SCALABLE PARALLEL OCTREE MESHING FOR TERASCALE APPLICATIONS

TIANKAI TU *, DAVID R. O'HALLARON †, AND OMAR GHATTAS ‡

Abstract. We present a new methodology for generating and adapting octree meshes for terascale applications. Our approach combines existing methods, such as parallel octree decomposition and space-filling curves, with a set of new methods that address the special needs of parallel octree meshing. We have implemented these techniques in a parallel meshing tool called Octor. Performance evaluations on up to 2000 processors show that Octor has good isogranular scalability, fixed-size scalability, and absolute running time. Octor also provides a novel data access interface to parallel PDE solvers and parallel visualization pipelines, making it possible to develop tightly coupled end-to-end finite element simulations on terascale systems.

1. Introduction. The emergence of terascale computing has created unprecedented new opportunities for scientists and engineers to simulate complex physical phenomena on a larger scale and at a higher resolution than heretofore possible. This trend has introduced new challenges to applications, algorithms, and systems software. In particular, a significant challenge for terascale finite element simulations is how to generate and adapt high-resolution unstructured meshes with billions of nodes and elements, and how to deliver such meshes to the processors of terascale systems.

A typical approach for preparing a finite element mesh for simulation is to first generate a large mesh structure offline on a server [34, 35], and then upload the mesh to a supercomputer, where additional steps such as mesh partitioning [22] and data redistribution are performed. The result is a series of time-consuming file transfers and disk I/O that consume large amounts of network and storage resources while contributing nothing to the applications that will be using the mesh structure. Further, since the meshes are generated offline on a server, the offline algorithm is unable to adapt the mesh dynamically at runtime. We propose a better approach where the meshes are generated *in situ*, on the same processors where they will later be used by applications such as finite element solvers and visualization pipelines.

In this paper, we describe a new methodology for parallel octree meshing for terascale applications. At first glance, it may appear that parallel octree meshing is simply a direct application of the well-known parallel octree method. In fact, parallel octree meshing is fundamentally different from other parallel octree applications in that it must manipulate the vertices (corners) of the octants, which correspond to the mesh nodes. This is required to support finite element simulations, which associate unknowns to mesh nodes and then solve the resulting linear system. Complicated correlations between octants and vertices, and between vertices and vertices, either on the same processor or on different processors, must all be identified and tracked. Thus, parallel octree meshing presents a set of new problems that do not exist in other parallel octree applications.

Our work builds on a foundation of previous work on parallel octrees [4, 10, 14, 32, 37, 39], mesh generation [6, 7, 8, 13, 14, 21, 26, 31], parallel adaptive mesh refinement [2, 9, 23, 38], parallel adaptive finite element methods [18, 28] and space-filling curves [5, 15]. To address the special requirements of parallel octree meshing, we have developed a set of new algorithms and techniques: (1) a new algorithm called *parallel prioritized ripple propagation* that balances an octree efficiently; (2) a sophisticated memory management scheme that facilitates data migration between processors; (3) a new method of extracting parallel mesh structures that delivers mesh data directly to solvers; and (4) a novel data access interface that allows solvers or visualizers to easily interact with the mesher.

We have implemented our methodology within a new parallel meshing tool called *Octor*. Performance evaluations show that Octor has good isogranular scalability, good fixed-size scalability, and good absolute

*Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA (tutk@cs.cmu.edu)

†Computer Science Department and Department of Electrical and Computer Engineering, Carnegie Mellon University, PA 15213, USA (droh@cs.cmu.edu)

‡ Institute for Computational Engineering and Sciences, and Departments of Geological Sciences, Mechanical Engineering, Computer Science, and Biomedical Engineering, University of Texas, Austin, TX 78712, USA (omar@ices.utexas.edu)

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

SC|05, November 12-18, 2005, Seattle, Washington, USA

(c) 2005 ACM 1-59593-061-2/05/0011...$5.00

running time. Roughly speaking, the runtime increases only logarithmically with the problem size (as measured by the number of mesh elements).

To our knowledge, Octor is the first parallel meshing tool that is capable of both generating massive unstructured meshes statically (that is, before a solver starts running) on thousands of processors, and supporting dynamic mesh adaptation (that is, while a solver is still executing). Furthermore, it is the first parallel meshing tool that supports tight coupling with other components downstream in the simulation process, thus making it possible to develop terascale end-to-end finite element simulations [36] without the overheads of intermediate file transfers and disk I/O.

We have run Octor on Lemieux, the HP AlphaServer system at the Pittsburgh Supercomputing Center, to generate unstructured finite element meshes for simulating the 1994 Northridge earthquake in the Greater Los Angeles Basin. The largest mesh, with 1.22 billion elements and 1.37 billion nodes, resolves seismic wave frequencies up to 2 Hz. It takes only 333 seconds to generate this mesh on 2000 PEs. Given that the execution time of a highly efficient seismic wave propagation solver [3] would be more than 60 hours on the same 2000 PEs for a typical 80 second earthquake simulation, the meshing time is inconsequential.

The rest of the paper describes the design, implementation and evaluation of Octor. Section 2 provides the background of octree meshing. Section 3 briefly discusses related work. Section 4 presents an overview of the design of Octor. Section 5 explains the algorithm and implementation details. Section 6 presents the performance evaluation of Octor.

2. Background. Among the different types of finite element meshes, octree-based hexahedral meshes provide a trade-off between modeling capability and simplicity. On the one hand, octree meshes provide multi-resolution capability by local adaptation. On the other hand, octree meshes simplify the treatment of geometry and boundaries by spatial approximation. Octree mesh methods are employed in at least three ways. First, for PDEs posed in simple domains characterized by highly heterogeneous media in which solution length scales are known *a priori* (such as in linear wave propagation), octree meshes that resolve local solution features can be generated up front. Second, for PDEs in simple domains having solution features that are known only upon solution of the PDEs, octree meshes — driven by solution error estimates — can be adapted dynamically to track evolving fronts and sharp features at runtime (for example to capture shocks). Third, for PDEs posed on complex domains, octrees meshes in combination with special numerical techniques (such as fictitious domain, embedded boundary, or extended finite element methods) can be used to control geometry approximation errors by adapting the octree mesh in regions of high geometric variability, either *a priori* for fixed geometries, or at runtime for evolving geometries. Large-scale examples of *a priori* adapted octree mesh generation can be found in seismic wave propagation modeling [24], while octree mesh methods for compressible flow around complex aircraft configurations provides an excellent example of geometry- and solution-driven dynamic adaptivity [40].

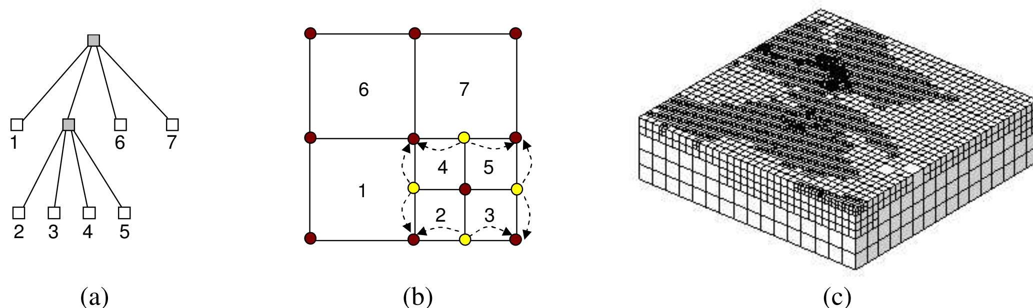


FIG. 2.1. *Octree-based hexahedral meshes. (a) Octree domain decomposition. (We use a 2D quadtree to illustrate concepts, but all techniques are designed and implemented for 3D cases.) (b) Octants map to elements and vertices map to mesh nodes. The dark colored dots represent the anchored nodes and the light colored dots represent the dangling nodes. The dashed arrows represent the explicit correlations between dangling nodes and anchored nodes. (c) An example 3D octree mesh.*

Conceptually, generating or adapting an octree-based hexahedral mesh is straightforward. As shown in Figure 2.1, we first refine a problem domain recursively using an octree structure. We require that two

adjacent octants sharing an edge of a face should not differ in edge size by a factor of 2, a constraint often referred to as the **balance condition** or, more intuitively, **2-to-1 constraint**. We then map octants to **mesh elements** and vertices to **mesh nodes**. The nodes hanging at the midpoint of an edge or the center of the face of some element (due to the 2-to-1 constraint) are **dangling nodes**. The remaining nodes are **anchored nodes**. For conforming finite element methods, each dangling node is dependent on the anchored nodes at the endpoints of the edge or the face on which it is hanging through an explicit algebraic constraint. Explicit correlations between dangling nodes and anchored nodes are established.

