11-2008

# Materialized community ground models for large-scale earthquake simulation

Steven W. Schlosser
*Intel Pittsburgh*

Michael P. Ryan
*Intel Pittsburgh*

Ricardo Taborda
*Carnegie Mellon University*

Julio Lopez
*Carnegie Mellon University*, jclopez@andrew.cmu.edu

David R. O'Hallaron
*Carnegie Mellon University*, droh@cs.cmu.edu

***See next page for additional authors***



Published In
.

# Materialized community ground models for large-scale earthquake simulation

Steven W. Schlosser†, Michael P. Ryan†, Ricardo Taborda⋆,
Julio López⋆, David R. O'Hallaron†⋆, Jacobo Bielak⋆
†*Intel Research Pittsburgh,* ⋆*Carnegie Mellon University*

## Abstract

Large-scale earthquake simulation requires source datasets which describe the highly heterogeneous physical characteristics of the earth in the region under simulation. Physical characteristic datasets are the first stage in a simulation pipeline which includes mesh generation, partitioning, solving, and visualization. In practice, the data is produced in an ad-hoc fashion for each set of experiments, which has several significant shortcomings including lower performance, decreased repeatability and comparability, and a longer *time to science*, an increasingly important metric.

As a solution to these problems, we propose a new approach for providing scientific data to ground motion simulations, in which ground model datasets are fully materialized into octress stored on disk, which can be more efficiently queried (by up to two orders of magnitude) than the underlying community velocity model programs. While octrees have long been used to store spatial datasets, they have not yet been used at the scale we propose. We further propose that these datasets can be provided as a service, either over the Internet or, more likely, in a datacenter or supercomputing center in which the simulations take place. Since constructing these octrees is itself a challenge, we present three data-parallel techniques for efficiently building them, which can significantly decrease the build time from days or weeks to hours using commodity clusters. This approach typifies a broader shift toward *science as a service* techniques in which scientific computation and storage services become more tightly intertwined.

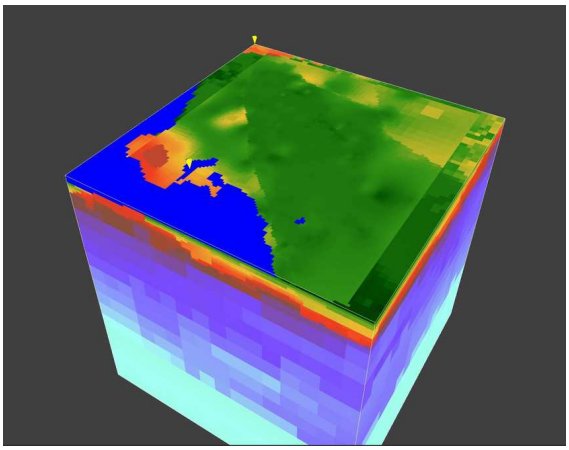


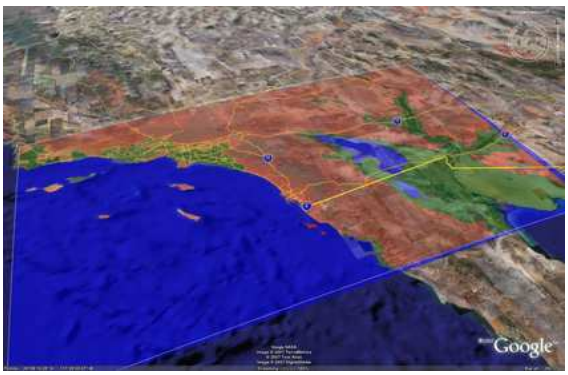
**Figure 1:** A visualization of a materialized etree representing the area surrounding Los Angeles. It was generated using the high-resolution region of the Harvard CVM (CVM-H) sampled at 10 m spacing. The softer soils at the top (shown in green) are quite heterogeneous and appear in greater detail than the deeper, harder soils (shown in light blue.)

## 1 Introduction

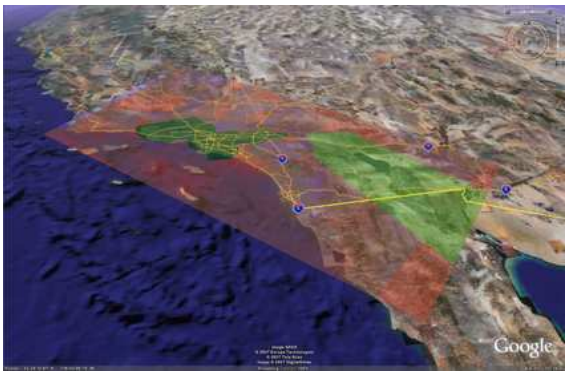
The main input to large-scale ground motion modeling are large source datasets which describe the physical characteristics of the earth. These datasets drive a simulation pipeline which includes mesh generation, partitioning, solving, and visualization [31]. In practice, these datasets are not fully materialized until simulation time. Rather, the data is produced in an ad-hoc fashion for each set of experiments, a process that, in our experience, takes on the order of days or weeks using existing techniques, and significantly increases the *time to science* for seismologists.

Ground characteristic data is generated by sampling generator programs (often called *community velocity models*, *ground models* or *CVMs*) such as those developed by the Southern California Earthquake Center (SCEC) [8, 15, 20, 21] and Harvard University [9, 28]. These programs take as input latitude, longitude, depth tuples and output a set of ground characteristics at those locations. Specifically, the key properties required are the density and the two velocities at which elastic waves travel in each material. Other properties, such as the attenuation characteristics of the material, are derived from the other three.

In addition to increasing the time to science, this ad-hoc approach to dataset generation has several significant shortcomings which can limit performance, as well make simulations less repeatable, comparable, and verifiable. First and foremost, repeatedly sampling the CVM programs is expensive, as we demonstrate in Section 2. We show that it is much more efficient (by a factor of $12 - 414\times$) to query a prebuilt data structure such as an etree, an indexed disk-resident octree [29]. Common operations on ground characteristic data such as model segmentation and mesh generation are well-suited to etrees rather than the common CVM programs [17], as they traverse paths through the




(a) Harvard (CVM-H)




(b) SCEC (CVM-S)

**Figure 2:** Each underlying CVM program we use covers a distinct region of southern California, but both cover metropolitan Los Angeles and San Diego. These images superimpose the surface octants of the etree covering each model's region onto a map using Google Earth. Harder soils are shown in red and softer soils are shown in green. CVM-S includes values for water (shown in blue), while CVM-H does not.

space. Second, a scientist must choose among the available velocity models (i.e., those from SCEC and Harvard). Each has different performance characteristics, input and output formats, and are not entirely consistent—while quite similar globally, the two models can produce different values at a particular location. The ability to operate on precomputed data offers scientists a significant ease-of-use advantage over having to build each dataset themselves. In our past experience, generating the ground model data for several large earthquake simulations took on the order of days or weeks. Lastly, experiments that use different ground model configurations are difficult to validate against each other. In the interest of verification and validation, there is good reason for scientists to be able to share input velocity models that use a common query interface.

