Optimizing Monte-Carlo based Ray Tracing

AYOBAMI EPHRAIM ADEWALE*

University of Tartu
adewale@ut.ee

January 26, 2017

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 time. In this paper we discuss two different ray tracing techniques and the implementation of a parallel distributed ray tracing technique which in its sequential form is known to be computational intensive and costly for previous computers.

I. INTRODUCTION

Light tracing is an important aspect of computer graphics which have been adopted to simulate the real life behavior of light on an object, environment or scene in different areas such as animations, games and image rendering over the years. Due to its importance, there have been different techniques adopted for this purpose but the two most popular are:

- Rasterization
- Ray Tracing

Rasterization works by taking an image which is in a vector format and converts that image into pixels known as raster image for output on a video display or static environment [2], while Ray tracing on the other hand, takes an image from a 3D scene by tracing the trajectories of light rays through pixels in a view plane [1]. Basically, the main differences between these two techniques is the way they render images and the rendering performance. Rasterization has been adopted and has been the most popular of the two because it balances the performance needed with the ability to create acceptable images. Ray tracing creates more realistic images compared to rasterization and it is subdivided into two:
  - Traditional Ray Tracing
  - Distribution Ray Tracing

An example of traditional ray tracing is the Turner whitted technique while Monte Carlo technique is an example of the distributed ray tracing which extends the traditional ray tracing through the use of Monte Carlo integration. The Monte-Carlo algorithm simulates light effects better and produces better realistic images than the whitted algorithm and this makes it more computational intensive and too costly for earlier computers compared to Turner Whitted.

With the recent advances in computing power, various researchers have picked up an interest in finding different ways of reducing the computational intensity of distributed ray tracers while retaining its property of creating quality realistic images. The use of parallel frameworks such as MPI, OPENMPI and OPENCL have been proposed by researchers over the years and in this paper we demonstrate how OPENMP was used to speedup the rendering time of a distributed Monte-Carlo
ray tracer. This test is based on a sequential distributed Monte-Carlo ray tracer called smallpt in [4].

The remainder of this paper is organized as follows, section 2 gives an overview of ray tracer and an algorithm, test and discussion are presented in section 3, and the last section covers future works and conclusion.

II. RAY TRACING OVERVIEW

As described above, ray tracing takes an image from a 3D scene by tracing the trajectories of light rays through pixels in a view plane and this is referred to as forward ray tracing. The other type of ray tracing is known as backward ray tracing, where rays are traced from the light source to the eyes. This method is computation intensive because each rays emitted by the light source have 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 camera, scene, object(s) and light source.

Figure 1 above describes a simple ray tracing scenario. A ray from the eyes is cast in a straight line for each pixel of the view plane into the scene and a ray intersection is checked to see if ray intersects with any object in the scene. This first ray is referred to as the primary ray and it 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 Intersection

The third case only happens if the object in the scene is opaque. If an intersection exist, 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 cast as a shadow and if it successfully reaches the light source, that pixel is lighten up. The lighting of the pixel is determined by this secondary ray. If the point of intersection is two, the distance of the two points to the eyes is 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 ray tracing technique to create better realistic rendering.
Result: RAY TRACER ALGORITHM
for each pixels 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

The algorithm above 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 line which has an origin represented as $O$ and a direction represented as $l$. At any time, a point on the line or ray can be represented as

$$\text{Point} = O + (l \times d)$$

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 casted 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 $dc$:

$$dc = C_0 \cdot l; \quad \text{where} \quad C_0 = C - O.$$  

dc is the distance of the casted 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 $\cdot$ denotes a dot product. If $dc$ is less than 0 then it can be assumed that there is no intersection but if it is greater than 0 we proceed to the second step which is to calculate the distance from the center of the sphere to the casted ray. This can be calculated using pythagoras theorem because now, we have formed a right angle triangle.

