Balance Refinement of Massive Linear Octree Datasets

Tiankai Tu and David R. O'Hallaron

April, 2004
CMU-CS-04-129

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many applications that use octrees require that the octree decomposition be smooth throughout the domain with no sharp change in size between spatially adjacent octants, thus impose a so-called 2-to-1 constraint on the octree datasets. The process of enforcing the 2-to-1 constraint on an existing octree dataset is called balance refinement. Although it is relatively easy to conduct balance refinement on memory-resident octree datasets, it represents a major challenge when massive linear octree datasets are involved. Different from other massive data problems, the balance refinement problem is characterized not only by the sheer volume of data, but also by the intricacy of the 2-to-1 constraint. Our solution consists of two major algorithms: balance by parts and prioritized ripple propagation. The key idea is to bulk load most of the data into memory only once and enforce the 2-to-1 constraint locally using sophisticated data structure built on the fly. The software package we developed has successfully balanced world-record linear octree datasets that are used by real-world supercomputing applications.

This work is sponsored in part by the National Science Foundation under Grant CMS-9980063, in part by a subcontract from Southern California Earthquake Center as part of NSF ITR EAR-01-22464, and in part by a grant from the Intel Corporation. Experiments in Section 10.3 and 10.4 were performed on the National Science Foundation HP GS1280 system at the Pittsburgh Supercomputing Center.

Keywords: linear octree, balance refinement, balance by parts, prioritized ripple propagation

1 Introduction

The extensive applications of the octree data structure can be dated back to as early as the 1970's [14]. Given three decades of research, it is often considered that octrees have been fully studied. Unfortunately, an indispensable operation required by many applications called *balance refinement* has somehow been largely ignored in the past.

The purpose of balance refinement is to enforce a continuity condition on an existing octree so that the octree decomposition becomes relatively smooth throughout the domain and there is no sharp change in size between spatially adjacent leaf octants. Although rediscovered and renamed many times by researchers in different fields, the continuity condition bears the same characteristics in all applications: no two leaf octants that share a face or an edge should differ by a factor of more than 2 in terms of their edge sizes. Or equivalently, all spatially adjacent leaf octants that share a face or an edge should differ by at most 1 in their tree levels. To make the term more intuitive, we refer to the continuity condition as the *2-to-1 constraint.*

Important applications that require the 2-to-1 constraint on octrees include scientific computing [4, 19, 11, 2], quality mesh generation [18, 7, 12, 15], and computer graphics [3]. Even though many authors have based their work on the critical condition that an octree be balanced, it has often been conveniently assumed there exists some balance refinement algorithm, for example, to facilitate further theoretical analysis. As a result, the question of how to efficiently balance an octree is left unanswered. Although this may not be a big problem for small applications where the dataset can be completely cached in main memory, it does represent a serious problem when massive linear octree datasets are involved, esp. in scientific computing area [19, 2]. Usually, in order to simulate large and complex physical phenomena, scientific applications require billions of octants or even more to model the domain of interest. The sizes of these datasets are commonly in the order of tens of gigabytes, with terabyte datasets on the horizon. How to efficiently balance such massive linear octree datasets that cannot be completely cached in main memory constitutes a major challenge.

Interestingly, the problem of balance refinement of massive linear octree datasets falls in-between the two canonical categories: the *batched problems* and the *online problems* [17]. It is a batched problem because every item (octant) in the dataset has to be processed in order to enforce or verify the 2-to-1 constraint. It is an online problem because changes to the dataset (linear octree) are only performed in response to violations of the continuity condition, and are mostly confined to a small portion of the dataset. Extending the taxonomy defined by Vitter [17], we refer to the balance refinement problem and the like as *hybrid problems.*

This paper presents an efficient and scalable balance refinement solution. Like many other database algorithms, our solution exploits locality of reference to reduce disk I/O. The main algorithm is called *balance by parts.* The key idea is to divide the domain represented by an linear octree into 3D volumes (*volume parts*) that can completely fit in main memory. Each volume is streamed into main memory by a sequential scan and is cached in a temporary pointer-based octree called *cache octree.* Interactions between volumes are resolved by balancing octants on the inter-volume boundaries (*boundary parts*). Octants of a boundary part are fetched by range queries issued on the linear octree and are also stored in a temporary cache octree. Once a cache octree is initialized, we apply an algorithm called *prioritized ripple propagation* to balance it efficiently. While adjusting the structure of the cache octree, we update the linear octree dataset accordingly. Evaluation results show that our solution is both efficient and scalable. It runs 3 times faster than existing algorithms when used to balance a 20GB dataset with 1.2 billion octants on a Linux workstation with 3GB main memory. It also keeps high throughput rate when used to balance extremely large dataset (56GB).

Our paper makes the following contributions:

- *Real-world impact:* We show that besides the conventional query-based services, database techniques, when combined with new algorithms, can deliver unprecedented capability to support large-scale scientific applications. The software package we developed is used in practice to solve problems previously undoable.

- *Database design principle:* We introduce a unified design framework to simultaneously address the problems of reducing disk I/O time and improving the running time of the operation (balance refinement) itself. Our new algorithms are provably efficient and easy to implement.

Section 2 introduces the basic concepts related to octrees and balance refinement. Section 3 briefly surveys the related work. Section 4 defines the performance problems we target to resolve. Section 5 is an overview of the balance by part algorithm. Section 6 proves its correct. Section 7 explains how to retrieve data efficiently. Section 8 illustrates how to build a temporary internal data structure to cache the data. Section 9 presents an algorithm that efficiently balance the cached data. Section 10 evaluates the performance of our solution. Section 11 concludes our work.

2 Background

2.1 Octrees and Linear Octrees

An *octree* recursively subdivides a three-dimensional domain into eight equal size *octants* until certain criteria are satisfied [14]. One common way to represent an octree is to link the tree nodes using pointers. Figure 1 shows the pointer-based octree[1] representation and its corresponding the domain decomposition. A node with no children is a *leaf node.* Otherwise, it is a *non-leaf node.* The number of hops from a node to the root node defines the *level* of the node. The larger the value, the lower the level. The corresponding domain decomposition is shown is Figure 2.

