From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing

Tiankai Tu[*] Hongfeng Yu[†] Leonardo Ramirez-Guzman[‡] Jacobo Bielak[§]
Omar Ghattas[¶] Kwan-Liu Ma[||] David R. O'Hallaron[**]

Abstract

Parallel supercomputing has typically focused on the inner kernel of scientific simulations: the solver. The front and back ends of the simulation pipeline—problem description and interpretation of the output—have taken a back seat to the solver when it comes to attention paid to scalability and performance, and are often relegated to offline, sequential computation. As the largest simulations move beyond the realm of the terascale and into the petascale, this decomposition in tasks and platforms becomes increasingly untenable. We propose an end-to-end approach in which *all simulation components*—meshing, partitioning, solver, and visualization—are tightly coupled and execute in parallel with shared data structures and no intermediate I/O. We present our implementation of this new approach in the context of octree-based finite element simulation of earthquake ground motion. Performance evaluation on up to 2048 processors demonstrates the ability of the end-to-end approach to overcome the scalability bottlenecks of the traditional approach.

1 Introduction

The traditional focus of parallel supercomputing has been on the inner kernel of scientific simulations: the *solver*, a term we use generically to refer to solution of (numerical approximations of) the governing partial differential, ordinary differential, algebraic, integral, or particle equations. Great effort has gone into the design, evaluation, and performance optimization of scalable parallel solvers, and previous Gordon Bell awards have recognized these achievements. However, the front and back ends of the simulation pipeline—problem description and interpretation of the output—have taken a back seat to the solver when it comes to attention paid to scalability and performance. This of course makes sense: solvers are usually the most cycle-consuming component, which makes them a natural target for performance optimization efforts for successive generations of parallel architecture. The front and back ends, on the other hand, often have sufficiently small memory footprints and compute requirements that they can be relegated to offline, sequential computation.

However as scientific simulations move beyond the realm of the terascale and into the petascale, this decomposition in tasks and platforms becomes increasingly untenable. In particular, multiscale three-dimensional PDE simulations often require variable-resolution unstructured meshes to efficiently resolve the different scales of behavior. The problem description phase can then require generation of a massive unstructured mesh; the output interpretation phase then involves unstructured-mesh volume rendering of even larger size. As the largest unstructured mesh simulations move into the hundred million to billion element range, the memory and compute requirements for mesh generation and volume rendering preclude the use of sequential computers. On the other hand, scalable parallel algorithms and implementations for large-scale mesh generation and unstructured mesh volume visualization are significantly more difficult than their sequential counterparts.[1]

We have been working over the last several years to develop methods to address some of these front-end and back-end performance bottlenecks, and have deployed them in support of large-scale simulations of earthquakes [3]. For the front end, we have developed a computational database system that can be used to generate unstructured hexahedral octree-based meshes with billions of elements on workstations with sufficiently large disks [26, 27, 28, 30]. For the back end, we have developed special I/O strategies that effectively hide I/O costs when transferring individual time step data to memory for rendering calculations [32],

[*]Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, tutk@cs.cmu.edu

[†]Department of Computer Science, University of California, Davis, CA 95616, hfyu@ucdavis.edu

[‡]Department of Civil and Environmental Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, lramirez@andrew.cmu.edu

[§]Department of Civil and Environmental Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, jbielak@andrew.cmu.edu

[¶]Institute for Computational Engineering and Sciences, Jackson School of Geosciences, and Departments of Mechanical Engineering, Biomedical Engineering and Computer Sciences, The University of Texas at Austin, Austin, TX 78077, omar@ices.utexas.edu

[||]Department of Computer Science, University of California, Davis, CA 95616, ma@cs.ucdavis.edu

[**]Computer Science Department and Department of Electrical and Computer Engineering, Carnegie Mellon University, PA 15213, droh@cs.cmu.edu

[1]For example, in a report identifying the prospects of scalability of a variety of parallel algorithms to petascale architectures [22], mesh generation and associated load balancing are categorized as Class 2—"scalable provided significant research challenges are overcome."

which themselves run in parallel and are highly scalable [16, 17, 19, 32]. Figure 1 illustrates the simulation pipeline in the context of our earthquake modeling problem, and, in particular, the sequence of files that are read and written between components.

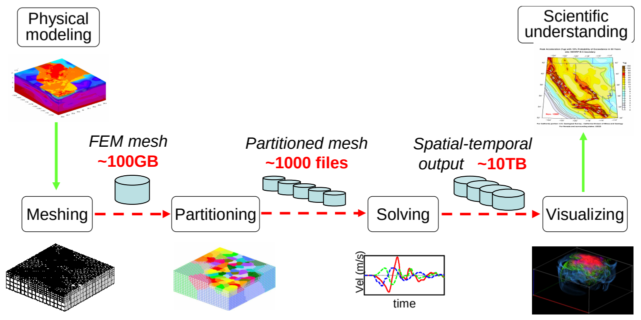


Figure 1: **Traditional simulation pipeline.**

However, despite our best efforts at devising scalable algorithms and implementations for the meshing, solver, and visualization components, as our resolution and fidelity requirements have grown to target hundred million to multi-billion element simulations, significant bottlenecks remain in storing, transferring, and reading/writing multi-terabyte files between these components. In particular, I/O of multi-terabyte files remains a pervasive performance bottleneck on parallel computers, to the extent that *the offline approach to the meshing–partitioning–solver–visualization simulation pipeline becomes intractable for billion-unknown unstructured mesh simulations.* Ultimately, beyond scalability and I/O concerns, the biggest limitation of the offline approach is its inability to support interactive visualization of the simulation: the ability to debug and monitor the simulation at runtime based on volume-rendered visualizations becomes increasingly crucial as problem size increases.

Thus, we are led to conclude that in order to (1) deliver necessary performance, scalability, and portability for ultrascale unstructured mesh computations, (2) avoid unnecessary bottlenecks associated with multi-terabyte I/O, and (3) support runtime visualization steering, we must seek an *end-to-end solution* to the meshing–partitioning–solver–visualization parallel simulation pipeline. The key idea is to replace the traditional, cumbersome file interface with a scalable, parallel, runtime system that supports the simulation pipeline in two ways: (1) providing a common foundation on top of which all simulation components operate, and (2) serving as a medium for sharing data among simulation components.

Following this design principle, we have implemented a simulation system named *Hercules* that targets unstructured octree-based finite element PDE simulations running on multi-thousand processor supercomputers. Figure 2 illustrates the new computing method. All simulation components (i.e. meshing, partitioning, solver, and visualization) are implemented on top of, and operate on, a unified parallel octree data structure. There is only one executable (MPI code). All components are tightly coupled and execute on the same set of processors. The only inputs are a description of the spatial variation of the PDE coefficients (a material property database for a 3D domain of interest), a simulation specification (earthquake source definition, maximum frequency to be resolved, mesh nodes per wavelength, etc.), and a visualization configuration (image resolution, transfer function, view point, etc.); the only outputs are lightweight jpeg-formatted image frames generated *as the simulation runs*.[2] There is no other file I/O.

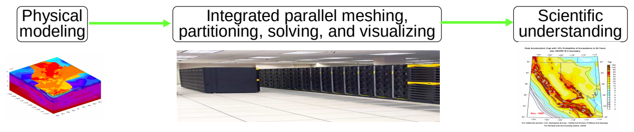

Physical modeling
Integrated parallel meshing, partitioning, solving, and visualizing
Scientific understanding

Figure 2: **Online, end-to-end simulation pipeline.**

The relative simplicity of the parallel octree structure has certainly facilitated the implementation of Hercules. Nevertheless, additional mechanisms on top of the tree structure are needed to support different scalable algorithms within the Hercules framework. In particular, we need to associate unknowns with mesh nodes, which correspond to the vertices of the octants in a parallel octree.[3] The problem of how to handle octree mesh nodes alone represents a nontrivial challenge to meshing and solving. Furthermore, in order to provide unified data access services throughout the simulation pipeline, a flexible interface to the underlying parallel octree has to be designed and exported such that all components can efficiently share simulation data.

