FUSION RESEARCH CENTER

DOE/ER/53266-50 FRCR #454

AN OBJECT-ORIENTED DECOMPOSITION OF THE ADAPTIVE-HP FINITE ELEMENT METHOD

PROCEEDINGS OF THE SECOND ANNUAL OBJECT-ORIENTED NUMERICS CONFERENCE
SUNRIVER, OREGON
APRIL, 1994

J.C. Wiley
Fusion Research Center
The University of Texas at Austin
Austin, Texas 78712-1068

December 13, 1994

THE UNIVERSITY OF TEXAS

Austin, Texas

Conf-9404121--4

An Object-Oriented Decomposition of the Adaptive-hp Finite Element Method

J. C. Wiley
Fusion Research Center
The University of Texas at Austin
Austin, Texas 78712

Abstract

Adaptive-hp methods are those which use a refinement control strategy driven by a local error estimate to locally modify the element size, h, and polynomial order, p. The result is an unstructured mesh in which each node may be associated with a different polynomial order and which generally require complex data structures to implement. Object-oriented design strategies and languages which support them, e.g. C++, help control the complexity of these methods.

Here an overview of the major classes and class structure of an adaptive-hp finite element code is described. The essential finite element structure is described in terms of four areas of computation each with its own dynamic characteristics. Implications of converting the code for a distributed-memory parallel environment are also discussed.

Introduction

Finite element techniques have been successfully used in engineering calculations for many years and have a rich history of fundamental mathematical support. Comparatively recent developments have shown that adaptive-hp finite element methods can achieve exponential as compared to polynomial decrease in the solution error for increasing resolution for certain problems and can therefore make feasible the solutions of problems not previously practical on a give size machine[2][4]. Adaptive-hp methods are those which use a refinement control strategy driven by a local *a posteriori* error estimate to locally modify the element size, h, and polynomial order, p. The result is an unstructured mesh in which each node may be associated with a different polynomial order. While these methods have great promise, they are not widely used. Part of the reason is that the data structures needed to support the adaptive hp method are complex. Object-oriented design strategies and languages like C++ present the engineer an opportunity to restructure the design of finite element codes to control the complexity of these modern methods. A typical design strategy for a finite element code examines the operations and information need

for each stage of the computation and then attempts to create a data structure which contains the minimum information needed for the entire calculation. Many efficient data structures have been published. The problem is that even in well written codes the details of these data structures then permeated all areas of the code and make understanding, maintaining, and modifying the code difficult. Significant modifications to the data structure generally require a complete rewrite of the code. Object-oriented design changes the emphasis on data structure and through the mechanism of encapsulation attempts to isolate the details of the data structures from the algorithmic considerations.

In this paper we present an object-oriented design of an hp finite element code primarily intended for transient nonlinear parabolic calculations. The object oriented design is based on work by Devloo[3] and Rude[7] and earlier hp work by Oden[2][4][6] and others. We first describe a typical hp finite element and the constrained 1-irregular meshes. A typical finite element calculation is then outlined and the additional steps required by the adaptive mechanism are discussed. Next we describe a set of abstract classes called grids, meshes, and nodes which are used to organize the actual class structure of the problem and to establish communication patterns and responsibilities. One of the goals of the design is to allow the code to easily migrate to a parallel distributed-memory message-passing computational environment. The problem is then divided into four main areas: geometric, topological, algebraic, and physical/material. Each of these areas is then divided into grids, elements, and nodes. The flow of the overall calculation is driven by an analysis class which present high level operations to the main application program.

Object-oriented class libraries are typically considered from two view points, that of the customer and designer. With scientific calculations, however, one finds the user often changing viewpoints going from one in which the library is viewed as a block box to one in which the scientist/engineer would like to explore new algorithms that require changes to the internal mechanism. One goal of this object-oriented design is to organize the calculation into a hierarchy which allows the user/designer to easily make modifications at any level with minimum propagation throughout the remainder of the code. This requires that each major component present a well defined interface which minimizes access to its internal data structures. With object-oriented design we are effectively replacing the minimum information data structure requirement with a minimum interface requirement.

