Performance Evaluation of Monte Carlo Based Ray Tracer

Ayobami Ephraim Adewale
University of Tartu
Tartu, Estonia
adewale@ut.ee

ABSTRACT

The main objective of computer graphics is to effectively depict an image in a virtual scene in its realistic form within a reasonable amount of time. This paper discusses two different ray tracing techniques and the performance evaluation of the serial and parallel implementation of ray tracing, which in its serial form is known to be computational intensive and costly for previous computers. The parallel implementation was achieved using OpenMP with C++, and the maximum speedup was ten times that of the serial implementation. The experiment in this paper can be used to teach high-performance computing students the benefits of multi-threading in computationally intensive algorithms and the benefits of parallel programming.

KEYWORDS

ray tracing, Monte Carlo, Open Multi-Processing (OpenMP), high performance computing, algorithm, cluster, parallel programming

1 INTRODUCTION

Light tracing is an important aspect of computer graphics that has over the years been adopted to simulate the real life behavior of illuminance on an object, environment, or scene in different areas such as animations, games, and image rendering. The computation of this illumination is being done by computer programs which calculate illuminance on a particular scene relatively precisely [10]. Due to its importance, there have been different techniques adopted for this purpose, but the two most popular are rasterization and ray tracing [3].

In rasterization, an image in a vector format is taken and converted into a pixelated image known as a raster image for output on a video display or static environment. In ray tracing, an image is rendered by tracing the trajectories of light rays projected through pixels in a view plane [6]. The main differences between these two techniques is the way an image is rendered and the time it takes to render an image. Rasterization has been the most popular of the two, because it is faster and balances the performance needed with the ability to create acceptable images. Regardless of its advantage, the application of rasterization is limited in computer graphics when a photo-realistic image is needed. This is due to its poor handling of light reflection when rendering 2D and 3D images.

© 2021 Journal of Computational Science Education
https://doi.org/10.22369/issn.2153-4136/12/1/4

Rendering an image with ray tracing is slow when compared to rasterization. However, it is able to render a more photo-realistic image, due to its good handling of shadows, reflections, and blurs. With the arrival of parallel programming and GPUs, significant performance gains have been achieved when rendering with the ray tracing technique. The ray tracing technique is divided into two types: traditional ray tracing and distributed ray tracing. Distributed ray tracing tries to extend the traditional ray tracing technique by sampling more rays than the number of pixels in the image. An example of traditional ray tracing is the Turner Whitted technique, while the Monte Carlo technique is an example of distributed ray tracing [11].

With recent advances in computing power, various researchers have picked up an interest in ray tracing. The main objective is to find different ways of reducing time to solution while retaining the property of creating realistic images. The use of parallel frameworks such as the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP) have been proposed by researchers over the years. In this paper, a parallel implementation and a serial implementation of a ray tracing algorithm were studied based on rendering performance. The serial implementation was derived from a parallel implementation that was developed using C++ and made parallel using OpenMP [1]. The performance evaluation was carried out on two high performance computing (HPC) clusters provided by University of Tartu High Performance Computing Center [9].

The paper is organized as follows. Section 2 introduces the concept of ray tracing and OpenMP. Section 3 discusses different state-of-the-art methods. Section 4 discusses the parallel implementation with OpenMP in [1]. In Section 5, the result of the empirical analysis is presented and discussed. Finally, in Section 6, the conclusion and reflection are discussed.

2 CONCEPTS

2.1 Ray Tracing

Ray tracing takes an image from a 3D scene by tracing the trajectories of light rays through pixels in a view plane. Light tracing in ray tracing can be done in two ways: forward tracing and backward tracing. In forward tracing, rays are traced from eyes to the light source, while in backward tracing, rays are traced from the light source to the eyes. This method is compute intensive, because each ray emitted by the light source has to be traced to the eyes, even those that were not able to reach the eyes.

A typical ray tracing environment is made up of eyes or a camera, a scene, object(s), and a light source. Figure 1 describes a simple ray tracing scenario. A ray from the eyes is projected in a straight line for each pixel of the view plane into the scene. Ray intersection is checked to see if the ray intersects with any object in the scene.