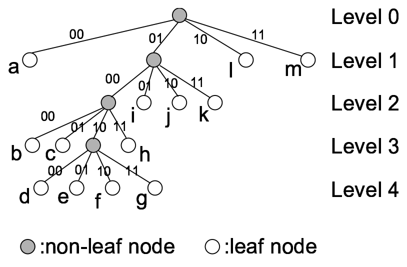

Figure 1: Pointer-based octree representation.

[1]We may draw 2D quadtrees in figures to illustrate concepts. But we use the term "octrees" and "octants" consistently, regardless of the dimension.

The other common way to represent an octree is the *linear octree* [10, 1]. The linear octree technique assigns a unique key to each node and *represents an octree as a collection of leaf node comprising it*. With keys assigned, leaf nodes can be organized on disk by an index structure such as a B-tree [8, 6]. Thus, the sizes of linear octrees are disk bound rather than memory bound. They are extremely useful in practice where main memory cannot accommodate an pointer-based octree. In this paper, we only consider linear octree datasets indexed by B-trees.

The key assigned to a node is generally referred to as its *locational code*, which encodes the location and size of the node. One particular locational code that is commonly used is obtained by concatenating the branches (bit-patterns) on the path from the root node to the leaf node. Zeroes may be padded to make all the locational codes of equal length. To distinguish the trailing zeroes of branch bit-patterns from zero paddings, the level of the node is attached to the path information. An equivalent way [9] to derive the locational code is based on bit-shuffling of the coordinates, which is often used in practice.

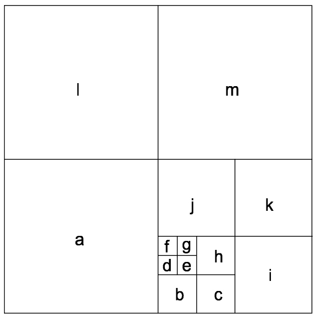

Figure 2: The corresponding domain decomposition.

For example, assume the maximum tree level supported is 4, then the locational code of node `a` in Figure 1 is 00000000_001_2. The underscore is for illustration purpose only; a locational code is just a fixed-length bit string. Note that the last six zeroes in the path information (before the underscore) are paddings added to make the code of equal length as others. Similarly, we can derive the locational code of node `f` as 01001010_100_2. Since `f` is already at the lowest level, no paddings are necessary. Obviously, when given a locational code, we can easily identify a node in a pointer-based octree by descending from the root node according to the path and level information encoded.

When we sort the leaf nodes according to their locational codes (treated as binary scalar value), the order we obtain is the same as the preorder traversal of the octree. Therefore when we index a linear octree in a B-tree, octants are stored sequentially on B-tree pages in the order of preorder traversal[2].

2.2 2-to-1 constraint and Ripple Effect

The 2-to-1 constraint requires that the edge size of two leaf octants sharing a face or an edge should be no more than twice as large or small. For example, octant `f` in Figure 2 is adjacent to octant `a` and `j`, both of which are more than twice as large. Figure 3 shows the result of refining the domain to a balanced form.

[2]In the context of disk-resident linear octree, we refer to the leaf nodes simply as octants, since no non-leaf nodes are stored on disk

The corresponding tree structure adjustment is shown in Figure 4. Note that in 2D, we only need to consider edge-neighbors.

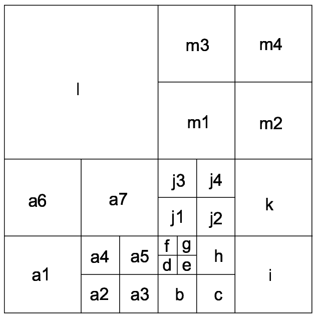

Figure 3: A balanced domain decomposition.

One interesting property of the refinement process is the so-called *ripple effect*. That is, a tiny octant may propagate its impact out in the form of a "ripple", causing subdivisions of octants not immediately adjacent to it. In our example, octant `m` is not directly adjacent to octant `f`, but it is forced to subdivide by the subdivision of octant `j`, which is triggered directly by `f`.

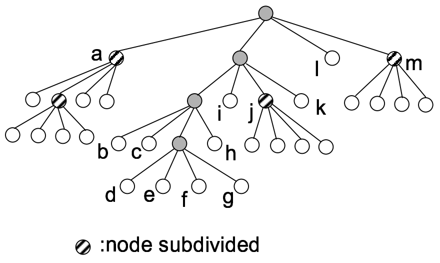

Figure 4: A balanced octree resulted from a series of subdivisions.

3 Related Work

Moore studied the space cost of balancing generalized quadtrees and proved that there is a *unique* least common balance refinement of an octree [13]. In other words, if we balance an octree with as few subdivisions as possible, then whichever valid[3] order we take to conduct the subdivisions, we obtain the same unique balanced octree. This theorem enables a greedy algorithm (such as ours) to subdivide too-large octants immediately and guarantees that the final result is correct.

Yerry and Shepard pioneered the work of automatic 3D mesh generation based on octrees. They introduced a balance algorithm on a breath-first expanded octree in [18]. This algorithm proceeds in two steps. In the first step, an unbalanced octree is traversed in breath-first order. On visiting a leaf octant, a series of neighbor-findings are performed to determine whether it is more than twice as large as any of its (leaf) neighbors. If so, the leaf octant is added to a list of subdivision. In the second step, each leaf octant in the

[3]We cannot subdivide a child before subdividing its parent.

list of subdivision is subdivided and its eight children inserted into the octree. Because any octant can be subdivided at most once in one iteration of the two-step procedure, multiple iterations may be invoked to resolve the ripple effect.

More recently, we proposed an algorithm called *local balancing* for the balance refinement of large linear octrees [15]. The idea is to partition the domain modeled by an octree into equal-size blocks and then traverse the domain block by block. Each block is cached in memory in a *blocking array*. A variant of the Yerry and Shepard's algorithm is invoked to balance each block. Interactions between adjacent blocks are handled by a post-processing step called *boundary balancing*, which may take multiple iterations. After each iteration, a new list of boundary octants is generated and taken as input for the next iteration.

4 Problem Statement

The balance refinement process basically involves two operations: (1) *neighbor-finding*, finding neighbors to obtain their edge sizes information in order to make comparison; (2)*subdivision*, delete a "too-large" octant from the dataset and insert its eight children. The deletion is necessary in the subdivision operation because the linear octree datasets we are balancing should contain only leaf octants.

