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. Our main idea is to map an octree to a database structure and perform all octree operations by querying the database. We use two database techniques — the linear quadtree and the B-tree to index and store the octants on disk. We introduce two two new techniques — auto navigation and local balancing to address the special need of mesh generation. Preliminary evaluation suggests that the etree method is an effective way of generating very large octree meshes (4.3 GB with 13.6 million elements) on a memory-limited machine (128 MB).

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

1. INTRODUCTION

This is an exciting moment for the mesh generation community. 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 tens of 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 $1/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 over 1 GHz the norm, and 2 GHz machines available.

These technology trends suggest a good opportunity to exploit out-of-core techniques for mesh generation. Faster CPUs provide the computational throughput to create and refine the mesh. Exploding disk capacities make it possible to store large meshes on conventional disks. Larger main memory capacities can provide effective caches for the data stored on disk. And increased I/O throughput makes it possible to stream large amounts of data into memory at rates that approach main memory transfer rates.

However, it is difficult to implement an out-of-core approach due to the irregularity nature of most meshes. Fortunately, the simplicity and adaptability of the octree structure gives us some hope that we can efficiently store and retrieve octree mesh stored on disk.

In this paper, we present the *etree*, a database-oriented method for generating large out-of-core octree meshes. The key idea is to extend database techniques to support mesh generation. We use the *linear quadtree* technique to assign to each octant a unique key that encodes its location and size, and store the octants in a well-known database structure — the *B-tree*. In order to address the special need of octree mesh generation, we develop two new techniques — *auto navigation* and *local balancing* to support octree-level oper-

ations. All these components are implemented in the *etree library*. An application can invoke etree library function calls to manipulate an octree mesh stored on disk. The etree library automatically performs extensive optimization to improve the running time and reduce the 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 physical memory.

Section 2 briefly discuss two different ways of generating octree meshes. Section 3 describe 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. Section 6 concludes our work.

2. OCTREE MESH GENERATION

An octree algorithm [2] recursively subdivides a problem domain into eight equal size octants until certain resolution level is achieved. Figure 1 is an octree decomposition of a two-dimensional domain. (For convenience, we use octree and octant to refer to quadtree and quadrant, respectively in this paper.)

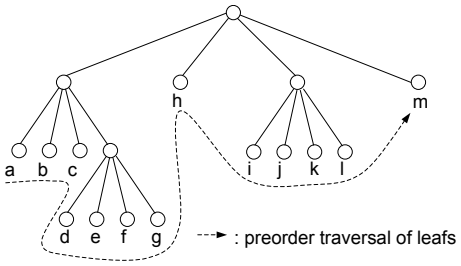

Figure 1: Quadtree decomposition of a domain.

In the area of mesh generation, octree decomposition has been a successful strategy for generating three-dimensional unstructured meshes. In practice, there are two different ways of using the octree data structure. One way is to warp the leaf octants to obtain tetrahedral elements [3, 4]. The other way is to use the leaf octants as finite elements directly without further modification [5, 6]. 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*). In this paper, we focus on how to generate a mesh that uses the leaf octants directly as elements, which we will refer to as an *octree mesh*.

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 the *virtual memory*, an operating system facility that enables an application to allocate much more memory than what's physically available. The virtual memory uses the physical memory efficiently by treating it as a cache for an address space stored on disk, keeping only the active areas in physical memory. This mechanism works silently and automatically, without any intervention from the application program. However, if the allocated virtual memory far exceeds the physical memory and the data accesses are not localized, severe swapping of data between the disk and the memory will occur, which, in turn, will hurt the performance of the in-core algorithm significantly. Therefore, an in-core algorithm is practically memory-bound.

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 disk-bound instead of memory-bound. 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 memory.

One candidate solution is to use the *out-of-core pointer* method [7], that is, each in-core pointer is mapped to an out-of-core pointer in the form of $\langle$*disk page number, offset*$\rangle$. The operation of following a pointer in an in-core algorithm is transformed to seeking an offset on a disk page and retrieve the data object. The application programs should make arrangement 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 is to leverage and extend database techniques to address the special need of mesh generation. As a result, applications can manipulate an octree mesh by simply querying the database instead of chasing the pointers.

