible. For the case of inviscid gas dynamics, DataBox is derived from the library class FARRAYBOX. Member functions, defined for class DataBox, correspond to reasonable actions that can be applied to the system state at a single point in time. For example, the state can be integrated forward in time, or interpolated onto another DataBox of finer resolution, or copied into another DataBox, etc. Of all the AMR classes, only classes DataBox and Integrator are dependent on the underlying PDE. The PDE dependence is encapsulated within these two classes, allowing the rest of AMR to be independent of the PDE. In addition to being dependent upon the PDE, the DataBox class is also dependent upon the problem being solved. For example, the DataBox class has a function that initializes a DataBox. Such an initialization function will frequently be changed as different problems are studied. The DataBox class calls user-defined routines written in Fortran that perform such problem-dependent functions. Fortran was chosen for this purpose to make the AMR algorithm more accessible to users.

Objects of class Grid contain two DataBox's, representing the state of a logically rectangular region at two successive times. Some Grid's also contain Flux Registers, specialized DataBox's that are used to store fluxes at the edges of the Grid for later use in refluxing. In addition to performing operations upon its constituent DataBox's, the Grid class is able to perform operations that require two DataBox's. For example a Grid is able to fill a DataBox with interpolated data at any time-level intermediate between the times of its constituent DataBox's. A Grid is also able to make an error estimation upon the solution in its domain and determine where further refinement is required. A Grid also has responsibility for managing its memory consumption in order to reduce the memory consumption of the entire application. DataBox's that are no longer needed are released as soon as possible. Flux registers are only allocated by the Grid object if the Grid will utilize them later. An object of class Grid can perform numerous operations required by the AMR algorithm upon the solution in a rectangular spatial region, for example, initialization, time-step advance, tagging cells requiring further refinement, various I/O functions, correcting the solution on a coarse grid to agree with an overlaying fine grid, etc.

All the Grids at a given level of refinement are collected together in an object of class GridList. This collection class is implemented as a doubly linked list and provides iteration functions used to step over the members of the list. Class GridList also implements AMR actions that are implemented on an entire level of refinement, for example, time-step advance or synchronization between levels.

The AMR implementation classes also contain two classes that are used to represent boundary conditions. The BndryReg class is a general representation of a boundary condition. We implement a boundary condition as a function that provides values outside of the physical problem domain. A BndryReg object contains a BOX describing where the boundary condition may be applied and a rule for generating data in this region. The BndryReg rule may generate the data independent of the solution in the interior (e.g., supersonic inflow boundary condition) or may be dependent upon the solution in some fashion (e.g., reflection boundary conditions). The "rule" function is provided to an object of class BndryReg when it is instantiated. We have also derived classes from BndryReg that handle specific types of boundary conditions, for example, periodic (BndryRegP) and reflecting (BndryRegR). Reflecting boundary conditions are used to represent walls and symmetry planes in the problem. These two derived classes do not require coding a rule function, and are consequently easier to use.

Objects of class BndryReg are collected together in an object of class RegDecomp. The regions associated with each BndryReg object must be disjoint and their union must cover all cell locations within one stencil width of the physical region. These conditions are checked by the RegDecomp object. A RegDecomp object can be asked to provide data from outside the physical domain. The RegDecomp object then requests each of its BndryReg objects that intersect the desired region to provide data in the intersection. The RegDecomp object orders the BndryReg objects internally so that the external data required by some RegDecomp objects is available before the object is invoked. The use of boundary condition objects has made it considerably easier to setup boundary conditions for applications than was our experience with the Fortran implementation.