Adaptive Finite Element Method

The adaptive finite element method typically starts with a partial differential equation, PDE, expressed in weak bilinear form over a given domain with appropriate boundary conditions. The domain is partitioned into elements and the solution is expressed in terms of continuous polynomials within each element and usually having C^0 continuity between elements. Each element is mapped to a master element and integrals are performed elementwise in normalized coordinates

using the Jacobian of transformation. The element integrations yield relatively small dense matrices and vectors which are then assembled (implicitly) into a large and usually sparse linear system. This system is then solved either directly or iteratively. A error estimate is then made. If the error is not below a specified tolerance, the partition is modified and the solution repeated. Nonlinear problems require additional iterations.

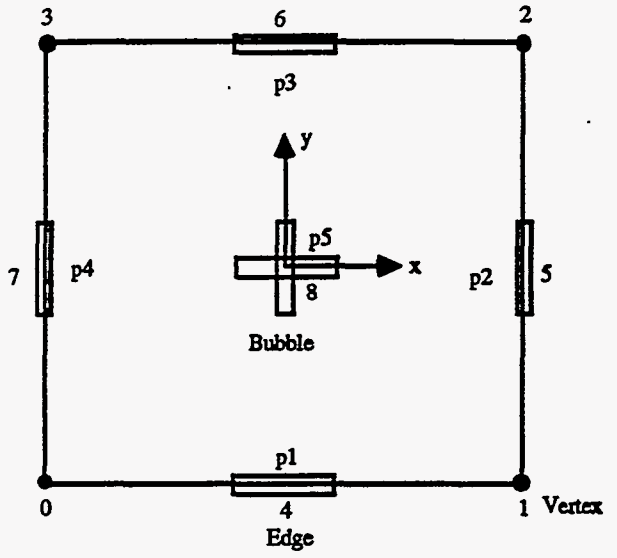

Figure 1: Master element.

The nine point quadrilateral shown in Fig.(1) is an example of a typical master element for two-dimensional calculations[5]. There are three different types of nodes and associated basis functions: corner, edge, and bubble. The basis functions are constructed from tensor products of the hierarchical polynomials. The corner functions are nonzero at only one corner node and fall linearly to zero at each of the other corners. For bilinear approximations, only the corner basis functions are used. Edge functions represent quadratic and higher order polynomials which are nonzero along one edge, vanish at the corners, and fall linearly to the opposite edge. Bubble functions are tensor products of higher order polynomials and vanish along the edges. The number of degrees of freedom, dof, associated with an element is $4+\sum_{i=1}^{4}(p_i-1)+(p_5-1)^2$ where p_i is the polynomial order. For example if each of the edge nodes is fourth order then the bubble node would typically have nine basis functions for a total of 25 dof. The element matrix associated with this element would then 25×25. It is not uncommon to have up to sixth or eight order polynomials appearing in the problem. The size of element matrices can be seen to range from

4×4 for linear elements to 49×49 for sixth order polynomials for a single equations. For systems, these sizes are multiplied by the number of equations. The goal of the p-adaptive finite element method is to choose an optimal p for each node in the grid. Due to the large variation in storage required to describe each element, dynamic storage allocation is important for storage efficiency.

For systems of equations like those found in fluid calculations, primary variables may require different polynomial approximations for consistency. For example in continuous pressure approximations, the velocity components need to be one or two orders higher[9]. This adds additional complexity to the description of the element as each component of the system may require a different polynomial order. When computing the dof for an element, the equation type as well as the node degree must be considered. The point is that the descriptions of these elements can be rather complex, and their properties change significantly throughout a calculation. A dynamic object provided by a object-oriented language can more easily accommodate the complexity than the data structures typically used in more procedural languages. Further, the object-oriented approach provides a mechanism for hiding this complexity in a small part of the code.

