Etree: A Database-Oriented Method for Generating Large Octree Meshes

Tiankai Tu[1] David R. O'Hallaron[2] Julio C. López[3]

[1] *Computer Science Department, tutk@cs.cmu.edu*
[2] *Computer Science Department and Electrical and Computer Engineering Department, droh@cs.cmu.edu*
[3] *Electrical and Computer Engineering Department, jclopez@cs.cmu.edu*
Carnegie Mellon University, Pittsburgh, PA, U.S.A.

ABSTRACT

This paper presents the design, implementation, and evaluation of the *etree*, a database-oriented method for large out-of-core octree mesh generation. The main idea is to map an octree to a database structure and perform all octree operations by querying and updating the database. We apply two standard database techniques, the linear octree and the B-tree, to index and store the octants on disk. Then we introduce two new techniques, auto-navigation and local balancing, to address the special needs of mesh generation. Preliminary evaluation suggests that the etree method is an effective way of generating very large octree meshes on desktop machines.

Keywords: octree mesh, etree, linear octree, auto-navigation, local balancing, B-tree

1. INTRODUCTION

A physical simulation generally involves three steps: (1) *mesh generation*, which models a continuous problem domain with a discrete structure; (2) *solving*, which simulates some physical process by approximating the solution of a set of partial differential equations at the grid points of the mesh; and (3) *visualization*, which creates a visual representation of the simulation results.

Traditional algorithms for physical simulation are designed to run in core, and thus the problem size is limited by the size of the main memory. When you are out of memory you are out of luck! So in order to solve large problems, scientists and engineers must somehow get access to servers or supercomputers with large memories.

Our goal is to rescue scientists and engineers from the supercomputing centers by enabling them to run large simulations using their desktop machines. The basic idea is to represent all input and output datasets in database structures stored on disk. The different steps of the physical simulation process are then performed by querying and updating these databases. With this approach, main memory serves as a cache for the database. Systems with larger memories will run faster than systems with smaller memories, but programs will always run if there is enough disk space.

The database approach exploits a number of long-term technology trends. Disk capacity is exploding and price per bit is plummeting, with gigabytes of storage available for under a thousand dollars. At the same time, mature RAID disk technology aggregates both storage and I/O bandwidth to provide hundreds of gigabytes to terabytes of fast storage for only a few thousands of dollars. Typical RAID I/O throughput (128 MB/s) is similar to typical main memory throughput (100 MB/s) [1]. After years of stagnation, DRAM prices have plummeted, from $30/MB to less than $.01/MB, with main memory sizes on typical systems increasing by an order of magnitude over the past several years. And of course, CPU speed continues to double every 18 months, with clock rates multiple GHz the norm.

This paper describes a database-oriented technique for mesh generation, the first step of the simulation process. In particular, we present the *etree*, a database-oriented method for generating large octree meshes on disk. The etree method extends existing database techniques to support the application-specific requirements of mesh generation. As a base, we use the well-known *linear octree* technique to assign each octant a unique key that encodes its location and size, and we store the octants in a well-known database structure known as a *B-tree*. In addition, we have developed two new techniques, called *auto-navigation* and *local balancing*, that address the special needs of octree mesh generation.

All of these components (linear octrees, B-trees, auto-navigation, and local balancing) are implemented in a C library called the *etree library*. An application uses the functions defined in the etree library to manipulate an octree mesh stored on disk. The etree library automatically performs extensive optimization to improve running time and reduce disk I/O. Our experiments show that with the etree method, it takes about 2.6 hours to generate a very large finite element octree mesh (4.3 GB with 13.6 million elements) on a machine with only 128 MB main memory.

Section 2 briefly discusses different approaches for generating octree meshes. Section 3 describes the etree method. Section 4 presents the the etree library and its components. Section 5 evaluates various aspects of the etree library with an etree-based finite element mesh generator.

2. OCTREE MESH GENERATION

An *octree algorithm* recursively subdivides a three-dimensional problem domain into eight equal size *octants* until a desired resolution level is achieved [2]. For two-dimensional domains, an analogous *quadtree algorithm* recursively subdivides the domain into four *quadrants*. Quadtrees are easier to draw and understand than octrees, so we will use them whenever we need to illustrate basic concepts. The techniques we describe in this paper generalize to all dimensions. For simplicity, we will refer only to octants and octrees, regardless of dimensionality.

Octrees can be drawn in different but equivalent ways. Figure 1 shows the *domain representation* of an octree, where the octree is drawn as an explicit decomposition of some rectangular domain. Figure 2 shows an equivalent *tree representation*, which depicts the tree induced by the decomposition in Figure 1. Both representations are useful in different contexts. For example, the idea of local balancing (Section 4.4) is most easily explained using the domain representation. On the other hand, ideas such as auto-navigation (Section 4.3) and preorder traversal of leaf nodes is best explained using

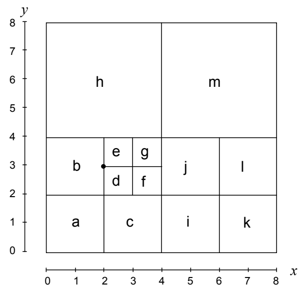

Figure 1: Domain representation of an octree.

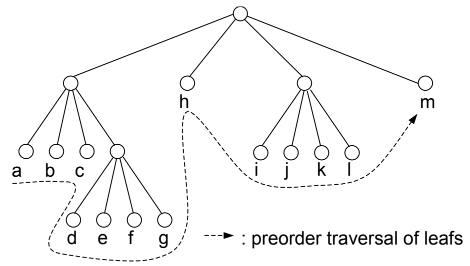

Figure 2: Tree representation of an octree.

the tree representation.

Octree decomposition has proven to be a successful strategy for generating three-dimensional adaptive meshes. One approach warps the leaf octants to obtain tetrahedral elements [3, 4]. The other main approach uses the leaf octants as finite elements directly without further modification [5, 6, 7]. To ensure good element quality, both methods require that the octree be *balanced*. That is, two leaf octants sharing a face or an edge are no more than twice as large or small (*2-to-1 constraint*). For example, the octree in Figure 1 does not satisfy the 2-to-1 constraint because the edge length of h is four times larger than that of e.

