

Project Report

Group 8

Ray Tracing using Map Reduce

Submitted by: Bhavika Kalani

Raghav Bhandari

Yuan-lin Hsu

Introduction/ Motivation

This is a paper published in 2013, Lesley et al. which builds a ray tracing renderer on Hadoop. This helps the algorithm to become scalable to computer cluster. This is a scalable ray tracing framework which can be done on general pay-as-go cloud computing services. Hadoop online ray tracer or HORT will be using map reduce to partition the computational workload and scene data. This will be different from the distributed memory ray tracing frameworks.

Ray tracing is the most realistic computer graphic rendering technique, yet it requires tremendous computational power as the complexity goes exponential to the number of objects in scene. Typically for Walt Disney animation, one frame takes hour(s) or day(s) to render in a single computer. Luckily for them, they have thousands of frames to render, and they can distribute the task on a per frame basis over machine farm. However, for other type of simulation such as internal design, it would accelerate the process if we can divide the rendering of one frame to many subtasks and utilize cluster computing by designing good map reduce algorithm.



It takes longer to render complicated objects. Eg: Snowflakes, Smog, Ripples.

The Apache Ambari project is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters. Ambari provides an intuitive, easy-to-use Hadoop management web UI backed by its RESTful APIs.

Concept of Ray Tracing:

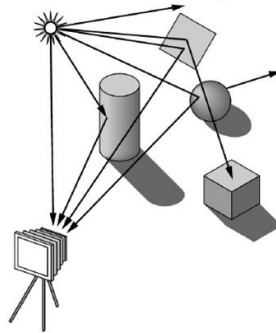
We can create an image from a three-dimensional scene in a two step process. The first step consists of projecting the shapes of the three-dimensional objects onto the image surface (or image plane). This step requires nothing more than connecting lines from the objects features to the eye. An outline is then created by going back and drawing on the canvas where these projection lines intersect the image plane. As you may have noticed, this is a geometric process. The second step consists of adding colors to the picture's skeleton.

Forward Tracing:

We replace our eyes with an image plane composed of pixels. In this case, the photons emitted will hit one of the many pixels on the image plane, increasing the brightness at that point to a value greater than zero. This process is repeated multiple times until all the pixels are adjusted, creating a computer generated image. This technique is called **forward ray-tracing** because we follow the path of the photon forward from the light source to the observer.

Forward Ray Tracing

- Rays as paths of photons in world space
- Follow photon from light sources to viewer
- Problem: Many rays will not contribute to image



