Balance Refinement of Massive Linear Octrees

Tiankai Tu† David R. O'Hallaron†∗
†Computer Science Department
∗Department of Electrical and Computer Engineering
Carnegie Mellon University, Pittsburgh, USA
email:{tutk,droh}@cs.cmu.edu Phone:1-412-268-3043 Fax:1-412-268-5576

Abstract

This paper presents a solution to the problem of balance refinement of massive linear octrees. We combine existing database techniques (B-tree, bulk loading, and range queries) with new algorithms (balance by parts, prioritized ripple propagation) and data structures (the cache octree) into a unified framework that provides new capabilities for large scientific applications.

1 Introduction

Extensive applications of octrees date back to the early 1970's [17]. Given three decades of research, it might be thought that octrees have been fully studied. Interestingly, an indispensable operation required by many applications, *balance refinement*, has somehow been largely ignored.

The purpose of balance refinement is to enforce a continuity condition on an existing octree such that the octree decomposition is relatively smooth throughout the domain, with no abrupt changes in size between adjacent leaf octants. This continuity condition typically requires that no two leaf octants sharing a face or an edge should differ by a factor of more than two 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 one in their tree levels. To make the term more intuitive, we will refer to the continuity condition as the *2-to-1 constraint*.

Applications that require the 2-to-1 constraint on octrees include scientific computing [5, 22, 13, 3], quality mesh generation [21, 8, 15, 18], and computer graphics [4]. In spite of this, there is relatively little work on balancing octrees. Although balance refinement is not a big issue when the dataset can be completely cached in main memory, it represents a serious problem for applications such as scientific computing that require massive balanced linear octree datasets. Typically, in order to simulate large and complex physical phenomena, scientific applications require billions of octants or even more to model and resolve the domain of interest. The sizes of these datasets can be on the order of hundreds of gigabytes to terabytes, and getting larger all the time.

Efficiently balancing massive linear octrees that cannot be completely cached in main memory constitutes a unique challenge. Unlike massive data *query problems* where sophisticated synopsis data structures can be used to provide fast, approximate responses [12], balance refinement must produce exact results as required by the 2-to-1 constraint. Unlike relatively lightweight *scientific workflow applications*, where conventional DBMS's can be naturally extended to provide the functionality [2], the process of balance refinement is much more complicated and the dataset is too large to be treated as flowing objects. Instead we view balance refinement as a hybrid of Vitter's canonical categories of massive data problems (*batched problems* and the *online problems*) [20]. It is a batched problem because every item (octant) in the dataset must 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.

This paper presents an efficient and scalable balance refinement method. Like many other database algorithms, our method exploits locality of reference to reduce disk I/O. The main new algorithm is called *balance by parts (BBP)*. 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 a *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 a new algorithm called *prioritized ripple propagation (PRP)* to balance it efficiently. While adjusting the structure of the cache octree, we update the linear octree dataset accordingly.

Evaluation results show that our method is both efficient and scalable. It runs over 3 times faster than any existing algorithm when used to balance a 20GB dataset with 1.2 billion octants on a Linux workstation with 3GB main memory. It also maintains high throughput rates when used to balance very large datasets (56GB).

The paper makes two main contributions. First, we present a new method that seamlessly integrates existing database techniques (B-tree, bulk loading, range queries) with new scalable algorithms (BBP and PRP). We introduce a more efficient caching mechanism (the cache octree) to incorporate locality directly into the algorithm design, thereby greatly outperforming algorithms that use conventional B-tree page level caching. Second, we provide a powerful data management tool for scientists and engineers to deal with massive linear octree datasets. In fact, our balance refinement algorithm has already had significant impact in the scientific computing world, making it possible for the first time to generate unstructured hexahedral finite element meshes with billions of elements. For generating and simulating earthquake ground motion in the Los Angeles basin using these meshes, the authors (along with other members of our group) received the 2003 Gordon Bell Award [3].

Sections 2 and 3 provide background on octrees and related work. Section 4 defines the fundamental performance problems we are attempting to solve. Sections 5–8 discuss aspects of the BBP algorithm, including the cache octree structure. Section 9 describes the PRP algorithm. Section 10 evaluates the performance of our approach.

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 [17]. 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.

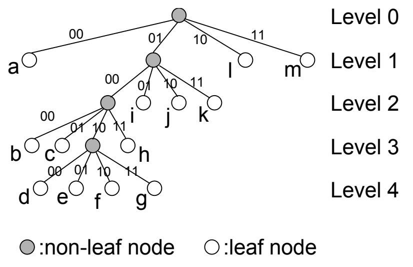

Figure 1. Pointer-based octree.