It is worth noting that while the only post-processing component we have incorporated in Hercules is volume rendering visualization, there should be no technical difficulty in adding other components. We have chosen 3D volume rendering over others mainly because it is one of the most demanding back ends in terms of algorithm complexity and difficulty of scalability. By demonstrating that online, integrated, highly parallel visualization is achievable, we establish the viability of the proposed end-to-end approach and argue that it can be implemented for a wide variety of other simulation pipeline configurations.

We have assessed the performance of Hercules on the Alpha EV68-based terascale system at the Pittsburgh Supercomputing Center for modeling earthquake ground motion in heterogeneous basins. Preliminary performance and scalability results (Section 4) show:

- Fixed-size scalability of the entire end-to-end simulation pipeline from 128 to 2048 processors at 64%

[2]Optionally, we can write out the volume solution at each time step if necessary for future post-processing—though we are rarely interested in preserving the entire volume of output, and instead prefer to operate on it directly *in-situ*.

[3]In contrast, other parallel octree-based applications such as N-body simulations do not need to manipulate octants' vertices.

overall parallel efficiency for 134 million mesh node simulations

- Isogranular scalability of the entire end-to-end simulation pipeline from 1 to 748 processors at combined 81% parallel efficiency for 534 million mesh node simulations
- Isogranular scalability of the meshing, partitioning, and solver components at 60% parallel efficiency on 2000 processors for 1.37 billion node simulations

Already we are able to demonstrate—we believe for the first time—scalability to 2048 processors of an entire end-to-end simulation pipeline, from mesh generation to wave propagation to scientific visualization, using a unified, tightly-coupled, online, minimal I/O approach.

2 Octree-based finite element method

Octrees have been used as a basis for finite element approximation since at least the early 90s [31]. Our interest in octrees stems from their ability to adapt to the wavelengths of propagating seismic waves while maintaining a regular shape of finite elements. Here, leaves associated with the lowest level of the octree are identified with trilinear hexahedral finite elements and used for a Galerkin approximation of a suitable weak form of the elastic wave propagation equation. The hexahedra are recursively subdivided into 8 elements until a local refinement criterion is satisfied. For seismic wave propagation in heterogeneous media, the criterion is that the longest element edge should be such that there result at least p nodes per wavelength, as determined by the local shear wave velocity β and the maximum frequency of interest f_{max}. In other words, $h_{max} < \frac{\beta}{p f_{max}}$. For trilinear hexahedra and taking into account the accuracy with which we know typical basin properties, we usually take $p = 10$. An additional condition that drives mesh refinement is that the element size not differ by more than a factor of two across neighboring elements (the octree is then said to be *balanced*). Note that the octree does not explicitly represent material interfaces within the earth, and instead accepts $O(h)$ error in representing them implicitly. This is usually justified for earthquake modeling since the location of interfaces is known at best to the order of the seismic wavelength, i.e. to $\mathcal{O}(h)$. If warranted, higher-order accuracy in representing arbitrary interfaces can be achieved by local adjustment of the finite element basis (e.g., [31]).

Figure 3 depicts the octree mesh (and its 2D counterpart, a quadtree). The left drawing illustrates a factor-of-two edge length difference (a *legal* refinement) and a factor-of-four difference (an *illegal* refinement). Unless additional measures are taken, so-called *hanging nodes* that separate different levels of refinement (indicated by solid circles and the subscript d in the figure) result in a possibly discontinuous field approximation, which can destroy the convergence properties of the Galerkin method. Several possibilities exist to remedy this situation by enforcing continuity of displacement field across the interface either strongly (e.g.,

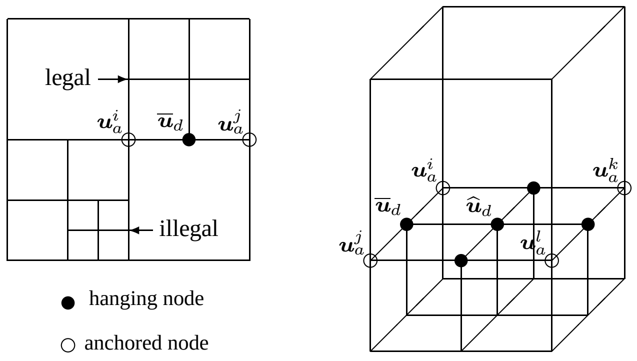


Figure 3: **Quadtree- and octree-based meshes.**

by construction of special transition elements) or weakly (e.g., via mortar elements or discontinuous Galerkin approximation). The simplest technique is to enforce continuity by algebraic constraints that require the displacement of the hanging node be the average of the displacement of its *anchored* neighbors (indicated by open circles and the subscript a). As illustrated in Figure 3, the displacement of an edge hanging node, $\overline{\boldsymbol{u}}_d$, should be the average of its two edge neighbors $\boldsymbol{u}_a^i$ and $\boldsymbol{u}_a^j$, and the displacement of a face hanging node, $\widehat{\boldsymbol{u}}_d$, should be the average of its four face neighbors $\boldsymbol{u}_a^i$, $\boldsymbol{u}_a^j$, $\boldsymbol{u}_a^k$, and $\boldsymbol{u}_a^l$. Efficient implementation of these algebraic constraints will be discussed in the next section. As evident from the figure, when the octree is balanced, an anchored node cannot also be a hanging node.

The previous version of our earthquake modeling code was based on an unstructured mesh data structure and linear tetrahedral finite elements [6, 7]. The present octree-based method [8] has several important advantages over that approach:

- The octree meshes are much more easily generated than general unstructured tetrahedral meshes, particularly when the number of elements increases above 50 million.
- The hexahedra provide relatively greater accuracy per node (the asymptotic convergence rate is unchanged, but the constant is typically improved over tetrahedral approximation).
- The hexahedra all have the same form of the element stiffness matrices, scaled simply by element size and material properties (which are stored as vectors), and thus no matrix storage is required at all. This results in a substantial decrease in required memory—about an order of magnitude, compared to our node-based tetrahedral code.
- Because of the matrix-free implementation, (stiffness) matrix-vector products are carried out at the element level. This produces much better cache utilization by relegating the work that requires indirect addressing (and is memory bandwidth-limited) to vector operations, and recasting the majority of the work of the

matrix-vector product as local element-wise dense matrix computations. The result is a significant boost in performance.

These features permit earthquake simulations to substantially higher frequencies and lower resolved shear wave velocities than heretofore possible. In the next section, we describe the octree-based discretization and solution of the elastic wave equation.

