Node-based dynamic adaptive grid with quadrilateral and hexahedral elements

G.H. Wang\textsuperscript{a}, J.M. Tyler\textsuperscript{b}, J.S. Weltman\textsuperscript{c}, J.D. Callahan\textsuperscript{d}

\textsuperscript{a}Geoquest (a unit of Schlumberger), Houston, TX 77056-2722, USA
\textsuperscript{b}Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803, USA
\textsuperscript{c}Innovative Emergency Management, Baton Rouge, LA 70809, USA
\textsuperscript{d}InControl Technologies, Houston, TX 77068, USA

Abstract

A node-based data structure is developed for use in either two-dimensional (2-D) or three-dimensional (3-D) finite element method (FEM) problems. This grid uses quadrilateral (2-D) or hexahedral (3-D) elements and is dynamically adaptive according to the 1-irregular requirement for refinement–derefinement. Implemented in C\textsuperscript{++}, the grid consists of node objects linked with pointers to neighboring node objects. The algorithm used to traverse the grid starts at the first node of the grid and constructs each FEM element by following the links of the nodes. This algorithm uses a stack to maintain the correct order of elements during the grid traversals. Because the elements and their connectivity are not permanently stored, the computer storage requirements for the grid are reduced significantly in 3-D compared with an element-based approach. Although the node-based approach may increase the CPU-time and complexity to access an element, it reduces the complexity and time required to refine and derefine an element. This implementation is applicable to many FEM domains. © 1998 Published by Elsevier Science Ltd. All rights reserved.

Keywords: Dynamic adaptive mesh; Adaptive finite elements; Nodal based-data structures; Object-oriented programming; C\textsuperscript{++}; FORTRAN

1. Introduction

The finite element method (FEM) is widely used for solving partial differential equations (PDE’s) over complex domains [1]. The discretization of the domain is generally known as the mesh (or grid) generation. The quality and accuracy of the numerical result of FEM depend upon the discretization of the domain and the type of elements used. It is state-of-the-art to generate smaller elements in regions where the unknown PDE solutions vary the most and larger elements in smoother regions. This minimizes the demands for computer storage to store the mesh and related data and minimizes the CPU-time required for solutions without compromising the overall accuracy of these solutions. It is generally true that the initial mesh is not optimal for these unknown solutions being determined because we do not know a priori where the smaller elements are needed.

In a transient problem, such as those involving fluid flow, the regions of activity vary with time and a dynamic adaptive mesh is needed to capture this. If this fluid flow problem is an oil well driven by an aquifer, the original oil–water contact will change and be distorted after oil is produced. This type of transient flow problem requires that the grid adjust dynamically according to intermediate PDE answers and the answers from the previous time step. Adaptive grid refinement in FEM adds or rearranges the finer elements into the regions where the PDE solutions vary the most. For transient problems, being able to derefine is almost as important as refinement. Derefinement means that we will remove the finer elements in the smoother regions and release the computer memory that these elements occupied.

The implementation of the adaptive refinement method presented has not been trivial, especially in three-dimensions (3-D). The 3-D grids for transient FEM problems are very difficult to maintain. In these problems, one must store a very large and often growing number of nodes. To keep track of the elements that have been refined or derefined, we must store and update the rapidly changing connectivity of the 3-D elements. The computation time and memory required to maintain this grid may be overwhelmingly large, and this can diminish the benefit derived from the use of adaptive grids. Remeshing and h-refinement are two of many adaptive refinement schemes. Their benefits and disadvantages are briefly discussed in the following.
Remeshing is used to describe the regeneration of the mesh over the whole domain with the knowledge of a new estimate for the unknown solution. To achieve remeshing, the capability of automatic mesh generation is required and examples of this usually use octree–Delaunay methods.[2, 3] The efficiency of these methods is compromised by its generality. Remeshing has several disadvantages, as observed by Connell et al.: [4]

...adaptive remeshing has three disadvantages. The first is that the mesh generation algorithm needs to be robust and fast. If the scheme is not fast, the mesh generation time can quickly exceed the solution time,... The second disadvantage relates to the difficulty in computing transfer operators from unstructured mesh... Finally, unless the meshes are generated through solution adaptation a large percentage of nodes on the finer meshes may serve no useful purpose.

