

A Computational Database System for Generating Unstructured Hexahedral Meshes with Billions of Elements *

Tiankai Tu [†]

David R. O'Hallaron [‡]

Abstract

For a large class of physical simulations with relatively simple geometries, unstructured octree-based hexahedral meshes provide a good compromise between adaptivity and simplicity. However, generating unstructured hexahedral meshes with over 1 billion elements remains a challenging task. We propose a database approach to solve this problem. Instead of merely storing generated meshes into conventional databases, we have developed a new kind of software system called *Computational Database System* (CDS) to generate meshes directly on databases. Our basic idea is to extend existing database techniques to organize and index mesh data, and use database-aware algorithms to manipulate database structures and generate meshes. This paper presents the design, implementation, and evaluation of a prototype CDS named *Weaver*, which has been used successfully by the CMU Quake project to generate queryable high-resolution finite element meshes for earthquake simulations with up to 1.22B elements and 1.37B nodes.

1 Introduction

Dramatic increases in computing power and storage capacity have allowed scientists to build simulations that model nature in more details than ever. However, massive computations often require massive input and output datasets, and in our experience, the size of these datasets is rapidly outpacing scientists' ability to manipulate and use them.

For example, for the past 10 years, the Carnegie Mellon Quake group has been building computer models that predict the motion of the ground during strong earthquakes in the Los Angeles Basin (LAB) [4, 5, 14, 3]. In 1993, the largest LAB simulation code required an input unstructured finite element mesh with only 50K nodes (1.5 MB) and produced a relatively small 500 MB output dataset. By 2003, the largest LAB simulation required a mesh with 1.37B nodes (45 GB) and generated an output dataset that was over one TB [3].

With such massive datasets involved, previously routine activities, such as generating unstructured meshes, defining earthquake source models, and partitioning meshes for parallel computing, become challenging tasks. This is because unstructured datasets require complex pointer-based structures to represent, thus require massive main memory. As a result, these activities can no longer be conducted by scientists on their desktop computers or lab machines with limited main memory.

*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.

[†]Computer Science Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, USA

[‡]Computer Science Department and Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, 15213, USA

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC2004, November 6-12, 2004, Pittsburgh, PA USA

0-7695-2153-3/04 \$20.00(c)2004 IEEE

We propose a database approach to solve this problem. However, instead of merely storing unstructured datasets into conventional databases, we propose to recast the complete physical simulation process, including mesh generation, solving, and analysis, to compute directly on databases.

At the first glance, this approach appears to yield a ready solution: take an existing relational database management system (RDBMS), embed SQL commands in simulation codes and then run the codes on the database. Unfortunately, conventional databases are designed and optimized for business applications, rather than physical simulations. In typical database applications such as online transaction processing (TPC-C) and decision support systems (TPC-H), data is relatively stable. Deletions are fairly uncommon. Typical insertions and updates involve only a small portion of the database, though queries may examine large volumes of data and have a high degree of complexity. In contrast, data in physical simulations is much more dynamic, being produced, updated and removed on the fly at a high rate, while the data access patterns (queries) are usually fairly simple and straightforward. If we would implement every data access operation with a standard SQL command (*select, insert, update or delete*), the overhead, such as selecting an execution plan, updating indices and logging, would be too large for such programs to be practically useful. In other words, we would have to trade off the performance (data access speed) for new functionality (query capability) if we would use an existing database system directly.

How can we have both performance and functionality? Our idea is to develop a new kind of software systems that will (1) incorporate computational data access patterns of physical simulations into the design of the underlying database structures, and (2) export a specialized set of tightly-coupled and highly-optimized functions to interact with the databases. To differentiate such systems from conventional RDBMSs, we refer to them as *Computational Database Systems* (CDSs).

This strategy is appealing in several dimensions. First, because every step of the simulation process will work by operating on databases, the sizes of the models will naturally be limited by the amount of available disk space, rather than the amount of available memory. Therefore, computational database systems will increase the number of applications that scientists can run on their desktop computers and lab machines, thus deferring the point at which they have to use supercomputing centers.

Second, by integrating physical simulation with database systems, we will be able to build on decades of previous database research, borrowing beautiful ideas such as B-tree/R-tree indexing [6, 9, 13], linear quadtrees [11, 1], space-filling curves [10], cache-aware data layout [2] and more. Since these results are being applied to a new application domain (physical simulations), we will need to augment them with new algorithms and techniques.

Third, by managing data on behalf of simulation codes, we will have the opportunity to organize the data in such a way that best exploits locality. At runtime, such locality can be carried up through the memory hierarchy, improving performance at different levels. In addition, by understanding how simulation codes are going to access the data, we will be able to implement effective data prefetching techniques [20, 7] in the runtime system, without asking application programmers to modify their codes.

This paper makes a case for computational database systems in the context of mesh generation. We present our recent work on a prototype CDS named *Weaver* that is capable of generating massive queryable unstructured octree-based hexahedral meshes on desktop systems. Since our main focus has been to study the spatial database structures for representing unstructured hexahedral meshes and its impact on mesh generation algorithms, we have only implemented the prototype system sequentially.

The Weaver system is a major extension and improvement over our previous work on a database-oriented method for generating large octree meshes [25]. By then (2002), we were only able to generate meshes with sizes on the order of tens of millions of elements. Driven by the CMU Quake project to generate higher resolution unstructured meshes, we re-evaluated our techniques and recognized that *the characteristics of the underlying database structures should be tightly coupled into the design of the high-level algorithms*. This principle was adopted in the development of the Weaver system. In 2003, we were able to generate unstructured hexahedral meshes with billions of elements using the Weaver system. For generating and simulating earthquake ground motion in the 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 [3].