2.1 Wave propagation model. We model seismic wave propagation in the earth via Navier's equations of elastodynamics. Let $\boldsymbol{u}$ represent the vector field of the three displacement components; λ and μ the Lamé moduli and ρ the density distribution; $\boldsymbol{b}$ a time-dependent body force representing the seismic source; $\boldsymbol{t}$ the surface traction vector; and Ω an open bounded domain in $\mathbb{R}^3$ with free surface Γ_{FS}, truncation boundary Γ_{AB}, and outward unit normal to the boundary $\boldsymbol{n}$. The initial–boundary value problem is then written as:

$$\begin{aligned}
\rho\,\ddot{\boldsymbol{u}} - \boldsymbol{\nabla}\cdot\left[\mu\left(\boldsymbol{\nabla}\boldsymbol{u} + \boldsymbol{\nabla}\boldsymbol{u}^{\mathsf{T}}\right) + \lambda(\boldsymbol{\nabla}\cdot\boldsymbol{u})\boldsymbol{I}\right] = \boldsymbol{b} &\quad \text{in} \quad \Omega\times(0,T)\,,\\
\boldsymbol{n}\times\boldsymbol{n}\times\boldsymbol{t} = \boldsymbol{n}\times\boldsymbol{n}\times\dot{\boldsymbol{u}}\sqrt{\rho\mu} &\quad \text{on} \quad \Gamma_{AB}\times(0,T)\,,\\
\boldsymbol{n}\cdot\boldsymbol{t} = \boldsymbol{n}\cdot\dot{\boldsymbol{u}}\sqrt{\rho(\lambda+2\mu)} &\quad \text{on} \quad \Gamma_{AB}\times(0,T)\,,\\
\boldsymbol{t} = \boldsymbol{0} &\quad \text{on} \quad \Gamma_{FS}\times(0,T)\,,\\
\boldsymbol{u} = \boldsymbol{0} &\quad \text{in} \quad \Omega\times\{t=0\}\,,\\
\dot{\boldsymbol{u}} = \boldsymbol{0} &\quad \text{in} \quad \Omega\times\{t=0\}\,,
\end{aligned} \tag{1}$$

With this model, p waves propagate with velocity $\alpha = \sqrt{(\lambda+2\mu)/\rho}$, and s waves with velocity $\beta = \sqrt{\mu/\rho}$. The continuous form above does not include material attenuation, which we introduce at the discrete level via a Rayleigh damping model. The vector $\boldsymbol{b}$ comprises a set of body forces that equilibrate an induced displacement dislocation on a fault plane, providing an effective representation of earthquake rupture on the plane. For example, for a seismic excitation idealized as a point source, $\boldsymbol{b} = -\mu v A \boldsymbol{M} f(t)\boldsymbol{\nabla}\delta(\boldsymbol{x}-\boldsymbol{\xi})$ [4]. In this expression, v is the average earthquake dislocation; A the rupture area; $\boldsymbol{M}$ the (normalized) seismic moment tensor, which depends on the orientation of the fault; $f(t)$ the (normalized) time evolution of the rupture; and $\boldsymbol{\xi}$ the source location.

Since we model earthquakes within a portion of the earth, we require appropriately positioned absorbing boundaries to account for the truncated exterior. For simplicity, in (1) the absorbing boundaries are given as dashpots on Γ_{AB}, which approximate the tangential and normal components of the surface traction vector $\boldsymbol{t}$ with time derivatives of corresponding components of the displacement vector.

Even though this absorbing boundary is approximate, it is local in both space and time, which is particularly important for large-scale parallel implementation. Finally, we enforce traction-free conditions on the earth surface.

2.2 Octree discretization. We apply standard Galerkin finite element approximation in space to the appropriate weak form of the initial-boundary value problem (1). (*The rest of this section has been commented out to satisfy the length requirement; it will be restored in the full paper.*)

2.3 Temporal approximation. The time dimension is discretized using central differences. (*The rest of this section has been commented out to satisfy the length requirement; it will be restored in the full paper.*)

The combination of an octree-based wavelength-adaptive mesh, piecewise trilinear Galerkin finite elements in space, explicit central differences in time, constraints that enforce continuity of the displacement approximation, and local-in-space-and-time absorbing boundaries yields a second-order-accurate in time and space method that is capable of scaling up to the very large problem sizes that are required for high resolution earthquake modeling.

3 An end-to-end approach

The octree-based finite element method just described can be implemented using a traditional, offline, file-based approach [3, 19, 26, 27, 32]. However, the inherent pitfalls, as outlined in Section 1, cannot be eliminated unless we introduce a major change in design principle.

Our new computing model thus follows an online, end-to-end approach. We view different components of a simulation pipeline as integral parts of a tightly-coupled parallel runtime system, rather than individual stand-alone programs. A number of technical difficulties emerge in the process of implementing this new methodology within an octree-based finite element simulation system. This section discusses several fundamental issues to be resolved, outlines the interfaces between simulation components, and presents a sketch of the core algorithms.

Some of the techniques presented here are specific to the target class of octree-based methods. On the other hand, the design principles are more widely applicable; we hope they will help accelerate the adoption of end-to-end approaches to parallel supercomputing where applicable.

3.1 Fundamental issues. Below we discuss fundamental issues encountered in developing a scalable octree-based finite element simulation system. The solutions provided are critical to efficient implementation of different simulation components.

3.1.1 Organizing a parallel octree. The octree serves as a natural choice for the backbone structure tying together all add-on components (i.e., data structures and algorithms). We distribute the octree among all processors to exploit data parallelism. Each processor retains 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 4(b)(c)(d). (For clarity, we use 2D quadtrees and quadrants in the figures and examples.)

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 4(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 a leaf octant, marked as *remote*. For example, on PE 0, octants 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 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.

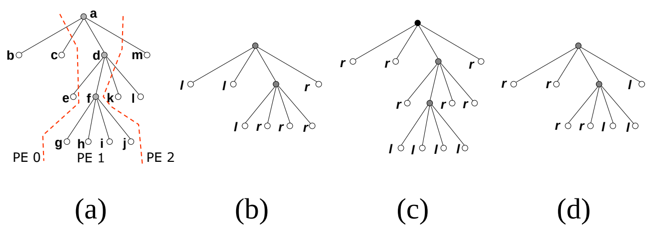


Figure 4: **Parallel octree organization on 3 processors.** Circles marked by l represent local leaf octants; and those marked by r represent aggregated remote leaf octants. (a) A global octree. (b)(c)(d) The local instances on PE0, PE1 and PE2, respectively.

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. To maintain consistency in the parallel octree, 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 differs from that 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 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 practice, a global octree can have many levels and needs to be distributed among hundreds or thousands of processors. In these cases, the local instance method pays off because each processor allocates enough memory to keep track of only its share of the leaf octants.

Due to massive memory requirements and redundant computational costs, we never—and in fact, are unable to—build a fully-grown global octree on a single processor 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 maintain a coherent global parallel octree.

3.1.2 Addressing an octant.

In order to manipulate octants in a distributed octree, we need to be able to identify individual octants, for instance to support neighbor-finding operations or data migrations.

The foundation of our addressing scheme is the linear octree technique [1, 13, 14]. 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. Let us label each tree edge with a binary *directional code* that distinguishes each child of an internal octant. A locational code is obtained by concatenating the directional codes on the path from the root octant to a target octant [23]. To make all the locational codes of equal length, we may need to pad zeroes to the concatenated directional codes. Finally, we append the level of the target octant to the bit-string to complete a locational code. Figure 5 shows how to derive the locational code for octant g, assuming the root octant is at level 0 and the lowest level supported is 4.

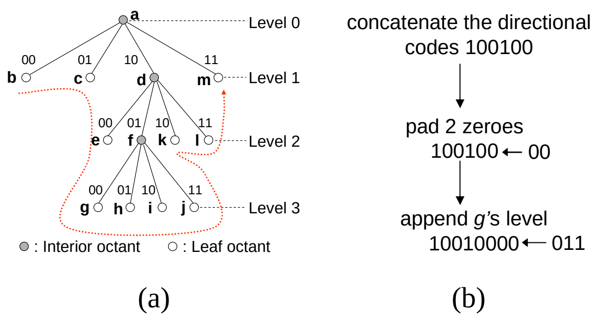


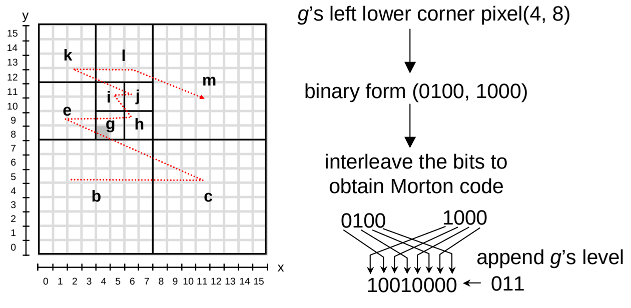
Figure 5: **Deriving locational codes using a tree structure.** (a) A tree representation. (b) Deriving the locational code for g.