On the other hand, h-refinement is a method that tries to add finer elements in the local regions. h-refinement separates the mesh generation from the mesh refinement processes and is usually fast because only part of the initial mesh is modified. During the adaptive process in h-refinement, more efficient storage schemes are possible for the mesh than were possible when the mesh was generated. For example, [4] the data structure used to handle hexahedra in 3-D refinement resulted in ‘a reduction in CPU-time by up to a factor of 10 over the same case without multigrid.’

2. Element based versus node-based data structures

Grids (meshes) can be stored using different types of data structures. A node-based data structure stores the grid as a list of nodes in contrast with an element-based approach which stores the grid as a list of FEM elements. The element-based approach is proposed in several papers [2, 3, 5, 6]. The node-based approach, as well as element-based approach, is discussed in a paper by Ruede, [7] although implementation details are not given. In an element-based approach, each Element data structure has, at a minimum, a list of pointers to each of its nodes and a way to find its neighbors. In contrast, the node-based approach does not explicitly store elements. Instead, its node data structures contain pointers to each of their neighboring nodes. An element can easily be constructed from a single node by starting with its root node (node at the southwest corner of the element) and finding its neighboring nodes (the vertices of the element). This construction procedure is referred to as ‘assembling’ an element. Once assembled, the edges and nodes of neighboring elements can be found in a similar manner. The memory required is significantly reduced because typically only one element is needed at a time, and it does not need to be stored after use.

The element-based approach is more costly in required memory when compared with the node-based. The 1-irregular requirement [5] means that the difference of refinement levels between neighboring elements cannot be greater than 1. To enforce this requirement, it is necessary to locate all the neighbors of an element with a pointer to each neighbor. Since a 3-D element has a maximum of 48 neighbors, the element approach requires 48 pointers per element. Although the quadtree–octree approach [2, 3] could eliminate the need for pointers to neighbors, it would become unwieldy and time consuming to search a large tree for neighbors without an extra neighbor data structure.

The node-based approach allows efficient access to neighbors with fewer pointers, and thus less memory, but the trade off for this savings is that elements must always be assembled before being accessed. We believe, as others [8], the sacrifice in CPU time, if this sacrifice is small, is better than using more memory. The minimal extra CPU time and software complexity needed to assemble an element in our approach is also offset by a reduction in time and complexity over the element-based approach when we refine–derefine an element in 3-D. Refining an element in the element-based approach requires a complex and error-prone insertion of eight new elements into the linked list of elements. Each new element requires the correct placement in its linked list of 48 neighbor pointers as well as the correct placement of the neighbors pointing to it.

Because the use of less memory and reduced complexity were desired, we developed a node-based approach. Our decision was to only use quadrilateral–hexahedral elements. The node-based approach is not well suited to triangular elements [7].

3. Description of data structure and operations

Our data structure stores the 1-irregular h-refinement grid of quadrilateral or hexahedral elements as a doubly linked list of node-objects. Rather than maintaining different data structures for edges, faces, and elements, we chose to store only the node from which all other grid relations are obtained. Our node data structure records its four neighboring nodes (with four node pointers) in two-dimensions (2-D) or six neighboring nodes (with six node pointers) in 3-D. It also stores data associated with the node such as X, Y, Z coordinates and additional numerical data values. The client specifies how many additional numerical data values are stored per node. When an element is refined, new nodes are inserted into the doubly linked list. When a node is not used by any of the neighboring elements after derefinement, it is removed from the doubly linked list. This grid software was implemented in C++ on a Unix platform. The software implementation of the grid in 2-D and 3-D will be discussed in separate sections. Although the 3-D grid follows a similar logic as 2-D, the coding in 3-D was much more complicated.
4. 2-D node, element and grid

A two-dimensional 1-irregular quadrilateral grid is shown on Fig. 1. Each intersection of the line segments is a node. It is clear that there is a maximum of four line segments extending out from each node. An element is simply a group of nodes that are connected by these segments. This observation leads to the basic data structure that was used in object node, that is, each of these connections of a node is represented by a node pointer. Therefore, an object node in 2-D has four pointers to other nodes. The only exceptions are the edge nodes or the nodes created during subsequent refinement processes and these can have null pointers. This data structure enables the storage of a 2-D grid as a linked-list of node objects. These node objects also store any data needed for solving FEM problems, such as the coordinates of the nodes and physical properties of the FEM domain at the node location. The solutions of the PDEs are also stored with the node objects.

