A METHOD FOR SPEEDING UP BEAM-TRACING SIMULATION USING THREAD-LEVEL PARALLELIZATION

Marjan Sikora, Ivo Mateljan

University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture,
R. Boskovica 32, 21000 Split, Croatia
(sikora@fesb.hr, ++385 21 305-859, ++385 21 305-776)

Abstract: *In recent years, the computational power of modern processors has been increasing mainly because of the increase in the number of processor cores. Computationally intensive applications can gain from this trend only if they employ parallelism, such as thread-level parallelization. Geometric simulations can employ thread-level parallelization because the main part of a geometric simulation can be divided into a subset of mutually independent tasks. This approach is especially interesting for acoustic beam tracing because it is an intensive computing task. This paper presents the parallelization of an existing beam-tracing simulation composed of three algorithms. Two of them are iterative algorithms, and they are parallelized with an already known technique. The most novel method is the parallelization of the third algorithm, the recursive octree generation. To check the performance of the multi-threaded parallelization, several tests are performed using three different computer platforms. On all of the platforms, the multi-threaded octree generation algorithm shows a significant speedup, which is linear when all of the threads are executed on the same processor.*

Key words: acoustic simulation, beam tracing, thread-level parallelization, multi-core processor

1. INTRODUCTION

Acoustic simulations are computationally intensive applications. Recent interactive auralizations compute the impulse response for a moving source and listener and even account for changing geometry [1-6]. Then, they perform auralization using the convolution of the room impulse response and the excitation audio signal. Another type of acoustic simulation computes the spatial distribution of the sound pressure, the reverberation time, the speech transmission index and other parameters, while including an ever-increasing number of sound wave effects. In ray-tracing and hybrid simulations, diffuse reflections and diffraction are now generally incorporated [7, 8], and recently, a beam-tracing simulation has been developed that includes refraction [9]. In the case of complex geometry and multiple sound sources, these simulations require a significant amount of computing power.

The increase in the computing demands of acoustic simulations must be met by an increase in the computing power of the system that performs the simulation. In recent years, the clock frequency of processors has not increased because of physical constraints, but the computational power of modern processors is increasing because manufacturers are multiplying the number of cores in the processor [10]. To use an increased number of cores, one must resort to parallelization. There are three ways in which a modern acoustic simulation can be parallelized:

(1) by computing with clusters or grids of computers or processors [11];
(2) by transferring parts of the simulation to a graphical processing unit (GPU), which has a large number of simple parallel cores [2];
(3) by using thread-level parallelization, which converts a single-threaded simulation to a multi-threaded simulation that can execute on a modern multicore central processor unit (CPU).

In some implementations, a hybrid GPU+CPU parallelization is used [5, 12].

Transforming an acoustic simulation to run on a cluster or a grid of computers would provide an enormous amount of processing power, but doing so would require a fundamental change in the code of the simulation and even a change in the programming language that is used. Additionally, such a simulation would require a substantial amount of time for communication between the cluster or grid nodes to synchronize the work because communication would be performed externally, over the network, which is much slower than the internal CPU or GPU communication.

Recently, simulations that run on a GPU [2, 13] have become common. Modern GPUs have large numbers of processing units; as a result, they provide an affordable yet very powerful platform for computationally intensive tasks. A recent implementation in which acoustic ray tracing was performed in parallel on a GPU and auralization was performed on a CPU has proven that auralization at interactive rates is possible, even for a highly complex changing geometry [5].

The drawback of this approach to parallelization is that it requires rewriting most of the simulation code because GPUs have proprietary programing interfaces such as CUDA or OpenCL. Additionally, the processor cores of the GPU have a limited number of registers and operations compared to CPU processor cores. This constraint imposes limitations on the programming methods, such as recursion, that are used in simulations.

The authors have recently developed a beam-tracing method with refraction [9]. This method is based on the geometrically complex process of beam tracing, and it calculates not only the reflection but also the refraction of the sound. Beam tracing in general is a computationally intensive method, and in the past, different acceleration techniques have been used to speed up simulation. The accelerated beam-tracing algorithm [6] uses several methods to accelerate the calculation, and it has succeeded in achieving interactive rates for scenes with moving sources in the case of simple geometry. This method is similar to our beam-tracing method in that it has a preprocessing phase in which spatial data structures are created. In the preprocessing phase, the method calculates the beam tree without any occlusion tests, while in the running phase, it employs fail-plane and skip-sphere accelerating techniques. However, the accelerated beam-tracing algorithm does not use parallelism even though (because beams are independent of each other) the beam-tracing method is a highly parallelizable technique.

The beam tracing with refraction method that we have developed is computationally intensive, which motivated us to parallelize the simulation to speed up processing. When considering which approach to use, we decided to use thread-level parallelization. The method is composed of two iterative algorithms and one recursive algorithm. We used a well-known method to parallelize the two iterative algorithms, and the main novelty in our work is the method used to parallelize the recursive octree generation algorithm.

The remainder of this paper is structured as follows. Section 2 presents previous work in the field of multi-threading in geometrical simulations. Our novel method of parallelizing the recursive octree generation algorithm is presented in Section 3. In Section 4, the results of testing several aspects of our multi-threading algorithms are presented. Finally, the conclusions are given in Section 5.

