A Parallel Time-Domain Wave Simulator Based on Rectangular Decomposition for Distributed Memory Architectures

Nicolas Morales[a,*], Ravish Mehra[a], Dinesh Manocha[a]

[a] *UNC Chapel Hill*

Abstract

We present a parallel time-domain simulator to solve the acoustic wave equation for large acoustic spaces on a distributed memory architecture. Our formulation is based on the adaptive rectangular decomposition (ARD) algorithm, which performs acoustic wave propagation in three dimensions for homogeneous media. We propose an efficient parallelization of the different stages of the ARD pipeline; using a novel load balancing scheme and overlapping communication with computation, we achieve scalable performance on distributed memory architectures. Our solver can handle the full frequency range of human hearing (20Hz-20kHz) in scenes with volumes of thousands of cubic meters. We highlight the performance of our parallel simulator on a CPU cluster with up to a thousand cores and terabytes of memory. To the best of our knowledge, this is the fastest time-domain simulator for acoustic wave propagation in large, complex 3D scenes such as outdoor or architectural environments.

Keywords: Time-domain wave acoustics, parallel algorithms, room acoustics

1. Introduction

The computational modeling and simulation of acoustic spaces is fundamental to many scientific and engineering applications [10]. The demands vary widely, from interactive simulation in computer games and virtual reality to highly accurate computations for offline applications, such as architectural design and engineering. Acoustic spaces may correspond to indoor spaces with complex geometric representations (such as multi-room environments, automobiles, or aircraft cabins), or to outdoor spaces corresponding to urban areas and open landscapes.

Computational acoustics has been an area of active research for almost half a century and is related to other fields (such as seismology, geophysics, meteorology, etc.) that deal with similar wave propagation through different mediums. Small variations in air pressure (the source of sound) are governed by the three-dimensional *wave equation*, a second-order linear partial differential equation. The computational complexity of solving this wave equation increases as at least the cube of frequency, and is a linear function of the volume of the scene. Given the auditory range of humans (20Hz - 20kHz), performing wave-based acoustic simulation for acoustic spaces corresponding to a concert hall or a cathedral (e.g. volume of $10,000$ - $15,000$ m^3) for the maximum simulation frequency of 20kHz requires tens of *Exaflops* of computational power and tens of terabytes of memory. In fact, wave-based numeric simulation of the high frequency wave equation is considered one of the more challenging problems in scientific computation [13].

Current acoustic solvers are based on either geometric or wave-based techniques. Geometric methods, which are based on image source methods or ray-tracing and its variants [2, 14, 28], do not accurately model certain low-frequency features of acoustic propagation, including diffraction and interference effects. The wave-based techniques, on the other hand, directly solve governing differential or integral equations which inherently account for wave behavior. Some of the widely-used techniques are the finite-difference time domain method (FDTD) [25, 6], finite-element method (FEM) [31], equivalent source method (ESM) [18], or boundary-element method (BEM) [9, 8]. However, these solvers are mostly limited to low-frequency (less than 2kHz) acoustic wave propagation for larger architectural or outdoor scenes, as higher-frequency simulation on these kinds of scenes requires extremely high computational power and terabytes of memory. Hybrid techniques also exist which take advantage of the strengths of both geometric and wave-based propagation [35].

Recently developed low-dispersion wave methods for solving the wave equation reduce the computational overhead and memory requirements. These include Kowalczyk and van Walstijn's interpolated wideband scheme [17] in addition to modifications to waveguide mesh approaches [27]. One of these methods is the *adaptive rectangular decomposition (ARD)* technique [22, 19], which performs three-dimensional acoustic wave propagation for homogeneous media (implying a spatially-invariant speed of sound). ARD is a domain decomposition technique that partitions a scene in rectangular (cuboidal in 3D) regions, computes local pressure fields in each partition, and combines them to compute the global pressure field using appropriate interface operators. Previously, ARD has been used to perform acoustic simulations on small indoor and outdoor scenes for a maximum frequency of 1 kHz using only a few gigabytes of memory on a high-end desktop machine. However, performing

*Corresponding author

Email address: nmorales@cs.unc.edu (Nicolas Morales)

accurate acoustic simulation for large acoustic spaces up to the full auditory range of human hearing still requires terabytes of memory. Therefore, there is a need to develop efficient parallel algorithms, scalable on distributed memory clusters, to perform these large-scale acoustic simulations.

Main Results: We present a novel, distributed time-domain simulator that performs accurate acoustic simulation in large environments. Our approach is based on the ARD solver and is designed for CPU clusters. The two major components of our approach are an efficient load-balanced domain decomposition algorithm and an asynchronous technique for overlapping communication with computation for ARD.

The primary use of our parallel simulator is to perform acoustic propagation in large, complex indoor and outdoor environments for high frequencies. We gain near-linear scalability with our load-balancing scheme and our asynchronous communication technique. As a result, when the scale of the computational domain increases, either through increased simulation frequency or higher volume, we can add and efficiently use more computational resources. This efficiency is compounded by the low-dispersion nature of the underlying ARD solver.

Our current implementation shows scalability up to 1024 cores of a CPU cluster with 4 terabytes of memory. Using these resources, we can efficiently compute sound fields on large architectural and outdoor scenes up to 4kHz. To the best of our knowledge, this is the first practical parallel wave-simulator that can perform accurate acoustic simulation on large architectural or outdoor scenes for this range of frequencies.

Organization: The rest of the paper is organized as follows. We briefly survey prior work on acoustic simulation and wave solvers in Section 2. We give an overview of ARD and highlight its benefits in Section 3. We present our parallel algorithm in Section 4 and describe its implementation in Section 5. We analyze the performance in Section 6.

2. Prior Work

There has been considerable work on acoustic simulation, parallel wave solvers, and domain-decomposition techniques. In this section, we give a brief overview of prior work in these areas.

2.1. Parallel Wave-based Solvers

There is considerable literature on developing parallel scientific solvers for wave equations as applied to seismic, electromagnetic, and acoustic simulation. These include general, parallel algorithms based on FDTD on large clusters [15, 36, 32] or GPUs [23, 26, 29, 30, 34]. Other techniques are based on parallel finite-element meshing [11, 5]. Many specialized parallel algorithms have been developed for wave equations for specific applications. These include PetClaw [1], a distributed solver for time-dependent nonlinear wave propagation, massively parallel multifrontal solvers for time-harmonic elastic waves in 3D anisotropic media [33], 3D finite-difference frequency-domain methods for 3D visco-acoustic wave propagation on distributed platforms [21], discontinuous Galerkin solvers designed on unstructured tetrahedral meshes for three-dimensional heterogeneous electromagnetic and aeroacoustic wave propagation [4], large-scale simulations of elastic wave propagation in heterogeneous media [3], parallel FDTD methods for computational electromagnetics [37], etc.