In this paper, we focus on how to generate a balanced mesh that uses the leaf octants directly as elements, which we will refer to as an *octree mesh*.

Octree meshes do have limitations, especially in their ability to model complex geometry. For those cases, more sophisticated techniques such as unstructured tetrahedral meshes are needed. But still, octree meshes represent an important class of meshes for applications with simpler geometries. For these applications, the octree meshes provide a good compromise between the structure and the modeling power.

An octree mesh can be constructed and balanced using either in-core or out-of-core methods. An in-core algorithm accommodates the octree in virtual memory, which is an operating system mechanism that enables an application to allocate much more memory than is physically available. The virtual memory uses the main memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active areas in main memory. This mechanism works silently and automatically, without any intervention from the application program. However, if the allocated virtual memory far exceeds the main memory size and the data accesses are not localized, severe swapping of data between the disk and the memory will occur, and performance degrades drastically. Thus, in practice in-core algorithms are limited to the size of the main memory.

An out-of-core method, on the other hand, uses the memory directly as a cache for the disk-resident octrees. The size of the octree mesh is thus limited by the size of the disk instead of the size of the much smaller main memory. The critical issue is to decide which part of the octree should be cached in the memory so that most data accesses can directly read from or write to the main memory.

One candidate solution is to use the *out-of-core pointer* method [8] where each in-core pointer is mapped to an out-of-core pointer in the form of ⟨*disk page number, offset*⟩. The operation of following a pointer in an in-core algorithm is transformed into seeking an offset on a disk page and then retrieving the data object. The application program must make arrangements to ensure that the out-of-core pointers are not scattered randomly on disk pages. Otherwise, the performance will be dominated by disk I/O latency. This method, though conceptually simple, is not easy to implement. And the result is often application-specific.

3. THE ETREE METHOD

The design goal of the etree is to provide a general method for efficiently generating large octree meshes out of core. Our approach leverages and extends database techniques to address the special needs of mesh generation.

Figure 3 shows the process of generating a mesh using the etree method. In the *construct* step, an octree is constructed in the same way as in an in-core algorithm, except that it is built and stored on disk. The decompositions of the octants are dependent on the geometry or physics being modeled. The result is an unbalanced octree. Next, the *balance* step recursively decomposes all the (big) octants that violate the 2-to-1 constraint until no illegal conditions exist. Finally, in the *transform* step, mesh-specific information such as the element-node relationship and the node coordinates are derived from the balanced octree and stored in two databases, one for the mesh elements, the other for the mesh nodes.

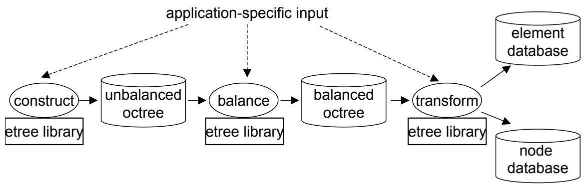

Figure 3: The etree method of generating octree meshes.

Conceptually, this process could be implemented using a traditional database system. A unique key can be assigned to each octant that encodes the location and size of an octant. Then the octants can be treated as *records* and stored in a *table* that corresponds to the octree. Since the structural information is already encoded in the key value, the non-leaf octants, which serve for the navigation purpose, can be optionally excluded from the table. Operations on the octree can be translated to *SQL* queries, a high-level declarative query language, to the database.

However, a direct database approach introduces tradeoffs. On the one hand, explicit pointer chasing is avoided. All operations are done through uniform queries to the database, which adds simplicity. On the other hand, an explicit operation history, such as a stack, must be maintained to record which octants have been processed and which have not. This adds complexity to the application program. In order to avoid these negative aspects, we use database techniques as one of the building blocks of the etree, and extend the core database functionality by introducing new techniques that support octree-level operations.

4. THE ETREE LIBRARY

Figure 4 shows the components of the etree library. An application accesses the library through a simple well-defined Application Programming Interface (API). The library is divided in the following modules: (a) *linear octree*, a powerful encoding scheme to assign keys to octants; (b) *auto-navigation*, a mechanism for traversing the octree automatically; (c) *local balancing*, a technique that speeds up octree balancing operations; and (d) *B-tree*, a database index structure for storing and accessing octants on disk.

4.1 The etree API

The etree API is inspired by Unix file I/O and can be categorized into three classes:

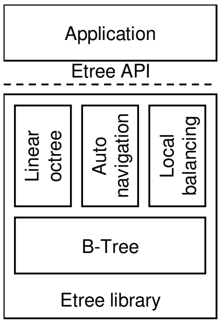

Figure 4: Etree architecture.

- *Initialization and cleanup*: An etree can be opened/created and closed as if it were an ordinary file.
- *Octant-level operations*: The etree provides a complete set of operations to manipulate octants stored on disk. For example, searching for an octant or inserting a new octant. Each octant is addressed by an abstract data type `location_t`, which we will explain in Section 4.2.
- *Octree-level operations*: This is the distinctive feature of the etree method. Instead of pushing the workload to the applications, the etree library supports octree-level operations directly. For example, the function call `etree_construct` automatically accomplishes the *construct* step in Figure 3. The only input from the application is a function that the etree library uses to determine how to process an octant.

Figure 5 lists the specific functions.

4.2 Linear Octree

In order to map an octree to a database structure, we use the well-known *linear octree* technique [9]. This method assigns a unique key to each leaf octant. The key encodes both the location and the size of the octant. We can then use a database structure such as the B-tree (see Section 4.5) to index the keys.

Octants are assigned keys using a variant of the well known Morton code [10]. A Morton code maps a d-dimensional point to a one-dimensional scalar. The mapping can be computed quickly by interleaving the bits of the binary representations of the coordinates of d-dimensional point.