Suppose we have a complete list that records the octants that need to be subdivided, then the process of balance refinement boils down to a sequence of simple B-tree deletions and insertions. Unfortunately, we do not a list of subdivision when given an unbalanced linear octree dataset. Worse, the ripple effect excludes the existence of such a complete list, which should grow gradually during the refinement process. So the only way to decide whether an octant is too-large and needs to be subdivided is by comparing its edge size with those of its neighbors. Therefore, neighbor-finding is the key operation for balance refinement.

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

Another method is to map the linear octree to an incore pointer-based octree and use the conventional pointer-based algorithm to find neighbors [14]. The advantage is that the average cost of neighbor-finding is reduced to $O(1)$, with a total cost of only $O(N)$ to conduct *all* the neighbor-findings. But the main memory should be large enough to build a pointer-based octree image for the linear octree.

How can we take advantage of both methods? That is, *how can we find neighbors efficiently (in $O(1)$ time) without excessive memory requirement (not mapping the entire linear octree in memory)*? This is the first problem we need to address.

The second performance problem is more subtle and is related to the ripple effect. After an octant is checked to be balanced with respect to its neighbors, one of its neighbors may be subdivided later and become smaller. This may cause the original octant to become "too-large". Consequently, another round of neighbor-findings must be invoked to discover this newly created unbalanced situation. However, multiple iterations of neighbor-findings increase the total running time by a constant factor. So the second problem we focus on is *how to avoid multiple iterations of neighbor-findings*?

5 Balance by Parts

Our solution is based on an observation that although balance refinement may cause ripple effect, the impact diminish quickly due to the 2-to-1 edge size ratio. In addition, most impact caused by a tiny octant is localized in a small region. For example, octant `f` in Figure 2 causes the subdivisions of octant `a` and its children. But both are spatially adjacent to `f`. In other words, the impact of a tiny octant is absorbed mostly by octants surrounding it in a small neighborhood.

The strong locality of reference suggests that we may map a small region to a pointer-based (sub)octree in memory and resolve the 2-to-1 constraint and the ripple effect without worrying about octants outside of the region. This is the type of solution that fits the paradigm of divide-and-conquer perfectly.

5.1 Overview

Figure 5 shows the outline of our main algorithm, called *balance by parts*. First the domain represented by a linear octree is partitioned (divided) into equal-sized 3D volumes called *volume parts*. The size and alignment of each 3D volume should correspond to some non-leaf node (of a conceptual pointer-based octree) at certain level. Next, each volume is cached in memory and balanced. After all the 3D volumes are processed, octants on the volume face boundaries, called *face boundary parts*, are balanced , followed by the balance of octants on the volume line boundaries (*line boundary parts*) and point boundaries (*point boundary parts*). Each part, regardless of its type, is cached in a temporary pointer-based octree called *cache octree*. While balancing an cache octree in memory, we update the B-tree to record the subdivisions of leaf octants. An algorithm called *prioritize ripple propagation* is used to efficiently balance cache octrees (see Section 9).

An intuitive way to understand the balance by parts algorithm is to imagine a moving window inside the 3D domain. At any moment, the content (octants) inside this window is retrieved from disk and cached in a temporary data structure (cache octree). When we adjust the data structure to enforce 2-to-1 constraint in memory, the content on disk is updated accordingly (by deleting subdivided octants and inserting their children in the B-tree). The window size is set differently for four separate stages, ranging from the largest (for the 3D volumes) to the smallest (for the point boundary parts).

Although similar in principle of divide-and-conquer, this solution is different from our previous work [15] in many key aspects. First, instead of caching data in a flat structure (blocking array), we install octants in a temporary pointer-based octree. Second, no additional iterations of boundary post-processing are necessary. Interactions between 3D volumes gradually diminish after being assimilated by face boundaries, then line boundaries and finally point boundaries. Third, we apply a same routine to balance all the parts. No special treatment for the boundary octants is needed. Fourth, we have developed a new algorithm that can balance a pointer-based octree efficiently (O(n)), rather than using a variant of Yerry and Shepard's algorithm.

5.2 An Example

Here is an example of applying our on the octree of Figure 1 and Figure 2. Note that in the context of linear octrees, the *global* pointer-based tree structure does not exist physically in memory or on disk.

Algorithm 1 (Balance by parts).

Input:
An unbalanced linear octree indexed/stored in a B-tree.

Output:
A balanced linear octree index/stored in the same B-tree.

Method:
Organize the dataset as smaller, memory cacheable parts, and balance each part independently.

Step 1: [Partition the domain into equal-sized 3D volumes.]
Based on the available memory size, decide the maximum number of octants that can be cached and calculate the corresponding subtree root level. Each 3D volume maps to a subtree root.

Step 2: [Balance the 3D volume parts.]
Fetch data from database: Octants belonging to each 3D volume is sequentially scanned from B-tree pages and cached in an internal temporary data structure called *cache octree*.

Balance cache octree: Each cache octree is balanced independently. Subdivisions of leaf nodes causes octants to be deleted from and inserted into the B-tree.

Release cache octree: After an cache octree is balanced and the B-tree updated accordingly, release memory used by the cache octree.

Step 3: [Balance the face boundary parts.]
Similar to Step 2 except that *range queries* are issued to fetch octants on the *face boundary* between adjacent 3D volumes for each part.

Step 4: [Balance the line boundary parts.]
Similar to Step 2 except that range queries are issued to fetch octants on the *line boundaries* of adjacent 3D volumes for each part.

Step 5: [Balance the point boundary parts.]
Similar to Step 2 except that range queries are issued to fetch octants on the *point boundaries* of adjacent 3D volumes for each part.

Figure 5: Balance by parts.

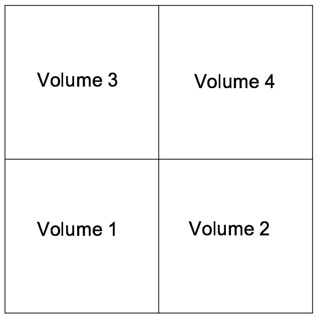

Figure 6: Partition the domain into 4 volumes corresponding to tree level 1.