3. Related work. Parallel octree structures have been successfully applied in many areas, especially in the area of N-body simulations [4, 32, 37, 39]. Unlike in octree meshing, these applications do not need to manipulate the vertices of the octants. Nor is it necessary to enforce the balance condition on the octree.

In the mesh generation area, a typical way to use a quadtree or octree is to use the tree structure to partition a 2D or 3D point set, and then either generate triangular or tetrahedral meshes by "warping" the vertices of the leaf octant [6, 26, 31], or convert the leaf octants to hexahedral meshes [30]. In recent years, significant research focus has been directed to parallel mesh generation [7, 8, 11, 12, 14]. Most parallel mesh generators, though, run on 32 or 64 processors. To our knowledge, none has run on more than 128 processors. However, it should be noted that generating geometry-conforming 3D meshes (either Delaunay or non-Delaunay) in parallel is a very difficult problem and is being actively researched.

For parallel mesh adaptation, there are two popular methods: adaptive mesh refinement (AMR) [2, 9, 23, 38] and adaptive finite elements [16, 18, 28]. Both methods start with an initial, coarse mesh, and as the numerical calculation proceeds, adapt the mesh structure (using different techniques) in those areas where error estimators or indicators dictate a need to do so.

4. Design overview. Figure 4.1 shows an end-to-end physical simulation process. First, a mesh is generated to model the material property or geometry of the problem. Next, a solver takes the mesh as input and conducts numerical calculations, for example, to solve a system of PDEs. In some applications, the mesh structure needs to be adapted dynamically while the solver is still executing. The numerical results produced by the solver, such as displacements or temperatures, are then correlated to the mesh structure by a visualizer to create 3D images or animations.

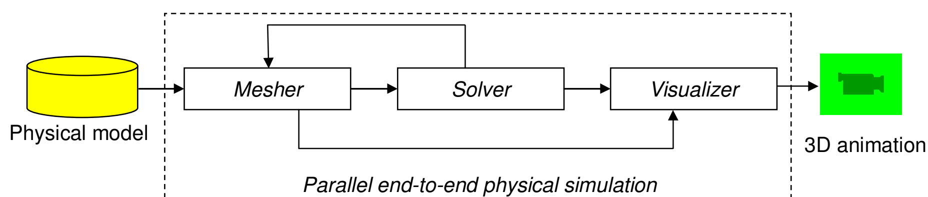


FIG. 4.1. *End-to-end physical simulation process. The backward arrow from solver to mesher represents mesh structure adaptations driven by numerical computations, for example, in adaptive finite element simulations.*

Our design goal of Octor is to develop a scalable, efficient, and easy-to-use mesher to support such end-to-end simulations for terascale applications. *Scalable* means that the mesher should be able to run on a large number of processors without performance degradation. *Efficient* means that mesh elements and nodes should be produced where they will be used instead of on remote processors. *Easy-to-use* means that other simulation components should be able to interact with Octor in a simple way.

To achieve this design goal, we have combined existing techniques with new methods. More specifically, we have used the well-known parallel octree to manage the backbone meshing data structure and the space-filling curve (SFC) based techniques to partition meshes. To address the special needs of parallel meshing, we have developed new algorithms and techniques: (1) *parallel prioritized ripple propagation*, a new algorithm that efficiently enforces the 2-to-1 constraint on a parallel octree; (2) a sophisticated memory management scheme that keeps track of application-specific meshing data internally and manage data migration among processors (caused by partitioning) automatically; (3) a new method of extracting mesh structures — instead of extracting a mesh structure first and then partitioning, we partition the backbone parallel octree first and then extract the mesh structure; and (4) a novel data access interface that allows a solver or a visualizer to

interact easily with the mesher.

Octor provides a small set of API functions (see Appendix) that allows users to write meshing applications. At a high level, Octor meshing consists of the steps shown in Figure 4.2. First, NEWTREE bootstraps a small octree on each processor. Next, the tree structure is adjusted by REFINETREE and COARSENTREE, either statically or dynamically. While adjusting the tree structure, each processor is responsible only for a small area of the domain. When the adjustment completes, there are many subtrees distributed among the processors. The BALANCETREE step enforces the 2-to-1 constraint on the entire parallel octree. After a balanced parallel octree is obtained, PARTITIONTREE redistributes the leaf octants among the processors. Finally and most importantly, EXTRACTMESH derives mesh element and node information and determines the various correlations between elements and nodes. The procedure just described provides a general framework for different types of meshing. A particular application may choose not to execute some of the listed steps. For example, for static mesh generation, it might be unnecessary to execute the COARSENTREE step. For dynamic mesh adaptation, it might be unnecessary to run PARTITIONTREE if the numbers of leaf octants on different processors differ only within a small percentage.

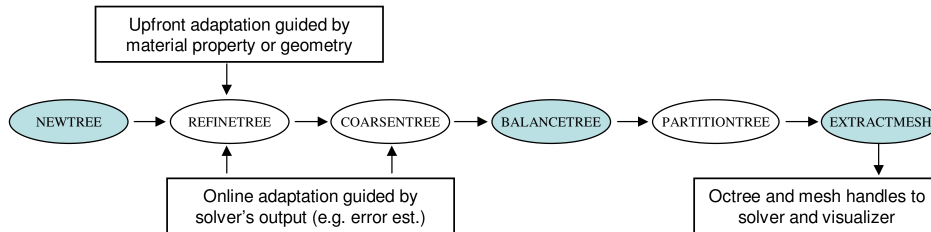


FIG. 4.2. *Octor meshing steps. Shaded ovals are required steps. Unshaded ovals are optional steps.*

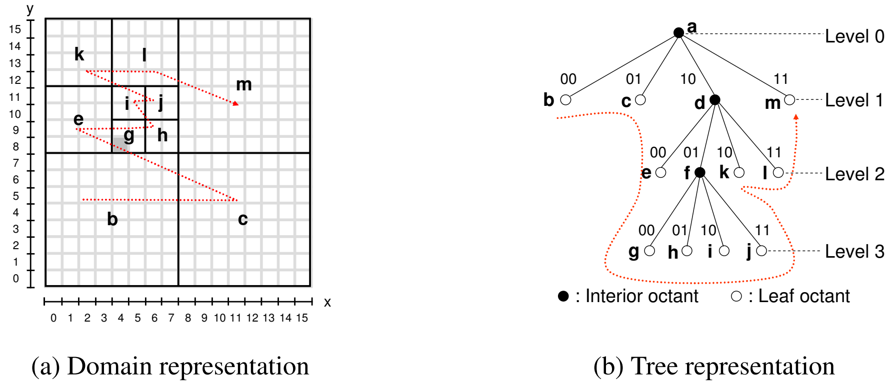
An application such as a finite element solver or a parallel visualization program can interact with Octor in one of two ways. (1) *Indirect interaction.* An application can guide the manipulation of Octor's internal data structures indirectly using *callback functions*. The function prototypes for REFINETREE, COARSENTREE, and BALANCETREE contain predefined callback function types. An application implements these callback functions and passes the function pointers as arguments when performing any of the three meshing steps. (2) *Direct interaction.* Octor also returns abstract data types (ADT) such as `octree_t`, `octant_t` and `mesh_t` to an application, and provides a set of macros that allows an application to manipulate these ADTs directly. For example, an application can attach to an ADT a pointer that references an arbitrary data structure in memory whose semantics are irrelevant to meshing. Although simple, these two data access methods have proved to be sufficiently powerful for integrating a parallel mesher, solver and visualizer [36].