It should be noted that although motivated by large-scale earthquake simulations, the Weaver system can be used to generate unstructured hexahedral meshes for other applications. The strength of the Weaver system is that it can generate arbitrarily large hexahedral meshes on any computer as long as there is enough disk space to hold the meshes. Machines with more memory will run faster. Machines with less memory will run slower, but they will run nonetheless. In addition, the Weaver system produces meshes in the form of queryable spatial databases. Users can easily and efficiently extract parts of the mesh structures for various purposes. The limitation of the Weaver system is that it is not an on-line solution-adaptive mesh generator that can dynamically adjust a mesh structure while a solver is working on the mesh.

In the broad context of high-performance computing, our research of the Weaver system is complementary to parallel mesh generation techniques [21, 8, 15]. In general, both approaches study how to exploit spatial and temporal locality to improve the performance of mesh generation. In particular, the research of parallel mesh generation focuses more on load balancing, inter-processor communication, latency-hiding and so forth, while our research focuses more on queryable representations of unstructured meshes and physical layouts of mesh data throughout the memory hierarchy. If we employ a CDS approach in conjunction with parallel mesh generation, we will be able to enable each processor to handle a larger (sub) problem, and thus increase the overall size of the problems that can be meshed on parallel computers.

Section 2 provides an overview of octree-based unstructured hexahedral mesh generation and how to use the Weaver system to implement the process. Section 3 describes the spatial database structures used by the Weaver system to represent meshes and how to manipulate such structures. Section 4 explains the Weaver mesh generation algorithms, which exploit the characteristics of the underlying spatial database structures and provide better performance. Section 5 presents a preliminary evaluation of the Weaver system. Section 6 summarizes our work.

2 Hexahedral Mesh Generation and the Weaver System

Among many different types of meshes, octree-based hexahedral meshes lie in between the extremes of arbitrarily unstructured meshes and regularly structured meshes [23]. 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 regularly structured meshes.



(a) An unbalanced octree domain decomposition.

(b) A balanced octree domain decomposition.

Figure 1: **Balance refinement of an octree.** We have represented octrees in the form of domain decompositions to clarify the concepts. The existence of a tiny octant f triggers a *ripple effect* that causes a remote octant m to subdivide.

Because the quality of a mesh depends on whether there exists sharp changes in size between spatially adjacent elements, it is *required* that a domain-decomposition octree should not contain any spatially adjacent octants that differ more than 2-fold in their sizes. Equivalently, this means that two neighboring octants sharing an edge or a face should be at most twice as large or small. Such a requirement is often referred to as the *2-to-1 constraint*. It can be enforced either on the fly when an octree is being constructed, or by a separate step after an octree is constructed. Either way, a balanced octree is produced, as shown in Figure 1¹.

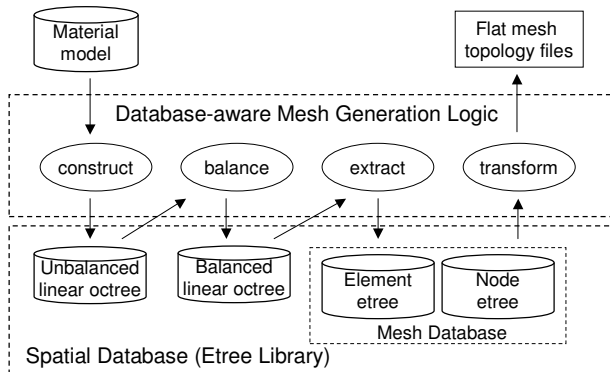


Figure 2: **Structure and Workflow of the Weaver system.**

A balanced octree is yet not a mesh. It only provides a template to generate mesh elements, nodes, and topology. Elements correspond 1-to-1 to (leaf) octants, and nodes correspond 1-to-1 to vertices of the (leaf) octants. It is a common practice to assign a unique id (from a consecutive integer sequence) to each element and create a list of element records. And similarly, a list of node records. The particular ordering of sequence numbers (ids) is immaterial since they serve the sole purpose of unique identifiers (or primary key in database term). Mesh topology is encoded using the element ids and node ids as a collection of 9-ary tuples (1 element id + 8 node ids).

The core idea of the Weaver system is to store and index (partially generated) hexahedral meshes using spatial database structures, and implement the mesh generation process by querying and manipulating the database. The structure of the Weaver system is shown in Figure 2.

Conceptually, the Weaver system consists of two parts: *spatial database* and *mesh generation logic*. The spatial database manages the unstructured mesh data such as elements and nodes on disk and in memory. A set of primitive API is exported by the spatial database layer and is used by the mesh generation logic to manipulate the data. The mesh generation logic implements different mesh generation steps by exploiting the characteristics of the database structure in order to reduce disk I/O and improve the running time (time-complexity). The following two sections explain the spatial database and mesh generation logic in detail, respectively.

3 Queryable Mesh Database Structure and the Etree Library

The primitive data objects to be manipulated in hexahedral mesh generation are octants, which are associated with an octree. An octree can be viewed in two equivalent ways: the *domain representation* and the *tree representation*. A *domain* is a Cartesian coordinate space that consists of a uniform grid of $2^n \times 2^n$ indivisible *pixels*. The *root octant* that spans the entire domain is said to be at level 0. Each child octant is one level lower than its parent (with a larger level value).

