An Improved Monte Carlo Ray Tracing for Large-Scale Rendering in Hadoop

Rui Li
Key Laboratory for Embedded and Network Computing of Hunan Province
Hunan University
ChangSha, China
rui@hnu.edu.cn

Yue Zheng
Key Laboratory for Embedded and Network Computing of Hunan Province
Hunan University
ChangSha, China
alexking1987@gmail.com

***Abstract*—To improve the performance of large-scale rendering, it requires not only a good view of data structure, but also less disk and network access, especially for achieving the realistic visual effects. This paper presents an optimization method of global illumination rendering for large datasets. We improved the previous rendering algorithm based on Monte Carlo ray tracing and the scheduling grids, and reduced the remote reads by slightly organizing the original data with considerations of locality and coherence. We implemented the rendering system in a Hadoop cluster of commodity PCs without high-end hardware. The large scene data are processed in splits by MapReduce framework, which increases scalability and reliability. The result shows that our algorithm of scheduling rays for each data split fits with large-scale scene and takes less reads and rendering time than previous works.**

Keywords-Monte Carlo ray tracing; large-scale scene; scheduling grids; Hadoop;

I. INTRODUCTION

The demand for realistic rendering increases rapidly in many fields. As the lighting condition is various and the modeling becomes more detailed, creating a realistic looking image is trivial, because the simulation of light is very complicated, and the scene data could be too large to handle in a single workstation. Many studies focused on the efficient rendering of visual effects or the implementation of large-scale scene display, but few researches have addressed the problem which combines the both.

In realistic rendering, Monte Carlo ray tracing [1] is an unbiased and most accurate algorithm of global illumination solution. The points on surface are shaded by integrating the incoming light over the hemisphere around the point. A large number of rays are sampled to estimate the illumination, which is very time-consuming. Many efficient algorithms traced the rays in parallel, e.g. using CUDA GPUs [2], but they assumed that each ray accesses the entire scene on demand to test for the nearest intersection. If the scene data exceed the memory, most of these algorithms will be inefficient or infeasible.

Although the scene can be divided and displayed in distributed memory systems [3], the global illumination that affects the objects of each other makes it hard to partition for independent tasks. Building acceleration structures can only load the scene records related to the search space of the ray being processed, but there is no efficient caching method for it. The number of ray is huge and the scene data are replaced frequently, leading to performance bottlenecks.

Another problem is that the tasks in recent large-scale rendering system depend on each other a lot [4], which is potentially lack of reliability. Moreover, they implemented the system in a cluster of high-end processors, which is expensive to own and difficult to maintain for common users. It is more reasonable to turn to cloud computing because of the high usability and cost-effective it affords, and the rendering workflow must be redesigned for the new framework as well.

In this paper, we improved the Monte Carlo ray tracing for large-scale scene and implemented the system in Hadoop [5], a reliable big data processing platform that can also be deployed in the cloud. The scene data stored in Hadoop Distributed File System (HDFS) are divided into equal-sized blocks and distributed randomly among several DataNodes in the cluster for loss prevention. Despite the scene records may be written in an unknown order, our method can schedule all related rays for any scene partition, and process the records as close to where the data physically resides as possible, which reduces network transmission significantly. The spatial coherence of scene data is increased by slightly organizing the original data, thus the nearest intersection test can be simplified with improvement in performance.

II. RELATED WORK

Ideally, the working set of rays would reside in the same memory without fetching any data from others during the process. DeMarle et al. increased available memory by building software shared memory layers for the cluster [6], which made the gory details of data fetching are transparent to the rendering tasks. However, it still has the problem of memory contention and threshing if the scene data and its acceleration structure grow larger than the aggregate memory of the cluster. Kato et al. presented Kilauea [7], a global illumination rendering system for extremely complex and large scenes. They loaded the all the working set once and passed the ray data to every worker through Pthread. This system requires scene data to load entirely in aggregate memory, and it is unclear whether the migration of all rays will cause significant network traffic.

To reduce the working set, Pharr et al. was first to subdivide the region of light propagation into scheduling voxels, and grouped the rays in the same voxel to trace with only a small part of scene data at a time [8]. Navrátil et al. extended this idea to support parallel volume rendering and scheduled the ray groups dynamically [4]. However, the inter-process communication is not bounded in their system,

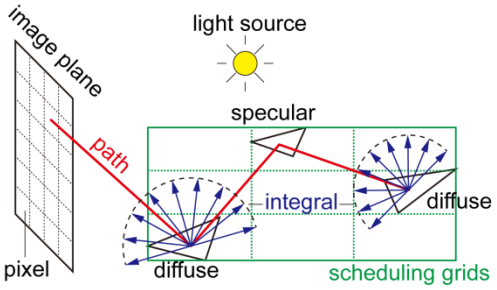

Figure 1. Monte Carlo ray tracing and scheduling grids.

and they ignored the structure and distribution of the actual data, which could be hard to get all related records for a specific voxel in scene space.

Lately, Northam et al. implemented the Hadoop Online Ray Tracer (HORT) using AWS Elastic MapReduce and S3 [9], which provide scalability and fault tolerance. But, they only traced the direct illumination without reflection. It is also unclear whether their algorithm will work well with the extremely large datasets.

