From Physical Modeling to Scientific Understanding — 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

January 2006
CMU-CS-06-105

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Conventional parallel scientific computing uses files as interface between simulation components such as meshing, partitioning, solving and visualizing. This approach results in time-consuming file transfers, disk I/O and data format conversions that consume large amounts of network, storage, and computing resources while contributing nothing to applications. We propose an *end-to-end* approach to parallel supercomputing. The key idea is to replace the cumbersome file interface with a scalable, parallel, runtime data structure, on top of which all simulation components are constructed in a tightly coupled way. We have implemented this new methodology within an octree-based finite element simulation system named *Hercules*. The only input to Hercules is material property descriptions of a problem domain; the only outputs are lightweight jpeg-formated images generated as they are simulated at every visualization time step. There is absolutely no other intermediary file I/O. Performance evaluation of Hercules on up to 2048 processors on the AlphaServer system at Pittsburgh Supercomputing Center has shown good isogranular scalability and fixed-size scalability.

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.

Keywords: physical simulations, end-to-end approach, parallel computing, finite element method, octrees, mesh generation, visualization

1 Introduction

Traditionally, parallel supercomputing has been directed at the inner kernel of physical 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 end and back end of the simulation pipeline — problem description and interpretation of the results — 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 cache-friendly and compute-intensive component, lending themselves naturally to performance evaluation of each new generation of parallel architecture; whereas the front and back ends often have sufficiently small memory footprints and compute requirements that they may be relegated to offline, sequential computation.

However as scientific simulations move into the realm of the terascale and beyond, 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 an unstructured mesh of massive size; the output interpretation phase involves unstructured-mesh volume rendering of even larger size. As the largest unstructured mesh simulations move into the multi-hundred million to billion element/grid point 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 [24, 25, 26, 28]. 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 [30], which themselves run in parallel and are highly scalable[14, 15, 17, 30]. 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.

However, despite our — and others' — best efforts at devising scalable algorithms and implementations for the meshing, solving, and visualization components, as our resolution and fidelity requirements have grown to target multi-hundred million to billion and greater element simulations, there remain the even bigger problems of storing, transferring, and reading/writing multi-terabyte files between these components. In particular, I/O of multi-terabyte files remains a pervasive and abiding performance bottleneck on parallel computers, to the extent that *the offline approach to the meshing–solving–visualizing simulation pipeline becomes intractable for billion mesh node unstructured mesh simulations.* Ultimately, beyond scalability and I/O concerns, the biggest limitation provided by the offline approach is its inability to support interactive visualization of the simulation: the ability to debug, monitor, adjust, and steer the simulation at runtime

[1]For example, in a report identifying the prospects of scalability of a variety of parallel algorithms to petascale architectures [20], mesh generation and associated load balancing are categorized as Class 2 — "scalable provided significant research challenges are overcome."

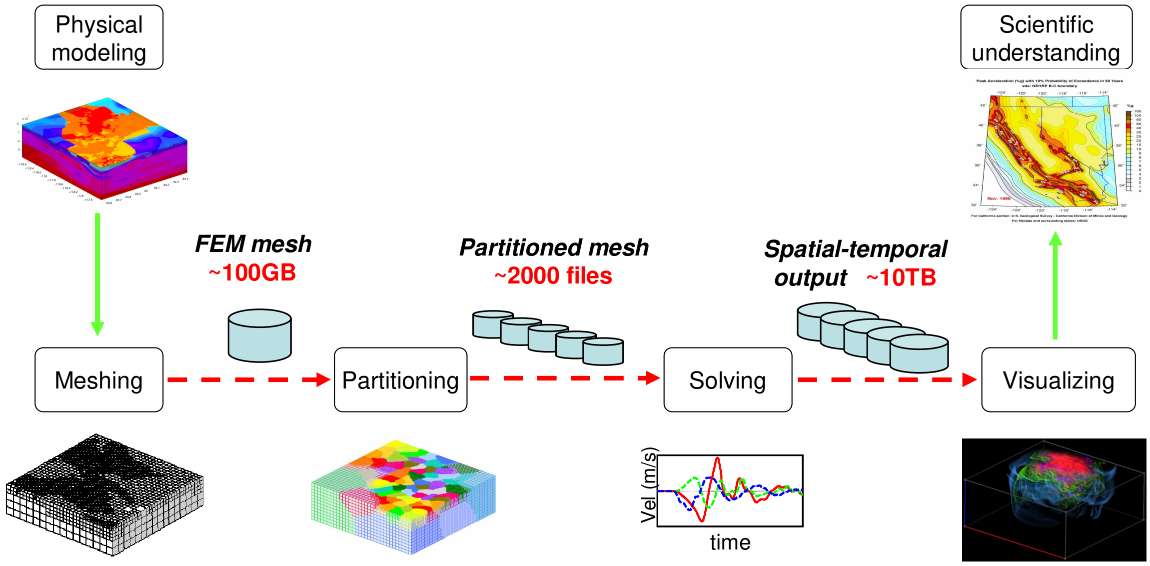


Figure 1: **Traditional simulation pipeline.**

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-based steering, we must seek an *end-to-end solution* to the meshing–solving–visualizing simulation pipeline. The key idea is to replace the traditional, cumbersome file interface with a scalable, parallel, runtime data structure that supports simulation pipelines in two ways: (1) providing a common foundation on top of which all simulation components operate, and (2) serving as a vehicle for data sharing among simulation components.

We have implemented this new methodology within a simulation system named *Hercules*, which targets unstructured octree-based finite element PDE simulations running on multi-thousand processor supercomputers. Figure 2 shows an instantiation of Hercules for our earthquake modeling problem. All simulation components, i.e. meshing, partitioning, solving, and visualizing, are implemented on top of, and operate on, a unified parallel octree data structure. There is only one executable (MPI code), in which all the components are tightly coupled and execute on the same processors. The only input is a description of the spatial variation of the PDE coefficients (a material database, in the case of the earthquake simulations); the only outputs are lightweight jpeg-formatted visualization frames generated *as they are simulated* at every visualization time step.[2] There is absolutely no other file I/O.

A quick first glance may lead to a misperception that that Hercules — which makes use of the well-known parallel octree data structure — is straightforward to implement. The fact is that we have to develop complex new mechanisms on top of the parallel octree structure to support large-scale end-to-end finite element simulations. For example, we need to associate unknowns with mesh nodes, which correspond to the vertices

[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.

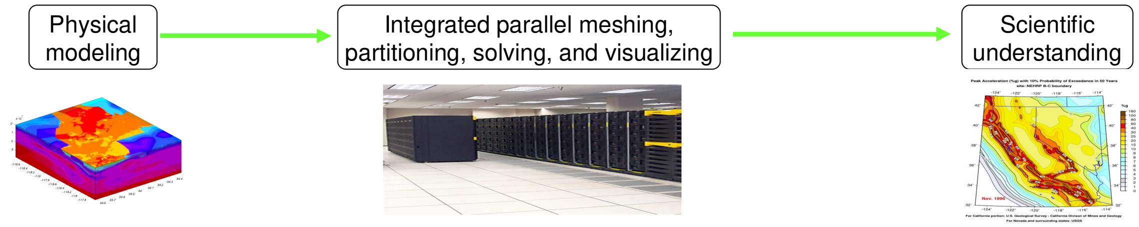


Figure 2: **Online, end-to-end simulation pipeline.**

of the octants in a parallel octree[3]. How to deal with octree mesh nodes alone represents a nontrivial challenge to meshing and solving, which demand new data structure and algorithm design. Furthermore, in order to provide unified data access services throughout the simulation pipeline, a flexible interface to the underlying parallel octree has to designed and exported such that meshers, solvers and visualizers can efficiently share simulation data.

