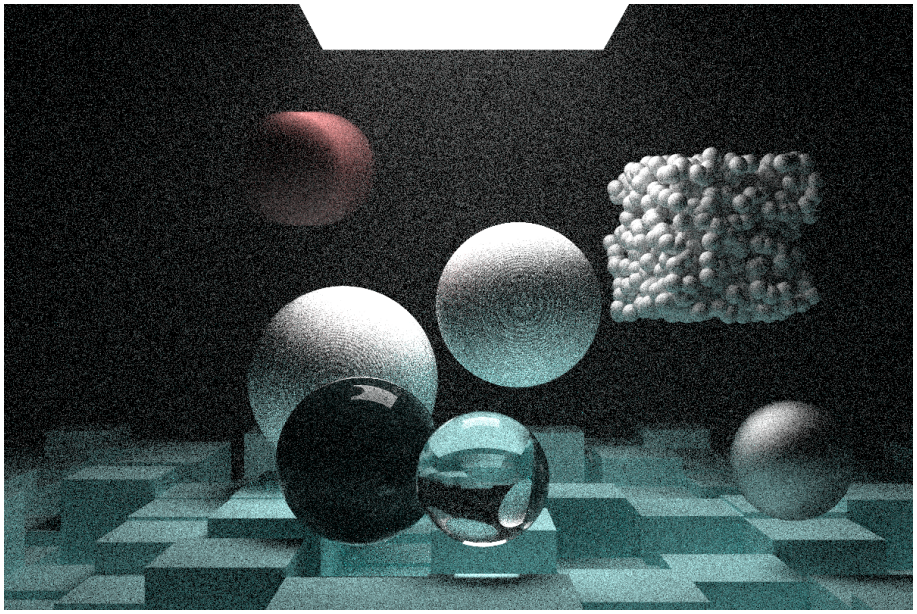
Parallel Approaches to Ray Tracing



Shounak Dey

Shivika Singh

Abstract

In this project we present an implementation of Monte Carlo Ray Tracer using a C++ code base. We also use two C++ libraries MPI and Cuda to present two different parallelization techniques of this algorithm. Using MPI, we process the pixels of the output image in batches, thus allowing us to parallelize the algorithm. Similarly, in Cuda, we parallelize the algorithm by. In the field of computer graphics, Ray Tracing is a rendering technique used to generate an image by tracing the path of light rays as pixels on a plane and simulating various effects and encounters with virtual objects. This technique is capable of producing high degrees of visual realism but at the sacrifice of computational costs. In this project we make an attempt at reducing the time taken to render an image using parallelization techniques.

1 Motivation

In parallel computing, an Embarrassingly Parallel problem is one where the problem can divided into subproblems with little or no extra work. This is also the case where there is no requirement of communication between two different process to complete the tasks given to each process.

We notice that in the problem of Ray Tracing, the individual values of each pixel in the final image pane are independent of each other and hence require no communications between any two pixels. This points towards the fact that problem of Ray Tracing is an Embarrassingly Parallel problem. This allows us to explore new approaches of parallelizations towards this algorithms.

2 Objectives

The objective of this project is to draw comparisons between different approaches towards implementing the Ray Tracing algorithm. We take three approaches here,

1. Serial approach
2. Parallelization using MPI
3. Parallelization using Cuda

Exploring new approaches towards the parallelization of this algorithm allows us to increase the efficiency of the algorithm and henceforth creating an option of using such approaches in Real time Rendering softwares.