Figure 2 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 nodes' coordi-

nates are derived from the balanced octree, and stored in two databases, one for the mesh elements, and the other for the mesh nodes.

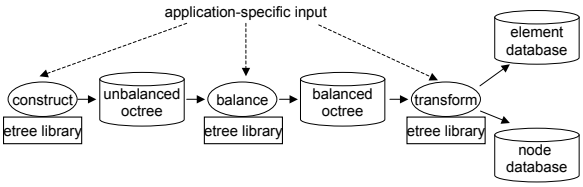

Figure 2: Etree method of generating octree meshes.

Conceptually, such a process can be implemented with 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 internal 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, we observe that the such a direct database approach introduces trade-offs. 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, has to be maintained to keep track of which octants have been processed and which have not. This adds complexity to the application program.

In order to avoid the negative aspect, we use the database technique as one of the building blocks in the etree, and extend the core database functionality by introducing new techniques that support octree-level operations.

4. THE ETREE LIBRARY

The technical details of the etree method can be best explained by an anatomy of its implementation — the *etree library*. Figure 3 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: 1. *linear quadtree*, a powerful encoding scheme to assign keys to octants; 2. *auto navigation*, a mechanism for traversing the octree automatically; 3. *local balancing*: a technique that speeds up octree balancing operations; and 4. *B-tree*, a database index structure to store and access octants on disk.

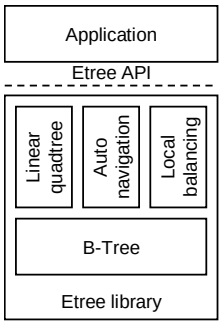

Figure 3: Etree architecture.

4.1 The etree API

The small API of the etree library is shown in Figure 4. The API is inspired by Unix file I/O and can be categorized into three classes.

- 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 2. The only input from the application is a function that the etree library can use to determine how to process an octant.

This API allows users to manipulate octree meshes at different levels. It also hides the complexities of the etree internal mechanisms and optimizations.

