Ray Tracing in the Cloud using MapReduce

Lesley Northam Khuzaima Daudjee Rob Smits Joe Istead
University of Waterloo
{*lanortha, kdaudjee, rdfsmits, jwistead*}*@uwaterloo.ca*

***Abstract*—We present the Hadoop Online Ray Tracer (HORT), a scalable ray tracing framework for general, pay-as-you-go, cloud computing services. Using MapReduce, HORT partitions the computational workload and scene data differently than other distributed memory ray tracing frameworks. We show that this unique partitioning significantly bounds the data replication costs and inter-process communication. Consequently HORT is fault-tolerant and cost-effective when rendering large-scale scenes (i.e., scenes that do not fit into local memory) without specific or dedicated high performance infrastructure. Our experiments demonstrate this scalability and fault tolerance using several CPU and GPU instances on Amazon AWS with the Hadoop open-source implementation of MapReduce.**

***Keywords*-MapReduce, rendering, cloud computing**

I. Introduction

Ray tracing is a high-quality rendering technique with many applications–it's used by film production studios to create movies like *Avatar*, by artists to create digital works of art, and by medical professionals to visualize CT scan data [1], [2]. The computational workload is highly parallelizable, and there exist several distributed algorithms that take advantage of specially constructed, high performance compute clusters to reduce render time [3], [4], [5]. Even on these fast, distributed ray tracers, the runtime bottlenecks due to rendering large scale scenes become prohibitive when the scenes no longer fit into memory. Furthermore, owning and maintaining a high performance compute cluster can be prohibitively expensive [6]. For these users, it's more cost effective to rent time on general infrastructure as-a-service (IaaS) offerings like Amazon's Elastic Compute Cloud (EC2) [6], [7]. Therefore, we created the Hadoop Online Ray Tracer (HORT), a scalable, distributed ray tracing framework designed for general, pay-as-you-go cloud computing services.

Distributed ray tracing algorithms follow either a shared or distributed memory model. Shared memory algorithms store scene data in a central location from which workers request data on demand for processing. This model allows data acceleration structures, such as bounding volume heirarchies and kd-trees, to be used. These structures enable efficient rendering of small scenes, but they cannot be used effectively for large scale scenes because structure generation time and disk paging is costly [2].

The alternative approach, distributed memory ray tracing, partitions scene data between workers. When a worker discovers that data required for processing a ray is not stored locally, it either requests the data from, or forwards the ray to another worker [3], [4]. The volume of data requests when rendering a large scene causes load balancing problems and network saturation [2]. Some approaches give each worker an entire copy of the scene to reduce inter-process communication, but the initial replication and on-going network bandwidth is still costly for large scenes.

HORT is a distributed memory ray tracing framework that leverages Hadoop, an open-source implementation of MapReduce, a parallel programming paradigm to partition tasks and scene data between workers [8]. Unlike other distributed memory approaches, HORT has a constant number of inter-process calls, making it suitable for rendering large scale scenes. Hadoop also provides HORT with desirable properties such as fault tolerance and automatic handling of underlying network and cluster configuration.

We implemented two versions of HORT; one that uses the CPU for rendering, and one that uses CUDA-enabled GPUs. We demonstrate HORT's scalability on Amazon's Elastic Compute Cloud (EC2) environment using both implementations in a variety of configurations. The results show that HORT reduces the data replication and inter-process communication bottlenecks, which in turn reduce the rendering time and enable the rendering of large scale scenes. The results also show that Hadoop's fault tolerance can reduce costs in the presence of nodal failure, and that the HORT rendering time overhead is low.

Section II discusses the ray tracing workload and the MapReduce platform interface. After explaining the design considerations faced by distributed memory ray tracing on general purpose cloud computing infrastructure, Section III presents the HORT framework which overcomes those challenges. Section IV evaluates the performance of two HORT implementations. The related work and conclusions are presented in Sections V and VI, respectively.

II. Background

In this section, we present an overview of ray tracing and MapReduce.

A. Ray Tracing

Ray tracing is a rendering technique that traces rays of light originating from a virtual camera as they pass through the image plane and interact with scene objects and light

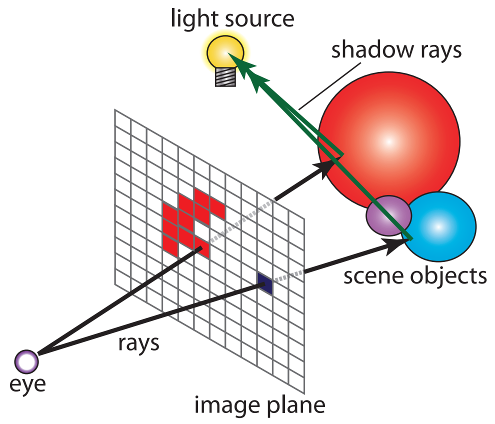


Figure 1. The basic ray tracing algorithm. Note that at least one ray is generated for every pixel in the image plane, but shadow rays are only generated when the initial ray intersects a scene object.

sources. When rays intersect with scene objects, additional rays are generated to compute shadows and determine the final color of each image pixel (Figure 1).