To address these problems, we propose a new technique in which ground models are fully materialized (precomputed) over their respective regions of interest, and the resulting data efficiently stored in etrees [29] which can be shared among many scientists and used for many experiments. In order to fully materialize the data, the constituent CVM is sampled at high resolution, and then the resulting sample data is reduced by coalescing neighboring samples with homogeneous characteristics into increasingly larger octants. Coalescing is essential and results in significant data reduction, especially for deeper regions of the ground.

Figure 1 shows a visualization of a fully-materialized etree encompassing the region surrounding Los Angeles. It was materialized from the CVM-H program, and was sampled at a target resolution of under 10 m. By visual inspection, it is clear that the high-resolution regions near the surface maintain their detail, while the more homogeneous regions deeper underground are coalesced into larger octants. Figure 2 superimposes images of the CVM-H and CVM-S etrees onto a surface image using Google Earth.

Our goal is to sample the ground model at much higher resolution than has previously been attempted, which will lead to very large output etrees. To get an idea of scale, the Harvard CVM covers a region of Southern California that is 600 km $\times$ 400 km $\times$ 100 km, and we want to sample at under 10 m resolution. These parameters will lead to $24 \times 10^{12}$ raw samples, or on the order of petabytes of data. In reality, the majority of this raw sample data is homogeneous and is coalesced into larger octants, leading to significant compression.

We envision that these datasets can be provided as a service to scientists, with datasets parameterized at different resolutions, spatial regions, and aggregation (compression) parameters to fit specific needs. If a materialized dataset already exists that matches a scientist's needs, or if one can easily be derived from an existing file, then it can be used directly. Otherwise, a new dataset can be generated on the fly. Such a service would be most naturally provided within a datacenter or supercomputing center, co-located with or even resident on the machines used for simulation, in order to keep the data in place. The service can also be provided over the network.

Building such large materialized datasets is itself a significant computational challenge, so it is important to develop techniques to do so efficiently. Fortunately, the sampling process is data-parallel and if the samples are carefully partitioned, most of the coalescing phase can be locally carried out within a partition. We present three implementations for building these models on general computer cluster platforms: Map/Reduce, data-parallel with stack-based coalescing, and Map/Reduce with stack-based coalescing.

The Map/Reduce implementations use Hadoop [14], an open-source implementation of the Map/Reduce parallel programming model promoted by Google [10]. The first implementation uses the Map function to generate sam-

ples, gathers neighboring octants together by manipulating their locational codes, and then applies successive Reduce functions to coalesce homogeneous sibling tuples. While Map/Reduce is a useful and popular programming model, we found that a simplified data-parallel implementation using a parallel job scheduler (Maui/Torque [22], in our case) achieved much better performance as it avoids the shuffle and sort steps inherent in Map/Reduce programs. As well, using a stack-based coalescing strategy avoids the successive reduces. Lastly, in order to evaluate the overheads inherent in Hadoop, we built a Map/Reduce implementation of the stack-based coalescer. We evaluate the performance tradeoffs between the three implementations below.

The remainder of the paper is organized as follows. Section 2 describes how materialized ground models can be used to drive large-scale earthquake simulations. Section 3 presents background on etrees, locational codes, Map/Reduce, and community ground models. Section 4 describes three parallel implementations of our ground model builder, and Section 5 presents an evaluation examining properties of our constructed etrees and comparing the three construction techniques. Section 6 concludes.

# 2 Using community ground models in earthquake simulation

## 2.1 Earthquake simulation

Since the early 1970's, seismologists and engineers have understood the necessary methodology to represent the physical phenomena involved during the occurrence of earthquakes. Lysmer et al, introduced the use of the finite element method (FEM) in seismology [19]. Others adopted similar methodologies including the boundary element method (BEM), finite differences (FD) and spectral elements (SE). It was not, however, until the mid 1980's, with the rising aid of HPC, that numerical simulations took off, especially with the appearance of publicly-accessible supercomputing centers. The first fully three-dimensional simulation of an earthquake using parallel computers was performed in 1992 [11] using finite differences. It would be followed by others, e.g., [4, 13, 24]. Many of these simulations applied FD due to its structured nature and ease of implementation when used with a regular grid [5, 16]. Although FD is still the preferred method among some scientists, it is harder to scale up the problem size, specially when dealing with heterogeneous media. In particular, Tu et al, [31] built on the work of [5] and created *Hercules*, an octree-based parallel software system (developed by the Quake Group at Carnegie Mellon University), that implements a highly efficient algorithm for solving the wave field generated by a kinematic representation of the earthquake rupture in highly heterogeneous media. One of the special characteristics of Hercules is its fast generation of the mesh which is due in part to the fact that Hercules uses a data structure with high querying performance, an *etree*. This previously constructed etree is a discrete representation of any available community velocity model such as CVM-S or CVM-H described below.

## 2.2 Community velocity models

As SCEC's Community Modeling Environment group took on the construction of a multi-user framework for earthquake simulations, it rapidly became clear that a community velocity model accessible to all simulation groups was required. Therefore, SCEC adopted a seismic-velocity model [21] that, when queried at a certain latitude, longitude, and depth, returns values of S-wave ($V_s$) and P-wave ($V_p$) velocity, and density ($\rho$). This model is called CVM, and has since been released in four versions [15, 20], and is available at the SCEC web site [8]. At the same time, a group from Harvard University developed a similar velocity model which has recently gained increasing interest within the seismological community and is in the process of being adopted by SCEC's community as well. They are now referred to as CVM-S, for the former, and CVM-H, the latter. Although CVM-H has a much better querying performance than CVM-S, it covers a smaller area of Southern California than CVM-S does, and does not have enough information for low velocity soil profiles. Therefore, CVM-S continues to be the preferred velocity model source for large scale simulations such as ShakeOut—a magnitude 7.8 earthquake scenario that simulates a rupture in the southern part of the San Andreas fault in Southern California, affecting an area of 600 km $\times$ 300 km, that covers all of the major cities from San Diego to Santa Barbara, including the greater Los Angeles basin. This trend will very likely change in the next couple of years and as it does we are preparing to have an efficient interface between CVM-H and parallel simulation codes like Hercules. The methods for the efficient construction of etree representations of both CVM's is the objective of this paper.

## 2.3 The case for materialized community ground models

We believe that the community will transition away from building ad-hoc ground models based on CVM programs toward sharing pre-built, fully-materialized ground models stored in a standard data structure such as an etree. In fact, this transition is already underway with SCEC's adoption of an etree version of its velocity model CVM-S as the material model for the ShakeOut simulation.