Figure 3(a) shows the domain representation of an octree. Figure 3(b) shows an equivalent tree representation of the

¹We draw 2D quadtrees to illustrate concepts but use the term octrees and octants regardless of dimensionality.

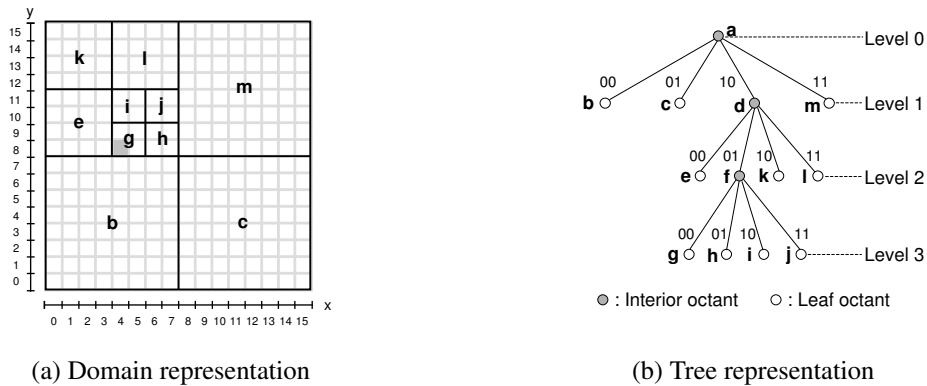


Figure 3: **Different representations of an octree.**

same octree. Each tree edge in Figure 3(b) is labeled with a binary *directional code* that distinguishes the children of each parent octant.

Linear Octree. To represent an octant, we use the *linear octree* technique [11, 1]. The basic idea of the linear octree is to encode each octant with a scalar key called a *locational code* that uniquely identifies the octant. Figure 4(a) shows how to compute a locational code. First, interleave the bits of the three coordinates of the octant’s lower left pixel to produce its Morton code [19]. Then append the octant’s level to arrive at the locational code. We refer to the lower left pixel of an octant as the octant’s *anchor*. For example, the shaded pixel in Figure 3(a) is the anchor for octant *g*.

B-tree. Given a unique locational code for each octant, we use the well-known B-tree [6, 9, 12] to index and store the 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 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 [10]; (2) It supports an important querying feature called *aggregate hits* [26]. The idea of an aggregate hit is that we can find an octant by specifying the locational code of any arbitrary point (pixel) contained in that octant. For example, Figure 4(b) shows the idea of traversing to reach octant *g* while searching for a pixel (5, 9) that is contained in *g*. To implement aggregate hits, we can modify the B-tree search algorithm slightly to return the key (of an octant) whose value is the maximum among all the keys that are less than or equal to a search key (of a pixel).

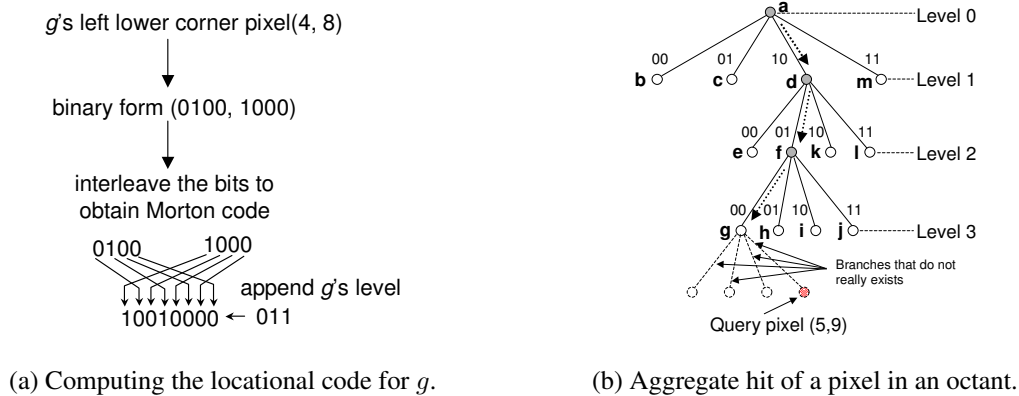


Figure 4: **Operations on octrees.**

We have developed a C library called *etree* [26] to manipulate queryable octrees on disk. Besides the linear octree and the (modified) B-tree described above, the *etree* library has the following runtime components:

Buffer manager. The buffer manager performs extensive data caching to reduce disk I/O [12] and implements a LRU policy to evict pages when there is a capacity conflict.

Schema manager. The schema manager allows an application program (for example, a mesh generator) to register a schema describing individual fields for the payload of the each octant. Once a schema is registered, the schema manager is responsible for extracting data from the payload and guarantees that platform-specific data alignment requirements and byte-ordering convention (little-endian vs. big-endian) are observed.

Metadata manager. The metadata manager keeps track of a linear octree’s structural information and records application-specific metadata.

The *etree* library operates in a 3D domain consisted of $2^{31} \times 2^{31} \times 2^{31}$ pixels. The 31-bit *etree* address space appears to provide sufficient spatial resolution for any applications we can imagine. For example, if we were to embed a continent-sized volume of 5,000 kilometers on a side in to the *etree* address space, then each pixel would have an edge size on the order of 2 millimeters.