2.2. Domain Decomposition

Domain-decomposition methods are widely used to solve large boundary value problems by iteratively solving subproblems defined on smaller subdomains. These techniques subdivide the problem domain into smaller partitions, solve the problem inside each of the smaller partitions to generate local solutions, and combine all the local solutions to compute the global solution. Such techniques are also used to design numeric solvers that can be easily parallelized on coarse-grain parallel architectures. At a broad level, these can be classified into *overlapping subdomain* and *nonoverlapping subdomain* methods [7]. Many parallel multi-grid methods have also been applied to computational acoustics, including solvers for the three-dimensional wave equation in underwater acoustics [24].

3. Adaptive Rectangular Decomposition

In this section, we provide an overview of the adaptive rectangular decomposition (ARD) technique.

3.1. Time-Domain Acoustic Simulation

ARD is a numerical simulation technique that performs sound propagation by solving the acoustic wave equation in the time domain [22]:

$$\frac{\partial^2}{\partial t^2} p(X,t) - c^2 \nabla^2 p(X,t) = f(X,t), \tag{1}$$

where $X = (x,y,z)$ is the point in the 3D domain, t is time, $p(X,t)$ is the sound pressure (which varies with time and space), c is the speed of sound, and $f(X,t)$ is a forcing term corresponding to the boundary conditions and sound sources in the environment. In this paper, we limit ourselves to homogeneous domains, where c is treated as a constant throughout the media.

The ARD simulator belongs to a class of techniques, referred to as *domain-decomposition techniques*. In this regard, ARD uses a key property of the wave equation: the acoustic wave equation has known analytical solutions for rectangular (cuboidal in 3D) domains for homogeneous media. The underlying solver exploits this property decomposing the domain in rectangular (cuboidal) partitions, computing local solutions inside the partitions, and then combining the local solutions with interface operators to find the global pressure field over the entire domain.

3.2. Pressure Field Computation

Consider a cuboidal domain in three dimensions of size (l_x, l_y, l_z) with perfectly reflecting walls. The acoustic wave equation on this domain has an analytical solution represented as:

$$p(x,y,z,t) = \sum_{i=(i_x,i_y,i_z)} m_i(t)\Phi_i(x,y,z), \quad (2)$$

where m_i are the time-varying mode coefficients and Φ_i are the eigen functions of the Laplacian for the cuboidal domain, given by:

$$\Phi_i(x,y,z) = cos\left(\frac{\pi i_x}{l_x}x\right) cos\left(\frac{\pi i_y}{l_y}y\right) cos\left(\frac{\pi i_z}{l_z}z\right). \quad (3)$$

We limit the modes computed to the Nyquist rate of the maximum simulation frequency.

In order to compute the pressure, we need to compute the mode coefficients. Reinterpreting equation (2) in the discrete setting, the discrete pressure $P(X,t)$ corresponds to an inverse Discrete Cosine Transform (iDCT) of the mode coefficients $M_i(t)$:

$$P(X,t) = iDCT(M_i(t)). \quad (4)$$

Substituting the above equation in equation (1) and applying a DCT operation on both sides, we get

$$\frac{\partial^2}{\partial t^2}M_i + c^2 {k_i}^2 M_i = DCT(F(X,t)), \quad (5)$$

where ${k_i}^2 = \pi^2\left(\frac{{i_x}^2}{{l_x}^2} + \frac{{i_y}^2}{{l_y}^2} + \frac{{i_z}^2}{{l_z}^2}\right)$ and $F(X,t)$ is the force in the discrete domain. Assuming the forcing term $F(X,t)$ to be constant over a time-step Δt of simulation, the following update rule can be derived for the mode coefficients:

$${M_i}^{n+1} = 2M_i^n cos(\omega_i \Delta t) - {M_i}^{n-1} + \frac{2\widetilde{F^n}}{\omega_i^2}(1 - cos(\omega_i \Delta t)), \quad (6)$$

where $\widetilde{F} = DCT(F(X,t))$. This update rule can be used to generate the mode-coefficients (and thereby pressure) for the next time-step. This gives us a method to compute the analytical solution of the wave equation for a cuboidal domain. Next, we describe how these solutions are used to solve the wave equation over the entire scene.

3.3. Interface Handling

In order to ensure correct sound propagation across the boundaries of these subdomains, we use a 6th order finite difference stencil to patch two subdomains together. A 6th order scheme was chosen because has been experimentally determined [19] to produce reflection errors at 40 dB below the incident sound field. We derive the stencil as follows.

First, we examine the projection along an axis of two neighboring axis-aligned cuboids. Our local solution inside the cuboids assumes a reflective boundary condition, $\left.\frac{\partial p}{\partial x}\right|_{x=0} = 0$. Looking at the rightmost cuboidal partition, the solution can be represented by the discrete differential operator, ∇^2_{local}, that satisfies the boundary solutions. Referring back to the wave equation, we have the global solution:

$$\frac{\partial^2 p}{\partial t^2} - c^2 \nabla^2_{global} p = f(X,t).$$

Using the local solution of the wave equation inside the cuboid (∇^2_{local}), we derive:

$$\frac{\partial^2 p}{\partial t^2} - c^2 \nabla^2_{local} p = f(X,t) + f_I(X,t),$$

where $f_I(X,t)$ is the forcing term that needs to be contributed by the interface to derive the global solution. Therefore, we need to solve for this term, using our two previous identities:

$$f_I(X,t) = c^2\left(\nabla^2_{global} - \nabla^2_{local}\right) p. \quad (7)$$

While the exact solution to this is computationally expensive, we can approximate the solution using a 6th order finite difference stencil, with spatial step size h:

$$f_I(x_j) = \sum_{i=j-3}^{-1} p(x_i)s[j-i] - \sum_{i=0}^{2-j} p(x_i)s[i+j+1], \quad (8)$$

where $j \in [0,1,2]$, $f_I(x_j) = 0$ for $j > 2$, and $s[-3 \ldots 3] = \frac{1}{180h^2}\{2, -27, 27$

3.4. ARD computational pipeline

The ARD technique has two main stages: *Preprocessing* and *Simulation*. During the preprocessing stage, the input scene is voxelized into grid cells. The spatial discretization of the grid h is determined by the maximum simulation frequency ν_{max} determined by the relation $h = c/(\nu_{max} s)$, where s is the number of samples per wavelength (=2.66 for ARD) and c is the speed of sound ($343 m/sec$ at standard temperature and pressure). The next step is the computation of adaptive rectangular decomposition, which groups the grid cells into rectangular (cuboidal) domains. These generate the partitions, also known as *air partitions*. Pressure-absorbing layers are created at the boundary by generating Perfectly-Matched-Layer (PML) partitions. These partitions are needed to model partial or full sound absorption by different surfaces in the scene. Artificial interfaces are created between the air-air and the air-PML partitions to propagate pressure between partitions and combine the local pressure fields of the partitions into the global pressure field.