2. PREVIOUS WORK

Thread-level parallelization exploits the features of the operating system to execute several threads simultaneously. This approach was originally developed to enable different programs to run concurrently on a single core processor by using time sharing. However, thread-level parallelization soon evolved and was then used for the parallel execution of a single program on a multicore processor. Such a multi-threaded program starts several working threads, which share the same address space. The workload distribution is performed by the master thread, which starts, stops and synchronizes working threads. If different working threads that run simultaneously write and read to the same data structure, then a synchronization mechanism must be established.

The complexity of the thread-level parallelization and its efficiency are governed by Amdahl's law [14]. In any parallel algorithm, there is always a part of the code that cannot be parallelized. Amdahl's law states that the smaller the amount of this type of code (the serial part of the code), the more efficiently the code with the same number of parallel threads will perform. Thus, the main issue in designing an efficient multi-threaded algorithm is to maximize the parallel part of the code and to minimize the serial overhead. In addition to the work that the master thread performs in preparing and managing the threads, the overhead includes thread communication and synchronization as well as thread idle times that result from sub-optimal load balancing and redundant computations. The measurements that we performed and presented in section 4.2 show that the multi-threading octree generation algorithm has a slower speedup than the other two iterative algorithms. This result occurs because the system has a large amount of serial code in the master thread that prepares the root node of the octree and that cannot be parallelized.

The operating system decides which thread will be run on which processor core. If there are more threads than there are cores in the processor, then the operating system uses time sharing and switches the execution of several threads on the same core. Thus, parallelization of the simulation would result in a speedup only when there is at least the same number of cores as there are threads that run in parallel. This arrangement is clearly shown in section 5, where even with hyper-threading, there was no speedup when the number of threads was greater than the number of physical cores on the machine.

Geometric simulations can employ thread-level parallelization because the main part of the geometric simulation can be divided into a subset of mutually independent tasks. A successful example of such parallelization is the ray tracing visualization and

auralization that has been recently developed [12, 15], as well as FastV [16]. FastV uses 4-sided volumetric frusta for conservative, from-point visibility computation and geometric sound propagation simulations. The algorithm is highly parallelizable because the calculations for each frustum can be performed independently. The results that each thread computes are combined to compute the final potential visibility set. The visibility computation technique has been implemented and tested on a 16-core computer, and it has been shown to be almost linear, with a 16 times speedup for simpler models and a 13 times speedup for more complex models.

The most difficult part of the thread-level parallelization is synchronization of the threads. The most efficient and robust multi-threaded application is an application in which the threads can work independently of each other. In a geometric simulation, such as the ray-tracing, the simulation traces the propagation of the rays through the simulated model. Rays do not depend on each other and instead depend only on the model geometry. Thus, the ray-tracing algorithm is a highly parallelizable algorithm [1, 15]. In geometric simulations, the synchronization between working threads, which allows one thread to change the global state or change the same data, is minimal or even nonexistent [11]. We have decided to use the standard master thread-working thread model without any synchronization, which is accomplished by using independent data queues.

Different implementations of thread-level parallelization employ threads differently. Some simulations have a fixed number of threads that are dedicated to specific tasks. Taylor et al. [1], in RESound, use one thread to calculate first-order specular reflections, seven threads to calculate the next three orders of specular reflections and two orders of diffraction, and finally seven threads to calculate the diffuse reflections. Other simulations, such as Manta, which was designed by Bigler et al. [11], use dynamic load-balanced multi-threading. In this method, each thread obtains tasks dynamically, depending on the current state of the simulation. Such a method often employs threads to process ray packets rather than using individual rays to decrease the overhead. We have decided to use the shared task queue and dynamical thread task management.

The approach we present in this paper uses a lock-free conservative local mechanism, where the work thread manages the task queue and each thread has an independent local data structure for its results. By using this approach, we avoid mutual exclusion locks and synchronization issues. The research community has proven that this mechanism is the fastest method [12]. We use this approach even in our multi-threading version of the recursive octree generation algorithm, where we have transformed the recursive method to the iterative method with the task queue.

Geometric simulations often use spatial data structures such as Binary Space Partition (BSP) trees and k-dimensional (kd) trees [11] to speed up geometric operations. The creation and the processing of such structures employ recursive algorithms. Thread-level parallelization is easily employed on iterative algorithms, but adapting the recursive algorithm to perform in a multi-threaded manner poses a substantial challenge [17].

Several papers present parallel octree algorithms that run on cluster of computers. Pombo [18] and Tu [19] present parallel octree algorithms used for the FEM simulation. The performance of Octor simulation developed by Tu [19] shows relatively low speedup – only 250% for 16 parallel processors, but it can scale up to 2000 processors. Harihan and Aluru [20] have achieved better results with their algorithm for creation and querying of the compressed octree. They got the speedup of 400% for eight threads, and 666% for 16 threads. Our method has achieved better speedup as shown in section 4, but also the limitation when it performs on more than one processor.

In the following section, we present our method for parallelizing the octree generation recursive algorithm.

3. MODELS AND METHODS

This paper presents the multi-threaded version of the beam-tracing method. Acoustic beam tracing can benefit from thread-level parallelization because it is a computing-intensive task. This task is suitable for being parallelized because the traced beams are independent of each other.