Brocher et al. [6] constructed an etree velocity model of

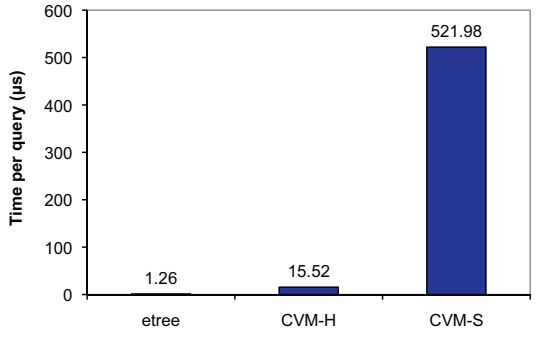



**Figure 3:** Average response times for queries by the path-based microbenchmark to the ground model programs and the pre-built etree. The etree is optimized for path-based queries, and so gives the best query performance.

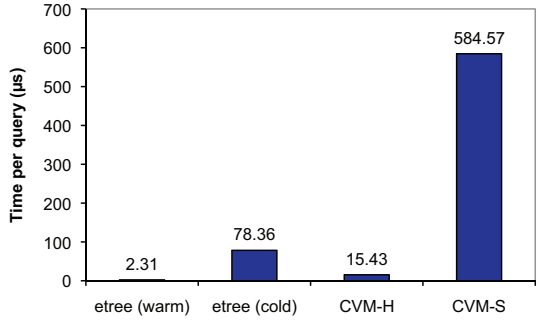



**Figure 4:** Average response times for queries by the random microbenchmark to the ground model programs and the pre-built etree. Random access to the etree performs well compared to sampling the CVM programs directly, especially when portions of the tree are pre-cached.

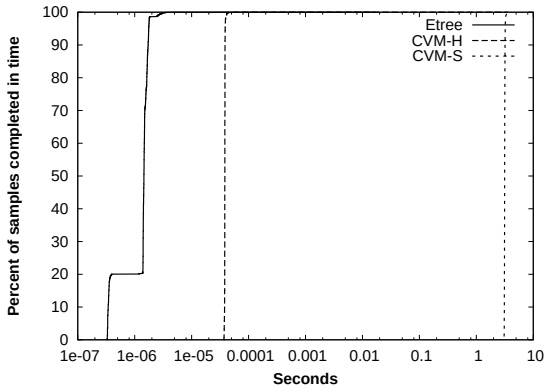



**Figure 5:** Distribution of query times during the path-based microbenchmark.

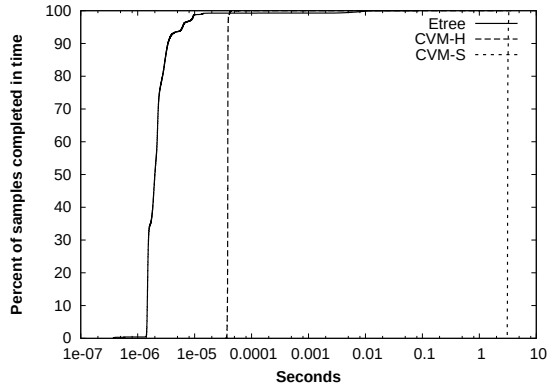



**Figure 6:** Distribution of query times during the random microbenchmark.

Northern California based on geologic data and detailed models of the San Francisco Bay Area. This model has been recently validated for moderate earthquake simulations [25] with very satisfactory results, and then used by [1, 2] for the reproduction and simulations of the 1989, Loma Prieta and 1906, San Francisco earthquakes.

We see three primary advantages to adopting materialized community ground models.

**Query cost.** We have observed that the cost of querying fully-materialized etrees is significantly lower than the cost of querying the CVM programs directly. In order to quantify this difference, we conducted the following experiments comparing the query costs of the CVM-S and CVM-H programs, as well as a pre-built etree. The etree covers a 75 km by 75 km by 75 km region around downtown Los Angeles, and is described in more detail in Section 5.1. We constructed a microbenchmark which generates 18 random coordinates from within the region, and samples 2 million coordinates either at random or along a Z-order path. (CVM-S is significantly slower than the etree or CVM-H, so we only report results for a single path of 2 million rather than the entire 18 paths.) The random mode exercises random lookup performance, and the Z-order mode replicates the query pattern of mesh generation.

Figure 3 shows the average time per query in the path-based benchmark for the CVM-S and CVM-H programs, as well as a pre-built etree. As expected, querying the CVM-S program is the slowest of the three, with queries taking, on average, 521.98 $\mu$s. The CVM-H program is $33\times$ faster, and the etree is another $12\times$ faster than CVM-H. As many of the workloads used in ground motion modeling tend to query ground models along paths (i.e., when generating meshes), the path-based microbenchmark does favor the etree, which is optimized for path accesses while the CVM programs are not. Figure 4 shows the average time per query for the random microbenchmark. In this case,

the etree performance is worse than with the path-based benchmark. In particular, random queries can incur many disk seeks when run with a cold disk cache. Warming up the disk cache improves performance dramatically.

The distribution of query times varies for the two ground model programs and for the etree, as shown in Figures 5 and 6. The CVM-H program has a startup cost of around 3.73 seconds for the first query, but then exhibits a nearly-constant query time. The etree library also exhibits a startup cost, as index nodes are fetched from disk. The performance of the CVM-S program is particularly low for random queries, as it can only process queries in batches. Therefore, each independent point query incurs the startup cost and is very expensive.

**Ease of use.** There is a clear need for a simplified workflow when using ground models, and using standardized, materialized models can address this. Research groups spend a considerable amount of time (days or weeks) configuring and building these datasets for their simulations in an ad-hoc fashion, time which would be better spent on research.

In some scenarios, it is not possible to easily directly integrate user-level ground model code into a parallel supercomputer infrastructure, especially one that uses a restricted operating system. In these cases, a scientist will need to pre-build datasets anyway.

We can say from our own experience that each of the ground model programs has its own quirks, which require an inordinate amount of time to deal with. Input and output formats differ, error conditions are reported differently, and the source code is difficult to modify. On the other hand, pre-built models using standard APIs and data formats are more transparent and usable.

**Standardization.** Cross-validation, comparison, and repeatability of results is a critical aspect of any scientific endeavor, and the use of shared, publicly-available model properties can provide a consistent basis for this. Providing canonical, shared source data for earthquake simulation fills a critical need in this regard.

# 3 Background

## 3.1 Octrees

*Octrees* are commonly-used data structures for representing spatial data in many domains [27]. Particularly in earth sciences, octrees are used to represent ground-velocity models, meshes and output wavefields [3, 30, 18]. Octrees offer a compromise between the simplicity of regular grids and the modeling power of fully unstructured meshes. They adapt to changes in the resolution of the underlying data, making them a good choice for representing data with drastic resolution variations across regions. Although octrees result in blocky representations of the data, in many cases these present no impediment for the simulations. Such is the case when interface jump conditions can be expressed in terms of equivalent loads.