During the simulation stage, the global acoustic field is computed in a time-marching scheme as follows (see Figure 1 (top row)):

1) Local update

For all air partitions

(a) Compute the DCT to transform force F to $\widetilde{F}$.

(b) Update mode coefficients M_i using update rule.

(c) Transform M_i to pressure P using iDCT.

For all PML partitions

Update the pressure field of the PML absorbing layer.

2) Interface handling

For all interfaces

(a) Compute the forcing term F within each partition.

3) Global update

Update the global pressure field.

The first stage solves the wave equation in each rectangular region by taking the Discrete Cosine Transform (DCT) of the pressure field, updating the mode coefficients using the update rule, and then transforming the mode coefficients back into the pressure field using inverse Discrete Cosine Transform (iDCT). Both the DCT and iDCT are implemented using fast Fast Fourier Transforms (FFT) libraries. Overall, this step involves two FFT evaluations and a stencil evaluation corresponding to the update rule. The pressure fields in the PML-absorbing layer partitions are also updated in this step.

The second stage uses a finite difference stencil to propagate the pressure field between partitions. This involves a time-domain stencil evaluation for each grid cell on the boundary.

The last stage uses the forcing terms computed in the interface handling stage to update the global pressure field.

For more details, please refer to the original texts [22, 19].

4. Parallel Acoustic Simulation

In this section, we describe our ARD-based distributed parallel acoustic simulator.

4.1. Partition and Interface Ownership

This first step involves distributing the problem domain onto the cores of the cluster. In the case of ARD, the different domains include air partitions, PML partitions, and the interfaces.

Air and PML partitions The ARD solver can be parallelized because the partition updates for both the air and the PML partitions are independent at each time step: each partition update is a localized computation that does not depend on the data associated with other partitions. As a result, partitions can be distributed onto separate cores of the cluster and the partition update step is evaluated in parallel at each time step without needing any communication or synchronization. In other words, each core exclusively handles a set of partitions. These *local partitions* compute the full pressure field in memory. The rest of the partitions are marked as *remote partitions* for this core and are evaluated by other cores of the cluster. Only metadata (size, location, etc.) for remote partitions needs to be stored on the current core, using only a small amount of memory (see Figure 1 bottom row (b)).

Interfaces Interfaces, like partitions, retain the concept of ownership. One of the two cores that owns the partition of an interface, takes ownership of that interface, and is responsible for performing the computations with respect to that interface. Unlike the partition update, the interface handling step has a data dependency with respect to other cores. Before the interface handling computation is performed, pressure data needs to be transferred from the dependent cores to the owner (Figure 1 bottom c). In the next step, the pressure data is used, along with the source position, to compute the forcing terms (Figure 1 bottom d). Once the interface handling step is completed, the interface-owning core must send the results of the force computation back to the dependent cores (Figure 1 bottom e). The global pressure field is updated (Figure 1 bottom f) and used at the next time step.

4.2. Parallel Algorithm

The overall technique proceeds in discrete time steps (see Figure 1 bottom row). Each time step in our parallel ARD algorithm is evaluated through three main stages described in Section 3.4; these evaluations are followed by a barrier synchronization at the end of the time step. Each core starts the time step at the same time, then proceeds along the computation without synchronization until the end of the time step.

Local Update This step updates the pressure field in the air and PML partitions for each core independently, as described in detail in Section 3.4.

Pressure field transfer After an air partition is updated, the resulting pressure data is sent to all interfaces that require it. This data can be sent as soon as it is available.

Interface handling This stage uses the pressure transferred in the previous stage to compute forcing terms for the partitions. Before the interface can be evaluated, it needs data from all of its dependent partitions.

Force transfer After an interface is computed, the owner needs to transfer the forcing terms back to the dependent cores. A core receiving forcing terms can then use them as soon as the message is received.

Global update Each core updates the pressure field using the forcing terms received from the interface operators.

Barrier synchronization A barrier is needed at the end of each time step to ensure that the correct pressure and forcing values have been computed. This is necessary before local update is performed for the next time step.

4.3. Efficient Load Balancing

Algorithm 1 Load balanced partitioning

Require: list of cores B, list of partitions P
Require: volume threshold Q
{Initialization}
for all $b \in B$ **do**
  $b \leftarrow Q$
end for
{**Splitting**}
while $\exists p_i \in P$ where volume$(p_i) > Q$ **do**
  $P \leftarrow P - \{p_i\}$
  $(p_i', q_i') \leftarrow$ split p_i
  $P \leftarrow P \cup \{p_i', q_i'\}$
end while
{**Bin-packing**}
sort P from greatest to the least volume
for all $p_i \in P$ **do**
  volume$(b) \leftarrow \max$ volume(B)
  assign p_i to core b
  volume$(b) \leftarrow$ volume$(b) -$ volume(p_i)
end for