Application programs interacts with the *etree* library through a small API, which is comprised of two types of functions: stateless and stateful. Stateless functions leave no state in the runtime system once they terminates. A stateless function call is independent of any other stateless function calls. *Insert*, *delete*, *update*, *search* and various helper functions are stateless. Stateful functions, on the other hand, set or change the state of the *etree* runtime system. *Append* and *cursor* functions are stateful. Stateful functions can be viewed as the building blocks of transactions. A state is kept until a transaction completes. During the lifetime of a transaction, no other transactions can be started and no stateless functions can be called.

Element *etree*. Using the *etree* library, we can conveniently represent and index hexahedral mesh elements, which correspond to the octants of a balanced linear octree with some application-specific information recorded in the payload of each octant. For simplicity, we refer to such a database as the *element etree*, as shown in Figure 2.

Node *etree*. Representing mesh nodes is a little more subtle because the smallest representable data object in the *etree* address space is a pixel, which is a tiny octant with some space volume, while a mesh node is a geometric entity that does not have a volume. However, noticing that a node is always located at the lower-left corner of some pixel, we may represent each node as the pixel whose lower-left corner has the coordinate of the mesh node. For example, in Figure 3(a), the node at the intersection of octants *b*, *g*, and *e* is represented as the grayed pixel. The beauty of this approach is that the set of nodes can be represented the identical way that the elements are represented. Each pixel has a locational code. So the set of nodes can also be stored in locational-code order and indexed with a B-tree, like any other linear octree. The resulting *node etree* can be viewed as a very sparse instance of the complete octree, consisting of a tiny subset of the level-31 leaf octants.

In summary, the *etree* library provides the capability to efficiently query and manipulate (partially generated) hexahedral mesh structures.

4 Database-Aware Mesh Generation Algorithms

This section provides a high-level description of database-aware mesh generation algorithms. Our purpose is not to illustrate the gory details of these algorithms, but to highlight the design principle of exploiting the characteristics of the underlying database structure.

The Weaver mesh generation logic consists of the following closely related steps, as shown in Figure 2. The *construct* step builds an indexed linear octree on disk. 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 enforce the 2-to-1 constraint. The *extract* step uses the balanced linear octree as a template to generate a queryable

mesh database that consists of an element etree and a node etree. The *transform* step queries the data in the mesh database and generate a *flat mesh topology file* that establishes the element-node connectivity relationship. This kind of files are needed by existing mesh partitioning tools [16] and solver packages [17].

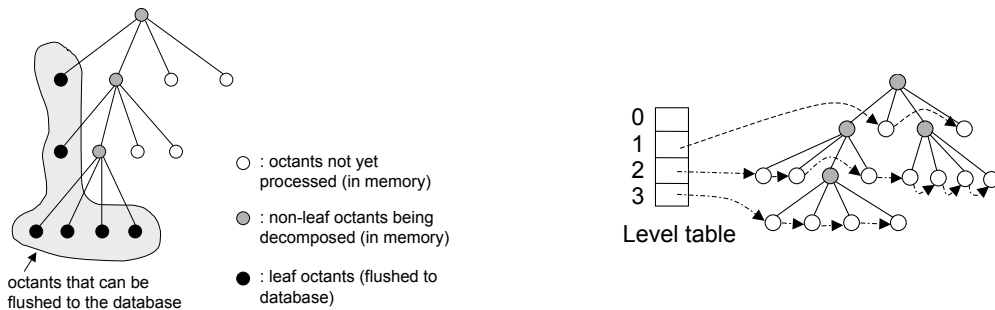
4.1 Construct: Using Auto-Navigation to Build Linear Octrees

Although linear octrees nicely solve the problem of how to address individual octants, it does not tell us how to build a linear octree efficiently. Although it is possible to construct an octree by repeatedly inserting and deleting octants from an etree, we would have to keep records of which octants have been subdivided and which have not. Worse, many insertions are in fact unnecessary because those octants are later subdivided and removed from the database. To solve this problem, We have developed a technique called *auto-navigation* [25]. The basic idea of auto-navigation is simple. Since the ordering of expanding an octree under construction is independent of the correctness of the result, the octree-traversing logic can be decoupled from the application and incorporated into the Weaver mesh generation logic.

The main data structure used to implement auto-navigation is a memory-resident pointer-based octree called a *navigation octree* (shown in Figure 5(a)) that is dynamically grown and pruned in depth-first order (same as the preorder traversal). Whether a leaf octant needs to be subdivided is determined by an application via a call-back function. With the depth-first expansion and pruning, we can guarantee that the memory requirement of a navigation octree of depth d is bounded by $O(8d)$, in contrast to $O(8^d)$ for a complete octree being constructed in the main memory.

Because the preorder traversal ordering of an octree is the same as the order imposed by the locational codes of the (leaf) octants (see Section 3), a new leaf octant being pruned off the navigation octree must have a larger locational code value than other octants already in the database (who have been pruned off earlier). Thus, instead of calling a standard insertion operation that costs $O(\log N)$, we invoke an *append transaction* to append octants that are pruned off a navigation octree to the etree database. Since append operations keep a state in the runtime system, the etree library knows where to store the new octant and thus can finish the operation in constant time $O(1)$.

Thus, by interacting with the underlying database structure through a navigation octree, we have been able to simplify the programming effort of building a massive linear octree and minimize the overall cost to $O(N)$, where N is the number octants being generated.



(a) A snapshot of a *navigation octree*.

(b) Structure of a *cache octree*.