The key associated with an octant is computed as follows. First we compute the Morton code of the left-lower corner of the octant. Next, we append the level

```
/* Initialization and cleanup */
   etree_t *etree_open(const char *path, int flag, ...);
   int etree_close(etree_t *ep);

/* Octant-level operations */
   int etree_insert(etree_t *ep, location_t loc, const void *value);
   int etree_search(etree_t *ep, location_t loc, location_t *hitloc, void *value);
   int etree_update(etree_t *ep, location_t loc, const void *value);
   int etree_delete(etree_t *ep, location_t loc);
   int etree_append(etree_t *ep, location_t loc, void *value);

/* Octree-level operations */
   typedef int decom_t(location_t loc, const void *value, void *childvalue[8]);
   int etree_construct(etree_t *ep, location_t rootloc, const void *rootvalue,
                       decom_t *autodecom);
   int etree_balance(etree_t *ep, decom_t *baldecom);
   int etree_sprout(etree_t *ep, location_t loc, const char *childvalue[8]);
   int etree_initcursor(etree_t *ep, location_t loc);
   int etree_getcursor(etree_t *ep, location_t *loc, void *value);
   int etree_advcursor(etree_t *ep);
```

Figure 5: Etree API.

of the octant to the Morton code in order to distinguish the octant from any ancestors that share the same left-lower corner. The result is a unique key assignment for each octant, which is usually referred to as the *locational code* [2]. For example, Figure 6 shows how we compute the locational code 00110111_2 for octant e in Figure 1. Notice that the underscore in the locational code in the figure is for illustration purposes only; a locational code is just a binary string.

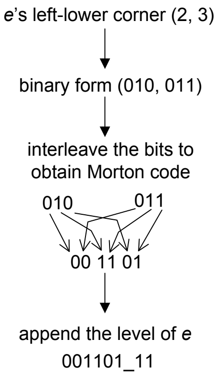

Figure 6: Interleaving bits and appending level to obtain the locational code.

To isolate users from the details of the bit-interleaving and level appending, we define an abstract data type `location_t` that can be used to address an octant. It consists of the 3D coordinates of an octant's left-lower corner and the octant's level in the octree, as shown in Figure 7. With this unique key, we can find an octant directly. Therefore, we only store leaf octants in the etree database.

```
typedef struct location_t {
    unsigned long long x, y, z;
    int level;
} location_t;
```

Figure 7: Definition of `location_t`.

An important property of the Morton code is that the ordering of leaf octants based on their locational code is exactly the preorder traversal of the octree leaf nodes. For example, Figure 8 shows the ordering, based on locational codes, for the octree in Figure 2.

a → b → c → d
000000_10 000100_10 001000_10 001100_11

e → f → g → h
001101_11 001110_11 001111_11 010000_01

i → j → k → l → m
100000_10 100100_10 101000_10 101100_10 110000_01

Figure 8: Leaf octants in Figure 2 sorted according to their locational code.

There are two important applications of this property in the etree library: (1) clustering nearby octants on disk pages; (2) finding an octant without knowing its exact locational code. The first application exploits the fact that the preorder traversal of the leaf octant in the corresponding domain follows a Z pattern, as shown in Figure 9. Such an ordering is called the *Z-order* [11, 12], which tends to cluster spatially close data points in their one-dimensional ordering [13]. Therefore, we can store the leaf octants sequentially on the disk pages according to their locational code, which naturally results in the clustering of nearby octants.

The second application is more subtle and and is best explained with an example. To find the neighbor on the south side of octant e in Figure 1, we assume the neighbor is of equal size as e and derive the neighbor's locational code, (001100_11_2). Searching the database will return octant d, since its key matches the search key. Similarly, to find the neighbor of e on the north side, we derive a key (011000_11_2). However, searching the database for this key does not return an exact match since there is no such leaf octant in the domain. Observe that the expected neighbor is actually octant h whose key is (010000_01_2). We can see from Figure 8 that h's key value is the maximum among all the keys that are less than the search key (011000_11_2).

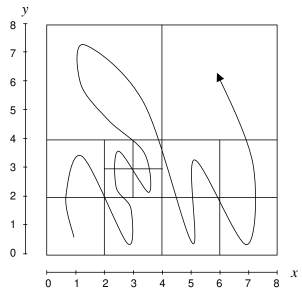

Figure 9: Z-order curve through the domain.

This occurrence is not accidental. In the preorder traversal, a subtree root is always visited earlier than any of its descendants. Hence, the key value, i.e. the locational code, of the subtree root is always less than those of its descendants. Therefore, the locational code of the subtree root is the lower bound for the interval that covers these key values. In a degenerate case where the subtree root is a leaf octant, its locational code can be viewed as the lower bound for the key interval that covers the non-existent *virtual* subtree. In other words, an attempt to find an octant in the virtual subtree will search the database using a locational code that falls in the virtual subtree's key interval. Since no octants in the virtual subtree actually exist in the database, the locational code that is the maximum among all the keys that are less than the search key corresponds to the lower bound for this interval. Therefore, we are able to find an octant, such as the neighbor octant h in our example, without knowing its exact locational code.

4.3 Auto-navigation

The linear octree nicely solves the problem of how to address individual octants. However, it does not provide a programming model for octree construction. Although it is possible for an application to construct an octree by repeatedly inserting and deleting octants from the database, the application program would have to keep records of which octants have been decomposed and which have not. Worse, many insertions are in fact unnecessary because those octants are later decomposed and removed from the database.

To address this problem, the etree provides a higher-level abstraction to support automatic octree construction through the function call `etree_construct`. The underlying technique is called *auto-navigation*. The idea of auto-navigation is based on a simple insight: since the ordering of expanding an octree under construction is independent of the correctness of the result, the octree traversal logic can be decoupled from the application's logic and incorporated into the etree library.

Auto-navigation is implemented in the etree library through a data structure called the *navigation octree*, which is an in-core octree that is dynamically grown and pruned in the main memory, as shown in Figure 10. In addition, the application provides a *splitting predicate* that takes an octant as input and determines whether or not the octant should be decomposed.