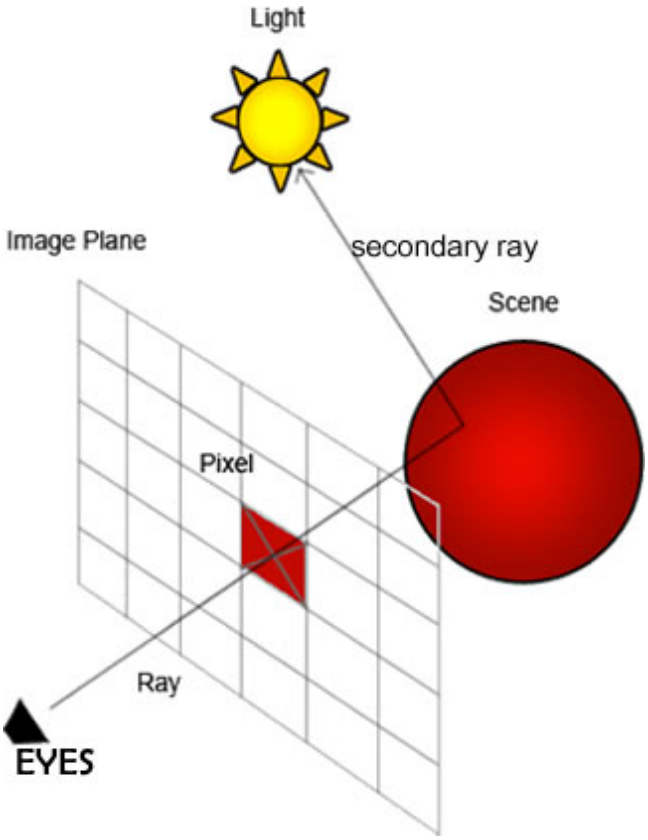

Figure 1: Ray Tracing Scenario.

This first ray is referred to as the primary ray, and its purpose is to discover the objects in the scene. There are three intersection possibilities for this ray, and Figure 2 depicts a simple scenario.

- No Intersection.
- One Intersection.
- Two Intersections.

The third case only happens if the object in the scene is opaque. If an intersection exists, a second ray referred to as the secondary ray is sent from the point of intersection to the light source. If this secondary ray that was emitted is blocked by an object in the scene, the color of the object is projected as a shadow, and if it successfully reaches the light source, that pixel is lit up. The lighting of the pixel is determined by this secondary ray. If there are two points of intersection, the distances of the two points to the eyes are calculated, and the closest point is selected. In ray tracing, the secondary ray is divided into three: shadow ray, reflection ray and refractive ray. These rays make it possible for the ray tracing technique to create a more realistic rendering.

Algorithm 1 is a simple ray tracer algorithm as described by Turner Whitted. In the algorithm, each ray cast through the pixel in the plane can be represented as a line that has an origin represented with O and a direction represented with l. At any time, a point on the line or ray can be represented with

$$Point = O + (l * d) \tag{1}$$

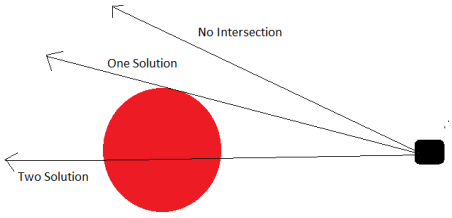

Figure 2: Ray Tracing Intersection Possibilities.

```
for each pixel in the viewing plane do
    for each object in the scene do
        if ray intersects an object in the scene then
            select min(d1,d2);
            recursively ray trace the reflection and refraction
              rays;
            calculate color;
        end
    end
end
```

Algorithm 1: Ray Tracing Algorithm

where d is the distance of the point from the ray origin. Having declared the origin and direction of the ray and also the location of the sphere, we proceed to know if a ray projected from the eye through a pixel in the view plane intersects with any sphere in our scene at any points. say, P_0, and P_1.