The procedure just described assumes the existence of a tree structure to assist with the derivation of a locational code. Since we do not maintain a global octree structure, we have devised an alternative way to compute the locational codes. Figure 6 illustrates the idea. Instead of a tree structure, we view an octree from a domain decomposition perspective. A *domain* is a Cartesian coordinate space that consists of a uniform grid of $2^n \times 2^n$ indivisible *pixels*. To compute the locational code of octant g, we first interleave the bits of the coordinate of its lower left pixel to produce the so-called Morton code [21]. Then we append g's level to compose the locational code.

It can be verified that the two methods of deriving locational codes are equivalent and produce the same result. The second method, though, allows us to compute a globally unique address by simple local bit operations.

3.1.3 Locating an octant.

Given the locational code of an octant, we need to locate the octant in a distributed environment. That is, we need to find out which processor hosts a given octant. Whether we can efficiently locate


y
x
k
l
m
i
j
e
g
h
b
c
g's left lower corner pixel(4, 8)
binary form (0100, 1000)
interleave the bits to obtain Morton code
0100
1000
append g's level
10010000 ← 011

(a) (b)

Figure 6: **Computing locational codes without using a tree structure.** (a) A domain representation. (b) Computing the locational code for g.

an octant directly affects the performance and scalability of our system. We have developed a simple but powerful technique based on a *locational code interval table* to solve this problem.

The basis of our solution is a simple fact: the pre-order traversal of the leaf octants produces the same ordering as the ascending locational code ordering. The dotted lines in Figure 5(a) and Figure 6(a) illustrate the two identical orderings, respectively. Since we assign to each processor a contiguous chunk of leaf octants in pre-order traversal ordering (as explained in Section 3.1.1), we have effectively partitioned the locational code range in its ascending order. Each processor hosts a range of ascending locational codes that does not overlap with others.

Accordingly, we implement a locational interval table as an array that maps processor ids to locational codes. Each element of the array records the smallest locational code on the corresponding processor (i.e. the ith element records the smallest locational code on processor i). This table is replicated on all processors. In our example (Figure 4(a)), a locational code interval table contains three entries, recording the locational codes of octant b, g, and k, respectively.

We use a locational code interval table to perform quick inverse lookups. That is, given an arbitrary locational code, we conduct a binary search in the locational code interval table and find the interval index (i.e., processor id) of the entry whose locational code is the largest among all those that are smaller than the given locational code. Note that this is a local operation and incurs no communication cost.

A locational code interval table is efficient in both space and time. The memory overhead on each processor to store an interval table is $\mathcal{O}(P)$, 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 12 MB. Time-wise, the overhead of an inverse lookup is $\mathcal{O}(\log P)$, the cost of a binary search.

3.1.4 Manipulating an octant.

There are various situations when we need to manipulate an octant. For example, we need to search for a neighboring octant when generating a mesh. If the target octant is hosted on the same processor where an operation is initiated, we use the standard pointer-based octree algorithm [23] to traverse the local instance to manipulate the octant. If the target octant is hosted on a remote processor, we compute its locational code and conduct a lookup using the locational code interval table to find its hosting processor. We store the locational code of the target octant and the intended operation in a request buffer destined for the hosting processor. The request buffers are later exchanged among processors. Each processor executes the requests issued by its peers with respect to its local octants. On the receiving end, we are able to locate an octant using its locational code by reversing the procedure of deriving a locational code from a tree structure (shown in Figure 5). Without a locally computed and globally unique address, such cross-processor operations would have involved much more work.

3.2 Interfaces.

There are two types of interfaces in the Hercules system: (1) the interface to the underlying octree, and (2) the interface between simulation components.

All simulation components manipulate the underlying octree to implement their respective functions. For example, the mesher needs to refine or coarsen the tree structure to effect necessary spatial discretization. The solver needs to attach runtime solution results to mesh nodes. The visualization component needs to process the attached data. In order to support such common operations efficiently, we implement the backbone parallel octree in two abstract data types (ADTs): `octant_t` and `octree_t`, and provide a small application program interface (API) to manipulate the ADTs. For instance, at the octant level, we provide functions to search for an octant, install an octant, and sprout or prune an octant. At the octree level, we support various tree traversal operations as well as the initialization and adjustment of the locational code lookup table. This interface allows us to encapsulate the complexity of manipulating the backbone parallel octrees within the abstract data types.

Note that there is one (and the only one) exception to the cleanliness of the interface. We reserve a place-holder in `octant_t`, allowing a simulation component (e.g., a solver) to install a pointer to a data structure where component-specific data can be stored and retrieved. Nevertheless, such flexibility does not undermine the robustness of the Hercules system because any structural changes to the backbone octree must still be carried out through a pre-defined API call.

We have also designed binding interfaces between the simulation components. However, unlike the octree/octant interface, the inter-component interfaces can be clearly explained only in the context of the simulation pipeline. Therefore, we defer the description of the inter-component interfaces to the next section where we outline the core algorithms of individual simulation components.

3.3 Algorithms.

Engineering a complex parallel simulation system like Hercules not only involves careful software architectural design, but also demands non-trivial algorithmic innovations. This section highlights important algorithm and implementation features of Hercules. We have omitted many of the technical details.

3.3.1 Meshing and partitioning.

We generate octree meshes online *in-situ* [29]. That is, we generate an octree mesh in parallel on the same processors where a solver and a visualizer will be running. Mesh elements and nodes are created where they will be used instead of on remote processors. This strategy requires that mesh partitioning be an integral part of the meshing component. The partitioning method we use is simple [5, 11]. We sort all octants in ascending locational code order, often referred to as the Z-order [12], and divide them into equal length chunks in such a way that each processor will be assigned one and only one chunk. Because the locational ordering of the leaf octants corresponds exactly to the pre-order traversal of an octree, the partitioning and data redistribution often involve leaf octants migrating only between adjacent processors. Whenever data migration occurs, local instances of participating processors are adjusted to maintain a consistent global data structure. As shown in Section 4, this simple strategy works well and yields almost ideal speedup for fixed-size problems.

The process of generating an octree-based hexahedral mesh is shown in Figure 7. First, NEWTREE bootstraps a small and shallow 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 parallel octree. After a balanced parallel octree is obtained, PARTITIONTREE redistributes the leaf octants among the processors using the space-filling curve partitioning technique. Finally and most importantly, EXTRACTMESH derives mesh element and node information and determines the various associations between elements and nodes. The overall algorithm complexity of the meshing component is $\mathcal{O}(N \log E)$, where N and E are the numbers of mesh nodes and elements, respectively.

