LA-UR 98-293

CONF-980509--

Approved for public release; distribution is unlimited

Title: AMR++: Object-Oriented Design for Adaptive Mesh Refinement

Author(s): Dan Quinlan

Submitted to: HPC '98
April 5-9, 1998
Boston, Massachusetts

Los Alamos
National Laboratory

DISCLAIMER

AMR++: OBJECT-ORIENTED DESIGN FOR ADAPTIVE MESH REFINEMENT

Dan Quinlan
Scientific Computing Group CIC-19
Computing, Information, and Communications Division
Los Alamos NM, USA, 87545
dquinlan@lanl.gov

Keywords: Adaptive Mesh Refinement, Object-Oriented Design.

ABSTRACT

The development of object-oriented libraries for scientific computing is complicated by the wide range of applications that are targeted and the complexity and wide range of numerical methods that are used. A problem is to design a library that can be customized to handle a wide range of target applications and increasingly complex numerical methods while maintaining a sufficiently useful library for simple problems. These problems have been classically at odds with one another and have compromised the design of many object-oriented library solutions. In this paper we detail the mechanisms used within AMR++, and object-oriented library for Adaptive Mesh Refinement (AMR), to provide the level of extensibility that is required to make AMR++ easily customizable for the more obscure applications while remaining small and simple for less complex applications. The goal has been to have a complexity that matches the complexity of the target application. These mechanisms are general and extend to other libraries as well.

INTRODUCTION

Structured Adaptive Mesh Refinement (SAMR) is a numerical technique to tailor the resolution of the computational grid by locally placing structured grid refinement where appropriate to obtain greater accuracy where required within a physical simulation. The use of structured grids to form the adaptive process greatly simplifies the process and permits the use of efficient and general structured grid techniques and their combination into efficient AMR solvers. The use of structured grids permits the use of array abstractions for the simplified expression of numerical algorithms and the encapsulation of parallelism using parallel array abstractions. The use of array abstractions does not imply only FORTRAN 90, but implies any means of abstracting the array operations on array data. In the case of AMR++ we use internal parts of the OVERTURE Framework which provides array objects through the use of the A++/P++ serial/parallel array class library. The point of this abstraction is that OVERTURE builds more powerful abstractions from the A++/P++ array objects (grid functions) that include the details of handling complex geometry permitting application solvers to then be independent of geometry. Such abstractions are algorithmicly more powerful than the array objects represented by the array class library. AMR++ uses these more sophisticated grid function abstractions within OVERTURE which in current work addresses the requirements and complexities of industrial level applications such as the modeling of internal combustion engines.

The uses of AMR are numerous within physical solutions and arise naturally as a result of the increasingly disparate length scales associated with larger more sophisticated computer simulations. The problems with AMR within physical simulations are numerous and result from a combination of the complexity of the numerics and the complexity of the software required to support it. In the parallel environment the complexity of AMR is greatly exacerbated by the added degrees and types of parallelism. AMR++ [1] is a C++ object-oriented library that simplifies the development of serial and parallel adaptive mesh refinement. AMR++ forms a part of

[1] Information is available from the AMR++ Home Page: http://www.c3.lanl.gov/~dquinlan/AMR++.html.

the OVERTURE Framework, a much larger hierarchy of class libraries specific to the serial and parallel solution of numerical partial differential Equations (PDEs). OVERTURE [2] handles complex geometries using either single or multiple (overlapping) grid mechanisms. From these mechanisms we build a set of higher level abstractions in AMR++ specific to simplifying the development of AMR applications. AMR++ adds the ability to handle adaptive mesh refinement within OVERTURE applications. OVERTURE is based upon the A++/P++ array class library, where it obtains its architecture independence on both serial and parallel architectures. A++/P++ [3] is a C++ array class library which provides the architecture independence through FORTRAN 90 like syntax and encapsulates the data parallelism of array statements. Other mechanisms within OVERTURE use the HPC++ thread library to permit access to task parallelism. Using these tools the user may develop serial code using the OVERTURE Framework and recompile that code to run it on a wide number of parallel architectures.

AMR++ was developed to simplify the development of Adaptive Mesh Refinement applications. It is designed to leverage an existing serial application as completely as possible and permit its reuse within the context of an AMR application with minimal extra work.

OBJECT-ORIENTED DESIGN

An example of the adaptive mesh refinement grid is presented in figure 1. This sort of adaptive grid is referred to as a structured adaptive grid due to its construction from structured grid pieces. An alternative is the use of cell by cell refinement which permits the refinement of only regions where refinement is desired but has higher overhead and can reduce to an unstructured organization of the underlying data which complicates its use in the parallel environment.