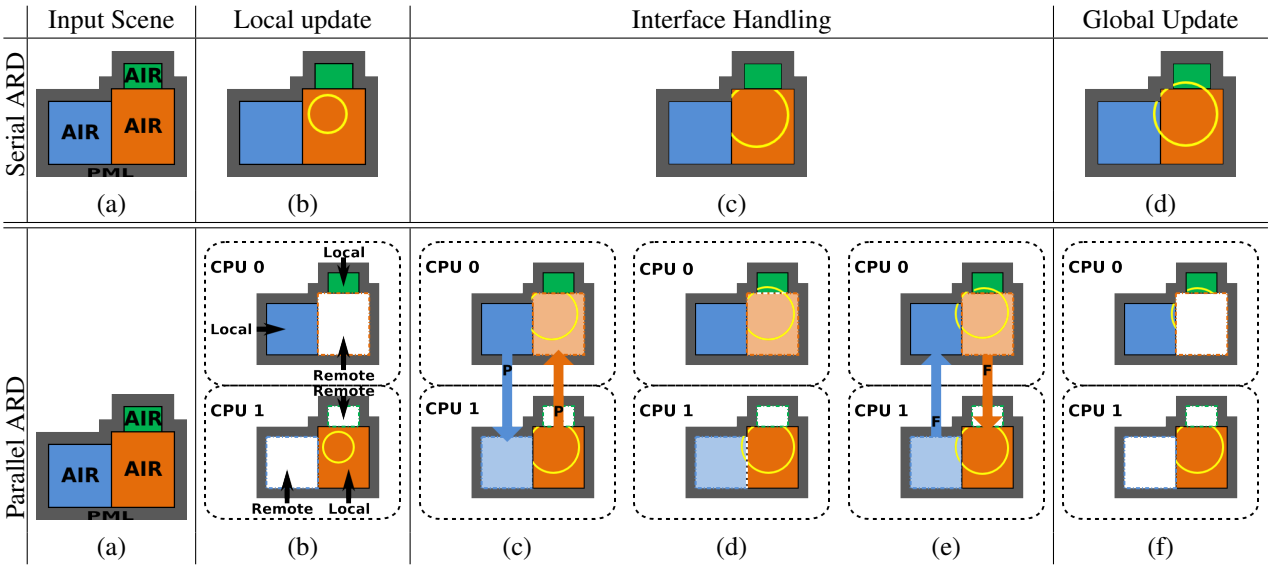


Figure 1: **Serial ARD Pipeline** (top row): Step (a) is the input into the system; step (b) is the analytical-solution update in the rectangular partitions and pressure updates in the PML partitions (c) is interface evaluation; step (d) is the global pressure-field update. **Parallel ARD** (bottom row): We start with the input (Step (a)) and the local update (b). Interface handling is split into 3 parts: the pressure is transferred to dependent interfaces in (c), the interface is evaluated in (d), and the forcing terms are transferred back in (e). Finally, the global pressure-field update is done in (f).

For each time step, the computation time is proportional to the time required to compute all of the partitions. This implies that a core with larger partitions or more partitions than another core would take longer to finish its computations; this would lead to load imbalance in the system, causing the other cores of the cluster to wait at the synchronization barrier instead of doing useful work. Load imbalance of this kind results in sub-optimal performance.

Imbalanced partition sizes in ARD are rather common as ARD's rectangular decomposition step uses a greedy scheme based on fitting the biggest possible rectangle at each step. This can generate a few big partitions and large number of small partitions. In a typical scene (e.g. the Cathedral benchmark), the load imbalance results in poor performance.

A naive load balancing scheme would reduce the size of each partition to exactly one voxel, but this would negate the advantage of the rectangular decomposition scheme's use of analytical solutions; furthermore, additional interfaces introduced during the process at partition boundaries would reduce the overall accuracy of the simulator [22]. The problem of finding the optimal decomposition scheme to generate perfect load balanced partitions while minimizing the total interface area is non-trivial. This problem is known in computational geometry as *ink minimization* [16]. While the problem can be solved for a rectangular decomposition in two dimensions in polynomial time, the three-dimensional case is NP-complete [12]. As a result, we approach the problem using a top-down approximate technique that bounds the sizes of partitions and subdivides large ones yet avoids increasing the interface area significantly. This is different from a bottom-up approach that would coalesce smaller partitions into larger ones.

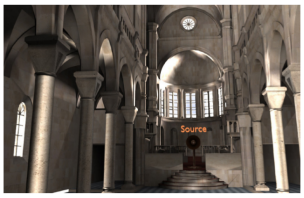
Our approach splits the partitions that exceed a certain volume $Q = \frac{V}{\mathbf{f}\ \mathbf{num\_procs}}$, where V is the total volume of the simulation domain, **num_procs** is the number of cores available, and **f** is the load balancing factor (typically 1-4, but in our implementation and our results we use $f = 1$). To split the partition, we find a dividing orthogonal plane that separates the partition p_i into two partitions of size at most Q and of size volume$(p_i) - Q$. Both are added back into the partition list. The splitting operation is repeated until no partition is of size greater than Q. Once all the large partitions are split, we allocate partitions to cores through a greedy bin-packing technique. The partitions are sorted from the greatest volume to least volume, then added to the bins (where one bin represents a core) in a greedy manner. The bin with the maximum available volume is chosen during each iteration (see Algorithm 1).

4.4. Reduced Communication Cost

Each interface in the simulation depends on the data from two or more partitions in order to evaluate the stencil. In the worst case, when the partitions are very thin, the stencil can cross over multiple partitions. This dependence means that data must be transferred between the cores when the dependent partition and the interface are in different cores.

In order to reduce the cost of this data transfer, we use an asynchronous scheme where each core can evaluate a partition while waiting for another core to receive partition data, effectively overlapping communication and computation costs.

As the problem size grows, the communication cost increases. However, the communication cost grows with the surface area of the scene, while computation grows with the volume. Therefore, computation dominates at higher problem sizes and can be used effectively to hide communication cost.



(a) The Cathedral benchmark. Alcoves, curves, and fine details generate a large number of partitions and interfaces.

(b) The Twilight benchmark. An open area in the center and sloped surfaces create partitions of widely varying sizes.

(c) The Village benchmark. The huge open area can create very large partitions.

(d) The KEMAR benchmark, meant to emulate a human head.

Figure 2: Indoor and outdoor acoustic benchmarks: our parallel simulator is the first approach that can perform high frequency numeric acoustic propagation in such benchmarks.

5. Implementation

In this section, we describe the implementation of our simulator on a distributed memory architecture, and highlight its performance on various indoor and outdoor scenes. We use a CPU cluster with 119 Dell C6100 servers or 476 compute nodes, each node with 12-core, 2.93 GHz Intel processors, 12M L3 cache, and 48 GB of physical memory. All the nodes of the cluster are connected by Infiniband (PCI-Express QDR) interconnect.

Preprocessing The preprocessing stage is single threaded and run in two steps. The first step is the voxelization, which can be done in seconds even on very complex scenes. The second step, partitioning, can be done within minutes or hours depending on the scene size and complexity. However, this is a one time cost for a specific scene configuration. Once we have a partitioning, we can further upsample and refine the voxel grid to simulate even higher frequencies; however, this will smooth out finer details of the mesh. This allows us to run the preprocessing step only once for a scene at different frequency ranges.

Simulator Initialization An initialization step is run on all partitions, determining which interfaces they belong to and whether or not they will receive forcing terms back from the interface. Therefore, each core knows exactly which cores it should send to and receive from, allowing messages to be received from other cores in no particular order. This works hand in hand with the independence of operations (see Section 4.1): interfaces can be handled in any order depending on which messages are received first.

Interface Handling Optimizations In scenes with a large number of interfaces on separate cores, the amount of data that is sent between cores can quickly grow. Although a partition stores the pressure and forcing data, the interface handling computation only needs the pressure as an input and generates the forcing data as the output. Therefore, only half of the data at the surface of the partition needs to be sent and received.

We send partition messages using an asynchronous communication strategy instead of a collective communication approach. Asynchronous communication allows us to send messages while computation is being performed. The collective communication approach requires that all cores synchronize at the data-transfer call (since each core needs to both send and receive data), causing some cores to wait and idle.