These algorithms are computationally expensive—at least one ray is generated per image pixel, and each of these rays must be tested against every polygon to find the physical intersection [9]. In the worst case, the algorithm performs $O(whkl)$ of these ray-object intersection tests for an image with pixel resolution (w, h) and a scene with k objects and l light sources, However, the number of intersection tests and execution time can be reduced by using kd-trees and bounding volume hierarchies to restrict the search space of each ray.

Ray tracing algorithms are often parallelizable because ray-object intersection tests are independent. This embarrassing parallelism is exploited by many distributed ray tracing algorithms to reduce render time [10]. However, performance bottlenecks still arise from large scene sizes and there are few techniques that address this concern [2].

B. *MapReduce*

MapReduce is a programming paradigm and framework for batch processing large sets of data over a cluster of commodity machines [11].

A MapReduce program's input and output is of the nature $map(list(k_1, v_1)) \Rightarrow list(k_2, v_2)$, and $reduce(list(k_2, v_2)) \Rightarrow list(v_2)$, where k_i and v_i refer to keys and values from domain i. The input keys and values that any map worker receives is drawn from a different domain than its output [11]. Further, a map worker's output becomes input to a reducer, which reduces this input to a list of values from the same domain. MapReduce does not enforce this type of data relationship, but the model is useful when considering how a problem can be defined in terms of MapReduce.

Users executing a MapReduce job will supply a large input set for map workers. The MapReduce framework instantiates many map workers across a number of computers and divides input among them. The map workers' output is sorted and similarly divided up for many reduce workers.

The MapReduce implementation as described by Dean et al. is a proprietary system. Therefore, we implement HORT using Hadoop, an open source implementation of MapReduce [8].

III. Design

Previous distributed ray tracing algorithms (Section V) often rely on dedicated high performance infrastructure–in particular, specialized network topologies–to mitigate bandwidth bottlenecks caused by inter-process communication. Consequently, these algorithms are unsuitable for implementation on general purpose cloud computing resources such as MapReduce and IaaS. However, a distributed memory model that partitions scene data between workers can still be implemented for MapReduce provided that inter-nodal data dependencies are avoided. This section describes how HORT accomplishes that using multiple MapReduce stages.

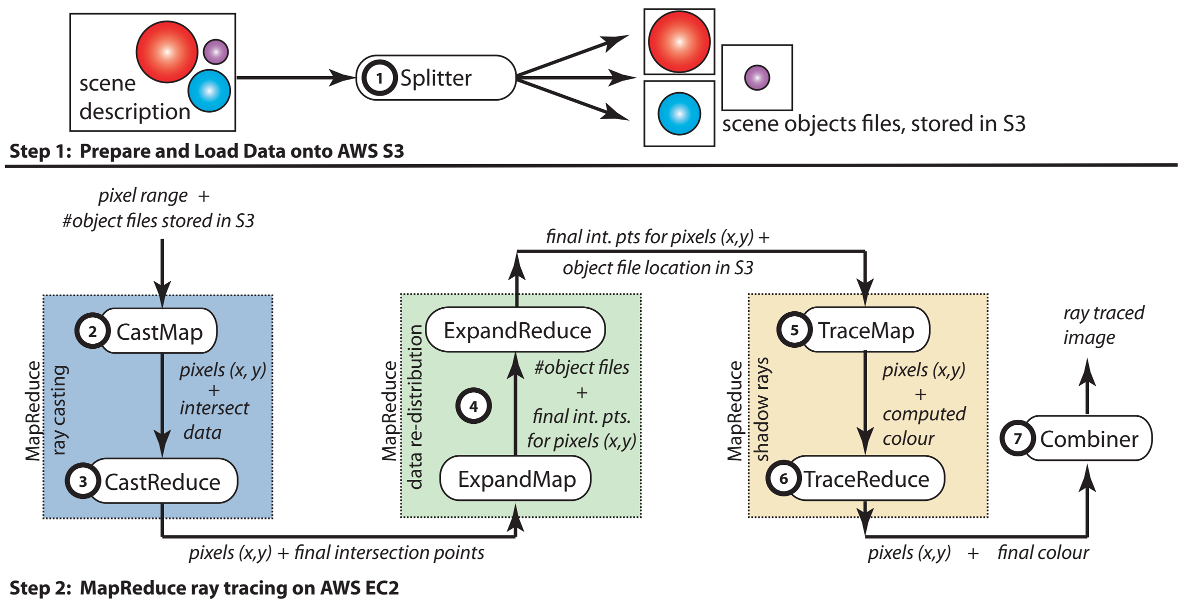
A. *Ray Tracing with MapReduce*

There are many ways that ray tracing can be phrased using MapReduce. A simple but limited method is to let the input to each Map worker be a subset of pixels (i.e., rows x to $x + j$) and a copy of the scene. Each Map worker performs the complete ray tracing algorithm on all input pixels. Workers output a data stream of pixel colors that are collected by a Reduce worker, which outputs the final image. This implementation can take full advantage of data acceleration structures but is inefficient for large scenes. Added execution costs arise from data replication and transfer, increased disk access, and acceleration structure generation [2].