5. Internals. This section explains the internal mechanism of Octor. We start with a description of how to organize the underlying parallel octree; then we examine the individual meshing steps in more details.

5.1. Data structures. Before describing the organization of the backbone parallel octree, let us first outline important properties of octrees that we will exploit for parallelism. For simplicity of illustration, we will use 2D quadtrees and quadrants in the figures and examples. Nevertheless, all the techniques and properties are applicable to 3D octrees and octants.

An octree can be viewed in two equivalent ways: the *domain representation* and the *tree representation*. A *domain* is a Cartesian coordinate space that consists of a uniform grid of $2^n \times 2^n$ indivisible **pixels**. The *root octant* that spans the entire domain is defined to be at level 0. Each child octant is one level lower that its parent (with a larger level value). Figure 5.1(a) and (b) show the domain representation and the equivalent tree representation of an octree, respectively. Each tree edge in Figure 5.1(b) is labeled with a binary **directional code** that distinguishes each child of an internal octant.

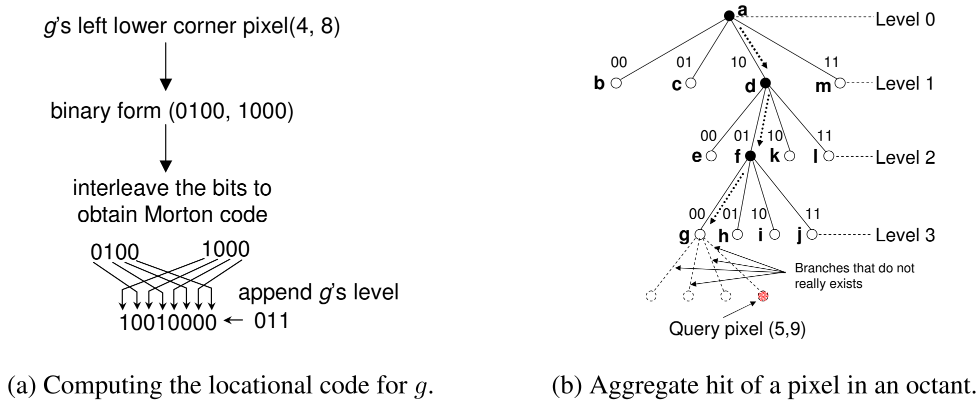
Linear octree. In order to address an octant so that it can be unambiguously distinguished from other octants, we make use of the *linear octree* technique [1, 19, 20]. The basic idea of a linear octree is to encode each octant with a scalar key called a **locational code** that uniquely identifies the octant. Figure 5.2(a) shows how to compute the locational code of octant g. First, interleave the bits of the three coordinates of the octant's lower left pixel to produce its Morton code [27]. Then append the octant's level to compose the



(a) Domain representation (b) Tree representation

FIG. 5.1. *Equivalent representations of an octree.*

locational code. We refer to the lower left pixel of an octant as the octant's **anchor**. For example, the shaded pixel in Figure 5.1(a) is the anchor for octant g.



(a) Computing the locational code for g. (b) Aggregate hit of a pixel in an octant.

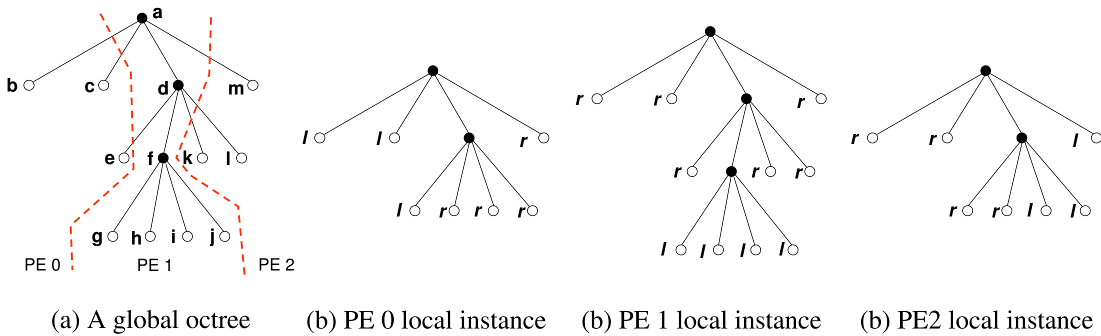
FIG. 5.2. *Operations on octrees.*

Aggregate hit. Given a locational code, we can descend a pointer-based octree to locate an octant. The descending procedure works in the following way: We extract two bits from the start of the locational code repeatedly and follow the branch labeled with the matching directional code until reaching a leaf octant. The fact that we are able to locate an octant this way is not a coincidence. Actually, an alternative way to derive a locational code is to concatenate the directional codes from the root octant to a leaf octant, pad zeroes to make the code equal length, and then append the level of the leaf octant. Figure 5.2(b) shows an example of locating g using its locational code. Note that we have used only the leading bits (100100); the trailing bits (00011) do not correspond to any branches since g itself is already a leaf octant. Generally, we can specify the coordinate of any pixel within the geometric span of an octant, convert it to a locational code, and still be able to locate the enclosing octant. We refer to such a property as **aggregate hit** because the returned octant is an aggregating ancestor of a non-existent octant.

Z-ordering. If we sort all the leaf octants of an octree according to their locational codes, we obtain a total ordering of all the leaf octants. Given the encoding scheme of locational code, it is not difficult to verify that the total ordering is identical to the *pre-order traversal* of the leaf octants of the octree (See Figure 5.1). If we traverse the leaf octants in this order in the problem domain, we follow a Z pattern in the Cartesian space. This is the well-known **Peano space-filling curve** or simply **Z-order curve** [17], which has the nice property that spatially nearby octants tend to be clustered together in the total ordering. This property has enabled us to use a space-filling curve based strategy to partition meshes and distribute workload among processors.

5.1.1. Parallel octree organization. We seek data parallelism by distributing an octree among all processors. Each processor keeps its **local instance** of the underlying global octree. Conceptually, each local instance is an octree by itself whose leaf octants are marked as either *local* or *remote*, as shown in Figure 5.3(b)(c)(d).

The best way to understand the construction of a local instance on a particular processor is to imagine that there exists a pointer-based, fully-grown, global octree (see Figure 5.3(a)). Every leaf octant of this tree is marked as *local* if the processor needs to use the octant, for example, to map it to a hexahedral element, or *remote* if otherwise. We then apply an aggregation procedure to shrink the size of the tree. The predicate of aggregation is that if eight sibling octants are marked as *remote*, prune them off the tree and make their parent as a leaf octant marked as *remote*. For example, on PE 0, octant g, h, i, and j (which belong to PE 1) are aggregated and their parent is marked as a remote leaf octant. The shrunken tree thus obtained is the local instance on the particular processor. Note that all the internal octants — the ancestors of leaf octants — are unmarked. They exist simply because we need to maintain a pointer-based octree structure on each processor (to implement aggregate hits).



(a) A global octree (b) PE 0 local instance (b) PE 1 local instance (b) PE2 local instance

FIG. 5.3. *Parallel octree organization on 3 processors (PE 0, PE 1, and PE 2). Circles marked by* `l` *represent local leaf octants; and those marked by* `r` *represent aggregated remote leaf octants.*

We partition a global octree among all processors with a simple rule that each processor is a host for a contiguous chunk of leaf octants in the pre-order traversal ordering. In order to keep the parallel octree in a consistent way, we also enforce an invariant that a leaf octant, if marked as *local* on one processor, should not be marked as *local* on any other processors. Therefore, the local instance on one processor is different from the one on any other processor, though there may be overlaps between local instances. For example, a leaf octant marked as *remote* on one processor may actually correspond to a subtree on another processor.

So far, we have used a very shallow octree to illustrate how to organize a parallel octree on 3 processors. In our simple example, the idea of local instances may not appear to be very useful. But in real applications, a global octree can be very deep and needs to be distributed among hundreds or thousands of processors. In these cases, the local instance method excels because each processor only needs to allocate enough memory to keep track of its share of the leaf octants.

