MULTITHREADED BEAM-TRACING

Marjan Sikora, Ivo Mateljan
Faculty of electrical engineering, R.Boskovica 32, 21000 Split, Croatia (sikora@fesb.hr)

Abstract: In the last years the computational power of modern processors is increasing mainly due to the increase in the number of processor cores. Computationally intensive applications can gain from this trend only if they employ the parallelism such as the multithreading programming. Geometric simulations can employ multithreading because the main part of geometric simulations can be divided into subset of mutually independent tasks. A successful example of such parallelization is the interactive ray tracing visualization that has been recently developed. The acoustic beam tracing can also benefit from the multithreading because it is an intensive computing task, and because it traces large number of beams that are independent of each other. This paper presents the parallelization of the existing beam tracing with the refraction algorithm.

Key words: acoustic simulation, beam tracing, multithreading

1. INTRODUCTION

Acoustic simulations are computationally very intensive applications. Recent interactive auralizations compute the impulse response for the moving source and listener and some for the changing geometry [1, 2, 3, 4]. Then they perform the auralization by convolution of the impulse response and the anechoic acoustic signal. The other type of acoustic simulation computes the spatial distribution of the sound pressure, reverberation time, speech transmission index and other, while including ever increasing number of sound wave effects. In ray tracing and hybrid simulations, diffuse reflections and the diffraction are now standardly incorporated [5, 6] and recently, beam tracing simulations are developed with included refraction simulation [7].

The increase in computing demands of acoustic simulations has to be met by the increase of the computing power of the system that performs the simulation. In last years the computational power of modern processors is not increasing anymore because of the increase in the processor clock frequency [9], rather new processors have greater number of cores. To use the increase in the power of modern computers, one has to resort to the parallelization. There are three ways that modern acoustic simulation can be parallelized:

(1) computing on the clusters or grids computers or processors [13],
(2) transferring parts of the simulation to the graphical processing unit (GPU) with large number of simple parallel cores [2],
(3) converting the single threaded simulation to multithreaded one that can execute on the modern multicore single central processor unit (CPU).

In some implementations the hybrid GPU+CPU parallelization is used [10]. Transforming an acoustic simulation to run on the cluster or the grid of computers would provide the enormous processing power, but requires fundamental change in the code of the simulation, even changing the type of programming language used for the simulation. Also such simulation would spend much time on the communication between cluster or grid nodes, in order to synchronize the work, because communication is done externally, over network, which is much slower than the internal CPU or GPU communication.

In recent time simulation that run on the GPU [2, 11] are a common thing, because modern GPU-s are massive parallel processors. They provide affordable, yet very powerful platform for computationally intensive task. The drawbacks of this way of parallelization is that code requires fundamental changes in order to adapt it to work with CUDA or OpenMP programming interfaces that open GPU to a programmer. Also the processor cores of the GPU are limited in the number of registers and operations compared to CPU processor cores, which
imposes limitations to the programming methods used in a simulation, such as recursion.

The authors have recently developed the beam tracing method with refraction (BTR). This method [7] is very processing intensive because it is based on the geometrically complex beam tracing, and because it calculates not only the reflection but also the refraction of the sound. This motivated the authors to parallelize the simulation in order to speed up the processing. When considering which approach to use, authors decided to use the third approach – CPU multithreading.

2. PREVIOUS WORK

The multithreading exploits the feature of the operating system to execute several threads at the same time. This feature was originally developed to enable different programs to run concurrently on the single core processor, using the time sharing. But multithreading soon evolved and was used for parallel execution of a single program on the multicore processor. A single program starts several working threads, with each thread executing the same program code. All threads within a program see a single shared address space. The workload distribution is performed by one master thread that starts, stops and synchronizes working threads. If different working threads that run simultaneously write and read to same data structures, the synchronization mechanism has to be established to ensure that data is not corrupted because it was simultaneously accessed by different threads.

The operating system decides which thread would be run on which processor core. If there are more threads than processor has cores, the operating system uses time sharing, and switches the execution of several threads on the same core. So the parallelization of the simulation would result with speedup only in the case where there is at least the same number of cores as there are threads that run in parallel.

Geometric simulations can employ multithreading because the main part of the geometric simulation can be divided into subset of mutually independent tasks. A successful example of such parallelization is interactive ray tracing visualization and auralization, that have been recently developed [12, 10].