Assume the largest volumes that can fit in memory corresponds to non-leaf nodes (of the conceptual quadtree) at level 1, we partition the domain into 4 volumes, shown conceptually in Figure 6. Each volume is cached in memory independently as a temporary pointer-based cache octree and then balanced. Figure 7 shows the cache octree for volume 2.

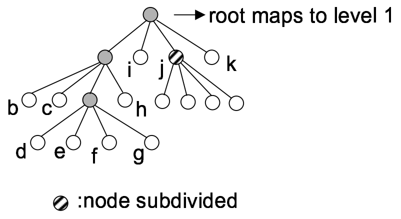

Figure 7: The balanced cache octree representing volume 2.

After all the volumes are processed, line boundary parts are balanced one by one, in arbitrary order. Note that for 2D cases, there are only line boundaries and point boundaries. Figure 8 show the cache octree representing octants on the boundaries between volume 1 and volume 2 and the corresponding region is shown in Figure 9. Note that all the subdivisions triggered by octant `f` are confined on the boundary. Also, the cache octree root for boundary parts is mapped to the entire domain (level 0) instead of the subtree root level as does the cache octree for the volumes. As a result, the cache octree has null branches. We will justify these design decisions in the remainder of this paper.

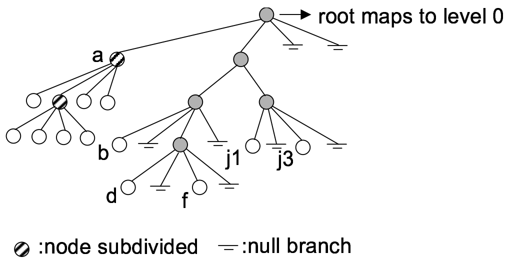

Figure 8: The balanced cache octree representing line boundary part between volume 1 and volume 2.

The point boundary part consists of the four octants anchors at the center point of the domain (the corner

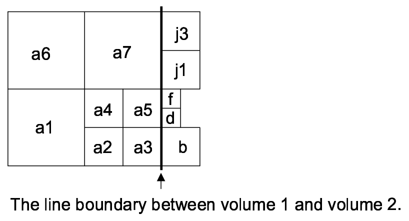

Figure 9: The boundary part between volume 1 and volume 2.

boundary of the 4 volumes). In this example, no subdivision occurs inside the point part. So we skip the cache octree representation for the point boundary part.

5.3 Questions to be Answered

Although structurally simple, our solution may raise many questions. First of all, is it correct? How can a linear octree be balanced when only parts of the tree are being balanced? Second, how to fetch *parts* of different types from a linear octree dataset and what is the I/O implication? Third, how to build the internal temporary cache octree and speed up the balance operation itself?

A short answer is that the algorithm is correct and all the design choices are made to resolve the performance problems stated in Section 4. Detailed answers are presented in the next three sections.

6 Correctness

We first define an important concept called *stable octants*:

Definition 1. *An octant is* stable *if (1) it will not trigger other octants to subdivide; and (2) it will not be triggered to subdivide.*

Stable octants are isolated from other octants in terms of interactions that might trigger subdivisions. While other octants are undergoing subdivisions, stable octants remain intact in the dataset. They exist in a balanced linear octree dataset in the same form as when they become stable. It is trivial to show:

Theorem 1. *An octree is balanced if and only if all its octants are stable.*

So instead of directly proving that each individual octant conforms to the 2-to-1 constraint with respect to its neighbors, we prove the correctness of our algorithm by constructing the set of stable octants $\mathcal{S}$. Initially empty, $\mathcal{S}$ is augmented monotonously every time we balance a part of some type. When the algorithm terminates, $\mathcal{S}$ contains all the octants in the domain. Thus, we have a balanced octree.

To complete the proof by construction, we need to show: (1) which octants become stable *after* a part of some type is balanced; (2) why $\mathcal{S}$ contains all the octants on termination of the algorithm. Both problems can be solved by using the concept of internal octants and boundary octants.

6.1 Internal Octants and Boundary Octants

As shown in Figure 5, our algorithm works on four different types of parts in order: 3D volume, face boundary, line boundary, and point boundary. *After a part of some type is balanced*, we can partition its octants in two disjoint sets: *boundary set* , which contains the octants on boundary of the part (*boundary octants*); and *internal set*, which contains all the remaining octants (*internal octants*).

Obviously, different definitions for “boundary” and “internal” are needed for parts of different types.

Definition 2.

- *In a balanced 3D volume part, an octant is a boundary octant if it is adjacent to some octant outside the 3D volume. Otherwise, it is an internal octant.*
- *In a balanced face boundary part, an octant is a boundary octant if it is on some line boundary shared by adjacent 3D volumes. Otherwise, it is an internal octant.*
- *In a balanced line boundary part, an octant is a boundary octant if it is on a some point (corner) boundary shared by adjacent 3D volumes. Otherwise, it is an internal octant.*
- *In a balanced point boundary part, all octants are internal octants.*

Boundary octants associated with balanced parts of one type correspond to the parts of the next type to be balanced. For example, the boundary octants of balanced 3D volumes are those on volume face boundaries and, by definition (see Section 5.1), form the face boundary parts.

If we perceive our algorithm as consisted of four stages as shown in Figure 10, every stage processes the balance inflow data and separate the result as internal octants and boundary octants. The latter form the inflow data stream to the next stage. The final stage does not produce any boundary octants. So if we can show all internal octants are stable, we are done with the proof.

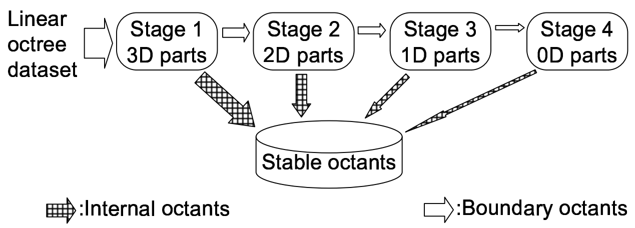

Figure 10: Four stages of balancing parts of different types.

6.2 A Proof Template

The proofs of internal octants being stable in the context of different types (3D volumes, face boundary, line boundary, and point boundary) are identical in their structures. So, without loss of generality, we present, as a template, a detailed proof showing that internal octants of balanced 3D volumes are indeed stable. Other proofs can be adapted from this template easily and are not presented in this paper.