HORT overcomes the limitations of the simple method (i.e., data replication and increased disk access) by splitting scene data into memory-manageable chunks. Then, HORT uses multiple MapReduce stages to perform the actual ray tracing calculations. Algorithm 1 lists MapReduce-specific HORT code while Figure 2 depicts the HORT workflow for a single instance on Amazon AWS. The numbers in Figure 2 correspond to those in Algorithm 1.

There are three distributed components to HORT, each implemented as a MapReduce job: *Cast*, *Expand*, and *Trace*. The Cast step is the first MapReduce job, and performs ray casting calculations. The Expand step builds the input for the Trace step, which calculates object shading. Next, we expand on each of the steps from Algorithm 1.

1) Splitter: The first step in HORT's workflow is to split a scene S containing n polygons into k scene object files (G_i represents one scene object file), where $k \leq n$. To reduce data replication, k should equal the number of workers. Our implementation is simple and divides the n polygons evenly amongst workers with no consideration given to



scene description
1 Splitter
scene objects files, stored in S3
Step 1: Prepare and Load Data onto AWS S3
pixel range + #object files stored in S3
MapReduce ray casting
2 CastMap
pixels (x, y) + intersect data
3 CastReduce
pixels (x,y) + final intersection points
final int. pts for pixels (x,y) + object file location in S3
MapReduce data re-distribution
ExpandReduce
4
#object files + final int. pts. for pixels (x,y)
ExpandMap
MapReduce shadow rays
5 TraceMap
pixels (x,y) + computed colour
6 TraceReduce
pixels (x,y) + final colour
7 Combiner
ray traced image
Step 2: MapReduce ray tracing on AWS EC2

Figure 2. HORT workflow on Amazon AWS.

Algorithm 1 Basic HORT Framework. Note that statements in for-all loops are parallelized using Map or Reduce operations.

```
Let $S$ be a scene with $n$ polygons and resolution $w \times h$
Let $k$ be the number of machines or workers
(1) Splitter($S$, $k$): Divide the $n$ polygons of $S$ into $k$ groups, $G_k$, with $\frac{n}{k}$ polygons each
for all $G_i$ where $0 \leq i \leq k-1$ do
    (2) CastMap($G_i$): Ray cast sub-scene $G_i$ and produce intersection point list $L_i$
end for
for all $L_i$ where $0 \leq i \leq k-1$ do
    (3) CastReduce($L_i$): Collect intersection points from lists $L_i$ and produce final ray-object intersection point list $L'$
end for
(4) Expand and Re-distribute intersection points $L'$ and sub-scenes $G_i$ for next phase
for all $G_i$ where $0 \leq i \leq k-1$ do
    (5) TraceMap($G_i$): Compute shadows using scene $G_i$ and produce pixel colour list $C_i$
end for
for all $C_i$ where $0 \leq i \leq k-1$ do
    (6) TraceReduce($C_i$): Collect pixel colours from lists $C_i$ and compute final pixel colours $C'$
end for
(7) Combine($C'$): Convert list of pixel colours into final image
```

position; with a runtime of $\theta(n)$ though other more data-aware algorithms may be used.

2) CastMap: The CastMap step performs ray casting, which determines the initial ray-polygon intersection points. First, the raw data used as input to the Cast MapReduce job must be prepared.The simplest method is to enumerate all possible pixel-scene subset G_i combinations. However, this would produce a prohibitively large input stream. Therefore, an input stream of d integers (d is a user-specified parameter, where $k \leq d \leq 100 \times k$) is created. The stream of d integers is evenly divided across Map workers and maps to a range of coordinates and scene split URLs; otherwise, if $d < k$, data will not be evenly distributed and replication may occur. After input generation, each CastMap has a list of numbers representing a span of sequential (x, y) pixel coordinates and a URL to a scene object file, G_i, which is a subset of the scene. The CastMap task then computes the ray-polygon intersections for the list of coordinates and objects G_i. Each CastMap worker returns a list of (x, y) values corresponding to scene polygon intersection points. Only inputted coordinates that intersect with any assigned scene objects G_i have corresponding output values.