It is worth noting that although we have integrated only visualization (Section 3.3.3) in the Hercules framework, there is no technical difficulty to plug in other post-processing tools such as spectrum analysis in place of visualization. The reason we have chosen 3D volume rendering visualization over others is that the former is by far the most demanding back end in terms of the volume of data to be processed. By showcasing that online, integrated visualization is implementable, we demonstrate that the proposed end-to-end approach is feasible for implementing a wide variety of other simulation pipeline configurations.

We have assessed the performance of Hercules on the Alpha EV68-based terascale system at Pittsburgh Supercomputing Center for our earthquake modeling problem. Preliminary performance and scalability results (Section 3.4) show:

- Fixed-size scalability of the entire end-to-end simulation pipeline from 128 to 1024 processors at 76% overall parallel efficiency for 134 million mesh node simulations
- Fixed-size scalability of the meshing and solving components from 128 to 2048 processors at combined 84% parallel efficiency for 134 million 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
- Scalability of the meshing and solving components on 2000 processors for 1.37 billion node simulations

Already we are able — we believe for the first time — to demonstrate scalability to 1024 processors of the entire end-to-end simulation pipeline, from mesh generation to wave propagation to scientific visualization, a unified end-to-end approach.

[3] In contrast, other parallel octree-based applications such as N-body simulations have no need to manipulate octants' vertices

2 Octree-based finite element method

In general, octree-based finite element method 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, octree 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 [12], while octree mesh methods for compressible flow around complex aircraft configurations provides an excellent example of geometry- and solution-driven dynamic adaptivity [29].

This section provides a brief description of octree-based finite element method in the context of earthquake ground motion modeling. Details on our computational methodology and underlying algorithms may be found in [5, 6, 7, 12].

2.1 Wave propagation equation

We model seismic wave propagation in the earth via Navier's equation of linear 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, and $\boldsymbol{L}^{AB}$ a linear differential operator that vanishes on the free surface, and applies an appropriate absorbing boundary condition on truncation boundaries. Let Ω be an open bounded domain in $\mathbb{R}^3$. The initial–boundary value problem is then written as:

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

where $\boldsymbol{n}$ represents the outward unit normal to the boundary. With this model, longitudinal waves propagate with velocity $v_p=\sqrt{(\lambda+2\mu)/\rho}$, and shear waves with velocity $v_s=\sqrt{\mu/\rho}$. 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. Explicit expressions for such a body force will be given below in the case of antiplane shear.

On a face with a unit normal $\boldsymbol{n}$ and two tangential vectors $\boldsymbol{\tau}_1$ and $\boldsymbol{\tau}_2$, such that the three vectors form a right-handed orthogonal coordinate system, the absorbing boundary condition (Stacey's formulation) takes the form

$$\boldsymbol{Sn} = \begin{bmatrix} -d_1\frac{\partial}{\partial t} & c_1\frac{\partial}{\partial \tau_1} & c_1\frac{\partial}{\partial \tau_2} \\ -c_1\frac{\partial}{\partial \tau_1} & -d_2\frac{\partial}{\partial t} & 0 \\ -c_1\frac{\partial}{\partial \tau_2} & 0 & -d_2\frac{\partial}{\partial t} \end{bmatrix} \begin{Bmatrix} u_n \\ u_{\tau_1} \\ u_{\tau_2} \end{Bmatrix} \equiv \boldsymbol{L}^{AB}\boldsymbol{u}$$

where $\boldsymbol{S}$ is the stress tensor and

$$\begin{aligned} c_1 &= -2\mu + \sqrt{\mu(\lambda+2\mu)}, \\ d_1 &= \sqrt{\rho(\lambda+2\mu)}, \\ d_2 &= \sqrt{\rho\mu}. \end{aligned}$$

Even though Stacey's absorbing boundary is not exact, it is local in both space and time, which is particularly important for large-scale parallel implementation.

2.2 Octree-based spatial discretization

We apply standard Galerkin finite element approximation in space to the appropriate weak form of the initial-boundary value problem (1). Let $\mathcal{U}$ be the space of admissible solutions (which depends on the regularity of $\boldsymbol{b}$), $\mathcal{U}_h$ be a finite element subspace of $\mathcal{U}$, and $\boldsymbol{v}_h$ be a test function from that subspace. Then the weak form is written as follows.
Find $\boldsymbol{u}_h \in \mathcal{U}_h$ such that

$$\int_{\Omega} \left\{ \rho\ddot{\boldsymbol{u}}_h \cdot \boldsymbol{v}_h + \frac{\mu}{2}\left(\boldsymbol{\nabla u}_h + \boldsymbol{\nabla u}_h^{\mathsf{T}}\right) \cdot \left(\boldsymbol{\nabla v}_h + \boldsymbol{\nabla v}_h^{\mathsf{T}}\right) + \lambda(\boldsymbol{\nabla}\cdot\boldsymbol{u}_h)(\boldsymbol{\nabla}\cdot\boldsymbol{v}_h) - \boldsymbol{b}\cdot\boldsymbol{v}_h \right\} d\Omega = \int_{\partial\Omega} (\boldsymbol{L}^{AB}\boldsymbol{u}_h)\cdot\boldsymbol{v}_h \, dA, \quad \forall \boldsymbol{v}_h \in \mathcal{U}_h. \tag{2}$$

Spatial approximation is effected via piecewise trilinear basis functions and associated trilinear hexahedral elements on an octree mesh. This strikes a balance between simplicity, low memory (since all element stiffness matrices are the same modulo scale factors), and reasonable accuracy.

Upon spatial discretization, we obtain a system of ordinary differential equations of the form

$$\boldsymbol{M}\ddot{\mathbf{u}} + \left(\boldsymbol{C}^{AB} + \alpha\boldsymbol{M} + \beta\boldsymbol{K}\right)\dot{\mathbf{u}} + \left(\boldsymbol{K} + \boldsymbol{K}^{AB}\right)\mathbf{u} = \mathbf{b}, \tag{3}$$

where $\boldsymbol{M}$ and $\boldsymbol{K}$ are mass and stiffness matrices, arising from the terms involving ρ and (μ,λ) in (2), respectively; $\mathbf{b}$ is a body force vector resulting from a discretization of the seismic source model; and damping matrices $\boldsymbol{C}^{AB}$ and $\boldsymbol{K}^{AB}$ are contributions of the absorbing boundaries to the mass and stiffness matrices, respectively. We have also introduced damping matrices in the form of the Rayleigh material model $\alpha\boldsymbol{M}+\beta\boldsymbol{K}$ to simulate the effect of energy dissipation and resulting wave attenuation due to anelastic material behavior. The constants α and β are determined locally (elementwise) so that the resulting damping ratio is as close as possible to a constant value dictated by the local soil type, over a band of frequencies. Since Rayleigh damping increases both linearly and inversely with frequency, we seek a least squares solution to this optimization problem over each element. This provides a reasonable damping model for many soils, although very low and very high frequencies are overdamped.