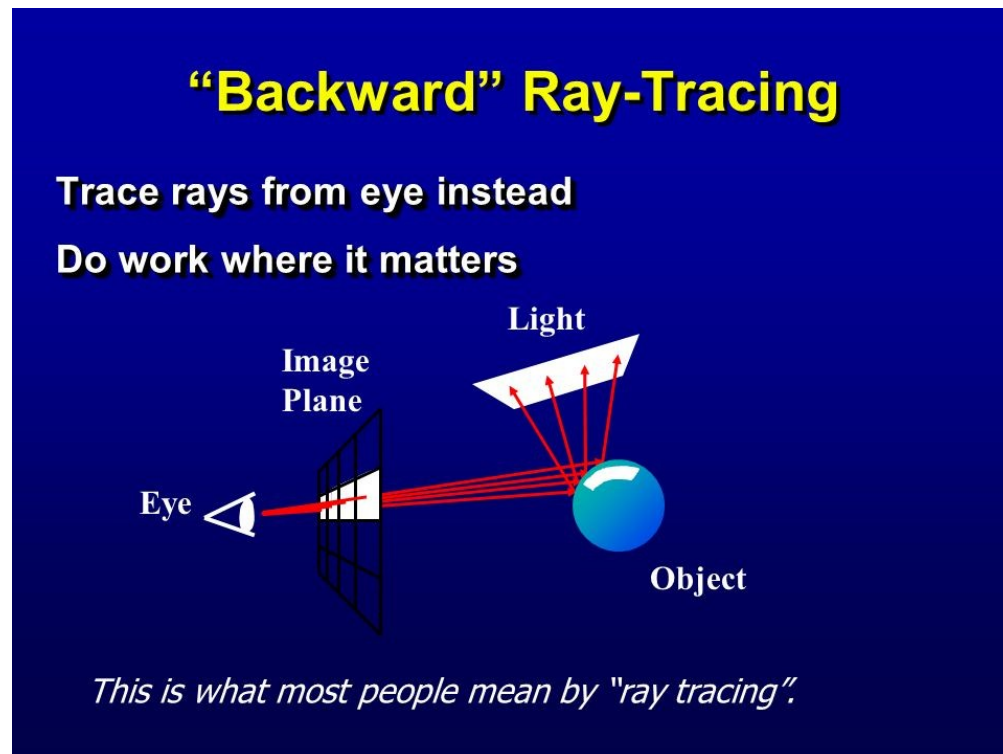
Issues with Forward Ray tracing

Rays are essentially reflected in every possible direction, each of which have a very, very small probability of actually hitting the eye. We would potentially have to cast zillions of photons from the light source to find only one photon that would strike the eye. In the computer world, simulating the interaction of that many photons with objects in a scene is just not practical solution for reasons we will now explain. **Forward ray-tracing** (or **light tracing** because we shoot rays from the light) makes it technically possible simulate the way light travels in nature on a computer. However, this method, as discussed, is not efficient or practical.

Backward Tracing

Instead of tracing rays from the light source to the receptor (such as our eye), we trace rays backwards from the receptor to the objects. Because this direction is the reverse of what happens in nature, it is fittingly called **backward ray-tracing** or **eye tracing** because we shoot rays from the eye position. This method provides a convenient solution to the flaw of forward ray-tracing. Since our simulations cannot be as fast and

as perfect as nature, we must compromise and trace a ray from the eye into the scene. If the ray hits an object then we find out how much light it receives by throwing another ray (called a light or shadow ray) from the hit point to the scene's light. Occasionally this "light ray" is obstructed by another object from the scene, meaning that our original hit point is in a shadow; it doesn't receive any illumination from the light. For this reason, we don't name these rays light rays instead **shadow rays**. Let's call, the first ray we shoot from the eye into the scene a **primary ray**, **visibility ray**, or **camera ray**.



Ray tracing Algorithm

The ray-tracing algorithm takes an image made of pixels. For each pixel in the image, it shoots a primary ray into the scene. The direction of that primary ray is obtained by tracing a line from the eye to the center of that pixel. Once we have that primary ray's direction set, we check every object of the scene to see if it intersects with any of them. In some cases, the primary ray will intersect more than one object. When that happens, we select the object whose intersection point is the closest to the eye. We then shoot a shadow ray from the intersection point to the light. If this particular ray does not intersect an object on its way to the light, the hit point is illuminated. If it does intersect with another object, that object casts a shadow on it. If we repeat this operation for every pixel, we obtain a two-dimensional image projection of our three-dimensional scene.

Pseudocode:

<define objects, light source and cameras in the scene>

For (int r=0; r<nRows; r++)

For (int c=0; c<nCols; c++)

<1.Build the rc-th ray>

<2.Find all intersections of the rc-th ray with objects>

<3.Find intersection which lie closest to the eye>

<4.compute the hit point where the ray hits this object, and the normal vector at that point >

<5.Find the color of the light returning to the eye along the ray from point of intersection>

<6.Place the color in the rc-th pixel>

Using MapReduce for Ray Tracing

To utilize the power of multiple processors in a computing cluster, one can use MapReduce on ray tracing problem. There are several directions to consider:

1. How to efficiently distribute the scene files into blocks that are easy for each computing node to access.
2. How to distribute the task such that each computing node would have an equal shared amount of workload.
3. Jointly optimize the above two points, that is, apply an algorithm to divide the scene file into (overlapping) parts, and each parts is collectively optimal for accessing by nodes running equally distributed workloads.

Our project is mainly addressing on the second target, as it's the simplest of the three, and is suitable for our scope. Therefore, we made the assumption that each node would access a complete copy of scene file with ease. (Note that this assumption may fail if in a industrial standard, where scene file can be as large as several TB, and the memory for a single node lies typically in scale of GB). In our experiments, we further simplified the scene composition to contain only two basic type of geometric shape, triangle plane and sphere. These setting is quite common for ray tracing benchmark because complex shape can be decomposed into triangular plane, and sphere is the simplest form of algebraic surface model.

Methodology

Step1: Set-up and Install Ambari

Ambari is a technology build on Hadoop ecosystem that provide graphic user interfaces for real time monitoring of the hadoop cluster and dynamic deployment.