At the highest level of abstraction are objects of class AmrSS. An object of this class contains all the data and member functions necessary to implement AMR for a hyperbolic system of conservations laws. Each AmrSS object contains one GridList for each level of refinement it uses. An AmrSS object contains a RegDecomp object to describe the boundary conditions. An AmrSS object has member functions that implement actions affecting multiple levels of refinement. For example, an AmrSS object can advance the solution by one coarse grid time-step. This requires multiple time-steps at the finer levels and a synchronization between levels when two levels of refinement are advanced to the same point in time. The AmrSS object is also responsible for computing the size of the next time-step, based on the CFL numbers reported by each of the Grid objects as they are time-stepped. At specified intervals during the time-step advance, new grid positions are calculated. Fine grid levels are "regridded" more often than coarse grid levels. Individual Grid objects on each level are responsible for tagging cells requiring refinement within their domain, but the AmrSS object assembles the tagged cells and finds an efficient grid structure that covers the tagged cells. During many phases of the AMR algorithm (e.g., time-step advancing a Grid) it is necessary to fill a rectangular array with data values representing the solution at a specified time. The AmrSS class has a member function that is the publicly accessible interface to that information. In the general case, gathering this data may involve copying from grids of one level, interpolating data from a coarser level, and deriving data from a boundary condition. Because this is a multigrid, multilevel operation, it is appropriate that the AmrSS class implement it.

Because the C++ objects we have implemented always pass computationally intensive parts of the calculation to Fortran subroutines, we expected that the overhead of C++ would not have a measurable effect upon the AMR algorithm. To test this assumption, we performed a standard 2D calculation of a shock wave hitting a cloud of denser gas with both the original Fortran calculation and the C++ implementation. The C++ implementation was actually about 1% faster. We cannot conclude that C++ is faster than Fortran from this experiment because the algorithms had changed slightly in the intervening year (due in some part to our ability to write better algorithms in C++). But we can eliminate the hypothesis that C++ is drastically slower than Fortran in this application. This conclusion should be true in 3D as well.

5 REUSE OF AMR-SPECIFIC CLASSES

With C++, we have the ability to define abstract data types and manipulate them as if they were part of the language. This has made it easier to write and read complex adaptive algorithms. There is another notable contribution to our development work that C++ has made. C++ enables us to derive new algorithms from old algorithms in analogy to the way classes are derived from base classes.

In the previous section we described how we employed the "data-hiding" property of C++ to decouple the higher levels of AMR from the DataBox and Integrator classes, which are dependent on the underlying PDE. There are other logical cleavage lines in the algorithm that allow different portions of the algorithm to be treated independently. A simple example of this is our variable-ratio version of AMR for inviscid gas dynamics.

In the original AMR implementation for hyperbolic systems, the ratio between time-steps at different levels of refinement is fixed to be the same as the ratio of spatial refinement. This is a reasonable strategy when the variation between velocities in the problem domain is not too far from one. However, if the problem domain has regions of high velocity, the CFL time-step limit in these regions will dominate the determination of the time-step for the entire domain at all levels of resolution. Considerable improvement is possible if the high velocity region is for some reason not represented on the finest level of refinement. Then the size of the time-steps that the finest level can take should be limited by the highest velocity in the region covered by the finest level, not by the physical domain as a whole. However, this requires changing the ratio between time-steps at different levels of refinement.

In the AMR C++ implementation, the determination of time-steps is localized to the AmrSS class. It is possible to replace the standard time-step determination logic with an alternative logic that dynamically determines the optimum ratio of time-steps for the solution at each time-step. However, most of the AmrSS implementation is unrelated to time-step determinations, for example, the regridding logic. We do not wish to simply copy this logic into a new class because this creates a second implementation of the regridding logic that must be constantly updated with improvements and corrections. Instead we use class derivation.

We have derived a class AmrVR from the existing AmrSS class. The only code in the AmrVR class is an overloading of the AmrSS member functions that advance the solution by one time-step. It differs from the AmrSS member function by choosing the time-step on each level of refinement to be as large as the CFL restriction allows, subject to the restriction that ratios between time-steps must be integral. The class AmrVR otherwise inherits the existing AmrSS functions for regridding, user interface, etc. The AmrVR implementation involves approximately 200 lines of new code compared to the 2,000 lines of code in the AmrSS implementation. Because class AmrVR is derived from AmrSS, it can be used as a direct replacement in any program using AmrSS. We have used the AmrVR class in very large scale 3D gas dynamic simulations with considerable time savings.