The other common way to represent an octree is the *linear octree* [1, 11], which assigns a unique key to each node and represents an octree as a collection of sorted leaf nodes, which can be organized on disk by an index structure such as a B-tree [9, 7]. Linear octrees are useful in practice where main memory cannot accommodate a complete pointer-based octree. In this paper, we only consider linear octree datasets indexed and stored in 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

[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 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 [10] to derive the locational code is based on bit-shuffling of the coordinates, which is often used in practice.

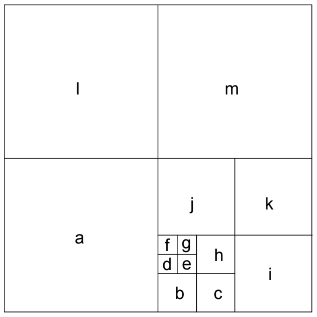

Figure 2. Domain decomposition.

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. (In the context of disk-resident linear octrees, we refer to the leaf nodes simply as octants, since no non-leaf nodes are stored on disk.)

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

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`.

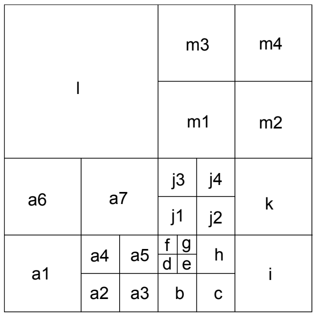

Figure 3. Balanced domain decomposition.

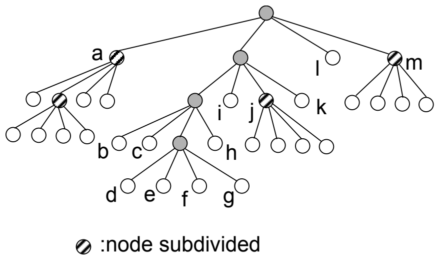

Figure 4. Balanced octree.

3 Related Work

Moore studied the space cost of balancing generalized octrees and proved that each octree has a *unique* least common balance refinement [16]. Yerry and Shepard developed a balance refinement algorithm based on a breath-first expanded octree in [21]. We recently proposed a balance refinement algorithm called *local balancing* [18]. (*The full paper will elaborate.*)

4 Problem Statement

The balance refinement process consists of two main operations: (1) *neighbor finding*: finding neighbors to obtain their edge sizes information in order to make comparison; (2) *subdivision*: deleting a “too-large” octant from the dataset and inserting its eight children. The deletion is necessary 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 $\mathrm{O}(\log N)$, 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 $\mathrm{O}(N \log N)$. 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 [17]. The advantage is that the average cost of neighbor finding is reduced to $\mathrm{O}(1)$, with a total cost of only $\mathrm{O}(N)$ 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 (BBP)

Our method 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 new algorithm, called *balance by parts (BBP)*. 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 a *cache octree*. While balancing a cache octree in memory, we update the B-tree to record the subdivisions of leaf octants. Another new algorithm called *prioritized ripple propagation (PRP)* 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 to divide-and-conquer, this method is different from our previous

Algorithm 1 (Balance by parts).

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

Output:
A balanced linear octree indexed 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.

work [18] 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.

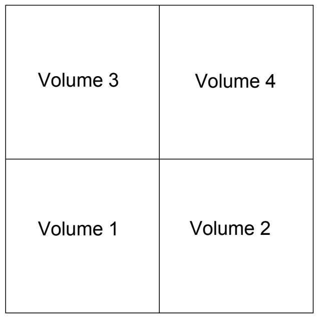

Figure 6. Partition the domain into 4 volumes corresponding to tree level 1.

5.2 An Example

Here is an example of applying BBP 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.

Assuming that the largest volumes that can fit in memory are the non-leaf nodes (of the conceptual quadtree) at level 1, we partition the domain into 4 volumes, as shown 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.

After all the volumes are processed, the 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 shows the cache octree representing octants on the boundary between volume 1 and volume 2. The corresponding region is shown in Figure 9. Note that all the subdivisions triggered by

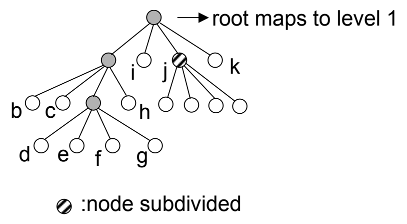

Figure 7. The balanced cache octree representing volume 2.

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.

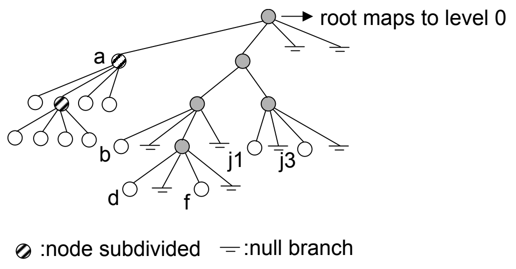

Figure 8. The balanced cache octree representing line boundary part between volume 1 and volume 2.