To do this, we need to solve for dc:

$$dc = C_0 \quad . \quad l; \quad where \quad C_0 = C - O. \tag{2}$$

dc is the distance of the projected ray from the center of the sphere to the ray origin, C is the center of the sphere, C_0 is the distance from the center of the sphere to the ray origin and . denotes a dot product. If dc is less than zero, then it can be assumed that there is no intersection, but if it is greater than zero, we proceed to the second step, which is to calculate the distance from the center of the sphere to the projected ray. This can be calculated using Pythagoras' Theorem, because we have now formed a right-angled triangle.

$$D = \sqrt{dc^2 - (C_0)^2} \tag{3}$$

If the value of D is greater than the radius of the sphere, then it means the ray does not intersect any point of the sphere, and we can move on to the next sphere. If the value of D is not greater than the radius, then we continue by looking for the points of intersection on the sphere. The point of intersection is calculated as:

$$P = O + (l * d_p) \tag{4}$$

where d_p is the distance from P to O, and it is calculated using:

$$d_p = dc - tc \tag{5}$$

where tc can be obtained from the second right-angled triangle in our sphere using Equation 6,

$$tc = \sqrt{r^2 - D^2} \tag{6}$$

where r is the radius of the sphere. If the ray intersects at two points, the distance of the second point represented as d_{p2} is calculated as $d_{p2} = dc + tc$.

Algorithm 2 describes the ray intersection algorithm.

FUNCTION: $boolean\ intersection(Ray * R,$
$Sphere * S, float * d_{p1}, float * d_{p2})$
$float\ C_0 = S \longrightarrow center - R \longrightarrow Ray\_Origin$
$floatdc = dot(C_0, R \longrightarrow direction)$
if $dc \leq 0.0$ **then**
  return $false$
else
  $float\ D = \sqrt{dc^2 - (C_0)^2}$
  if $D \geq S \longrightarrow radius$ **then**
    return $false$
  end if
  $float\ tc = \sqrt{(S \longrightarrow radius)^2 - D^2}$
  $d_p = dc - tc$
  $d_{p2} = dc + tc$
  return $true$;
end if

Algorithm 2: Ray Intersection Algorithm.

Merging Algorithms 1 and 2, we can derive the time complexity of a simple ray tracing algorithm as:

$$O(pnd) \tag{7}$$

where p is the number of pixels, which can be represented as:

$$pixels = width * height \tag{8}$$

and n is the number of objects in the scene. d is the time complexity of computing the intersection, which can be represented as 1. The time complexity can then be rewritten as:

$$O(whn) \tag{9}$$

This time complexity means that given a screen resolution of x pixels, a scene with a large number of objects will take more rendering time than a scene with fewer objects.

2.2 Distributed ray tracing

The major drawback of the traditional ray tracing method is that it causes aliasing. When an intersection is not found, the background color is returned, and this means that for every point, a color is always returned. This behaviour can always lead to rendering of unintended patterns. The solution to this is to introduce more rays into the scene and also more randomness.

Distributed ray tracing extends the traditional ray tracing method by introducing the concept of sampling to remove the aliasing effects that exist in traditional ray tracing. By removing the aliasing effects, a photo-realistic image with better shadows, reflections, and refraction is rendered. The single ray in traditional ray tracing is replaced with a distribution of rays, and an average of a random sampling of the rays is taken to reduce aliasing effects. These random samples can be generated using Monte Carlo (MC) or Quasi-Monte Carlo (QMC) algorithms. The rendering time of a distributed ray tracing then becomes a function of the number of random rays sampled. This means that the rendering time of a distributed ray tracing application is always more than that of a traditional ray tracing technique.

In traditional ray tracing, tracing of rays is usually terminated after reaching a diffuse surface, but in the distributed technique, after a ray hits a diffuse object, child rays are generated randomly according to the bi-directional reflection and refraction distribution function of the diffuse surface [5].

A simple MC-based distributed ray tracing algorithm is presented in Algorithm 3. The algorithm shows how random rays are distributed in a single pixel, and for each ray a computation is done. The MC algorithm is then used for sampling, and the actual color of the pixel is computed based on the sampled rays.

for *each pixel in the viewing plane* **do**
  for *each ray in random rays* **do**
    for *each object in the scene* **do**
      if *ray intersects an object in the scene* **then**
        select min(d1,d2);
        recursively ray trace the reflection and refraction rays;
        return calculated color;
      end
      if *no intersection* **then**
        return background color;
      end
    end
  end
  random sample rays with montecarlo;
  calculate color average;
end

Algorithm 3: Distributed Ray Tracing Algorithm.

The time complexity of the algorithm is given as:

$$O(whrn) \tag{10}$$

where r is the number of distributed rays projected to each pixel. This time complexity means that a distributed ray tracing algorithm using the same configurations as a traditional ray tracing algorithm will surely take more rendering time.