It should be mentioned that in practice, due to huge memory requirements and redundant computational costs, we never — and in fact, are unable to — build a fully-grown global octree on a single processor first, and then shrink the tree by aggregating remote octants as an afterthought. Instead, local instances on different processors grow and shrink dynamically in synergy at runtime to conserve memory and keep the global parallel octree in a coherent way.

5.1.2. Locational code lookup table. Searching for an octant is an indispensable operation for generating or adapting a mesh is to search for an octant. If the target octant is hosted on the same processor where the search operation is initiated, then we follow standard octree search algorithms [29] to traverse the local instance to find the octant. But what if the algorithms encounters a leaf octant that is marked as *remote*? In this case, we need to somehow forward the search operation to the remote processor that hosts the target octant, and resume the search on that remote processor.

Our solution, which leverages the octree properties previously described, works in the following way. First, we compute the locational code of the target octant. For example, when we are looking for the neigh-

boring octant of g to its left (see Figure 5.1(a)), we know the position of g itself. Thus it is straightforward to compute its left neighbor's anchor coordinate (assuming the neighbor is of the same size as g) and derive the corresponding locational code. Next, we search for the hosting processor id in an auxiliary data structure called the *locational code lookup table* (discussed shortly). Finally, on the remote processor, we resume the search operation using the aggregate hit search method to locate the target octant.

It would be extremely inefficient, and in most cases, infeasible, to record where every remote octant is hosted. The memory cost would be $\mathcal{O}(N)$, where N is the number of octants, which can be as high as hundreds of millions or even tens of billions. We avoid such excessive memory overhead by taking advantage of a simple observation: Each processor holds a contiguous chunk of leaf octants in the pre-order traversal ordering, which is identical to the total ordering imposed by the locational codes, thus we are given a partitioning of the locational codes in ascending order for free. We exploit this fact to build and replicate a **locational code lookup table** on each processor. Each entry in the table has two fields <*key, value*>. The *key* is the smallest locational code among all the leaf octants hosted by a processor; and the *value* is the corresponding processor id. The table is sorted in ascending locational code order. When searching for a remote processor id using an octant's locational code, we perform a binary search on this table. Note that we do not have to find an exact hit, but rather, we only need to find the entry whose key is the largest among all those that are smaller than the search key, that is, the highest lower-bound.

Using a locational code lookup table, we have reduced the overhead of keeping track of remote octants to $\mathcal{O}(P)$, where P is the number of processors. Even when there are 1 million processors, the memory footprint of the locational code lookup table is only about 13 MB. Since compute nodes of recent new parallel architectures tends to have large physical memory per processor (500 MB— 8 GB), the memory requirement of the locational code lookup table is minimal and should not constitute a scalability bottleneck.

5.2. Algorithms. In the rest of this section, we highlight algorithm and implementation techniques involved in each individual meshing step shown in Figure 4.2, omitting many of the technical details.

5.2.1. NEWTREE. This step operates in three stages: *tree expansion*, *task assignment*, and *tree pruning*.

Tree expansion. Each processor starts with a root octant that encloses the entire domain, and then independently expands this tree until a certain level is reached. The level is determined by the number of processors and is usually a small value (3 or 4). During the expansion process, any octant that falls outside of the problem domain is discarded. When the expansion stops, each processor has an identical copy of an initial octree.

Task assignment. For mesh generation purpose, a *task* corresponds to the refinement of a leaf octant in the initial octree. Using the preorder traversal as the total ordering, we perform a parallel block data decomposition of the leaf octants. Since all of the processors have complete information about the initial octree, no communication is required during this step. We assign to each processor an equal number of leaf octants to refine, marking those assigned as *local* and the others as *remote*.

Tree pruning. On each processor, we prune octants marked as *remote* from the initial octree using the aggregation procedure described in Section 5.1.2. On completion, each processor has its local instance of the initial (global) octree.

One caveat: because the refinements of the cubic areas (leaf octants) are determined by an application, the workload of each task can be highly uneven, for example, due to heterogeneity within the problem domain. There is no way to determine an optimal task assignment scheme. The best we can do is to assign an approximately equal number of tasks to each processor.

5.2.2. REFINETREE/COARSENTREE. In the REFINETREE step, each processor traverses the leaf octants of its local instance. On visiting every octant marked as *local*, we invoke an application-supplied callback function to determine whether the octant needs to be decomposed. In case a decomposition is triggered, we create eight new children octants and link them into the tree structure. In addition, we also allocate extra internal memory for each new octant to store application-specific meshing data such as material density or velocity. To set the initial value properly, we invoke another application-specific callback function.

The COARSENTREE step follows the reverse procedure. However, coarsening cannot be carried out if the eight children of an interior octant are spread over across two processors. See Figure 5.4(a) for an example. In this case, we skip the coarsening of that particular region. Since it occurs only on the boundary between

adjacent processors, most internal coarsening can still be performed without problem. Further, since the mesh is likely to be re-partitioned after the REFINETREE/COARSENTREE step, those octants spread over a boundary are likely to be re-distributed to the same processor. Then in the next round of coarsening, they can be successfully merged.

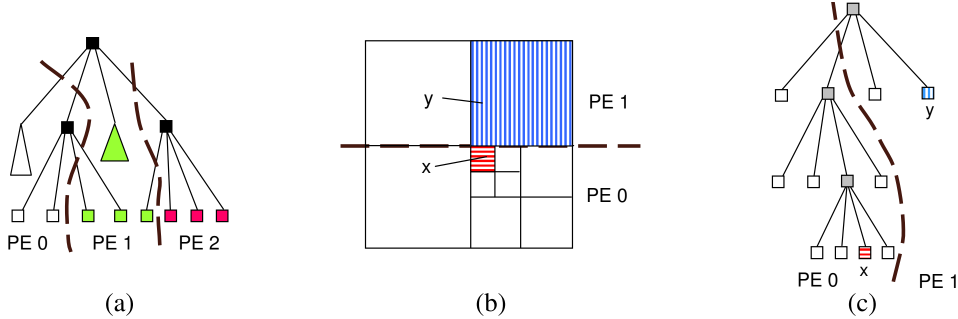


FIG. 5.4. *Operations across processor boundaries. The dashed lines represent the boundaries between processors. (a) Coarsening is skipped on processor boundaries. (b) A violation of the balance condition. Octant x and y are adjacent to each other spatially but differ in size by a factor of 4. (c) The two spatially adjacent octants (x and y) are distributed on two processors.*

5.2.3. BALANCETREE. *Prioritized ripple propagation* is an algorithm we previously proposed in [34] to enforce the 2-to-1 constraint in a mesh database. The basic idea of the algorithm is to eliminate the effect of small octants on large octants level by level. The algorithm first visits all leaf octants at the lowest level of an octree. For each leaf octant, a series of neighbor-findings are carried out. Any neighboring leaf octants that are more than twice as large as the current octant are decomposed repeatedly until the 2-to-1 constraint is satisfied. When all the leaf octants on the lowest octree level are processed, we finish one iteration and moves one level up to start another iteration (to process leaf octants at a higher level). The algorithm terminates when all the levels are processed. The neighbor-finding operation is implemented using a variant of the standard pointer-based neighbor-finding algorithm [29].

The above algorithm works well on a single processor but would run into trouble on a multiprocessor. Figures 5.4(b) and 5.4(c) illustrate the problem. If octant x and y are spatially adjacent to each other but are distributed on PE 0 and PE 1, respectively, then when x searches its neighbor in the north direction, the neighbor-finding algorithm will get stuck.

The *parallel prioritized ripple propagation* algorithm overcomes this difficulty in the following way. When the standard neighbor-finding algorithm gets stuck, we search the locational code lookup table to find the id of the remote processor that hosts the neighbor. Instead of forwarding the ongoing neighbor-finding operation to the remote processor immediately, we create a request (containing the information of the current octants as well as partial tree-traversal information) and store it a *request buffer* that is destined for the remote processor. When all processors finish processing their local octants at the current level, the request buffers are exchanged between processors. Each processor then services the neighbor-finding requests from other processors and decomposes its large octants as necessary. When all of the processors finish servicing remote requests, we proceed to the next higher octree level. The key implementation technique of the parallel prioritized ripple propagation algorithm is a concise but expressive encoding scheme for the neighbor-finding requests.