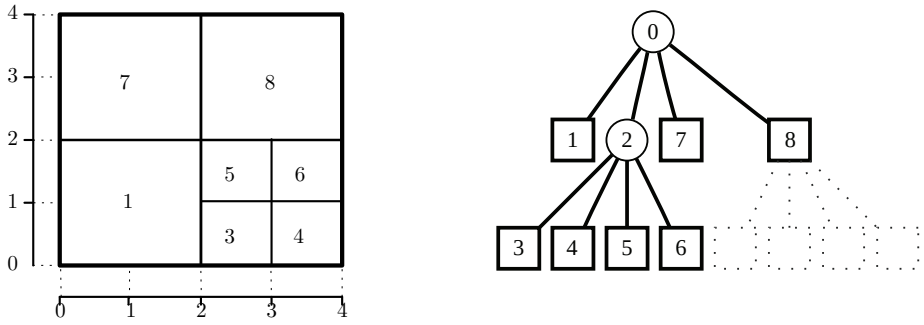
Octrees have two equivalent representations: *domain* representation and *tree* representation. To explain key properties of 3D octrees, we use 2D *quadtrees* for illustration purposes. These structures are built by recursively decomposing the space, thus they are hierarchical in nature. An octree divides a 3D region into 8 disjoint subregions, similarly a quadtree divides a 2D region into 4 smaller regions or quadrants, until it achieves a desired resolution. Figure 7 (a) shows a sample $4 \times 4$ rectangular domain (heavy line), one of its quadrants is further divided into 4 smaller *non-divisible* $1 \times 1$ quadrants.

In the equivalent tree representation shown in Figure 7 (b) each *node* in the tree corresponds to a quadrant in the domain, and their child nodes correspond to their subdivisions. For example, the root of the tree (node 0) represents the whole domain, and nodes 1 and 2 correspond the $2 \times 2$ quadrants shown in the domain representation. Nodes with descendants, e.g., node 2, are known as *interior nodes*. *Leaf nodes* have no descendants, e.g., nodes 3, 4, etc. Each node in the tree has an associated level $l$. The level of the *root node* is 0, for a node with level $l$, its children have level $l + 1$. The node level encodes the quadrant's size ($d \times d$), where $d = 2^{(\text{max-level}-l)}$ and *max-level* is the maximum level of any node in the tree, 2 in this example. The set of *ancestors* for a node $n$ is composed by its parent (immediate ancestor) and its parent's ancestors. The path from $n$ to the root is made up by all of the ancestors for $n$.

Notice that the leaf nodes do not need to be all at the same level. In a *complete* tree all leaves have the same level and is equivalent to a regularly-spaced grid of the same resolution. For datasets with large homogenous regions, siblings with the same properties can be *aggregated* into a single parent node. For example, Node 8 in Figure 7 (b) is an aggregation of its descendants (grayed out nodes).

## 3.2 Locational codes

The structure of a quadtree can be mapped to a linear address space by using *locational codes* [12]. This strategy consists of assigning addresses of the form $(i, j, l)$ to quadrants, such that each address uniquely identifies and implicitly encodes a quadrant's location and size. The $i, j$ components of the address are the grid domain coordinate components of the quadrant's lower-left corner, and $l$ is the quadrant's level in the tree representation. A child node at the lower-left quadrant of a larger enclosing quadrant has the same grid coordinates as its ancestors. The level in the quadrant address disambiguates this situation and also en-




(a) Domain representation

(b) Tree Representation

**Figure 7:** Sample Quadtree

codes the quadrant size. The domain in Figure 7 (a) has a $4 \times 4$ regular grid as a coordinate system. For example, the locational code for quadrant 3 is $(2, 0, 2)$, and its parent's (node 2) is $(2, 0, 1)$. The quadrant locational code [12], which is a variant of the *Morton code* [23], is obtained by interleaving the bits of the quadrant address fields except for the level $l$, and then appending the level to the interleaved bits. For example, in a $4 \times 4$ domain, two bits are required to represent each component and 2 bits are reserved for the level. Quadrant addresses are of the form $(x_0x_1, y_0y_1, l_0l_1)$, and the corresponding locational codes are of the form $(x_0y_0x_1y_1l_0l_1)$. The total ordering produced by the locational codes is known as Z-ordering or Morton curve [26].

## 3.3 Etree library

Dealing directly with locational codes increases application complexity. The etree library [29, 30] provides abstractions where applications operate on large persistent octrees in terms of octree addresses. Internally, the library transforms quadrant addresses to locational codes and stores octant data on disk. The etree API allows operations such as searching, inserting, deleting, appending and updating nodes in an octree.

To efficiently access octrees on disk, the library stores octants in Z-order and uses a *B-tree* [7] to index octants using locational codes as keys. Z-ordering has the interesting property that it corresponds to the pre-order traversal of the tree representation. Performing a pre-order traversal of the tree is equivalent to a sequential scan over the locational code key space, thus allowing for efficient utilization of storage streaming bandwidth [18]. Thus etrees are optimized for Z-order access.

## 3.4 Map/Reduce

Map/Reduce has recently been promoted by Google as a useful programming model and runtime system for data-intensive applications [10], and has generated a great deal of interest in the wider community. Google uses Map/Reduce in various large-scale applications, including building the reverse indices of the entire web. One goal of this work is to use Map/Reduce in new ways, beyond its roots in text processing toward scientific applications.

Basic Map/Reduce programs consist of two functions: $Map$ and $Reduce$. Input, output, and intermediate data in Map/Reduce programs take the form of key value pairs. The input to the $Map$ function is a set of tuples of type $< k_1, v_1 >$. Each invocation of the $Map$ function processes one of these tuples and produces zero or more *intermediate* tuples of type $< k_2, v_2 >$. Note that the types of the keys and the values that are output can be different than the types that were input.

The Map/Reduce runtime system will collect all of the tuples with equal intermediate keys into lists for processing by the $Reduce$ function. This is equivalent to the common *group-by* operation in databases.

The $Reduce$ function is invoked once for every intermediate key, and is given a list of all intermediate values having that key. Therefore, its input type is $< k_2, list(v_2) >$. Since the runtime system performs the group-by over all of the tuples generated in the $Map$ phase, the $Reduce$ function is guaranteed to see every intermediate value with that key. An optional *Combiner* function can apply reduction immediately after the $Map$ task, before the intermediate tuples are written to disk or sent over the network to the $Reduce$ tasks.

We use an open-source implementation of a Map/Reduce runtime called Hadoop [14]. Hadoop implements a distributed filesystem, HDFS, which spreads data blocks across the cluster nodes, and exposes the locations of those

blocks via a centralized metadata server. The Hadoop runtime implements much of the functionality described in [10], and is built in Java.

# 4 Implementation

## 4.1 Oversampling

Our approach is to oversample the underlying model at the finest resolution required, and then coalesce sibling octants if they are homogeneous within some threshold. We choose to oversample because the underlying CVM is a black box, and we can make few or no assumptions about the structure and the features of the ground that it models. As well, the CVM programs themselves are combinations of many underlying measurements, each of different accuracy and resolution. For example, the regions directly under the major cities such as Los Angeles are, for good reason, characterized at the highest resolution, while outlying areas are characterized much more coarsely. Areas around major faults and other features are also well-characterized. Resolution of the models also decreases deeper underground as soils become harder and more homogeneous, as well as being measured more coarsely.

