EXTRACTING HEXAHEDRAL MESH STRUCTURES FROM BALANCED LINEAR OCTREES

Tiankai Tu[1] David R. O'Hallaron[1,2]

[1]*Computer Science Department*
[2]*Electrical and Computer Engineering Department*
{tutk,droh}@cs.cmu.edu
Carnegie Mellon University, Pittsburgh, USA

ABSTRACT

Generating large 3D unstructured meshes with over 1 billion elements has been a challenging task. Fortunately, for a large class of applications with relatively simple geometries, unstructured octree-based hexahedral meshes provide a good compromise between adaptivity and simplicity. This paper presents recent work on how to extract hexahedral mesh structures from a class of database structures known as balanced linear octrees. The proposed technique is not memory bound and is capable of extracting mesh structures with billions of elements and nodes, provided there is enough disk space to store the mesh. In practice, our new algorithm runs about 11 times faster than a conventional database search-based algorithm and uses only 10% of the storage space.

Keywords: balanced linear octree, unstructured hexahedral mesh, mesh database, two-level bucket sort, dangling and anchored nodes

1. INTRODUCTION

Rapidly evolving high-performance computer systems have enabled scientists to simulate nature at ever increasing levels of detail. For example, for the past decade, the CMU Quake group has been modeling earthquake ground motions in large sedimentary basins on parallel computers [1, 2, 3, 4]. Over the years, the earthquake simulation codes have run on the Thinking machines CM-2, Intel iWarp and Paragon, SGI Origin, Cray T3D and T3E, and most recently the HP AlphaServer cluster. Each generation of architecture has prompted and enabled a larger and finer model. In 1993, the largest simulation code used an unstructured finite element mesh with only 50K nodes (1.5MB). By 2003, the largest simulation required a mesh with 1.37B nodes (45GB).

The dramatic increase in scales of such physical simulations has made many routine tasks — such as generating unstructured meshes, defining source models, and visualizing mesh structures — hard to accomplish on scientists' desktop machines. This is because unstructured meshes often require complex pointer-based structures to represent, thus require massive main memory to manipulate. Desktop computers with limited main memory are thus unable to accommodate such massive meshes.

We envision that a database approach can effectively solve such massive data problems. Our basic idea is to organize meshes as indexed spatial database structures and use a specialized set of tightly-coupled and highly-optimized functions to manipulate the mesh databases. It should be emphasized that our idea is to generate meshes directly from databases, rather than merely storing *generated* meshes in standard relational database management systems. Thus, as long as there is enough disk space, scientists will be able to generate massive unstructured meshes and interactively explore mesh structures by querying mesh databases stored on their desktops.

For a large class of applications with relatively simple geometries, octree-based hexahedral meshes provide a good compromise between adaptivity and simplicity. In our earlier work [5], we introduced a new method on how to generate large octree-based hexahedral meshes from databases. Since then, we have made major extensions and improvements to the original design. Different (existing and new) software components have now been incorporated into a prototype system named *Weaver* [6].

Figure 1 shows the structure of the Weaver system, which consists of two parts: *spatial database* and *mesh generation logic*. The spatial database manages unstructured hexahedral mesh data such as elements and nodes on disk and in memory. High-level applications can efficiently query and manipulate the spatial databases through a runtime library called *etree* [7]. The mesh generation logic implements different mesh generation steps by exploiting the characteristics of the underlying database structures. In particular, the *construct* step builds an indexed linear octree on disk [5]. The sizes of the octants are determined by an application, for example, by the density of the material they enclose. The *balance* step recursively subdivides octants as necessary to ensure that spatially adjacent octants (sharing an edge or face) differ no more than 2-fold in their (edge) sizes [8]. The *extract* step uses a balanced linear octree as a template to generate mesh elements and nodes and store the mesh structure in a queryable mesh database. Finally, the *transform* step queries the data in the mesh database and generate a *flat mesh topology file* that establishes the element-node connectivity relationship. Such files are needed by existing mesh partitioning tools [9] and solver packages [10].

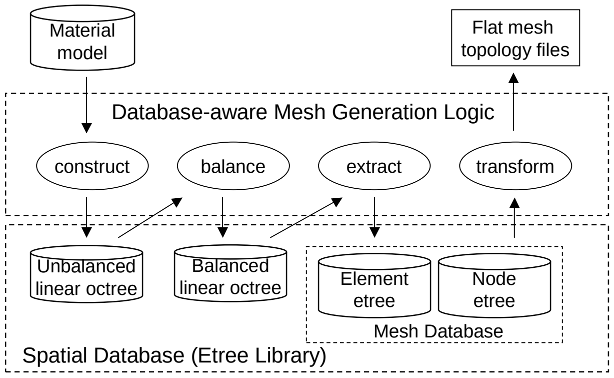


Figure 1: **Structure of the Weaver system.**

The Weaver system has been successfully used to generate unstructured octree-based hexahedral meshes with billions of elements on the same desktop machine where previously only meshes with tens of millions of elements could be generated [5]. For generating billion-element unstructured hexahedral meshes and simulating earthquake ground motion in the Los Angeles Basin using these meshes on terascale computers at Pittsburgh Supercomputing Center, and for work on inversion, the authors, along with our colleagues of the CMU Quake team, received the 2003 Gordon Bell Award for Special Achievement [4].