3) CastReduce: Prior to executing the CastReduce step, the MapReduce platform collects all intersection points output by the CastMap workers, sorts them according to coordinate, then distributes the sorted points to the CastReduce workers. For each coordinate that has more than one scene object intersection, CastReduce determines which of these objects is closest to the "camera" or "eye". The object whose intersection is nearest to the eye is the object which will appear at this coordinate in the final image. Since the input to each CastReduce worker is in sorted order, the runtime of this operation is $\theta(m_i)$ where m_i is the size of the list given to each worker as input.

4) ExpandMap and ExpandReduce: The purpose of the Expand step is to prepare the CastReduce output for the TraceMap input. Specifically, for each intersection point from the CastReduce output, Expand will output a line that prepends this information with each scene split URL. Once again we want to avoid the situation where each worker needs to access the same, potentially very large, scene split. Expand allows HORT to continue to divide parts of a scene among workers, minimizing the number of distinct splits a worker will receive using the same input generation mechanism as CastMap.

5) TraceMap: Trace performs the secondary ray calculations which compute shadows and light contributions as described in Section II. TraceMap receives the output of ExpandMap which includes a coordinate, an object intersection point, and a specific scene split URL. TraceMap traces rays from the input intersection points to all light sources and for each point produces a list of colors (C) that represents the contributions of each light source to that coordinate's color. Similar to the Cast step, between TraceMap and TraceReduce, the output data is sorted and divided amongst TraceReduce workers.

6) TraceReduce: TraceReduce combines color data calculated by TraceMap workers to determine the final color of each pixel. For each pixel p, TraceReduce collects all colors C output by TraceMap, and determines the actual contribution of each light. If the list of possible contributions of a light consists of m colors, the actual contribution is the darkest one. This is because if a pixel is shadowed by a scene object, the color will be darker than if that pixel had not been in shadow. The final pixel color is then computed as the sum of all final light contributions and the scene's ambient light. TraceReduce outputs coordinates and corresponding final color values. Again, since the input to TraceReduce is sorted, the runtime is $\theta(n)$.

7) Combiner: Each TraceReduce worker outputs a list of pixel colors which define a part of the desired output image. The final step in HORT's workflow, the Combiner, collects all coordinate-color pairs output by the TraceReduce jobs and inserts these values into the final image. The Combiner is not a MapReduce job, and can run on a single machine. If there are f total coordinate-color pairs, the Combiner has a runtime of $\theta(f)$.

Note that inter-process communication only occurs between Map and Reduce phases, individual workers are unware that others exist.

To phrase other ray tracing operations in HORT, create a Map operation to perform the computation on a subset of available data, and a Reduce operation that combines Map worker results into a final answer.

B. HORT and GPUs

GPUs are specialized hardware with hundreds of compute cores that can be used for parallel numeric computations. Since Amazon (and other IaaS providers) have GPU instances available for MapReduce jobs, we constructed Cast and Trace MapReduce jobs for the HORT framework that utilize these instances. We note that while OptiX [12] could be used to implement the ray tracing portions of our CUDA Map and Reduce jobs, we chose not to use it so our CUDA implementation resembles the non-CUDA implementation.

Offloading work from the CPU to the GPU introduces latency as data must be passed from RAM to the GPU's global memory. While GPUs have high memory bandwidth, the frequency of memory transfers can negatively impact performance. Additionally, the number of GPU calls can negatively affect performance if the amount of work done by the GPU per function call is small. Therefore, our design of CastMap and TraceMap steps minimizes both GPU calls and memory transactions.

C. Reducing Data Replication and Data Transfers

HORT reduces data replication by using a distributed memory model, where scene data is partitioned between workers instead of replicated at each worker. The Map phases of HORT perform traditional ray tracing operations on the partition of pixel and scene data. The Reduce phases identify the correct per-pixel result from the list of all potential candidates output by the Map phase. The Reduce phases eliminate the need for Map phase workers to communicate with each other, thus reducing the number of data transfers. Precisely, our implementation of HORT makes a constant number of data transfers, as shown in Figure 2.

With the exception of copying scene partitions to CastMap workers, the size of each transfer is bounded by the number of pixels in the image. For example, given two CastMap workers each operating on a pixel range of $(0, y) \times (0, y)$ where one worker has polygons $(0, m)$ and the other worker has polygons $(m+1, n)$, each worker will output a maximum of y^2 intersection points — one for each pixel. The CastReduce phase recieves these intersection points, sorted by pixel value, and outputs the intersection point nearest to the eye for each pixel, which has a maximum size of y^2. Some ray tracing implementations use more than one ray per pixel, thus if r per pixel are sent, then the size of each call is bounded by $O(ry^2)$. Additionally, since data is transferred in blocks, instead of individual ray forwards [3], or triangle requests, compression algorithms can be utilized to effectively reduce message size.