The most difficult part of multithreading is the synchronization of threads. The most efficient and robust multithreaded application is one where threads can work independently of each other. Geometric simulation, such as the ray tracing method tracks the propagation of rays through the simulated model. Rays don’t depend on each other, but only on the model geometry. So the ray tracing algorithm is a highly parallelizable one [1, 12]. In geometric simulations, synchronization between working threads, which allows one thread to change the global state or change the same data, is minimal or even nonexistent [13].

Different implementations of multithreading employ threads in different fashion. Some simulations have fixed number of threads dedicated to specific tasks. Taylor et al. [1] in RESound use one thread to calculate first order specular reflections, seven threads to calculate further three orders of specular reflections and two orders of diffraction, and finally seven threads to calculate diffuse reflections. Other simulations, such as Manta designed by Bigler et al. [13], use dynamically load balanced multithreading. In this method each thread gets tasks dynamically, depending on the current state of the simulation. Such method often employs threads to process ray packets rather than individual rays to decrease the overhead.

Geometric simulations often use spatial data structures such as Binary Space Partition (BSP) trees and k-dimensional (kd) trees [13] to speed up geometric operations. The creation and the processing of such structures employ recursive algorithms. Multithreading is easily employed on iterative algorithms, but adapting a recursive one to perform in a multithreaded manner poses a big challenge.

The complexity of multithreading (parallel algorithm) and its efficiency is governed by the Amdahl’s law [14]. Amdahl law is shown as equation (1)

\[ S = \frac{1}{r_s + \frac{r_p}{n}} \]

where \( S \) is the speedup gained by parallelization, \( r_s \) is the percent of serial code, \( r_p \) is the percent of parallel code and \( n \) is the number of threads. The sum of \( r_s \) and \( r_p \) must equal 1. In any parallel algorithm there is always a part of code that can’t be parallelized. The smaller the amount of serial part of the code the efficient parallelization on the same number of threads would perform. The speedup calculated with Amdahl’s law is displayed in Fig. 1.

![Fig.1. Amdahl’s law](image)

Fig. 1 shows that algorithms with larger percent of parallel code better scale up with the increase in the number of threads. So, the main issue in designing an efficient multithreaded algorithm is to maximize the parallel part of the code and minimize the serial overhead. Overhead includes thread communication and synchronization, thread idle times due to sub-optimal load balancing and redundant computations.
Workload distribution in multithreaded algorithms is mainly done using some shared data structure. Individual working threads access this data structure to retrieve data. In ray tracing processing results in set of new, secondary rays created either by specular or diffuse reflections or by diffraction and refraction. The secondary rays are stored on the shared data structure. Since all threads can read and write on these shared data structure, a data access control mechanism is needed in order to avoid corruption and conflicts. The traditional mechanism that is used is when access to the shared memory is controlled by mutual exclusion, which ensures the integrity of data structure. However this approach complicates the algorithm and increases the overhead because threads wait until mutual exclusion locks are in place, which occurs often when the number of threads increases. Our approach uses lock-free conservative local mechanism, where there is an independent local work queue and local data structure for each thread. By using this approach we avoid any mutual exclusion, and synchronization issues. The research community has proven that this mechanism is the fastest one [10].

3. MODELS AND METHODS

This paper presents the multithreaded version of the BTR. The acoustic beam tracing can benefit from the multithreading programming because it is a computing intensive task, and it traces large number of beams that are independent of each other. In this aspect it is similar to ray tracing, and multithreaded ray tracing algorithms are already well described in literature. However, the BTR is much more computer intensive than ray tracing, since the geometric operations are performed on beams which are three-dimensional structures, instead on rays, which are one-dimensional structures. Furthermore, the BTR calculates not only the reflection of sound but also the refraction of sound. Because of this, the BTR has bigger complexity than the ray tracing, which makes it even more suitable for multithreading.

The BTR consist of four main parts [7]:

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

The first phase converts the geometry of the simulated model from the triangle mesh to the BSP tree, in order to speed up geometrical operations of the beam tracing. After this the beam tracing is performed. It is initiated by the creation of 20 initial beams using an icosahedron with center in the place of the sound source. Each of 20 initial beams is then processed: it is divided according to the geometry of the surface it hits, and reflected and refracted beams are then created from divided incoming beams.