The AMR++ design is organized into objects that represent the adaptive grid structure and that provide specific functionality with those representing the adaptive grid. The organization of the adaptive grid is split into three parts, each of which is optionally represented by a derived object build by the user. The adaptive grid is divided as represented in figures 2, 3, and 4.

- grid patch: which defines the single grids used in the Structured Adaptive Mesh Refinement (SAMR) structure
- refinement level: which defines the collection of grid patches that represent the same resolution
- adaptive grid: which defined the collection of refinement levels

Within each abstraction (i.e. grid patch, refinement level, and adaptive grid) the user can derive objects and use them to permit the customization of the AMR++ library for general application use. The derivation from the refinement level and adaptive grid levels of abstraction are optional and thus far in the development of AMR applications not used. The derivation from the grid patch level of abstraction is required and is the principle mechanism by which the solution process and the equations being solved are specified. Numerous other opportunities exist within the design of AMR++ to customize details of the behavior of the library's functionality. The extensibility of the AMR++ library is an explicit consideration in its design.

EXTENSIBILITY OF OBJECT-ORIENTED LIBRARIES

AMR++ has an extensible design, meaning that it is readily customized to the specifics of different types of adaptive mesh refinement applications. The point of the library is to abstract as much as possible of the reusable parts of adaptive mesh refinement and thus simplify the development of AMR applications. However, the specialized details and differences between both applications and numerical methods complicates this task. Addressing the increasingly more complex applications invariablely implies ignoring increasing amounts of the AMR library, unless it can be made readily customizable. There are two principle mechanisms that AMR++ uses to provide this extensibility, *virtual functions* and *template specialization*, both are features of C++ and its template mechanism. We have designed AMR++ to permit the user to readily use these features to customized the behavior of the AMR++ library.

C++ DERIVATION AND VIRTUAL MEMBER FUNCTIONS

Derivation in C++ object-oriented design refers to the use of a base class to define an interface and the use of the base class within the construction of more specific classes. Member functions of the base class

[2] Information is available from the OVERTURE Home Page: http://www.c3.lanl.gov/~henshaw/Overture/Overture.html.
[3] Information is available from the A++/P++ Home Page: http://www.c3.lanl.gov/~dquinlan/A++P++.html.

are defined to be *virtual* if the derived class is intended to be able to redefine those member functions. That these functions are *virtual* also means that member function calls to the base class within the AMR++ library instead call the derived class member function if it is defined. Thus the use of derivation combined with virtual functions provides a mechanism for the user to derive from AMR++ classes and customize its execution behavior, however it requires that the user derive a new class from an AMR++ class. However, forcing the user to derive new classes is often difficult as a design style for a library using multiple abstractions since it does not provide sufficient guidance to avoid mixing abstractions. Additionally, it may not often be required to derive new classes to customize the execution behavior. A much more comprehensive introduction to C++ derivation is available from any introductory C++ book. A simpler mechanism is the use of template specialization.

Within the design of AMR++ the use of derivation is reserved for the use within the application where significant portions of the application must be defined (in the case of AMR++ this is in the grid patch abstraction level, where the equations to be solved and the solution process is a significant part of defining the application). As much as possible we limit the places where derivation is required since this otherwise burdens the user with a much more complex interface and does so within the initial learning curve of developing an initial AMR++ application (in our experience, the most troublesome time to force the user to deal with such complexity).

As an example, in an AMR++ application we define the member function: *User_Solver<T>::error()*. In this case the User_Solver is derived from a base class provide by AMR++. Note that different applications would have different definitions of the *error()* function on a grid patch.

```
// This function overloads the base class
   void User_Solver<T>::error ()
      {
     // Array object encapsulate the loops over the elements
        Error = abs(Solution - Exact_Solution);
      }
```

Here the base classes *error()* member function is virtual and by using derivation AMR++ can refer to the *User_Solver<T>::error()* by only referencing the base class member function. This permits the user in the derived class to customize the behavior of the *User_Solver<T>* class (i.e. the user's representation of the solver on a grid patch) beyond that which the AMR++ library would provide through its base class. This is just one mechanism for the customization of the AMR++ behavior, but it required derivation. Since the equations and the numerical method to be used must be specified anyway, derivation of this single class is an acceptable method of specifying these detail (i.e. a large granularity functionality is being defined). However, we seek less burdensome methods for tailoring the behavior of other parts of the AMR process.

C++ TEMPLATE SPECIALIZATION OF MEMBER FUNCTIONS

Template specialization is a mechanism specific to the C++ templating mechanism. It permits objects with a template interface to have their member functions defined for the specialized use with particular template parameter. Thus we can define a template based class in AMR++ which defines a default behavior, and permit the user to optionally redefine the same member function for the case of a specific template parameter. This mechanism of redefining member functions of AMR++ classes is similar to the virtual function mechanism, however it avoids the requirement of the user building any additional classes. Thus the use of template specialization within AMR++ is specific to individual member functions of specific library objects, it therefore avoids the mixing of abstractions.