The breakthrough that has enabled us to generate much larger meshes is a new method of extracting mesh structures from balanced linear octrees, i.e., the extract step shown in Figure 1. This step used to require massive disk space and take a long time to run, severely limiting the size of meshes we were able to generate. The main new algorithm we have developed is called *Two-level Bucket Sort*. This new algorithm not only significantly reduces the storage requirement, but also greatly improves the running time of the extract operation. The key ideas are: (1) treating mesh nodes as the tiniest octants in the domain instead of volumeless geometric points; and (2) converting the mesh node extraction problem to a sorting problem.

It should be noted that the techniques presented in this paper have other important applications besides merely generating non-conforming unstructured octree meshes in the context of the CMU Quake project. For example, an octree decomposition can be first used to discretize a problem domain, and the vertex coordinates of the octants (i.e., the product of the *extract* step) can then be used as the input point-set for generating a conforming mesh structure such as a Delaunay tetrahedralization.

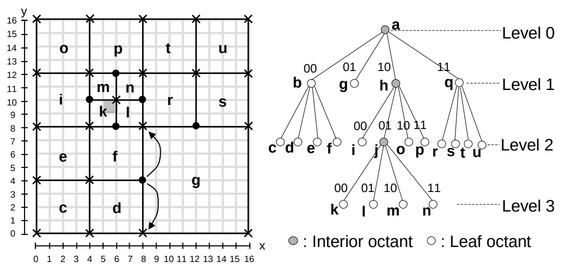
The rest of the paper is organized as follows. Section 2 provides a brief overview of octree-based hexahedral mesh generation. Section 3 describes the problems we try to solve. Section 4 presents the main new algorithm. Section 5 explains how to determine whether a mesh node is *dangling* or *anchored*, a property specific to unstructured octree (hexahedral) meshes. Section 6 evaluates our new solution empirically.

2. BACKGROUND

Among the many types of meshes, octree-based hexahedral meshes fall between the extremes of arbitrarily unstructured meshes and regular structured meshes [11]. They provide a compromise between modeling power and simplicity. On the one hand, they are able to subdivide an octant to resolve local heterogeneity and provide multi-scale resolution as do other unstructured meshes. On the other hand, they produce only one primitive shape for all elements. The recursive process of subdivision leads to a relatively structured placement of mesh nodes, similar to many regular structured meshes. Due to these features, octree-based hexahedral meshes have been used successfully by many scientific computing applications [12, 13, 14, 4].

We have developed a prototype system named Weaver (see Figure 1) to generates octree-based hexahedral meshes. As a first step, we embed a problem domain into a 3D uniform grid consisting of $2^{31} \times 2^{31} \times 2^{31}$ indivisible *pixels*. We refer to this 3D grid as the *etree address space* [7]. Figure 2(a) illustrates an example 2D $2^4 \times 2^4$ grid and the embedding of an octree decomposition. (For convenience, we only use the terms *octree* and *octant* in this paper, even when we are referrring to 2D quadtrees and quadrants.) Figure 2(b) shows the equivalent tree representation.

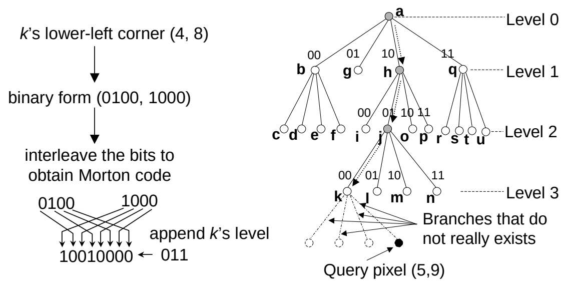
To represent octants in the etree address space, we use the well-known *linear octree* technique [15, 16]. The basic idea



(a) Octree domain decomposition.

(b) Tree representation.

Figure 2: **Octree decomposition as a template to generate meshes.**



(a) Computing the locational code for k.

(b) Aggregate hit of a pixel inside k.

Figure 3: **Operations on octrees.**

of the linear octree is to encode each octant with a scalar key called a *locational code* that uniquely identifies the octant. A locational code can be easily derived by first interleaving the bits of the coordinate of the lower-left corner of an octant, and then appending the *level* of the octant [7]. The level of an octant refers to its level in the equivalent tree representation, Figure 3 illustrates how to compute the locational code for octant k.

Given unique locational codes for each octants, we use the well-known B-tree [17, 18, 19] to index and store octants. As a result, octant records are laid out on disk (B-tree pages) in locational code order (hence the term linear octree). It is not difficult to verify that the ordering imposed by the locational codes corresponds to a *preorder traversal* of the leaf octants of the tree representation. There are two interesting properties related to the preorder traversal property: (1) It clusters spatially nearby octants on B-tree pages in the locality-preserving Z-order [20]; (2) It supports an important feature called *aggregate hits* [7]. The idea of an aggregate hit is that given the locational code of a pixel, we can extract its bit-patterns and quickly locate the octant that contains the pixel. Figure 3(b) shows an example of finding octant k while searching for pixel $(5, 9)$, the grayed pixel in Figure 2(a). If we think of pixels as the tiniest representable octants in the domain, a simple explanation of aggregate hits is that a pixel would have been a descendant of the containing octant if the latter had been fully expanded to the deepest level.

3. PROBLEM STATEMENT

In this paper, we assume that we already have a *balanced* linear octree that is indexed and stored in a B-tree (for example, using the *balance* operation shown in Figure 1). The term "balanced" refers to the fact that any spatially adjacent octants sharing an edge or a face differ no more than 2-fold in their edge sizes, a condition to guarantee the quality of a mesh.