This is repeated until one of two termination criteria is fulfilled:
- the order of reflection is greater than the maximum order of reflection
- the sound intensity of the beam drops below the intensity threshold.

In the third phase all beams resulting from the beam tracing are put in the octree. This is done to speed up the spatial search of beams performed in the last phase. In the last, fourth phase, the spatial distribution of levels of sound intensity is created in the form of a raster of points. For each point the octree is filtered to get 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 place of the point.

The first and the third phase are recursive algorithms, and the second and the fourth are iterative ones. In the rest of this section we will describe how iterative algorithms were converted to the multithreaded form, and discuss the method for converting recursive algorithms.

3.1. Multithreaded beam tracing

The single threaded algorithm of the second phase is shown in Fig.2.

```plaintext
create initial raw beams
put initial raw beams in initial beams queue (IBQ)
while IBQ is not empty
    pop one initial beam (ib) from IBQ
    do beam tracing of ib
    push finished beams created from ib in finished beams list (FBL)
repeat
```

Fig.2. The single threaded algorithm of the beam tracing.

First 20 initial beams are created and put in the initial beams queue (IBQ), which is a first-in-first-out (FIFO) structure. Then algorithm enters the loop that runs while the IBQ is not empty. In each iteration of the loop one initial beam from the IBQ is traced, resulting in reflected and refracted beams. Tracing is done until the termination criteria are met. Finished beams produced with tracing of one initial beam are put in the finished beams list (FBL).

The multithreaded algorithm of the second phase is shown in Fig. 3 and Fig. 4.

The first procedure is the control procedure that distributes the work. The second procedure is the thread procedure containing with processing that occurs inside one thread.

The multithreaded beam tracing algorithm begins with creation of 20 initial beams. All threads are then started, and to each thread one initial beam is passed for processing (creating threads part of Fig. 4.).

The central part of the algorithm is a loop in which program restarts the thread which has finished its task (running threads part of Fig.4.). When one thread finishes processing, it is restarted with a new initial beam to process. This loop repeats until there are no initial beams to process.
Then final loop waits for all threads to finish processing, and then terminates the threads (terminating threads part of Fig.4). For each terminated thread, finished beams of that thread are moved to the common finished beams list (FBL).

**multithreaded_beam_tracing_procedure**
- create initial beams
- put initial beams in initial beams queue (IBQ)
- for $i$ equals 1 to $n$
  - create thread $i$
  - pop initial beam $ib$ from IBQ
  - start thread_procedure(ib) for thread $i$
- next $i$
- while IBQ is not empty
  - wait until one thread $x$ is finished
  - pop initial beam $ib$ from IBQ
  - start thread_procedure($ib$) for thread $x$
- repeat
- while there are still threads working
  - wait until one thread $x$ is finished
  - move all beams from FBL$i$ to finished beams list (FBL)
- terminate thread $x$
- repeat
- thread_procedure(initial beam $ib$, thread $i$)
  - do beam tracing of $ib$
  - push finished beams created from $rb$ in finished beams list for thread $i$ (FBLi)

**Fig.3.** The multi-threaded algorithm of the beam tracing.

The thread procedure first traces an initial beam. This results in the set of finished beams. These finished beams are then added to the thread’s local finished beams data structure (FBL$i$). The fact that all threads have their own data structures ensures that threads don’t have to use mutual exclusion principle to access the universal data structure. Instead, every thread during its lifespan uses its own data structure. In the end of the algorithm all finished beams from the thread’s local lists (FBL$i$) are moved to the final finished beam list (FBL), which contains all beams.

**Fig.4.** Scheme of the multithreading algorithm of the beam tracing.

### 3.2. Multithreaded generation results

The fourth phase, that generates results, was converted to the multithreaded version in the same manner as the beam tracing, since it is also an iterative algorithm. Fig. 5. presents the single threaded version of the generation of results.

```plaintext
create raster points
put raster points in raster points queue (RPQ)
while RPQ is not empty
  pop one raster point (rp) from RPQ
  calculate level of intensity (LI) for rp
  push LI in raster of level intensities (RLI)
repeat
```