A larger scale example of "algorithm derivation" is our development of an AMR Navier-Stokes simulator for compressible gas flows. A Navier-Stokes simulator requires the addition of parabolic terms that model the diffusive transport of momentum and heat. We use a higher order Godunov scheme for the advective portion of the equations. The viscous terms are integrated with a second-order accurate Crank-Nicolson scheme. The advective and viscous terms are coupled together in a predictor-corrector cycle. The implicit Crank-Nicolson step requires the solution of a linear system on the grids of a given level of refinement.

A number of new operations are required in the compressible Navier-Stokes algorithm. Viscous terms must be calculated, linear systems must be solved, and viscous fluxes must be accumulated and refluxed to conserve energy and momentum. The cycle of time-steps must be modified to include the linear solve. The basic explicit time-step integrator for the advective terms is modified. However, much remains of the original hyperbolic AMR algorithm. The grid structure is the same; the advective time-step integrator is almost identical; the regriding algorithm is the same; the synchronization operation between levels of refinement is the same, with the caveat that the fluxes contain viscous fluxes as well as advective fluxes.

Unlike the case of variable-ratio AMR, the compressible Navier-Stokes AMR cannot be implemented with a single changed C++ class. The required changes extend from the highest levels of AMR down to the base advective time-step integration. We have derived five new classes from the standard hyperbolic AMR implementation classes. For the most part they differ from the standard hyperbolic AMR classes by the addition of new capabilities related to the viscous terms. For example, the class DataBoxCNS has the capability of computing the viscous fluxes from its state in its rectangular region. In a few places, the compressible Navier-Stokes classes overload

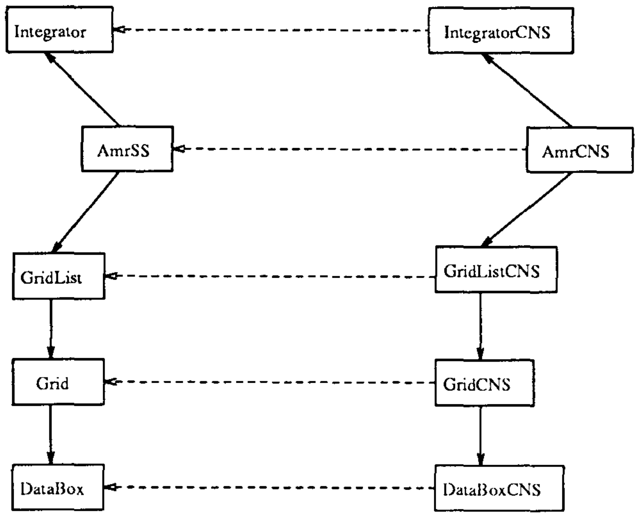

FIGURE 4 Derivation/composition graph showing relationship between compressible Navier-Stokes AMR classes and the hyperbolic AMR classes. Dashed lines show class derivation and solid lines indicate class composition. Arrows point from larger, more composite class to smaller, more fundamental class.

member functions of the hyperbolic classes, mainly in the time-step cycle which is modified to include the solution of the Crank-Nicolson linear problem for the viscous terms. Figure 4 shows the inheritance and composition relationships.

With this derivation scheme we have been able to easily implement an AMR algorithm for compressible Navier-Stokes. We had to write approximately 1,900 lines of additional code, mainly in Fortran to implement the calculation of viscous terms. The program retains all the improvements and corrections we have implemented over the years in the hyperbolic AMR program because it is derived from it. It will continue to incorporate new improvements that we introduce into hyperbolic AMR except for improvements we make in the few member functions that are overloaded by new functions in the AMR compressible Navier-Stokes class structure. For example, all the code we have introduced into the hyperbolic AMR code to write graphics files in several different formats is present and working in the compressible Navier-Stokes AMR program. In our opinion, the ability to transparently track improvements in the hyperbolic AMR implementation is at least as important as the ability to write an important new application with only 1,900 lines of new code. We have applied the compressible Navier-Stokes algorithm in 2D to important unsolved problems in viscous shock dynamics [14].