The beam-tracing method comprises three main algorithms [9]:

1. the beam tracing
2. the creation of the octree of beams
3. the raster generation

where the first and the third phases are iterative algorithms, and the second phase is the recursive algorithm.

The first phase performs the beam tracing. Beam tracing is initiated by the creation of 20 initial beams, using an icosahedron with the center in the location of the sound source. Each of 20 initial beams is then divided according to the geometry of the surface that it hits. From the divided incoming beams, reflected and refracted beams are created.

In the second phase, all of the beams that result from the beam tracing are placed into the octree. This task is performed to speed up the spatial search of the beams, which is performed in the last phase.

In the third and last phase, the spatial distribution of the levels of sound intensity is created in the form of a raster of points. For each point in the raster, the octree is filtered to obtain those beams that contain the point. The intensity level of the sound for the point is then calculated by adding the level of intensity of each beam in the position of the point.

To parallelize the two iterative algorithms, we have used an already known method that has a set of symmetric working threads; these threads receive their tasks from the shared task queue.

For the recursive algorithm, we first attempted a naive approach in which we parallelized only the first level of the octree subdivision. However, because of load balance problems, we changed our approach and used full parallelization in which each level of subdivision is added to the task queue and run in parallel.

In the remainder of this section, we will describe how we converted to the multi-threaded form, first for the two iterative algorithms and then for the recursive algorithm.

3.1. Two iterative algorithms

The two iterative algorithms that present the first and third phase of the beam tracing are parallelized in a similar fashion. Both algorithms have two procedures: the control procedure, which distributes the work, and the thread procedure, which performs the processing.

The control procedure begins with the creation of the task queue. Tasks are initial beams in the case of beam tracing and are raster points in the case of raster generation. The master thread then starts all of the working threads simultaneously by giving each working thread one task from the shared task queue.

The central part of the control procedure is a loop in which the algorithm restarts the thread that has finished its task. When one thread finishes processing, the master thread restarts it and gives it a new task. This loop repeats until there are no tasks left in the task queue.

Then, the final loop waits for all of the threads to finish processing and terminates the threads. Although the master thread distributes tasks from the shared task queue, the results of each working thread are stored in a local result queue. For each terminated thread, the results are aggregated. In the case of beam tracing, the finished beams of each thread are moved to the common finished beams list, and in the case of raster generation, the raster points are moved to the unified raster point list.

3.2 Multi-threaded generation of the beam octree

The second phase of the beam-tracing simulation is the recursive algorithm that generates the octree of beams. The octree contains beams from the beam-tracing phase, and it is used to speed up the next (third) phase – calculation of the results.

```
process_octree_node(node n, beams list BL)
   create eight subspace nodes SN[1] .. SN[8]
   create eight node beam lists NBL[1] .. NBL[8]
   for each node number i in SN[1] .. SN[8]
      for each beam b in BL
         if node number i contains b
               push b in NBL[i]
      if NBLi contains more beams than threshold
         process_octree_node(SN[i], NBL[i])
```

Fig 1 The single-threaded recursive octree algorithm

The original recursive algorithm is presented in Figure 1. The space of the processed node is first divided into eight equal subspaces, and for each subspace, a new node is created. For each new node, the list of beams is created, and all of the beams that are located in the node subspace are added to it. If the number of beams in the beam list of the sub-node is below the predefined threshold, then that sub-node is not processed further. For those nodes that have more beams than the predefined threshold, the divide node procedure is called recursively. Finally, when all of the sub-nodes reach the threshold, the octree is finished, and it contains all of the beams in its leaves.

This recursive algorithm cannot be easily transformed into multi-threading. Let us consider the cause of the problem: suppose that we have a processor that has four cores that can run four threads simultaneously. The algorithm starts with the procedure for the processing of the root node of the octree, which contains all of the beams. This procedure runs in the first thread and creates eight branch nodes, which contain eight subspaces. The simulation then starts new threads, with one thread for the recursive dividing procedure of each branch node. After three threads for the first three branch nodes are started, the simulation runs out of available threads. The thread that is processing the root node will not finish until all of the branch nodes are processed. Three branch node threads cannot finish because they cannot start processing their sub-nodes because there are no threads available. Five remaining branch nodes of the root node also cannot be processed for the same reason. Thus, the program comes to a stall.

If the tree is processed in a depth-first fashion instead, the same problem occurs. The program can process a maximum of three levels in depth. If the depth of the tree is greater than the maximum number of threads, processing will stop.

To solve this problem, we have substituted the recursive octree generation algorithm with an iterative, breadth-first approach. The first, naive version of the iterative algorithm is shown in Figure 2.

```
divide the root node into an array of eight
 subspace nodes
push all nodes in the subspace node queue (SNQ)
for i equals 1 to max number of threads
   pop one subspace node n from SNQ
   start thread i with process_octree_node(
    n, beams list BL)
next i
while the SNQ is not empty and there are working
 threads
   if STQ is not empty and there is available
    thread i
      pop one subspace node n from SNQ
      start thread i with process_octree_node(
       n, beams list BL)
   endif
   if thread i has finished
      stop thread i
   endif
repeat
```