Spatial discretization via refinement of an octree produces a non-conforming mesh, resulting in a discontinuous displacement approximation. Whenever a refined hexahedron (octant) neighbors an unrefined one, dangling nodes, which belong to refined elements but not to unrefined neighbors, are produced. In such cases, we restore continuity of the displacement field by imposing algebraic constraints that require the displacements at dangling mesh nodes to be consistent with anchored neighbors. For linear hexahedra and provided *the 2-to-1 constraint* is enforced between neighbors, these constraints state simply that hanging mid-edge values must be the average of the two anchored endpoint vertices, and hanging mid-face values must be the average of the four anchored vertex neighbors. We can express these discrete continuity constraints in the form

$$\boldsymbol{u} = \boldsymbol{B}\bar{\boldsymbol{u}},$$

where $\bar{\boldsymbol{u}}$ denotes the displacements at the independent anchored mesh nodes, and $\boldsymbol{B}$ is a sparse constraint matrix. In particular, $\boldsymbol{B}_{ij} = \frac{1}{4}$ if (dependent) dangling mesh node i is a face neighbor of (independent) anchored node j and $\frac{1}{2}$ if it is an edge neighbor, $\boldsymbol{B}_{ij} = 1$ simply identifies an anchored node, and $\boldsymbol{B}_{ij} = 0$ otherwise. Rewriting the linear system (5) as

$$\boldsymbol{A}\boldsymbol{u}_{k+1} = \boldsymbol{b}(\boldsymbol{u}_k),$$

we can impose the continuity constraints via the projection

$$\boldsymbol{B}^\mathsf{T}\boldsymbol{A}\boldsymbol{B}\bar{\boldsymbol{u}} = \boldsymbol{B}^\mathsf{T}\boldsymbol{b}(\boldsymbol{u}_k). \tag{4}$$

The constrained update (4) remains explicit, since the projected matrix $\boldsymbol{B}^\mathsf{T}\boldsymbol{A}\boldsymbol{B}$ preserves the diagonality of $\boldsymbol{A}$. The work involved in enforcing the constraints is proportional to the number of dangling mesh nodes, which can be a sizable fraction of the overall number of mesh nodes for a highly irregular octree, but is at most of $\mathcal{O}(N)$. Therefore, the per-iteration complexity of the update (4) remains linear in the number of mesh nodes.

2.3 Temporal approximation

The time dimension is discretized using central differences. The algorithm is made explicit using a diagonalization scheme that lumps the mass matrix—and possibly $\boldsymbol{C}^{AB}$—and splits the diagonal and off-diagonal portions of the stiffness and absorbing boundary damping matrix. The resulting update for the displacement field at time step $k+1$ is given by

$$\begin{aligned}\left[\left(1+\alpha\frac{\Delta t}{2}\right)\boldsymbol{M} + \beta\frac{\Delta t}{2}\boldsymbol{K}_{diag} + \frac{\Delta t}{2}\boldsymbol{C}^{AB}_{diag}\right]\mathbf{u}_{k+1} = \\ \left[2\boldsymbol{M} - \Delta t^2\left(\boldsymbol{K}+\boldsymbol{K}^{AB}\right) - \beta\frac{\Delta t}{2}\boldsymbol{K}_{off} - \frac{\Delta t}{2}\boldsymbol{C}^{AB}_{off}\right]\mathbf{u}_k \\ + \left[\left(\alpha\frac{\Delta t}{2}-1\right)\boldsymbol{M} + \beta\frac{\Delta t}{2}\boldsymbol{K} + \frac{\Delta t}{2}\boldsymbol{C}^{AB}\right]\mathbf{u}_{k-1} + \Delta t^2\mathbf{b}_k.\end{aligned} \tag{5}$$

The time increment Δt must satisfy a local CFL condition for stability. Space is discretized over an octree mesh (each leaf corresponds to a hexahedral element) that resolves local seismic wavelengths: given a (typically highly-heterogeneous) material property distribution and highest resolved frequency of interest, a local mesh size is chosen to produce p mesh nodes per shortest wavelength (we typically take $p = 10$ for trilinear hexahedra). This insures that the CFL-limited time step is of the order of that needed for accuracy, and that excessive dispersion errors do not arise due to over-refined meshes.

2.4 Summary

Due to the trilinear hexahedral elements and local dense element-based data structures, octree-based finite element method has several important advantages:

- The hexahedral meshes stem from wavelength-adapted octrees, which are more easily generated than general unstructured tetrahedral meshes, particularly when the number of elements increases above 50 million.
- The hexahedra provide somewhat greater accuracy per mesh node (the asymptotic convergence rate is unchanged, but the constant is typically improved over tetrahedral approximation).
- The element-based data structure 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.
- The hexahedra all have the same element stiffness matrices, modulo 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 grid-point-based tetrahedral code.

These features permit earthquake simulations to substantially greater resolutions than heretofore possible.

3 The Hercules system

Motivated to overcome the many pitfalls of the traditional, offline, file-based approach described in Section 1, we have adopted an end-to-end, approach to parallel supercomputing and developed a new octree-based finite element simulation system named Hercules. This section presents the design, implementation, performance assessment of the Hercules system.

3.1 Data structures

The design goal of Hercules is to have all the simulation components operate on and share data from a consistent, overarching data structure: a parallel octree, which is the backbone that ties all add-on pieces (i.e., data structures and algorithms) together.[4] There are three distinctive advantages of using a parallel octree as the backbone: (1) octrees are simple, scalable, hierarchical structures, (2) algorithms on octrees have been well studied and successfully applied, and (3) we can exploit interesting properties of octrees for parallelism.

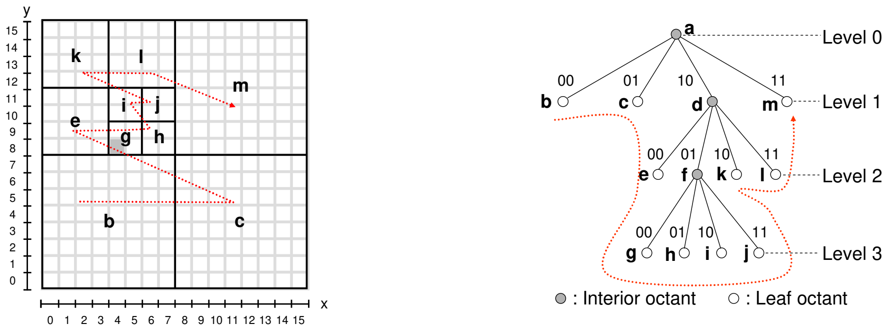
Before explaining how we organize the backbone parallel octree structure, let us first examine the important properties of octrees. For simplicity of illustration, we use two-dimensional quadtrees and quadrants in the

[4]The add-ons either implement general operations on the underlying parallel octree or realize specific functions of a simulation component.

figures and examples. All the techniques and properties are applicable to three-dimensional 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).

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 3(a) and (b) show the domain representation and the equivalent tree representation of an octree, respectively. Each tree edge in Figure 3(b) is labeled with a binary **directional code** that distinguishes each child of an internal octant.