The second feature of the adaptive-hp method is local mesh refinement. A small region of a mesh that has two levels of refinement is shown in Fig (2). Note that the mesh contains irregular nodes. A node in the mesh is regular if it is a vertex for each neighbor element, otherwise, it is irregular. If the maximum number of irregular nodes on an element side is one, then the mesh is 1-irregular[2]. There are essentially two strategies for dealing with local refinement. One strategy does not allow irregular nodes and creates blending elements as necessary[1]. The second allows irregular nodes but additional constraints are placed on the basis functions to enforce continuity of the solution. In the hp-methods considered here, we restrict the mesh to having only 1-irregular nodes. Further the mesh refinement is achieved by dividing elements so that the refined elements are nested. The mesh retains a quasi-regular property, however, this property also places additional constraints on the mesh refinement algorithm. Refining one node may create a chain of refinement that propagates across the mesh and in particular in a multiprocessor environment may cross processor boundaries. It is this property that requires the refinement algorithm maintain a global view of the grid.

The adaptive-hp method attempts to reduce the number of degrees of freedom needed to achieve a prescribed accuracy by optimally modifying the local polynomial order of approximation, p, and the mesh size, h; and thereby reduce the size of the linear system to be solved. This more optimal approximation is achieved at the cost of increased complexity in both the elements and the mesh and it is in controlling this complexity that the object-oriented approach makes a significant contribution.

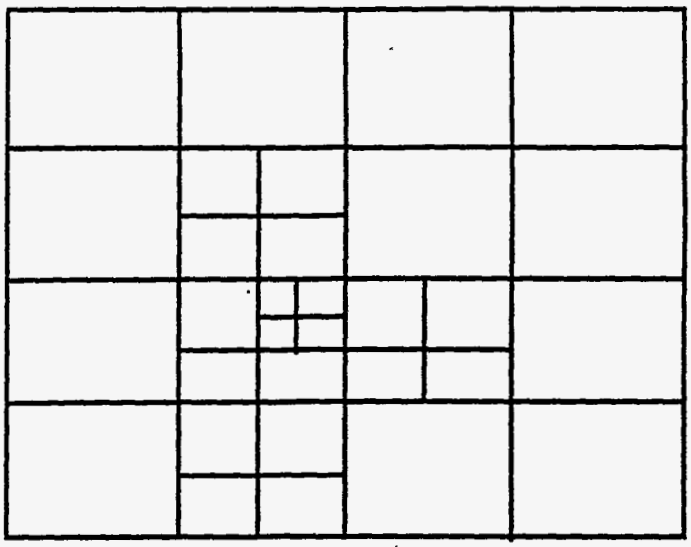

Figure 2: A 1-irregular mesh.

Design

A typical finite element problem can be divided into these areas: partition maintenance, element integration, linear system solution, error estimation, mesh refinement. A typical finite element code design would examine each of these activities and determine the minimum data required for each activity. A data structure would then be developed and each of the activities coded in terms of the data structure. The object-oriented method modifies this design strategy by first identifying the components of the system and then associating the activities with the components. This has been called a noun based rather than a verb based design. Object-oriented designs of finite-element codes often focus on the element as the main object and then consider subclasses of the element base class to represent triangular or quadrilateral elements of different orders. Here, while we do consider different types of elements, the primary focus is on handling the complexity of the 1-irregular gird and variable elements.

The problem is divide into four main areas: the refinement tree (geometry), a domain partition (topology), the equation and material (physical/material), and the linear system. Each of these areas has a different lifetime during the calculation and in a parallel environment has a different extent across processors. The geometric partition is responsible for maintaining the relevant history of the grid refinement and unrefinement. Its lifetime is that of the calculation and it must

have a global awareness across processors. When a solution is to be constructed, a particular partition of the domain must be selected. This is called the topological partition since it is composed of a logical grid of master elements. This partition is used to compute the matrix elements. The physical object contains the description of the differential operators and evaluates the integrands using the material coefficients supplied by the material object. Finally, the linear system object is responsible for organizing the element matrices and solving the global linear system. It does not necessarily assemble the global linear system. Subclasses of these class can implement both direct and iterative linear solvers.