Benchmark Scenes In order to evaluate our parallel algorithm, we use five benchmark scenes. The first, Cube, is an optimal and ideal case, designed to show the maximum scalability of our simulator with frequency and volume. It is perfectly load-balanced and contains a minimum number of interfaces. The second scene, Cathedral, was chosen for its spatial complexity. Cathedral is a large indoor scene that generates partitions of varying sizes: many large-sized partitions (which can cause load imbalance) and a large number of tiny partitions and interfaces. The third scene, Twilight, is the Twilight Epiphany skyspace from Rice University. The main feature of the scene is its sloped surfaces, which can cause the generation of smaller partitions. The fourth scene, Village, is a huge open area with scattered buildings. This outdoor scene is useful since it can generate very large partitions. This can cause problems for a simulator that does not handle load imbalance properly. The final scene, KEMAR, is a head model used to simulate how acoustic waves interact with the human head. The fine details of the human head (esp. ears) cause the generation of very small partitions. The scene is small (only $33m^3$) and can be simulated up to 22kHz.

6. Results and Analysis

6.1. Scalability

Figure 3 shows the performance of our simulator with increasing numbers of CPU cores of the cluster. We show near-linear scaling for three benchmark scenes: Village, Cathedral, and Twilight, up to 1024 CPU cores.

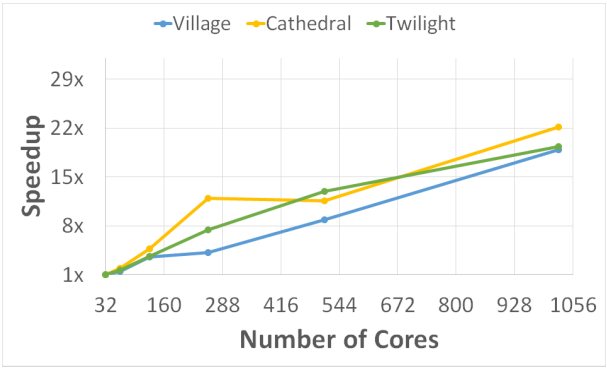
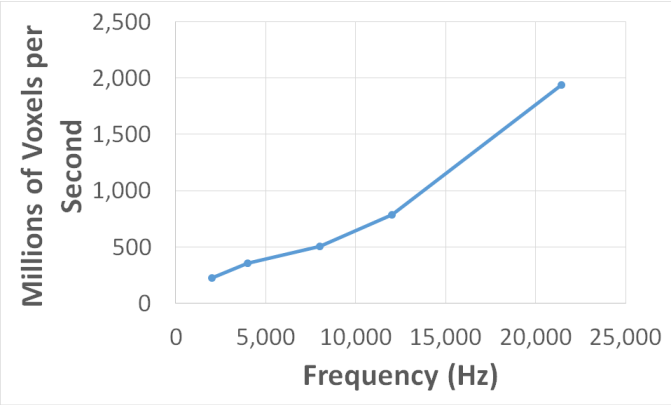
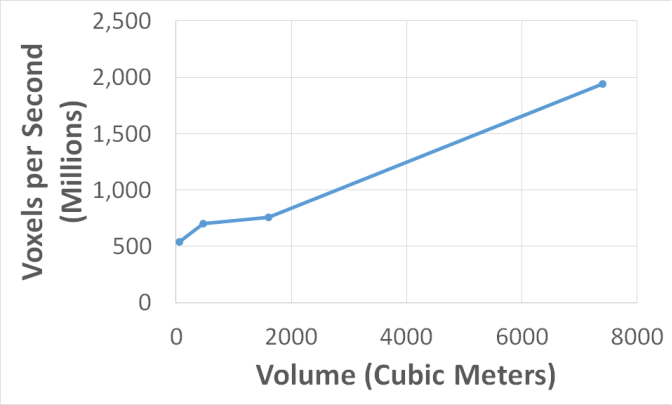


Figure 3: Performance scaling of our simulator with increasing number of CPU cores. Speedup with X cores = (Simulation time on 32-cores / Simulation time over X-cores). Village was run at 2000Hz, Cathedral was run at 3000Hz, and Twilight was run at 6000Hz

We perform scalability analysis of our parallel simulator as the scene volume and scene frequency increase. We run the acoustic simulation on the cube scene while keeping the number of CPU cores fixed at 128 and scene volume size fixed at $20m$ x $20m$ x $20m$; we vary the simulation frequency using these values: $2kHz$, $4kHz$, $8kHz$, $12kHz$ and $22kHz$ (see Figure 4(a)). The computational complexity of the ARD wave-solver increases with the fourth power of frequency. For smaller frequencies, the amount of computation is relatively small, so the system does not fully utilize 128 CPU-cores. It also becomes much harder to hide the communication cost by overlapping it with computation, since the computation cost per node is small. However, as the frequency increases, all the 128 cores are utilized, and the computation cost easily overcomes the communication cost, resulting in higher peak performance. We then perform the acoustic simulation for the cube scene with varying volume (Figure 4(b)). We scale the scene from a volume of $50m^3$ to $8000m^3$, while keeping the maximum frequency fixed at $22kHz$ and the number of cores fixed at 128. The computational complexity of the ARD wave-solver varies almost linearly with the volume of the scene. As shown in Figure 4(b), the amount of computation available at low volumes is considerably less, resulting in under-utilization of the CPU cores and computation cost not hiding the communication cost. As the volume increases, both these factors are ameliorated, and we observe higher performance and throughput.



(a)



(b)

Figure 4: Performance of the simple cube scene. (a) Our simulation performs better as frequency increases. At high frequencies, compute begins to dominate over communication cost and interface handling cost. (b) As the volume of the scene increases, partition computation begins to dominate over communication and interface handling. Since the computation has no data dependencies, the overall throughput increases.

6.2. *Load balancing*

Table 1 shows the benefit of our load balancing scheme in terms of the reduction of wait time at the synchronization barrier. The computation time at each node is proportional to the total number of grid cells in local partitions residing on that node. In the case of perfect load balancing, all the nodes will have same number of grid cells, require identical computation time, and reach the synchronization barrier at the same moment, resulting in zero wait time. In the case of load imbalance, some nodes will have more number of grid cells than others. In this case, the node with the minimum number of grid cells will finish the computation first, and the node with maximum number of grid cells will finish last. Therefore, the wait time at the barrier will be proportional to the load-imbalance factor, defined as $\frac{|N\_max - N\_min|}{N\_min}$, where N_max and N_min are the maximum and minimum number of grid cells over all nodes, respectively. Table 2 shows the average memory used per core for our simulator on the CPU-cluster. The average memory used per core is below the maximum memory available per core (48 GB / 12 cores = 4 GB per core) on the cluster we used.