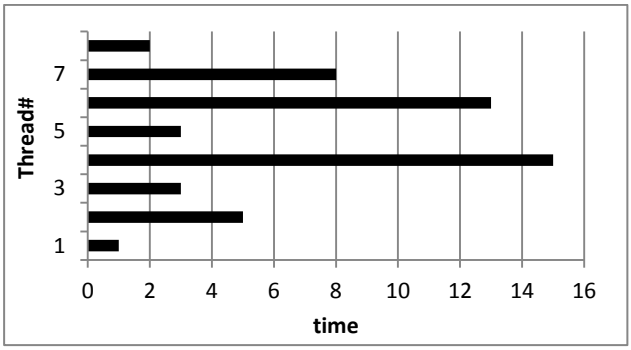
Fig 2 The naive multi-threaded iterative algorithm

In this algorithm, only the first level of the octree subdivision is parallelized. The algorithm in the first loop

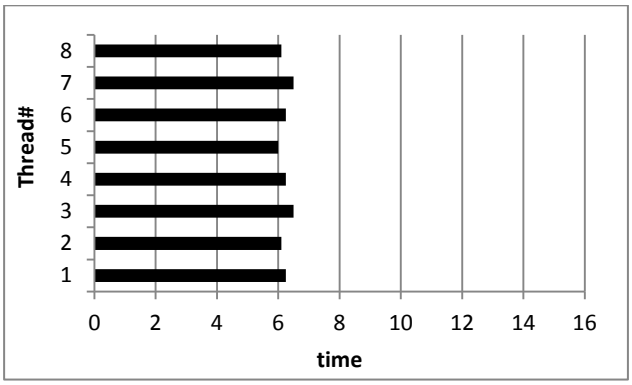
runs *n* threads to work in parallel on the first subdivisions, where *n* is the maximum number of threads. If *n* is less than 8, then some of the subspace nodes will not be processed, and the next loop processes the remaining subspace nodes as some of the threads finish their job. Each subspace node is processed to completion in one call to the *process_octree_node* procedure from Figure 1.

This approach is simple because only the topmost level of the octree is parallelized, but it suffers from load balance problems. If the number of available threads is more than 8, then the excess threads will not be utilized. Additionally, some of the subspace nodes could require much more work than others, which could cause the other threads to be idle while the busiest nodes continue processing (Figure 3a). The measurements confirmed this weakness, which is shown in Figure 8 in section 4.2.

To avoid load balance and performance issues from the naive approach, we decided to use an iterative algorithm that stores newly generated tasks in a shared task queue, hoping to achieve better load balancing and a shorter time for execution (Figure 3b).



a)



b)

Fig 3 The naive (a) and task queue (b) multi-threading octree algorithm

The first step in creating the multi-threaded task queue iterative algorithm was to create the single-threaded iterative version of the algorithm, which is shown in figure 4.

This algorithm starts with the creation of the root node and its accompanying list of contained beams. The root node and the beam list are pushed to the octree node queue (*OTQ*) and the beam list queue (*BLQ*), respectively. These two structures will be used to avoid recursive calls by storing new tasks for the iteration. The algorithm then enters the loop. Inside the loop, one node and its beam list are popped from the *OTQ* and *BLQ*. If the list contains more beams than the threshold, then the popped node and the list are processed with the *process_octree_node* procedure. This procedure returns two arrays: one array contains eight new subnodes *SN[8]*, and the other array contains eight subnode beams lists *NBL[8]*. The nodes and lists from these arrays are then added to the queues *OTQ* and *BLQ*, respectively. This action completes one iteration of the loop. The loop runs as long as *OTQ* and *BLQ* are not empty.

```
create root node rn and push it in octree node
 queue (OTQ)
push beams list of all beams in beam list queue
 (BLQ)
while the OTQ is not empty
   pop one octree node (n) from OTQ
   pop one beam list (bl) from BLQ
   if bl contains more beams then treshold
      process_octree_node( n, bl, SN, NBL )
      push all nodes from SN to OTQ
      push all beam lists form NBL to BLQ
   endif
repeat

process_octree_node(node n, beams list BL, nodes
SN[8], beam lists NBL[8])
   create eight subspace nodes SN[1] .. SN[8]
   for each node number i in SN[1] .. SN[8]
      for each beam b in BL
         if node SN[i] contains b
               push b in NBL[i]
```

Fig 4 The single-threaded iterative algorithm

From this single-threaded iterative algorithm, we derived the multi-threaded iterative algorithm that is shown in Figure 5.

```
for i equals 1 to max number of threads
   create array of arrays of subspace nodes
    AAS[i]
   create array of arrays of node beams lists
    AAB[i]
next i
create root node rn and push it in octree node
 queue (OTQ)
push beams list of all beams in beam list queue
 (BLQ)
while the OTQ is not empty or there are working
 threads
   if OTQ is not empty and there is available
    thread i
      pop one octree node n from OTQ
      pop one beam list bl from BLQ
      if bl contains more beams than treshold
         start thread i with thread_procedure(
          n, bl, AAS[i], AAB[i] )
      endif
   endif
   if thread i has finished
      push all nodes from AAS[i] to OTQ
      push all beam lists form AAB[i] to BLQ
      stop thread i
   endif
repeat
```

```
thread_procedure(node n, beams list BL, nodes
SN[8], beam lists NBL[8])
   create eight subspace nodes SN[1] .. SN[8]
   for each node number i in SN[1] .. SN[8]
      for each beam b in BL
         if node SN[i] contains b
               push b in NBL[i]
```

Fig 5 The multi-threaded iterative algorithm

The first step of the algorithm is to create the two-dimensional array of subspace nodes *AAS* and the two-dimensional array of node beam lists *AAB*. These two data structures are used to store the array of subnodes *SN* and the array of subnode beams *NBL*, which are the result of each thread node's processing procedure. In the single-threaded algorithm, only one pair of arrays for the subnodes and beams lists (*SN[8]* and *NBL[8]*) are sufficient, but in the multi-threaded version, several arrays are required, one for each thread. The capacity (of the second dimension) of these arrays is the maximum number of threads that the simulation can run. These data structures ensure that every thread has its own data structures, which avoids mutex and synchronization problems.