Abstract Classes

Finite elements deal with grids or arrays of elements, each of which is defined in terms of nodes. The triple: grid, element, node, can be abstracted so that in general a grid can be considered to be a database object which is used to store and manipulate elements and nodes. A grid represents the whole domain, an element represents a small area, and a node a point like object. We have used this generalized nomenclature to divided each of the four areas into three parts. A grid provides a container class for element and nodes. It maintains a global view of the structure and provides for inter-element communication. It can be considered to be a database with defined access methods. The elements maintain local element information and element methods. The elements are associated with area like objects. The nodes maintain local point information. We have, for example, a geometric grid, a geometric element, and a geometric node. Similarly, we have a topological grid, a topological element, and a topological node. The same terminology can be used for the linear system in which an algebraic grid represents the global linear system, a algebraic element is an element matrix, and a algebraic node is a single entry. While this abstraction perhaps should not be pushed too far, it was useful for the initial design.

Geometric Grid

Consider some arbitrary region Ω as shown if Fig.(3). The region is divided into elements, with possibly curved sides. Each element is transformed by an invertible map into an image of a master element. The geometric grid operates on a logical grid of these master elements. Traditionally these transformations are expressed in terms of the same basis functions that describe the solution function. Here we allow the transformations to be expressed in terms of a similar but not necessarily identical basis set. The description of the decomposition of the domain and the transformation coefficients for each element are maintained by the geometric grid class. The domain is initially partitioned and this root level remains fixed, then, as the solution proceeds, elements are refined and unrefined.

The domain partition can be represented as a tree, Figs.(4) (5), usually an oct-tree in three

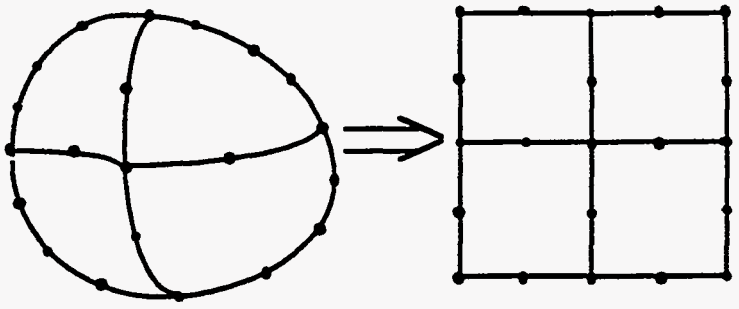

Figure 3: Region Ω and the geometric grid.

dimensions or a quadtree in two dimensions. It is necessary to maintain this tree throughout the solution so that elements can be unrefined as well as refined. (Unused leaf nodes can of course be deleted.) The tree is also used for multigrid schemes as the tree can contain both coarse and fine partitions of the domain. We use a representation of the tree developed for N-body particle simulations[8]. Each element is given a key that is derived from its location in the tree and can be related to its location in the logical grid. In Fig.(5) the binary representations of some keys are shown. When an element is divided into four new elements the keys for the new elements are constructed by multiplying the parent key by four and adding 0, 1, 2, 3 respectively for the new elements. These keys are used instead of pointers as references to the elements and provide a global address space in a multiprocessor system. The coordinates of the nodes are stored in node objects which are labeled by a similar key derived from their coordinates. The elements store the node keys not the actual coordinates. By setting the high bit in the node keys, the node keys remain distinct and the nodes can be stored in the same hash table as the elements. The structure of the key allows simple algorithms for finding the keys of an element's parent or neighbors to be constructed. Neighbors of elements in the initial partition may have to be explicitly stored, if the connectivity is different from a regular grid. Elements can also be marked as holes which are not to be refined. This allows multi-connected regions to be described.