We have also applied our methods for algorithm derivation to an even more ambitious goal. We have developed AMR algorithms for modeling geometrically complex boundaries in inviscid gas dynamics. We refer to these algorithms as using a Cartesian Grid representation of the boundary [15]. We differ from Berger and Leveque [15] by employing a tracked front [5, 16] representation of the interface. The Cartesian Grid AMR algorithm has proved to be very successful in simulating inviscid gas dynamics in a geometrically complex region [17]. This has required rather more extensive changes to the class hierarchy than did compressible Navier-Stokes because many functions must now be provided with extra arguments describing geometry. Seven new Cartesian Grid classes are derived from their counterparts in the hyperbolic AMR hierarchy: AmrCart, DataBoxCart, GridCart, GridListCart, ContourCart, RasterCart, and CartInteg. In addition three new classes, RDData, SparseData, and SparsePencil were introduced to store the geometric description in sparse data structures. Approximately 16,000 new lines of code were required for a combined 2D and 3D implementation, or about half of the total number of lines of code.

6 CONCLUSIONS

Since we first began implementing our codes in C++ in early 1991, we have accumulated many man-years of experience with implementing complex adaptive algorithms in an object-oriented style. In this section, we would like to recapitulate where we think C++ has really been a success for us, and where it has held us back.

In terms of the greatest benefit for the smallest amount of work, our greatest object-oriented success is our ability to write dimension-independent code in C++. Our geometrically based algorithms have the same expression in 1D, 2D, and 3D when we use a geometric language of intersections and unions of BOX's. Using the libraries and macros described in Section 3, we can do that in C++. In Fortran, we were forced to maintain a different implementation of AMR for each dimension. We find that if we test and debug a geometrically complex algorithm in 2D, it will very often work immediately in 3D. When errors do occur, they are usually in the Fortran portion of our code. Given that debugging our algorithms in 3D is expensive, this is important to us.

More generally, we certainly believe that we have benefited from C++'s ability to define abstract data types. This makes our algorithms much easier to express and to understand. The ease of writing complex algorithms in C++ has had the important effect of lowering the threshold for algorithm changes. For example, we have known for years that it was possible to reduce a "memory bulge" in the algorithm by getting rid of old time-level DataBox's on the finest level as soon as a Grid can guarantee that no other Grid needs its old time-level data. However, the difficulty of keeping track of the data dependencies in Fortran was so daunting that it was never implemented. Using C++ the threshold of effort for implementing the modification was low enough that we did implement the changes. The changes took a few hours.

Another object-oriented benefit we can confirm is that data hiding is beneficial for increasing the modularity of our implementations. By localizing the physics dependence in DataBox and Integrator, we made it much easier to introduce totally new physics packages without breaking the program.

Reuse of code through the use of object-oriented libraries and inheritance has been another success. The preceding section described some of our work with algorithm derivation. We have also greatly benefited from the generic object libraries, not only in the predominantly hyperbolic AMR algorithms described in this article, but also in our parallel development of AMR methods for incompressible fluid flow. We have not as yet been able to fit these AMR algorithms into the class structure of hyperbolic AMR in the same fashion that we did with compressible Navier-Stokes. But our generic libraries have been useful and are an important part of that effort.

Our dual language Fortran/C++ implementation has been a success in terms of efficiency and portability. If we could not be assured of writing implementations at least as efficient as the original Fortran implementations, we could not have begun this project. Of course, our ability to write an efficient Fortran/C^{++} implementation is dependent upon our primary work objects, i.e., DataBox's, being large objects that can easily hide the overhead of a call to a Fortran routine. Other algorithms may not have that luxury.