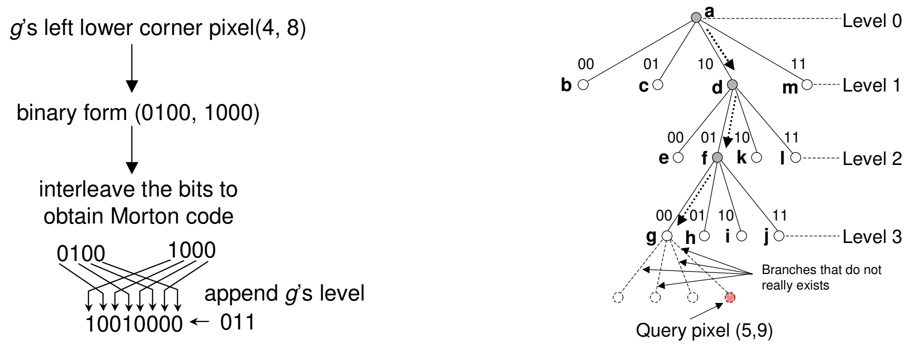
(a) Domain representation

(b) Tree representation

Figure 3: **Equivalent representations of an octree.**

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, 10, 11]. 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 4(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 [19]. Then append the octant's level to compose the locational code. We refer to the lower left pixel of an octant as the octant's **anchor**. For example, the shaded pixel in Figure 3(a) is the anchor for octant g.

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 4(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



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

Figure 4: **Operations on octrees.**

octant is an aggregating ancestor of a non-existent octant.

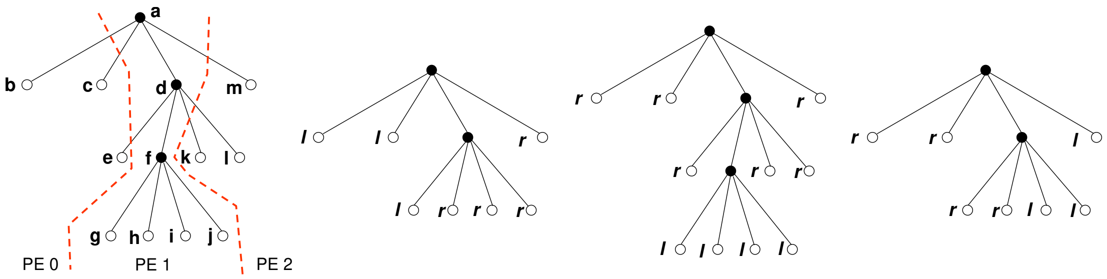
Z-ordering. If we sort all the leaf octants of an octree according to their locational codes, we obtain a total ordering of all the leaf octants. Given the encoding scheme of locational code, it is not difficult to verify that the total ordering is identical to the *pre-order traversal* of the leaf octants of the octree (See Figure 3). 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** [9], 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.

3.1.1 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(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(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).

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



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

Figure 5: **Parallel octree organization on 3 processors.** Circles marked by `l` represent local leaf octants; and those marked by `r` represent aggregated remote leaf octants.

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 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.

3.1.2 Locational code lookup table

Searching for an octant is an indispensable operation frequently invoked by the various simulation components. 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 the search operation is initiated, then we follow standard octree search algorithms [21] 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 neighboring octant of g to its left (see Figure 3(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.

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.

Simulation components need to operate on the underlying octree to implement their respective functions. For example, a mesher needs to refine or coarsen the tree structure to carry out spatial discretization as dictated by material properties. A solver needs to attach runtime solution results to mesh nodes; and a visualizer needs to consume the attached attached. In order to support 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, 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. Such an interface allows us to encapsulate the complexity of manipulating the backbone parallel octrees within the abstract data types.

Note that there is one (and 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 buffer 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 still have to carried out through a pre-defined API call.

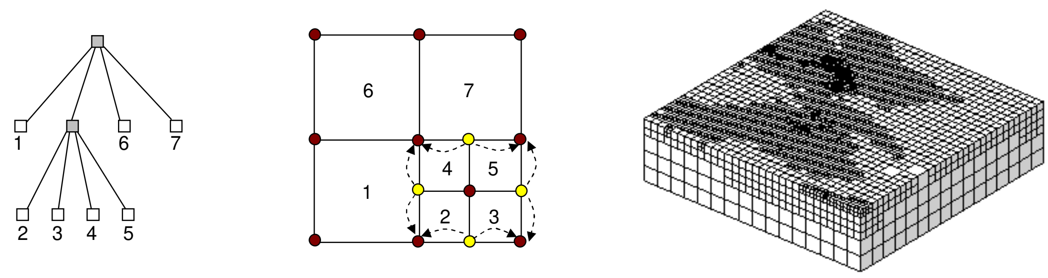
We have also designed binding interfaces between the simulation components. However, unlike the octree/octant interface, the inter-component interfaces can only be clearly explained in the context of the simulation pipeline. Therefore, we embed the description of the inter-component interfaces in the next subsections where we cover 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

Conceptually, generating or adapting an octree-based hexahedral mesh is straightforward. As shown in Figure 6, 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.



(a) (b) (c)

Figure 6: **Octree-based hexahedral meshes.** (a) Octree domain decomposition. (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.

Following an end-to-end approach, we generate octree meshes online *in-situ* [27]. 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 produced where they will be used instead of on remote processors. Such an *in-situ* strategy requires that mesh partitioning becomes an integral part of the meshing component. The partitioning method we used is simple [4, 8]. We sort all the octants in ascending Z-order and divide them into each length chunks in such a way that each processor will be assigned one and only one chunk of the octants. Because the Z-ordering of the leaf octants corresponds exactly to the pre-order traversal of an octree, the partitioning and data re-distribution often only involve leaf octants migration between adjacent processors. Whenever data migration occurs, local instances of participating processors need to be adjusted accordingly to maintain a consistent global data structure. As will be shown in Section 3.4, such a simple strategy works well and yields almost ideal speedup for solving 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 only responsible 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 elements and nodes information and determine the various correlations 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.

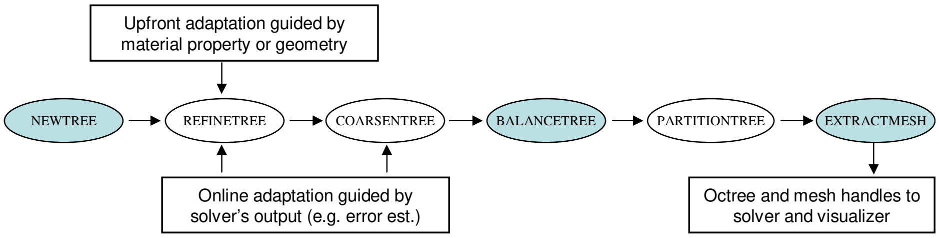


Figure 7: **Meshing component.** Shaded ovals are required steps. Unshaded ovals are optional steps.

Using the well-known octree algorithms [21], we have been able to implement the NEWTREE , REFINETREE ,COARSENTREE , and PARTITIONTREE steps in a simple way. However, the parallel octree alone — though scalable and elegant for locating octants and distributing workloads — is not sufficient for implementing the BALANCETREE and EXTRACTMESH steps.

The key challenge here is how to deal with octants' vertices, i.e. mesh nodes. We can easily compute the coordinates of the vertices based on the configurations of the local instances in parallel. But we can only obtain a collection of geometry objects (octants and vertices), which by themselves are not a finite element mesh yet. To generate a mesh *in-situ* and make it usable to a solver, we must identify the correlations between octants and vertices (mesh connectivity), and between vertices and vertices, either on the same processor (dangling-to-anchored dependences) or on different processors (inter-processor sharing information). Therefore, in order to implement the BALANCETREE and EXTRACTMESH steps, which require capabilities beyond those offered by standard parallel octree algorithms, we have incorporated auxiliary data structures such as hash tables and double-link lists, and develop new algorithms such as *parallel ripple propagation* and *parallel octree bucket sorting* for meshing. These add-ons are internal and are not visible to other simulation components downstream in the pipeline.