Our goal is to extract the hexahedral mesh structure from a balanced linear octree. Conceptually, mesh elements correspond 1-to-1 to leaf octants, and have unique ids, i.e. *element numbers*, drawn from a consecutive integer sequence. Mesh nodes correspond 1-to-1 to vertices of the elements, and have unique ids, i.e. *node numbers*, drawn from a separate consecutive integer sequence.

Finite element solvers that operate over octree-based hexahedral meshes [13, 10] distinguish between two different types of mesh nodes. Mesh nodes that are located at the middle of an edge or at the center of a face are said to be *dangling* nodes. All other mesh nodes are *anchored* nodes. For example, in Figure 2(a), the dangling nodes are marked by the dark dots while the anchored nodes are marked by the cross signs. (Note that in 2D there only exist dangling nodes that are located at the middle of some edges.)

As a part of the process of extracting mesh structure, the mesh generator must identify each node as either dangling or anchored. Furthermore, it must explicitly identify the anchored nodes each dangling node depends on. A dangling node located on an edge is said to be *dependent* on the two anchored nodes of the edge. Similarly, a dangling node located at the center of some face is dependent on the four anchored nodes of the face. The arrows in Figure 2(a) illustrate an instance of the dangling/anchored dependences.

Extracting mesh elements is simple. Since elements are simply the octants stored and indexed in the balanced linear octree, we can iterate octants one by one in their locational code (key) order, and assign element numbers to octants in ascending order as we encounter them. The difficulty lies in two other operations: (1) extracting mesh nodes, and (2) identifying whether mesh nodes are dangling or anchored. Because a balanced linear octree does not contain any explicit information about mesh nodes, we must extract mesh nodes by computing the coordinates of vertices as we encounter new elements. Since most mesh nodes are shared by multiple elements, we need to make sure that a newly computed coordinate indeed represents a new mesh node rather than a duplicate one. In other words, we must get rid of duplicates while extracting mesh nodes. Furthermore, we cannot determine whether a mesh node is dangling or anchored by examining its coordinate alone. Instead, we need to check the sizes and locations of the elements surrounding a mesh node.

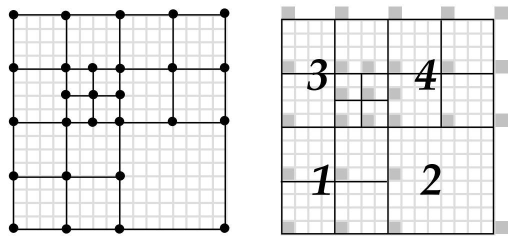
A straightforward solution to implement the two difficult operations works by searching database B-tree index structures. First, we index the coordinates of mesh nodes in a *node B-tree*. On encountering each new element *elem*, we compute the coordinates of its eight vertices. For each coordinate, we search the node B-tree to check whether a node with the same coordinate exists or not. If so, the node must have been generated due to some other element we have processed earlier. In this case, we append the locational code of *elem* to the corresponding node record. If the coordinate does not match that of any existing nodes, we create a new node record with the coordinate as its key and the locational code of *elem* as its payload. After all mesh elements are processed, the node B-tree must have stored and indexed all the mesh nodes. Then we iterate through all the mesh nodes, assigning node numbers in ascending order as we encounter them (similar to the element number assignment process). For each node, we determine whether it is dangling or not by analyzing the locational codes of those elements recorded in its payload. A separate collection of *dangling node records* are produced in this process. Each dangling node record contains the coordinate of a dangling node, the number of dangling nodes it depends on (either 2 or 4), and the coordinates of those anchored nodes.

This database search-based algorithm works fine for relative small datasets with tens of millions of elements and nodes. However, the algorithm's time and space complexity severely limit its scalability. First, the running time of the algorithm is O($n \log n$), where n is the number of mesh nodes. The dominant cost is due to the search operations on the node B-tree. Since each computed coordinate incurs a node B-tree search operation of cost O($\log n$), the total cost due to searches is O($n \log n$). Second, the storage requirement of the algorithm is roughly $e + 9n$. We emphasize the constant factor 9 to show that besides the space to hold the coordinates of mesh node, we also need to allocate extra payload space for each mesh node to hold up to 8 (element) locational codes. Since locational codes are obtained by interleaving the bits of coordinates (and append the level information that is encoded in an extra byte), we can conveniently regard the size of a locational code as large as that of a coordinate.

So the problem we try to solve is more of a performance issue than a functional one. More specifically, *how can we extract mesh nodes and determine their dangling/anchored properties more efficiently both time-wise and space-wise?*

4. TWO-LEVEL BUCKET SORT

This section presents the main algorithm we developed to solve the problem. We refer to the algorithm as *Two-level Bucket Sort*. The basic idea is to convert the search-based mesh node extraction process to a more efficient sorting procedure. Section 4.1 and 4.2 present the rationale of the design. Section 4.3 describes the algorithm itself.



(a) Mesh nodes as volumeless points. (b) Mesh nodes as pixels.