**Fig.5.** The single threaded algorithm for generation of results.

The multithreaded algorithm of the fourth phase is shown in Fig. 6. The first part presents the control procedure that distributes the work, and the second part presents the thread procedure with processing that occurs inside one thread.

```plaintext
create raster points
put raster points in raster points queue (RPQ)
for $i$ equals 1 to $n$
  create thread $i$
  pop one raster point (rp) from RPQ
  start thread_procedure(rp) for thread $i$
next $i$
while the RPQ is not empty
  wait until one thread ($x$) is finished
  pop one raster point (rp) from RPQ
  start thread_procedure(rp) for thread $x$
repeat
while there are still threads working
  wait until one thread ($x$) is finished
  move all intensity levels RLI$x$ to raster of level intensities (RLI)
  terminate thread $x$
repeat

thread_procedure(raster point rp, thread $i$)
  calculate level of intensity (LI) for rp
  push LI in raster of levels of intensity for thread $i$ (RLI$i$)
```

**Fig.6.** The multithreaded algorithm of the generation of results.

As in Fig.4., each thread has its own data structure for storing calculated intensity levels. This enables the multithreading without the need for thread synchronization.

### 3.3. Recursive multithreading

The first and the third phase of the BTR use recursive algorithms. Both phases result in $kd$ spatial data structure which speeds up the subsequent processing: the first
phase calculates the binary tree and the third phase calculates the octal tree.

The problem occurs when these two recursive algorithms are transformed to the multithreaded version – it is the same problem with all recursive algorithms. Let us consider the cause of the problem: suppose that we have a four-core processor so we can start four threads simultaneously. Simulation enters the procedure for dividing the root node of octree with the whole model space. This procedure runs in the first thread. Then we create eight branch nodes, with eight subspaces and start running new threads for the dividing procedure of each branch node. After we start three threads for the first three branch nodes, we run out of threads. The thread with the root node won’t finish until all the branch nodes are processed. Three threads of three branch nodes of the root node can’t finish because they can’t even start processing their subspaces since there are no threads available. Three remaining branch nodes of the root node also can’t start processing because of the lack of the free threads. So the program comes to a stall.

If we process the tree in depth and not in width as is the case above, the problems stays the same. In the case of in-depth processing, the program can process maximum of three levels in depth, and after that the processing is blocked.

The solution to this problem is to transform the recursive algorithm to the iterative one. The conversion of phases one and three from the recursion to the iteration, and its transformation to the multithreaded version remains for the future work.

4. RESULTS

The multithreaded beam tracing was tested on three different processors, with characteristics displayed in Table 1.

<table>
<thead>
<tr>
<th>Processor</th>
<th>#Cores</th>
<th>L2 Cache (MB)</th>
<th>Clock (GHz)</th>
<th>RAM (GB)</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i5 (2400) Sandy Bridge</td>
<td>4</td>
<td>6</td>
<td>3.10</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>Intel Core 2 Duo (E8400) Woldale</td>
<td>2</td>
<td>6</td>
<td>3.00</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Intel Core 2 (T5600) Merom</td>
<td>2</td>
<td>2</td>
<td>1.86</td>
<td>2</td>
<td>XP</td>
</tr>
</tbody>
</table>

Table 1. Computer platforms used for testing.

Core i5 and Core 2 Duo processors had the same amount of L2 cache, similar clock frequency, and the only thing different was the number of cores. Core 2 processor had the same number of cores as Core 2 Duo, but significantly lower clock frequency and smaller cache.

The BTR simulation that was run in tests was coded in C++ language, using Microsoft Foundation Class library and Standard Template Library. It was compiled with Microsoft Visual Studio 2010 development environment, to the 32-bit runtime.

The model used for testing simulation was a simple auditorium 30 m long, 30 m wide and 15 m high. The model has two parts: a stage with parallel walls and a trapezoidal auditorium area with a balcony (Fig. 7.). The source was positioned in the middle of the front end of the stage, 1 m above the floor. The scene geometry was composed of 40 triangles.

Fig.7. The model used in tests.

The simulation calculated the distribution of the sound intensity in the form of a raster with 2400 points. Raster was a rectangular one, and it was calculated for the height 1m above the floor. The distribution of the intensity of the sound is presented in Fig. 8. All tests produced the same raster, regardless of the number of threads.

Fig.8. The raster of sound intensities produced in tests.

For Core i5 processor that has four cores, tests were executed with 1, 2, 4 and 8 threads respectively. For core 2 Duo and Core Duo processors, that have two cores, tests were executed with 1, 2 and 4 threads. The reason that for each processor one test was performed with greater number of threads than number of cores, was to confirm that the multithreading doesn’t bring speedup in the case when there are more threads than available cores.