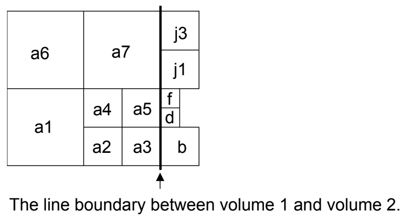

Figure 9. The boundary part between volume 1 and volume 2.

The point boundary part consists of the four octants anchored at the center point of the domain. Since no subdivision occurs inside the point part, we omit the cache octree representation for the point boundary part.

5.3 Questions to be Answered

Although structurally simple, our algorithm raises a number of 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? 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 monotonically 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 using the concepts of internal octants and boundary octants. See Appendix A for details.

7 Data Retrieval

We retrieve data from a linear octree dataset in two different ways: *bulk loading* and *range queries*. 3D volume parts can be retrieved by bulk loading, since 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 algorithm 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 10). 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[2]. For a 3D volume part, we map the root node to the non-leaf node corresponding the 3D volume. For other parts, we map the root node to level 0. 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*. The cost of building a cache octree is linear to the number of leaf nodes.

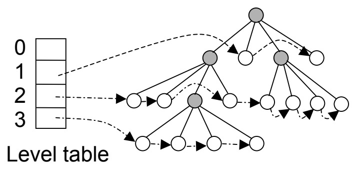

Figure 10. A cache octree is a pointer-based octree with leaf nodes at the same level linked together.

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*.

We have now 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 and then cache each piece in a pointer-based cache octree. In this way, we can work with small main memory and still take advantage of the faster pointer-based neighbor finding algorithm.

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

9 Prioritized Ripple Propagation (PRP)

The algorithm presented in this section resolves the second performance problem, namely avoiding multiple iterations of neighbor findings. The key ideas are to (1) decouple node visiting from tree structure traversal; and (2) combine neighbor findings with node subdivisions.

Figure 11 shows the outline of our algorithm named *prioritized ripple propagation* (PRP), which operates on 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.

Algorithm 2 (Prioritized ripple propagation).

Input:
An unbalanced cache octree.

Output:
A balanced cache 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, terminate.]

Figure 11. Prioritized ripple propagation.

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 to this algorithm is Step 3, where a *ripple routine* is invoked to implement the eager strategy. The ripple routine combines neighbor findings with node subdivisions. It is based on the well-known pointer-based neighbor finding algorithm [17], 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 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.

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. Plus the cost of building cache octrees, the overall cost is still O(N).

10 Evaluation

This section attempts to answer the following questions: (1) How does running time of our algorithms compare to previous approaches? 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 method scalable to massive linear octrees? (3) What is the impact of memory size on performance?

10.1 Methodology

We implemented three different balance refinement algorithms: (1) our method (BBP/PRP), (2) an external memory version of Yerry and Shepard's algorithm (YS) [21], and an improved version of our previously published algorithm (IMR) [18]. BBP/PRP was implemented using the etree library, a runtime system for creating and manipulating linear octrees stored on disk [19].

The experiments were conducted with a collection of massive real-world linear octree datasets from the Carnegie Mellon Quake project [3, 6]. The original (unbalanced) octrees partition a $100\text{km} \times 100\text{km} \times 37.5\text{km}$ region of the Greater Los Angeles Basin. The sizes of the particular octants are determined by the density of the ground and the desired frequency resolution, with softer regions represented by smaller octants. An unbalanced octree is first balanced and then transformed into an unstructured hexahedral mesh that serves as the input dataset for an octree-based finite element solver [3, 14, 22].

Dataset	Octs (before)	Octs (after)	Subdivs	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 12. Quake project LA datasets.

Figure 12 summarizes the characteristics of the balanced linear octree datasets. The dataset names characterize the frequency resolution of the octree in Hz. For example, the la2h octree can resolve seismic waves of up to 2 Hz. The "Octs (before/after)" columns record the numbers of octants in the linear octree before and after the balance refinement, respectively; "Subdivs" records the number of subdivisions triggered; "Size"reports the sizes of the B-tree files storing the linear octrees after balance refinement. (The unbalanced datasets are about 10% smaller).

10.2 Is the method efficient?

This set of experiments was conducted on a Linux 2.4 workstation with PIII 1GHZ processor and 3GB physical memory. Figure 13 shows the running times for all three algorithms on all but the largest octree (la3h). 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).

Dataset	YS	IMR	BBP/PRP
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 13. Running times.