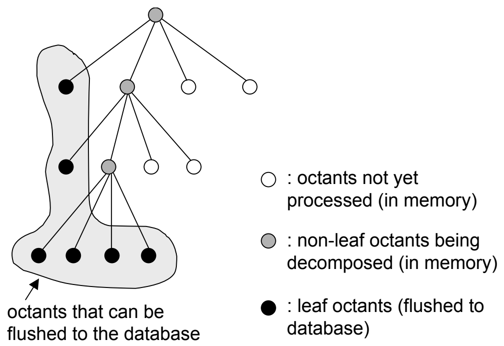

Figure 10: Auto-navigation through an octree being constructed.

Initially, the navigation octree consists of the root octant. The splitting predicate is invoked to decide what to do with the root. If a decomposition is needed, eight children octants are allocated and linked to the root using ordinary in-core pointers. This procedure is then continued in depth-first order. Whenever the splitting predicate determines that an octant does not need to be decomposed, then we know for certain that it is a leaf node that can be safely pruned from the navigation octree and flushed to the database, typically in a bulk transfer with other leaf octants. With this depth-first expansion and pruning, we can guarantee that the memory requirement of a navigation octree of depth d is bounded by $\mathcal{O}(8d)$, in contrast to $\mathcal{O}(8^d)$ for a complete octree being constructed in the main memory.

With auto-navigation, applications are relieved from the burden of traversing an out-of-core octree for expansion, which is a complicated and error-prone task. Further, database operations can be significantly optimized. Since the order (preorder) of flushing the leaf octants is the same as the order imposed by the locational codes, a new leaf octant that is pruned off the navigation octree can be appended to the end of all octants that are currently stored in the database. In fact, the complexity of an append operation is only $\mathcal{O}(1)$.

4.4 Local balancing

An octree obtained after the construction step must be balanced to conform to the 2-to-1 constraint. For example, the initial domain decomposition shown in Figure 1 violates this constraint because octant e has a neighbor h on the north whose edge size is four times as large as e's. A balancing operation will discover this illegal status and decompose octant h further to obtain a legal octree mesh as shown in Figure 11.

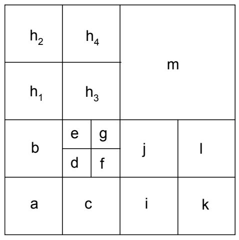

Figure 11: A balanced domain decomposition.

The etree library provides a function call `etree_balance` to isolate the applications from the details of enforcing the 2-to-1 constraint. One straightforward way to implement this function is to iterate through all the octants in the unbalanced octree and check their neighbors to determine whether further decomposition is necessary. We refer to this approach as *global balancing*. Though conceptually simple, global balancing has two major drawbacks. First, multiple iterations through the domain may be needed to propagate the impact of a tiny octant. Second, each neighbor-finding operation will incur the cost of searching the database.

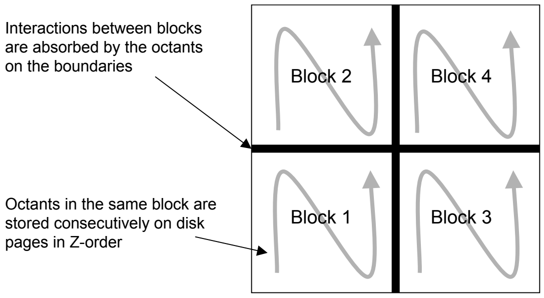

Figure 12: Local balancing.

We avoid global balancing by doing *local balancing*, which consists of three steps. First, we partition the whole domain into *equal-size blocks*. Next, we conduct *internal balancing* to enforce the 2-to-1 constraint within each block. Finally, we do *boundary balancing* to resolve interactions between adjacent blocks. Figure 12 gives a conceptual view about this scheme.

Appropriate block size: The size of a block is chosen to satisfy the following two conditions: (1) it is equal to the size of some subtree root's size; (2) it is at least as large as the largest leaf octant in the domain.

With the first condition, each block is mapped to a subtree root. (We assume the blocks are aligned properly.) We will use this condition to prove the correctness of the local balancing scheme. With the second condition, we can guarantee that all the octants in the domain can fit into some block and thus be processed.

Internal balancing: Internal balancing is performed *block by block*. We traverse the entire domain, one block at a time. For each block, we read all the octants that belong to the block. Based on the position and size information retrieved, we initialize a structure called the *blocking array*. The cardinality of each dimension of the array is set to be the size of the block divided by the size of the smallest octant in the domain.

The blocking array is initialized in the following manner. At beginning of processing each block, all array elements are set to be 0. Upon reading a new octant, we determine the position of the octant's left-lower corner relative to the left-lower corner of the containing block. Suppose the position is $\langle i, j \rangle$, then the array element $[i, j]$ is set to be the size of the octant divided by the size of the smallest octant in the domain. Figure 13 shows the content of the blocking array after retrieving information of the octants belonging to the left-lower block of the domain shown in Figure 1.

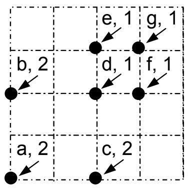

Figure 13: Content of the blocking array.

After the blocking array is initialized, we can resolve the 2-to-1 constraint within the block without querying the database. The reason is that all the information regarding octants in a vicinity (block) has already been recorded in the blocking array. Finding a neighbor (of equal size) is done by modifying the array indexes and accessing an array element directly.

To be more specific, we iterate through the blocking array element by element. For each element $[i, j]$, we check whether there is an octant anchoring its left-lower corner at $\langle i, j \rangle$ and if so, whether its neighbors will trigger its further decomposition. The fact that an octant needs to be decomposed is recorded but the decomposition is not performed immediately. For a decomposing octant, we initialize the array elements corresponding to its eight children. Notice that one of its children must have the same left-lower corner as itself. Therefore, the old size value at $[i, j]$ will be overwritten. By doing so, we always keep the latest octant information in the blocking array. It should be noted that several iterations may be needed to propagate the impact of a tiny octant.