Oversampling produces a large amount of data. Sampling the 600 km by 300 km by 100 km region covered by the SCEC model at 10 m resolution will produce $18 \times 10^{12}$ samples, leading to on the order of 1 PByte of raw data. In reality, the majority of these samples are homogeneous and will be coalesced together into larger octants over the course of running the program.

We can and do relax oversampling somewhat in order to reduce the raw data produced in the very hard, homogeneous soils found deep in the ground. For regions below 200 m we slowly begin to drop the sampling density based on the depth in order to save time. However, this is not a requirement of our system and we could very easily reconfigure these parameters and sample at higher resolution as necessary.

## 4.2 Thresholding

The CVM programs output three parameters of ground characteristics at each point: the velocity at which it propagates P-waves, $V_p$; the velocity at which it propagates S-waves, $V_s$; and its density, $\rho$. We coalesce a set of 8 sibling octants into a larger one subject to 3 different threshold values, one for each of the model characteristics ($V_s, V_p, \rho$). The octants are coalesced if *all* of the values for all the siblings are within a specified percentage thresholds of their mean values. Our baseline percentages are 2% for $V_s$, 5% for $V_p$, and 2% for $\rho$. Our choice of thresholding function was dictated by the characteristics of the simulations

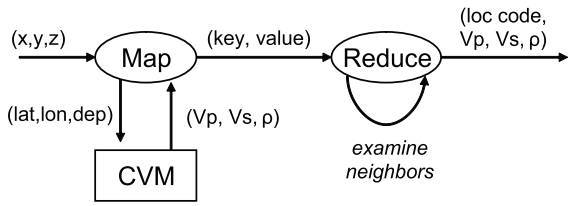



**Figure 8:** A sketch of the Map/Reduce algorithm.

which will use the resulting octree. The threshold values combined with the maximum simulated frequency can be used to compute a target error in the resulting simulation. In our case, we chose thresholds to meet a particular error. We evaluate the effects of varying thresholds on compression in Section 5.3.

## 4.3 Map/Reduce algorithm

The algorithm begins by partitioning the overall region into equal-sized cubic subregions. These regions consist of 262,144 samples (64 in each dimension). This is necessary in order to work with the existing CVM-S software. Each subregion is independent, and its samples are generated by a separate $Map$ task.

Figure 8 shows a sketch of the Map/Reduce algorithm. Samples are generated by the $Map$ function, which queries the underlying CVM program. The resulting tuples each represent a leaf node in the octree, and are each assigned an intermediate key based on the octant's locational code. The intermediate key is manipulated in order to gather together sibling tuples for the $Reduce$ function. Finally, the values of sibling tuples are examined by the $Reduce$ function for homogeneity. If they are homogeneous to within the given threshold, they are coalesced into a larger octant at the next higher level in the octree. In order to fully coalesce the data in the tree, the $Reduce$ function is applied again successively to the resulting data, which is accomplished by running the Map/Reduce program again with an identity mapper. Since the results of each iteration is stored into the distributed filesystem which spreads the data evenly across the compute nodes, the load remains balanced.

The input of $Map$ is the coordinate of the upper left corner of the subregion. The $Map$ function proceeds to generate coordinates for each of the sample points within the subregion, converting them to tuples of latitude, longitude, and depth (in meters) for input to the CVM program. It forks the chosen CVM program, pipes the latitude, longitude, depth tuples to its standard in file descriptor and collects the resulting ground characteristics. Coordinates to be sampled are generated in Z-order by incrementing locational codes.

The $Reduce$ phase examines sibling octants for homogene-

ity and coalesces them if their values are within the threshold. The keys of the intermediate tuples are based on the locational codes of the octants generated by the $Map$ function. By manipulating the keys, we can cause neighboring tuples to be processed by a single invocation of $Reduce$. For a given octant, clear sets of three remaining low-order bits based on the level. For octants at the lowest level, the lowest three bits are cleared, for octants in the next level up, six bits are cleared, and so on. We operate only on the bits in the locational code corresponding to the coordinates. Those bits that represent the level are not modified. In this way, sets of eight sibling octants will have equal intermediate keys, and will be processed in a single invocation of $Reduce$.

The Map/Reduce model also allows the programmer to optionally specify a $Combiner$ function, which is a reduction that is applied to the intermediate tuples immediately after the $Map$, before data is written to disk. In our case, we use our $Reduce$ as the $Combiner$, which has the effect of applying one round of coalescing right after the $Map$. Since many octants are homogeneous at the highest resolution, this saves a great deal of data transfer, leading to increased performance.

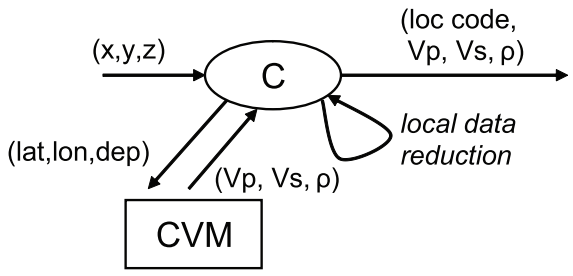



**Figure 9:** A sketch of the data-parallel stack-based implementation. The core is a simple C program which forks the external CVM.

```
sn = 0
while samples available from CVM do
  push(sample from CVM)
  sn++
  d = 1
  while (sn mod 8^d ≡ 0) and (last 8 samples are within threshold) do
    < samples >= pop(8)
    push(coalesce(< samples >))
    d++
  end while
end while
```

**Figure 10:** Pseudocode of the stack-based coalescing algorithm.

## 4.4 Data-parallel stack algorithm

By carefully choosing the partitions, it is possible to locally decide whether to coalesce a given set of octants within a partition. This approach does not require the complete data shuffle used in the Map/Reduce case. In fact, we found that the shuffle and data copies incurred by the Map/Reduce model were quite expensive, as we will see in Section 5. Thus, we were motivated to build a second version of the system which, while still running in parallel on our cluster, performs all of the data generation and reduction for a single subregion in one process.

As with the Map/Reduce implementation described above, we start by partitioning the overall space to be sampled into fixed-sized subregions. We use an octree to determine the size and location of the subregions. Thus, each subregion is aligned with an octree partition at a given level and contains only octants in the branch of the tree corresponding to that partition. This allows us to treat each subregion independently as all the coalescing in the corresponding octree branch can be performed locally.

All of the samples for a single subregion are generated by a single invocation of a simple C program which forks the chosen CVM program, feeds it with the appropriate latitude, longitude, depth tuples corresponding to the subregion, and collects the output values for reduction. This program is run on the cluster using a simple open-source job scheduling system, Maui/Torque [22].

In contrast to the Map/Reduce algorithm used above, in this version we use a stack-based coalescing algorithm, shown in Figure 10. Each group of eight sibling tuples may be considered independently immediately after it is created. By exploiting the fact that the CVM program returns coordinates in the same order they are given, it is possible to produce sample values that are already in Z-order. This allows us to process the output of the CVM using a stack and avoid having to store the raw output of the program at any time.