In order to prove *internal octants* of a *balanced* 3D volume are stable, we need to show that they satisfy the two sufficient conditions of Definition 1. The first condition is trivially true. Since all neighbors of an internal octant belong to the same 3D volume, the internal octant will not cause any of them to subdivide any more. Otherwise, the 3D volume must have not been balanced, contradicting the assumption.

The second condition requires a more careful analysis. If an internal octant's neighbors are all internal, the second condition holds because none of its neighbors will trigger it to subdivide (by applying condition 1 on all the neighbors). However, if an internal octant has a boundary neighbor, the boundary neighbor may be triggered to subdivide by some octant outside the 3D volume. The question is will the internal octant be triggered to subdivided by the ripple effect? The answer is no. The proof is built on the next two lemmas.

Lemma 2. *Suppose a 3D volume is balanced, the edge size of a boundary octant is either (1) twice as large, or (2) as large as those of its internal neighbors.*

Proof. When a 3D volume is balanced, there are only three possibilities of edge size ratio between a boundary octant and its internal neighbors: (1) twice as large, (2) as large, or (3) half as large. We now prove that the third possibility does not exist.

Recall that a 3D volume must have the same size and alignment as some non-leaf node (of a conceptually pointer-based octree) at certain level. Thus, every octant inside the 3D volume must be a child of the subtree root corresponding to the 3D volume.

Now suppose a boundary octant is only half as large as one of its internal neighbors, then the internal neighbor is not properly aligned and cannot be a child of the 3D volume subtree root (see Figure 11). Thus the third possibility does not exist. □

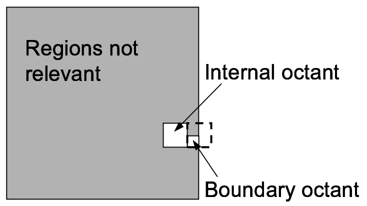

Figure 11: The boundary octant in a balanced 3D volume cannot be half as large as its internal neighbors.

Lemma 3. *Suppose a 3D volume is balanced, if one of its boundary octant is triggered to subdivide by an octant outside the 3D volume, the the 3D volume can be re-balanced without subdividing any internal octants.*

Proof. Due to Lemma 2, if a boundary octant subdivides, its children, half of its edge size, are either (1) as large or (2) half as large as its internal neighbors. Therefore, the 2-to-1 constraint is maintained between its children octants and its internal neighbors.

So if the 3D volume becomes unbalanced, it must have been caused by a violation of the 2-to-1 constraint between some new children octants and other boundary octants. In order to re-balance the 3D volume, these boundaries octants need to be subdivided. Although they may trigger subdivision of more boundary octants, none of the boundary octant subdivision will trigger subdivision of any internal octants, by the same arguments in the previous paragraph. □

Consider the interactions between a tiny octant outside an initially balanced 3D volume. Every time the tiny octant triggers a subdivision of a boundary octant, the 3D volume can be re-balanced without subdividing any internal octants (Lemma 3). This process terminates when the tiny outside octant is adjacent to some boundary octants (descendants of the original boundary octant) that are no more than twice as large. Therefore, the ripple effect of a tiny octant outside of a balanced 3D volume only propagates on the volume face and never gets into the 3D volume.

The application of Lemma 3 is critical in the above reasoning. The claim that internal octants are not subdivided are based on the premise that the 3D volume is balanced. Lemma 3 provides a "self-healing" mechanism to re-balance a 3D volume so that its premise becomes valid repeatedly.

It is worth noting that after a boundary octant is subdivided, its children who are not on the boundary become new internal octants. Thus, when we apply Lemma 3 again, we are actually referring to an expanded internal set. Nevertheless, the original internal octants, which belong to the expanded set, are still not subdivided.

This proves that all internal octants satisfy the second sufficient condition of stable octants. So we have:

Theorem 4. *Internal octants of a balanced 3D volume are stable.*

In a similar way, we can prove that internal octants of balanced parts of other types are stable. So on completion of the four stages of balancing, all the octants in the domain become stable.

This concludes the theoretical proof of the correctness of our algorithm. Next, we explain the systems aspect of our algorithm and show how the performance problems stated in Section 4 are resolved.

7 Data Retrieval

We retrieve data from a linear octree dataset in two different ways: *bulk-loading* and *range query*. 3D volume parts are retrieved by bulk-loading. Because a 3D volume maps to a virtual subtree root whose leaf octants are clustered sequentially on B-tree pages (see Section 2.1). We can identify the position of the *first* octant of a 3D volume in the B-tree by a simple search operation, and then sequentially scan each octant from the B-tree in constant time until we encounter an octant that is outside of the 3D volume. The first octant of a 3D volume is well-defined. It refers to the octant that occurs first in the preorder traversal of the subtree represented by the 3D volume. Since the first octant is always anchored at the left-lower corner of a 3D volume, we can easily derive its locational code.

To retrieve octants for parts of other types, we implement range queries on linear octree datasets. For example, face boundary parts are fetched by searching for octants tangentially intersecting particular rectangles (shared by 3D volumes) in space. Since our solution reduces interactions from face boundaries, to line boundaries and finally to point boundaries, the sizes of range queries are reduced over the stages. In fact, our experiments (see Section 10.3) show that only about 1.5% of a linear octree dataset is fetched by range queries.

In summary, the structural design of our algorithm results in an I/O optimal case where most data is efficiently retrieved by bulk-loading and the remainder is retrieved by standard spatial database range queries.

8 Cache Octree

When the octants for a part (of any type) are retrieved from the linear octree, we cache them in a temporary data structure called *cache octree*. A cache octree is a pointer-based octree with special link lists embedded (see Figure 12). We use this single data structure repeatedly to cache all the parts, regardless of their types. The advantage is that we can apply the same algorithm on all cache octrees. No special treatment of boundary parts is needed. The procedures of building cache octrees for parts of different types are almost identical except for a few minor details.