As an example, in AMR++ we define the member functions: *AMR_RefinementSolver<T>::error()* and we demonstrate how the user can use the template specialization mechanism to rewrite this function. As an example, in AMR++ we define the member functions: *AMR_RefinementSolver<Users_Solver>::error()* using the template specialization mechanism. Independent of the definition in AMR++, the user can use template specialization to customize the implementation of the member functions of AMR++ object directly (without derivation of new objects which would complicate the interface).

```
double AMR_RefinementSolver<Users_Solver>::error()
  {
    double Total_Error = 0.0;
  // STL lists are used throughout AMR++ and so the
  // following loop and iterator syntax is due to STL.
    for (list<Users_Solver*>::iterator n=GridSolverList.begin();
         n != GridSolverList.end(); n++)
       {
         Total_Error += (*n)->error();
       }

    return Total_Error;
  }
```

In the *AMR_RefinementSolver<Users_Solver>::error()* example we have specialized its execution independent of the definition of the function's templated implementation within AMR++. This demonstrates

the mechanism by which an application can customize the definition of AMR++ over that of its default behavior. Note that this mechanism is similar to the virtual function mechanism in terms of what it provides the user, however it does not require the derivation of a class from the AMR++ AMR_RefinementSolver object. Thus by combining the use of virtual functions, where we want to provide an interface and intend to use derivation, with template specialization, where we want to provide customization without forcing derivation; we can provide a more powerful mechanism than the exclusive use of either mechanism. In particular, the combination of these mechanisms is superior to that of using only derivation as is done within many object-oriented libraries.

CONTRASTING THE TWO MECHANISMS

The most recognizable difference between the two mechanisms is that the use of virtual functions to modify the behavior of AMR++ requires the user to derive an object from an AMR++ object, while the member function template specialization mechanism avoids the introduction of any new classes. The avoidance of introducing additional classes as part of the interface avoids the confusion associated with mixing or complicating the abstractions, this simplifies the user interface for AMR++. This further simplifies the user's ability to get simple AMR examples implemented quickly and separates the work required to learn how to use AMR++ along the learning curve. We feel this simplifies the user's initial and later use of AMR++.

CONCLUSION

The design of AMR++, as an example, and more general scientific libraries in particular requires simple mechanisms to permit extensibility of the library for widely different sorts of applications and numerical methods. This requirement forces the consider of mechanisms to that permit the object-oriented libraries to be customized for a wide number of applications and numerical methods that the library developer can not be expected to predict in advance. This paper has demonstrated two different mechanisms that are of use in the development of extensible libraries and demonstrated there use within the AMR++ library. Together these mechanism permit the execution behavior of the object-oriented library to be tailored to individual applications.

REFERENCES

Brown, D., et al. 1997. *OVERTURE: An object-oriented software system for solving partial differential equations in serial and parallel environments.* Proceedings of the SIAM Parallel Conference (Minneapolis, MN, March 1997).

Quinlan and Parsons. 1994. *A++/P++ Array Classes for Architecture Independent Finite Difference Computations.* Proceedings of the Second Annual Object-Oriented Numerics Conference (Sunriver, OR, April 1994).