Figure 4: **Two different ways of representing hexahedral mesh nodes.**

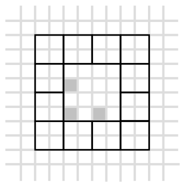
4.1 An Alternative Way of Representing Mesh Nodes

Mesh nodes are typically defined as the vertices of mesh elements. That is, mesh node are volumeless geometric points, shown as dark dots in Figure 4(a). (Ignore the difference between dangling nodes and anchored nodes for the time being.) However, the relatively regular placement of mesh nodes of octree-based hexahedral meshes gives us an opportunity to represent mesh nodes in a different way. It can be seen that in the etree address space where a problem domain is embedded, the recursive subdivision process of octants ensures that the coordinates of vertices always have integer values. As a result, a mesh node must be located at the lower-left corner of some pixel. Since different nodes have different coordinates, each mesh node can be uniquely represented by the pixel whose lower-left corner has the same coordinate as that of the node, as shown in Figure 4(b) by the grayed pixels. Note that the nodes on the far-side boundaries are represented by pixels outside of the original address space. We will discuss how to handle these cases in Section 4.3.

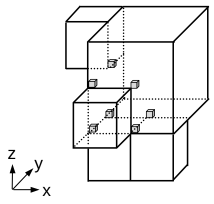
This alternative way of representing mesh nodes allows us to approach the problem of extracting mesh nodes from a new perspective: treat mesh nodes as pixels spatially distributed in the etree address space. For simplicity, we refer to those pixels as *nodal pixels*. Extracting mesh nodes and indexing them in a node B-tree is thus equivalent to sorting nodal pixels according to their locational codes and bulk-load them to a B-tree.

4.2 Buckets at Two Levels

It is well known that comparison based sorting algorithms run in O($n \log n$) time [21]. Linear time (O(n)) algorithms such as count sort, radix sort and bucket sort run faster by exploiting the value information of the records being sorted. Therefore, to develop an efficient algorithm to sort nodal pixels, we need to somehow make use the semantics of the locational codes.

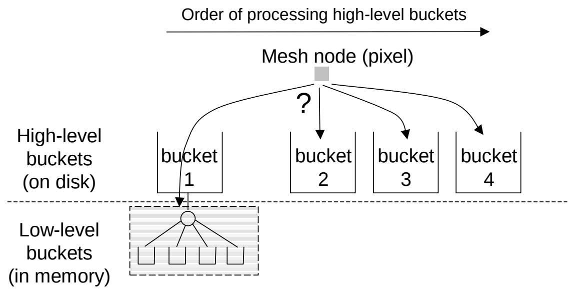


(a) At most 3 nodal pixels fall in a 2D bucket.



(b) At most 7 nodal pixels fall in a 3D bucket.

Figure 5: **Treat octants as buckets to hold nodal pixels.**



Order of processing high-level buckets
Mesh node (pixel)
?
High-level buckets (on disk)
bucket 1
bucket 2
bucket 3
bucket 4
Low-level buckets (in memory)

Figure 6: **A nodal pixel falls in either a low-level bucket or a high-level bucket.**

As explained in Section 2, locational codes have an important property called aggregate hit. That is, given the locational code of a pixel, we can extract its bit-patterns and traverse down an octree to hit the octant that contains the pixel. Therefore, we can treat octants (mesh elements) as buckets. All nodal pixels except for those on the far-side boundaries are distributed among the buckets. Note that different from traditional buckets that represent equal-sized scalar value intervals [21], the buckets we defined have spatial spans in the etree address space. Given the way we define nodal pixels, it can be verified that each bucket may contain at most 3 pixels in 2D, and 7 pixels in 3D, as shown in Figure 5.

Since a balanced linear octree may contain billions of octants, what shall we do if the main memory is not large enough to accommodate all the buckets (octants)? As the name of our algorithm suggests, we solve this problem by using buckets of different resolutions at two levels. The fine-grained low-level buckets correspond to the leaf octants (mesh elements) in an "active" region that we are processing. These buckets are organized in a pointer-based octree structure in memory to support aggregate hits and mesh node sorting. The coarse-grained high-level buckets correspond to (virtual) subtree roots that cover "inactive" regions in the etree address space. Mesh nodes fall in those high-level buckets are flushed to disk and processed later. We will explain what "active" and "inactive" mean and how to create high-level buckets in Section 4.3.

For example, the domain of Figure 4(b) consists of 4 high-level buckets corresponding to the four quadrants of the domain. Since the high-level buckets constitute a partition of the domain, each nodal pixel must belong to some high-level bucket. (For simplicity, ignore those mesh nodes on the far-side boundary of the domain.) Figure 6 shows the buckets at the two different levels while we are extracting elements and nodes from high-level bucket 1 (that is, the active region). When we obtain the coordinate of a vertex, i.e. nodal pixel, we can derive it locational code. This nodal pixel will either fall in some inactive high-level bucket or fall in the currently active high-level bucket. In the latter case, the pixel will fall in some low-level bucket (octant) due to the aggregate hit property. Either way, a nodal pixel will be assigned to a proper bucket.

4.3 Algorithm Outline

Figure 7 presents an overview of the two-level bucket sort algorithm. Many (gory) details have been omitted. Our purpose is to show the structure of the algorithm and highlight the important steps involved.

The input, as mentioned earlier, is a balanced linear octree indexed by a B-tree. The output is an indexed mesh structure. Because we use nodal pixels to represent mesh nodes, we can treat mesh nodes as the tiniest octants in the domain and index them in the same way as we index mesh elements. Since we use the etree library [7] to manipulate B-tree indexed linear octrees, we simply refer to the indexed mesh structure as element etree and node etree.

```
Input:
      A balanced linear octree indexed in a B-tree.

Output:
      An element etree and a node etree.

Method:
      Compute nodal pixels while creating mesh elements
      and assign nodal pixels to proper buckets for sorting.

  S1:   Expand the domain on the far-side boundaries.
  S2:   Decide the size of high-level buckets.
  S3:   Partition the expanded domain into equal-sized
        high-level buckets.
  S4:   Initialize to the first high-level bucket.
  S5:   Extract elements and nodes belonging to the cur-
        rent high-level bucket.
  S6:   Move to the next high-level bucket in Z-order.
  S7:   Goto S5 if not NULL; otherwise, terminate.
```

Figure 7: Two-level Bucket Sort.