As we mentioned earlier, the interface between simulation components provides the glue that ties the Hercules system together. The interface between the meshing and solving 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 the solver. The `mesh_t` ADT contains all the information a solver would need to initialize its 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 carry out online mesh adaptation, which is critical for including inverse solvers in the Hercules system in the future.

3.3.2 Solving

Figure 8 shows the solving 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 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 solving component, which repeats the same procedure for the next time step until done. The solving component has the optimal complexity of $\mathcal{O}(N^{\frac{4}{3}})$, where $N^{\frac{1}{3}}$ is the number of mesh node in each direction. This results 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.

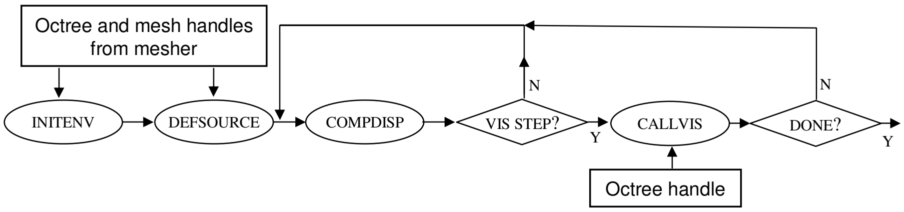


Figure 8: **Solving component.**

The COMPDISP step (conducing local element-wise dense matrix computation, exchanging data between processors, averaging dangling node values, etc.) presents no major technical difficulty, since the inner kernel is by far the most well studied and understood part. More interesting is how the solving component interacts with other simulation components and with the underlying octree in the INITENV ,DEFSOURCE , and CALLVIS steps.

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 correlation, dangling-anchored node correlation), and the sharing information (which processor shares which of my local mesh nodes), and so forth. Thus, all initialization work, including the setup of a communication schedule, can be performed in parallel without any communication among processors.

Along with the `mesh_t` ADT, the solving component 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). In Hercules, 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 convert 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 same technique for single point source multiple times. In other words, whatever kinematic source is involved, we always have to locate the enclosing elements of arbitrary coordinates. Hence, we are able to implement the DEFSOURCE step using the octree/octant interface, which provides such important services as searching for octants.

The other important application of the `octree_t` ADT is to serve as a vehicle for a solver to pass data to a visualizer. Recall that we have reserved a place-holder in the `octree_t` ADT. Thus, we allocate a buffer that holds the results of the computation (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 memory copying, we do not store 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 visualizer's perspective, the solving component has provided a nifty data service interface. 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.

A side note: When a time step does not need to be visualized, no data access macros are called; thus no memory access or computation overhead occurs.

3.3.3 Visualizing

Simulation-time 3D volume rendering has rarely been attempted in the past for three major reasons. First, scientists are reluctant to use the precious supercomputing hours for visualization. Second, data organization designed for the solving component is generally very different from what is needed by rendering algorithms. Third, executing a visualization pipeline on a different set of processors causes difficult communication problems and increase the complexity of a simulation code.

By taking an online, end-to-end approach, we have been able to incorporate a highly adaptive parallel visualizer into Hercules, which executes the meshing, solving, visualizing components all on the same set of processors.

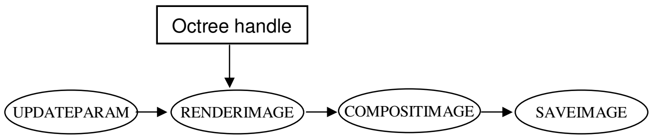


Figure 9: **Visualizing component.**

Figure 9 shows how the visualizing component works in Hercules. First, the UPDATEPARAM step updates the viewing and rendering parameters. [5] 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 [17, 30]. The rendered (partial) images are then composited together in the COMPOSITIMAGE step. Different from most other the parallel image compositing algorithms that are designed for a specific

[5] In our current implementation, we fix the parameters and hence the UPDATEPARAM step is executed only once when the visualizer is bootstrapped. However, when we incorporate online steering in the future, UPDATEPARAM will need to be executed for each visualization step.

network topology [2, 13, 16], we have made use of the scheduled linear image compositing (SLIC) [22] technique, which has proved to be the most flexible and efficient parallel image compositing algorithm. Finally, the SAVEIMAGE step stores an image to disk. Figure 10 shows a sequence of example images). The cost of the visualizing component per invocation is $\mathcal{O}(xyE^{\frac{1}{3}}\log E)$, where x, y represent the two-dimensional image resolution and E is the number of mesh elements.

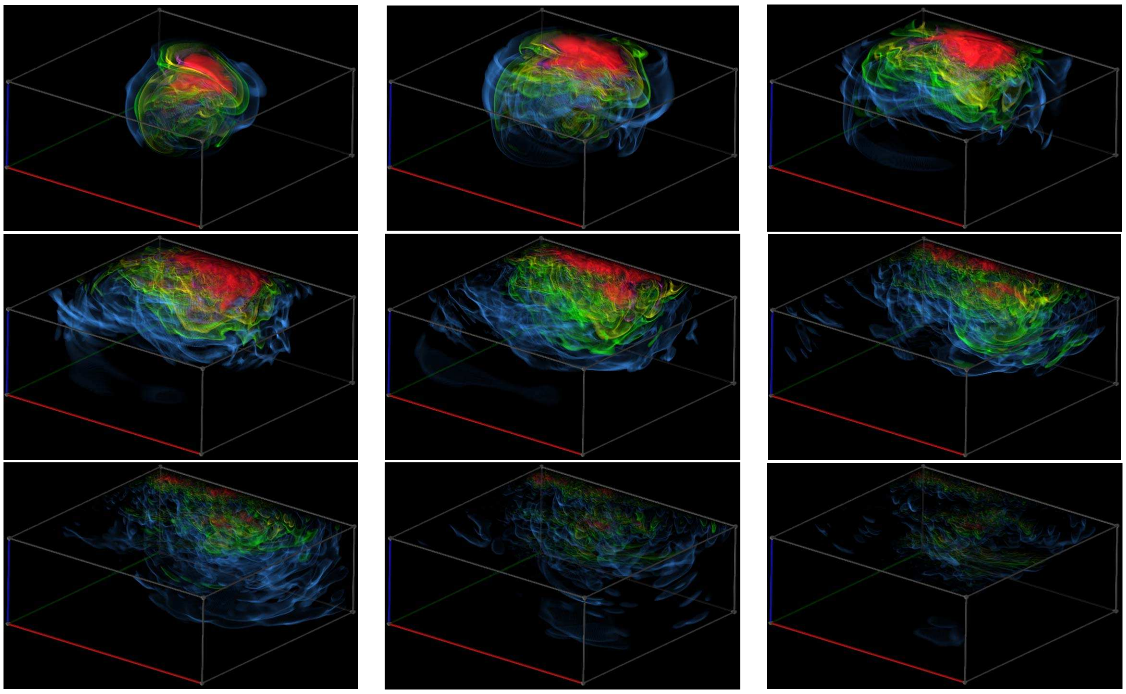


Figure 10: **A sequence of snapshot images of propagating waves of 1994 Northridge earthquake.**

The visualizing component relies on the underlying parallel octree to (1) retrieve simulation data from the solver, and (2) implement its adaptive rendering algorithm. We have explained the first point in the previous subsection. Now let us take a closer look at the second point. Our visualizing component needs to traverse the octree structure to implement its ray-casting based rendering algorithm. By default, we have to process all leaf octants a particular ray shoots through in order project a pixel. However, we might not always want to render at the highest resolution. For example, when rendering hundreds of millions of elements to a small image with 512×512 pixels, it would not reveal much more perceivable details if we render at the highest resolution level, unless when a close-up view is selected. So in order to achieve better performance of rendering without compromising the quality of the images, we perform a view-dependent pre-processing step to choose an appropriate octree level before actually rendering an image [30]. Operationally, it means that we may need to move up in the octree structure and render images at a coarser level. Again, the small set API functions to manipulate the backbone octree comes to serve as the critical building block, this time, for supporting adaptive visualization.