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

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

$$P = O + (l \times d_p)$$

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

$$d_p = dc - tc$$

where $tc$ can be obtained from the second right angled triangle in our sphere.

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

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 $d_{p2} = dc + tc$.

The algorithm below describes the ray intersection algorithm.

**FUNCTION**: boolean intersection(Ray*R, Sphere*S, float * d_p1, float * d_p2)

1. float $C_0 = S \rightarrow \text{center} - R \rightarrow \text{Ray Origin}$
2. float $dc = \text{dot}(C_0, R \rightarrow \text{direction})$
3. if $dc \leq 0.0$ then
   return false
4. else
   float $D = \sqrt{dc^2 - (C_0)^2}$
   if $D \geq S \rightarrow \text{radius}$ then
      return false
   end if
5. float $tc = \sqrt{(S \rightarrow \text{radius})^2 - D^2}$
6. $d_p = dc - tc$
7. $d_{p2} = dc + tc$
8. return true;

end if
As mentioned in the previous section, distributed ray tracing extends the traditional ray tracing. It does these by combining both the forward ray tracing and the backward ray tracing so as to achieve a photo realistic image. In the 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. Monte Carlo Integration method is then introduced to estimate the color of pixels according to the contributions of each particles by converting the problem into an equivalent expected value computation. This approach is popularly referred to as the Monte Carlo technique because it samples the signal represented by incident light on the shading point and therefore ends up producing a photo-realistic image.

III. IMPLEMENTATION

It was mentioned in the previous section that distributed ray tracing is computational intensive because intensive ray-geometry intersection computation must be done for rays casted into the scene through each pixel of the view plane. Since each ray is not dependent on the other, this makes ray tracing also embarrassingly parallel. To show that a distributed ray tracing is embarrassingly parallel, the smallpt code was used as a case study. The scene consist of a spherical light source, a glass, a mirror and one cornell box which was made of 6 spheres.

OPENMP divide and conquer model was used in parallelizing the serial ray tracing code and this made it possible to divide the intensive ray-geometry intersection among all participating threads. Each thread created by the process is allocated a task by the master thread, 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 a shared memory in form of a data structure and this eliminated the possible overhead that could be introduced during data communication between processor or threads as threads could access the scene data.

\[
\text{Taskperthread} = \frac{\text{ImageWidth} \times \text{Height}}{\text{NumofThread}}
\]

OpenMP parallelism was implemented by placing OpenMP parallefor pragma with dynamic schedule clauses on the outer for nested loop. Since ray is private to each threads, the ray construct was made private to each participating thread using the private clause.

IV. TEST 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, I made use of 25000 samples per pixel (pixel). The serial smallpt code was compiled using two different compilers G++ compiler and intel c++ (icc) compiler. It ran on two nodes of intel processors with a memory size of 3GB on Rocket which is the University of Tartu cluster and the execution time was measured so as to calculate the performance Speed-up of the parallel code.

![Serial Ray Tracing Image with Cornell Box, One Light Source and Two Spheres making using of 25000 samples per pixel.](image-url)
Figure 4: Speed Up Graph

Figure 5: Speed Up Graph for Thread Affinity (Compact and Scatter)
The rendering time of the parallel implementation discussed in section III was measured for both the intel and g++ compilers for thread of $2^n$ where $n$ is between 1,2,3 and 4. Due to the number of cores available on Rocket Cluster, the 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 all thread range.

Speed up was calculated using the formula:

$$\text{Speedup} = \frac{T_s}{T_p}$$

where $T_s$ is the rendering time in serial and $T_p$ is the rendering time in parallel for different thread numbers. In the performance test, time to write output to file was not measured.

On Intel icc compiler, the serial code took 120 minutes (2 hours) to render image and while on the g++ compiler, it took 182 minutes (3 hours) to render same image with the same samples per pixel.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Rendering Time</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>60 minutes</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>33 minutes</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>17 minutes</td>
<td>7.0</td>
</tr>
<tr>
<td>16</td>
<td>8 minutes</td>
<td>14</td>
</tr>
<tr>
<td>20</td>
<td>7 minutes</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 1 shows the rendering time in seconds and the speed up for the different thread numbers while executing on intel compiler. The graph in figure 4 was generated from the table.