2.3 OpenMP

Open Multi-Processing (OpenMP) has often been referred to as the de-facto standard for writing parallel programs with shared memory architectures, and these parallel programs can simply be achieved by adding compiler pragmas to the serial equivalent [7]. It allows for shared memory parallel programming in languages like C, C++, and Fortran. With OpenMP being a shared memory architecture, all threads spawned out have access to the same main memory and the same data. OpenMP is often used when there is a need to facilitate the execution of legacy code on a multi-core processor in order to utilize all its cores [7].

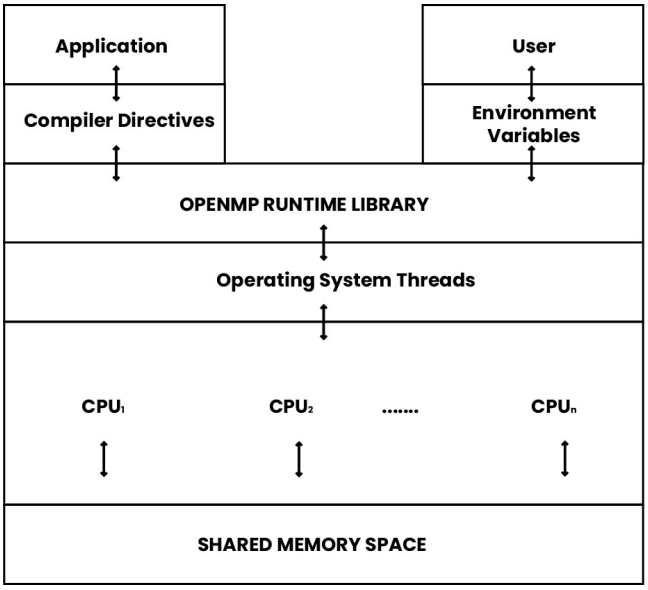

Figure 3: OpenMP Architecture.

In Figure 3 an architecture of OpenMP is presented. It can be seen that all the operating system threads have access to the same shared memory space, and the threads are being managed by the OpenMP run-time library. The parallelism in the run-time library is then specified using compiler directives, which are available in the application code.

In OpenMP, threads run in parallel, execute the same code, and share the same memory space. Each thread spawned out has a unique identifier that can be obtained using *omp_get_thread_num*().

Table 1 gives a list of some useful OpenMP directives and an explanation of what they do. The directives are always defined at the beginning of a blocked region. The *#pragma omp parallel* marks the entry point of a parallel region, and without that, threads cannot be spawned out.

A simple OpenMP algorithm that prints to console *hello world* is presented in Algorithm 4. The algorithm starts by specifying the parallel region of the code with the *#pragma omp parallel* directive, and the default number of threads is used. The blocked region is executed by all the threads in parallel. Each thread gets its identifier and prints *hello world* plus the identifier of the thread.

In summary, there are five major elements of parallel programming with OpenMP:

- Create threads with shared memory.
- Loop parallelism.
- Nested parallelism.
- Dynamic task scheduling.
- Thread synchronization.

3 RELATED WORK

Different state of the art methods have been developed to guarantee faster ray tracing rendering in computer graphics. Some of the most popular methods have explored the application of parallel programming techniques and the use of fast computation hardware like GPUs. The use of the Message Processing Interface (MPI) was explored in [2]. MPI is a message passing library which allows all available processors to communicate while executing the same program. All processors execute the same code, but in a separate isolated memory spaces, and communication between each processor is handled with an API. In the research, a near linear relationship between the number of processors used and the ray tracing rendering rate was reported. It was also reported that the efficiency rate achieved after adapting the serial ray tracing code to use MPI was over 98%. Unlike in [2], the focus of this paper is in the use of a shared memory device and not a distributed memory device which is used by MPI. Also, unlike in OpenMP, the time it took to communicate between processors used in MPI was also factored in when calculating the ray tracing rendering time.

Also in [6], an empirical study on the use of both OpenMP and MPI was explored for creating a parallel ray tracing. In the report, a pixel-wise load balancing scheme was introduced to allow load

Table 1: OpenMP Directives.