A 2-D node object is presented on Fig. 2. Left, right, up and down are used as the mnemonics for these four pointers because they are similar to the directions of the pointers. In the computer program, an array of four node pointers is used to store these. Element 0 of this array stores the pointer to the right, and element 1 stores the up pointer, and so on. This enumerating scheme allowed very efficient code to be written. One example of this is presented later when a finite element is assembled.

A 2-D element consists of 4–8 interconnected 2-D nodes; see Fig. 3. The local numbering scheme used for the nodes in an element is annotated on this figure. This annotation starts at the south-west node, and the four corners are numbered counterclockwise as node 0, 1, 2, 3. These four nodes exist for all 2-D quadrilateral elements. The 0 node was used as the root of the element because one and only one element can be assembled starting from this node. The node shown on Fig. 3 between 0 and 1 was numbered as local node 4. Node 4 only exists if the neighboring element below had been refined. Similarly, nodes 5, 6, and 7 exist only if there had been refinement of the neighboring elements in the corresponding direction. The object element allocates an array of eight node pointers to store all the pointers to the nodes that belong to this element. When a middle node, 4, 5, 6, or 7, does not exist, the node pointer is set to null. With this local node numbering scheme, an element can be assembled efficiently given the pointer of node 0:

```c
for (int i = 0; i < 4; i++) {  
if (nodes[i] != dir[i] != dir[(i + 1)%4] != 0) {  
nodes[(i + 1)%4] = nodes[i] = dir[i];  
nodes[i + 4] = 0;  
// no middle node  
}  
else {  
nodes[i + 4] = nodes[i] = dir[i];  
nodes[(i + 1)%4] = nodes[i + 4] = dir[i];  
}  
}
```

Fig. 1. An example of the 1-irregular grid in 2-D.

Fig. 2. A 2-D node that has four directional pointers.

Fig. 3. 2-D element and the direction of assembly.
This code fragment is compact, and this is a result of the numbering scheme for the node and element objects. It assembles each element without requiring any prior knowledge of the refinement status of the neighboring elements. Note, the actual code must check for null pointers in case the node is not the root of an element. An efficient element assembling method is essential to this new data structure because it is called frequently throughout a FEM calculation.

5. Build and traverse the 2-D grid

The program that uses the grid (the client) requires that all the elements be visited to assemble the stiffness matrix in a FEM calculation. Because the elements are not stored, there must be a grid traversing algorithm that ensures all the elements will be visited once and only once. To traverse the grid is equivalent to locating all the root nodes of the elements. This task is achieved by using a stack for the Node pointers. Once an element is assembled, the configurations of its nodes and the content of the stack uniquely determine the next element to be visited in this traversal. The order of traversing, also a priori, determines the global node numbering scheme.

The head of the grid was chosen to be the south-west node in the grid. While traversing the grid, the relative position and the configuration of the elements are used to determine the next element to be visited. Our program currently visits elements row-first. To add a column-first traversal would be a trivial task. An example of the row-first traversal is shown in Fig. 4, where the order of elements traversed is annotated with underscored numbers. The global number of all the nodes is also shown in Fig. 4.

Our traversal algorithm is also illustrated by the content of the stack shown on Fig. 5. Starting from the head of the grid (global node 0 on Fig. 4), as an element is assembled, its node 3 is examined. Using Fig. 5, if the element is the leftmost element of the grid, e.g. element 1, 5, 12, or the south-west element of a group of refined elements, e.g. element 7, 14, 15, the pointer to its node 3 should be pushed onto the stack unless this element is located on the last row of the grid or the group. Once this information is stored, the terminal proceeds to the element at the right whose root node is node 1 of the current element. However, if the current element is the rightmost element of the grid, e.g. element 4, 9, 18, or the north-east element in a refined group, e.g. element 11, 20, there will not be an element to its right, and then the stack is popped. This pop provides the
root pointer of the next element. This scheme assures visiting all the elements once no matter how complicated the refinement pattern becomes.

Because this traversal determines the global node numbering scheme, a counter is set to 0 before the traversal begins. Then any unnumbered node, indicated by a negative global number, is numbered according to their counter’s current value, and the contents of the counter are increased by 1. The results of this numbering method are also shown in Fig. 4. This numbering scheme yields a smaller bandwidth for the stiffness matrix when it’s compared with a numbering of all of the refined nodes with the new nodes being the largest node number in the grid plus one. This reduced bandwidth represents an additional savings in storage for the FEM system.