Step S1 conceptually expands the far-side boundaries of the domain by adding a layer of the largest leaf octants. The purpose is to provide a set of low-level buckets to accommodate the nodal pixels on the far-side boundaries as shown in Figure 4(b). Note that this set of extra octants does not physically exist in the input balanced linear octree.

Step S2 conservatively assumes that the problem domain is filled with the smallest octants only, and compute the largest subtree that can be cached in memory. The size of the high-level buckets is set to be that of the correspond subtree root. All high-level buckets are of the same size and are properly aligned as subtree roots.

Step S3 uses an octree decomposition to partition the expanded domain into high-level buckets. An easy way to understand this partition is to imagine a cut-off of a fully-grown octree at the subtree root level calculated in S2. Each high-level bucket covers a region corresponding to a subtree.

Steps S4 – S7 process high-level buckets in Z-order. A high-level bucket currently being processed represents an *active* region where new mesh elements and nodes are originated. Other regions are said to be *inactive*. The most important step is S5, which extracts elements and nodes from an active region. We outline the operations involved in Figure 8.

S5-1:	Initialize a pointer-based octree to represent low-level buckets.
S5-2:	Create nodal pixel records.
S5-3:	Assign nodal pixels to buckets.
S5-4:	Set to the leftmost leaf octant in the pointer-based octree.
S5-5:	Create an element record and append to the element etree.
S5-6:	Sort nodal pixels (7 at maximum) assigned to the current leaf octant.
S5-7:	Set dangling/anchored properties for the nodal pixels sorted.
S5-8:	Create node records and append to the node etree.
S5-9:	Traverse to the next leaf octant in preorder.
S5-10:	Goto S5-5 if not NULL, otherwise, terminate.

Figure 8: How to extract mesh elements and nodes belonging to a high-level bucket (S5).

An important data structure used by S5 is a hash table. We create an empty hash table before each invocation of S5, and then use the hash table to keep track of nodal pixels encountered in the “active” region. Since each vertex is shared by multiple octants (elements), the coordinate of each nodal pixel will be computed multiple times (S5-2). We install a nodal pixel record in the hash table the first time we encounter a coordinate. The record contains the following fields: (1) the coordinate of the nodal pixel (i.e. the hash entry tag), (2) the tree level of the octant (element) that produces the nodal pixel, and (3) a reference count that is initialized to 1. When we encounter the same coordinate again while processing other octants, we simply increment the reference count associated with the nodal pixel by 1. The information collected is later used to derive the dangling/anchored properties of nodal pixels (S5-7) (see Section 5 for details). Besides quickly identifying duplicates, a second main function of the hash table is to speed up the assignment of nodal pixels to low-level buckets. Generally, a nodal pixel finds its accommodating low-level bucket by traversing down the pointer-based octree built in S5-1. Noticing that each low-level bucket certainly encloses the nodal pixel located at its lower-left corner, we let low-level buckets to directly adopt their corner nodal pixels by searching the hash table.

Because two-level bucket sort outputs both mesh elements and nodes in Z-order, which is the same as the locational code ordering, we can safely use append operations (O(1) cost) to add elements and nodes to the database. It can be shown that the running time of two-level bucket sort is O($n + (n - e)\log(n/b) + b\log(b)$), where n is the number of mesh nodes, e is the number of elements, and b is the number of high-level buckets. Besides, since we can resolve dangling/anchored property within the sorting procedure, there is no need to store element locational codes with mesh node records. The storage requirement of the algorithm is O($e + n$).

5. IDENTIFYING DANGLING NODES

This section explains how we use the information collected in nodal pixel records to determine whether a mesh node is dangling or anchored (S5-7). The basic idea is to exploit the fact that hexahedral meshes only yield nodes at the vertices of the octants, and use modular arithmetic to identify dangling nodes. For ease of illustration, we only discuss 2D cases. Also, for clarify, we use geometric points instead of nodal pixels in our figures to represent mesh nodes.

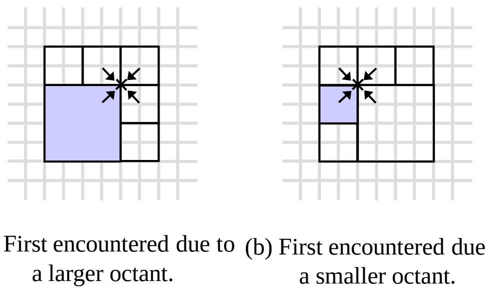
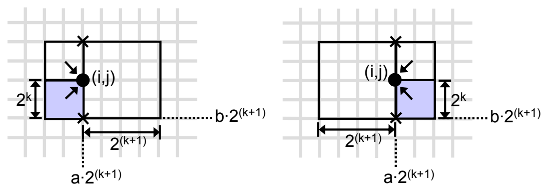

(a) First encountered due to a larger octant.
(b) First encountered due to a smaller octant.