There are three interesting observations: (1) BBP/PRP runs much faster than YS or IMR. When applied on a large dataset (la2h), BBP is about 3 times faster than IMR, and 2 orders of magnitude faster than

YS. (2) The benefit of finding neighbors using an in-core 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 a 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. For example, the YS algorithm ran for more than 2 weeks on the la2h dataset. (3) The benefit of avoiding multiple iterations of neighbor findings is clear from the performance difference between IMR, which uses the conventional multiple-iteration algorithm, and BBP/PRP, which uses prioritized ripple propagation. Although not a critical issue in complexity analysis, constant factors introduced by multiple iterations do make a big difference in practice, especially for large datasets.

An important conclusion from these observations is that a sophisticated caching mechanism (cache octrees) is much more effective than simple B-tree page-level caching.

10.3 Is the method scalable?

The experiments in this section were run on an HP AlphaServer with 64 1.1 GHz EV7 processors and 256 GB of shared memory. All of our experiments were run on one PE and requested 2GB of main memory.

Figure 14 summarizes the performance of the BBP/PRP algorithm on the large la2h and la3h datasets. “Queries” is the number of octants retrieved by range queries; “Time” is the measured running time (hh:mm:ss); “DB” is the percentage of time spent in database operations, including range queries and B-tree updates (bulk loading time not included); and “Thruput” is the throughput rate in octants/sec (i.e., total octants in the balanced dataset as shown in Figure 12 divided by running time).

Dataset	Queries	Time	DB	Thruput
la2h	15,595,416	03:04:50	10.3%	111k
la3h	55,340,273	10:00:15	6.1%	104k

Figure 14. Sustained BBP/PRP throughput.

The most striking result is that the throughput for the la3h dataset (104k octants/sec) is almost identical to that of the la2h dataset (111k octants/sec), even though its size is almost three times as large (56GB vs. 20GB). Given that both experiments requested only 2GB memory, the sustained throughput rate suggests that BBP/PRP scales gracefully to handle very large datasets without requiring a larger main memory. Second, fewer than 1.5% octants are retrieved by range queries, representing a very small overhead to process the boundary parts. Third, database operations account for about 10% of the total running time. In other words, 90% of time is spent in “real computation”, i.e., constructing and balancing cache octrees. Since the average running time of PRP is O(N), the scalability achieved as sustained high throughput rate has a theoretical foundation. (*The final paper will elaborate.*)

10.4 What is the impact of memory size?

Figure 15 summarizes the result of running BBP/PRP on the la2h dataset using different main memory sizes (on the same HP AlphaServer). “Memory” lists peak memory usage; “Time” is the measured running time. The important conclusion is that BBP/PRP does not require an extremely large memory size in order to run efficiently. (*The final paper will elaborate.*)

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 15. Impact of memory size on run time.

11 Conclusion

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

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

Acknowledgements

We thank Christos Faloutsos and Natassa Ailamaki for helping us understand spatial databases. This work is sponsored in part by the National Science Foundation under Grant CMS-9980063, in part by a subcontract from Southern California Earthquake Center as part of NSF ITR EAR-01-22464, and in part by a grant from the Intel Corporation.

References

[1] 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] A. Ailamaki, Y. Ioannidis, and M. Livny. Scientific workflow management by database management. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, Capri, Italy, 1998.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

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

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

[12] P. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science: Special Issue on External Memory Algorithms and Visualization*. 1998.

[13] 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.

[14] E. Kim, J. Bielak, and O. Ghattas. Large-scale northridge earthquake simluation using octree-based multiresolution mesh method. In *Proceedings of the 16th ASCE Engineering Mechanics Conference*, Seattle, Washington, July 2003.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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).

[21] 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.

[22] 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.

A Detailed Correctness Proof for the Balance by Parts Algorithm

A.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 16, every stage processes the 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.

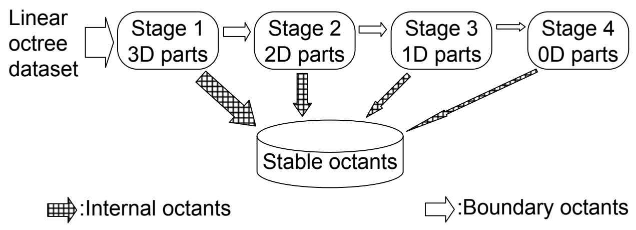

Figure 16. Four stages of balancing parts of different types.

A.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 17). Thus the third possibility does not exist. □

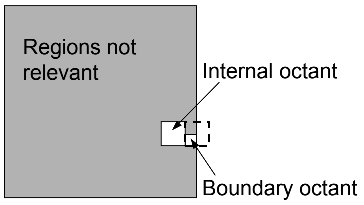

Figure 17. The boundary octant in a balanced 3D volume cannot be half as large as its internal neighbors.

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 the BBP algorithm.