Currently, our 2-D grid cannot contain a discontinuous boundary and the head of this grid must point to the first element of the first row. Because the coordinates of the node are stored in the node objects, many different domains can use this same scheme without modification. One example is a quadrant of a grid that could be used with a well as shown in Fig. 6. The connections of the nodes on the radius are represented with node pointers 0 and 2, while the connections of the nodes with the same radius are implemented with pointers 1 and 3.

6. Grid initialization

As already stated, many domains can be discretized with this grid structure. One example is the quadrant of a well block, in either 2-D (Fig. 6) or 3-D (Fig. 12). Reservoir domains often contain many wells and may require multiple well blocks. Domains which encompass a larger geographical region may also contain regions in which no nodes are needed. Different grid structures may require different initializations and traversal routines which are specifically designed for that domain. For more complex domains (e.g. reservoirs containing multiple wells), we have developed some additional guidelines which maintain a consistent and extensible grid. It is important to note that these guidelines are not always required for domains but are required for domains which have several different physical regions and are made up of a conglomerate of smaller sub-grids.

The consistency of Node pointers, see Fig. 3, means that all subsequent links are made in the desired direction. All grids must be fully connected, see the 2-D radial domain in Fig. 7. If we embed the grid in a larger grid, the problem shown in Fig. 7 could occur. There are five directional node pointers from node N, but there are only four node pointers (0 right, 1 up, 2 left, 3 down) for each node object, and this is inconsistent. If any of these pointers are removed then the desired connectivity is lost. Mislinked or unlinked node pointers destroy the structure of the grid. Each of the node pointers must either be connected to a neighboring node or be a null pointer. For simple domains, grid construction is not difficult. For more complex domains (those containing regions without nodes or with irregularly shaped domains), the grid construction is more difficult and requires additional work. A bound and scan method was developed for these more complex grids. This bound and scan method finds the extent of all regions within the domain, creating nodes, and scanning the existing nodes to link the Node pointers to the correct neighboring nodes. A doubly linked list of Node pointers is created by this method. In 3-D, the nodes are ordered by their physical location, starting from the upper, left, front corner of the domain and ending with the lower, right, back corner of the domain. During the grid initialization, a temporary index is created for this list. With the temporary list, a binary search for a particular node requires only $O(\log n)$. This index is unallocated after the grid is constructed. The process involves these basic steps.

/* Proceed from upper, left, front corner of domain to lower, right, back corner of domain */

While (< within range of domain >) {
< determine location of prospective node >
if ( < node should be created at this location? > ) {
  < Create a new node >
  < search for upper neighbor and link to node >
  < search for left neighbor and link to node >
  < search for back neighbor and link to node >
}

This initial grid construction requires the determination of whether a node needs to be created at each potential node location. A computer screen is used to select node locations relative to the domain boundaries. The initial node locations are displayed on a computer screen to provide interactive intervention if desired later. When the node locations \((x, y, z)\) are final, the node structure is inserted into the doubly linked list and the node pointers are linked to its neighboring nodes. The temporary list of nodes is scanned in descending order until the neighboring node is found, and it stores this node’s location. The neighboring nodes to it are located at coordinates \((x - \Delta x, y, z)\), \((x, y - \Delta y, z)\) and \((x, y, z - \Delta z)\). The list is searched for these three neighbors and the node pointers of the new node and neighboring nodes are reconciled. For \(N\) nodes, this is \(O(N \log N)\). This generates a grid which has nodes that are evenly spaced. Many implementations benefit from a post refinement generation phase in which additional nodes can be inserted interactively prior to the initial use of the grid.

The example problem, the 2-D well grid (Fig. 6) has \(5 \times 5\) (25) initial nodes. The initial grid together with other information (e.g. \(X-Y\) coordinates) is stored in an ASCII file after being generated. A square and a well quadrant domain were used in this paper. If a more complex domain is required, then the traversing algorithm may have to be modified.

7. 2-D element refinement and derefinement

When the threshold criteria are met, an element needs to be refined or derefined. The criteria will depend on the client using the grid and will be calculated for each element while traversing the grid. A method to traverse the grid and mark the elements that need to be refined or derefined was developed. If there is at least one element to be refined in the entire grid, the grid will be refined. If no element needs to be refined, then the grid will derefine those elements that meet the derefine requirements. The algorithm either refines or derefines the grid but not both in a time step. The client may impose a maximum level of refinement, or alternately, let the grid refine as many levels as the resources allow.