5.2.4. PARTITIONTREE. This step creates a new partition for the parallel octree and then migrates octants among processors according to the partition.

The partition is computed in a very simple way. We sort all the leaf octants in Z-order, split them into equal-size chunks and assign the ith chunk to the ith processor. Since the preorder traversal of octree leaves is the same as Z-order, there is no need to physically sort the leaf octants. All the leaf octants on processor i must be sorted in between the leaf octants on processor $(i-1)$ and the leaf octants on processor $(i+1)$. See Figure 5.4(a) for an example. Therefore, the only information needed to compute a balanced partition is the number of leaf octants on each processor. After a few reduction-type communications, each processor receives the information it needs to compute the partition.

The more difficult part of PARTITIONTREE step is actually migrating octants between processors. We have developed an internal memory manager for Octor to facilitate this procedure. As mentioned earlier, when octants are created in the refinement process, internal memory is allocated to hold the application-specific data such as velocity or density. While executing the PARTITIONTREE step, the Octor internal memory manager marshals the data that needs to be migrated and sends it to the destination processor in bulk. The unused memory on the sending processor is immediately reclaimed. On the receiving end, the Octor memory manager installs the newly arrived leaf octants in the local octree, and allocates extra memory space to store any application-specific meshing data. There are two advantages of relying on the Octor internal memory manager. First, the function call interface for PARTITIONTREE is simple to use (see Appendix). Second, Octor is more robust, encapsulating and hiding complex memory management issues from the application.

A side note: an interesting design feature of Octor is that we have chosen to partition the tree *before* extracting the mesh structure. By migrating octants to their destinations first, each processor can obtain all necessary information for extracting the mesh structures for the simulation or visualization code that is going run on that processor.

5.2.5. EXTRACTMESH. As mentioned earlier, parallel octree meshing is fundamentally different from other parallel octree applications because there are complicated operations associated with the vertices of the octants, i.e., the mesh nodes.

All of the complexities of handling mesh nodes and other mesh data structures are encapsulated in the EXTRACTMESH step, which performs the following operations: (1) extracting mesh nodes from the parallel (balanced) octree and computing their coordinates; (2) discovering dangling nodes; (3) assigning each element and each node to an owner processor; (4) establishing sharing relationships between nodes on different processors; (5) allocating unique *global* element ids and node ids; (6) assigning per-processor *local* element ids and node ids, (7) establishing the mapping between global ids with local ids; (8) correlating local element ids with local node ids (mesh connectivity information); and (9) correlating local dangling node ids with local anchored node ids. Octor implements these operations in a tightly coupled fashion in order to reduce communication cost and algorithm complexity. The rest of this section focuses on three interesting issues that will shed some light on the complexity unique to parallel octree meshing.

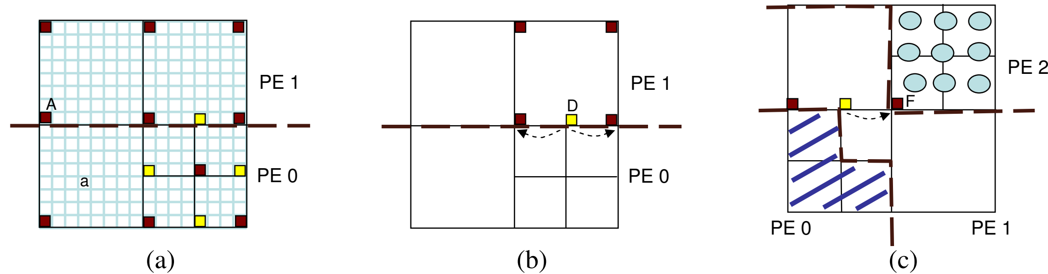


FIG. 5.5. *How to extract mesh nodes and identify indirect sharing. (a) Mesh nodes are mapped to the smallest octants in the domain and assigned to processors that cover the respective SFC segments. (b) Subtle sharing case 1: PE 1 is assigned the ownership of a dangling node but does not use it. (c) Subtle sharing case 2: The patterns in the figure illustrate the partitioning of the mesh. PE 0 indirectly uses an anchored node that is on PE 2.*

Determining the ownership of mesh nodes. Finite elements with Lagrange bases associate unknowns to mesh nodes. If two or more elements sharing a node are distributed on different processors, then each element (and thus its processor) will make some contribution to the values associated with the shared node. The question is which processor will accumulate the value contributions that are distributed among different processors? A commonly used method is to assign each node to an owner processor who will become responsible for all the computation related to a node. A naive implementation would let all the processors sharing that node enter a negotiation process and somehow select a processor as the owner. Our implementation employs a different technique. The basic idea is to treat each mesh node as a smallest octant in the domain. Then each mesh node must fall in the area covered by some processor, as shown in Figure 5.5(a). The processor that encloses the mesh node is assigned as the owner. This procedure can be easily implemented, using the replicated locational code lookup table, with no interprocessor communications.

Determining the processors that share a mesh node. The dual problem of assigning a node to an owner processor is to determine which other processors share a particular node. From the standpoint of an owner processor of a particular node, sharing means that a remote processor needs to access the values associated with this node. A simple case of sharing is shown in Figure 5.5(a). The mesh node marked `A` is owned by PE 1. PE 0 has element `a` that will contribute to the values associated with A. Thus, PE 1 and PE 0 should both expect to exchange data with each other. However, there are more subtle cases of sharing. Figure 5.5(b) shows a case where PE 1 does not know the existence of the dangling node `D` because none of the octants on PE 1 has D as a vertex. However, on PE 0, node D is indeed a vertex of some octant and is produced as a mesh node. Using the node ownership assignment algorithm just described, PE 0 determines PE 1 is the owner of node D and will send data associated with node D to PE 1. Now a discrepancy in communication occurs. Another subtle case is illustrated in Figure 5.5(c). PE 0 has a dangling node that is dependent on node `F`, which is owned by PE2. So PE 0 needs to make use of the values associated with `F`. But PE 2 would not be able to determine that PE 0 is sharing `F` with it. Note that PE 0 and PE 2 are not adjacent to each other in the partition. To resolve such subtle sharing situations, we use a single round of all-to-all communication. Fortunately, since the subtle sharing cases only occur when dangling nodes are involved in the processor boundary area, they are much less common than the simple cases.

Returning a mesh structure that can be directly used by a solver. Since we are already extracting mesh structures on the same processors where solvers will be running, we have gone one step further to extract a mesh structure that can be directly used by finite element solvers. By *directly*, we mean that when a handle to the local mesh is returned to an application, a solver can immediately build its communication schedule without incurring any communication. Operations (6)—(9), listed earlier in this section, accomplish this task on a solver's behalf. The advantage of carrying out all these mapping operations within Octor is that we can make efficient use of the various internal data structures built during the EXTRACTMESH step. For example, we use the same node hash table repeatedly for various operations such as discovering dangling nodes and correlating mesh elements to mesh nodes.

6. Performance evaluation. In this section, we present a performance evaluation of Octor for statically generating large unstructured finite element meshes.[1] The meshes are generated for modeling earthquake simulations, and are directly used by an explicit seismic wave-propagation solver. The target region of these simulations is the Greater Los Angeles Basin, which comprises a three-dimensional volume of 100 km x 100 km x 37.5 km. The material model we used to drive the mesh generation process is the Southern California Earthquake Center (SCEC) 3D velocity model [25] (Version 3, 2002). A sequence of meshes is generated to satisfy different simulation frequency requirements. Roughly speaking, the higher the frequency, the finer (larger) the mesh. Performance data presented in this section were obtained from production and experimentation runs of earthquake simulations on Lemieux, the HP AlphaServer system at Pittsburgh Supercomputing Center, on a number of processors ranging from 1 to 2000.