After these arrays are created, the root node is processed in the same manner as in the single-threaded iterative algorithm, following the loop that processes the node runs while there are still unprocessed nodes and while all of the threads are not finished. The first part of the loop checks whether there are nodes that are waiting in the *OTQ* and whether there is a free thread available. If both conditions are met, then one node is popped from the queue *OTQ*, and a single beam list is popped from the *BLQ*. If the number of beams of the node is larger than the threshold, the node is processed. Processing of the node is performed by starting a new worker thread and calling the thread node processing procedure, which performs the same job as the single-threaded procedure.

The second part of the loop checks whether there are any threads that have finished processing. If so, all of the sub-nodes and lists created by this thread are places on the node and beam list queues, and the thread is stopped.

The loop stops when all of the nodes have been processed and when all of the threads have finished their job.

This algorithm cannot run into a deadlock similar to the recursive multi-threaded algorithm because the processing of one node is independent of the processing of the other nodes. Additionally, each thread has a separate data structure; as a result, the synchronization of threads is avoided, and corruption of the data cannot occur.

4. RESULTS

Multi-threaded beam tracing was tested on three different computers, for which the characteristics are displayed in Table 1.

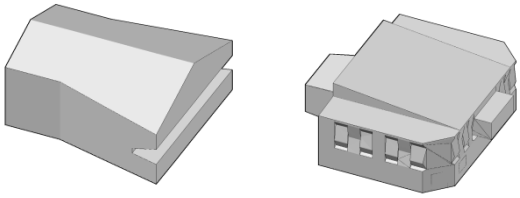
Computer	#Cores	Cache (MB)	Clock (GHz)	RAM (GB)	Total Cores
Intel Core i5 2400	4	6	3,10	8	**4**
2x AMD Opteron 4171 HE (cloud)	6	3	2,10	32	**12 (8)**
2x Intel Xeon E5 2660 (hyper-threading)	8	20	2,20	32	**16 (32)**

Tab 1 Computer platforms used for testing

The first computer was a standard workstation equipped with an Intel Core i5 processor with 4 cores and the Microsoft Windows 7 operating system. The second computer was a server with 2 AMD Opteron processors, each with 6 cores running Microsoft Server 2012. This computer was accessed through a Microsoft Azure cloud service, which allows only 8 logical cores to be available for testing because of virtualization. The third computer was a server with two Intel Xeon 8 core processors and Microsoft Server 2008. This server was accessed directly, which allowed all 16 cores to be available. On this computer, the hyper-threading test (32 logical cores) was performed.

The beam-tracing simulation that was run in the tests was coded in C++ language, using the Microsoft Foundation Class (MFC) Library and the Standard Template Library. Multi-threading was implemented using MFC multi-threading functions. The simulation was built in the Microsoft Visual Studio 2010 development environment, with a 32-bit runtime. To obtain deeper insights into the multi-threading performance, we used the Microsoft Windows Performance Toolkit.

Three models were used to test the simulation. The first model was a shoebox-shaped room that was 24 m long, 10 m wide and 8 m high with 12 triangles; the second model was a simple auditorium that was 30 m long, 30 m wide and 15 m high with 120 triangles (figure 6a); and the third model was a multi-functional theatre that was 20 m long, 21 m wide and 8 meters high, with 894 triangles (figure 6b).



a) b)

Fig 6 Room models used in the tests: a) the simple auditorium (120 triangles), b) the multi-functional theatre (894) triangles

The simulation for each of the three models traced approximately 100.000 beams and calculated the distribution of the sound intensity in the form of a raster with approximately 5.000 points (Figure 7). All of the tests for the same model produced the same distribution,

regardless of the number of threads and computer platforms.

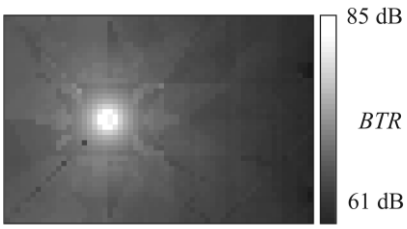


Fig 7 Distribution of sound intensities produced for the simple auditorium model and a single sound source

On the Intel Core i5 workstation with four cores, tests were executed with up to 4 threads. On the AMD Opteron virtual server, tests were executed with up to 8 threads because there was a maximum number of 8 logical cores that were available. On the Intel Xeon server, the tests were executed with a maximum of 16 threads when hyper-threading was turned off and with a maximum of 32 threads when hyper-threading was turned on.

In each test, the execution times were measured for beam tracing, for generation of the octree, and for raster generation. Parallel applications rely on the generation of rasters with the resulting sound intensity distribution. To minimize the influence of the operating system scheduling process on the results, each test was repeated 10 times, and the results were averaged.