Before a part is fetched from database, we initialize an cache octree with a single root node[4]. For a 3D volume part, we map this node to the non-leaf node corresponding the 3D volume. For other parts, we map this node to the root node (level 0) that represents the whole domain. We will justify this arrangement shortly. For each octant retrieved, we install it in the cache octree as a leaf node. The installation process is straightforward. As we have shown in Section 2.1, it is trivial to descend from the root node to find a leaf node by extracting the path information (branch bit-patterns) from the its locational code. The only difference here is that we do not have a tree structure in place. So some extra work needs to be done to create non-leaf nodes as necessary when we descend down a cache octree to install a leaf node. Leaf nodes at same tree level are linked together and is accessible from an array called *level table.*

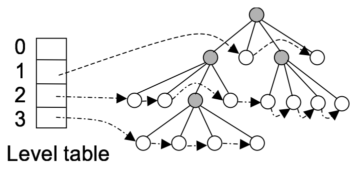

Figure 12: A cache octree is a pointer-based octree with leaf nodes at the same level linked together.

We must guarantee that each octant of a particular part can be properly installed by traversing down a cache octree from its root node. This is not a problem for a 3D volume part since all the octants belong to the same subtree and we have mapped the cache octree root node to that level. For a part with type other than 3D volumes, two octants *may* have different bit-pattern at the first branch in their locational codes. To see an example, check the locational codes of octant `a` and `f` in Figure 2. Therefore, we have to to map a cache octree root node to level 0 to ensure proper installation of all octants of the part. In this case, some branches of non-leaf nodes may be empty (null) and the cache octree becomes *sparse*.

Till this point, we have developed all the techniques needed to resolve the first performance problem stated in Section 4. The solution is to divide the domain into small pieces (balance by parts) and build incore pointer-based octree for each piece (cache octrees). In this way, we can work with small main memory and still take advantage of the faster pointer-based neighbor-finding algorithm.

9 Prioritized Ripple Propagation

The algorithm presented in this section resolves the second performance problem (voiding multiple iterations of neighbor-findings) . The key idea is to (1) decouple node visiting from tree structure traversal; and

[4]We use *nodes* to refer to octants in a cache octree to avoid confusion with octants stored in the linear octree on disk.

Algorithm 2 (Prioritized ripple propagation).

Input:
An unbalanced pointer-based octree.

Output:
A balanced pointer-based octree.

Method:
Visit leaf nodes directly from the level link list and change the tree structure immediately when a too-large (neighbor) leaf node is identified.

```
Step 1:  [Set the current level to the lowest level.]

Step 2:  [Initialize a link list traversal for the current
         level.]

Step 3:  [Apply ripple routine on each node at the current
         level.]
         For each node, search for its neighbors to check
         their sizes. If a neighbor is too large, divide it
         (and its descendants) as many times as needed.

Step 4:  [Set current level to one level up.]

Step 5:  [Goto Step 2 if the current level is more than 1
         below the highest level recorded; Otherwise, ter-
         minate.]
```

Figure 13: Prioritized ripple propagation.

(2) combine neighbor-findings with node subdivisions.

Figure 13 shows the outline of our algorithm named *prioritized ripple propagation* (PRP). It works on the cache octrees. The overall structure is to visit the link lists of leaf nodes at different levels in a prioritized manner. The link list of each level is accessible from the level table associated with an cache octree. We start from the lowest level (with the largest value) and move one level up after processing leaf nodes at each level. The benefit of visiting leaf nodes directly from the link lists is that we can now take an eager approach of subdividing neighbor (leaf) nodes and changing the tree structure on the fly. Had we tied node visiting with tree structure traversal in whatever order, an eager approach would cause great difficulty if a previously visited node were to be subdivided. We would have to interrupt the tree traversal and roll back to the newly subdivided node to check its impact on others.

The key of this algorithm is Step 3 where a *ripple* routine is invoke to implement our eager strategy. The ripple routine combines neighbor-findings with node subdivisions. It is based on the well-known pointer-based neighbor-finding algorithm [14], which consists of two stages: (1) ascending the octree to locate the nearest common ancestor; and (2) descending the octree (on a mirror-reflected path) to find the desired neighbor of equal size or larger.

The ripple routine implements the first stage without modification and record the path traced in a stack. But in the second stage, the ripple routine may subdivide neighbor leaf nodes in order to descend deep enough in the octree. A neighbor leaf node needs to be subdivided if it is more than twice as large as the leaf node we are visiting. Three actions are taken when a neighbor leaf node is subdivided:

1. Allocate eight new children nodes and link them to the subdividing node.
2. Remove the subdividing node from the leaf node link list of its level and add its children leaf nodes to the link list one level lower.
3. Delete the subdividing node from the linear octree and insert its eight children.

The first two actions adjust the incore cache octree to maintain a valid data structure. The third action performs the actual database update to synchronize the image on disk.

After a too-large neighbor leaf node is subdivided, we obtain the next level's branch information from the stack and descend down to one of its newly created children node who now becomes the new neighbor. This subdivide-descend process continues until we reach a level that is 1 above the level of the leaf node we are processing. When the ripple routine is completed, a leaf node is surrounded by neighbors no more than twice as large.

The intuition of the prioritized ripple propagation algorithm is that we should eliminate "problem makers" level by level, starting from the most troublesome level. Thus we first look at the smallest leaf nodes in the domain, and subdivide their neighbors as necessary to ensure that all their impacts are absorbed by their immediate neighbors and they will not *directly* cause any other leaf nodes to subdivide in the future. Then we move up to the next level, which may contain leaf nodes newly created as a result of the previous step. While subdividing neighbors of leaf nodes at this level, we are actually carrying on possible ripple effects originated from lower levels. The "problem maker" elimination process stops when we reach the level that is 1 below the highest level. Leaf nodes in the upper two levels are too big to cause any "problems" (subdivisions of other leaf nodes).

With the PRP algorithm, we avoid multiple iterations of neighbor-findings. The proof of the correctness of the algorithm consists of three parts. First, the algorithm terminates. Since the smallest leaf nodes never subdivide, the total number of leaf nodes to be processed is bounded. Second, a leaf node becomes stable (see Definition 1) after we apply the ripple routine on it (proof by induction). Third, all the leaf nodes are processed by the ripple routine and thus become stable. This is because newly created leaf nodes are always added to link lists at least one level above the current level being processed (due to the 2-to-1 edge size ratio). Given the prioritized level processing order, we are guaranteed to process all newly created leaf nodes.