D. Interacting with AWS

HORT is comprised of Python scripts and C++ programs. The Splitter, Cast, Trace, and the Combiner are entirely implemented in C++ and utilize the OpenMP parallelization

64-bit Instance	API Name	Memory	Virtual Cores	EC2 Compute Units
Large	m1.large	7.5 GB	2	4
High-CPU Ex. Large	c1.xlarge	7 GB	8	20
Cluster GPU Quad. Ex. Lg.	cg1.4xlarge	22GB	8+GPU	33.5+GPU

Table I
AMAZON EC2 INSTANCES USED TO EVALUATE HORT

library for loop paralellization.The Input Generator, Expand, and AWS Orchestrator programs are Python scripts.

Our implementation of HORT interacts with AWS in two ways. First, the scene object URLs described above are Amazon Simple Storage Service (S3) URLs. The CastMap and TraceMap binaries make use of `libaws` to fetch object files from S3 [13].

The AWS Orchestrator script prepares and creates the AWS Elastic MapReduce (EMR) workflow and also executes the local portions of HORT. Specifically, the Orchestrator uploads scene object pieces and MapReduce programs in S3. It creates and monitors an EMR task, and upon successful completion it grabs the TraceReduce output for the Combiner. The Orchestrator interacts with AWS through the Python library called *boto* [14]. While our implementations target AWS and EMR, they could execute on any commodity Hadoop cluster with very minor changes.

IV. PERFORMANCE EVALUATION

We evaluated HORT's scalability and fault tolerance using Amazon Elastic Compute Cloud (EC2) with Elastic MapReduce (EMR). We constructed several ray tracing implementations with HORT to evaluate different scenarios:

1) (a) A traditional single-executable ray tracer [TRAD], and (b) a single executable CUDA-enabled version [TRADG].
2) (a) a MapReduce ray tracer that divides input based strictly on ranges of coordinates [MR1], and (b) the same ray tracer but with CUDA capabilities [MR1G].
3) (a) a MapReduce ray tracer that divides input based on both ranges of coordinates and split pieces of the scene [MR2], and (b) the CUDA-enabled version [MR2G]

Ray tracers (2) and (3) differ only in their handling of input, as (3) contains the mechanisms for handling scene splits. We will use the indicated labels to differentiate the implementations.

We also used several configurations of the EC2 instances listed in Table I to test HORT's scalability. We constructed several input scenes with 5804 to 742912 polygons and resolutions of HDTV $(1920, 1080)$ and 4K $(4096, 2304)$. These scenes are similar in size to those in [12]. The very high 4K resolution is rarely used for testing, however, an increase in the availability of 4K consumer devices makes this resolution a desirable test resolution for our experiments. Note that higher polygon counts and 4K resolution are used only in our GPU experiments as the price-per-experiment is significantly less for the high computation needs of the tests. For each experiment, the total rendering time was recorded in seconds. Finally, we ran each experiment three times and averaged the results.

We demonstrate the scalability of the general algorithm on both GPU and CPU instances. We also demonstrate that reducing data replication reduces total execution time, and we quantify the overheads associated with HORT and the effect of machine faults.

A. Scalability

We tested the scalability of our coordinate range-based MapReduce with the CUDA-enabled ray tracing framework [MR1G] and the CPU framework [MR1]. We used two, four and eight GPU EC2 instances with the CUDA-enabled HORT [MR1G], and four, eight, and sixteen Extra Large EC2 instances with [MR1]. We expected that doubling the number of instances would reduce execution time by a factor of two as the number of polygons in the scene increased. Figure 3 demonstrates how doubling the number of GPU instances affects render time for scenes with a 4K resolution $(4094, 2304)$. Note that for higher polygon counts, doubling the number of instances halves the render time.

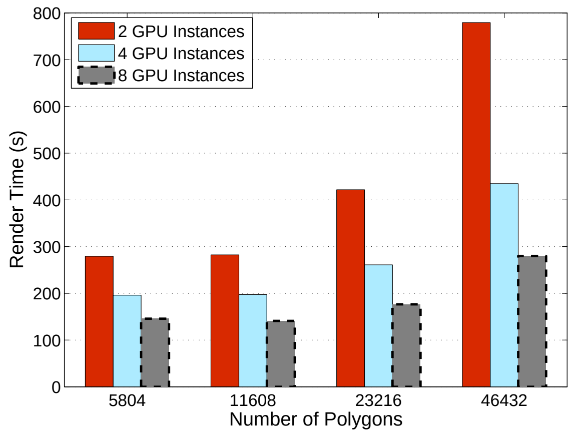


Figure 3. Total rendering time for two, four and eight GPU instances with 4K resolution. Note how doubling the number of instances reduces the rendering time.

Figure 4 presents the total rendering time for HDTV resolution scenes when [MR1] and *Extra Large High-CPU* EC2 instances are used. Note how increasing the number of instances decreases the rendering time in a predictable manner.