After processing all the blocks, the etree obtains a complete list of octants that violate the 2-to-1 constraint locally and need to be further decomposed. Then the actual database operations of deleting and inserting octants are carried out in batch mode.

Also, at the time of processing each block, the octants on the block's boundary are recorded in a separate list.

Boundary balancing: In this step, we iterate through the boundary octant list, checking each octant's neighbors to determine whether the 2-to-1 constraint is violated, and if so, performing a decomposition. During each iteration, a list of new boundary octants is generated, which is used as the input to the next iteration.

Note that the total cost of searching neighbors in the database for boundary balancing is much cheaper than the case of global balancing because the number of boundary octants is far less than the number of the octants in the entire domain.

Correctness of local balancing: We first study the relationship between a boundary octant b and its *internal* neighbors in the same block after internal balancing is performed. Since the 2-to-1 constraint is enforced locally inside this block, there are only three possible scenarios for the internal neighbors of b, as shown in Figure 14: (1) half as large as b; (2) as large as b; or (3) twice as large as b.

However, scenario (3) cannot occur. Because the first condition imposed on the block size requires that each block maps to a subtree root, thus octant b must be a descendant of this subtree root (block). In other words, octant b must be a child of some octant that is twice as large as b. However, since an octree is a disjoint decomposition of the domain, it is impossible

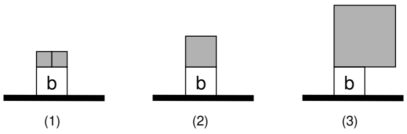

Figure 14: Three scenarios for internal neighbors of a boundary octant.

for octant b to have an internal neighbor that overlaps the region covered by b's parent, as shown in Figure 15.

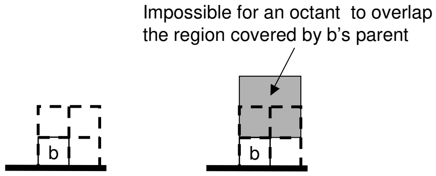

Figure 15: Scenario (3) is impossible to occur.

Now we can show the correctness of the local balancing scheme. We prove, by induction on the number of iteration of boundary balancing, that the decomposition of any boundary octant b will not cause decompositions of its internal neighbors. That is, the interactions between adjacent blocks are always *absorbed* by octants on the boundaries between blocks and will never propagate into the blocks. In the following proof, we only consider non-trivial cases where a boundary octant b does decompose during an iteration of boundary balancing.

For the base case (the first iteration of boundary balancing), the relationship between a boundary octant b with its internal neighbors is either scenario (1) or (2).

Figure 16 shows that if scenario (1) holds for the base case, the decomposition of b will result in four children octants of equal size to b's internal neighbors. Therefore, no decomposition is needed for b's internal neighbors. On the other hand, the two children of b in the bottom half become the new boundary octants whose internal neighbors are their siblings of the same size. Consequently, scenario (2) is the relationship between the new boundary octants and their internal neighbors. (For clarity, we use unfilled (white) squares to represent boundary octants and filled (dark) squares to represent internal octants.)

Figure 17 shows that if scenario (2) holds for the base case, the decomposition of b will result in four children

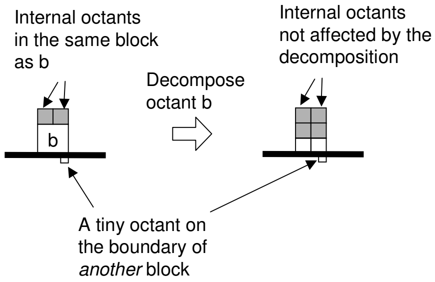

Figure 16: Scenario (1): internal neighbors not affected by the decomposition of the boundary octant.

octants half as large as b's internal neighbor. Since the 2-to-1 constraint is still maintained, no decomposition of b's internal neighbors is needed. Again, the relationship between the new boundary octants and their internal neighbors is scenario (2).

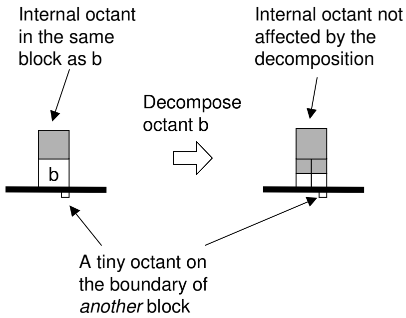

Figure 17: Scenario (2): internal neighbors not affected by the decomposition of the boundary octant.

The inductive case is straightforward to prove. Because scenario (2) is the only possible relationship between the boundary octants and their internal neighbors for iteration k, we can use the same argument for the base case to show that the internal neighbors do not need to be decomposed and scenario (2) will hold for iteration $k + 1$.

Note that in Figures 16 and 17, it is a drawing artifact that the *tiny* octant on the boundary of another block appears to be one fourth of the size of octant b. It seems that only one iteration of boundary balancing will suffice to enforce the cross-boundary 2-to-1 constraint. But in general, the tiny octant can be arbitrarily small. Thus, multiple iterations of boundary balancing will be needed to enforce the cross-boundary 2-to-1 constraint. Also, the *trigger* that causes the decomposition of a boundary octant b can be either a tiny octant in another block as shown in Figure 16 and Figure 17, or a newly generated tiny boundary octant in the same block as b, not shown pictorially in the figures. The proof given above is valid for the general cases as well as for both types of triggers.

In summary, the local balancing scheme is correct because the impact between adjacent blocks is always absorbed by the (dynamically changing) boundary octants.