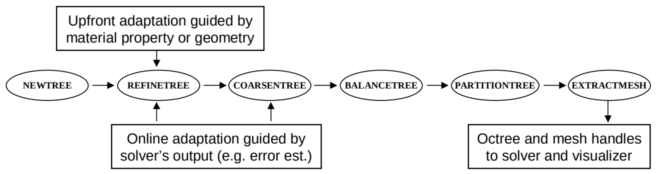

Upfront adaptation guided by material property or geometry
NEWTREE
REFINETREE
COARSENTREE
BALANCETREE
PARTITIONTREE
EXTRACTMESH
Online adaptation guided by solver's output (e.g. error est.)
Octree and mesh handles to solver and visualizer

Figure 7: **Meshing and partitioning component.**

It should be noted that the parallel octree alone—though scalable and elegant for locating octants and distributing workloads—is not sufficient for implementing all meshing functionality. The key challenge here is how to deal with octants' vertices (i.e., mesh nodes). We can calculate the coordinates of the vertices in parallel and obtain a collection of geometric objects (octants and vertices). But by themselves, octants and vertices are not a finite element mesh. To generate a mesh and make it usable to a solver, we must identify the associations between octants and vertices (mesh connectivity), and between vertices and vertices, either on the same processor (hanging-to-anchored dependencies) or on different processors (inter-processor sharing information). Therefore, in order to implement steps such as BALANCETREE and EXTRACTMESH, which require capabilities beyond those offered by parallel octree algorithms, we have incorporated auxiliary data structures and developed several new algorithms such as *parallel ripple propagation* and *parallel octree bucket sorting* [29].

As mentioned in Section 3.2, the interface between simulation components provides the glue that ties the Hercules system together. The interface between the meshing and solver components consists of two parts: (1) abstract data types, and (2) callback functions. When meshing is completed, a mesh abstract data type (`mesh_t`), along with a handle to the underlying octree (`octree_t`), is passed forward to a solver. The `mesh_t` ADT contains all the information a solver would need to initialize an execution environment. On the other hand, a solver controls the behavior of a mesher via callback functions that are passed as parameters to the REFINETREE and COARSENTREE steps at runtime. The latter interface allows us to perform runtime mesh adaptation, which is critical for future extension of the Hercules framework to support solution adaptivity.

3.3.2 Solving.

Figure 8 shows the solver component's workflow. After the meshing component hands over control, the INITENV step sets an execution environment by computing element-independent stiffness matrices, allocating and initializing various local vectors, and building a communication schedule. Next, the DEFSOURCE step converts an earthquake source specification to a set of equivalent forces applied on mesh nodes. Then, a solver enters its main loop (inner kernel) where displacements and velocities associated with mesh nodes are computed for each simulation time step (i.e., the COMPDISP step). If a particular time step needs to be visualized, which is determined either *a priori* or at runtime (online steering), the CALLVIS step passes the control to a visualizer. Once an image is rendered, control returns to the solver, which repeats the procedure for the next time step until termination. The explicit wave propagation solver has optimal complexity, i.e. $\mathcal{O}(N^{\frac{4}{3}})$ This stems from the fact that simply writing the solution requires $\mathcal{O}(N^{\frac{4}{3}})$ complexity, since $\mathcal{O}(N)$ mesh nodes are required for accurate spatial resolution, and $\mathcal{O}(N^{\frac{1}{3}})$ time steps for accurate temporal resolution, which is of the order dictated by the CFL stability condition.

The COMPDISP step (performing local element-wise dense matrix computation, exchanging data between processors, enforcing hanging node constraints, etc.) presents no major technical difficulty, since this inner kernel is by far the most well studied and understood part. What is more interesting is how the solver interacts with other simulation components and with the underlying octree in the INITENV, DEFSOURCE, and CALLVIS steps.

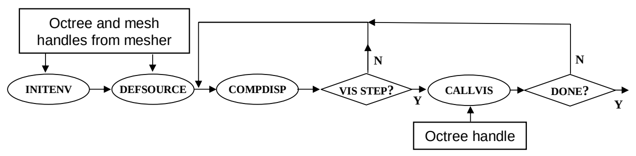


Figure 8: **Solving component.**

In the INITENV step, the solver receives an *in-situ* mesh via an abstract data type `mesh_t`, which contains such important information as the number of elements and nodes assigned to a processor, the connectivity of the local mesh (element–node association, hanging-anchored node association), and the sharing information (which processor shares which local mesh nodes), and so forth. All initialization work, including the setup of a communication schedule, is thus performed in parallel without any communication among processors.

Along with the `mesh_t` ADT, the solver also receives a handle to the backbone octree's local instance `octree_t`. One of the two important applications of the `octree_t` ADT is to provide an efficient search structure for defining earthquake sources (the DEFSOURCE step). We support kinematic earthquake sources whose displacements (slips) are prescribed. The simplest case is a point source. Note that the coordinate of a point source is not necessarily that of any mesh node. We implement a point source by finding the enclosing hexahedral element of the coordinate and converting the prescribed displacements to an equivalent set of forces applied on the eight mesh nodes of the enclosing element. For general cases of fault planes or arbitrary fault shapes, we first transform a fault to a set of point sources and then apply the technique for the single point source multiple times. In other words, regardless of the kinematic source, we must always locate the enclosing elements of arbitrary coordinates. We are able to implement the DEFSOURCE step using the octree/octant interface, which provides the service of searching for octants.

The other important application of the `octree_t` ADT is to serve as a vehicle for the solver to pass data to the visualization component. Recall that we have reserved a place-holder in the `octree_t` ADT. Thus, we allocate a buffer that holds results from the solver (displacements or velocities), and install the pointer to the buffer in the place-holder. As new results are computed at each time step, the result buffer is updated accordingly. Note that to avoid unnecessary double buffering, we do not copy floating-point numbers directly into the result buffer. Instead, we store pointers (array offsets) to internal solution vectors and implement a set of macros to manipulate the result buffer (de-reference pointers and compute results). So from a visualization perspective, the solver has provided a concise data service. Once the CALLVIS step transfer the control to a visualizer, the latter is able to retrieve simulation result data from the backbone octree by calling these macros.[4]

3.3.3 Visualization.

Simulation-time 3D visualization has rarely been attempted in the past for three main reasons. First, scientists are reluctant to use their supercomputing allocations for visualization work. Second, a data organization designed for a simulation is generally unlikely to support efficient visualization computations. Third, performing visualization on a separate set of processors requires repeated movement of large amounts of data, which competes for scarce network bandwidth and increases the complexity of the simulation code.

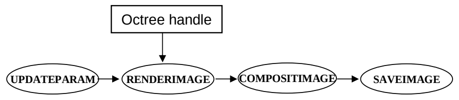


Figure 9: **Visualizing component.**

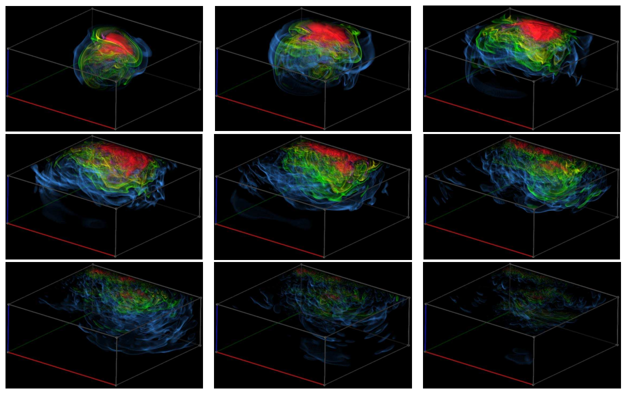


Figure 10: **A sequence of snapshot images of propagating waves of 1994 Northridge earthquake.**

By taking an online, end-to-end approach, we have been able to incorporate highly adaptive parallel visualization into Hercules. Figure 9 shows how the visualization component works. First, the UPDATEPARAM step updates the viewing and rendering parameters. Next, the RENDERIMAGE step renders local data, that is, values associated with blocks of hexahedral elements on each processor. The details on the rendering algorithm can be found in [19, 32]. The partially rendered images are then composited together in the COMPOSITIMAGE step. We use *scheduled linear image compositing* (SLIC) [24], which has proven to be the most flexible and efficient algorithm. Previous parallel image compositing algorithms are either not scalable or designed for a specific network topology [2, 15, 18]. Finally, the SAVEIMAGE step stores an image to disk. Figure 10 shows a sequence of example images. The cost of the visualization component per invocation is $\mathcal{O}(xyE^{\frac{1}{3}}\log E)$, where x,y

[4]When visualization does not need to be performed at a given time step, no data access macros are called; thus no memory access or computation overhead occurs.