In each test execution times were measured for the beam tracing and for the raster generation. Each test was repeated 10 times and results were averaged. Test results are presented in Table 2. and Fig. 9.

The results show that in all cases the multithreading results with speedup. The speedup scales well with the increased number of threads. In the case of Core i5 and Core 2 Duo, for two threads, the speedup is almost the same, and for the Core 2 it is slightly smaller, probably due to the smaller amount of cache. The results for the case where number of threads is greater than the number of cores are displayed in grayed cells. One can see that
there is no speedup in such case compared to the case where number of threads is same as the number of cores.

<table>
<thead>
<tr>
<th>Phase</th>
<th>#Threads</th>
<th>Core i5</th>
<th>Core 2 Duo</th>
<th>Core 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beam Tracing</td>
<td>1</td>
<td>0.82</td>
<td>1.31</td>
<td>2.70</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.46</td>
<td>0.74</td>
<td>1.73</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0.28</td>
<td>0.74</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.29</td>
<td>283%</td>
<td></td>
</tr>
<tr>
<td>Raster Generation</td>
<td>1</td>
<td>6.77</td>
<td>10.86</td>
<td>18.28</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3.42</td>
<td>5.76</td>
<td>12.04</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1.81</td>
<td>5.70</td>
<td>12.03</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>1.73</td>
<td>391%</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.** Results of testing.

Let us now compare the speedup of the beam tracing and the raster generation algorithm. The Table 3. and Fig.10. show the speedup of two algorithms in the case of Core i5 processor, compared with the ideal speedup when all processing is done in parallel.

The beam tracing algorithm speedup is smaller than the speedup of the raster generation — 177% vs. 198% (two threads) and 297% vs. 374% (four threads). This means that serial part of the raster generation algorithm is smaller, and thus the parallelization of this algorithm is better.

**Table 3.** The speedup of two algorithms compared to the ideal speedup of the multithreading.

Let us now compare the speedup of the beam tracing and the raster generation algorithm. The Table 3. and Fig.10. show the speedup of two algorithms in the case of Core i5 processor, compared with the ideal speedup when all processing is done in parallel.

The beam tracing algorithm speedup is smaller than the speedup of the raster generation — 177% vs. 198% (two threads) and 297% vs. 374% (four threads). This means that serial part of the raster generation algorithm is smaller, and thus the parallelization of this algorithm is better.

**Table 3.** The speedup of two algorithms compared to the ideal speedup of the multithreading.

The results show that the ratio between the serial and the parallel portion of the code is different for two algorithms. In order to determine the serial and the parallel part of algorithm code the simulation of speedup was done using Amdahl’s law (eq. 1). The different values of r_s and r_p were entered into the equation until the speedup for different number of threads closely matched the experimental results.

**Table 4.** The comparison of speedup obtained by the testing and by the Amdahl’s law.

Table 4. shows the result of the simulation. According to this simulation the serial part of the beam tracing algorithm is 12% and for the raster generation algorithm the serial part is 2%. This means that the raster generation algorithm is almost ideally parallelized, and that in the case of the beam tracing algorithm there is significant part
of serial code. The cause for this is the initial part of the multithreading algorithms, which is more processing intensive in the beam generation. The generation of initial beams (Fig. 3.) is more complex than the generation of raster points, and the result is lower speedup. To get better results of the parallelization of the beam tracing, this part of the beam tracing algorithm should also be parallelized.

5. CONCLUSION AND FUTURE WORK

This paper presented the parallelization of the beam tracing algorithm. Several types of parallelization were considered and the multithreading was chosen as the most suitable one. The iterative part of the BTR simulation was converted to multithreading. This paper presents the detailed description of the old single threaded and new multithreaded algorithms, as well as data structures used in them. The multithreaded BTR was tested on three different multicore computer platforms, and results show that there is significant speedup in the case of increased number of threads. Tests also confirm that if the number of threads is raised above the number of processor cores, multithreading results with no speedup. Two iterative algorithms that were converted to multithreading show different speedup, due to the different amount of the serial code. For the future work, two recursive algorithms would be also converted to multithreading, after they were changed to the iterative form.

REFERENCES