Figure 5: Internal data structures used by the *construct* and *balance* steps, respectively

4.2 Balance: Bulk-loading Octants to Balance by Parts

The linear octrees produced by the auto-navigation process are not necessarily balanced. The process of transforming an unbalanced octree into a balanced one is known as *balance refinement*. Conceptually, balance refinement consists of two main operations: (1) *neighbor finding*: Finding the neighbors of an octant to check whether the 2-to-1 constraint is violated. (2) *subdivision*: Deleting a “too-large” octant from the database and inserting its eight children. The deletion is necessary because the linear octree datasets being balanced should contain only leaf octants.

One method for implementing neighbor finding is to manipulate the locational code of an octant to generate the keys for its neighbors and search an etree directly. The average (also worst-case) cost for a search operation is $O(\log N)$, where N is the number of octants indexed. As a result, the total cost of neighbor findings for all the octants in the dataset is $O(N \log N)$. The advantage of this method is that there is no excessive requirement on the size of main memory, as long as there is enough space to cache a few B-tree pages.

A second method is to map the linear octree to a memory-resident pointer-based octree, and then use conventional pointer-based algorithms to find neighbors [22]. The advantage is that the average cost of neighbor finding is reduced to $O(1)$, with a total cost of only $O(N)$ to conduct *all* neighbor findings. The main disadvantage is the main memory must be large enough to hold a pointer-based image of the entire linear octree. Thus, the first problem is how to take advantage of both methods. In particular, *how do we find neighbors efficiently (in $O(1)$ time) without having to map an entire linear octree in memory?*

Another performance problem is more subtle and is related to the so-called *ripple effect*. That is, a tiny octant may propagate its impact out in the form of a “ripple” that triggers subdivisions of octants not immediately adjacent to it. See Figure 1 for an example. If we were to couple neighbor-finding with tree traversal, we would have to traverse the tree multiple times to assimilate the ripple effect. However, multiple iterations of neighbor findings (and thus tree traversals) increase the total running time by a constant factor. So a second problem we try to resolve is *how to avoid multiple iterations of neighbor findings?*

Our solution is based on the observation that although balance refinement may cause ripple effect, the impact diminishes quickly due to the 2-to-1 edge size ratio. In addition, most impact caused by a tiny octant is localized in a small region. For example, octant ϵ in Figure 1 causes the subdivisions of octant a and its children. But both are spatially adjacent to ϵ . In other words, the impact of a tiny octant is absorbed mostly by octants surrounding it in a small neighborhood. The strong locality of reference suggests that we may map a small region to a pointer-based (sub)octree in memory and resolve the 2-to-1 constraint and the ripple effect without worrying about octants outside of the region. This is the type of solution that fits the paradigm of divide-and-conquer.

Our main new algorithm is called *balance by parts* (BBP) [24], which works as follows. First the domain represented by a linear octree is partitioned (divided) into equal-sized 3D volumes called *volume parts*, whose alignments correspond to non-leaf nodes (of a conceptual pointer-based octree) at a certain level. To determine the size of the volumes, we assume conservatively that each 3D volume consists of only the smallest octants of the domain. Next, each 3D volume is cached in memory and balanced. After all the 3D volumes are processed, octants on the volume face boundaries, called *face boundary parts*, are balanced, followed by the balance of octants on the volume line boundaries (*line boundary parts*) and point boundaries (*point boundary parts*). Each part, regardless of its type, is cached in a temporary pointer-based octree called a *cache octree* as shown in Figure 5(b). While balancing a cache octree in memory, we update the etree database to record the subdivisions of leaf octants.

Because each 3D volume maps to some sub-octree root whose leaf octants are clustered sequentially on B-tree pages (see Section 2), we can invoke the etree cursor operations to load all octants belonging to a 3D volume into memory. In particular, we first initialize a cursor in the etree to identify the position of the *first* octant of a 3D volume, which is defined as the octant that occurs first in the preorder traversal of the subtree represented by the 3D volume. Since the first octant is always anchored at the left-lower corner of a 3D volume, we can easily derive its locational code and thus initialize a cursor. We then repeatedly retrieve and advance the cursor to sequentially scan all the needed octants until we encounter an octant that is outside the scope of the 3D volume of interest. From the

database perspective, this is a bulk loading operation with a cost of $O(1)$ for each octant retrieved.

To retrieve octants for parts of other types, we implement range queries on octrees. For example, face boundary parts are fetched by searching for octants tangentially intersecting particular rectangles (shared by 3D volumes) in space. Since the cost of retrieving a boundary octant by searching an octree is $O(\log N)$, where N is the total number of octants, the total cost of retrieving all the boundary octants (belonging to those parts other than 3D volumes) is thus $O(b \log N)$, where b is the number of boundary octants. On average, b is at least 1 order of magnitude smaller than N (2D vs. 3D). In practice, only about 2% of a linear octree needs to be fetched by range queries.

When we cache in a part, regardless of its type, we map it to a cache octree, a special type of pointer-based octree whose leaf nodes at the same tree level are linked together and is accessible from an array called *level table* (see Figure 5(b)). The cost of building a cache octree is linear to the number of leaf nodes. We apply another new algorithm called *Prioritized Ripple Propagation* (PRP) to balance cache octrees. The PRP algorithm makes use of the pointer structure of a cache octree to conduct neighbor-finding in constant time on average (solution to the first problem). Besides, the PRP algorithm avoid multiple iterations of neighbor-findings (solution to the second problem) by accessing leaf octants directly from the level table and subdividing octants on the fly when neighbor-findings are being performed.