Figure 9: **A reference count of 4 indicates an anchored node.**

Figure 9 illustrates the octants surrounding an anchored node. These octants are part of a large mesh whose other octants are not shown in the figure. The grayed octant is the one that produces the anchored node (the cross sign) in the first place. The arrows mark the octants that have the node as a vertex and contribute to its reference count. It can be seen that regardless which octant produces an anchored node first, the reference count of the nodal pixel record is always 4. Therefore, we can conveniently determine that a node is anchored if its reference count is equal to 4. The other two fields of nodal pixel records are not used.

Besides a reference count of 4, the other possible values are

2 and 1 (in 2D). Note that only the four nodes at the domain boundary corners have a reference count of 1. However, both edge boundary nodes and dangling nodes may have a reference count of 2. Fortunately, because all (domain) boundary nodes are anchored (in 2D), we can label their properties easily by checking their coordinates. The remaining problem is how to identify dangling nodes.



(a) On the left vertical edge of a larger octant. (b) On the right vertical edge of a larger octant.

Figure 10: **The reference count of a dangling node is always 2.**

Figure 10 shows the case of a dangling node (the dark dot) that is located at the middle of a vertical edge. The cross signs mark the anchored nodes the dangling node depends on. Note that if the octant that produces the anchored node in the first place (the grayed octant) is at octree level k, then the other octant that has the dangling node as one of its vertices must also be at at tree level k. Otherwise, the input linear octree cannot be balanced.

Suppose the coordinate of the dangling node in our example is (i, j), it is not difficult to derive that $i = a * 2^{k+1}$ and $j = b * 2^{k+1} + 2^k$, where $a, b \in Z^+ \cup 0$. Therefore, we can use (i, j) and k (the other two fields in the nodal pixel record) to deduce that the dangling node is either at the middle of a vertical edge if $i = 0 \bmod 2^{k+1}$ and $j \neq 0 \bmod 2^{k+1}$ (the case shown in our example); or at the middle of a horizontal edge if $i \neq 0 \bmod 2^{k+1}$ and $j = 0 \bmod 2^{k+1}$. In either case, the coordinate of the anchored nodes (cross signs) can be easily derived.

We note that in 3D, the possible reference counts of a mesh node is 8, 6, 4, 2 and 1, and not all boundary nodes are anchored. The analysis is more complicated (and tedious). Nevertheless, the basic principle of modular arithmetic is still applicable.

6. EVALUATION

This section evaluates the effectiveness of our techniques empirically. We have conducted experiments to answer the following two questions: (1) How much faster can we extract hexahedral mesh structures from balanced linear octrees using two-level bucket sort than using a search-based algorithm? (2) How much storage space can we save by using modular arithmetic to identify dangling/anchored nodes?

6.1 Methodology

The meshes we generated are used for earthquake ground motion simulations. The purpose of such simulations is not to predict *when* an earthquake would occur, but rather what would happen *if* that earthquake would occur. In heterogeneous geological structures such as sedimentary basins where material properties vary significantly through the domain, multi-resolution unstructured hexahedral meshes allow a tremendous reduction (approx. three orders of magnitude) in the number of mesh nodes (as compared to uniform grids), because element sizes can adapt locally to the high-variable wavelength of propagating seismic waves.

The target region of our experiments is the Los Angeles Basin (LAB), which comprises a 3D volume of 100km x 100km x 37.5 km. The material model we used to drive the mesh generation process is the Southern California Earthquake Center (SCEC) 3D velocity model [22] (Version 3, 2002). Meshes of different resolutions are generated for simulations with different frequency requirements. Roughly speaking, the higher the frequency, the finer the mesh. Figure 11(a) summarizes the characteristics of the mesh structures of our experiments. Each mesh is tagged with the frequency it can resolve. “Elements” records the number of elements, “Nodes” records the number of nodes, which includes both the dangling and the anchored, and “Danglings” records the number of dangling nodes, which is a subset of “Nodes”.

Our experiments were conducted on a desktop machine with a Pentium III 1GHz processor running Linux 2.4.17. The memory subsystem consisted of 3GB physical memory and 1GB swap space. Each experiment took a balanced linear octree (produced by the *balance* step of Figure 1) as input and produce a mesh database (stored and indexed in etrees) as output.

Mesh	0.2Hz	0.5 Hz	1Hz	2Hz
Elements	433k	9.92m	113m	1.22b
Nodes	556k	11.3m	134m	1.37b
Danglings	125k	1.43m	20m	147m

Figure 11: **Summary of LAB meshes.**

One caveat is that for the purpose of our experiments, we only create node etrees, which is what the two-level bucket sort algorithm does. The element etrees are exactly the same as the input linear octree since no other information (physical quantities) needs to be associated with element records in our experiments.

6.2 How much faster can we extract mesh structures?

Figure 12 shows the running time of two different algorithms. “Search-Alg” represents the conventional database search-based algorithm described in Section 3. “Sort-Alg” represents the two-level bucket sort algorithm introduced in Section 4.2. "T" shows the execution times of a particular algorithm and "ρ" shows the corresponding throughputs in terms of nodes/second. "Speedup" is calculated by dividing the running time of Search-Alg by the running time of Sort-Alg. It quantifies how much faster we can extract mesh structures from balanced linear octrees using the two-level bucket sort algorithm. Note that the running time for extracting the structure of 2Hz mesh using the conventional search-based algorithm is unavailable (n/a) because it ran out of disk space (120GB) on the desktop machine where we conducted the experiment. In addition, the running time of the search-based algorithm does not include the time of post-processing nodes to determine their dangling/anchored properties. So the speedups reported are conservative.