4.1 Results of three multi-threading algorithms

This section shows a comparison of the percentage speedup *S* for the two iterative algorithms and one recursive algorithm that comprise the multi-threaded beam-tracing method. The tests were performed on the Intel Xeon Server, and the results are presented in Figure 8. The linear (ideal) speedup is shown with a dotted line for reference.

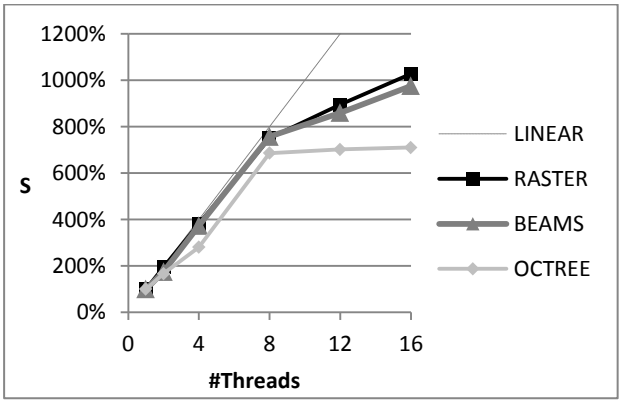


Fig 8 Parallelization percentage speedup on the Intel Xeon server

Two iterative algorithms have almost ideal speedup when the number of threads is below 8. The speedup of the octree algorithm in that part of the graph is also very good, just slightly lower than the ideal. The fact that the octree has a smaller speedup results from the existence of the large part of serial code being located in the beginning of the algorithm, where the root node of the octree is calculated serially.

All three curves show a breaking point when the number of threads reaches 8. This feature arises because up to 8 threads are executed on the same processor, and beyond this point, the threads must be distributed between two processors. When the threads are not executed on the same processor, they cannot optimally use the processor cache, and subsequently the performance drops.

4.2 Results of two octree parallelization approaches

The remaining tests concentrate on the octree parallelization algorithm because it is a novel technique. Let us first consider which of the two parallel octree generation algorithms that are proposed in section 3.2 gives better results. The first algorithm is the naive approach, where only the first subdivision is parallelized, and the second algorithm is full parallelization with the task queue. Figure 9 shows the results of both versions.

This figure clearly shows the load balancing problems of the naive approach, which we predicted in section 3.2. This approach results in a speedup only for two of the threads, while the task queue approach results in a much better speedup. For this reason, we decided to use the task queue approach.

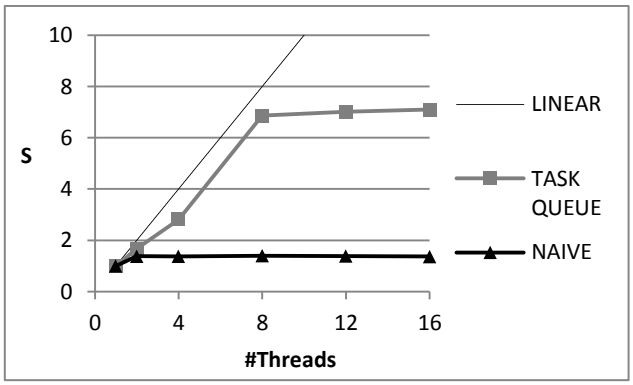


Fig 9 Parallelization percentage speedup for the naive and task queue approaches in the octree generation

Our task queue octree parallelization algorithm shows better performance than the Harihan and Aluru [19] compressed octree parallelization – for 8 parallel tasks it is 700% vs. 400%. On the other hand the speedup of their parallelization scales better – our speedup for 16 threads is almost the same as for 8 threads, and their performance is 166% better. This is because their simulation runs on the cluster of computers, which doesn't show the two-processor breaking point like our multi-core parallelization.

4.3 How does the complexity of the model influence the speedup of the multi-threading parallelization?

With this test, we intended to explore how the multi-threading octree algorithm performance scales with the complexity of the simulated models. Table 2 and Figure 10 show the speedup of the octree algorithm in the case of the shoebox-shaped room (24 triangles), the simple auditorium (120 triangles) and the multi-functional theatre (894 triangles). All of the tests were performed on the Intel Xeon server with up to 16 threads.

	Shoebox Room (24 triangles)		Simple Auditorium (120 triangles)		Multi-Theatre (894 triangles)	
#Threads	time (s)	speedup	time (s)	speedup	time (s)	Speedup
1	4,61	**100%**	12,21	**100%**	16,92	**100%**
2	2,65	**174%**	7,33	**167%**	10,35	**163%**
4	1,57	**294%**	4,35	**281%**	5,87	**288%**
8	0,75	**615%**	1,78	**686%**	2,26	**749%**
12	0,78	**591%**	1,74	**702%**	2,26	**749%**
16	0,8	**576%**	1,72	**710%**	2,22	**762%**

Tab 2 The speedup of the octree algorithm in cases with different model complexity

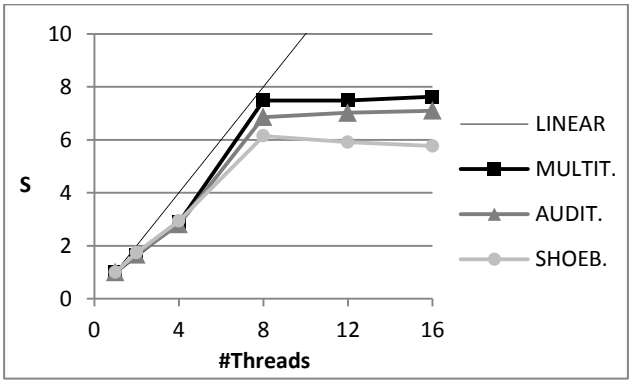


Fig 10 The speedup of the octree algorithm in cases with different model complexity

The speedup for all three models is similar until it reaches the breaking point of 8 threads. The speedup grows almost linearly with the increasing number of threads. After the breaking point, the slope of the speedup growth becomes much smaller.