In Figure 5, we compare the performance scaling for the simulation with and without the load balancing method. This experiment is repeated for four scenes: Cathedral, Twilight, Village, and KEMAR. As shown, the simulation with load balancing scales almost linearly with the number of cores, whereas the simulation without load balancing fails to scale after 64 cores. This is primarily due to the long wait times at the barrier synchronization. All the nodes have to wait for the slowest node (with the maximum number of grid cells) to catch up to the barrier before they can start the computation for the next time-step. This is shown in the timing results in Figure 6, where the simulation without the load balancing shows a significant increase in the barrier time compared to the load-balanced one. The load imbalance also affects the partition update and the remote interface handling step, since remote partitions residing on the slowest node can delay the processing of all the neighboring nodes.

Because the partition-splitting algorithm adds new interfaces to the simulation, splitting can introduce error. We have quantified this error on a simple 10kHz test scene. Because we can determine the exact solution of the wave equation on a rectangular scene, we use a single partition as the ground truth reference. We then subdivide the rectangular area in order to increase the interface area and compare the resulting pressure field with the exact solution. Figure 7 shows the error as the total interface area increases on the 10kHz cube scene.

6.3. *Asynchronous Communication*

There are two main categories of communication possible in our simulator: asynchronous and collective. The asynchronous

Scene	#cores	Thresh.	Load Ratio Before	Load Ratio After	#partitions added	Area added (in m^2)
Cathedral	64	300	20.0	1.06	23	0.53 M
(19K m^3,	128	150	47.5	1.07	56	1.1 M
4kHz)	256	75	123.4	1.07	138	2.3 M
Twilight	64	128	6.9	1.06	26	0.16 M
(8.2K m^3,	128	64	20.8	1.07	67	0.35 M
4kHz)	256	32	65.7	1.06	164	0.75 M
Village	64	5514	53.3	3.35	18	0.36 M
(353K m^3,	128	2757	189.1	4.70	43	0.76 M
2kHz)	256	1378	720.3	7.20	103	1.6 M
KEMAR	64	2185454	19354.4	1.30	24	1.6 M
(33 m^3,	128	1092727	121497	1.32	54	3.2 M
22kHz)	256	546363	456933	1.28	118	6.4 M

Table 1: **Benefits of load balancing**: Comparison of the rectangular decomposition step without and with our load balancing for the three benchmark scenes. Abbreviations: *#cores* denotes the number of CPU cores, $Thresh$ = scene volume/*#cores* is the parameter of our load balancing method, and *#partitions added* is the total number of partitions added from splitting. Load Ratio is a metric to measure the total wait time of the system and is equal to the worst case load imbalance in the system, computed as $\frac{|N\_max - N\_min|}{N\_min}$, where N_max and N_min are the maximum and minimum number of grid cells over all nodes, respectively.

Scenes	sim. freq.	avg. mem./core
Cathedral	3 kHz	3.9 GB
Village	2 kHz	3.0 GB
Twilight	6 kHz	2.2 GB

Table 2: **Memory usage:** The average memory usage per core for our simulator on the CPU-cluster with 1024 cores. Abbreviations: "sim. freq." denotes the simulation frequency and "avg. mem./core" is the average memory used per core.

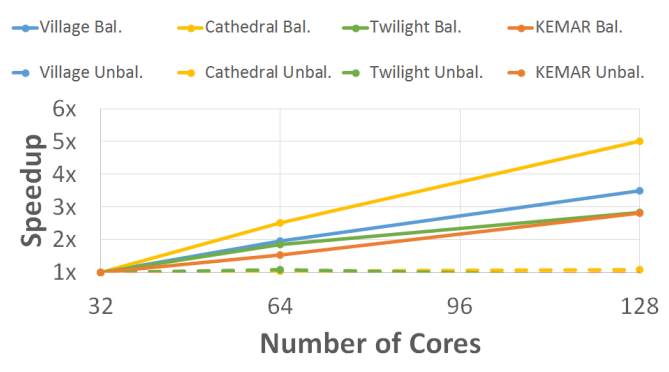


Figure 5: Our load balancing method allows us to scale to a larger number of cores. Speedup with X cores = (Simulation time on 32-cores / Simulation time over X-cores). Note how the simulator fails to achieve speedup past 64 cores without load balancing since all the nodes get stalled and wait for computation to finish for the slowest node.

communication approach is what was discussed earlier in section 4.4. This is different from a collective communication approach which transfers messages from all cores at the same time.

In our acoustic simulations, the asynchronous communication approach shows clear advantages over a more traditional collective communication MPI approach. Figure 8 shows the reduction of force-transfer times using asynchronous communication. Even the complex Cathedral scene, with its many small interfaces that make overlapping communication with computation difficult, still proves more efficient on a larger number of cores. Additionally, we show in Figure 9 that our approach improves the scalability of the simulation; as we increase the number of cores, the performance gap widens.

6.4. Comparison with FDTD

Wave-based methods for solving the acoustic wave equation are typically prone to numerical dispersion errors, in which waves with different frequencies do not travel at the same speed. This results in loss of phase relations in the source signal, effectively destroying the signal after certain distance: the result is a muffled sound. In order to keep the numerical dispersion error low, the spatial discretization for the wave-based method needs to be kept high. Ideally, the spatial discretization for a given maximum frequency is determined by the Nyquist theorem: the size of the grid cell must be at most half the minimum wavelength. In practice, wave-based methods such as FDTD often require a spatial discretization which is 1/10 times the minimum wavelength, although recent methods have focused on reducing the refinement of spacial discretization [17, 27].

Because ARD uses the rectangular-domain analytical solution of wave equation, it exhibits extremely low numerical dispersion error. Therefore, ARD can work at a spatial discretization that is 1/2.6 times the minimum wavelength [22, 19]. Since the memory requirements of both ARD and FDTD scale inversely with the third power of spatial discretization, ARD is 25-50 times more memory efficient than FDTD. The computational cost also scales inversely with the fourth power of the spatial discretization: ARD algorithm is 75-100 times faster than the FDTD algorithm.

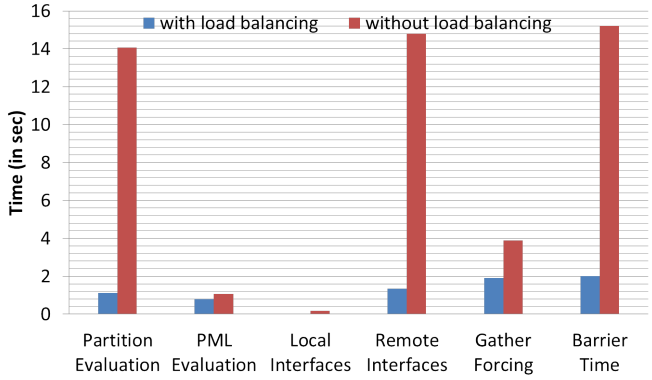


Figure 6: In this example from the Village scene, the breakdown of the maximum timing of each stage for a core shows how the delay from an unbalanced partition propagates throughout the time step. The maximum remote interface handling time and the maximum barrier time are longer as a result of the imbalance due to which the nodes are stalled and waiting for computation of the slowest node to finish.