Despite its success, the necessity of writing part of our algorithm in Fortran is a major complaint for us. Although not intrinsically difficult, Fortran imposes an unwanted level of bureaucracy in our implementation with the necessity of making an extra call, rearranging the arguments into something Fortran can understand, and writing macros to translate from the Fortran naming convention into C++'s naming convention. If C++ and C had good, reliable, robust, and highly vectorizing compilers, this effort would not be needed. Of course, this support would have to be universal across the machine architectures that we target before we could take advantage of it. In addition, we would need to have decent debugger support for C++.

A C++ language problem we have encountered is the difficulty with using derivation on tightly coupled data structures. Consider the relationship between DataBox, DataBoxCNS, Grid, and GridCNS shown in Figure 4. A Grid contains pointers to objects of type DataBox. GridCNS is derived from Grid. We would like GridCNS to contain pointers to objects of type DataBoxCNS, but C++ rules of derivation force GridCNS to have pointers to type DataBox. We have overcome this problem with explicit casting in those portions of the GridCNS implementation that need to use member functions of class DataBoxCNS. This is inelegant and requires special care to document the relationships that cannot be expressed in the C++ language.

Our use of C++ for AMR algorithms has proven so successful that we no longer perform any development in pure Fortran. Our mainstay AMR implementations have been converted to C++/Fortran hybrid implementation. Numerous new development projects in the LLNL's Applied Math Group are taking place using C++ and object-oriented libraries.

REFERENCES

[1] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *J. Comp. Phys.*, vol. 53, pp. 484–512, 1984.

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

[3] J. B. Bell, M. Berger, J. Saltzman, and M. Welcome, "Three dimensional adaptive mesh refinement for hyperbolic conservation laws," *SIAM J. Sci. Stat. Comput.* 1994.

[4] J. B. Bell, P. Colella, J. A. Trangenstein, and M. Welcome, *Proceedings of AIAA 9th Computational Fluid Dynamics Conference.* Buffalo, New York: AIAA, 1989.

[5] J. B. Bell, P. Colella, and M. Welcome, *Proceedings of AIAA 10th Computational Fluid Dynamics Conference.* Honolulu, Hawaii: AIAA, 1991.

[6] M. J. Berger and I. Rigoutsos, "An algorithm for point clustering and grid generation," *IEEE Trans. Systems, Man Cybernet.* vol. 21, pp. 1278–1286, 1991.

[7] P. N. Hilfinger and P. Colella, *Symbolic Computation: Applications to Scientific Computing.* Moffet Field, CA: SIAM Frontiers in Applied Mathematics, Vol. 5, 1989.

[8] S. B. Lippman, *C++ Primer.* Reading: MA, Addison-Wesley, 1989.

[9] B. Stroustrup, *The C++ Programming Language.* Reading: MA, Addison-Wesley, 1986.

[10] P. Colella and P. R. Woodward, "The piecewise parabolic method (PPM) for gas-dynamical simulations," *J. Comp. Phys.*, vol. 54, pp. 174–201, 1984.

[11] P. Colella, "A direct eulerian MUSCL scheme for gas dynamics," *SIAM J. Sci. Stat. Comput.*, vol. 6, p. 104, 1985.

[12] P. Colella and H. M. Glaz, "Efficient solution algorithms for the Riemann problem for real gasses,", *J. Comp. Phys.*, vol. 59, pp. 264–289, 1985.

[13] P. Colella, "Multidimensional upwind methods for hyperbolic conservation laws," *J. Comp. Phys.*, vol. 87, pp. 171–200, 1990.

[14] J. B. Bell, P. Colella, W. Y. Crutchfield, and M. Welcome, "An adaptive mesh refinement scheme for the compressible Navier-Stokes equations" (in preparation).

[15] M. J. Berger and R. Leveque, *AIAA 9th Computational Fluid Dynamics Conference.* Buffalo, NY: AIAA, 1989.

[16] I. Chern and P. Colella, "A conservative front tracking method for hyperbolic conservation laws," *J. Comp. Phys.* (in press).

[17] R. J. Pember, J. B. Bell, P. Colella, W. Y. Crutchfield, and M. J. Welcome, "A three dimensional adaptive Cartesian grid algorithm for inviscid compressible flow" submitted to *J. Comp. Phys.*