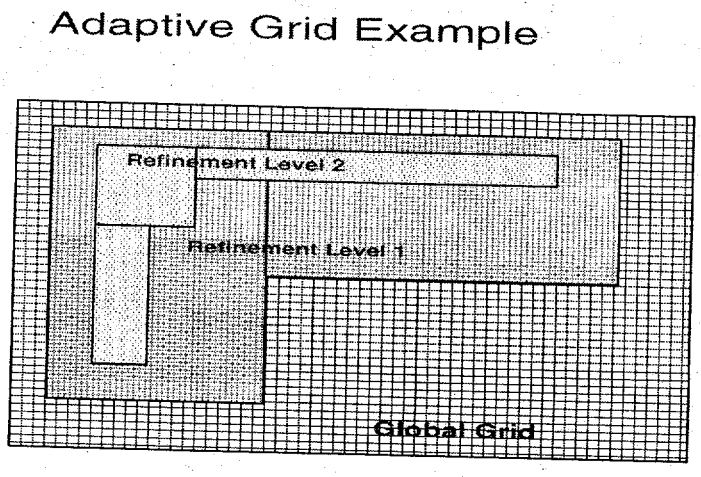

Figure 1: Example Adaptive Grid

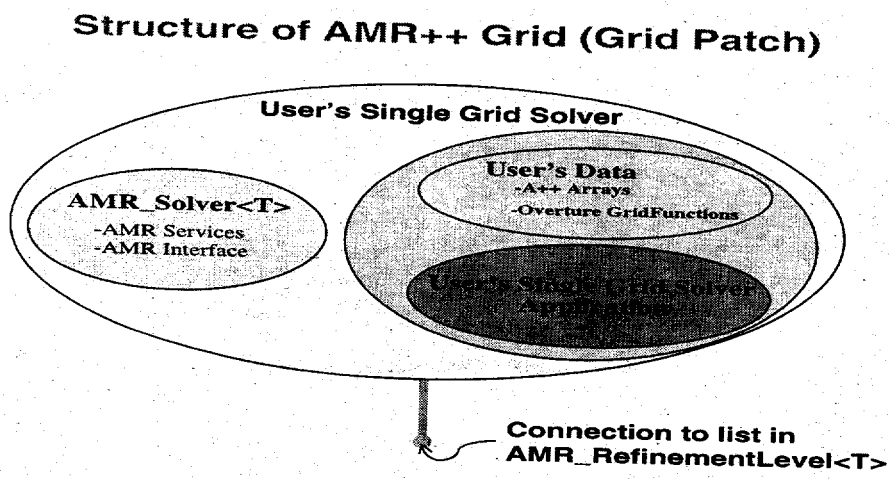

Figure 2: Grid Patch Solver

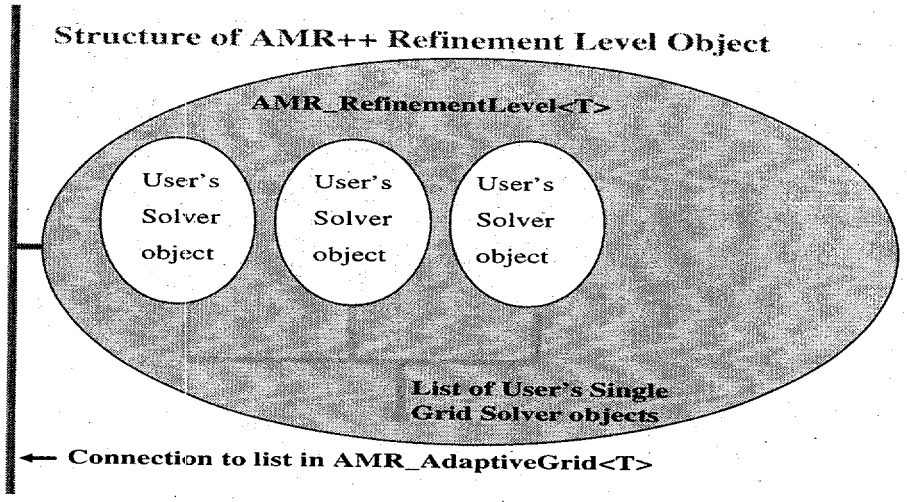

Figure 3: Refinement Level

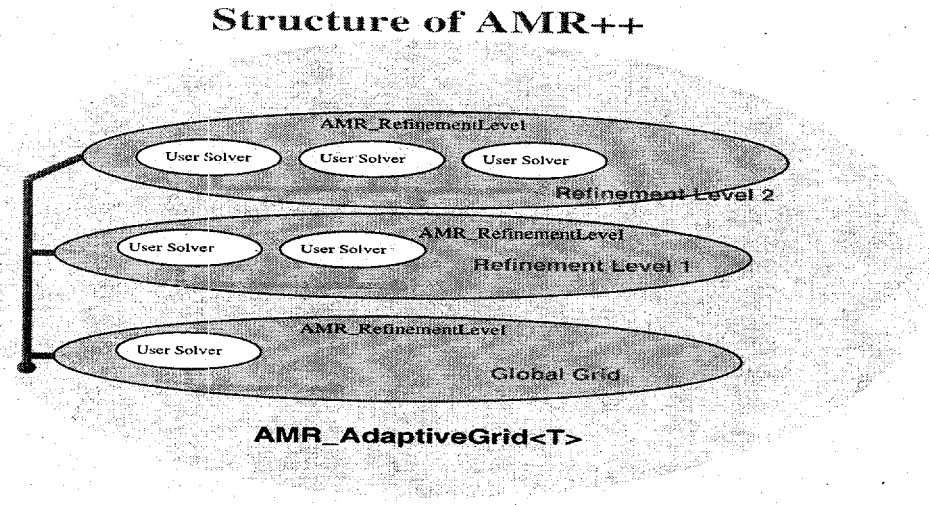

Figure 4: Adaptive Grid