Since the average running time of pointer-based neighbor-finding algorithm is O(1), the ripple routine runs in O(1) on average. Thus the PRP algorithm has an average cost of O(n), where n is the number of leaf nodes in a cache octree. Since the PRP algorithm is applied repeatedly on all cache octrees, the total cost of running the PRP algorithm is O(N) on average, where N is the total number of octants in the linear octree.

10 Evaluation

In this section, we present the performance evaluation of our balance refinement solution. A series of experiments are conducted to answer the following questions: (1) Is our solution efficient in terms of running time as compared with other algorithms? Referring back to the two performance problems stated in Section 4, what is the impact of performing neighbor-findings using pointer-based cache octrees rather than directly searching the B-tree? And what is the impact of avoiding multiple iterations of neighbor-findings? (2) Is our solution scalable? What is its behavior when used to balance extremely large linear octrees? (3) What is the impact of memory size on performance?

10.1 Methodology

We implemented our balance refinement algorithms using the *etree* library [16], a runtime system for manipulating large linear octrees stored on disk. For convenience, we refer to this program as BBP (balance by parts).

Besides, we implemented an out-of-core version of Yerry and Shepard's algorithm (YS) [18] (see Section 3 for details). Like its incore version, the out-of-core YS algorithm constructs a list of subdivision for each iteration and needs multiple iterations to balance a linear octree. Neighbor-findings are performed by B-tree search operations.

We also developed an improved version of our previous algorithm (IMR) [15]. In particular, we replaced the original post-processing step by balancing the parts on face boundaries, line boundaries and point boundaries. Every part was still balanced by the incore version of the YS algorithm. Two caveats. First, the modification is definitely an improvement because multiple iterations of post-processing are no longer needed. Second, the purpose of this modification is that we can compare the performance of cache octree based balancing algorithm (PRP) and conventional balancing algorithm (YS) directly.

Our experiments were conducted on a collection of real-world massive linear octree datasets. The datasets, after being balanced, were transformed to a set of world-record unstructured hexahedral meshes used by the Quake project (2003 Gordon Bell Award) [5, 2] to assess seismic hazard. In fact, a long-time blocker to high-resolution simulations has been the lack of capability to deal with massive linear octree datasets. Our database solution turned out to be the blocker-remover.

Figure 14 summarizes the characteristics of the linear octree datasets. The columns "Octants (before/after)" record the numbers of octants in the linear octree before and after the balance refinement, respectively. The column "Subdivisions" records the number of subdivisions triggered. The column "Size"reports the sizes of the B-tree files storing the linear octrees after the balance refinement. The unbalanced datasets are about 10% smaller.

Name	Octants(before)	Octants(after)	Subdivisions	Size
la0.5h	9,903,330	9,922,286	2,708	139MB
la1h	113,642,903	113,988,717	49,402	1.6GB
la2h	1,192,888,861	1,224,212,902	4,474,863	20GB
la3h	3,656,944,427	3,734,593,936	11,092,787	56GB

Figure 14: Summary of massive linear octree datasets.

10.2 Is the solution efficient?

This set of experiments were conducted on a Linux 2.4 workstation with PIII 1GHZ processor (Coppermine) and 3GB physical memory. The purpose is to evaluate the efficiency of our solution in terms of the running time as compared with existing algorithms.

We ran the three different algorithms on all the datasets except for the largest one, respectively. This result is shown in Figure 15. While performing the experiments for the YS algorithm, we allocated as much memory as available (up to 3GB) to cache B-tree pages (with an underlying LRU buffer manager).

First of all, the experiment results show that our solution is very efficient and runs much faster than other existing algorithms. When applied on a large dataset (la2h), it only uses about 3 times faster than IMR and 2 order of magnitude faster than YS.

Second, the benefit of finding neighbors using an incore octree rather than searching a B-tree is significant. The YS algorithm suffers from the O$(\log N)$ cost of searching a neighbor from the B-tree. With total cost of O$(N \log N)$ for neighbor-findings, its running time is not linearly scalable. Worse, when the dataset size far exceeds that of main memory, neighbor-findings may cause page faults and disk I/O. The performance degradation is detrimental. For example, the YS algorithm ran for more than 2 weeks on the la2h dataset whose size is 20GB.

Third, the benefit of avoiding multiple iterations of neighbor-findings is evidenced by the performance difference between IMR, which uses the conventional multiple-iteration algorithm and BBP, which uses prioritized ripple propagation. Although not a critical issue in complexity analysis, such constant factor as introduced by multiple iterations does make a big difference in practice, esp. for large datasets.

10.3 Is the solution scalable?

We conducted this set of experiments on a HP AlphaServer with 64 1.15 GHz EV7 processors and 256 GB of shared memory, running the Tru64 Unix operating system. Each experiment was submitted to the system

Name	YS	IMR	BBP
la0.5h	00:29:36	00:05:37	00:01:57
la1h	10:07:09	1:44:55	00:28:06
la2h	$>$ 2 weeks	19:48:24	05:51:30

Figure 15: The running time of different algorithms on the same datasets.

as a job through PBS (Portable Batch System). A job must explicitly specify the number of processors to use and the size of memory needed. All of our experiments were run on one processor and requested 2GB memory. The purpose of these experiments is to evaluate the scalability of our solution.

Figure 16 summarizes the statistics of running the BBP algorithm on dataset la2h and la3h. The "Queries" column reports the number of octants returned by range queries. The "Lev" specifies the subtree root level corresponding to the 3D volume parts. The "Time" column reports the total running time of balancing the datasets in the form of hh:mm:ss. The "DB" column shows the percentage of time spent in database operations, including range queries and B-tree updates (bulk-loading time not included). The "Thruput" column presents the throughput rates of octants/second.

Name	Queried	Lev	Time	DB	Thruput
la2h	15,595,416	3	03:04:50	10.3%	111k
la3h	55,340,273	4	10:00:15	6.1%	104k

Figure 16: High throughput sustained for extremely large datasets

The most striking result is the throughput of balancing the la3h dataset (104k octants/sec) is almost identical to that of the la2h dataset (111k octants/sec), while its size is almost three times as large (56GB vs. 20GB). Given the fact that both experiments requested only 2GB memory, the sustained throughput rate is a solid proof that our algorithm scales gracefully to handle very large datasets without extra memory requirement.