4.2 Linear quadtree

In order to map an octree to a database structure, we use the *linear quadtree* [8] technique, a method that assigns a unique key to each leaf octant. The key encodes the location and size information of the octant. We can use a database structure such as the B-tree (see Section 4.5) to index the keys.

```
/* 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 4: Etree API.

One of the well-known encoding schemes for linear quadtree is the *Morton code*. The Morton code [9] maps $d-$dimensional points to one-dimensional scalars. The mapping can be done quickly by interleaving the bits of a point's coordinate (the binary representation).

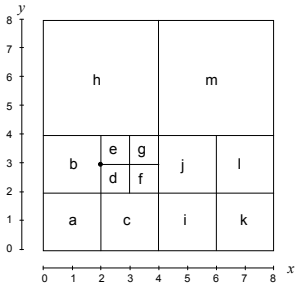

Figure 5: A domain decomposition.

We adopt this scheme and transform the left-lower corner of each octant to the Morton code. To distinguish an octant from its ancestors that share the same left-lower corner, we append the level of an octant to the Morton code of its left-lower corner. The result is a unique key assignment for each octant, which is usually referred to as the *locational code* [2]. An example[1] is shown in Figure 6.

[1] The underscore in the locational code is for illustration purpose only. A locational code is just a binary string.

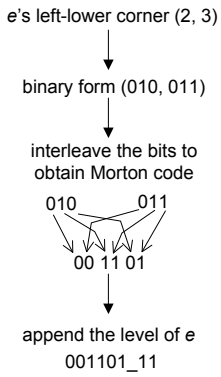

Figure 6: Interleaving bits and appending level to obtain the locational code.

To isolate 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 coordinate 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`.

One 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 (leafs). Figure 8 shows the ordering based on locational codes for the octree in Figure 8. There are two 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.

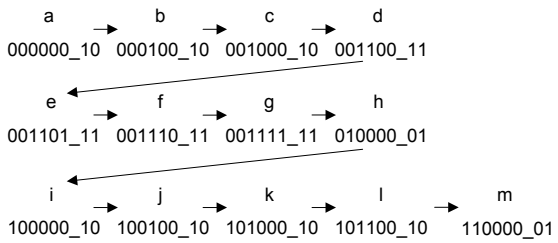

Figure 8: Leaf octants sorted according to their locational code.

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* [10, 11], which tends to cluster spatially close data points in their one-dimensional ordering [12]. 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.

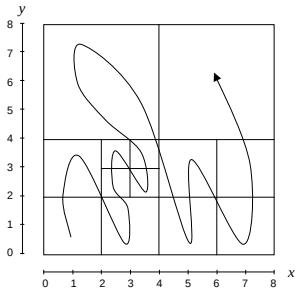

Figure 9: Z-order curve through the domain.

The second application entails a more subtle explanation. We will illustrate our point by the following example. To find the neighbor on the south side of octant e in Figure 5, 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).

This occurrence is not accidental. Because a subtree root is always the first one among all the octants of the subtree in the preorder traversal. Even when the subtree (except for the subtree root) does not really exist, this relationship still holds. Therefore, by returning the octant whose key is the maximum among all the keys that are less than the search key, we can always find the subtree root for the search key. Thus, we are able find an octant, such as the neighbor octant in our example, without knowing its exact locational code.

4.3 Auto navigation

The linear quadtree only solves the problem of how to address individual octants, but does not provide a programming model for octree construction. Although it is possible to construct an octree by repeatedly inserting and deleting octants from the database, the application programs, however, have to keep record of which octants have been decomposed and which have not. On the other hand, many insertions are in fact unnecessary because those octants are later decomposed and removed from the database.

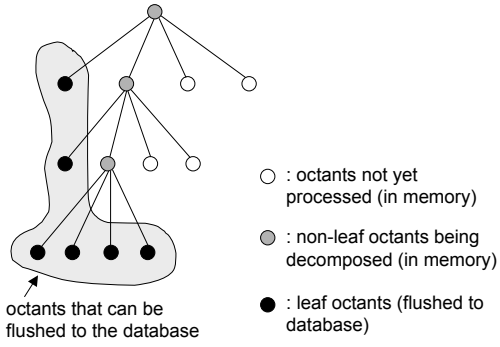

Figure 10: Auto navigation through an octree being constructed.

To solve this problem, the etree provides a higher-level abstraction to support automatic octree construction through the function call `etree_construct`. The underlying technique is what we called *auto navigation*. The basic idea of auto navigation is very 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'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. The application provides a function to guide the operation of the navigation octree.