In Fig.(5) the geometric grid contains many possible decompositions of the domain. For example suppose the original partition was P_1 ={4, 5, 6, 7}. Refining elements 4 and 6 produces the partition P_2 ={16, 17, 18, 19, 5, 24, 25, 26, 27, 7}. Unrefining element 4 and further refining element 24 gives the partition P_3 ={4, 5, 96, 97, 98, 99, 25, 26, 27, 7} which of course violates the 1-irregular constraint and is therefore not an acceptable partition. Note that the active partition is not necessarily the leaves of the tree. If the algorithm is required to keep track of the previous partitions, then there is a problem of where to keep the partition information. Here,

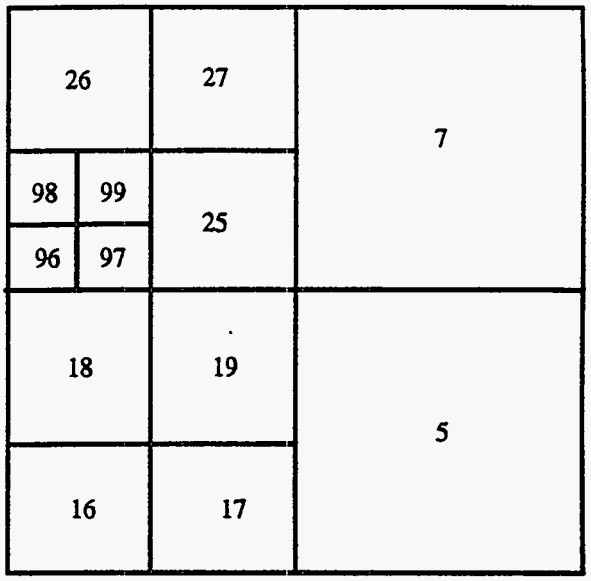

Figure 4: Partition of simple region.

partition information is not kept in the geometric grid but in the topological grids which are discussed below. The geometric elements, however, do contain flags that can be set or cleared to denote membership in particular sets. Geometric elements have member functions which allow set operations to be performed. These flags are used for example to mark elements for refinement or unrefinement, to mark elements as belonging to the current active topological grid, etc. Iterators are constructed to operate on sets of elements.

The important properties of the geometric grid are that it exists for the lifetime of the calculation, it stores the transformations from each element to the master element, and it is responsible for computing the Jacobian. It maintains other information about the elements such as whether the element touches a boundary, whether the element represents a hole in the region which can not be refined, etc. The geometric grid is also responsible for implementing the h-refinement strategy. This entails applying a refinement strategy based on an elementwise error estimate and imposed constraints on the grid to select elements to be refined or unrefined. The 1-irregular constraint on the grid means that the refinement of one element can trigger the refinement of other elements. The geometric grid, therefore, must maintain a global view of the grid in both space and time.

Summarizing properties of the Geometric grid:

- maintains metric information

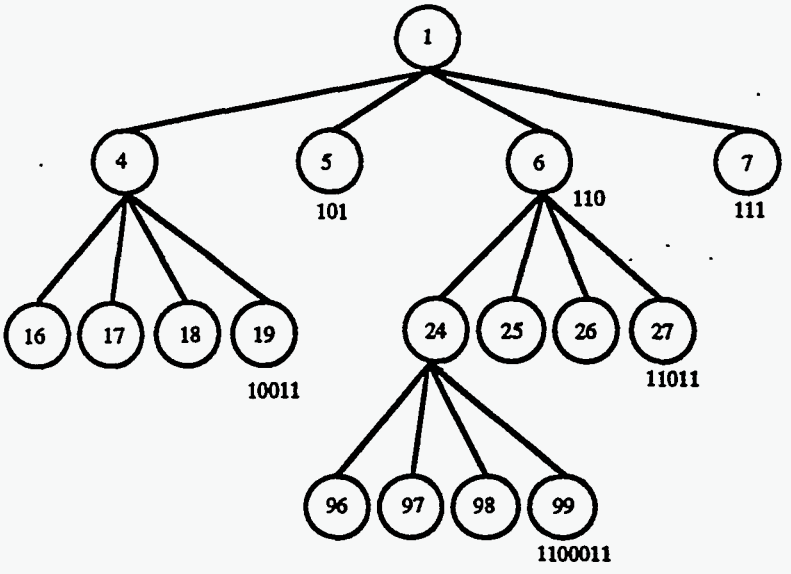

Figure 5: Quadtree associated with partition in Fig.(4).