The graph in figure 4 showed that linear speed up was achieved in the parallel execution of the code and this is because ray-tracing algorithm is embarrassingly parallel.

Table 2 below shows the rendering time and speed up on g++ compiler in rocket cluster.

<table>
<thead>
<tr>
<th>Thread</th>
<th>Rendering Time</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>86 minutes</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>46 minutes</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>24 minutes</td>
<td>8.0</td>
</tr>
<tr>
<td>16</td>
<td>12 minutes</td>
<td>15</td>
</tr>
<tr>
<td>20</td>
<td>10 minutes 20 Sec-</td>
<td>14 seconds</td>
</tr>
</tbody>
</table>

The impact of using different thread affinities was also measured and the graph is shown in figure 5. In the graph, we can see that the two thread affinities only increased the efficiency of the parallel execution of the code. We can also see that the speed up on 32 threads when using compact thread affinity reduced compared to the scatter thread affinity.

After computing the performance of the parallel code on the Rocket Cluster, attempts were made to see the performance of the parallel code on Vedur Cluster which is also the University of Tartu cluster. This was done because the Vedur Cluster is not limited to 20 cores on one node like the Rocket Cluster but 32 cores on one node. This gave me the advantage to see the performance of the code from 2 threads to 32 threads which was not possible on Rocket Cluster. 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>
<thead>
<tr>
<th>Thread</th>
<th>Rendering Time</th>
<th>Speed Up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>126 minutes</td>
<td>2.0</td>
</tr>
<tr>
<td>4</td>
<td>62 minutes</td>
<td>4.0</td>
</tr>
<tr>
<td>8</td>
<td>31 minutes</td>
<td>8.0</td>
</tr>
<tr>
<td>16</td>
<td>16 minutes</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>10 minutes</td>
<td>25</td>
</tr>
</tbody>
</table>

Comparing the vedur performance table to that of Rocket cluster, it can be observed that there was 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 to upto 32 threads and 100% efficiency was achieved to upto 16 threads when running the parallel code on Vedur cluster. Again, the quality of image rendered was the same on all thread range.

V. Conclusion and Future Works

In this paper, I have successfully shown how OPENMP can be used to speed up the performance of a Monte-Carlo based distributed ray tracing technique while using smallpt [4] as a case study. The test was carried out on University of Tartu cluster and the performance test showed that linear speed up was achieved and while using thread affinities of type compact and scattered, an 100% efficiency was achieved for different thread numbers. It would be worth the mention that the OPENMP implementation discussed in this paper could have been compared with an OPENMPI implementation but as at the time of writing, this is yet to be carried out. A very much interesting future work will be to compare a sequential real time distributed ray tracing implementation with a parallel implementation of both OPENMP and OPENMPI. Since OPENMPI is a distributed memory framework, overhead due to data communication between processors can be anticipated in OPENMPI and this might make it inefficient for parallelizing scenes with large data sets such as data sets used in real time ray tracing.

Furthermore, attempt to run the parallel version of the smallpt code on Intel Xeon Phi was not successful and this was due to issues with pragma offload target (mic) clause on Rocket Cluster. Though the code was optimized through loop vectorization while compiling to run on XEON PHI, the result of this optimization was not visible till it was ran using 32 threads. The result of this optimization is not recorded in this paper as this was not the aim of the additional experiment, aim was to offload sections of the code to XEON PHI then measure the rendering time while considering different thread numbers. This experiment can be added to future works, as it will be interesting to see how it will affect the rendering time.

References

Scientific Computing, Royal Institute of Technology (KTH), Stockholm, Sweden.


Smallpt is a c++ implementation of a Monte-Carlo based ray tracing technique.