3.4 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, which have been identified in the process of understanding the behavior of Hercules as an end-to-end simulation system.

Our simulations have been conducted to model seismic wave propagations during historical and synthetic earthquakes in the Greater Los Angeles Basin, which comprises a three-dimensional volume of 100 by 100 by 37.5 kilometers. The performance data presented was collected from Lemieux, the HP AlphaServer system at Pittsburgh Supercomputing Center. The execution time is obtained by measuring the wallclock time as returned by the `MPI_Wtime()` function call. Other performance data, including Mflops, processor cycles, cache misses, and TLB misses are obtained from the HP Digital Continuous Profiling Infrastructure (DCPI).[6]

The material property model we used to drive our simulations is the Southern California Earthquake Center 3D community velocity model [18] (Version 3, 2002), known as the SCEC CVM model. The model is a standalone Fortran program that takes a list of coordinates (longitude, latitude, depth) as input, and produces a list of corresponding records (i.e., primary velocity, shear velocity, and density) as output. However, the execution model is not suitable for online unstructured mesh generation on parallel computers due to the unknown coordinates. We discretize the domain at runtime to figure out the coordinates that need to be further queried from the model. In other words, we have to query the physical model interactively rather than in a batch. Unfortunately, we are unable to switch context to run the Fortran program on the compute nodes at runtime. [7] To solve this problem and enable our online, end-to-end simulations, we query the SCEC CVM model at very high-resolution in advance offline, and then compress, store and index the results in a material database [23] (approx. 2.5GB in size). Note that this is a one-time effort. The database thus generated can be used repeatedly by many simulations. In our initial implementation, we let all processors query a single material database stored on a parallel file system. But the performance was unacceptable, especially when the number of processor exceeds 16. As a result, we have modified our implementation to replicate the material database onto the local disk attached to each compute node before a simulation. We note that this is not a particularly restrictive design choice since most cluster-based supercomputers usually have sufficiently large local disks for compute nodes.

3.4.1 Isogranular scalability study

Our primary focus is to understand how the Hercules system performs when we increase the problem size and the number of processors, and maintain more or less the same amount of work on each processor.

Figure 11 summarizes the characteristics of the isogranular experiments. “PEs” shows the number of processors used in each simulation, which is denoted by the “Frequency” of the seismic wave to be resolved. “Element”, “Nodes”, “Anchored” and “Dangling” show the statistics of the octree-based hexahedral finite element meshes. “Max leaf level” and “Min leaf level” represent the smallest and largest elements in the meshes, respectively. “Elements/PE” is used as a rough indicator of the workload on each processor. Given

[6]Due to unknown reasons that are still being investigated, DCPI has caused significant execution time slowdown on larger number of processors.

[7]In fact, even if we could fork a process to run the Fortran program, the bootstrapping cost would be forbiddingly expensive (on the order of the seconds).

the unstructured nature of the finite element meshes, it is impossible to guarantee the per-processor element (node) number to be exactly the same over different simulation runs. Nevertheless, we have contained the difference to within 10%. "Steps" shows the number of simulation steps Hercules executes. The "E2E time" represents the absolute running time of a Hercules simulation from the beginning when the code is loaded onto a supercomputer to the end when the code exits the system. The running time includes the time of replicating a material database stored a shared parallel file system to the local disk attached to each compute node ("Replicating"), the time to generate an *in-situ* unstructured finite element mesh ("Meshing"), the time to simulate seismic wave propagation ("Solving"), and the time to create visualizations and output jpeg images ("Visualizer"). "E2E time/step/elem" and "Solver time/step/elem" are the amortized cost per element per time step for end-to-end time and solving time, respectively. "Mflops/sec/PE" stands for the sustained megaflops per second per processor.

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
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
Steps	2000	4000	10000	8000	2500	2500
E2E time (sec)	12911	19804	38165	48668	13033	16709
Replicating (sec)	22	71	85	94	187	251
Meshing (sec)	20	75	128	150	303	333
Solving (sec)	8381	16060	31781	42892	11960	16097
Visualizing (sec)	4488	3596	6169	5528	558	*
E2E time/step/elem (μs)	9.769	7.984	7.927	7.856	8.436	*
Solver time/step/elem (μs)	6.341	6.475	6.601	6.924	7.741	*
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 have not yet been obtained due to either supercomputer scheduling or performance measurement problems.

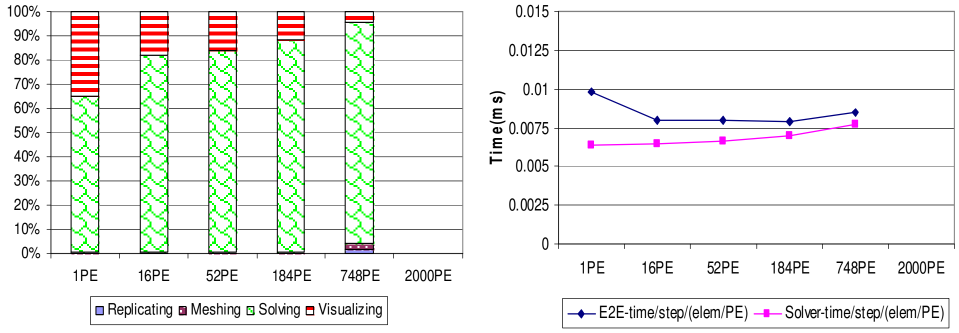
We can see from the table that the simulations involved highly unstructured meshes, with the largest elements being 64 times as large in edge size as the smallest ones. Because of the multi-resolution of the meshes, there are many dangling nodes, which account for 11% to 20% of the total mesh nodes.

A traditional way to assess the overall isogranular parallel efficiency is examine the degradation of the sustained average Mflops per processor. In our case, we achieve 28% to 33% of the peak performance (2 GFlops/sec/PE) on the Alpha EV68 processors.[8] However, there is no degradation in the sustained average floating-point rate. On the contrary, the Mflops/sec/PE increases as we solve larger problems on larger numbers of processors. We were initially puzzled by this counter-intuitive observation. But a careful second

[8]These are respectable numbers as compared to less than 10% reported by most ASCI applications.

thought reveals an interesting explanation: Solving is the most computation-intensive and time-consuming component; as the problem size increases, processors spend more time in the solving component executing floating-point instructions, thus boosting the overall Mflops per processor rate.

Therefore, in order to assess the isogranular parallel efficiency in a more meaningful way, we have turned to analyze the running times. Figure 12(a) shows how each component of Hercules contribute to the total running time in percentage. From bottom-up, it shows the contribution by the "replicating", "meshing", "solving", and "visualizing", respectively. The one-time costs such as replicating material database and generating a mesh are so inconsequential that they are almost invisible.[9] On the other hand, since the visualizing component has much better per time step time complexity as compared with that of the solving component ($\mathcal{O}(xyE^{\frac{1}{3}} \log E)$ vs. $\mathcal{O}(N)$, where x and y are the 2D image resolution, E and N are the numbers of mesh elements and nodes, respectively, and N is always greater than E in an octree-based hexahedral mesh), as the problem size (E and N) and the number of processors increase, the solving time overwhelms the visualizing time by larger and larger margins.