In summary, the structural design of the balance algorithms results in an I/O optimal case where most data is efficiently retrieved by bulk loading ($O(1)$ cost per octant) and the remainder is retrieved by standard spatial database range queries ($O(\log N)$ cost per octant). Moreover, we can avoid the costly operations of finding neighbors from the octree database and apply a fast incore algorithm to enforce the 2-to-1 constraint. The overall cost BBP/PRP is $O(N + b \log N)$, where N is the total number of octants in the linear octree and b is the number of octants in the parts other than 3D volumes, that is, octants on the face boundaries, line boundaries and corner boundaries.

4.3 Extract: Producing Mesh Nodes by Two-Level Bucket Sort

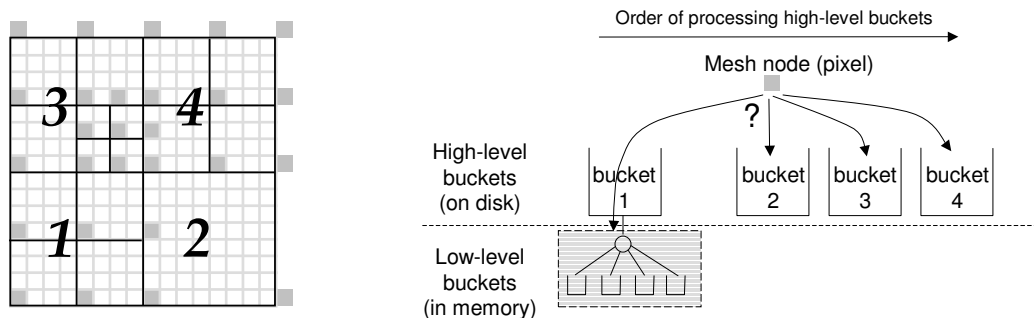
A balanced linear octree is used as a template to extract the mesh structure (elements and nodes). Because mesh elements correspond 1-to-1 to the octants, we can use a cursor operator to iterate each octant in the balanced linear octree to extract the elements. The cost is $O(N)$.

The difficulty lies in the extraction of mesh nodes. Two important issues must be considered. First, we must get rid of duplicate nodes. This is because each mesh node is shared by multiple elements. We should ignore duplicates and record each node (with unique coordinate) only once. Second, we need to distinguish two different types of nodes in the mesh: *dangling* and *anchored*. A mesh node is defined as dangling if it is located on the edge or the face of some octant. Otherwise, it is anchored. As per definition, a dangling node is dependent on either 2 anchored nodes if it is on an edge, or 4 anchored nodes if it is on a face. The dependence of dangling nodes on anchored nodes must be identified explicitly.

An obvious way to implement node extraction is to make use of the node octree database, where partially generated nodes are stored and indexed. On encountering a “new” node (computed as a vertex of a newly visited octant), we search the node octree to decide whether it is a duplicate or not. Hence, the cost of creating a new node is $O(\log M)$, where M is the number of mesh nodes. Note that we cannot simply append nodes to an octree because the order we encounter new nodes are *not* the same as the Z-order of the nodes. Therefore, the total cost of extracting mesh nodes and eliminating duplicates is $O(M \log M)$. To identify dangling nodes, we let each node record carry extra information of how many elements are sharing it and the locational codes of those elements, and then apply a post-processing procedure to analyze the geometric position of each node within the mesh and determine whether it is dangling or anchored, and if dangling, whom it depends on. Unfortunately, the running time and disk space requirement of this algorithm are both excessively large.

To overcome the obstacles, we re-define the problem from a different angle: instead of treating node extraction as a dynamic process that gradually discovers new mesh nodes, we can think of all mesh nodes as statically distributed in the domain already. Figure 6(a) shows the mesh node distribution resulted from a balanced octree. Recall that we

use the tiniest octants in the etree address space, i.e. pixels, to represent mesh nodes. Since our goal is to produce a node etree that is properly indexed, the problem of extracting mesh nodes is equivalent to the problem of sorting all mesh nodes in the domain according to their locational codes (Z-order) and load (append) them to the node etree.



(a) Mesh node distribution in a balanced octree domain. (b) A mesh node falls in either a lower-level bucket or some other high-level bucket.

Figure 6: **Treat mesh nodes as pixels and use two-level bucket sort to produce mesh nodes.**

We have developed an algorithm called *two-level bucket sort* to implement this idea. The algorithm works as follows. First, we partition the domain into equal-sized 3D volumes that map to sub-octree roots (same as the 3D volume concept used in the balance step). But we now refer to these 3D volumes as the *high-level buckets* to emphasize the fact that each volume will accommodate mesh nodes that are fully enclosed in it. For example, the domain of Figure 6 (a) 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, every mesh node (pixel) must belong to some high-level bucket. (For simplicity, ignore those mesh nodes on the far-side boundary of the domain.)

We then process the high-level buckets one by one in Z-order. For each high-level bucket, we build an incore (sub) octree to represents its octants (mesh elements), which becomes the so-called *low-level buckets*, as shown in Figure 6(b). On encountering each octant, we derive the locational codes for its eight nodes (corners). Due to the aggregate hit property explained in Section 2, each derived mesh node (pixel) will be enclosed either by some low-level bucket or by some other high-level bucket. Either way, a derived mesh node is assigned to a proper bucket.