When a 2-D element is refined, it is divided into four smaller elements. This division is accomplished by inserting four middle nodes on each link and then inserting a center node for these four middle nodes. The properties of this middle node, such as coordinates, are taken as the average of its corner nodes. In some cases, the middle node may already exists from a previous refinement of a neighbor, then the program only needs to update its pointers.

To preserve the 1-irregular structure of the elements, the neighboring element may have to be refined before the element under consideration can be refined. Such elements are easily recognized by checking the four corner nodes of the element. If any corner node has a null directional pointer not caused by being a boundary node of the grid, the neighbor in that direction must be refined first. The grids shown in

![Fig. 8. Refinement of two neighbors before refining element A.](image)
Fig. 8 illustrate how this refinement is done for element A. Node 1 and node 3 of element A have only three connections (non-null direction pointers). The null directional pointer of node 0 is pointer 3 of the node, which is pointing to the element below as shown on Fig. 2. Therefore, the neighbor element below must be refined before element A can be refined. Depending on whether the south-west quadrant is refined or not, as illustrated by the dotted line segments, the root node of the neighbor element is 1 or 2 steps away from the node 0 of element A. This same procedure is used recursively to pre-refine this neighboring element. Similarly, the null pointer of node 3 in element A requires the neighboring element to the left to be refined first. Once the 1-irregular requirement is satisfied, the element is refined. Because the new nodes created are inserted into the grid permanently and may belong to several elements, an element must be assembled immediately before the refining. The result is shown on the right of Fig. 8.

When an element is refined, four children elements are formed. The first element has the same root node as its parent. In fact, if we assemble this element after it is refined, its four corner nodes are the root nodes for the four children elements. This parent–child relationship must be kept in order to properly reverse the refinement process, i.e. derefine the grid. The initial nodes must never be deleted from the linked-list.

An element is uniquely determined by its root node (node 0) because the element can be assembled from this node. The right-most and top nodes of the grid cannot be root nodes because no element can be assembled using them. Some middle nodes, such as node 4 and node 7 on Fig. 3, cannot be root nodes. To distinguish such nodes and to also record the parent–child relationship, an extra pointer called parent is stored in each node object. It has three values: 0 if the node cannot be a root node for an element; it will point to the node itself if it is not created by element refinement; otherwise, it will point to the root node of the element that created these nodes during refinement. Note that there is no need to have four pointers to the children because the four corner nodes of the parent element are the root nodes of the children. This saves more storage space than the comparable quadtree–octree.

Derefinement is almost the reverse of refinement except the criteria are more conservative. When an element is marked for derefinement, the number of nodes in the element is checked. If the element has any middle nodes, it cannot be derefined because the 1-irregular requirement would be violated. No attempt to recursively derefine the neighbors was made because it could force derefinement of the elements that should not be derefined. A forced derefinement results in oscillation. In addition, its three siblings have to be checked. An element can be refined only when all its siblings are eligible for derefinement.

Derefining an element does not always remove all the nodes of that element because some nodes are also shared by neighboring elements. When this is the case, only the node links are updated. If the node cannot be a root node any more, the parent pointer is set to 0. After the nodes are removed, the traversal stack is checked to remove any node pointers that no longer exist or that cannot be parent nodes.

8. 3-D node and element

The same procedure (a linked list) used for 2-D was also used to manage the 3-D grid. The node pointer stack was used for the 3-D grid without modification. The node object needed an adaptation for the 3-D structure, instead of four links, the 3-D node has six directional pointers to link nodes. With two pointers in each dimension, the traversal of nodes is more flexible. The numbering scheme of the 3-D Node object is shown in Fig. 9. Comparing this with 2-D, the modification of the node object is minor; however, the software that handles the node objects becomes much more complex.