We note that continuously doubling the number of instances (CPU or GPU) will not continuously halve the render time since at some point, the amount of computation executed by each worker will be less than the overhead of HORT. This behaviour was observed with lower resolution

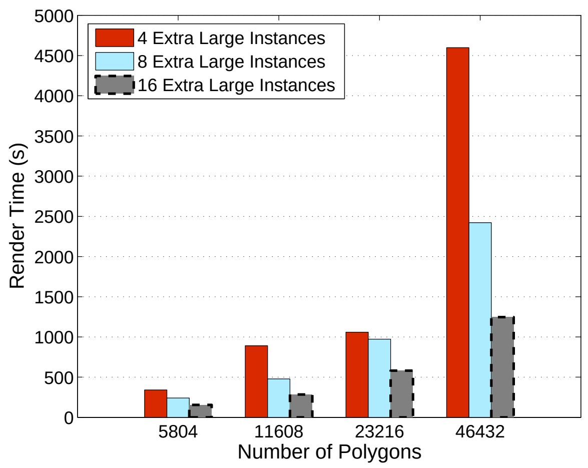


Figure 4. Total rendering time for four, eight, and sixteen extra large instances with HDTV resolution. Note how doubling the number of instances reduces the rendering time by half as image resolution and polygon count increase.

images and polygon counts, and the plots are not included due to space constraints.

In our experiments, each instance hosted one Map or Reduce worker, even though it is possible to have more than one worker per instance. We tested the effect of increasing the number of workers per instance using sixteen *Extra Large High-CPU* EC2 instances with one, two, four, and eight workers per instance. We expected that increasing the number of workers per instance would not continuously improve performance because our Map and Reduce tasks utilize OpenMP for local parallelization, thus, the number of threads would exceed the number of cores, causing increased context switching and execution latencies. The results of our experiment confirmed that when the number of workers per instance increased, the performance decreased. Experimentally, we found that the number of workers per instance should not exceed the number of cores. Logically, this will also reduce the occurrence of context switching.

B. Data Replication

One of the primary goals of HORT is to reduce data replication, and therefore reducing time/cost. Our ray tracing implementations [MR2] and [MR2G] target this goal, differing from [MR1] and [MR1G] only by how input is handled. [MR2] and [MR2G] divide input based on both coordinate range and scene splits, therefore, each worker receives only a subset of the scene. We expected that the amount of time saved by reducing the amount of data replication would increase with the number of polygons, and that for scenes with few polygons, time may not be saved at all. To test our hypothesis, we compare the rendering time when one and eight replicas of the scene data are made. For this experiment, we use eight GPU EC2 instances with our scene-splitting ray tracer [MR2G] with one worker per instance. Figure 5 demonstrates the effects of scene splitting on render time when eight replicas of the scene are made versus one. Note how reducing data replication reduces rendering time by $50 - 80\%$. Next, we tested the

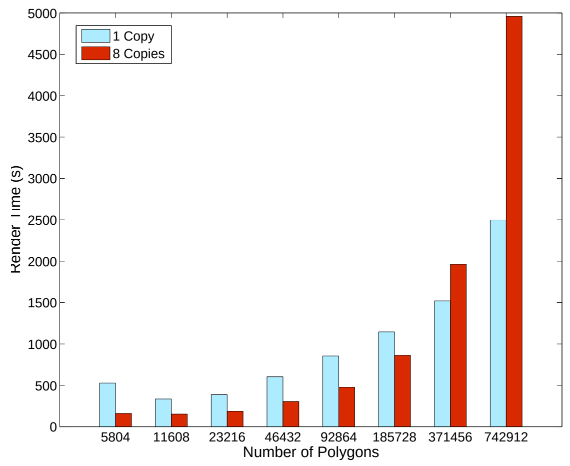


Figure 5. The effect of reducing the number of scene data copies on rendering time when eight GPU instances are used with a 4K resolution. Note how reducing data replication increases performance, but these savings are only notable for higher polygon counts.

effect of splitting scene polygons across workers for *Extra Large High CPU* EC2 instances. Figure 6 shows the total rendering time when there are no data replications (1 copy of scene data), and when data is replicated (32 copies of scene data). As expected, the time savings are significant for large polygon counts.The same test was executed with sixteen instances with similar results.