Our performance evaluation focuses on two issues: (1) The isogranular scalability [33] of Octor. That is, how scalable is Octor when we generate larger meshes on more processors while keeping the average number of elements on each processor more or less the same? (2) The fixed-size scalability of Octor. That is, how scalable is Octor when we fix the problem size and increase the number of processors?

6.1. Isogranular scalability study. Figure 6.1 summarizes the results of generating meshes of various sizes on different number of processors. Since the earth is highly heterogeneous, the largest elements of the meshes are 64 times as large in edge size as the smallest ones (the difference between "max leaf level" and "min leaf level"). Also, because of the multi-resolution in the mesh, there are a large number of dangling nodes in the mesh (between 11% and 20%).

The larger meshes (1 Hz, 1.5 Hz, and 2 Hz) are for terascale finite element simulations that run on thousands of processors [3]. However, for the purpose of this study, we have chosen the numbers of processors to ensure that the memory utilization per processor across the different experiments is roughly the same. Given the unstructured nature of the meshes, it is impossible to guarantee the per-processor element (node) number to be exactly the same over different runs. Nevertheless, we have contained the difference to within

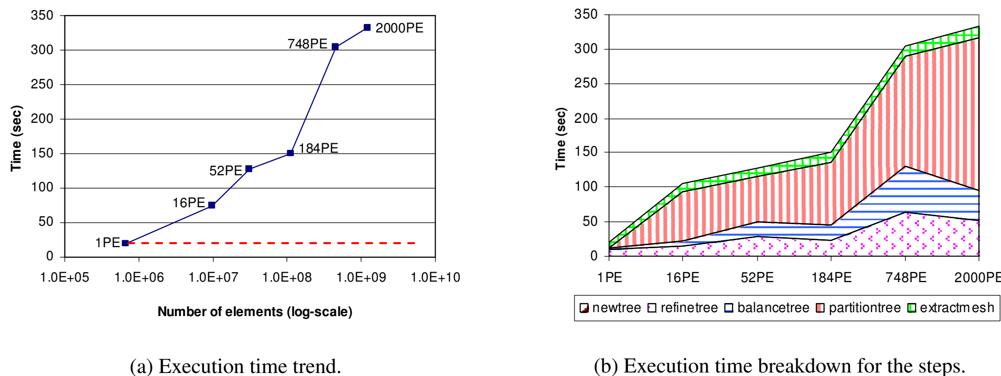
[1] We have not yet evaluated the performance of Octor for dynamic meshing applications. Thus, the performance of COARSENTREE is not presented.

10% (as shown on the “Element/PE” row). The smallest mesh (0.23 Hz) has 661K elements and is generated on one processor, while the largest one (2 Hz) has 1.22B elements and is generated on 2000 processors.

PEs	1	16	52	184	748	2000
Mesh name	0.23 Hz	0.5 Hz	0.75 Hz	1 Hz	1.5 Hz	2 Hz
Elements	*6.61E+5*	*9.92E+6*	*3.13E+7*	1.14E+8	4.62E+8	1.22E+9
Nodes	8.11E+5	1.13E+7	3.57E+7	1.34E+8	5.34E+8	1.37E+9
Anchored	6.48E+5	9.87E+6	3.12E+7	1.14E+8	4.61E+8	1.22E+9
Dangling	1.63E+5	1.44E+6	4.57+6	2.03E+7	7.32E+7	1.48+8
Max leaf level	11	13	13	14	14	15
Min leaf level	6	7	8	8	9	9
Elements/PE	6.61E+5	6.20E+5	6.02E+5	6.20E+5	6.18E+5	6.12E+5
Octor time(sec)	19.98	74.79	127.95	150.26	305.35	332.85

FIG. 6.1. *Summary of the meshes of different resolutions for earthquake wave-propagation simulations and the time to generate them on different number of processors.*

Figure 6.2(a) shows how the Octor meshing time increases as the problem size increases (along with the number of processors used). Note that the horizontal axis (problem size) is in log-scale, while the vertical axis (time) is in linear scale. It can be seen that, roughly, the meshing time increases as just a logarithm of problem size. As shown in Figure 6.2(b), the increased running time is mostly due to the PARTITIONTREE step. Although Octor does not achieve the theoretical optimal isogranular performance (the dashed line), the linear meshing time trend suggests that Octor has good isogranular scalability. We can speculate that if Octor were to be used to mesh a problem 10 times as large as the 2 Hz mesh on 20,000 processors, then the meshing time would only increase moderately. Here is another way to appreciate the efficiency of Octor: Assume that a 2 Hz mesh (80 GB in size) already exists on a lab server and that we have to move it across the network to a supercomputer. Transferring the mesh at peak rates over a gigabit ethernet connection would require more than 10 minutes. In comparison, Octor takes less than 6 minutes to generate the mesh from scratch *in situ*.



(a) Execution time trend.

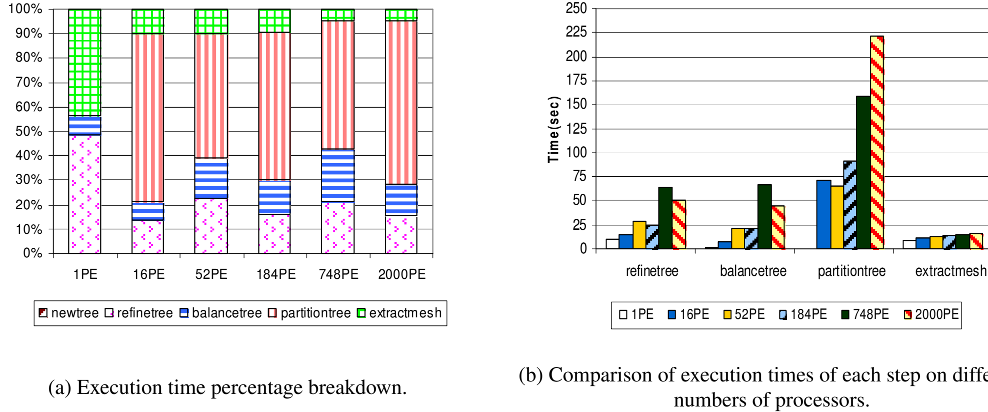
(b) Execution time breakdown for the steps.

FIG. 6.2. *Octor meshing execution time and detailed breakdown. (a) The running time of Octor increases roughly as a logarithm of problem size (as determined by the total number of elements). (b) The trend of execution time increases in different steps. The time for each step is plotted with a unique pattern, which is stacked on top of the previous step. From bottom up, the stripes represent time spent in the* NEWTREE, REFINETREE, BALANCETREE, PARTITIONTREE, *and* EXTRACTMESH *step, respectively. Note that since the* NEWTREE *time is very short (less than 0.5 sec), its stripe is invisible in the figure.*

We now examine the running time of different meshing steps in more detail for the isogranular runs. Figure 6.3(a) shows the execution time breakdown for generating the different meshes (annotated by the number of processors used). Each bar represents the contribution of the five meshing steps as a percentage of the total execution time. From the fact that percentage breakdowns (except for the single-PE run that does not invoke PARTITIONTREE) do not change significantly between different runs, we can deduce that none of the meshing steps is a scalability bottleneck. Otherwise, we would have seen disproportionately large fluctuations in the percentage breakdowns. We can also see that the dominant cost of meshing is associated with PARTITIONTREE, which is probably the most bandwidth (memory and network) intensive operation. However, further research is needed to quantitatively determine the reason.

Figure 6.3(b) shows the running time of the meshing steps in a different way. We group together the absolute running times of each step for different runs and compare the running time change of each individual

step over different problem sizes. Since the NEWTREE times are below 0.5 seconds for all the runs, we have not presented them as a group in the figure. It is clear that REFINETREE, BALANCETREE, and EXTRACTMESH perform well in an isogranular setting, while the performance of PARTITIONTREE deteriorates somewhat with larger numbers of processors. It is surprising that EXTRACTMESH, which is the most complex step in terms of algorithms and communications, runs faster and scales better than all of the other steps except for the relatively simple NEWTREE step. This is strong evidence that our techniques for extracting mesh structures are effective and efficient.



(a) Execution time percentage breakdown.