PEs	1	16	52	184	748	2000
Frequency	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
Hanging	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
Time steps	2000	4000	10000	8000	2500	2500
E2E time (sec)	12911	19804	38165	48668	13033	16709
Replication (sec)	22	71	85	94	187	251
Meshing (sec)	20	75	128	150	303	333
Solver (sec)	8381	16060	31781	42892	11960	16097
Visualization (sec)	4488	3596	6169	5528	558	*
E2E time/step/elem/PE (μs)	9.77	7.98	7.93	7.86	8.44	10.92
Solver time/step/elem/PE (μs)	6.34	6.48	6.60	6.92	7.74	10.52
Mflops/sec/PE	569	638	653	655	*	*

Figure 11: **Summary of the characteristics of the isogranular experiments.** The entries marked as "*" are data points that we have not yet been able to obtain due to various technical reasons.

represent the 2D image resolution and E is the number of mesh elements.

The visualization component relies on the underlying parallel octree for two purposes: (1) to retrieve simulation data from the solver, and (2) to implement its adaptive rendering algorithm. We have described the first usage in the previous section. Let us now explain the second. To implement a ray-casting based rendering algorithm, the visualization component needs to traverse the octree structure. By default, all leaf octants intersecting a particular ray must be processed in order to project a pixel. However, we might not always want to render at the highest resolution, i.e. at the finest level of the octree. For example, when rendering hundreds of millions of elements on a small image of 512 $\times$ 512 pixels, little additional detail is revealed if we render at the highest level, unless a close-up view is selected. Thus, to achieve better performance of rendering without compromising image quality, we perform a view-dependent pre-processing step to choose an appropriate octree level before actually rendering the image [32]. Operationally, it means that we need to ascend the tree structure and render images at a coarser level. The small set of API functions that manipulate the backbone octree (see Section 3.2) serves as a building block for supporting such adaptive visualization.

4 Performance

In this section, we provide preliminary performance results that demonstrate the scalability of the Hercules system. We also describe interesting performance characteristics and observations identified in the process of understanding the behavior of Hercules as a complete simulation system.

The simulations have been conducted to model seismic wave propagation during historical and postulated earthquakes in the Greater Los Angeles Basin, which comprises a 3D volume of $100 \times 100 \times 37.5$ kilometers. We report performance on *Lemieux*, the HP AlphaServer system at the Pittsburgh Supercomputing Center. The Mflops numbers were measured using the HP Digital Continuous Profiling Infrastructure (DCPI) [10].

The earth property model is the Southern California Earthquake Center 3D community velocity model [20] (Version 3, 2002), known as the SCEC CVM model. We query the SCEC CVM model at high resolution offline and in advance, and then compress, store and index the results in a material database [25] (≈ 2.5GB in size). Note that this is a one-time effort, and the database is reused by many simulations. In our initial implementation, all processors queried a single material database stored on a parallel file system. But unacceptable performance led us to to modify our implementation to replicate the database onto local disks attached to each compute node prior to a simulation.

4.1 Isogranular scalability study. Our main interest is understanding how the Hercules system performs as the problem size and number of processors increase, maintaining more or less the same problem size (or work per time step) on each processor.

Figure 11 summarizes the characteristics of the isogranular experiments. *PEs* indicates the number of processors used in a simulation. *Frequency* represents the maximum seismic frequency resolved by a mesh. *Element*, *Nodes*, *Anchored*, and *Hanging* characterize the size of each mesh. *Max leaf level* and *Min leaf level* represent the smallest and largest elements in each mesh, respectively. *Elements/PE* is used as a rough indicator of the workload per time step on each processor. Given the unstructured nature of the finite element meshes, it is impossible to guarantee a constant number of elements per processor. Nevertheless, we have contained the difference to within 10%. *Time steps* indicates the number of explicit time steps executed. The *E2E time* represents the absolute running time of a Hercules simulation from the moment the code is loaded onto a supercomputer to the moment it exits the system. This time includes the time of replicating a material database stored on a shared parallel file system to the local disk attached to each compute node (*Replication*), the time to generate and partition an unstructured finite element mesh (*Meshing*), the time to simulate seismic wave propagation (*Solver*), and the time to create visualizations and output jpeg images (*Visualization*). *E2E time/step/elem/PE* and *Solver time/step/elem/PE* are the amortized cost per element per time step per processor for end-to-end time and solver time, respectively. *Mflops/sec/PE* stands for the sustained megaflops per second per processor.

Note that the simulations involve highly unstructured meshes, with the largest elements 64 times as large in edge size as the smallest ones. Because of spatial adaptivity of the meshes, there are many hanging nodes, which account for 11% to 20% of the total mesh nodes.

A traditional way to assess the overall isogranular parallel efficiency is to examine the degradation of the sustained average Mflops per processor as the number of processors increases. In our case, we achieve 28% to 33% of peak performance on the Alpha EV68 processors (2 GFlops/sec/PE). However, no degradation in sustained average floating-point rate is observed. On the contrary, the rate increases as we solve larger problems on larger numbers of processors (up to 184 processors). This counter-intuitive observation can be explained as follows: the solver is the most floating point-intensive and time-consuming component; as the problem size increases, processors spend more time in the solver executing floating-point instructions, thus boosting the overall Mflops/s per processor rate.

To assess the isogranular parallel efficiency in a more meaningful way, we analyze the running times. Figure 12 illustrates the contribution of each component of Hercules to the total running time. One-time costs such as replicating the material database and generating a mesh are inconsequential and are almost invisible in the figure.[5] Among the recurring costs, visualization has lower per-time step algorithmic complexity ($\mathcal{O}(xyE^{\frac{1}{3}} \log E)$) than that of the solver ($\mathcal{O}(N)$). (N, the number of mesh nodes, is of the order of E, the number of mesh elements, in an octree-based hexahedral mesh.) Therefore, as both the problem size and number of processors increase, the solver time overwhelms the visualization time by greater and greater margins.

Figure 13 shows the trends of the amortized end-to-end running time and solver time per time step per element per processor. Although the end-to-end time is always higher than the solver time, as we increase the problem size, the amortized solver time approaches the end-to-end time due to the solver's asymptotic dominance. The key insight here is that, with careful design of parallel data structures and algorithms for the entire simulation pipeline, the limiting factor for high isogranular scalability on a large number of processors is the scalability of the solver proper, rather than front and back end components. Therefore, it is reasonable to use the degradation in the amortized solver time to measure the isogranular efficiency of the entire simulation pipeline (in place of the end-to-end time, which in fact implies greater than 100% efficiency). As shown in Figure 11, the solver time per step per element per processor is 6.34 μs on a single PE and 7.74 μs on 748 PE. Hence, we obtain an isogranular parallel efficiency of 81%, a good result considering the high irregularity of the meshes. The 2000-PE data point shown in the figure corresponds to solution of a 1.37 billion node problem without visualization. In this case, we achieve an isogranular parallel efficiency of 60%.

[5]For the 1.5 Hz (748-PE run) and 2 Hz (2000-PE run) cases, we have computed just 2,500 time steps. Because of this, the *replication* and *meshing* costs appear slightly more significant in the figure than they would had a full-scale simulation of 20,000 time steps been executed. Also note that the 2000-PE case does not include the visualization component.