- evaluates geometric basis functions and computes Jacobian
- h-refinement algorithm
- maintains refinement tree
- global view of mesh across processors and domains
- draws shape of elements
- lifetime of calculation but changes dynamically

Topological Grid

The topological grid serves as the container class for the topological elements. The topological elements correspond to the usual finite element and have methods that perform the usual finite element functions. They contain the description of the local approximating polynomials and are

responsible for computing the polynomial evaluations. They control the Gaussian quadrature to compute the element matrix elements. There is a topological base class that contains all common functionality for the topological elements and subclasses to represent specialized elements such as quadrilaterals or triangles.

The topological elements and nodes are maintained by the topological grid class. The topological grid is instantiated with an iterator over an acceptable partition of the geometric grid. It is responsible for creating the topological elements and nodes, and interpreting the topology so as to recognize constrained nodes and other special situations. Unlike the geometric grid, the topological grid is a static structure since the number of elements and their sizes are known when the topological grid is created. This means that the data structures for the topological grid are simpler and, for example, array indexing can be used. The topological elements can be organized in memory to maintain locality during the stiffness matrix calculations. In a multiprocessor system, the domain can be divided into many topological grids with one or more grids per processor. The topologic grids need to interchange boundary data, however, most of the element matrix computations can proceed in parallel. Each topological element has a key pointing to its corresponding geometric element. This key allows the geometric element to be called to compute the Jacobian as necessary. The topological grid in effect saves all the keys for a particular partition in the geometric grid. By allowing all topological elements to set a flag in the corresponding geometric element, we have a mechanism for recovering partitions.

The topological grid creates not only elements and nodes, but objects of a class called tags. Tags contain the actual unknowns for the problem once the geometric constraints have been removed. The reason that the unknowns have been separated from the nodes, is that in order to satisfy certain constraints it is useful to have the same unknown set associated with different nodes. We also maintain the unknowns in a tag as a vector. The unknowns are not individually numbered but only as tags. This means that the smallest unit in the global matrix is dense matrix of the same order as the number of unknowns in a tag. By making the smallest unit of arithmetic operation a relatively small, dense matrix-vector operation, we attempt to maintain locality and capitalize on the operational efficiency of RISC processors.

The topological grid is also responsible for computing the local error estimate and for implementing the p-refinement strategy. The topological grid only needs to exits while the solution on a particular partition of the domain is being found. Once the solution is computed and an error estimate generated, then a new topological grid is created and the old one can be deleted. For multigrid solution, however, there may be several topological grids active at one time. The lifetime of topological grid is therefore typically shorter than that of a geometric grid.

Summarizing the properties of the topological grid:

- size and structure known when created
- may be local to a processor

- maintains normalized partition at single refinement level
- orders unknowns
- evaluates solution basis functions
- does Gaussian integration to compute element matrices
- p-refinement algorithm
- *a posteriori* error estimate

Physical and Material Grids

When a topological grid is initialized, it is given a physical element and a material element. The physical element describes the particular differential operator being used. One of the more powerful features of finite element methods is the great range of differential equations that can be accommodated by the same code. Each physical element corresponds to a different problem. A database (grid) of different physical elements can be maintained. For a particular differential operator, different problems can be solved by simply changing the coefficients in the differential equations. In much of the original finite element literature, these are known as material properties and hence the name. Once a domain and differential operator have been defined, then different physical situations can be described by simply changing the material elements. A user who wants to modify the material properties of a problem that has already been programmed then only needs to subclass the material coefficient class. A user who wants to solve a new equation set only needs to modify the physical element class.