While a 2-D element may have a maximum of eight nodes, a 3-D element may have a minimum of eight to a
maximum of 26 nodes depending upon the refinement status of its neighbors. The numbering scheme of all possible nodes is shown in Fig. 10. The sequence to assemble a 3-D element is as follows: using the existing code for the 2-D assembling function, the nodes on the smallest loop are allocated. If node 8 does exist, the loop would consist of only nodes 0, 4, 8 and 7. Otherwise, the loop will correctly encompass nodes 0, 1, 2, 3 and possibly nodes 4, 5, 6 and 7. Thus, when node 8 does exist, the output of the 2-D assemble function had to be adjusted to include all the nodes on the top face. With the knowledge of the structure of the nodes on the top face, it is possible to allocate node 9. Subsequently, the nodes on the bottom face are allocated and followed by the nodes on the middle plane. As in 2-D elements, an array of node pointers are used to store the addresses of the nodes but the size of the array is now increased to 26. When a node does not exist, the pointer is set to 0; node 0 of the element is the root node of the element. The parent–child relationship is stored by the same pointer named ‘parent’.

9. 3-D grid traversal

The initial 3-D grid has an analogous structure to the 2-D grid. The initial grid of the 1-irregular 3-D grid in Fig. 11 has five nodes on both the X and Y axes and three nodes on the Z axis. Using the pointer links to represent the axes, many different domains can be mapped with this grid structure. One useful example is a quadrant of a 3-D grid of a well block shown on Fig. 11. Its initial grid, $5 \times 5 \times 4$ (100) nodes, are linked with the respective directional pointers.

Although similar, the 3-D grid traversal procedure is more difficult than the 2-D equivalent. The head of the grid is chosen as the south-west node on the top face because of convention [9], the positive z-direction is downward. The order of traversal is X-Y-Z when possible, and an example of the resulting global numbering is shown in Fig. 12. The stack content is shown in Fig. 13. The critical task is to push the correct node number out to the stack.

This stack content is done by starting from the head of the grid, the element at root node 0 as it is assembled. Because node 4 is the south-west corner of the lower Z plane, it is pushed onto the stack. Then, node 3 is pushed onto the stack because it is the first node on the next row of the top face. When these nodes are on the stack, the element at node 1 is visited. After the first row is traversed, the next element to be visited is popped from the stack. Once an element is assembled, the structure of the nodes in this element can be used to determine whether any of its nodes should be pushed onto the stack and no prior knowledge of the distribution of its neighbors is needed.
10. 3-D element refinement and derefinement

All the elements in a 3-D grid are also checked for refinement or derefinement according to the user's criteria. If any of the elements need to be refined, the grid is refined. Similarly, all the neighbors of an element have to be checked to ensure the 1-irregular requirement. A 3-D element has a maximum 24 face neighbors and a maximum 24 edge neighbors. Face neighbors share at least four common nodes on a face, denoted on Fig. 14 by letter ‘F’. Edge neighbors share at least two edge nodes, and they are denoted by ‘E’. The number of neighbors depends on the refinement status of the neighbors. When there has been no refinement, each element has six face and 12 edge neighbors. A physical model was constructed and used during this development to help visualize the complicated 3-D neighboring elements.

The 3-D algorithm works as follows: first, the face neighbors are checked for pre-refinement. The eight corner nodes of an element are examined, and if any of the corner nodes have a null directional pointer(s) and is not on the boundary, then the face neighbor in that direction needs to be refined first. After the face neighbors have been pre-refined, the edge neighbors will be allocated and examined. To allocate edge neighbors on the top face is equivalent to finding the horizontal face neighbors of the element above the current element. The same logic applies to the edge neighbors of the lower face. We also use this method to locate part of the edge neighbors. Only the vertical edge neighbors required a new method.

When an element requires refinement, the element is divided into eight smaller elements. This parent–child relationship needs to be recorded for use in derefinement. This is recorded as seven child elements that point to the parent element, which is denoted the 0 node. The refining element A in Fig. 14 is shown in Fig. 15.

If derefinement of a 3-D elements is compared with the similar 2-D problem, the main difference is more directional pointers need to be updated when a node is removed.
11. Comparison of memory usage in node based versus element based

The FORTRAN data structure [5] is a good basis for comparing the memory requirements of element based versus node based approaches. Although FORTRAN does not support structures and pointers directly, this scheme [5] can easily be analyzed in terms of any programming language.

12. Element based

Each element must store pointers for each of its nodes and possible neighbors. Furthermore, a group array [5] has been suggested to record the nodes of refined elements.

In 2-D, there are four nodes at the vertices and four more nodes possible at interior points where a neighbor can be refined. There are eight possible neighbors, and the group array requires four pointers, one to each of the sub-elements that are created during refinement. In total, there are 16 pointers per element in total.