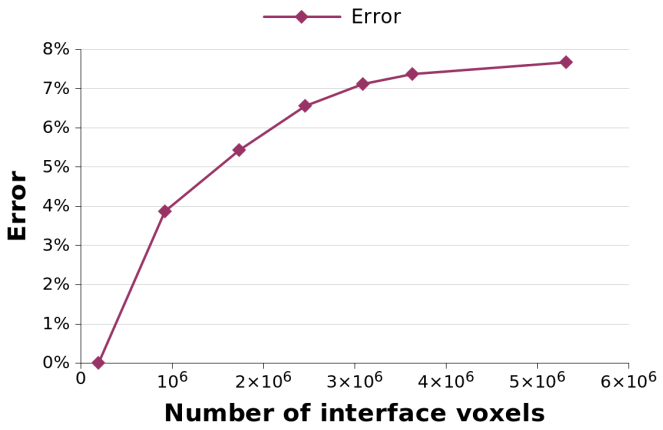


Figure 7: A comparison between interface area and overall normalized error over the entire scene. The error percent is calculated as $\frac{\Sigma_{v_i}[p_{sim}(v_i)-p_{ref}(v_i)]^2}{\Sigma_{v_i}[p_{ref}(v_i)]^2}$ where v_i loops over every voxel in the scene over all time steps, p_{sim} is the simulated pressure and p_{ref} is the reference pressure.

7. Conclusion and Future Work

We present a massively parallel time-domain solver for the acoustic wave equation. In order to accelerate the performance, we present novel algorithms for load balancing and overlapping the computation of pressure field with communication of interface data. We take advantage of the separability of partition updates to independently calculate pressure terms in rectangular domains distributed over multiple cores. We perform asynchronous MPI calls to ensure that a process running on each core is performing useful computations while communicating with other nodes.

Our load balancing scheme provides a marked improvement over a naive bin-packing approach. We achieve better performance for all three scenes, as well as providing scalability. Moreover, as the size of the scene increases or the frequency of the simulation increases, performance improves. This allows

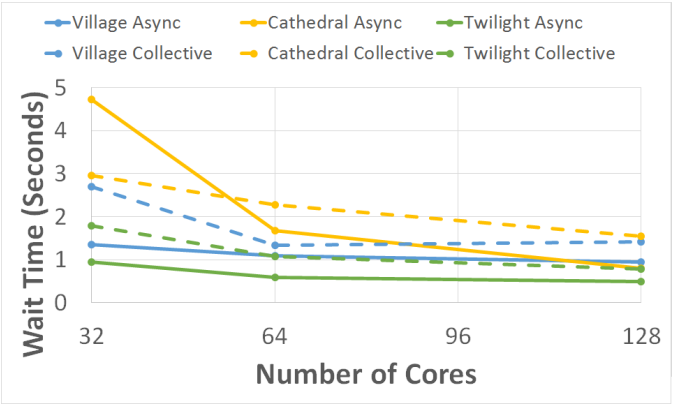


Figure 8: Asynchronous communication reduces the force transfer time compared to a traditional collective communication approach.

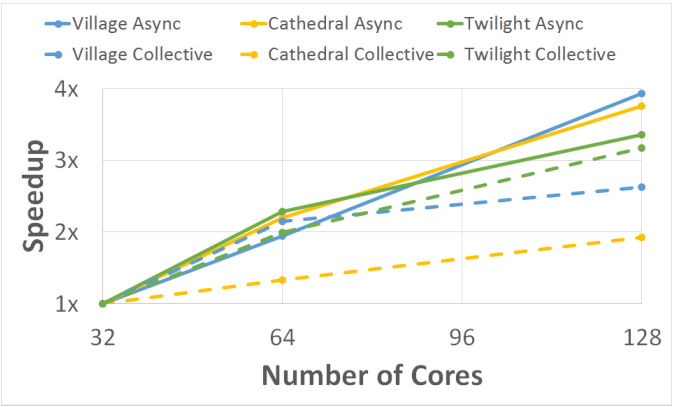


Figure 9: Asynchronous communication provides better scalability than the traditional collective communication approach. While the Twilight scene has approximately the same scaling, Cathedral shows a clear win as does Village at a higher number of cores.

us to simulate at high frequencies while taking full advantage of cluster resources. As compared to prior wave solvers, our solver is the first algorithm and system that can perform numeric simulation at high frequencies for large indoor and outdoor scenes.

There are many avenues for future work. We would like to evaluate the performance of our solver on larger CPU or GPU clusters with tens or hundreds of thousands of cores and achieve petaFLOPS performance on very high frequencies (an improved version of our algorithm scalable to tens of thousands of cores is presented in [20]). It will also be useful to extend our approach to heterogeneous environments with varying speed of sound.

Acknowledgments

This research was supported in part by the Link Foundation Fellowship in Advanced Simulation and Training, ARO Contract W911NF-14-1-0437, and the National Science Foundation (NSF awards 1320644 and 1345913). We want to thank Alok Meshram, Lakulish Antani, Anish Chandak, Joseph Digerness, Alban Bassuet, Keith Wilson, and Don Albert for useful discussions and models.

[1] Alghamdi, A., Ahmadia, A., Ketcheson, D. I., Knepley, M. G., Mandli, K. T., Dalcin, L., 2011. Petclaw: A scalable parallel nonlinear wave propagation solver for python. In: Proceedings of the 19th High Performance Computing Symposia. Society for Computer Simulation International, pp. 96–103.

[2] Allen, J. B., Berkley, D. A., 1979. Image method for efficiently simulating small-room acoustics. The Journal of the Acoustical Society of America 65 (4), 943–950.

[3] Bao, H., Bielak, J., Ghattas, O., Kallivokas, L. F., O'Hallaron, D. R., Shewchuk, J. R., Xu, J., 1998. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. Computer methods in applied mechanics and engineering 152 (1), 85–102.

[4] Bernacki, M., Fezoui, L., Lanteri, S., Piperno, S., 2006. Parallel discontinuous galerkin unstructured mesh solvers for the calculation of three-dimensional wave propagation problems. Applied mathematical modelling 30 (8), 744–763.