Summarizing the properties of the physical and material classes

- defines differential operators
- implements boundary conditions
- does Gaussian quadrature
- defines equation coefficients

Algebraic Grid

The finite element method ultimately leads to a large linear algebra problem. The global matrices can be either explicitly constructed or maintained implicitly in terms of the element matrices. Both direct and iterative solution methods are used. The common abstract global linear algebra

```
void  TAnalysis::StiffProb(TTopoGrid* topoGrid, int solverType)
  {
    TAlgeGrid *algeGrid;
    algeGrid = new TAGSparse(topoGrid);
       algeGrid->StiffToSSB();
      algeGrid->MoveSSToAA();
                ....
     algeGrid->Solve(TAlgeGrid::eBiCG);
     algeGrid->XToTopoGrid();
   delete algeGrid;
  }
```

Listing 1

problem is encapsulated into the algebra base class. This class is then specialized for explicit or implicit storage of the global linear system, and for different solution techniques. Each of these classes shares a common interface with the topological class. Each topological element has a method which creates an elemental mass, or stiffness matrix and the corresponding list of tag numbers. The algebra class is responsible either for the assembly of the corresponding global matrix or storage of the elemental matrices. This creates a clean separation of the problem of computing the local element matrices from the global linear algebra problem. The common interface is through comparatively small dense real matrix and vector objects. The algebra class only needs the matrix coefficients, the associated unknown number, and possibly certain characteristics of the unknowns.

The algebra class is designed as a simple calculator that supports load and store operations between the registers and the topological grid. Other operations include move operations between registers. Two of the registers are designated as A, b registers respectively to represent the linear system $Ax = b$. When the solve operation is invoked, the linear system is to be solved and the result is placed in register x. The algebra class is a base class which is subclassed for different representations of the global matrix structure. For example, the global matrix can be assembled into a sparse matrix, and then either direct or iterative methods can be used to solve the system. For iterative methods that operate only on the element matrices, the algebra class is subclassed by a class that stores the individual element matrices. The algebra class provides a common template to describe which global linear algebra problem is to be solved. An instance of a subclass of the algebra class is given a topological element iterator so that it can obtain each of the element matrices. It is the particular subclass's responsibility to handle the stream of element matrices and solve the linear system, when requested. At this point any of the interesting C++ libraries that are being developed for large linear system and especially those developed for parallel sparse

systems can be used without modifying the remainder of the code. Listing (1) shows a code fragment from a method in the analysis class that is used to solve elliptic problems. A subclass of the algebra class that implements a global sparse matrix is constructed. The SS and B registers are then filled with the stiffness matrix and the load vector. The SS matrix is then moved to the AA register and the system is solved using a bi-conjugate gradient method. The result is in register X which is then stored into the topological elements.

Note that while the algebra class may be responsible for the major storage usage and usually requires the majority of CPU time, it has the shortest lifetime of the major components.

Summarizing properties of the Algebraic grid:

- solves global linear system
- major cpu user
- shortest lifetime

Analysis Class

The analysis class is orchestrates the operation of the geometric grid, topological grid and the algebraic grid. Given a input data set, packaged as an input class, it handles the initial construction of the geometric grid. The analysis class also packages sequences of operations on the geometric, topological, and algebraic classes into higher level commands for the main program. The relationships among the main computational areas are shown in Fig.(6).

Supporting Classes

In addition to the major classes that define the finite element technique there are other classes and class libraries that are used. One essential addition to the usual C++ class libraries is a vector/matrix class. Here a simple vector class was constructed which defines the usual vector and dense matrix operations. This class is relatively isolated so that other more advanced libraries could be used to replace it.

There is also a class for describing the code input. This class was modeled after the input definitions required for PLTMG[1] and isolates the code from the form of the input file.