(a) Running time percentage breakdown. (b) Amortized running time.

Figure 12: **The overall isogranular scalability characteristics of the Hercules system.** (a) The percentage contribution of each simulation component to the total running time. (b) The amortized running time per processor per step. The top curve corresponds to the amortized end-to-end running time and the lower corresponds to the amortized solver running time.

Figure 12(b) shows the trend of the amortized end-to-end running time and solving time per time step per element. Although the end-to-end time is always higher than the solving time, as we increase the problem size, the end-to-end time curve is pulled towards closer to the solving time curve because the latter becomes more and more dominant, . The key insight here is that the limiting factor of achieving high isogranular scalability on a large number of processors is the scalability of the solver proper, rather than other simulation components that we have bundled with the solving component. Therefore, it is reasonable to use the degradation in solving time to measure the isogranular efficiency of the entire end-to-end system. As shown in Figure 11, the solving time per step per element is 6.341μs on a single PE and 7.741μs on 748 PE. Therefore, we have an isogranular parallel efficiency around 81%, a very good result considering the high irregularity of the meshes.

A side note: we have successfully run the meshing and solving components to simulate the 2 Hz prob-

[9] For the 1.5 Hz simulation (748-PE run), we have only simulated 2,500 time steps. The "replicating" and "meshing" cost appear to be slightly more significant percentage-wise. But a full-scale simulation of 10,000 time steps would wipe out any residual percentage contribution due to "replicating" or "meshing".

lem (1.37 billion mesh nodes). However, in this case, the isogranular parallel efficiency drops to 60% as compared with the 1 PE run. We are actively investigating the reason for the performance degradation.

3.4.2 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 13.

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

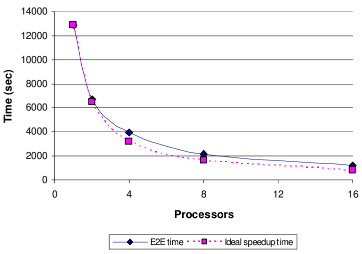
Figure 14 shows the performance of Hercules for different fixed-sized problems. Each column represents the results for a set of fixed-size experiment. From left to right, we display the speedup plots for the small case, medium case and large case, respectively.

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 skirts the ideal speedup curve very closely. The end-to-end parallel efficiencies on 16$\times$ processors are 66%, 76%, and 64%, for the small case (1 PE vs. 16 PEs), medium case (8 PEs vs. 128 PEs), and large case (128 PEs vs. 2048 PEs), respectively.

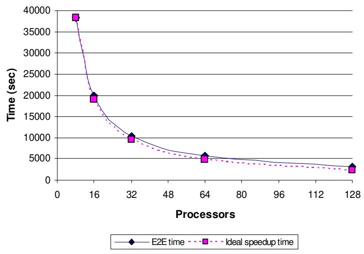
The second row shows the performance of the meshing component only. Although not perfect, the meshing component achieves reasonable speedups while running on a large number of processors. In fact, if a mesh is only generated statically up front — that is, before the computation starts — and does not change over time, the cost of the meshing is completely amortized by thousands of simulation time steps as long as the mesh is generated in a reasonable amount of time. This is exactly the case for our forward earthquake simulations. To appreciate the efficiency of the meshing component: assume that a 1 Hz mesh (7.6 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 60 seconds. In comparison, the meshing component generates the mesh on 2048 processors *in-situ* in 35 seconds.

An interesting question is how the meshing component would perform if dynamic mesh adaptation is needed[10]. In this case, the speedup of the meshing component on larger numbers of processors becomes important. In a detailed study of the meshing component [27], we have identified that the least scalable and the most time-consuming (approx. 60% of total meshing time) operation of meshing is the PARTITIONTREE step. This step migrates data among processors, and is likely to put stress on the bandwidth of the intercon-

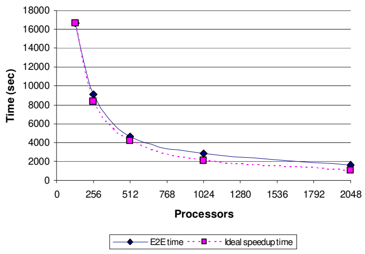
[10]We have not yet evaluated this case empirically.



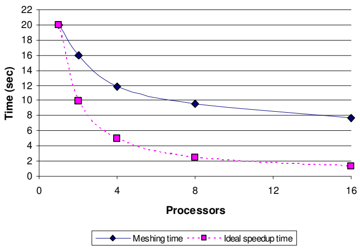
Small case end-to-end speedup.



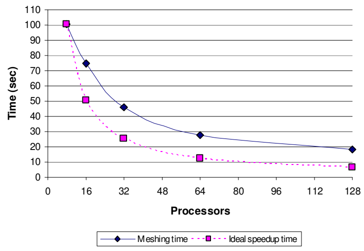
Medium case end-to-end speedup.



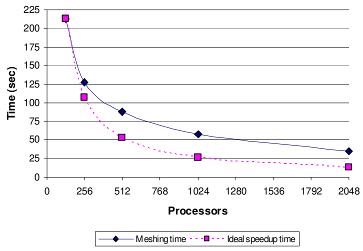
Large case end-to-end speedup.



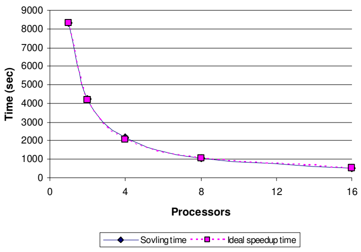
Small case meshing speedup.



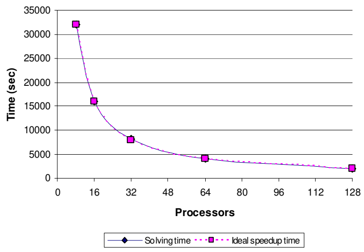
Medium case meshing speedup.



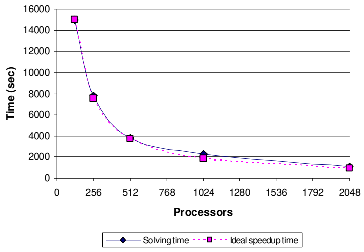
Large case meshing speedup.



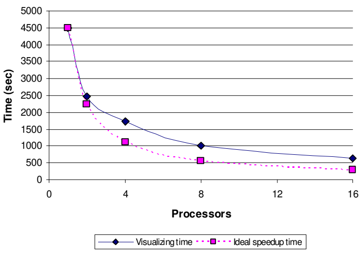
Small case solving speedup.



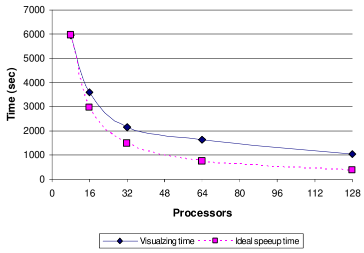
Medium case solving speedup.



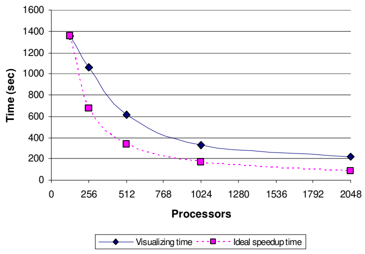
Large case solving speedup.



Small case visualizing speedup.



Medium case visualizing speedup.