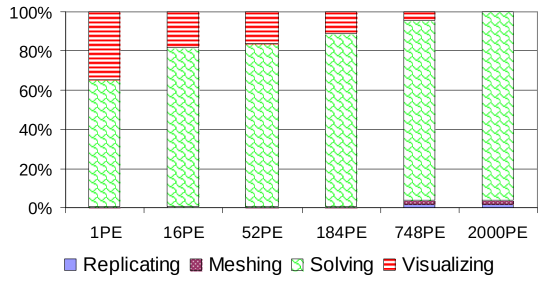


Figure 12: **The percentage contribution of each simulation component to the total running time.**

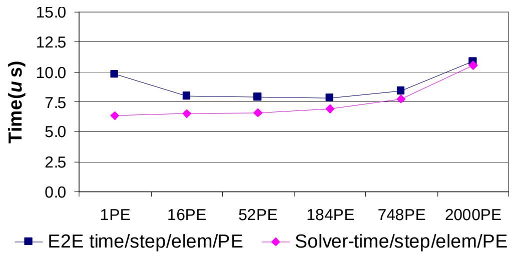


Figure 13: **The amortized running time per per step per element per processor.** The top curve corresponds to the amortized end-to-end running time and the lower corresponds to the amortized solver running time.

4.2 Fixed-size scalability study.

In this set of experiments, we investigate the fixed-size scalability of the Hercules system. That is, we fix the problem size and solve the same problem on different numbers of processors to examine the performance improvement in running time.

We have conducted three sets of fixed-size scalability experiments, for small size, medium size, and large size problems, respectively. The experimental setups are shown in Figure 14. The performance results are shown in Figure 15. Each column represents the results for a set of fixed-size experiments. From left to right, we display the plots for

PEs	1	2	4	8	16	32	64	128	256	512	1024	2048
Small case (0.23 Hz, 0.8M nodes)	x	x	x	x	x							
Medium case (0.5 Hz, 11M nodes)				x	x	x	x	x				
Large case (1 Hz, 134M nodes)								x	x	x	x	x

Figure 14: **Setup of fixed-size speedup experiments.** Entries marked with "x" represent experiment runs.

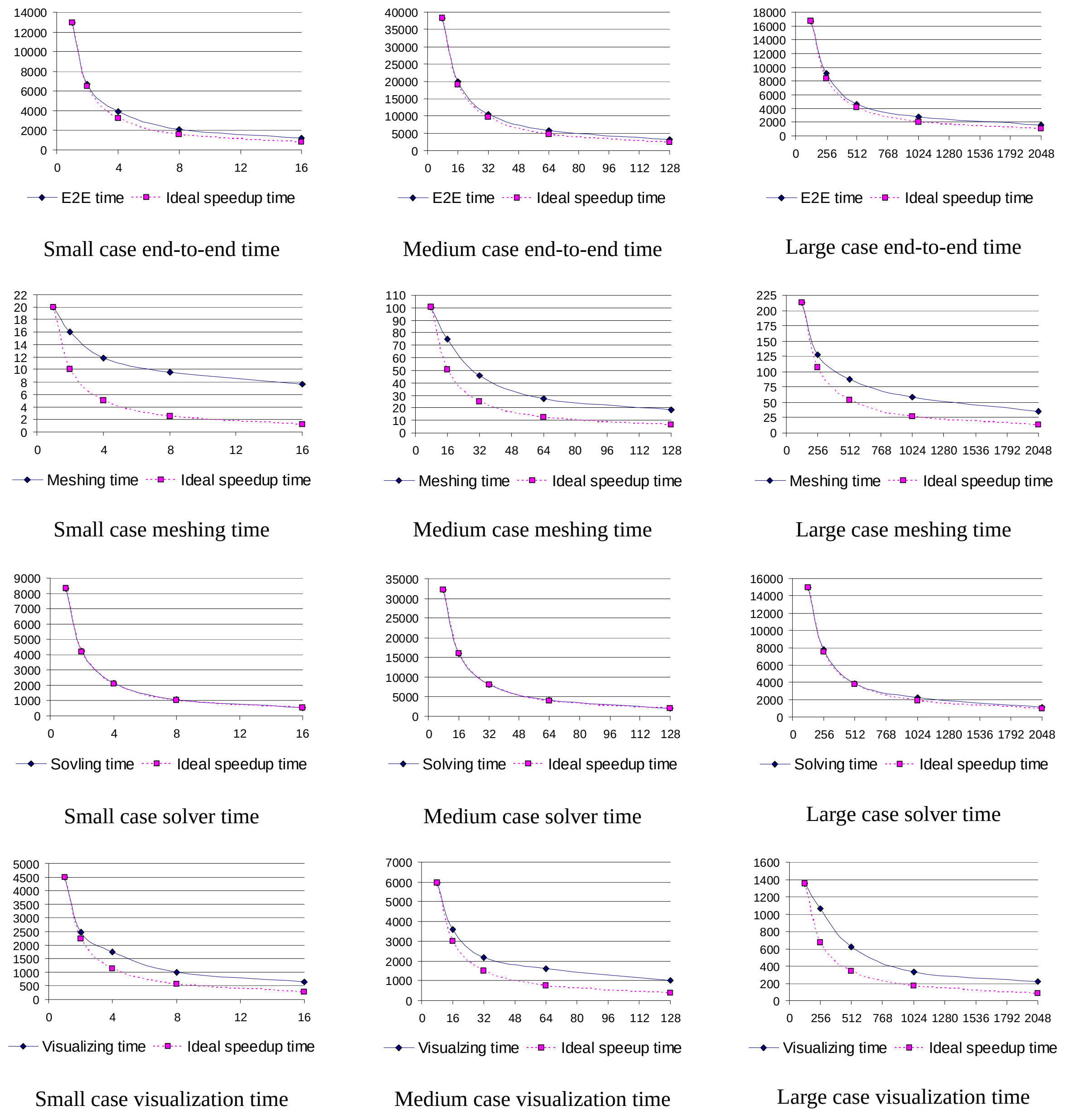


Figure 15: **Speedups of fixed-size experiments.** The horizontal axes represent the number of processors. The vertical axes represent the running time in seconds. The first row shows the end-to-end running time; the second the meshing and partitioning time; the third the solver time; and the fourth the visualization time.

the small, medium, and large cases, respectively.

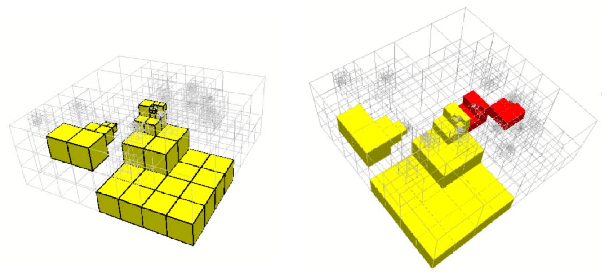
The first row of the plots shows that Hercules, as a system for end-to-end simulations, scales well even for fixed-size problems. As we increase the numbers of processors (to 16 times as many for all three cases), the end-to-end running times improve accordingly. The actual running time curve tracks the ideal speedup curve closely. The end-to-end parallel efficiencies are 66%, 76%, and 64%, for the small case (16 PE vs. 1 PEs), medium case (128 PEs vs. 8 PEs), and large case (2048 PEs vs. 128 PEs), respectively.

The second row shows the performance of the meshing/partitioning component only. Although not perfect, this component achieves reasonable speedups while running on a large number of processors. For earthquake wave propagation simulations, meshes are static and are only generated just once prior to computation. Therefore, the cost of meshing and partitioning is dwarfed by thousands of simulation time steps, provided mesh generation is reasonably fast and scalable.