(b) Comparison of execution times of each step on different numbers of processors.

FIG. 6.3. *Understanding the performance of Octor from two different perspectives. Figure (a) shows where the execution time goes. Figure (b) shows how each step's running time varies on different numbers of processors.*

6.2. Fixed-size scalability study. For applications that are computation bound, larger numbers of processors might be used to solve a moderately-sized problem just to improve the turn-around time. Memory utilization is not of particular importance to these applications. Therefore, we also need to investigate the performance of Octor when we fix the problem size and increase the number of processors.

Three sets of fixed-size scalability experiments were conducted, for small size, medium size and large size problems, respectively. The experimental setups are shown in Figure 6.4.

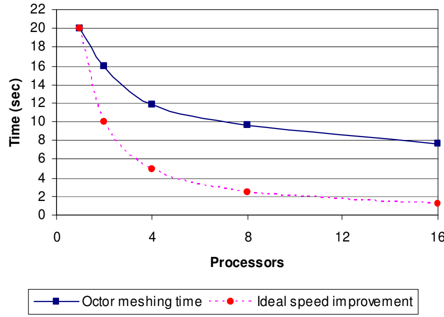
PEs	1	2	4	8	16	32	64	128	256	512	748	1024	2000
Small case (0.23 Hz, 661K elements)	x	x	x	x	x								
Medium case (0.5 Hz, 9.92M elements)				x	x	x	x	x					
Large case (1 Hz, 114M elements)								x	x	x	x	x	x

FIG. 6.4. *Setup of fixed-size speedup experiments. Entries marked with "x" represent experiment runs.*

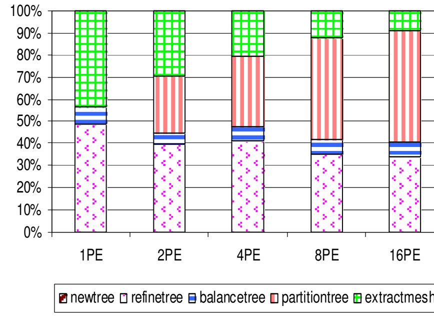
Figure 6.5 shows the performance of Octor for different fixed-sized problems. The figures on the left side shows the speedup of meshing on more processors. The dashed line represents the ideal speedup. Although not perfect, Octor achieves good speedups while running on a large number of processors. The figures on the right show the execution time percentage breakdowns for different fixed-sized runs on different numbers of processors. That there is no major fluctuation in the breakdowns (except for the small case problem where 1 PE is involved) is more evidence that there are no scalability bugs among the meshing steps.

An important and closely related question is what happens to a solver when a fixed-sized problem is meshed on larger numbers of processors? Figure 6.6 shows the speedup of an explicit finite element seismic wave propagation solver for earthquake simulation. The solver takes a mesh produced by Octor *in situ* and runs on the same processors. The solver runs 4,000 time steps for the medium case (Figure 6.6(a)) and 2,500 time steps for large case. Each time step requires the same amount work for a fixed-size problem. A complete earthquake simulation would run somewhere between 10,000 to 80,000 time steps.

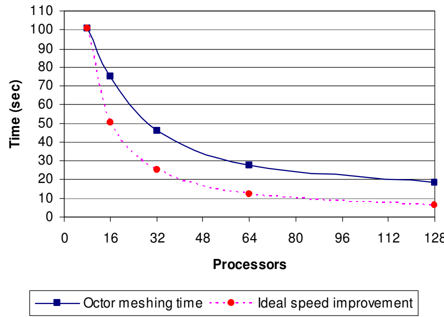
A surprising finding from these figures is that the solver achieves almost perfect speedup on hundreds and thousands of processors, even though the strategy we used to partition the tree is extremely simple. Also, a closer look at the vertical axes of Figure 6.5(c)(e) and Figure 6.6(a)(b) reveals that the meshing time becomes inconsequential when compared with the running time of the solver. So the bottom line is that as



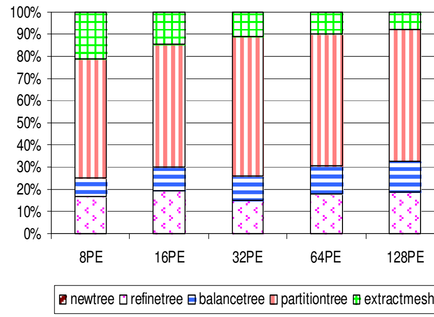
(a) Small case meshing speedup.



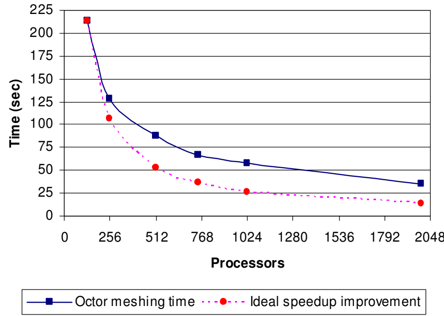
(b) Small case running time percentage breakdown.



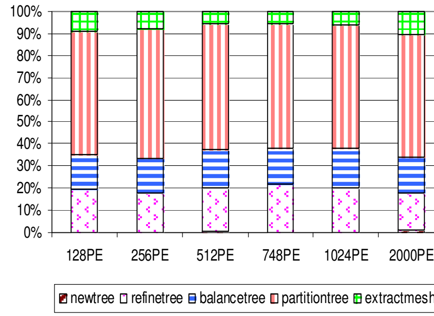
(c) Medium case meshing speedup.



(d) Medium case running time percentage breakdown.



(e) Large case meshing speedup.



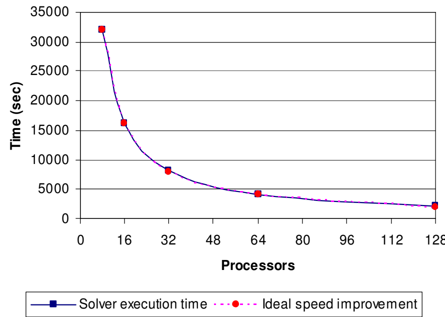
(f) Large case running time percentage breakdown.

FIG. 6.5. *Execution time of generating a fixed-sized mesh on different numbers of processors and how different steps of meshing contribute to the total running time percentage wise.*

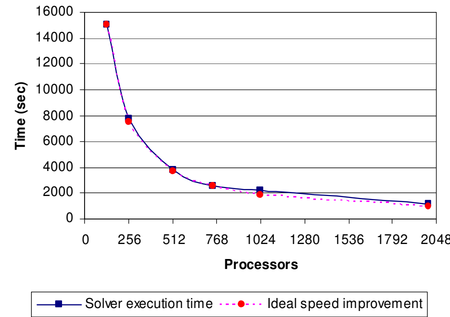
long as a physical problem can be meshed in a reasonable amount of time, Octor meshing will not constitute a bottleneck for terascale simulations.

7. Conclusion. This paper has demonstrated how to combine existing, well-known techniques, such as parallel octrees and space-filling curves, with a new set of algorithms and methods, such as parallel prioritized ripple propagation, to deliver a new capability for generating and adapting massive octree meshes for high-performance scientific computing.

We have implemented these new ideas in Octor, a parallel octree meshing tool that has been used successfully for terascale finite element earthquake simulations. The good scalability of Octor on up to 2000 processors indicates that it has the potential to continue to scale on more processors. Future work will focus on applying Octor in the context of terascale applications with adaptive meshing requirements.



(a) Medium case solver speedup.



(b) Large case solver speedup.

FIG. 6.6. *Solver achieves almost perfect speedup using the partitioned meshes generated by Octor. (We have omitted the plotting for the small case, which looks identical to that of the medium case.)*

Acknowledgments. This work is sponsored in part by NSF under grant IIS-0429334, in part by a subcontract from the Southern California Earthquake Center as part of NSF ITR EAR-0122464, in part by NSF under grant EAR-0326449, in part by DOE under the SciDAC TOPS project, and in part by a grant from Intel. Supercomputing time at the Pittsburgh Supercomputing Center is supported under NSF TeraGrid grant MCA04N026P. We would like to thank our Quake project colleagues Jacobo Bielak and Leonardo Ramirez-Guzman, and our SCEC CME partners Tom Jordan and Phil Maechling, for providing such a compelling application context. Thanks also to Hongfeng Yu and Kwan-Liu Ma for helping us understand the data access requirements of parallel visualization packages. Special thanks to John Urbanic, Chad Vizino, and Sergiu Sanielevici at PSC for their outstanding technical support.