Consider pushing each new sample value on the stack. For each group of eight samples, a reduction may be attempted on them. If this succeeds, replace the last eight samples on the stack with one sample that is representative of them all, i.e., by using their mean values. With this simple procedure, it is possible to perform the same computation done by one round of Map and Reduce in the previous description, saving a great deal of time spent dealing with intermediate data.

Even more effective is the ability to apply successive coalescing steps on the data in place, which eliminates the need for successive reduce phases required in the Map/Reduce algorithm. As octants are coalesced at one level, the resulting larger octant is pushed back onto the stack for comparison in the next round.

### 4.5 Stack-based Map/Reduce algorithm

The data-parallel stack-based algorithm described in the previous section handily outperforms the Map/Reduce implementation by operating on the sample data entirely in place, thus avoiding intermediate data movement and allowing successive rounds of coalescing. We found that the stack-based algorithm was nearly an order of magnitude faster than the Map/Reduce version described above, as it avoids the costs of the distributed group-by.

As another point of comparison, we also implemented the stack-based algorithm as a Map-only Map/Reduce program. In this version, the $Map$ function is a simple port of the C-based program described above into Java. The Hadoop Map/Reduce runtime is implemented in Java, so we expect to see some performance overhead compared to the C version. As well, the resulting data tuples are output using the standard Map/Reduce *collect* method, rather than by writing directly to an output file, which causes tuples to be initially gathered in memory buffers and then eventually written out to files in the Hadoop distributed filesystem.

As with the C-based stack implementation, samples are generated in Z-order using the external CVM program and are pushed onto a stack. The algorithm examines each 8 neighboring octants on the stack and coalesces them if they are within the threshold. If the siblings are coalesced, the resulting larger octants are pushed back onto the stack to be examined at the next higher level.

Comparing the performance of the data-parallel stack-based system (built in C) to the Map/Reduce stack-based system provides a more apples-to-apples evaluation of the costs of using Hadoop.

## 5 Evaluation

### 5.1 Experimental setup

All of our experiments are run on a cluster of Dell PowerEdge 1955 blade servers. Each server contains two quad-core Intel Xeon E5345 CPUs, clocked at 2.33 GHz, 8 GB of memory and two 146 GB Seagate Savvio 10K.2 disks. While the entire cluster consists of 50 servers, it is shared among many users running various applications. Therefore, we isolated ten servers to use for our testbed, meaning that nearly all of our performance measurements were gathered using 80 cores.

Each server runs 64 bit Ubuntu Linux server edition version 7.04. We use Hadoop version 0.15.0 [14], running in Sun Java Runtime 1.6.0. Each node in the cluster serves both as a Hadoop task tracker (i.e., compute node), and as a data node for the Hadoop distributed filesystem, HDFS. Hadoop has too many configuration parameters to describe here, but the most relevant for the purposes of this evaluation is the number of concurrent $Map$ and $Reduce$ tasks that are allowed to run on each node. We configured our cluster to run eight concurrent tasks per server. Each Map/Reduce program that is run is partitioned into $M$ map tasks and $R$ reduce tasks. Input and output data for the Map/Reduce programs is stored in HDFS, while input and output data for the data-parallel stack-based implementation is stored directly on the local disks.

While we are able to use both CVM programs supported by SCEC, CVM-S and CVM-H, we found that CVM-S is at least an order of magnitude slower than CVM-H. Therefore, for the purposes of this evaluation, we only report performance results using CVM-H.

For the evaluation, we chose a 75 km by 75 km by 75 km region of Southern California that is covered by both the CVM-S and CVM-H models. The region covers most of metropolitan Los Angeles, and encompasses the highest-resolution and most-heterogeneous portions of both models. We use an octree with thirteen levels to sample the region, leading to a best possible resolution of 9.15 meters ($75000/(2^{13})$). The upper-left corner of the region is at latitude and longitude N34.190280 W-118.550948, and the lower-right corner is at N33.516643 W-117.739757.

### 5.2 Sampling and compression

Table 1 summarizes the results of building octrees for both the CVM-S and CVM-H models, for the two regions of interest that we studied. The first region is the 75 km by 75 km by 75 km region described above that we used for performance benchmarking. The second is the entire region supported by each model. When building the etrees, we set the depth of the tree such that the best possible resolution was under 10 meters.

For the first region, we generate sample data for $5.74 \times 10^9$ octants. The data from the two underlying models coalesce to different degrees ($11 - 15\times$) because the measurements and interpolation functions that they use are different. The etrees built from the entire regions are interesting as well, in that they coalesce by almost an extra order of magnitude compared to the etrees generated from the smaller regions. This is because the majority of the uncoalesced samples are from the high-resolution center of the region around Los Angeles, which is included in both cases.

### 5.3 Error rates

Using a threshold to compress the sampled data into an octree introduces errors that are dependent on the underlying CVM program used and the thresholding parameters. We characterize the error by measuring the signal-to-noise ratio (SNR) and peak signal-to-noise ratio (PSNR) between

| Model | Region | Octree depth | Sampled octants | Coalesced octants | Compression |
|---|---|---|---|---|---|
| CVM-S | $75 \times 75 \times 75$ | 13 | $5.74 \times 10^9$ | $5.26 \times 10^8$ | $11\times$ |
| CVM-H | $75 \times 75 \times 75$ | 13 | $5.74 \times 10^9$ | $3.79 \times 10^8$ | $15\times$ |
| CVM-S | $600 \times 300 \times 100$ | 16 | $1.84 \times 10^{11}$ | $2.57 \times 10^9$ | $72\times$ |
| CVM-H | $600 \times 400 \times 100$ | 16 | $2.45 \times 10^{11}$ | $3.54 \times 10^9$ | $69\times$ |

**Table 1:** Comparison of the two CVM models and their results as etrees. We built materialized etrees of each model over two regions of interest: the 75 km by 75 km by 75 km region we used for generating results (described in Section 5.1), and the entire region that each model covers. In order to reach a target resolution of under 10 m, the octrees were 13 or 16 levels deep, depending on the side of the region. We report the total number of sampled octants for each case, the final number of octants in the fully-coalesced etree, and the compression factor.

the uncompressed data and the coalesced samples stored in the resulting octree. For this, we use the uncompressed model as the reference signal ($S = \{s[0], s[1], \cdots, s[n-1]\}$) and compute the error for each octant as the difference between the value for the octant in the uncompressed and compressed models ($x[i] - s[i]$). Then, we compute SNR as the ratio of the mean square signal values to the mean square error (MSE). Similarly, PSNR is computed as the square of the maximum signal value over the MSE.

| Threshold factor | SNR(dB) | | | PSNR(dB) | | |
|---|---|---|---|---|---|---|
| | Vp | Vs | Rho | Vp | Vs | Rho |
| 1/4 | 70 | 75 | 76 | 87 | 85 | 86 |
| 1/2 | 62 | 71 | 74 | 78 | 81 | 84 |
| 1 | 55 | 65 | 69 | 72 | 76 | 80 |
| 2 | 49 | 60 | 64 | 65 | 70 | 74 |
| 4 | 44 | 55 | 59 | 61 | 66 | 69 |