After all mesh nodes induced by octants in the current high-level bucket are accounted for, we sort the mesh nodes assigned to the low-level buckets. The interesting aspect of this sorting algorithm is that we only need to sort mesh nodes assigned to each individual low-level bucket, respectively. Any low-level bucket contains 7 mesh nodes at maximum, though most of the low-level buckets may only contain 1 node (its own lower-left corner node). So any simple sorting algorithm can be applied. We then traverse the incore (sub) octree in preorder, appending the sorted mesh nodes of each low-level bucket to the node etree. Mesh nodes assigned to other high-level buckets will be sorted and appended to the node etree when those high-level buckets are later processed.

It can be shown that the order we visit the (high-level and low-level) buckets guarantees that the nodes are always produced in their locational codes order (Z-order) and can be safely appended to a node etree. In addition, it can be shown that dangling node identifications can be efficiently embedded in the two-level bucket sort algorithm.

4.4 Transform: Deriving Flat Mesh Topology through Spatial Join

Given an element etree and a node etree, the transform step generates flat *topology file* to represent the element-node connectivity relationship. In such a topology file, elements are identified by unique ids drawn from a consecutive integer sequence starting from 0, and so are the nodes (from another independent integer sequence). Therefore two tasks need to be accomplished: (1) id assignments, and (2) correlate element ids to node ids.

A simple way to associate ids with the elements is to traverse the element etree in the ascending locational code order and assign an element the sequence number in which we encounter it. Similarly, we can assign ids to the nodes in the same way. The cost of id assignments is thus $O(M + N)$, where N is the number of elements and M is the number of nodes. As mentioned early, the nice property of the locational code ordering is that it corresponds exactly to the Z-ordering, thus the id assignment has the property that spatially close elements are clustered together in the 1D id space. So are the case with the mesh nodes.

The second task is much more challenging. Correlating element ids to node ids represents a special type of spatial-join problem. A naive way of implementing such join operation is to visit the element one by one. Use the element locational code to derive the locational codes for its eight nodes. Then use the node locational codes to search the node etree to find out the node ids for each of the node. The cost of such an algorithm is $O(N \log M)$. Noticing that N and M are approximately the same and are usually very large (from hundreds of millions to several billions), such a cost is very expensive.

We have been working on an improved spatial-join algorithm with an average cost of $O(N + M)$. The key insight is that each element is only correlated to eight nodes, which are close-by in space. Given the clustering property of the locational codes, we only need to cache in memory those the mesh elements and nodes that are indeed correlated. While iterating elements one by one in Z-order (using the etree cursor operator), we prefetch related mesh nodes into memory (also using the cursor operator) and build a hash table to keep track of the cached nodes so that we can access mesh nodes in constant time on average.

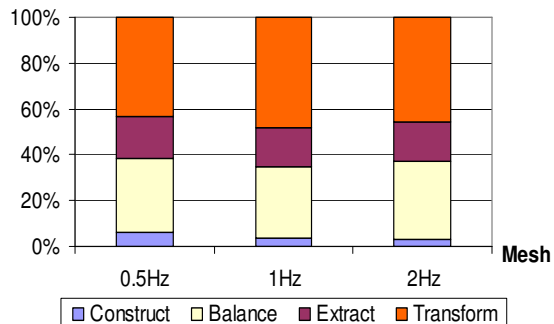
5 Evaluation

In this section, we present the performance evaluation of the Weaver system for generating massive unstructured hexahedral meshes. We have conducted experiments to answer the following two questions: (1) How effective is the Weaver system? and (2) Where does the time go while generating a mesh?

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 throughout the domain, multi-resolution unstructured hexahedral meshes allow a tremendous reduction (approx. three orders of magnitude) in the number of mesh nodes (compared to uniform meshes), because element sizes can adapt locally to the high-variable wavelength of propagating seismic waves.

Frequency	0.5 Hz	1Hz	2Hz
Num of elements	9.92M	111M	1.22B
Num of nodes	11.3M	134M	1.37B
Mesh database	340 MB	4.00 GB	45.6 GB
Flat topology files	655 MB	7.59 GB	80.5 GB
Mesh gen. time	00:05:58	01:22:06	15:13:08

(a) Summary for LAB meshes.



(b) Execution time breakdown for different steps.

Figure 7: **LAB meshes generated by the Weaver system on a Linux desktop machine and the running times.** The 2 Hz mesh was used for terascale earthquake simulations on the TCS system (Lemieux) at PSC [3].

Our target region is the Los Angeles Basin (LAB), which comprises a 3D volume of 100 km x 100 km 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 [18] (Version 3, 2002). We generate different meshes to satisfy different simulation frequency requirements. Roughly speaking, the higher the frequency, the finer (larger) the mesh.

All our experiments were conducted on a desktop machine with a PIII 1GHz processor running Linux 2.4.17. The memory subsystem consisted of 3GB physical memory and 1GB swap space.

Figure 7(a) summarizes the characteristics of three large meshes we generated. The columns correspond to the meshes that are capable of resolving 0.5 Hz, 1 Hz and 2 Hz seismic wave, respectively. The last row records the running times of the Weaver system in hh:mm:ss format. This table shows that the Weaver system is capable of generating extremely large meshes on a desktop system in a reasonable amount of time. For example, the 2 Hz mesh, with 1.37B nodes, involves creating a mesh database of size 45GB and flat topology files of size 80.5GB. If we had built all mesh data structures in main memory, we would have used more than 300 GB memory (on an Alpha system with 8-byte pointers). Given that the machine we used has only 3GB memory, generating such a massive and complicated mesh in about 15 hours (overnight) appear to be an effective solution.