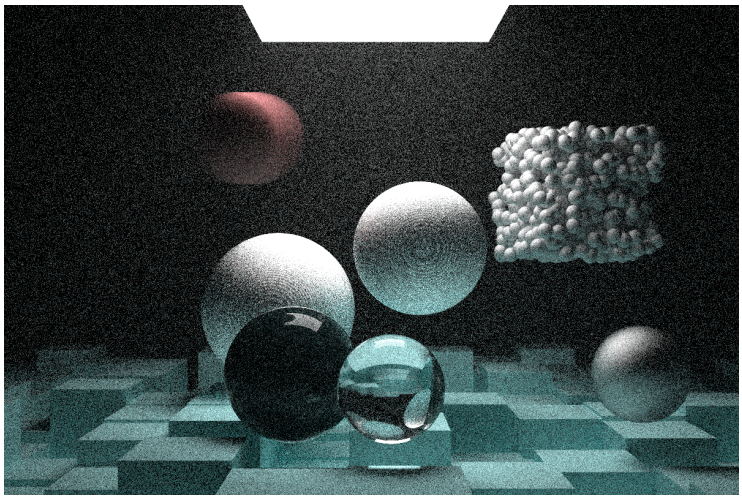


Figure 1: A complex scene generated using 100 samples for each pixel

3 Introduction

In the field of computer graphics, Ray Tracing is a rendering technique used to generate an image by tracing the path of light rays as pixels on a plane and simulating various effects and encounters with virtual objects. This technique is capable of producing high degrees of visual realism but at the sacrifice of computational costs.

Optical ray tracing describes a method of producing visual images constructed in 3D computer graphics environments, with more photorealism than either casting or scanline rendering techniques. It works by tracing the path of an imaginary ray through each pixel of the image pane (virtual screen).

In this project, the scenes are given to us as a collection of predefined simple objects like spheres, cuboids, etc of different properties. Each object can be made of different kinds of materials like dielectrics, metals, etc.

4 Methodology

At the core of the ray tracer we send rays through each pixel of the Image pane and color seen at the end of this ray. This is of the form *compute which ray goes from the eye to the pixel, compute what the ray intersects and find the color at these intersection points.*

Our color function in the file tracer finds the color at the end point of a ray, given the world of objects and its properties. The color function recurses a maximum of 50 times or in other words, the ray will interact with any object in the world a maximum of 50 times. Once it hits an object, we return the emitted

color of the object. At this point we also calculate if there is any scattering. If there is we scatter the ray and return the computed color multiplied with a factor also called attenuation. If the ray does not hit any object, we return the color Black signifying darkness.

We implement multiple types of materials in our code base. Let us briefly look at some of their key differentiating properties,

- **Lambertian (or) Diffuse Materials:** These types of materials modulate their own color with their own intrinsic color. Light that reflects off a diffuse material has its direction randomized as shown in Figure 2.

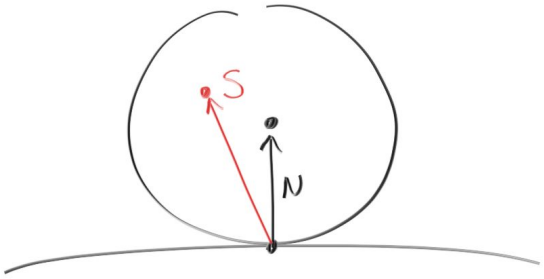


Figure 2: Computing the scattered ray in a diffuse material

- **Metals:** A metal material just reflects the incident rays according to the laws of reflection. The computation of a reflected ray in the case of metal is shown in Figure 3.

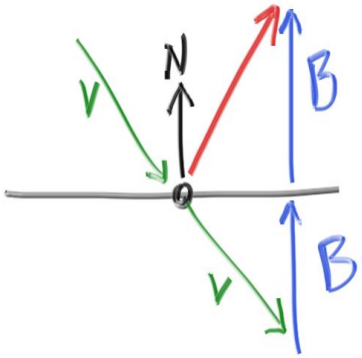


Figure 3: Computing the reflected ray for a metal

  In addition to this, the metal class also has a fuzz attribute which determines the fuzziness of the reflection.

- **Dielectrics:** Clear materials such as water,glass, and diamonds are dielectrics. When a light ray hits them, it splits into a reflected and refracted ray. To reduce the computations we randomly select only of these rays to pursue.

4.1 MPI

In the MPI code base, we parallelize the computation of colors for each pixel in the image pane. We divide the image into rows of pixels and parallelize stacks of rows. So if there were 4 processes, then we divide the entire image into 4 stacks of rows and the process them in parallel. In each process we call the same function each time and save the pixel values into a 3D array of integers. We use PPM image format where each line consists of the RGB values of each pixel. To display the output we print the values of the final array in process 0 after calling an MPI_Barrier.

4.2 Cuda

In the Cuda version, the processing of the image has been done in the batch of 8 by 8 pixels. Four kernels are called in the main function. The first two kernels have been used for initializing and setting up the type of scene that needs to be rendered. The next two functions render_init and render have been used to initialize the random numbers for each thread and render the required image respectively.