Directive	Function
#pragma omp parallel	Specifies the parallel region of a code.
#pragma omp for	Defines the start of a loop parallelism.
#pragma omp for simd	Defines the start of a loop parallelism that uses SIMD instructions.
#pragma omp single	Specifies a region that should be executed by a single thread.
#pragma omp sections	A non-iterative shared parallel section.
#pragma omp master	Specifies a region that should be executed only by the main thread.
#pragma omp ordered	Specifies a region that must be executed in order.
export KMP_AFFINITY=value	Used to define thread affinity types and is specific to Intel compilers. *value* can be *verbose*, *scatter*, or *compact*.
omp_get_num_threads	Used to get the count of all currently running threads.
omp_get_thread_num	Used to get the identifier of a currently running thread.
omp_set_num_threads	Used to set the number of threads that should be used for executing parallel regions.

```
#pragma omp parallel;
if main then
    int id = omp_get_thread_num();
    print("hello (%d)", id);
    print("world (%d) ", id);
end
```

Algorithm 4: Simple OpenMP Hello world.

distribution for some specific scenes. The report proved that both OpenMP and MPI are capable of achieving near optimal efficiency when used in ray tracing. In the research, a near linear speedup was also reported for both methods. Also, compiler performance between an Intel compiler and a GNU C++ compiler were studied, and the research reported that the Intel compiler outperformed the GNU C++ compiler. Unlike in [6], in our paper, the effect of different thread affinity types and different distributed systems was explored.

The use of GPU clusters for ray tracing rendering was explored by *Chen et al* [4]. In the research, the researchers were able to use GPU clusters to improve the rendering performance of a real-time ray tracing. The frame per second (FPS) achieved by the GPU cluster was compared to a single node GPU and a CPU. The research showed that the GPU cluster achieved more FPS than a single node GPU, and the single node GPU achieved more FPS than the CPU. A massively parallel ray tracing algorithm using a GPU was also developed by *Qin et al* [8]. The research showed that the GPU significantly reduced the rendering time of the ray tracing algorithm.

4 METHODS

Distributed ray tracing is computationally intensive, because intensive ray-geometry intersection computation must be done for rays projected into the scene through each pixel of the view plane. Since each ray is not dependent on the other, this also makes ray tracing embarrassingly parallel. To show that a distributed ray tracing is embarrassingly parallel, a parallel implementation in [1] was used as a case study. The scene consists of a spherical light source, a glass, a mirror, and one Cornell box, which was made of 6 spheres.

A loop-level parallelism was introduced using OpenMP to achieve parallelism in [1]. The loop-level parallelism made it possible to divide the intensive ray-geometry intersection among all participating threads. Each thread created by the process is allocated a task, they execute their own part of the code, and return with the result. Since OpenMP is a shared memory framework, the scene to be rendered was placed in shared memory in the form of a data structure, and this eliminated the possible overhead that could be introduced during data communication between threads, as all threads could access the same scene data.

In Algorithm 5, it can be seen that two levels of loop-level parallelism were implemented: one for each pixel and another for each ray distributed into the pixel. This was done because each pixel is independent of the others, and the same can be said for each of the distributed rays. Based on the algorithm, an empirical analysis is then carried out on how compiler types and numbers of threads affect rendering time. The focus here is measuring scability using speedup. Speedup was calculated using the formula:

$$Speedup = T_s / T_p \tag{11}$$

where T_s is the rendering time in serial and T_p is the rendering time in parallel for different thread numbers.

Also, the effect of using different thread affinity was explored in this paper. With thread affinity, we are able to control OpenMP thread placement, and this allows us to study the effect on the parallel ray tracing algorithm. In this paper, two types of thread affinity were explored, namely Scatter and Compact. When the Compact thread affinity type is specified, all the spawned out threads are placed close to each other. With Scatter, the threads are distributed as evenly as possible, and this eventually reduces the cache and memory bandwidth contention between threads.

```
#pragma omp parallel;
#pragma omp for;
for each pixel in the viewing plane do
    #pragma omp for;
    omp_set_num_threads(thread_num);
    for each ray in random rays do
        for each object in the scene do
            if ray intersects an object in the scene then
                select min(d1,d2);
                recursively ray trace the reflection and
                 refraction rays;
                return calculated color;
            end
            if no intersection then
                return background color;
            end
        end
    end
    random sample rays with montecarlo;
    calculate color average;
end
```

Algorithm 5: Parallel Ray Tracing with OpenMP.

5 RESULTS AND DISCUSSION

The code takes a single parameter as an input, which is the number of samples per pixel (spp), and for the performance evaluation of the distributed ray tracing, 25,000 samples per pixel (pixel) were used. The experiment was done on HPC clusters provided by University of Tartu [9], and the configuration is presented in Table 2.