We can see that two-level bucket sort runs much faster than a search-based algorithm, achieving an average speedup of 11. From Figure 11, we can see there are only about 15% more nodes (n) than elements (e). Thus, it is expected the contribution of the second term in the time complexity of two-level bucket sort (O($n + (n - e)\log(n/b) + b\log(b)$)) is significantly discounted. Besides, the number of high-level buckets b is actually very small. The contribution of the third term is negligible. In contrast, the search-based algorithm always run in O($n \log n$) (see section 3). The drastic improvement in empirical running times indicate that two-level bucket sort is a much more efficient algorithm in practice.

Mesh		0.2Hz	0.5 Hz	1Hz	2Hz
Nodes		555k	11.3m	134m	1.37b
Search-Alg	T	00:00:28	00:12:59	02:51:57	n/a
	ρ	19.8k	14.5k	13.0k	n/a
Sort-Alg	T	00:00:03	00:01:03	00:14:19	02:38:44
	ρ	185k	166k	152k	144k
Speedup		9.3	11.5	12.0	n/a

Figure 12: **The running time and throughput of two different algorithms.**

Another interesting conclusion we can draw from Figure 12 is that two-level bucket sort is quite scalable. The throughput of the algorithm is about 185k for extracting mesh structure for the 0.2Hz mesh (with 555k nodes), and 144k for the 2Hz mesh (with 1.37b nodes), dropping by only 22% while the latter is about 2,500 times larger than the former.

6.3 How much storage space can we save?

Figure 13 summarizes the disk space usage of node etrees produced by the two different algorithms. "Optimal" shows the sizes of flat files if we would store node records one after the other sequentially without any overhead. "Actual" shows the sizes of node etrees, each of which include node records, B-tree index structures and B-tree page internal fragmentations. "Overhead" represents the percentages of the "Actual" storage space used for B-tree index structures and B-tree page internal fragmentations. "Storage saved" indicates the percentages of actual disk space saved when using two-level bucket sort ("Sort-Alg") as compared with the case of using the conventional search-based algorithm ("Search-Alg").

Mesh		0.2Hz	0.5 Hz	1Hz	2Hz
Search-Alg	Optimal	68.7mb	1.29gb	16.5gb	169gb
	Actual	104mb	2.17gb	25.1gb	n/a
	Overhead	33.9%	40.6%	34.3%	n/a
Sort-Alg	Optimal	10.0mb	204mb	2.41gb	24.7gb
	Actual	10.1mb	206mb	2.44gb	25.0gb
	Overhead	1.00%	1.00%	1.23%	1.20%
Storage saved		85.3%	90.5%	90.3%	n/a

Figure 13: **Storage requirements of two different algorithms.**

Since two-level bucket sort does not store elements' locational codes with node records, it is not surprising that we can save about 90% of the disk space that would otherwise have been used. In other words, by using modular arithmetic to determine the dangling/anchored properties of mesh nodes in context of the sorting procedure, we can extract mesh structure using 1/10 of the disk space required by a search-based algorithm. The fact that the search-based algorithm ran out of disk space (n/a) while extracting mesh structure for the 2Hz mesh indicates that we should still conserve disk storage space even if we are designing algorithms to run directly on databases.

The difference between the "Optimal" size and the "Actual" size is the overhead associated with database structures. For the search-based algorithm, the overhead is approximately 36%. This is because the search-based algorithm loads the database via standard B-tree insertion operations. And it has been shown that the disk space utilization of B-trees is 69% [23] on average under random insertions. Therefore, the excessive overhead is due to the internal fragmentations of B-tree pages. In contrast, two-level bucket sort appends node records in ascending order to the database (B-tree). The disk space utilization is optimized, with only very small internal fragmentations per B-tree page. Thus, the extra $\tilde{1}$% disk space overhead is mainly used by B-tree index pages, which is a reasonable price for efficient (O($\log n$)) query capabilities.

7. SUMMARY

Generating meshes directly from databases will not only allow scientists to generate much larger meshes on their desktops, but also provide query capabilities at no extra cost. The challenge is how to design database-aware mesh generation algorithms that are both efficient and scalable.

This paper has presented our recent work on how to extract mesh structures from balanced linear octrees, the most resource-intensive step for generating unstructured hexahedral meshes from databases. By exploiting the characteristics of linear octrees, in particular, the clustering property and the aggregate hit property of the locational codes, we are able to approach the problem from a database perspective and develop an efficient new algorithm called *Two-level Bucket Sort*. In addition, we employ modular arithmetic in the context of the sorting procedure to identify dangling mesh nodes, thus save storage space that would otherwise be used to record the locational codes of surrounding elements.

ACKNOWLEDGEMENTS

We gratefully acknowledge Jacobo Bielak, Omar Ghattas and Eui Joong Kim for developing the octree-based finite element method and for using our massive hexahedral meshes of the Los Angeles Basin in their terascale ground motion simulations at Pittsburgh Supercomputing Center. We also thank Steve Day, Tom Jordan, Karl Kesselman, Phil Maechlin, Harold Magistrale, Kim Olsen, and our colleagues on the Southern California Earthquake Center (SCEC) Community Modeling Environment Project, for their support and encouragement. Finally, thanks to Anastassia Ailamaki and Christos Faloutsos for helping us understand spatial databases. This work is sponsored in part by the National Science Foundation under Grant CMS-9980063, in part by a subcontract from Southern California Earthquake Center as part of NSF ITR EAR-01-22464, and in part by a grant from the Intel Corporation.