Appendix. Major functions of the Octor API.

```
octree_t * octor_newtree(double x, double y, double z, int recsize, int myid, int groupsize);
int octor_refinetree(octree_t *octree, toexpand_t *toexpand, setrec_t *setrec);
int octor_coarsentree(octree_t *octree, toshrink_t *toshrink, setrec_t *setrec);
int octor_balancetree(octree_t *octree, setrec_t *setrec);
int octor_partitiontree(octree_t *octree);
mesh_t * octor_extractmesh(octree_t *octree);
void octor_deletemesh(mesh_t *mesh);
void octor_deletetree(octree_t * octree);
```

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 (1983), pp. 1–13.

[2] T. ABEL, G. L. BRYAN, AND M. L. NORMAN, *The formation and fragmentation of primordial molecular clouds*, Astron. Astrophys., (2000), pp. 39–44.

[3] V. AKCELIK, J. BIELAK, G. BIROS, I. IPANOMERITAKIS, ANTONIO FERNANDEZ, O. GHATTAS, E. KIM, J. LOPEZ, D. R. O'HALLARON, T. TU, AND J. URBANIC, *High resolution forward and inverse earthquake modeling on terasacale computers*, in SC2003, Phoenix, AZ, Nov. 2003. Gordon Bell Award for Special Achievement.

[4] S. ALURU, G. M. PRABHU, AND J. GUSTAFSON, *Truly distribution-independent algorithms for the n-body problem*, in SC1994, Washington, D.C., 1994.

[5] S. ALURU AND F. E. SEVILGEN, *Parallel domain decomposition and load balancing using space-filling curves*, in Proceedings of the 4th IEEE Conference on High Performance Computing, 1997.

[6] M. BERN, D. EPPSTEIN, AND J. GILBERT, *Provably good mesh generation*, in Proceedings of 31st Symposium on Foundation of Computer Science, 1990, pp. 231–241.

[7] M. BERN, D. EPPSTEIN, AND S. TENG, *Parallel construction of quadtrees and quality triangulations*, International Journal of Computational Geometry and Applications, 9 (1999), pp. 517–532.

[8] D. K. BLANDFORD, G. E. BLELLOCH, AND C. KADOW, *Compact parallel Delaunay tetrahedralization*. Submitted to SC2005.

[9] A. CALDER, B. C. CURTIS, J. DURSI, B. FRYXELL, G. HENRY, P. MACNEICE, K. OLSON, P. RICKER, R. ROSNER, F. TIMMES, H. TUFO, J. TRURAN, AND M. ZINGALE, *High-performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors*, in Proceedings of SC2000, 2000. Gordon Bell Award for Special Achievement.

[10] P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Teresco, *Dyanmic octree load balancing using space-filling curves*, Tech. Report CS-03-01, Department of Computer Science, Williams College, jan 2003.

[11] N. Chrisochoides, *Parallel mesh generation*, in Numerical Solution of Partial Differential Equations on Parallel Computers, M. Bruaset, P. Bjorstad, and A. Tveit, eds., Springer, 2005. to appear.

[12] N. Chrisochoides and D. Nave, *Parallel Delaunay mesh generation kernel*, International Journal for Numerical Methods in Engineering, 1 (2001).

[13] ———, *Parallel Delaunay mesh generation kernel*, Int. J. Num. Methods in Engineering, 58 (2003), pp. 161–176.

[14] H. L. De Cougny and M. S. Shephard, *Parallel refinement and coarsening of tetrahedral meshes*, International Journal for Numerical Methods in Engineering, 46 (1999), pp. 1101–1125.

[15] J. M. Dennis, *Partitioning with space-filling curves on the cubed-sphere*, in Proceedings of Workshop on Massively Parallel Processing at IPDPS'03, Nice, France, 2003.

[16] H. C. Edwards and J. C. Browne, *Scalable dynamic distributed arrays and its application to parallel hp adaptive finite element code*, in Proceedings of POOMA '96, Santa Fe, New Mexico, 1996.

[17] C. Faloutsos and S. Roseman, *Fractals for secondary key retrieval*, in Proceedings of the Eighth ACM SIGACT-SIGMID-SIGART Symposium on Principles of Database Systems (PODS), 1989.

[18] J. E. Flaherty, R. Loy, C. Ozturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Parallel structures and dynamic load balancing for adaptive finite element computation*, Applied Numerical Mathematics, (1998).

[19] I. Gargantini, *An effecive way to represent quadtrees*, Communicatoins of the ACM, 25 (1982), pp. 905–910.

[20] ———, *Linear octree for fast processing of three-dimensional objects*, Computer Graphics,and Image Processing, 20 (1982), pp. 365–374.

[21] C. Kadow and N. J. Walkington, *Adaptive dynamic projection-based partitioning for parallel Delaunay mesh generation and refinement*, in SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, Feb 2004.

[22] G. Karypis and V. Kumar, *A course-grain parallel formulation of multi-level k-way graph partitioning algorithm*, in 8th SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[23] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings, *Predictive performance and scalability modeling of a large-scale application*, in SC2001, 2001.

[24] 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.

[25] H. Magistrale, S. Day, R. Clayton, and R. Graves, *The SCEC Southern California reference three-dimensional seismic velocity model version 2*, Bulletin of the Seismological Soceity of America, (2000).

[26] S. A. Mitchell and S. A. Vavasis, *Quality mesh generation in three dimensions*, in Proceedings of the 8th ACM Symposium on Computational Geometry, 1992, pp. 212–221.

[27] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*, tech. report, IBM, Ottawa, Canada, 1966.

[28] A. K. Patra, A. Laszloffy, and J. Long, *Data structures and load balancing for parallel adaptive hp finite-element methods*, International Journal on Computers and Mathematics with Applications, 46 (2003), pp. 105–123.

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

[30] R. Schneiders, *An algorithm for the generation of hexahedral element meshes based on an octree technique*, in Proceedings of 6th International Meshing Roundtable, 1997.

[31] M. S. Shephard and M. K. Georges, *Automatic three-dimensional mesh generation by the finite octree technique*, International Journal for Numerical Methods in Engieering, 32 (1991).

[32] J. P. Singh, C. Holt, J. L. Hennessy, and Anoop Gupta, *A parallel adaptive fast multipole method*, in SC1993, Portland, OR, 1993.

[33] X. Sun and L. Ni, *Scalable problems and memory-bounded speedup*, Journal of Parallel and Distributed Computing, 19 (1993), pp. 27 – 37.

[34] T. Tu and D. R. O'Hallaron, *A computational database system for generating unstructured hexahedral meshes with billions of elements*, in SC2004, Pittsburgh, PA, Nov. 2004.

[35] T. Tu, D. R. O'Hallaron, and J. Lopez, *Etree – a database-oriented method for generating large octree meshes*, in Proceedings of the Eleventh International Meshing Roundtable, Ithaca, NY, Sept. 2002, pp. 127– 138. Also in Engineering with Computers (2004) 20:117–128.

[36] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K. Ma, and D. R. O'Hallaron, *From physical model to scientific understanding: An end-to-end approach to parallel supercomputing*. Working paper.

[37] M. S. Warren and J. K. Salmon, *A parallel hashed oct-tree n-body algorithm*, in SC1993, Portland, OR, 1993.

[38] A. Wissink, R. Hornung, S. Kohn, S. Smith, and N. Elliott, *Large scale parallel structured AMR calculations using the SAMRAI framework*, in SC2001, Denver, CO, 2001.

[39] L. Ying, G. Biros, D. Zorin, and H. Langston, *A new parallel kernel-independent fast multipole method*, in SC2003, Phoenix, AZ, 2003.

[40] 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 (1991), pp. 1–66.