Table 2: HPC Configuration.

Rocket cluster	Vedur cluster
20 cores	32 cores
2x Intel Xeon	2x AMD Opteron
64GB RAM	150GB RAM
1TB hard disk drive	500GB hard disk drive
4x QDR Infiniband	4x QDR Infiniband

The serial implementation of the parallel code in [1] was compiled using two different compilers: g++ compiler and Intel C++ (ICC) compiler. Figure 4 shows the rendered scene when the serial implementation was executed. The time taken to render the scene was measured, and the result is presented in Table 3.

Figure 5 is the scene rendered when the parallel implementation in [1] was executed, and it can be seen that the quality of the scene was preserved.

The rendering time of the parallel implementation was measured for both the Intel and g++ compilers for 2^n threads, where n is 1, 2, 3 or 4. Due to the number of cores available on Rocket Cluster, the

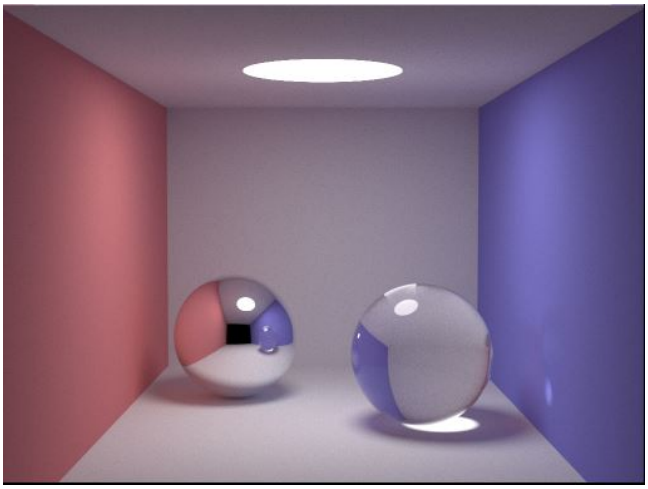

Figure 4: Serial ray tracing image with Cornell box, one light source, and two spheres, making use of 25,000 samples per pixel.

Table 3: Serial Ray Rendering.

Compiler	Rendering time(minutes)
Intel ICC compiler	120
g++ compiler	182

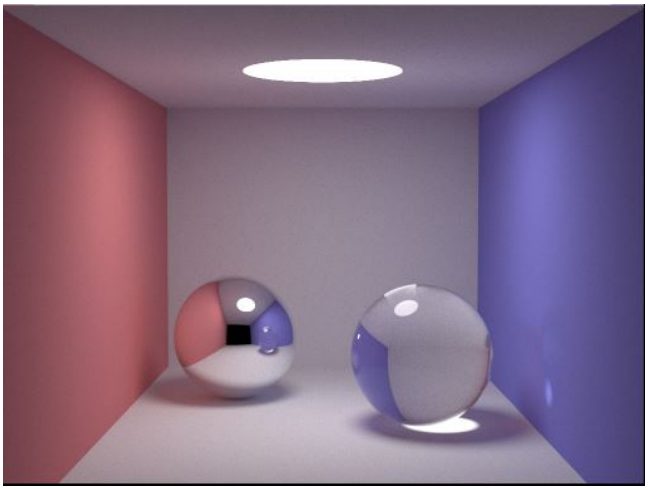

Figure 5: 20 threads on an Intel Xeon based cluster (Rocket cluster).

last thread number used during performance testing was 20 threads. Since the main aim of parallelism is to speed up the performance while maintaining the quality of the image, the quality of the image was studied throughout the test, and it was observed that the quality was preserved for each thread range.

Table 4 shows the rendering time in seconds and the speedup after optimizing using OpenMP for different thread numbers while executing with the Intel compiler. In Table 4, it can be seen that a linear speedup was achieved in the parallel implementation, and this is because the ray tracing algorithm is an embarrassingly parallel algorithm.