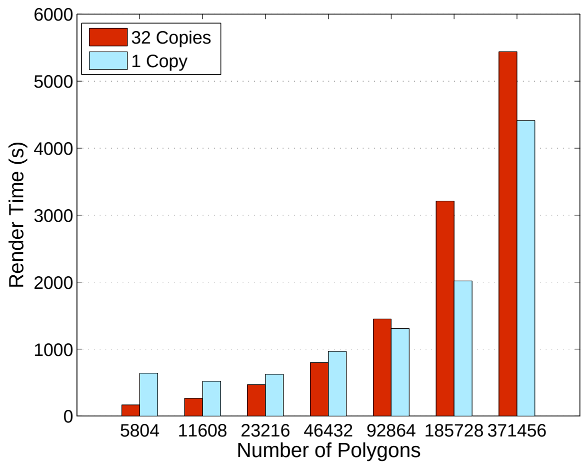


Figure 6. The effect of reducing the number of scene data copies on rendering time when thirty-two extra large instances are used with HDTV resolution. Note how reducing data replication increases performance, but these savings are most notable for higher polygon counts.

C. Overhead

We expected that HORT would have higher overheads than tradititional single-executable ray tracers because

HORT is a multi-stage MapReduce algorithm. This is not only because data needs to be moved between the multiple executables, but data structures have to be recreated in memory each time. We measured this overhead by comparing our traditional ray tracing implementations [TRAD] and [TRADG] to HORT [MR1] and [MR1G] using the same instance. The overhead for HORT on GPU instances, which have a higher rendering throughput, is less than 20% at a resolution of 4K with forty-six thousand polygons. With higher polygon counts, we project that this overhead will reduce to less than 5% beyond one hundred and eight-five thousand polygons at a resolution of 4K. We also found that for CPU instances and both SDTV and HDTV resolutions, the overheads associated with HORT become insignificant beyond twenty-three thousand polygons.

D. Fault Tolerance

Ray tracing failures, such as those caused by segmentation faults or hardware failures are costly because they require analysis and repair prior to re-starting the rendering job. A scene that takes k hours to render, but fails has a total rendering time of at least $2k$ if fault tolerance is not built into the ray tracer. HORT has built-in fault tolerance because Hadoop has fault tolerance — no additional effort is required by the developer or user. We verified HORTs fault toleranance by creating a set of ray tracing job over sixteen *Extra-Large High CPU* EC2 instances, then randomly selecting an instance and terminating it (this can be done through a web interface). All jobs completed with correctly rendered images at an added cost of 60% of the total rendering time (when no faults occur).This cost is a fraction of the time it would take to finish the job if re-started from scratch.

V. Related Work

There have been several distributed ray tracing proposals. For example, Cleary et. al. present a method which divides polygonal scene data across an array of machines. Rays pass between machines similar to traversing the 3D space, and transmit with them a list of intersection data thus far [15]. Kim and Kyung present a similar solution based on a ring-arrangement of processors. This framework distributes scene data over nodes, and passes rays with acquired intersection data cyclically around the ring until the entire scene has been traced [16]. Kobayashi et al. use a 6-dimensional hypercube arrangement to intelligently pass rays to neighboring regions while bypassing those nodes which the ray could not possibly intersect [17]. These early methods reduce scene data replication, however, there is significant inter-process communication exhibited as rays which are used to transmit intersection and shading data between network nodes can end up traveling to every node, and carry ever increasing amounts of data with them. Execution time is dominated by the longest ray — the ray(s) which travel through the most nodes. Additionally, none of these algorithms exhibit fault tolerance—if one node fails, the ray tracing job fails to complete.

More recently, Pharr et. al, presented a data-driven distributed ray tracing framework which divides scene data into voxels and queues rays at voxels until a certain threshold is reached before loading and processing that voxel [3]. Navratil et. al, recently expanded Pharr's work by adding a ray scheduling algorithm to significantly improve performance when rendering large scale scenes [2]. Wald et. al, constructed a similar distributed ray tracer which divides scene data using a high-level BSP-tree of small, self-contained voxels [4]. Workers process a subregion of the image and acquire new voxels from a central server when rays require them. To reduce latency, only the worker rays which request the voxel must wait for data transmission — other rays executing on the same worker are not stalled. These ray tracing frameworks use a single, central server to store scene data. When many workers attempt to retrieve data from a central server, bottlenecks can arise due to network bandwidth restrictions, or other platform restrictions. Also, these methods can lead to worker retrieving a copy of the entire scene over time. For these data-driven frameworks, data could be distributed across workers instead of using a central server. When data is needed, a worker would need to acquire it from the owner instead of the central server. However, this requires that each worker is aware of data ownership, and can quickly communicate with all other workers. Additionally, these worker data requests can interfere with local computations.

A paper by Green and Paddon presents a caching structure for ray tracing data to reduce the frequency of requiring data from a neighboring machine. Their method is adaptable to a variety of ray tracing algorithms, but still requires inter-process communication — which, for large scenes would be costly [5].

Using a GPU for ray tracing can improve distributed performance [12]. However, since the GPU has limited memory, large scenes cannot be cached entirely in GPU memory. When data which is not in the GPU's memory space is needed, it must be transferred from the system's main memory. This transaction introduces latencies, especially for large scenes where many memory transactions are needed. Our ray tracing framework decreases the number of required memory transactions since each GPU worker receives only a portion of the scene data.