4.5 B-tree

The B-tree is the most important index structure in database and file systems [14, 15, 16, 17]. Figure 18 shows the structure of a B-tree. There are two types

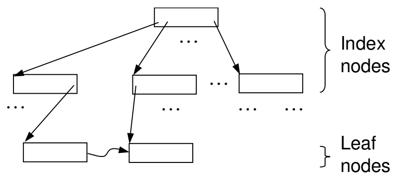

Figure 18: B-tree structure.

of nodes in a B-tree: the *leaf nodes* and the *index nodes*. The leaf nodes contain data to be searched. The structure of a leaf node is an array of records with the form $\langle key, data \rangle$, shown in Figure 19. The entries are stored in ascending key order and all the keys in leaf node is smaller than any key stored in the next leaf node.

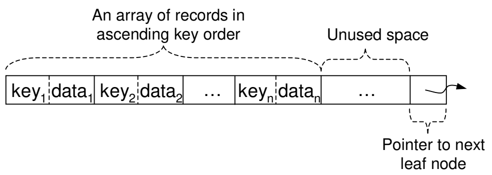

Figure 19: B-tree leaf node.

The index nodes contain routing information to guide the search for a given key value. The structure of an index node is an array of pairs $\langle key, pointer \rangle$, shown in Figure 20. These entries are also stored in ascending order. The sequence of keys in an index node ($K_1 < K_2 < \ldots < K_M$) divides the search space covered by that node. Each key value K_i has an associated pointer P_i, which points to a successor node that contains further information about all keys K_x such that $K_i \leq K_x < K_{i+1}$.

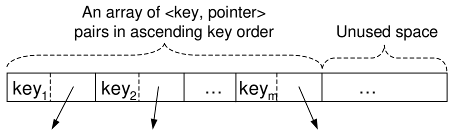

Figure 20: B-tree index node.

B-tree nodes are mapped to disk pages. B-tree *pointers* are actually disk page numbers. A B-tree index node can have a large number of pointers, in contrast to a binary tree where each index node has at most two successors. As a result, B-tree tends to be wide and short in structure.

Operations on a B-tree are defined in such a way that the B-tree structure is always balanced. That is, every path from the root to the leaf always has the same length. Therefore, the dominant performance factor — disk page accesses, is always the same for all the search operations.

Searching for a key k starts from the B-tree root, which is an index node. A pointer P_i is followed such that $K_i \leq k < K_{i+1}$. If P_i leads to an index node, repeat the procedure. Otherwise, search the leaf node for the entry with a matching key value.

Inserting a new record into the B-tree may cause a split of a B-tree node into two if the node is fully occupied. Similarly, deleting a record may cause two B-tree nodes (at the same level) to be merged if one of the node's occupancy drops below 50%. As a result, a B-tree has a minimum space utilization of 50% and an average of 69% [18].

5. EVALUATION

In this section, we present the performance evaluation of the etree. We conducted a series of experiments to answer the following questions: (1) Is the etree method feasible? (2) How does the running time vary with the physical memory size? (3) What is the impact of auto-navigation? (4) What is the impact of local balancing?

5.1 Methodology

In order to evaluate the etree method in a real-world scenario, we developed an etree-based octree mesh generator that produces a family of finite element meshes for San Fernando earthquake wave-propagation simulations [19].

Figure 21 summarizes the characteristics of each mesh, which comprises an identical volume of 50 km × 50 km × 12.5 km. Roughly speaking, mesh sf$_k$ is sufficiently fine to resolve a wave with a period of k

seconds, under the assumption of 10 mesh nodes per wavelength. The *Elements* and *Nodes* columns contain the total number of finite (octant) elements and nodes in the domain, respectively. The *Slave Nodes* column records the number of the nodes that are located on an edge or a face of another element, which is a subset of the total nodes in the mesh.

Mesh	Elements	Nodes	Slave Nodes
SF10	7,940	12,118	4,432
SF5	76,330	105,886	34,858
SF2	1,838,524	2,213,035	407,336
SF1	13,597,124	15,097,365	1,649,855

Figure 21: Summary of San Fernando meshes.

The mesh generation process is driven by the material model developed by Harold Magistrale and Steve Day at San Diego State University [20]. We sampled the material model and stored the result in an etree database, which we will refer to as the *material database*. The size of the material database is 785 MB.

We conducted all the experiments on a PIII 1GHz machine with Ultra 160 SCSI controller and disk running Linux 2.4.17. The physical memory for the experiments ranges from 128 MB to 880 MB. Before each experiment, we sequentially scan two 1.5 GB files to flush the operating system's buffer cache.

5.2 Is the etree method feasible?

To answer this question, we generated meshes of different sizes and measured the time required to generate them. For this experiment we configured the physical memory to be 128 MB, and fixed the size of the B-tree buffer in the etree library to 8 MB and the size of the blocking array to 16 MB. Figure 22 shows the elapsed wall clock times to generate meshes of different sizes. Although these measurements are specific to this particular application, they show the result of generating a non-trivial, real-world octree mesh, serving as a good indication of whether the etree method is feasible in terms of running time and memory requirement.

Mesh	Elements	DB size (MB)	Time (s)	Thrput (elem/s)
SF10	7,940	2.5	39.9	199
SF5	76,330	24	186.0	410
SF2	1,838,524	583	1,636.7	1,123
SF1	13,597,124	4,300	9,448.8	1,439

Figure 22: Etree-based mesh generator running time and throughput.

Our first observation is that the total running time to generate our largest mesh, SF1, is approximately 2.6 hours. Although we have no benchmark to compare our results, generating a 13.6 million element mesh (4.3 GB) in the order of 2 to 3 hours appears to be reasonable.

Second, the overall throughput increases with mesh size. Our intuition is that since the etree library caches mesh data in its memory buffer, as more data access to the cache occur, higher throughput is achieved. Therefore, larger mesh generations are more likely to benefit from the caching. The bottom line is that the throughput does not decrease as the mesh size increases, which implies that the total running time increases at most linearly as the mesh size increases.

Third, all the experiments are performed on a machine with only 128 MB of physical memory. The buffers allocated by the etree library is only 24 MB. Thus, the etree method is a feasible solution to generate large octree meshes on memory-limited machines.