Table 4: Intel Xeon based cluster with Intel compiler

Threads	Rendering Time	Speedup
2	60 minutes	2.0
4	33 minutes	4.0
8	17 minutes	7.0
16	8 minutes	14
20	7 minutes	18

Table 5: Intel Xeon based cluster with g++ compiler

Threads	Rendering Time	Speedup
2	86 minutes	2.0
4	46 minutes	4.0
8	24 minutes	8.0
16	12 minutes	15
20	10 minutes 20 seconds	14

Table 5 shows the rendering time and speedup with the g++ compiler in the Rocket cluster. It is evident that the Intel compiler rendered the image faster than the g++ compiler; however, they both achieved a linear speedup for up to 16 threads.

The impact of using different thread affinities was also measured, and this result can be seen in Figure 6, which shows that the two thread affinities only increased the efficiency of the parallel execution of the code. We can also see that the speedup on 20 threads when using the compact thread affinity reduced compared to the scatter thread affinity.

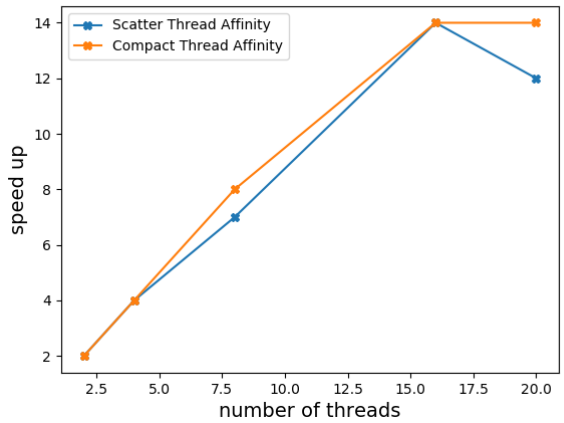

Figure 6: Speedup for Thread Affinity (Compact and Scatter)

After computing the performance of the parallel code on the Intel Xeon cluster, attempts were made to see the performance of the parallel code on an AMD based cluster, which is also one of the University of Tartu clusters. This was done because the AMD based cluster is not limited to 20 cores on a single node like the Intel Xeon cluster, but rather 32 cores. The rendering time of the parallel implementation was measured on only the Intel compiler, and the serial code rendering time was 252 minutes (4 hours 21 minutes).

Table 6: AMD based cluster with Intel compiler.

Threads	Rendering Time	Speedup
2	126 minutes	2.0
4	62 minutes	4.0
8	31 minutes	8.0
16	16 minutes	16
20	10 minutes	25

Comparing the AMD based cluster performance to that of the Intel Xeon based cluster, it can be observed that there is a difference in the rendering time, and this is due to the difference in hardware architecture of the two clusters. It is evident that linear speedup was also achieved up to 20 threads, and 100% efficiency was achieved up to 16 threads when running the parallel code on the AMD cluster. Again, the quality of image rendered was the same across the thread range.

In Figure 7 and Figure 8, the graphical representations for rendering time and speedup of all three cases are presented. It can be seen in Figure 7 that the rendering time decreases as the number of threads increases. However, as the number of threads approaches 20, the difference in rendering time starts to decrease for all cases. In Figure 8, the graphical representation of the speedup is presented. It can be seen that the speedup for each number of threads is specific to each case. However, the speedup achieved is the same when the number of threads spawned is less than five.

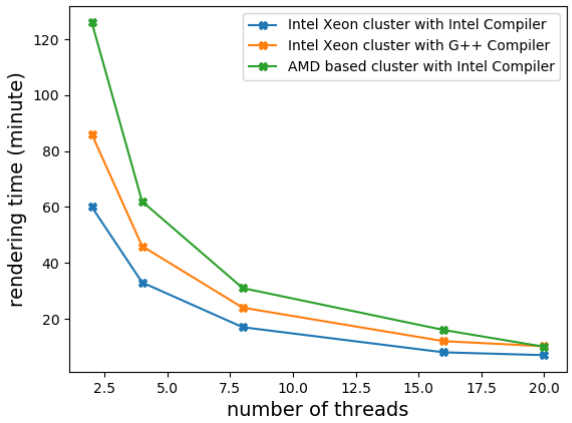

Figure 7: Render Time.