[5] Bhandarkar, M. A., Kalé, L. V., 2000. A parallel framework for explicit fem. In: High Performance ComputingHiPC 2000. Springer, pp. 385–394.

[6] Bilbao, S., 2013. Modeling of complex geometries and boundary conditions in finite difference/finite volume time domain room acoustics simulation. Audio, Speech, and Language Processing, IEEE Transactions on 21 (7), 1524–1533.

[7] Chan, T. F., Mathew, T. P., 1994. Domain decomposition algorithms. Vol. 3. Cambridge Univ Press, pp. 61–143.

[8] Chen, J.-T., Lee, Y.-T., Lin, Y.-J., 2010. Analysis of mutiple-shepers radiation and scattering problems by using a null-field integral equation approach. Applied Acoustics 71 (8), 690–700.

[9] Ciskowski, R. D., Brebbia, C. A., 1991. Boundary element methods in acoustics. Computational Mechanics Publications Southampton, Boston.

[10] Crocker, M. J., 1998. Handbook of acoustics. John Wiley & Sons.

[11] Diekmann, R., Dralle, U., Neugebauer, F., Römke, T., 1996. Padfem: a portable parallel fem-tool. In: High-Performance Computing and Networking. Springer, pp. 580–585.

[12] Dielissen, V. J., Kaldewaij, A., 1991. Rectangular partition is polynomial in two dimensions but np-complete in three. Information Processing Letters 38 (1), 1–6.

[13] Engquist, B., Runborg, O., 2003. Computational high frequency wave propagation. Acta numerica 12, 181–266.

[14] Funkhouser, T., Tsingos, N., Jot, J.-M., 2003. Survey of methods for modeling sound propagation in interactive virtual environment systems. Presence and Teleoperation.
URL http://www-sop.inria.fr/reves/Basilic/2003/FTJ03

[15] Guiffaut, C., Mahdjoubi, K., 2001. A parallel fdtd algorithm using the mpi library. Antennas and Propagation Magazine, IEEE 43 (2), 94–103.

[16] Keil, J. M., 2000. Polygon decomposition. Handbook of Computational Geometry 2, 491–518.

[17] Kowalczyk, K., van Walstijn, M., 2011. Room acoustics simulation using 3-d compact explicit fdtd schemes. Audio, Speech, and Language Processing, IEEE Transactions on 19 (1), 34–46.

[18] Mehra, R., Antani, L., Kim, S., Manocha, D., 2014. Source and listener directivity for interactive wave-based sound propagation. Visualization and Computer Graphics, IEEE Transactions on 20 (4), 495–503.

[19] Mehra, R., Raghuvanshi, N., Savioja, L., Lin, M. C., Manocha, D., 2012. An efficient gpu-based time domain solver for the acoustic wave equation. Applied Acoustics 73 (2), 83–94.

[20] Morales, N., Chavda, V., Mehra, R., Manocha, D., 2015. Mpard: A scalable time-domain acoustic wave solver for large distributed clusters. Tech. rep., Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina.

[21] Operto, S., Virieux, J., Amestoy, P., LExcellent, J.-Y., Giraud, L., Ali, H. B. H., 2007. 3d finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. Geophysics 72 (5), SM195–SM211.

[22] Raghuvanshi, N., Narain, R., Lin, M. C., 2009. Efficient and accurate sound propagation using adaptive rectangular decomposition. Visualization and Computer Graphics, IEEE Transactions on 15 (5), 789–801.

[23] Saarelma, J., Savioja, L., 2014. An open source finite-difference time-domain solver for room acoustics using graphics processing units. Acta Acustica united with Acustica.

[24] Saied, F., Holst, M. J., 1991. Multigrid methods for computational acoustics on vector and parallel computers. Urbana 51, 61801.

[25] Sakamoto, S., Ushiyama, A., Nagatomo, H., 2006. Numerical analysis of sound propagation in rooms using the finite difference time domain method. The Journal of the Acoustical Society of America 120 (5), 3008–3008.

[26] Savioja, L., 2010. Real-time 3d finite-difference time-domain simulation of low-and mid-frequency room acoustics. In: 13th Int. Conf on Digital Audio Effects. Vol. 1. p. 75.

[27] Savioja, L., Valimaki, V., 2000. Reducing the dispersion error in the digital waveguide mesh using interpolation and frequency-warping techniques. Speech and Audio Processing, IEEE Transactions on 8 (2), 184–194.

[28] Schissler, C., Mehra, R., Manocha, D., 2014. High-order diffraction and diffuse reflections for interactive sound propagation in large environments. ACM Transactions on Graphics (SIGGRAPH 2014) 33 (4), 39.

[29] Sheaffer, J., Fazenda, B., 2014. Wavecloud: an open source room acoustics simulator using the finite difference time domain method. Acta Acustica united with Acustica.

[30] Sypek, P., Dziekonski, A., Mrozowski, M., 2009. How to render fdtd computations more effective using a graphics accelerator. Magnetics, IEEE Transactions on 45 (3), 1324–1327.

[31] Thompson, L. L., 2006. A review of finite-element methods for time-harmonic acoustics. J. Acoust. Soc. Am 119 (3), 1315–1330.

[32] Vaccari, A., Cala'Lesina, A., Cristoforetti, L., Pontalti, R., 2011. Parallel implementation of a 3d subgridding fdtd algorithm for large simulations. Progress In Electromagnetics Research 120, 263–292.

[33] Wang, S., de Hoop, M. V., Xia, J., Li, X. S., 2012. Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3-d anisotropic media. Geophysical Journal International 191 (1), 346–366.

[34] Webb, C. J., Bilbao, S., 2012. Binaural simulations using audio rate fdtd schemes and cuda. In: Proc. of the 15th Int. Conference on Digital Audio Effects (DAFx-12), York, United Kingdom.

[35] Yeh, H., Mehra, R., Ren, Z., Antani, L., Manocha, D., Lin, M., 2013. Wave-ray coupling for interactive sound propagation in large complex scenes. ACM Transactions on Graphics (TOG) 32 (6), 165.

[36] Yu, W., Liu, Y., Su, T., Hunag, N.-T., Mittra, R., 2005. A robust parallel conformal finite-difference time-domain processing package using the mpi library. Antennas and Propagation Magazine, IEEE 47 (3), 39–59.

[37] Yu, W., Yang, X., Liu, Y., Ma, L.-C., Sul, T., Huang, N.-T., Mittra, R., Maaskant, R., Lu, Y., Che, Q., et al., 2008. A new direction in computational electromagnetics: Solving large problems using the parallel fdtd on the bluegene/l supercomputer providing teraflop-level performance. Antennas and Propagation Magazine, IEEE 50 (2), 26–44.