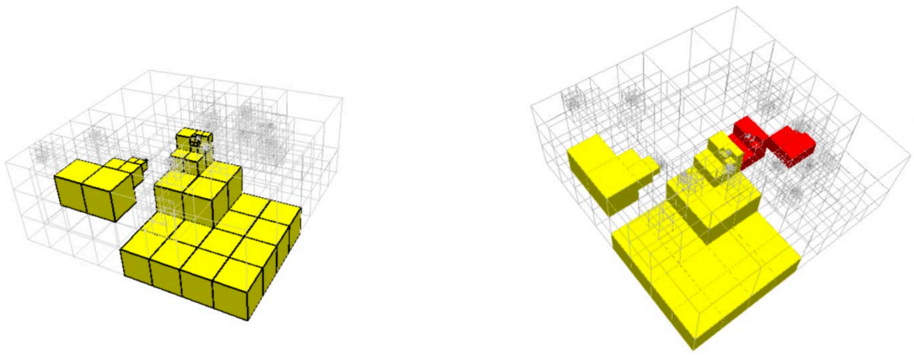
Large case visualizing speedup.

Figure 14: **Speedups of fixed-size experiments.** The first row represents the end-to-end running time; the second the meshing and partitioning time; the third the solving time; and the fourth the visualizing time.

nect network and the memory subsystem. However, if each processor has more or less the same number of elements, which is probably true when dynamic mesh adaptation occurs, the cost of PARTITIONTREE would be significantly reduced. Therefore, we expect the speedup of the meshing component would be much better in the dynamic adaptation cases. However, further research is needed to substantiate this speculation.

The third row of Figure 14 shows a somewhat surprising result: the solving component achieves almost perfect speedup on hundreds and thousands of processors, even though the partitioning strategy we used (dividing a Z-ordered sequence of elements into equal chunks) is extremely simple. In fact, the solving component's parallel efficiency on 16× processors is 97%, 98%, and 86%, for the small case (1 PE vs. 16 PEs), medium case (8 PEs vs. 128 PEs), and large case (128 PEs vs. 2048 PEs), respectively. Since solving is the most dominant component of the Hercules system, its high fixed-size parallel efficiency has obviously promoted the performance of the entire end-to-end system.

The speedup of the visualizing component, as shown in the fourth row of Figure 14, is however less satisfactory, even though the general trend of the running time indeed shows improvement as more processors are used. But since this component is executed at each visualization time step (usually every 10th simulation time step), the less than optimal speedup actually has a much bigger impact on the overall end-to-end performance than the meshing component. The visualizing parallel efficiency on 16× processors is actually 44%, 36%, and 38%, for the small case (1 PE vs. 16 PEs), medium case (8 PEs vs. 128 PEs) and large case (128 PEs vs. 2048 PEs), respectively.



(a) (b)

Figure 15: **Workload distribution.** (a) Elements assigned on one processor. (b) Unbalanced visualization workload on two processors.

The visualizing performance degradation has nothing to do with our rendering and compositing algorithm. Rather, it is caused by our simplistic partitioning strategy, which assigns equal number of elements to each processor. At a visualization step, each processor renders its local data associated with the elements (octants) hosted on that processor, as shown in Figure 15(a). But two processors may have dramatically different data block sizes and the resulting projected image sizes, as shown in Figure 15(b) where the light blocks represent elements assigned to one processor and the dark blocks another processor. The net result of such discrepancy in size is that the workload may become highly unbalanced for the RENDERIMAGE and COMPOSITIMAGE steps, especially, when larger numbers of processors are involved.

In our current implementation, we do not attempt to re-distribute the data blocks since the visualizing time only accounts for a small fraction of the total simulation time. Our partitioning strategy has been chosen in favor of the solving component. However, in the future when very sophisticated rendering techniques, such as time-accurate vector field visualization, is to be included, the dynamics of the system may change in such a way that visualizing may become as time-consuming as solving. So one interesting research topic is to re-evaluate the space-filling curve based partitioning strategy and develop a hybrid scheme that can benefit both the solving and visualizing components.

4 Conclusion

We have demonstrated that an end-to-end approach to parallel supercomputing is not only desirable, but also feasible for high-performance physical simulations. By eliminating the traditional, cumbersome file interface, we have been able to turn "heroic" runs — large-scale simulation runs that often require days or even weeks of preparations — into daily exercises that can be conveniently launched 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 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 only one framework that targets octree-based finite element simulations for earthquake modeling, the basic principles and design philosophy should be applicable to other types of large-scale physical simulations. For example, for tetrahedral mesh simulations, some type of scalable hierarchical structure — which serves a similar function as a parallel octree in the Hercules system — should be designed and implemented, though such an endeavor is of course much more challenging than supporting octree-based hexahedral mesh simulations.

Our new approach also calls for new ways of assessing high-performance computing. We need to take into account all the simulation components instead of merely the inner kernel of the solvers. No time — either used by processors or human beings — should be excluded in the evaluation of the effectiveness of a simulation system. After all, the turnaround time is *the* most important performance metric for real-world scientific and engineering simulations. Sustained average Mflops of inner kernels only helps explain the achieved high performance (faster running time). They should not be treated as the only indicator of high performance or scalability.

Acknowledgments

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, Nov. 2003. Gordon Bell Award for Special Achievement.

[4] 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.

[5] H. BAO, J. BIELAK, O. GHATTAS, L. KALLIVOKAS, D. O'HALLARON, J. SHEWCHUK, AND J. XU, *Earthquake ground motion modeling on parallel computers*, in Proc. Supercomputing '96, Pittsburgh, PA, Nov. 1996.

[6] ——, *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] J. BIELAK, L. KALLIVOKAS, J. XU, AND R. MONOPOLI, *Finite element absorbing boundary for the wave equation in a halfplane with an application to engineering seismology*, in Proc. of the Third International Conference on Mathematical and Numerical Aspects of Wave Propogation, Mandelieu-la-Napule, France, Apr. 1995, INRIA-SIAM, pp. 489–498.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] ——, *Parallel visualization of large-scale aerodynamics calculations: A case study on the cray t3e*, in Proceedings of 1999 IEEE Parallel Visualization and Graphics Symposium, 1999, pp. 15–20.

[16] KWAN-LIU MA, JAMIE S PAINTER, C.D. HANSEN, AND M.F. KROGH, *Parallel Volume Rendering Using Binary-Swap Compositing*, IEEE Computer Graphics Applications, 14 (1994), pp. 59–67.

[17] K.-L. MA, A. STOMPEL, J. BIELAK, O. GHATTAS, AND E. KIM, *Visualizing large-scale earthquake simulations*, in Proceedings of the Supercomputing 2003 Conference, November 2003.

[18] 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).

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

[20] *1997 petaflops algoritms workshop summary report*. `http://www.hpcc.gov/pubs/pal97.html`, 1997.

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

[22] 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 (to appear), October 2003.

[23] 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.

[24] T. TU AND D. R. O'HALLARON, *Balance refinement of massive linear octrees*, Tech. Report CMU-CS-04-129, Carnegie Mellon School of Computer Science, April 2004.

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

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

[27] T. TU, D. R. O'HALLARON, AND O. GHATTAS, *Scalable parallel octree meshing for terascale applications*, in SC2005, Seattle, WA, Nov. 2005.

[28] 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.

[29] 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.

[30] H. F. YU, K-L MA, AND J. WELLING, *A parallel visualization pipeline for terascale earthquake simulations*, in Proceedings of the Supercomputing 2004 Conference, November 2004.