We have used Ambari to run our experiment mainly because of its easy to use file system interface is easier to use then hadoop command line. Besides that, it also handles the software installation/deployment to every nodes of the computing cluster.

1.1 Minimum requirements for Ambari:

1.Operating Systems Requirements

2.Browser Requirements: The Ambari Install Wizard runs as a browser-based Web application.

3.It needs a machine capable of running a graphical browser to use this tool.

4. Software Requirements

5. JDK Requirements

6. Database Requirements: Ambari requires a relational database to store information about the cluster configuration and topology.

7. Memory Requirements: The Ambari host should have at least 1 GB RAM, with 500 MB free

8. Package Size and Inode Count Requirements

9. Recommended Maximum Open File Descriptors

1.2 Prepare the Environment

We need to prepare our deployment environment which include the following steps:

Set Up Password-less SSH

Set Up Service User Accounts

Enable NTP on the Cluster and on the Browser Host

Check DNS and NSCD

Configuring iptables

Disable SELinux and PackageKit and check the umask Value

Step 2: Installing, configuring and Deploying a cluster

The following steps were followed

- Start the Ambari Server
- Log In to Apache Ambari
- Launch the Ambari Cluster Install Wizard
- Name Your Cluster
- Select Version
- Install Options
- Confirm Hosts
- Choose Services
- Assign Masters
- Assign Slaves and Clients
- Customize Services
- Review
- Install, Start and Test
- Complete

Step 3: Running up experiment

Ray Tracing with CPU cluster

Ray tracing is state of the art computer graphics rendering technique, but requires lots of computational power. By using cluster, we can distribute tasks with divide and conquer paradigm and gain speed advantage. Therefore how to divide and distribute the task is crucial for optimization. We focus on optimizing the computing time of a single frame rendering.

The Original Method is Horizontal Approach where the following is done:

- 1.Split Input into horizontal regions.
- 2.Map: Do the raytracing work.
- 3.Reduce: Put back result together.

Issues with horizontal approach is Workload imbalance:

Because the split region is continuous, if the computing heavy (e.g. transparent/reflective) objects are distributed in one region, that region would need a lot of time to render.

We have used the following approach:

1. Split input by alternate horizontal rows (Odd/Even).
2. Map.
3. Reduce.
4. Post processing rearrange. (takes less than 50ms)

The experiment results show that our approach is better than horizontal approach.

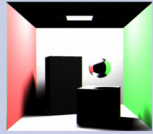
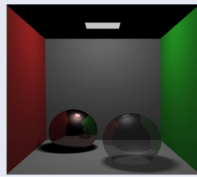
Scene	Resolution	Content
basiccornellbox	800/800	
cornellbox	1200/1200	

Figure 1 Rendering Target

The first target(**basiccornellbox**) is an example of rendering image where the upper half is as computationally hard as the bottom half for rendering. The second target(**cornellbox**) is an example of bottom half of image is much harder to render than the upper half, because the reflection and refraction usually take 3~5x more computation than diffusion.

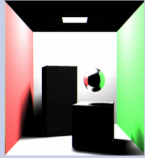
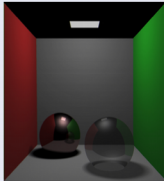

		(original)	(our method)
	62870 ms	41759 ms	41724 ms
	121448 ms	94944 ms	72887 ms+

Figure 2 Running Time

The running time starts counting before the hadoop job is started, and finish counting when the job is over. While our method takes about the same computing time as the naïve method in the case of **basiccornellbox**, our method render faster in the case of **cornellbox**. The first column shows the running time of a single node, just for comparism.



Map Tasks for job_1511750314412_0011

Application

Job

Overview

Counters

Configuration

Map tasks

Reduce tasks

Tools

Show 20 entries

Name	State	Task	Start Time	Finish Time	Elapsed Time	Start Time	Successful Attempt	Finish Time	Elapsed Time
task_1511750314412_0011_m_000000	SUCCEEDED	Tue Nov 28 11:42:08 -0800 2017	Tue Nov 28 11:42:35 -0800 2017	26sec	Tue Nov 28 11:42:08 -0800 2017	Tue Nov 28 11:42:35 -0800 2017	26sec		
task_1511750314412_0011_m_000001	SUCCEEDED	Tue Nov 28 11:42:09 -0800 2017	Tue Nov 28 11:42:35 -0800 2017	25sec	Tue Nov 28 11:42:09 -0800 2017	Tue Nov 28 11:42:35 -0800 2017	25sec		