**Table 2:** SNR and PSNR for compressed ground models (CVM-H)

| Threshold factor | SNR(dB) | | | PSNR(dB) | | |
|---|---|---|---|---|---|---|
| | Vp | Vs | Rho | Vp | Vs | Rho |
| 1/4 | 55 | 53 | 67 | 62 | 60 | 70 |
| 1/2 | 49 | 47 | 61 | 56 | 54 | 63 |
| 1 | 43 | 41 | 55 | 50 | 49 | 58 |
| 2 | 37 | 36 | 49 | 44 | 43 | 52 |
| 4 | 32 | 30 | 44 | 39 | 38 | 46 |

**Table 3:** SNR and PSNR for compressed ground models (CVM-S)

Tables 2 and 3 contain the SNR and PSNR values in decibels (dB) for compressed models generated from the CVM-H and CVM-S programs respectively. We built datasets with different thresholding parameters, which were generated by applying a scaling factor to the thresholds originally specified by our local ground modeling experts. The first table column shows the scaling factor for these parameters. For example, the row with a scaling factor of 1x shows the results for the original thresholding parameters (2%, 5%, 2%). A scaling factor of $1/2$ results in thresholding parameters of (1%, 2.5%, 1%), and so on. The SNRs for each of

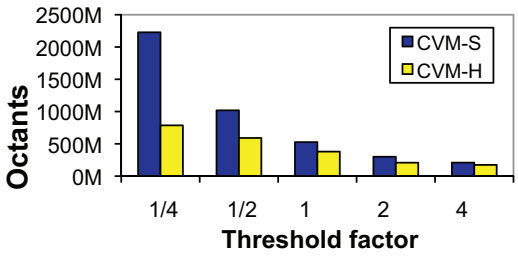



**Figure 11:** As the coalescing threshold becomes less stringent (i.e., the threshold factor increases), the number of octants in the compressed etree decreases. There were 5,737,807,872 octants in the uncompressed etree.

the data values in the model ($V_s$, $V_p$, $rho$) are contained in columns 2-4. The last 3 columns show the corresponding PSNR.

Figure 11 shows the effect of changing thresholds on the number of octants in the compressed etree. CVM-H etrees compress more readily than CVM-S etrees because CVM-S includes more detail in the central Los Angeles basin than does CVM-H.

As expected, the lower threshold values produce higher fidelity models (higher SNR and PSNR values). The results indicate that the generated models have good overall quality. As a point of reference, lossy compressed images with SNR values of 25 dB are considered to be of good quality. The model compressed with the original thresholding parameters has SNR values of (55, 65, 69) dB, indicating that they have more than sufficient quality for simulation purposes.

Models of this quality had not been built before as fine spatial sampling was not performed. Including the error rates in the metadata of materialized models enables scientists to determine the quality of the model with respect to its source model program.

## 5.4 End-to-end measurements

Figure 12 compares the runtime of the two stack-based implementations. Each result is the average of three

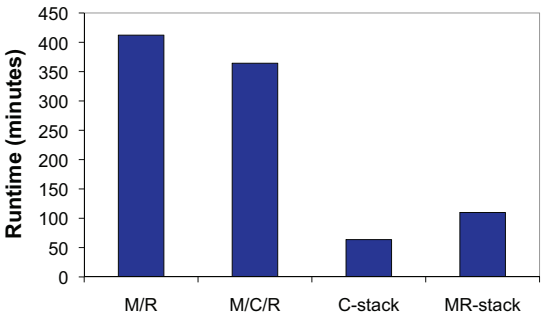



**Figure 12:** Runtime comparison of the Map/Reduce, data-parallel, and Map/Reduce-based stack implementations.

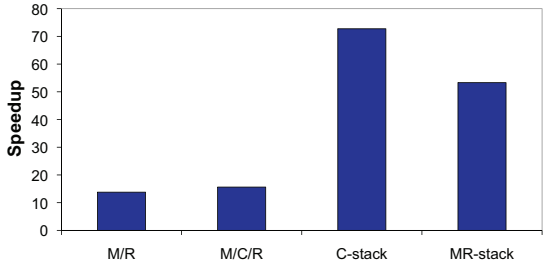



**Figure 13:** Parallel speedup of the Map/Reduce-based and data-parallel stack implementations on 80 CPUs.

runs. The Map/Reduce implementation, both with and without the optional Combiner function, is by far the slowest version, with the overall runtime being 412 and 364 minutes, respectively. The poor performance of the Map/Reduce implementation was surprising to us, especially compared to the much better performance of the data-parallel stack implementation, which completes in 64 minutes. The overheads of the distributed group-by and sorting are significant and, in our view, rule out the use of a pure Map/Reduce algorithm for this application. The performance of Map/Reduce-stack is closer to that of data-parallel stack, completing in 109 minutes.

Both stack-based implementations avoid three aspects of the Map/Reduce implementation, all of which prove to be very expensive. First, all of the sample data is processed and coalesced in place rather than shuffled between $Map$ and $Reduce$ tasks. Operating on data in place avoids extra writes to disk of intermediate tuples, as well as the time to transfer data across the network. Second, the $Reduce$ phase of the Map/Reduce implementation operates on eight octants at a time, resulting in several hundred million calls to $Reduce$. Lastly, the stack-based implementations are able to entirely coalesce the samples that they generate at all levels of the octree in a single invocation. The Map/Reduce implementation requires multiple passes of the $Reduce$ function in order to fully coalesce the octree.

Comparing the runtime of the data-parallel stack and the Map/Reduce-stack implementations is interesting, since they present a more apples-to-apples evaluation of the overheads of Hadoop. Given that they vary by roughly 50%, we conclude that the overheads imposed by Java and the use of HDFS are insignificant next to the overheads of intermediate data storage, sorting, and shuffling, which are required by the pure Map/Reduce implementation. This result surprised us, and suggest that a Map/Reduce application built using Hadoop must make significant use of the shuffling and sorting functionality in order to justify their overhead.

### 5.5 Parallel efficiency

In order to calculate the parallel speedup achieved by each implementation, we examined the ratio of the total time spent executing the external CVM program to the wall-clock time of the overall program. By doing so, we compare the time that it would have taken to generate all of the samples serially to the time that it took to generate and coalesce them in parallel.

Figure 13 shows the results. Generating all of the samples using the data-parallel stack implementation (C-stack) took, on average, 277,667 seconds of CPU time, which was accomplished in 3817 seconds of wall-clock time using 80 cores, achieving a parallel speedup of $72.7\times$. The Map/Reduce implementations do not scale nearly as well due to the extra data shuffling overheads, achieving a parallel speedup of $13.8\times$-$15.6\times$. The Map/Reduce-stack implementation fared better, achieving a $53.3\times$ speedup on 80 CPUs. The difference between the Map/Reduce-stack and the data-parallel stack implementations stem from the overhead of gathering output tuples in Hadoop and writing to HDFS. The data-parallel stack implementation need only write its output to a local file.