These ray tracing frameworks also require significant investments in hardware and software configuration. Additionally, as presented, none of them exhibit any form of fault tolerance. Our MapReduce framework requires minimal configuration, is easily scaled, and is fault tolerant by nature — that is, neither user or programmer need to handle or implement a fault tolerance method as we leverage the mechanism within Hadoop. Moreover, we demonstrate that our MapReduce framework scales to multiple GPUs.

MapReduce has been applied to other rendering operations. Stuart et. al. presented a multi-GPU volume rendering framework which uses MapReduce to handle task distribution [18]. They divide the scene into blocks, and distribute them across GPUs to be rendered with the assumption that scene blocks can fit into GPU memory. Their pipeline is short since volume rendering performs only ray casting, and only a single MapReduce operation is needed. Each worker renders an image of the information in the provided voxel, and when workers finish, the small images are composited into the final image. Their method works for ray casting because no additional rays need to be sent to compute object shading, but will not work for ray tracing or other high-quality rendering algorithms such as photon mapping. Our ray tracing framework sends shading rays and can be easily adapted to other rendering algorithms such as photon mapping.

VI. Conclusions

In this paper we presented a new distributed ray tracing framework called HORT that is designed for rendering large scenes on IaaS cloud offerings. HORT is hardware agnostic and fault tolerant, and utilizes the popular MapReduce programming paradigm to divide work across machines in a cloud environment. We demonstrated that HORT scales to different EC2 configurations, and for large scenes, doubling the number of instances reduces execution time by half. We also demonstrated the agnostic nature of the HORT framework by using the CUDA-enabled GPU instances within the EC2 cloud. We noted that this implementation of HORT shares the same scalability and data savings as our non-CUDA implementation.

HORT's design eliminates the need for data replication while reducing the overhead for performing any data-on-demand requests during computation. We tested HORT with Hadoop on the Amazon EC2 cloud and found that HORT generates savings of $50 - 80\%$ of the total render time by reducing data replication. Finally, we demonstrated that HORT is fault tolerant — when a cloud instance fails, the image rendering completes correctly and in less time than restarting the ray tracing job.

Acknowledgements

We wish to thank Amazon for their generous research grant of AWS computing resources. We also wish to thank Gavin Miller at Adobe Systems for his valuable advice and discussion.

References

[1] J. Rath, "The data-crunching powerhouse behind 'avatar'," http://www.datacenterknowledge.com/archives/2009/12/22/the-data-crunching-power, December 2009.

[2] P. A. Navrátil, D. S. Fussell, C. Lin, and H. Childs, "Dynamic scheduling for large-scale distributed-memory ray tracing," in *EGPGV*, 2012, pp. 61–70.

[3] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan, "Rendering complex scenes with memory-coherent ray tracing," in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '97, 1997, pp. 101–108.

[4] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive distributed ray tracing of highly complex models," in *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, 2001, pp. 277–288.

[5] S. Green and D. Paddon, "Exploiting coherence for multiprocessor ray tracing," *Computer Graphics and Applications, IEEE*, vol. 9, no. 6, pp. 12 –26, nov. 1989.

[6] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[7] "The cloud crowd," *The Economist*. [Online]. Available: http://www.economist.com/node/21564259

[8] D. C. A. Bialecki, M. Cafarella and O. O'Malley, "Hadoop: a framework for running applications on large clusters built of commodity hardware," http://lucene.apache.org/hadoop, 2005.

[9] D. Hearn and M. P. Baker, *Computer graphics (2nd ed.): C version*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[10] I. Notkin and C. Gotsman, "Parallel progressive ray-tracing," *Computer Graphics Forum*, vol. 16, no. 1, pp. 43–55, 1997.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[12] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, pp. 66:1–66:13, July 2010.

[13] 28msec Inc., "LibAWS: A C++ Library for Interfacing with Amazon Web Services," http://aws.28msec.com, 2009.

[14] M. Garnaat, "boto: Python interface to Amazon Web Services," http://boto.cloudhackers.com/, 2011.

[15] J. G. Cleary, B. M. Wyvill, G. M. Birtwistle, and R. Vatti, "Multiprocessor ray tracing," *Computer Graphics Forum*, vol. 5, no. 1, pp. 3–12, 1986.

[16] H.-J. Kim and C.-M. Kyung, "A new parallel ray-tracing system based on object decomposition," *The Visual Computer*, vol. 12, no. 5, pp. 244–253, 1996.

[17] H. Kobayashi, T. Nakamura, and Y. Shigei, "Parallel processing of an object space for image synthesis using ray tracing," *The Visual Computer*, vol. 3, pp. 13–22, 1987.

[18] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens, "Multi-gpu volume rendering using mapreduce," in *HPDC*, 2010, pp. 841–848.