References

[1] Bao H., Bielak J., Ghattas O., Kallivokas L., O'Hallaron D., Shewchuk J., Xu J. "Earthquake Ground Motion Modeling on Parallel Computers." *Proc. Supercomputing '96*. Pittsburgh, PA, Nov. 1996. URL `www.cs.cmu.edu/∼quake/`

[2] Bao H., Bielak J., Ghattas O., Kallivokas L., O'Hallaron D., Shewchuk J., Xu J. "Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers." *Computer Methods in Applied Mechanics and Engineering*, vol. 152, 85–102, Jan. 1998

[3] Hisada Y., Bao H., Bielak J., Ghattas O., O'Hallaron D. "Simulations of Long-Period Ground Motions during the 1995 Hyogoken-Nanbu (Kobe) Earthquake using 3D Finite Element Method." K. Irikura, H. Kawase, T. Iwata, editors, *2nd International Symposium on Effect of Surface Geology on Seismic Motion, Special Volume on Simultaneous Simulation for Kobe*, pp. 59–66. Yokohama, Japan, Dec. 1998

[4] Akcelik V., Bielak J., Biros G., Ipanomeritakis I., Fernandez A., Ghattas O., Kim E., O'Hallaron D., Tu T. "High Resolution Forward and Inverse Earthquake Modeling on Terasacale Computers." *SC2003*. Phoenix, AZ, Nov. 2003. Gordon Bell Award for Special Achievement

[5] Tu T., O'Hallaron D., Lopez J. "Etree – A Database-oriented Method for Generating Large Octree Meshes." *Proceedings of the Eleventh International Meshing Roundtable*, pp. 127– 138. Ithaca, NY, Sep. 2002. Also to appear in Journal of Engineering with Computers

[6] Tu T., O'Hallaron D. "A Compuational Database System for Generating Unstructured Octree Meshes with Billons of Elements." *To appear in SC04*. Pittsburgh, 2004

[7] Tu T., O'Hallaron D., Lopez J. "The Etree Library: A System for Manipulating Large Octrees on Disk." Tech. Rep. CMU-CS-03-174, Carnegie Mellon School of Computer Science, July 2003

[8] Tu T., O'Hallaron D. "Balance Refinement of Massive Linear Octrees." Tech. Rep. CMU-CS-04-129, Carnegie Mellon School of Computer Science, April 2004

[9] Karypis G., Kumar V. "A Course-Grain Parallel Formulation of Multi-level k-way Graph Partitioning Algorithm." *8th Siam Conference on Parallel Processing for Scientific Computing*. 1997

[10] Kim E., Bielak J., Ghattas O. "Large-scale Northridge Earthquake Simluation using Octree-based Multiresolution Mesh Method." *Proceedings of the 16th ASCE Engineering Mechanics Conference*. Seattle, Washington, July 2003

[11] Thompson J.F., Soni B.K., Weatherill N.P., editors. *Handbook of Grid Generations*. CRC Press, 1999

[12] Bank R.E., Sherman A.H., Weiser A. "Refinement Algorithms and Data Structures for Regular Local Mesh Refinement." *Scientific Computing*, pp. 3–17, 1983

[13] Young D.P., Melvin R.G., Bieterman M.B., Johnson F.T., Samant S.S., Bussoletti J.E. "A Locally Refined Rectangular Grid Finite Element: Application to Computational Fluid Dynamics and Computational Physics." *Journal of Computational Physics*, vol. 92, 1–66, 1991

[14] Griebel M., Zumbusch G.W. "Parallel Multigrid in an Adaptive PDE Solver based on Hashing and Space-Filling Curves." *Parallel Computing*, vol. 25, no. 7, 827–843, July 1999

[15] Gargantini I. "An Effecive Way to Represent Quadtrees." *Communicatoins of the ACM*, vol. 25, no. 12, 905–910, Dec 1982

[16] Abel D., J.L.Smith. "A Data Structure and Algorithm Based on a Linear Key for a Rectangle Retrieval Problem." *Computer Vision,Graphics,and Image Processing*, vol. 24, 1–13, 1983

[17] Bayer R., McCreight E.M. “Organization and Maintenance of Large Ordered Indices.” *Acta Informatica*, vol. 1, 173–189, 1972

[18] Comer D. “The ubiquitous B-Tree.” *ACM Computing Surveys*, vol. 11, no. 2, 121–137, Jun 1979

[19] Gray J., Reuter A. *Transaction Processing: Concepts and Techniques*, chap. 15. Morgan Kaufmann Publishers, Sep 1992

[20] Faloutsos C., Roseman S. “Fractals for Secondary Key Retrieval.” *Proceedings of the Eighth ACM SIGACT-SIGMID-SIGART Symposium on Principles of Database Systems (PODS)*. 1989

[21] Cormen T.H., Leiserson C.E., Rivest R.L. *Introduction to Algorithms*, chap. 9. McGraw-Hill, 1990

[22] Magistrale H., Day S., Clayton R., Graves R. “The SCEC Southern California Reference Three-Dimensional Seismic Velocity Model Version 2.” *Bulletin of the Seismological Soceity of America*, Dec. 2000

[23] Yao A.C. “On random 2,3 trees.” *Acta Informatica*, vol. 9, 159–170, 1978