Initially, the navigation octree only has the root octant. The application function 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 carried on in the depth-first order. At any moment when an octant does not need to be further decomposed, it is certainly a leaf octant and is pruned off from the navigation octree and flushed to the database. With the 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, the applications are relieved

from the burden of traversing an out-of-core octree for expansion, which is a complicated and error-prone task. On the other hand, database operation 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 $\mathcal{O}(1)$ while an insertion operation involves a search algorithm to locate the right position for the insertion, which may incur additional disk I/O to read in database pages.

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 5 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.

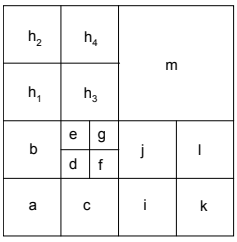

Figure 11: A balanced domain decomposition.

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.

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.

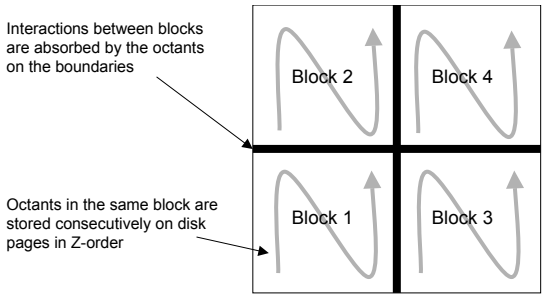

Figure 12: Local balancing.

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

With the first condition, each block is mapped to a subtree root[2]. 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 whole domain once to process all the blocks. 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 5.

After the blocking array is initialized, we can resolve the 2-to-1 constraint within the block without query-

[2]We assume the blocks are aligned properly.

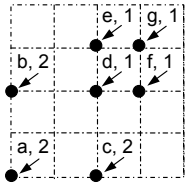

Figure 13: Content of the blocking array.

ing 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: We proceed to iterate through the boundary octant list, and check each octant's neighbors to determine whether the 2-to-1 constraint is violated, and if so, perform a decomposition. In this process, a new list for 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 show the correctness of the local balancing scheme by proving that the interactions between adjacent blocks are always *absorbed* by octants on the boundaries between blocks and will never propagate into the blocks.

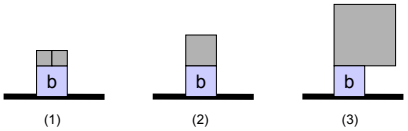

Figure 14: Three cases for internal neighbors of a boundary octant.

Figure 14 shows the three scenarios of the relationship between a boundary octant b and its *internal* neighbors in the same block. The dark line represents the boundary of the block. Since the 2-to-1 constraint is resolved locally, the neighbors of b can only be one of the following three cases: (1) half as large as b; (2) as large as b; or (3) twice as large as b.

However, case (3) is impossible. Because the first condition imposed on the block size implies 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 for octant b to have an internal neighbor that overlaps the region covered by b's parent, as shown in Figure 15.

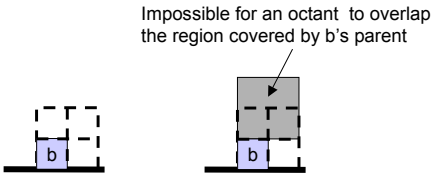

Figure 15: Case(3) is an impossible scenario.

With cases (1) and (2) as the possible initial condition for the first iteration of boundary balancing, we can prove by induction that the internal octants will never be affected by the interactions between adjacent blocks.

Figure 16 shows that with case (1) as the initial condition, the decomposition of b will result in four children octants of equal size to b's internal neighbors. So there is no further impact on the internal octants. 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. Thus case (2) is the new initial condition.