The third row of Figure 15 shows a somewhat surprising result: the solver achieves almost ideal speedup on hundreds to thousands of processors, even though the partitioning strategy we used (dividing a Z-ordered sequence of elements into equal chunks) is rather simplistic. In fact, the solver's parallel efficiency is 97%, 98%, and 86%, for the small case, medium case, and large case, respectively. Since solving is the most time-consuming component of the Hercules system, its high fixed-size parallel efficiency has improved the performance of the entire end-to-end system in a significant way.

The speedup of the visualization component, as shown in the fourth row of Figure 15, is, however, less satisfactory, even though the general trend of the running time indeed shows improvement as more processors are used. Because this component is executed at each visualization time step (usually every 10th solver time step), the less-than-optimal speedup has a much larger impact on the overall end-to-end performance than the meshing/partitioning component. The visualization parallel efficiency is actually 44%, 36%, and 38%, for the small case, medium case, and large case, respectively.

We attribute this performance degradation to the space-filling curve based partitioning strategy, which assigns an *equal number* of neighboring elements to each processor, a strategy that is optimal for the solver. However, this strategy is suboptimal for the visualization component, since the workload on each processor (Figure 16(a)) is proportional to *both the number and the size* of the local elements to be rendered. Therefore, different processors may have dramatically *different block sizes* to render, as shown in Figure 16(b). The light blocks represent elements assigned to one processor and the dark blocks another processor. As a result, the workload can be highly unbalanced for the RENDERIMAGE and COMPOSITIMAGE steps, especially when larger numbers of processors are involved. To remove this performance bottleneck, a viable approach is to use a new hybrid rendering scheme that balances the workload dynamically by taking into account the cost of transferring elements versus that of pixels [9]. An element is rendered locally only if the rendering cost is lower than the cost of sending the resulting projected image to the processor responsible for compositing the image. Alternatively, we can re-evaluate the space-filling curve based partitioning strategy and develop a new scheme that better accommodates both the solver and the visualization components. Striking a balance between the data distributions for the two is an inherent issue for parallel end-to-end simulations.



(a) (b)

Figure 16: **Workload distribution.** (a) Elements assigned on one processor. (b) Unbalanced visualization workload on two processors.

5 Conclusion

We have demonstrated that the bottlenecks associated with front-end mesh generation and back-end visualization can be eliminated for ultra-large scale simulations through careful design of parallel data structures and algorithms for end-to-end performance and scalability. By eliminating the traditional, cumbersome file interface, we have been able to turn "heroic" runs—large-scale simulations that often require weeks of preparation and post-processing—into daily exercises that can be launched readily on parallel supercomputers.

Our new approach calls for new ways of designing and implementing high-performance simulation systems. Besides data structures and algorithms for each individual simulation components, it is important to account for the interactions between these components in terms of both control flow and data flow. It is equally important to design suitable parallel data structures and runtime systems that can support all simulation components. Although we have implemented our methodology in a framework that targets octree-based finite element simulations for earthquake modeling, we expect that the basic principles and design philosophy can be applied in the context of other types of large-scale physical simulations.

The end-to-end approach calls for new ways of assessing parallel supercomputing implementations. We need to take into account all simulation components, instead of merely the inner kernels of solvers. Sustained floating point rates of inner kernels can help explain achieved faster run times. But they should not be used as the only indicator of high performance or scalability. No clock time—used either by processors, disk, network, or humans—should be excluded in the evaluation of the effectiveness of a simulation system. After all, overall turnaround time is *the* most important performance metric for real-world scientific and engineering simulations.

Acknowledgments

This work is sponsored in part by NSF under grant IIS-0429334, in part by a subcontract from the Southern California Earthquake Center (SCEC) 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 SCEC CME partners Tom Jordan and Phil Maechling for their support and help. Special thanks to John Urbanic, Chad Vizino, and Sergiu Sanielevici at PSC for their outstanding technical support.

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] J. Ahrens and J. Painter, *Efficient sort-last rendering using compression-based image compositing*, in Proceedings of the 2nd Eurographics Workshop on Parallel Graphics and Visualization, 1998, pp. 145–151.

[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, November 2003.

[4] K. Aki and P. G. Richards, *Quantitative Seismology: Theory and Methods*, vol. I, W. H. Freeman and Co., 1980.

[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] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D. O'Hallaron, J. Shewchuk, and J. Xu, *Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers*, Computer Methods in Applied Mechanics and Engineering, 152 (1998), pp. 85–102.

[7] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and Jifeng Xu, *Earthquake ground motion modeling on parallel computers*, in Supercomputing '96, Pittsburgh, PA, November 1996.

[8] J. Bielak, O. Ghattas, and E.J. Kim, *Parallel octree-based finite element method for large-scale earthquake ground motion simulation*, Computer Modeling in Engineering and Sciences, 10 (2005), pp. 99–112.

[9] H. Childs, M. Duchaineau, and K.-L. Ma, *A scalable, hybrid scheme for volume rendering massive data sets*, in Proceedings of Eurographics Symposium on Parallel Graphics and Visualization, Lisbon, Portugal, May 2006.

[10] *HP DCPI Tool*. `http://h30097.www3.hp.com/dcpi/`, 2004.

[11] 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.

[12] 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.

[13] I. Gargantini, *An effective way to represent quadtrees*, Communications of the ACM, 25 (1982), pp. 905–910.

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

[15] T.-Y. Lee, C. S. Raghavendra, and J. B. Nicholas, *Image composition schemes for sort-last polygon rendering on 2D mesh multicomputers*, IEEE Transactions on Visualization and Computer Graphics, 2 (1996), pp. 202–217.

[16] K.-L. Ma and T. Crockett, *A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data*, in Proceedings of 1997 Symposium on Parallel Rendering, 1997, pp. 95–104.

[17] ———, *Parallel visualization of large-scale aerodynamics calculations: A case study on the Cray T3E*, in Proceedings of 1999 IEEE Parallel Visualization and Graphics Symposium, San Francisco, CA, October 1999, pp. 15–20.

[18] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, *Parallel volume rendering using binary-swap compositing*, IEEE Computer Graphics and Applications, 14 (1994), pp. 59–67.

[19] K.-L. Ma, A. Stompel, J. Bielak, O. Ghattas, and E. Kim, *Visualizing large-scale earthquake simulations*, in SC2003, Phoenix, AZ, November 2003.

[20] 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).

[21] G. M. Morton, *A computer oriented geodetic data base and a new technique in file sequencing*. Tech. Report, IBM, 1966.

[22] *1997 Petaflops algorithms workshop summary report*. `http://www.hpcc.gov/pubs/pal97.html`, 1997.

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

[24] A. Stompel, K.-L. Ma, E. Lum, J. Ahrens, and J. Patchett, *SLIC: Scheduled linear image compositing for parallel volume rendering*, in Proceedings of IEEE Sympoisum on Parallel and Large-Data Visualization and Graphics, October 2003.

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

[26] T. Tu and D. R. O'Hallaron, *Balanced refinement of massive linear octrees*, Tech. Report CMU-CS-04-129, Carnegie Mellon School of Computer Science, April 2004.

[27] ———, *A computational database system for generating unstructured hexahedral meshes with billions of elements*, in SC2004, Pittsburgh, PA, November 2004.

[28] ———, *Extracting hexahedral mesh structures from balanced linear octrees*, in Proceedings of the Thirteenth International Meshing Roundtable, Williamsburgh, VA, September 2004.

[29] T. Tu, D. R. O'Hallaron, and O. Ghattas, *Scalable parallel octree meshing for terascale applications*, in SC2005, Seattle, WA, November 2005.

[30] 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, September 2002, pp. 127– 138. Also in Engineering with Computers (2004) 20:117–128.

[31] 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.

[32] H. F. Yu, K-L Ma, and J. Welling, *A parallel visualization pipeline for terascale earthquake simulations*, in SC 2004, Pittsburgh, PA, November 2004.