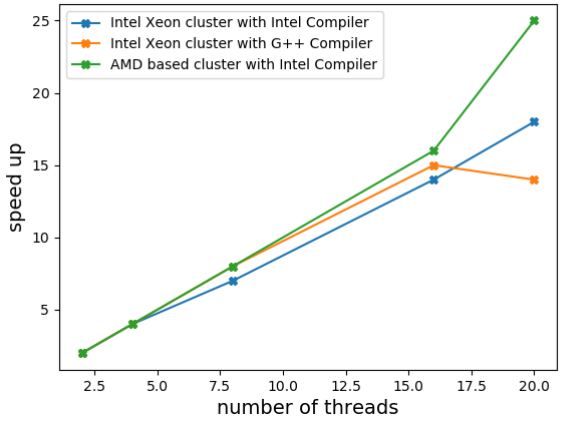

Figure 8: Speedup.

6 CONCLUSION AND REFLECTION

6.1 Conclusion

In this paper, a serial Monte Carlo based distributed ray tracing technique derived from a parallel implementation in [1] was compared to its parallel implementation, and the performance speedup on two compilers was measured while changing the number of threads. The test was carried out on University of Tartu's HPC cluster, and the performance test showed that a linear speedup was achieved, and while using thread affinities of type compact and scattered, an 100% efficiency was achieved for different thread numbers.

A very interesting future work will be to compare a serial real time distributed ray tracing implementation with a parallel implementation of both OpenMP and MPI. Since MPI is a distributed memory framework, overhead due to data communication between processors can be anticipated in MPI, and this might make it inefficient for pluralizing scenes with large data sets, such as data sets used in real time ray tracing.

6.2 Reflection

The experiment carried out in this paper was done while taking the parallel computing course at University of Tartu, and it was my first attempt at working in a high performance computing environment. Prior to this, I had no knowledge of parallel programming nor parallel computing. The greatest challenge for me while working on this project was understanding different frameworks for writing parallel programming codes like OpenMP and MPI. At the end, this experiment introduced me to parallel computing, different techniques for writing parallel programs, and methods for calculating parallel efficiency. In addition, it enriched my experience in the management and allocation of resources in leading-edge computing infrastructure through writing Slurm scripts.

Overall, working on this project was a unique experience and opportunity for me in the field of HPC. The experience gained was also useful when I was writing machine learning codes for my master's thesis.

ACKNOWLEDGMENTS

Huge thanks go to University of Tartu for making their HPC clusters available for this experiment. Also, big thanks go to Dr. Benson Muite for his support and guidance throughout the course of the experiment.

REFERENCES

[1] Kevin Beason. 2013. Monte Carlo Global illumination Code in C++. Retrieved April 4, 2020 from http://www.kevinbeason.com/smallpt/

[2] Charles B. Cameron. 2008. Parallel Ray Tracing Using the Message Passing Interface. *IEEE Transactions on Instrumentation and Measurement* 57, 2 (2008), 228–234.

[3] Chun-Fa Chang, Kuan-Wei Chen, and Chin-Chien Chuang. 2015. Performance comparison of rasterization-based graphics pipeline and ray tracing on GPU shaders. In *2015 IEEE International Conference on Digital Signal Processing (DSP)*. 120–123. https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7251842

[4] M. Chen and H. Huang. 2018. A Real-time Parallel Ray-tracing Method Based On GPU Cluster. In *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*. 327–330.

[5] Balázs Csébfalvi. 1997. A Review of Monte Carlo Ray Tracing Methods. Retrieved April 4, 2020 from http://www.cescg.org/CESCG97/csebfalvi/index.html

[6] Sadraddin A. Kadir and Tazrian Khan. 2008. *Parallel Ray Tracing using MPI and OpenMP*. Technical Report. Stockholm, Sweden.

[7] Manji Mathews and Jisha P. Abraham. 2016. Automatic Code Parallelization with OpenMP task constructs. In *2016 International Conference on Information Science (ICIS)*. 233–238.

[8] Y. Qin, J. Lin, and X. Huang. 2015. Massively parallel ray tracing algorithm using GPU. In *2015 Science and Information Conference (SAI)*. 699–703.

[9] University of Tartu. 2014. HPC cluster resources. Retrieved May 29, 2020 from https://hpc.ut.ee/en/resources/

[10] Jan Škoda and Martin Motyčka. 2018. Lighting Design Using Ray Tracing. In *2018 VII. Lighting Conference of the Visegrad Countries (Lumen V4)*. IEEE, 1–5. https://ieeexplore.ieee.org/document/8521111

[11] T. Whitted. 2020. Origins of Global Illumination. *IEEE Computer Graphics and Applications* 40, 1 (2020), 20–27.