5.3 How does the running time vary with the physical memory size?

We are not only interested in the effect of memory size on the running time of our method, but also interested in determining the running time of etree related operations and application-specific computation. We perform an experiment similar to the one described in the previous section; we generated the meshes with the same library parameters, i.e., B-tree buffer and blocking array size, except that this time we varied the amount of physical memory available to operating system.

Besides the total running time, we measured the time for the following operations of our mesh generator: (1) *construct*, (2) *balance*, (3) *transform*, (4) *query* and (5) *findslave*. The first three operations are general etree operations, i.e., excluding application-specific computation, in the corresponding construct, balance and transform step described in section 3. These operations account for the time executing library code to navigate and search the etree. Query is an application-specific operation and accounts for the time required to query the materials database in all the steps of the mesh generator. Findslave is also an application-specific operation and accounts for the time required to compute the master / slave relationship between nodes.

Figure 23 shows the running time for the mesh generation process. The results are presented in four groups, one for each mesh. Each group presents the running time for a given mesh with different physical memory configuration. Within each group the running time is normalized to the 128 MB case. We can see that the performance improves slightly (less than 15%) as the physical memory size increases from 128 MB to 880 MB.

The figure also shows us that the general etree op-

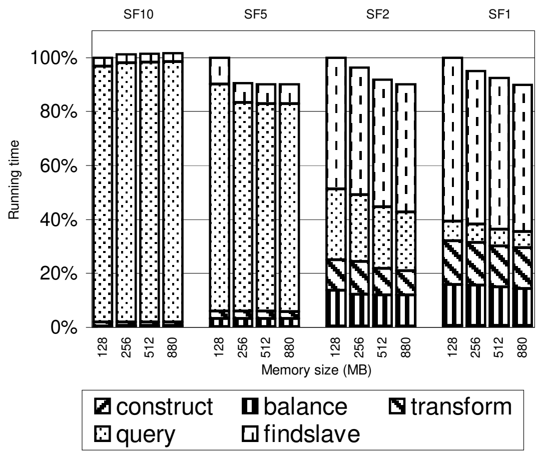

Figure 23: Total running time vs. Physical memory size.

erations of construct, balance and transform account for at maximum 30% of the total running time. Figure 24 magnifies the running time of the general etree operations to a larger scale. Again, there is no significant performance change as the physical memory size increases.

Both sets of results show that the memory size does not have a significant impact on running time. We also deduce from the results that the etree is not relying on the operating system's internal caching mechanism to achieve its performance.

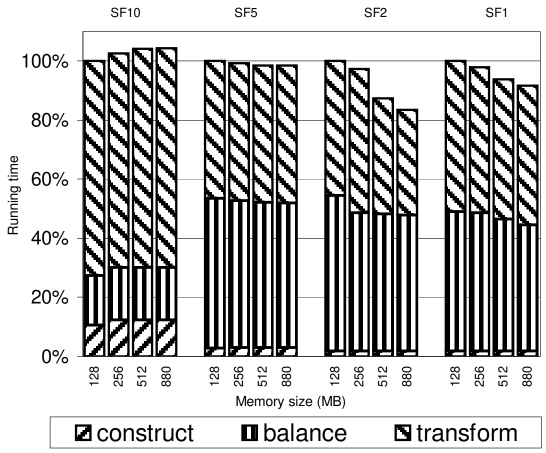

Figure 24: Etree operation running time vs. Physical memory size.

5.4 What is the impact of auto-navigation?

For these experiments we made 880 MB of physical memory available to the operating system and varied the size of the B-tree buffer in the etree library. Figure 25 shows the effectiveness of the auto-navigation technique. In all four cases, the construction time of the etree does not depend on the size of B-tree buffer as long as there is a buffer, even if it is a small one. auto-navigation requires only small amount of memory footprint to cache the B-tree nodes on the path from the B-tree root node to the rightmost B-tree leaf node, which is no more than a few disk pages. Reducing the B-tree buffer size does not increase the octree construction time as long as the buffer is big enough to hold this path.

We can also see from Figure 23 and Figure 24 that the construct step only accounts for a very insignificant portion of the total running time. This is a proof that the auto-navigation is very effective and does not deserve further effort for optimization.

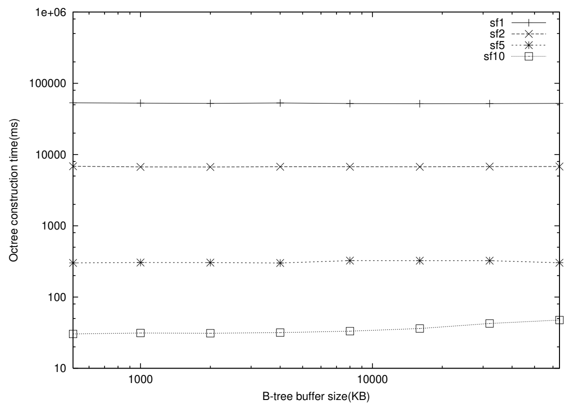

Figure 25: Octree construction time vs. B-tree buffer size (log-log scale)

5.5 What is the impact of local balancing?

In these experiments we fixed the amount of the physical memory and the B-tree buffer size and varied the blocking array size. Figure 26 shows the effect of local balancing. The x-axis is the maximum blocking array size allowed in the etree library. The y-axis is the time to balance an octree. The data-points plotted on the y-axis, i.e., $x = 1$, corresponds to the case where local balancing is disabled and global balancing is performed. In all the four cases, a significant reduction in execution time is achieved by enabling the blocking array and doing local balancing. The speed-up factor ranges from 8x (SF1) to 28x (SF10).