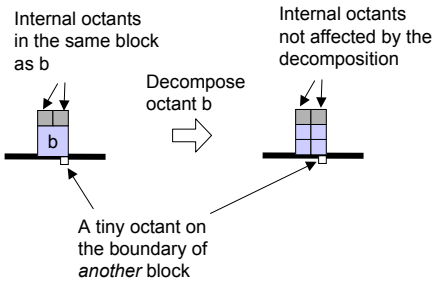

Figure 16: Case (1): internal neighbors not affected by the decomposition of the boundary octant.

Figure 17 shows that with case (2) as the initial condition, the decomposition of b will result in four children octants half as large as b's internal neighbor. But the 2-to-1 constraint is still maintained. So the impact is not propagated into the block either. Again, the new boundary octants have internal neighbors of the same size, which is case (2).

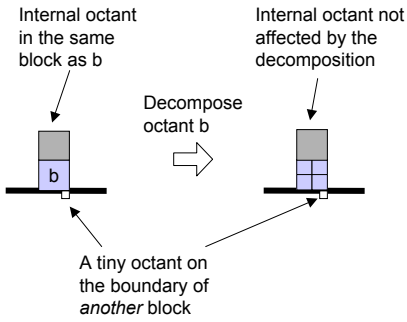

Figure 17: Case (2): internal neighbors not affected by the decomposition of the boundary octant.

Therefore, the impact from adjacent blocks is always absorbed by the (dynamically changing) boundary octants.

4.5 B-tree

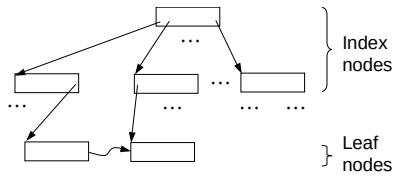

Figure 18: B-tree structure.

B-tree is the most important index structure in database and file systems [13, 14, 15, 16]. Figure 18 shows the structure of a B-tree. There are two types of nodes in a B-tree: the *leaf nodes* and the *index nodes*.

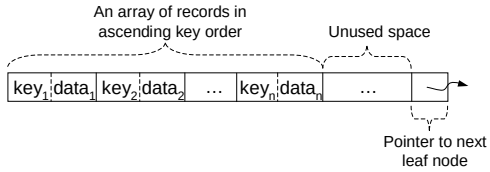

Figure 19: B-tree leaf node.

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.

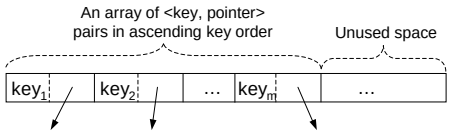

Figure 20: B-tree index node.

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 $K_i \leq K_x < K_{i+1}$.

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% [17].

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 [18].

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 [19]. 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.

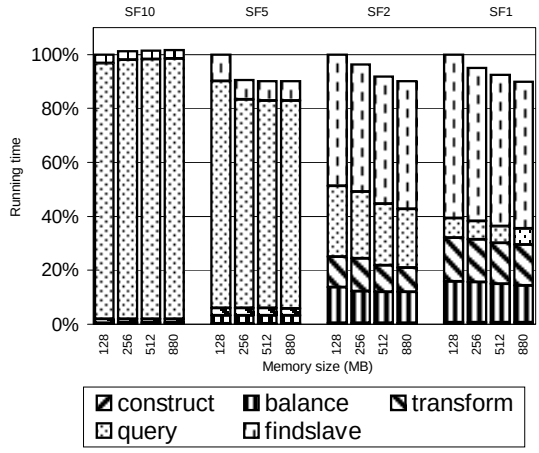

Figure 23: Total running time vs. Physical memory size.

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 operations 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.

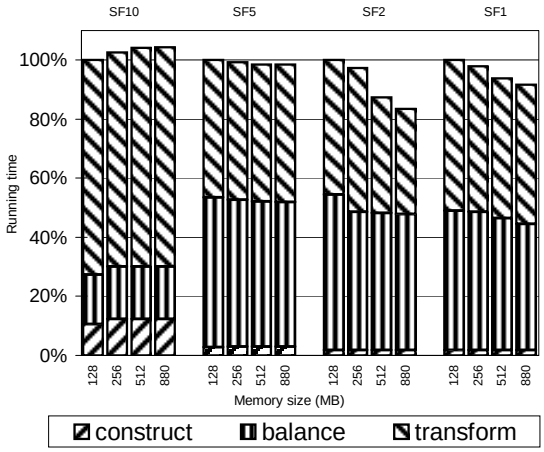

Figure 24: Etree operation running time vs. Physical memory size.

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 etree 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.

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

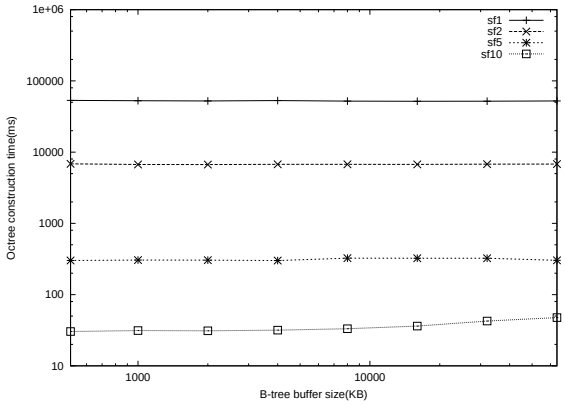

Figure 25: Etree construction time vs. B-tree buffer size (log-log scale)

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).

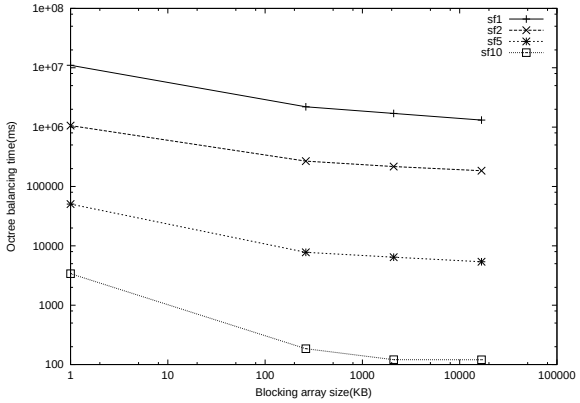

Figure 26: Octree balancing time vs. Etree blocking array size (log-log scale)

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 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 very large octree meshes in a reasonable amount of time with a low memory requirement.

ACKNOWLEDGEMENTS

We gratefully acknowledge the contributions of Jacobo Bielak, Omar Ghattas and Eui Joong Kim in the development of the etree method. This work is sponsored in part by the National Science Foundation under Grant CMS-9980063, 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, 2002

[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, Computational Mechanics Laboratory, Department of Civil and Environmental Engineering,Carnegie Mellon University, May 1999

[7] 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

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

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

[10] 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

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

[12] 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

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

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

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

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

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

[18] 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

[19] 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