Figure 7(b) shows the execution time breakdown for generating the three large meshes, respectively. Each bar represents the contribution of the four steps (construct, balance, extract, and transform) as a percentage of the total mesh generation execution time. Since the mesh database (340 MB) and the flat topology files (655 MB) for the 0.5 Hz mesh fit completely in the main memory of the experiment system (3 GB), the 0.5Hz mesh must have been generated completely from memory. Meanwhile, the 1Hz and 2Hz meshes must have been generated by interacting with the disk subsystem. The fact that all three cases have similar time breakdown patterns, as shown in Figure 7(b), suggests that the running time of each step is mostly determined by the problem size instead of database related disk I/Os. This implies that the Weaver system is exploiting locality efficiently and is processing data mostly from within memory. Otherwise, we would have seen large fluctuations due to disk I/O as the problem size increases. Also from Figure 7, we can see that although we have divided the mesh generation logic of the Weaver system into four steps, their complexities are quite different from each other, ranging from the simple bulk-loading operation (the construct step) to the convoluted spatial-join operation (the transform step). In fact, prior to the transform step, the balance and extract steps have been the performance bottlenecks, which we have resolved with the new algorithms and data structures explained in Section 4.

6 Summary

Despite the increase in computing power and storage capacity, scientists have not been able to take the full advantage of the technology trend to generate and manipulate massive unstructured simulation datasets on their desktops. We propose computational database systems (CDSs) to solve this problem. Developing CDSs requires research that lies at the intersection of database systems, computer systems and scientific computing.

This paper has presented the design, implementation and evaluation of the Weaver prototype CDS. Although still being developed, the Weaver system has already shown some merits of the idea of CDSs. First of all, the Weaver system has enabled us to generate massive unstructured hexahedral meshes on desktop machines with limited memory. Second, the meshes generated are stored in spatial databases that can be efficiently queried. Third, the design framework of the Weaver system allows us to approach mesh generation problems from a database perspective, resulting in new algorithms such as auto-navigation, balance by parts, and two-level bucket sort.

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 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.

References

- [1] D. Abel and J.L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24:1–13, 1983.
- [2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, Rome, Italy, Sep 2001.
- [3] V. Akcelik, J. Bielak, G. Biros, I. Ipanomeritakis, A. Fernandez, O. Ghattas, E. Kim, J. Lopez, D. O’Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terasacale computers. In *SC2003*, Phoenix, AZ, Nov. 2003. Gordon Bell Award for Special Achievement.
- [4] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O’Hallaron, J. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Proc. Supercomputing ’96*, Pittsburgh, PA, Nov. 1996.
- [5] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O’Hallaron, J. Shewchuk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 152:85–102, Jan. 1998.
- [6] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.
- [7] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proceedings of ICDE*, 2004.
- [8] N. Chrisochoides and D. Nave. Parallel delaunay mesh generation kernel. *Int. J. Num. Methods in Engineering*, 58(2):161–176, 2003.
- [9] D. Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, Jun 1979.
- [10] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMID-SIGART Symposium on Principles of Database Systems (PODS)*, 1989.
- [11] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, Dec 1982.
- [12] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Sep 1992.
- [13] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of SIGMOD*. ACM, Jun 1984.
- [14] Y. Hisada, H. Bao, J. Bielak, O. Ghattas, and D. O’Hallaron. Simulations of long-period ground motions during the 1995 Hyogoken-Nanbu (Kobe) earthquake using 3D finite element method. In K. Irikura, H. Kawase, and T. Iwata, editors, *2nd International Symposium on Effect of Surface Geology on Seismic Motion, Special Volume on Simultaneous Simulation for Kobe*, pages 59–66, Yokohama, Japan, Dec. 1998.
- [15] C. Kadow and N. J. Walkington. Adaptive dynamic projection-based partitioning for parallel delaunay mesh generation and refinement. In *SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, Feb 2004.

- [16] G. Karypis and V. Kumar. A course-grain parallel formulation of multi-level k-way graph partitioning algorithm. In *8th Siam Conference on Parallel Processing for Scientific Computing*, 1997.
- [17] E. Kim, J. Bielak, and O. Ghattas. Large-scale northridge earthquake simulation using octree-based multiresolution mesh method. In *Proceedings of the 16th ASCE Engineering Mechanics Conference*, Seattle, Washington, July 2003.
- [18] H. Magistrale, S. Day, R. Clayton, and R. Graves. The SCEC Southern California reference three-dimensional seismic velocity model version 2. *Bulletin of the Seismological Society of America*, Dec. 2000.
- [19] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.
- [20] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI '96)*, oct 1996.
- [21] D. Nave, N. Chrisochoides, and P. Chew. Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains. In *Proceedings of 8th ACM Symposium on Computational Geometry*, pages 135–144, Barcelona, Spain, June 2002.
- [22] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley Publishing Company, 1990.
- [23] J. F. Thompson, B. K. Soni, and N. P. Weatherill, editors. *Handbook of Grid Generations*. CRC Press, 1999.
- [24] T. Tu and D. O'Hallaron. Balance refinement of massive linear octrees. Technical Report CMU-CS-04-129, Carnegie Mellon School of Computer Science, April 2004.
- [25] T. Tu, D. O'Hallaron, and J. Lopez. Etree – a database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 127– 138, Ithaca, NY, Sept. 2002. Also to appear in *Journal of Engineering with Computers*.
- [26] T. Tu, D. O'Hallaron, and J. Lopez. The Etree library: A system for manipulating large octrees on disk. Technical Report CMU-CS-03-174, Carnegie Mellon School of Computer Science, July 2003.