A simple analysis can show why there is a wide discrepancy between the speed-up factors. The performance gain of local balancing comes from two aspects: the faster array-based neighbor-finding algorithm, and the one-time traversal of the domain. In the case of SF10, the size of the mesh etree is small enough to be entirely cached in the etree memory buffer, thus multiple traversals through the domain required by

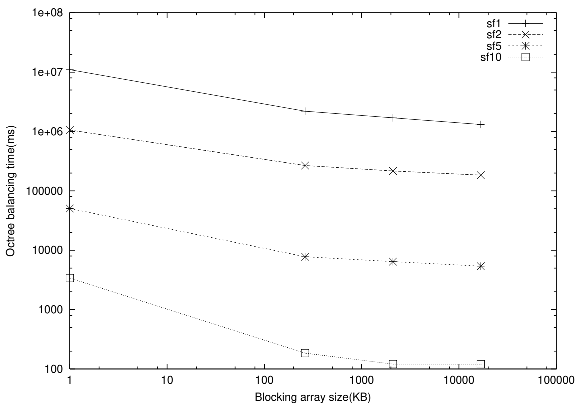

Figure 26: Octree balancing time vs. Etree blocking array size (log-log scale)

global balancing do not incur additional disk I/O. So the larger speed-up factor is mainly due to the faster array-based neighbor-finding algorithm, which avoids the costly operations of searching the B-tree.

On the other hand, the size of the SF1 mesh etree far exceeds the etree buffer size. As a result, multiple traversals of global balancing requires a substantial amount of repeated disk accesses, and thus becomes the bottleneck. Local balancing achieves its speed-up mainly by traversing the domain once, with the array-based neighbor-finding algorithm playing a secondary but still important role. In fact, for the SF1 mesh, the global balancing method traverses the domain four times. But we get a speed-up of 8 by enabling blocking array. The extra performance gain can be attributed to the array-based neighbor-finding.

6. CONCLUSION

We focus on how to generate large octree meshes out of core. Our solution is the etree, a database-oriented method that enables an application to generate meshes by querying a database. We introduce two new techniques, *auto-navigation* and *local balancing*, for fast construction and balancing of an out-of-core octree. The main result from our experiments is that the etree method can generate large octree meshes on memory-limited machines in a reasonable amount of time.

In sum, we have demonstrated that incorporating existing database techniques (linear octree and B-tree) with new algorithms (auto-navigation and local balancing) in a unified design scheme (the etree method) can deliver new capabilities (generating large octree meshes desktop machines).

In addition to the work described in this paper, we have also developed database methods for octree mesh finite element solvers and visualization services. For each timestep, the octree mesh solver iterates through the octree mesh, retrieves the displacement values for an element from the database produced in the previous timestep, solves the element-wise linear equations, and then updates the displacement values in the database for the current timestep. All intermediary results and final results for the simulation are stored and retrieved directly from databases. Similarly, the visualization service queries the result database to service requests for images.

Interesting questions have surfaced as we investigate such database-oriented methods. For example, what strategy should the solver adopt to iterate through the octree mesh such that the locality of reference can be best exploited? Similarly, how should we order a series of queries from the visualization service to better use the content already cached? Another example is how to automatically compress the (intermediary) result databases where spatial or temporal similarity is discovered? All these questions need to be studied and answered before we can fully support running large-scale physical simulations on desktop machines.

ACKNOWLEDGEMENTS

We gratefully acknowledge Jacobo Bielak, Omar Ghattas and Eui Joong Kim for developing the octree-based finite element method and for using our large octree meshes of the San Fernando Valley in their terascale ground motion simulations at Pittsburgh Supercomputing Center. We also thank Tom Jordan, Phil Maechlin, Kim Olsen, Steve Day, Harold Magistrale, and Karl Kesselman, our colleagues on the Southern California Earthquake Center (SCEC) Community Modeling Environment Project, for their suggestions and encouragement. Finally, thanks to Natassa Ailamaki and Christos Faloutsos for helping us understand spatial databases. This work was sponsored in part by the National Science Foundation under Grants CMS-9980063 and 0122464, and in part by a grant from the Intel Corporation.

References

[1] Bryant R., O'Hallaron D. *Computer Systems: A Programmer's Perspective.* Prentice-Hall, 2003

[2] Samet H. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS.* Addison-Wesley Publishing Company, 1990

[3] Shephard M.S., Georges M.K. "Automatic Three-Dimensional Mesh Generation by the Finite Octree Technique." *International Journal for Numerical Methods in Engieering*, vol. 32, 1991

[4] Bern M., Eppstein D., Gilbert J. “Provably Good Mesh Generation.” *Proceedings of 31st Symposium on Foundation of Computer Science*, pp. 231–241. 1990

[5] 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

[6] Wang J. *Octree-Based Finite Element Method for Elastic Wave Propagation with Application to Earthquake Ground Motion*. Master’s thesis, Department of Civil and Environmental Engineering,Carnegie Mellon University, May 1999

[7] Kim E.J. *Terascale Ground Motion Simulation Using Octree-Based Adaptive Mesh*. Ph.D. thesis, Dept of Civil and Environmental Engineering, Carnegie Mellon University, Jan. 2003

[8] Salmon J., Warren M.S. “Parallel, out-of-core methods for N-body simulation.” *Proceedings of the Eighth SIAM Conference on Parallel Processings for Scientific Computing*. 1997

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

[10] Morton G.M. “A computer oriented geodetic data base and a new technique in file sequencing.” Tech. rep., IBM, Ottawa, Canada, 1966

[11] Orenstein J.A., Merrett T.H. “A Class of Data Structure for Associative Searching.” *Proceedings of ACM SIGACT-SIGMOD*, pp. 181–190. Waterloo,Ontario,Canada, 1984

[12] Orenstein J.A. “Spatial Query Processing in an Object-Oriented Database System.” *Proceedings of ACM SIGMOD*, pp. 326–336. Washington D.C, 1986

[13] 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

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

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

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

[17] Silberschatz A., Korth H.F., Sudarshan S. *Database system concepts*, chap. 11. McGrill Hill Companies, Inc., third edn., 1997

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

[19] Bao H., Bielak J., Ghattas O., Kallivokas L., O’Hallaron D., Shewchunk J., Xu J. “Large-scale Simulation of Elastic Wave Propagation in Heterogeneous Media on Parallel Computers.” *Computer Methods in Applied Mechanics and Engineering*, 1998

[20] 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