In 3-D, there are six faces, 12 edges, and eight vertices per element. There is a node at each of the eight vertices, a possible midpoint node at each of the 12 edges, and a possible center node at each of the six faces for a total of 26 nodes. There are also face neighbors and edge neighbors. Each face can have four neighbors and each edge can have two neighbors, for a total of \((4 \times 6 \text{ faces}) + (2 \times 12 \text{ edges}) = 48 \text{ neighbors}\). The group array needs an additional eight pointers per element, since each element is broken into eight sub-elements during refinement. There are a total of 82 pointers per 3-D element.

13. Node based

In this node based approach, each node stores pointers to each of its possible neighboring nodes. There are four pointers per node in 2-D and six pointers per node in 3-D.

14. Node based savings

The node based and element based techniques store about the same number of basic data structures when a grid is large. For example, consider a cube where the number of nodes is \(X \times Y \times Z\) and the number of elements is \((X - 1)(Y - 1)(Z - 1)\). Consequently, the number of nodes and number of elements are not significantly different when the product of \(X, Y\) and \(Z\) is large. Let \(N\) be the number of data structures in either approach. The node based approach requires \(4 \times N\) pointers in 2-D and \(6 \times N\) in 3-D. The element based approach requires \(20 \times N\) pointers in 2-D and \(82 \times N\) in 3-D. A pointer usually takes four bytes of memory, so the node based approach saves \(16 \times N \times 4\) bytes in 2-D and \(76 \times N \times 4\) in 3-D. The savings in 3-D of \(304 \times N\) could be significant. Assuming half a million nodes in a large system, [8] the node based approach would save more than 150 megabytes of memory. This savings of memory is from the use of neighboring node pointers versus element pointers and all other storage requirements (coordinates, governing equation variables, etc.) are considered to be the same with either method. There is additional overhead CPU time required for the node based approach when it is necessary to assemble the elements, which is not needed for the element based approach. An accurate estimate of this additional CPU time for 2-D is \(12 \times N\) additional integer instructions and for 3-D is at least \(65 \times N\), but not more than \(80 \times N\) additional integer instructions. The maximum
additional CPU time required for $N = 500\,000$ (half a million nodes) and $80 \times N$, the 3-D maximum, is the time required to execute 40 million integer instructions or less than 1 s on a minimal workstation.

15. Interface to the client program

The routines that are used for manipulating the nodes of the grid, assembling elements, and refining–unrefining the elements reside in separate files in a grid software library, which is compiled independently of the client (a program that uses the grid library). The grid library is designed to interface with a client written in any general purpose programming language. The current client program is written in FORTRAN and implements the shape functions for quadrilateral or hexahedral elements [9, 10]. Since the grid library was implemented in C++, it could conceivably be used without any modification by a variety of clients.

A client uses several grid interface functions to establish the initial grid, access elements, change node values, refine the grid, and print the grid. The storage details of the nodes are isolated from the client, which only has access through interface functions (Table 1). The client initializes the grid with an input text file that specifies the beginning node layout and data values. Once the grid has been initialized with a call to InitGrid, the client can traverse the grid to create the stiffness matrix by repeated calls to GetElement. GetElement returns the data from the first element of the grid. Each successive call to GetElement retrieves the data to be used with the next element in the grid. After creating and solving a step of an FEM problem, the client stores new node values into the entire grid with a call to UpdateVar. Then the client may refine–derefine the grid by calling Adapt. Adapt calls a function ElmEval, which must be supplied by the client. ElmEval is used to evaluate the FEM answers to determine if the element should be refined–derefined. Finally, at any time, a client can create an output file by calling SaveResults.

16. Conclusion

We have implemented a node based approach for dynamic adaptive 2-D and 3-D grids. This data structure efficiently stores the necessary data in node objects, where the node objects are linked to each of their neighboring nodes. While traversing the grid, finite elements can be dynamically assembled, refined, and derefined. The connectivity between elements is stored, which saves computer memory. Bandwidth is also reduced because nodes are dynamically numbered. In the future, more complex topologies could be combined with the node based approach. For example, an octree structure could be implemented where each octant points to a root node of an element instead of an entire element data structure. This would provide the same memory savings and neighbor accessibility as the node based approach and allow the benefits of octree topology. In fact, the node based approach could be used as the underlying framework for many geometrical configurations, including configurations suitable for multi-processing.

Acknowledgements

The support of this work under contract number DE-F622-95BC14802 from DOE is gratefully acknowledged.

References