ID	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time
----	-------	------------	-------------	--------------	------------	-------------	--------------

Showing 1 to 2 of 2 entries

Logged in as: droth

Application

Job

Overview

Counters

Configuration

Map tasks

Reduce tasks

Tools

Show 20 entries

Name	State	Task	Start Time	Finish Time	Elapsed Time	Start Time	Successful Attempt	Finish Time	Elapsed Time
task_1511750314412_0033_m_000000	SUCCEEDED	Tue Nov 28 16:14:17 -0800 2017	Tue Nov 28 16:14:43 -0800 2017	26sec	Tue Nov 28 16:14:17 -0800 2017	Tue Nov 28 16:14:43 -0800 2017	26sec		
task_1511750314412_0033_m_000001	SUCCEEDED	Tue Nov 28 16:14:17 -0800 2017	Tue Nov 28 16:14:43 -0800 2017	25sec	Tue Nov 28 16:14:17 -0800 2017	Tue Nov 28 16:14:43 -0800 2017	25sec		

ID	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time
----	-------	------------	-------------	--------------	------------	-------------	--------------

Showing 1 to 2 of 2 entries



Map Tasks for job_1511750314412_0033

Figure 3 Map Task Running time for basiccornellbox

The running time for both nodes in basiccornellbox is about the same in both methods, because both way of splitting produces equally hard rendering tasks.



Map Tasks for job_1511750314412_0008

Logged in as: drwho

Application

Job

Overview

Counters

Configuration

Map tasks

Reduce tasks

Tools

Show 20 entries

			Task		Successful Attempt		
Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time
task_1511750314412_0008_m_000000	SUCCEEDED	Tue Nov 28 11:30:06 -0800 2017	Tue Nov 28 11:31:27 -0800 2017	1mins, 21sec	Tue Nov 28 11:30:06 -0800 2017	Tue Nov 28 11:31:27 -0800 2017	1mins, 21sec
task_1511750314412_0008_m_000001	SUCCEEDED	Tue Nov 28 11:30:07 -0800 2017	Tue Nov 28 11:30:38 -0800 2017	31sec	Tue Nov 28 11:30:07 -0800 2017	Tue Nov 28 11:30:38 -0800 2017	31sec
ID	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time

Showing 1 to 2 of 2 entries

First Previous 1 Next Last



Map Tasks for job_1511750314412_0034

Logged in as: drwho

Application		Show 20 entries					Search:			
Job		Task					Successful Attempt			
Overview		Name	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time	
Counters		task_1511750314412_0034_m_000000	SUCCEEDED	Tue Nov 28 16:28:37 -0800 2017	Tue Nov 28 16:29:34 -0800 2017	56sec	Tue Nov 28 16:28:37 -0800 2017	Tue Nov 28 16:29:34 -0800 2017	56sec	
Configuration		task_1511750314412_0034_m_000001	SUCCEEDED	Tue Nov 28 16:28:38 -0800 2017	Tue Nov 28 16:29:34 -0800 2017	56sec	Tue Nov 28 16:28:38 -0800 2017	Tue Nov 28 16:29:34 -0800 2017	56sec	
Map tasks		ID	State	Start Time	Finish Time	Elapsed Time	Start Time	Finish Time	Elapsed Time	
Reduce tasks		Showing 1 to 2 of 2 entries								

Figure 4 Map Task Running Time for Cornellbox

Map task running time for **cornellbox** rendering(Up: Original, Down: Ours)

We can see that in our method, both computing node finish the job about in the same time, with maximal utilization of both computing nodes, that is the workload division is balanced.

Conclusion

In this project, we did: Setup a Scalable Hadoop cluster, Research on the ray tracing algorithm and Optimize the workload balance.

For future scope a better division algorithm for the workload by sampling can be made and a use of mixture of GPU and CPU can be done. First the image pixel can be sampled and divide into diffusion pixels and ray tracing pixels, then GPU can handles diffusion pixel by simple rasterization, while CPU cluster can handles the ray tracing pixels.

References:

1. <https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/how-does-it-work>
2. https://docs.hortonworks.com/HDPDocuments/Ambari-2.6.0.0/bk_ambari-installation/content/review.html