A second interesting result is that although the memory (2GB) can only accommodate small 3D volumes of a dataset (in fact, 1/27 of the la2h dataset and 1/64 of the la3h dataset), range queries retrieved less than 1.5% of the octants in the dataset. Therefore, most octants (98.5% of the dataset) are streamed into memory via efficient sequential scan of B-tree pages.

Third, the time spent in standard database operations (range queries, insertions and deletions) only accounts for about 10% of the total running time, we can deduce that most of the running time (90%) is spent in "real computation", i.e. the construction of cache octrees and the execution of prioritized ripple propagation algorithm. Since the total cost of running the PRP algorithm is O(N) on average, the scalability (sustained high throughput rate) we achieved has a strong theoretical support.

We notice that the number of subdivisions (see Figure 14) in both cases are less than 0.4% and the total number of octants increased is no more than 3%. We argue that this low subdivision rate and dataset size increase, though specific to our datasets, is not uncommon in real-world datasets, where the original (unbalanced) octrees are usually built to model a physical field or geometry with inherent continuity at many locations.

10.4 What is the impact of memory size?

Experiment results shown in this section are obtained from the same HP AlphaServer mentioned earlier. We are interested in finding out the impact of memory size on our algorithm. Does the algorithm run faster with unbounded main memory size?

Figure 17 summarizes the result of running our algorithm on la2h dataset with different memory usage. The "Memory" row lists the actual peak memory usage by the experiments and the "Time" row shows the running time.

Memory	418MB	1.43GB	5.39GB	15.4GB	43.8GB
Time	03:07:15	03:04:50	03:08:00	03:23:37	03:25:23

Figure 17: The impact of the memory size on the running time.

It is clearly shown that our solution performs equally well no matter how much memory is available. In fact, the run using 418MB memory is as fast (if not faster) as the run using 43.8GB. A factor of more than 100 in terms of memory usage!

The reason why the memory usage (for the 43.8GB case) exceeds the size of the dataset (20GB for la2h) is that we have loaded the whole linear octree in a gigantic cache octree. With the extra overhead associated with pointer-based octree and other internal data structures, the actual memory usage doubled the size of the dataset itself.

It is surprising to see that the fastest run is the one that used only 1.43GB, which is followed by the 418MB run, and then the 5.39GB, the 15.4GB and finally the 43.8GB. A contradiction to the common belief that the more memory you have, the faster a program runs. However, a brief study of the system's architecture sheds some light on this abnormal phenomenon. Though claimed to be an SMP architecture, the EV7 based AlphaServer is actually a CC-NUMA (Cache Coherent Non-Uniform Memory Access) system. Since each processor has 4GB local memory, memory allocated beyond this threshold requires remote memory access that is slower than accessing local cache. Therefore our explanation for the unexpected results is that the performance is indeed improved when more memory is allocated *locally* but the benefit of larger memory vanishes when most memory is allocated *remotely*.

In summary, our experiments show that our balance refinement solution is both efficient and scalable.

11 Conclusion

This paper presents the solution to the problem of balance refinement of massive linear octrees. We combine existing database techniques (B-tree, bulk-loading, and range query) with new algorithms (balance by parts, prioritized ripple propagation) and data structure (cache octree) in a unified framework that delivers new capability to support large scientific applications.

In general, hybrid problems such as balancing massive linear octree datasets present a new challenge to dealing with massive data. The fundamental nature of such problems is that the entire dataset has to be processed, iteratively sometimes, to locate data items that need to be modified. Given the complexity of such problems, a good solution should not only reduce the disk I/O time but also improve the computational cost of data manipulation. As a result, conventional database techniques should be used with discretion in order to avoid creating unexpected performance bottleneck.

References

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision,Graphics,and Image Processing*, 24:1–13, 1983.

[2] V. Akcelik, J. Bielak, G. Biros, I. Epanomeritakis, A. Fernandez, O. Ghattas, E. J. Kim, J. Lopez, D. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terasacale computers. In *Proceedings of SC2003*, Phoenix, AZ, 2003.

[3] B. Aronov and H. Bönnimann. Cost prediction for ray shooting. In *Proceedings of the 18th Annual ACM Symposium on Computational Geometry*, pages 293–302, june 2002.

[4] R. E. Bank, A. H. Sherman, and A. Weiser. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, pages 3–17, 1983.

[5] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O'Hallaron, J. Shewchunk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 1998.

[6] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[7] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proceedings of 31st Symposium on Foundation of Computer Science*, pages 231–241, 1990.

[8] D. Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, Jun 1979.

[9] C. Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Press, 1996.

[10] I. Garnagntini. Linear octree for fast processing of three-dimensional objects. *Computer Graphics,and Image Processing*, 20:365–374, 1982.

[11] M. Griebel and G. W. Zumbusch. Parallel multigrid in an adaptive pde solver based on hashing and space-filling curves. *Parallel Computing*, 25(7):827–843, July 1999.

[12] S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. In *Proceedings of the Eighth Symposium on Computational Geometry*, pages 212–221, Feb 1992.

[13] D. Moore. The cost of balancing generalized quadtrees. In *Proceedings of the 3rd Symposium on Solid Modeling and Applications*, pages 305–312, 1995.

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

[15] T. Tu, D. O'Hallaron, and J. Lopez. Etree: A database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 127–138, Ithaca, NY, Sep 2002.

[16] T. Tu, D. O'Hallaron, and J. Lopez. The etree library: A system for manipulating large octrees on disk. Technical Report CMU-CS-03-174, School of Computer Science, Carnegie Mellon University, 2003.

[17] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Survey*, 33(2):209–271, june 2001. A shorter version appeared in Proceedings of the 17th Annual ACM Symposium on Principles of Database Systems (PODS '98).

[18] M. A. Yerry and M. S. Shepard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal for Numerical Methods in Engineering*, 20:1965–1990, 1984.

[19] D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant, and J. E. Bussoletti. A locally refined rectangular grid finite element: Application to computational fluid dynamics and computational physics. *Journal of Computational Physics*, 92:1–66, 1991.