5 Results

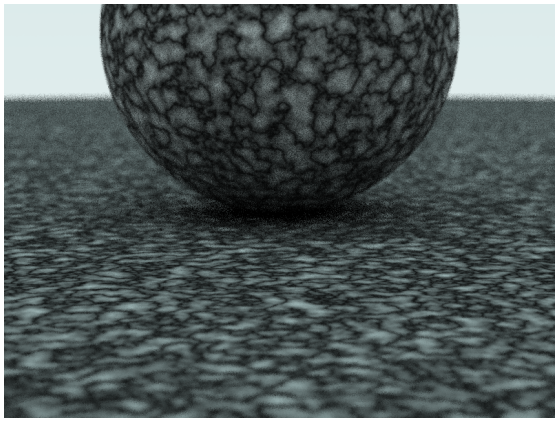


Figure 4: Two Perlin Spheres

Time Comparison				
Scene	Serial Runtime(s)	MPI Runtime(s)	Dimensions	Samples
Figure 4	105.931	22.01	800x600	100
Figure 6	67.20	16.07	300x200	100
Figure 5	13.64	7.039	400x200	100

Table 1: Comparison between MPI and Serial runtimes

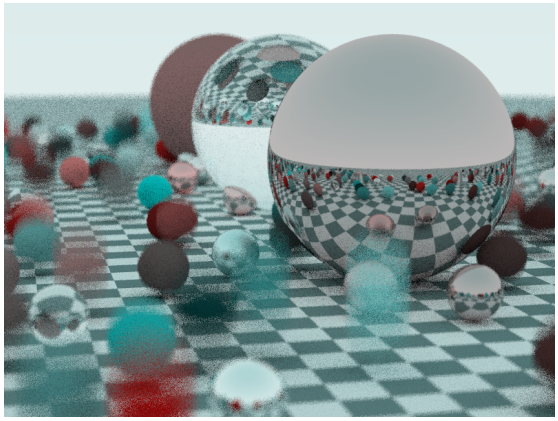


Figure 5: A scene with many random spheres and 3 fixed ones

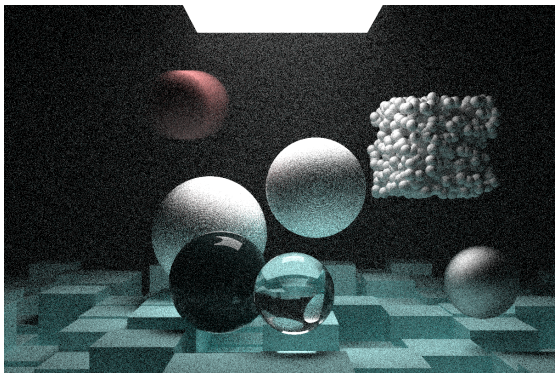


Figure 6: A complex scene with few random spheres and a cube of 1000 random spheres

Runtimes in Cuda		
Runtime(s)	Dimensions	Samples
22.8181	300x150	10
10.3382	300x200	10
12.6927	400x200	10
284.31	1200x800	10
11.2459	210x110	10

Table 2: Cuda runtimes

6 Conclusion

Ray tracing has been a field of study for over 25 years now, and its algorithms have been fine tuned and optimized. Current methods aim to achieve real-time ray tracing systems, with speed as a primary concern whilst not doing too much of a trade-off for quality. And so, when dealing with real-time systems the performance do not match the off-line rendered(ray-traced) images (Off line systems

take between a few seconds to a few days even to simulate a few images/graphics scenes). When coming to to the accuracy, well, the pictures speak for themselves. Current ray-tracing systems are capable of producing highly realistic images and parallelizing it's implementation decreases the compute cost.

References

[1] M Ashraful Kadir, S Khan, Tazrian. (2008). Parallel Ray Tracing using MPI and OpenMP

[2] Johansson G., Nilsson O., Sderstrm A. and Museth K.: Distributed Ray Tracing In An Open Source Environment, Proceedings of the SIGRAD 2006 Conference (2006), Linkping University Electronic Press and OpenMP

[3] Freisleben, B., Hartmann, D. and Kielmann, T.: Parallel raytracing: a case study on partitioning and scheduling on workstation clusters, Volume 1, Issue 7-10 (1997), Proceedings of the Thirtieth Hawaii International Conference on System Sciences

[4] https://sites.google.com/site/raytracingasaparallelproblem/

[5] Rendering large scenes using parallel ray tracing ErikReinhard, Frederik W.Jansen