The speedup is best for the multi-functional theatre, although the multi-functional theatre is the most complex model. The auditorium speedup is somewhat lower, and the shoebox room speedup is the lowest. This finding shows that the algorithm's performance does not decrease with an increase in the complexity of the models.

4.4 A comparison of the results on different computers

This section contains a comparison of the results of the multi-threaded octree generation algorithm using three different platforms: an Intel Core i5 workstation with 4 cores; a server with 2 AMD Opteron processors, each with 6 cores; and a server with 2 Intel Xeon processors, each with 8 cores. The AMD server was running in the cloud and was accessed through a virtualization layer, which allowed only 8 logical cores to be available for testing. The Intel server was tested with and without hyper-threading, with 32 and 16 logical cores, respectively. Table 3 and Figure 11 show the performance and speedup of the octree algorithm on the different platforms.

The best speedup is with the Intel Server; in this case, the speedup grows almost linearly until the 8-thread breakpoint; then, it continues to grow, but much more slowly. When hyper-threading is turned on, a similar speedup is obtained until using 8 threads; then, it falls and remains the same through 32 threads. This finding shows that there is no gain from hyper-threading in the case of such a parallel simulation.

Tab 3 The speedup of the octree algorithm in the case of different computer platforms

	Intel Works. (4 cores)		AMD Server (8 cores)		Intel Server (16 cores)		Intel Server HT (32 cores)	
#Threads	time (s)	speedup	time (s)	speedup	time (s)	Speedup	time (s)	Speedup
1	5,07	**100%**	12,15	**100%**	12,21	**100%**	12,45	**100%**
2	2,78	**182%**	5,95	**204%**	7,33	**167%**	7,08	**176%**
4	1,76	**288%**	4,21	**289%**	4,35	**281%**	4,46	**279%**
8			2,33	**521%**	1,78	**686%**	1,91	**652%**
12					1,74	**702%**	2,02	**615%**
16					1,72	**710%**	2,00	**622%**
32							2,00	**623%**

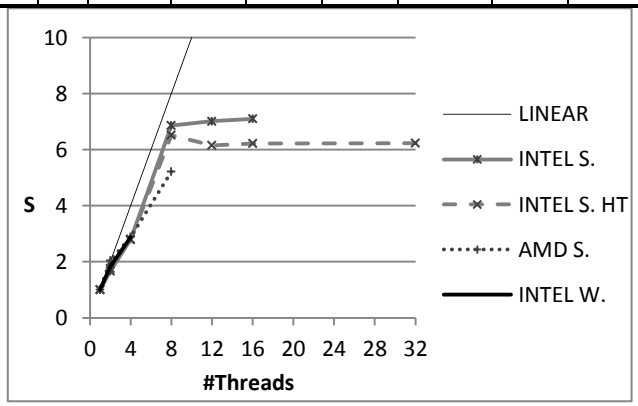


Fig 11 The speedup of the octree algorithm in the case of different computer platforms

The virtual AMD server showed a lower speedup compared with the real Intel server, which can be attributed to the fact that the virtualization presents one more layer between the algorithm and the hardware and also to the fact that this server runs on an AMD Opteron and not on an Intel Xeon processor.

The Intel workstation showed a similar speedup compared to the other platforms. The Intel workstation can run up to 4 threads because of its 4 cores. In this case, it is interesting to consider the absolute values of the measured performance instead of the speedup. The best time for the Intel workstation is 1,76 seconds when running on 4 threads, while the absolute best time was 1,72 seconds, which was scored by the Intel server with 16 threads. Although the clock of the workstation processor is significantly higher than the clock of the server processor, it is notable that the workstation with one processor (priced at approximately 200 US$) had almost the same performance as the server with two processors (priced at more than 1300 US$ each).

5. CONCLUSIONS AND FUTURE WORK

This paper has presented parallelization of the acoustic beam-tracing algorithm. Several types of parallelization were considered, and the thread-level parallelization was chosen as the most suitable implementation. The authors have used the symmetrical master thread-working thread model, with a shared task queue and local result queues, to avoid the need for the synchronization of threads. In addition to the already known parallelization of iterative algorithms, this paper presents the novel parallelization method of the recursive octree generation algorithm.