## 6 Conclusion

Materialized ground models have the potential to improve the science of earthquake simulation by providing better query performance, ease-of-use, and standardization. In this paper we have made the case for their use by demonstrating superior query performance and presenting three implementations that make materializing them efficient using parallel compute clusters.

Examining the three implementations of the ground model generator was, in and of itself, an interesting exercise. Our approach of using Map/Reduce was initially promising, as the problem had many of the characteristics for which Map/Reduce is suited. Samples are generated in the first phase and then are reduced, octants need to be grouped to-

gether in order to be coalesced, and repeated reduce passes can coalesce successive levels in the octree. However, after having built the implementation, it became clear that much of the machinery of Map/Reduce is simply unnecessary in this case. Tuples are naturally grouped using increasing locational codes, and reduction is entirely local to each subtask, which both obviate the need for a global group-by.

# References


[1] B. T. Aagaard, T. M. Brocher, D. Dolenc, D. Dreger, R. W. Graves, S. Harmsen, S. Hartzell, S. Larsen, K. McCandless, S. Nilsson, N. A. Petersson, A. Rodgers, B. Sjogreen, and M. L. Zoback. Ground-Motion Modeling of the 1906 San Francisco Earthquake, Part II: Ground-Motion Estimates for the 1906 Earthquake and Scenario Events. *BULLETIN OF THE SEISMOLOGICAL SOCIETY OF AMERICA*, 98(2):1012–1046, 2008. http://earthquake.usgs.gov/regional/nca/1906/simulations/.

[2] B. T. Aagaard, T. M. Brocher, D. Dolenc, D. Dreger, R. W. Graves, S. Harmsen, S. Hartzell, S. Larsen, and M. L. Zoback. Ground-Motion Modeling of the 1906 San Francisco Earthquake, Part I: Validation Using the 1989 Loma Prieta Earthquake. *BULLETIN OF THE SEISMOLOGICAL SOCIETY OF AMERICA*, 98(2):989–1011, 2008.

[3] V. Akcelik, J. Bielak, G. Biros, I. Ipanomeritakis, A. Fernandez, O. Ghattas, E. Kim, J. López, D. O'Hallaron, T. Tu, and J. Urbanic. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of Supercomputing SC'2003*, Phoenix AZ, USA, Nov 2003. ACM, IEEE. Available at `www.cs.cmu.edu/~ejk/sc2003.pdf`.

[4] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk, and J. Xu. Earthquake ground motion modeling on parallel computers. In *Proceedings of the 1996 ACM/IEEE Conference on High Performance Networking and Computing*, page 13, Pittsburgh, Pennsylvania, United States, 1996. IEEE Computer Society.

[5] 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(2):99–112, 2005.

[6] T. M. Brocher, B. Aagaard, R. W. Simpson, and R. C. Jachens. The usgs 3d seismic velocity model for northern california. *Eos Trans. AGU, Fall Meeting*, 87(52):Abstract S51B–1266, 2006. http://www.sf06simulation.org/geology/velocitymodel.

[7] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 2(11):121–138, 1979.

[8] 3d velocity model for Southern California. http://-www.data.scec.org/3Dvelocity/.

[9] Cvm-H. http://epicenter.usc.edu/cmeportal/cmodels.html.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating Systems Principles*, 2004.

[11] A. Frankel and J. Vidale. A three-dimensional simulation of seismic waves in the Santa Clara Valley, California, from a Loma Prieta aftershock. *Bulletin of the Seismological Society of America*, 82(5):2045–2074, 1992.

[12] I. Gargantini. Linear octree for fast processing of three-dimensional objects. *Computer Graphics and Image Processing*, 20(4):365–374, 1982.

[13] R. W. Graves. Simulating seismic wave propagation in 3d elastic media using staggered-grid finite differences. *Bulletin of the Seismological Society of America*, 86(4):1091–1106, 1996.

[14] The Hadoop Project. http://hadoop.apache.org/.

[15] M. D. Kohler, H. Magistrale, and R. W. Clayton. Mantle heterogeneities and the SCEC reference three-dimensional seismic velocity model version 3. *Bulletin of the Seismological Society of America*, 93(2):757–774, 2003.

[16] D. Komatitsch, J.-P. Vilotte, R. Vai, J. M. Castillo-Covarrubias, and F. J. Sánchez-Sesma. The spectral element method for elastic wave equations—application to 2-d and 3-d seismic problems. *International Journal for Numerical Methods in Engineering*, 45(9):1139–1164, 1999.

[17] J. Lopez. *Methods for Querying Compressed Wavefields*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, May 2007.

[18] J. López, D. O'Hallaron, and T. Tu. Big wins with small application-aware caches. In *Proceedings of Supercomputing 2004 (SC2004)*, Pittsburgh, PA, Nov 2004. IEEE.

[19] J. Lysmer and L. A. Drake. A finite element method for seismology. In B. Alder, S. Fernbach, and B. Bolt, editors, *Methods in Computational Physics*, volume 11, chapter 6. Academic Press, New York, 1972.

[20] H. Magistrale, S. Day, R. W. Clayton, and R. Graves. The SCEC southern California reference three-dimensional seismic velocity model version 2. *Bulletin of the Seismological Society of America*, 90(6B):S65–S76, 2000.

[21] H. Magistrale, K. McLaughlin, and S. Day. A geology-based 3D velocity model of the Los Angeles basin sediments. *Bulletin of the Seismological Society of America*, 86(4):1161–1166, 1996.

[22] Maui and Torque, Cluster Resources, Inc. http://-www.clusterresources.com/.

[23] G. M. Morton. A computer oriented geodetic database and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[24] K. B. Olsen, R. J. Archuleta, and J. R. Matarese. Three-dimensional simulation of a magnitude 7.75 earthquake on the san andreas fault. *Science*, 270(5242):1628–1632, December 1995.

[25] A. Rodgers, N. A. Petersson, S. Nilsson, B. Sjogreen, and K. McCandless. Broadband Waveform Modeling of Moderate Earthquakes in the San Francisco Bay Area and Preliminary Assessment of the USGS 3D Seismic Velocity Model. *BULLETIN OF THE SEISMOLOGICAL SOCIETY OF AMERICA*, 98(2):969–988, 2008.

[26] H. Sagan. *Space Filling Curves*. Springer, 1994.

[27] H. Samet. *Applications of Spatial Data Structures: Computer Graphics Image Processing and GIS*. Addison-Wesley, 1989.

[28] M. Süss and J. Shaw. Seismic velocity structure derived from sonic logs and industry reflection data in the Los Angeles Basin, California. *Journal of Geophysical Research*, 108(B3):2170, 2003.

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

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

[31] T. Tu, H. Yu, L. Ramírez-Guzmán, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron. From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing. In *Proceedings of the 2006 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, page 15, Tampa, Florida, November 2006. IEEE Computer Society.