Some of the more useful classes defined are the iterator classes. The geometric grid and topological grid classes define friend iterator classes which allows looping constructs over the elements to be easily constructed. There are a number of geometric iterators to loop over all leaf elements, or all elements associated with a particular topological grid, or all elements etc.

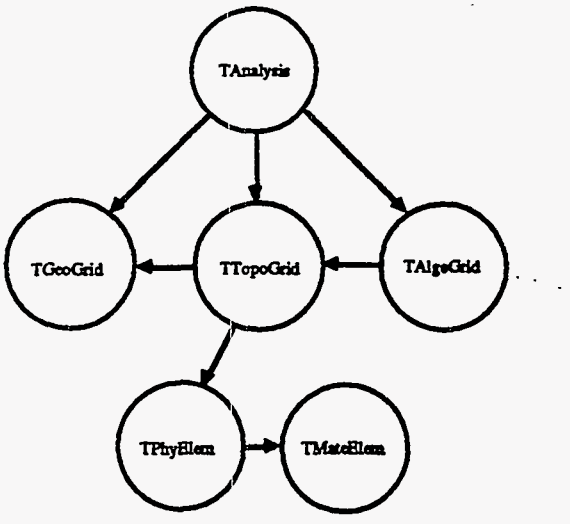

Figure 6: Relationships among components.

Conclusions

We found object-oriented design techniques useful in controlling the complexity of an adaptive hp finite element code. The problem was divided into different computational areas with different data structure requirements, varying computational lifetimes, and different extents in multiprocessor systems. By encapsulating each component in its own class with well defined interfaces, we were able to limit the range of influence of any particular data structure to a relatively small region of the code. This allows major parts of the code to be redesigned without changing the remainder of the code. The code then becomes simpler to build, maintain, and modify.

Acknowledgments

The author acknowledges with pleasure many extensive discussions with Phillipe Devloo at OON-Ski 93 on the class structures outlined in the paper. The author would also like to thank Prof. J. Tinsley Oden and coworkers especially Abani Patra and Yusheng Feng, for the introduction to adaptive-hp methods and William H. Miner, for many useful discussions. Work supported by the U. S. Department of Energy, Grant DE-FG05-88ER53266.

References

[1] Randolph E. Bank,*PLTMG: A Software Package for Solving Elliptic Partial Differential Equations, Users' Guide 6.0*, SIAM, Philadelphia, (1990).

[2] L. Demkowicz, J. T. Oden, W. Rachowicz, and O. Hardy, "Toward a Universal h-p Adaptive Finite Element Strategy, Part 1: Constrained Approximation and Data Structure," Comp. Meth. in Appl. Mech. and Engrg., 77, 79-112 (1989).

[3] Philippe R. B. Devloo, "On the development of a finite element program based on the object oriented programming philosophy", Proceeding of the First Annual Object-Oriented Numerics Conference, p.183 (1993).

[4] J. T. Oden, L. Demkowicz, W. Rachowicz, and T. A. Westermann, "Toward a Universal h-p Adaptive Finite Element Strategy, Part 2. A Posteriori Error Estimation", Comp. Meth. in Appl. Mech. and Engrg., 77, 113-180, (1989).

[5] J. Tinsley Oden, "Optimal hp-Finite Element Methods" TICOM Report 93-09, The Texas Institute for Computational Mechanics, The University of Texas at Austin, Austin, Texas, Sept. (1992).

[6] W. Rachowicz, J. T. Oden and L. Demkowicz, "Part 3. Design of h-p Meshes", Comp. Meth. in Appl. Mech. and Engrg., 77, 181-212, (1989).

[7] U. Rude, "Data Structures for Multilevel Adaptive Methods and Iterative Solvers", MGNet, Sept. 15, 1992.

[8] Michael S. Warren and John K. Salmon, "A Parallel Hashed Oct-Tree N-Body Algorithm", Proceedings of Supercomputing '93.

[9] O. C. Zienkiewicz and R. L. Taylor,*The Finite Element Method*, McGraw-Hill, New York, 4th ed., p121-122, (1989).