The multi-threaded beam-tracing simulation was tested on three different multicore computer platforms: a workstation (4 cores), a virtual server (8 cores) and a regular server (16 cores). The parallel octree generation algorithm showed the best speedup on the regular server. The speedup was excellent until 8 threads but dropped significantly afterwards because, at that point, excess threads had to be executed on the second processor and the cache was not optimally used. Hyper-threading did not cause a speedup when the number of threads was greater than the number of logical cores. The virtual server showed a somewhat lower speedup because of the additional virtualization layer. The workstation had the smallest number of cores and, accordingly, the smallest relative speedup, but it showed excellent absolute performance that compared favorably with the regular server (even though the workstation processor was more than ten times less expensive than the server processors).

We also tested the algorithm on models with different complexity. The performance did not drop when the complexity of the models increased; in contrast, the most complex model had the best speedup. Regarding the two versions of parallel octree generation algorithms, the task queue version performed substantially better than the naïve approach.

Finally, we compared the speedup of our novel octree generation algorithm with the speedup of two already known iterative algorithms. All three of the algorithms showed a significant, almost linear speedup until the 8 threads were used and lower speedup later. As expected, the octree algorithm speedup was the lowest, especially after the breaking point at 8 threads, which was the result of the larger portion of serial code.

This paper has shown that our novel octree generation algorithm, which has a multi-threading iterative form, has a significant speedup, especially when executed on a single processor. On the 4-core workstation processor, the algorithm worked 288% faster, and on the 8-core server processor, it provided a speedup of 686%. The algorithm is robust with respect to the complexity of the model.

To obtain better results, generation of the root octree node, which is now performed using a serial approach, will also have to be parallelized.

REFERENCES

[1] Taylor M, Chandak A, Antani L, Manocha D (2009) RESound: Interactive Sound Rendering for Dynamic Virtual Environments. Proc 17th Intern ACM Conf Multimed, 19.-24. 10. 2009. Beijing China

[2] Savoia L, Manocha D, Lin MC (2010) Use of GPUs in room acoustic modeling and auralization. Proc Intern Symp Room Acoust, 29.-31. 08. 2010. Melbourne Australia

[3] Noisternig M et al (2008) Framework for Real-Time Auralization in Architectural Acoustics. Acta Acust united with Acust, Vol. 94, pp. 1000 – 1015

[4] Lentz T et al (2007) Virtual Reality System with Integrated Sound Field Simulation and Reproduction. EURASIP J Adv Signal Proc, Vol. 2007, Article ID 70540

[5] Taylor M et al (2012) Guided Multiview Ray Tracing for Fast Auralization. IEEE Trans Vis Comput Graphics, Vol. 12, No. 11, pp. 1797-1810

[6] Laine S et al (2009) Accelerated beam tracing algorithm. Appl Acoust, Vol. 70, No. 1, pp. 172 – 181

[7] James A, Dalenback BI, Naqvi A (2008) Computer Modelling With CATT Acoustics - Theory and Practise of Diffuse Reflection and Array Modeling. Proc 24th Reprod Sound Conf, 20.-21.11.2008. Brighton UK

[8] Feistel S et al (2007) Improved methods for calculating room impulse response with EASE 4.2 AURA, Proc 19th Intern Congr Acous, 2.-7. September 2007. Madrid Spain

[9] Sikora M, Mateljan I, Bogunović N (2012) Beam Tracing with Refraction. Arch Acous,Vol. 37, No. 3, pp. 301-316

[10] Danowitz A et al (2012) CPU DB: Recording Microprocessor History. Commun ACM, Vol. 55, No. 4, pp. 55 – 70

[11] Bigler J, Stephens A, Parker SG (2006) Design for Parallel Interactive Ray Tracing Systems. Tech Rep, UUSCI-2006-027, SCI Institute, University of Utah

[12] Nunes M, Santos LP (2009) Workload Distribution for Ray Tracing in Multi-Core Systems. Proc. 17º Encontro Port. Comp. Graf., 29.30. 10. 2009. Covilha Portugal

[13] Jedrzejewski M, Marasek K (2006) Computation of room acoustics using programable video hardware. Comput Imag Vis, Vol. 32, pp. 587 - 592

[14] Amdahl GM (1967) Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. Proc Am Fed Inf Proc Soc Conf, pp. 483 – 485

[15] Spjut J, Kopta D, Brunvald E, Boulos S, Kellis S (2008) TRaX: A MultiThreaded Architecture for RealTime Ray Tracing. Proc Symp App Specif Proces, 08.-09.06. 2008 Anaheim USA

[16] Chandak A et al (2009) FastV: From-point Visibility Culling on Complex Models. Comp. Graphics Forum, Vol. 28, No. 4, pp. 1237-1246

[17] Gao L et al (2009) Exploiting Speculative TLP in Recursive Programs by Dynamic Thread Prediction. Proc 18th Int Conf Compil Constr, pp. 78 - 93

[18] Pombo JJ, Aldegunde M, Garcia-Loureiro AJ (2006) Optimization of an Octree-based 3-D Parallel Meshing Algorithm for the Simulation of Small-Feature Semiconductor Devices. Parallel Comp, Vol. 33, pp. 439-446

[19] Tu T, O'Hallaron DR, Ghattas O (2005) Scalable Parallel Octree Meshing for Terascale Applications. Proc 2005 ACM/IEEE Conf Supercomput, 12.-18.11. 2005. Seattle USA

[20] Hariharan B, Aluru S (2005) Efficient Parallel Algorithms and Software for Compressed Octrees with Applications to Hierarchical Methods. J Parallel Comp, Vol 31, No. 3+4, pp. 311 - 331