III. DESIGN

Our purpose is to build a framework that renders the large scale scene in Hadoop using Monte Carlo ray tracing and MapReduce [10], and limits the search space of rays by acceleration structures. Firstly, it should be easy to map the acceleration structure to the records in the scene file, so the storage structure has to be studied fully. Secondly, the acceleration structure must be small enough not to impact performance when replicating it over processors. Furthermore, as a compute-intensive application, the replacement of the scene data at runtime will slow down the rendering process, especially when fetching the distributed stored data over network. It is better to change the demand driven strategy to process the scene data on their location.

A. Scheduling Grids for Scene Splits

Typically, the block of scene in HDFS is loaded by a Map task as an input split, which can be processed on the local machine without moving over network. It requires a unique determination of the scope of the scene split to schedule related computation. However, the records may be not centralized with close spatial coherence (see Fig. 2), leading to a very large spatial scope for a split. Thus there may be some overlap between splits, and determining the nearest intersection requires loading all these splits. This is because the records are stored in the order they created, if there is a modification, the new record will be appended to the file. Therefore, we map the split into relatively dense grids, which extend the idea of [4].

Our acceleration structure includes two hierarchies. The task hierarchy can be any structure such as Bounding Volume Hierarchies (BVH), because all related data reside in memory at this point. For the global hierarchy, the scheduling grids can adapt with the interleaved order of records. An alternative structure, the bounding box, may lead to excessive overlap, as in Fig. 3.

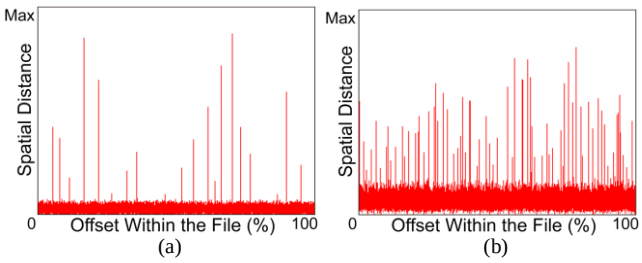

Figure 2. Spatial distance between two consecutive records in CT scan data (a) and Townhouse scene (b).

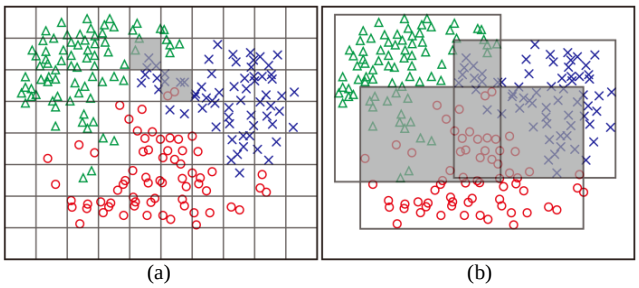

Figure 3. Regular girds (a) vs. bounding box (b). The grey area is overlapped regions.

Before the splits processing, the rays enter the scheduling grids and skip the grid with no objects. The list of grids can be determined according to the ray direction, and the Map task only schedules the rays grouped in grids which the input scene covered. Since there are multiple tasks processing in Hadoop at the same time, we select several scene splits with most rays for each job. The input of Map task consists of scene split and ray groups, which may be from different DataNodes. We ensure the scene data locality as a priority, because the ray data are generated on the fly and then distributed, and the output locations of rays are out of control in the current MapReduce framework.

B. Overlap Reduction

Different granularity of the grids will affect the efficiency of the scheduling. If the granularity is too dense, the acceleration structure may take up more memory, and the ray scheduling will be more complex. For more sparse grids, the number of records in a grid may be greater than a split, and there may be records from different splits in a grid, leading to overlapped grids.

The ideal distribution of data is that a grid contains records from only one split, so that there is no need to reduce the list of potential intersections. However, because of the less spatial coherence of the records, overlapped grids are inevitable. If we organize the entire scene data, the total sort will be very time-consuming and inflexible. Therefore, we use an adaptive strategy to get the least number of overlapped grids, and then organize these parts of data.

The subdivision of grids is initialized as $subdiv = \lceil \sqrt[3]{n} \rceil$ for each edge, where n is the total number of splits. Then we test the three conditions as follows by simple counting:

1. The max number of primitives in one grid is greater than the capacity of split, which means all the grids are still overlapped.
2. the ratio of the number of primitives in overlapped grids to total number in scene exceeds a certain threshold. If this condition is satisfied, increase the $subdiv$ and test these three conditions again.
3. The number of grids contained primitives is several times the split number. The granularity is too dense and the subdivision should be terminated.

After the adaption, there may be still some overlapped grids. We copy the records from them to new scene files. Note the size of the data being copied in one grid may be much smaller than the block size, which is not conducive to a balanced load. Because the time complexity of each task is $O(r\log p)$ where r stands for the number of rays scheduled by the task and p is the number of primitives in the split, and load balancing requires an equal number of them between tasks. By scheduling the tasks with most rays, the difference of r is minimal. For p, we fill the block with records being copied. When the block is filled near its capacity, turn to a new file to avoid the record truncation. In this way, the data in overlapped grids are copied to one-block files without overlap, and then the scheduling grids will be updated.

The cost is related to the number of records in overlapped grids. Although it could be expensive, the organized data can speed up rendering, especially for most applications which render the same data multiple times in different perspectives.

C. Reordering Ray Computation

The computation of each task is to test related rays for the nearest intersections with input split, and trace secondary rays from the intersections recursively. The processing of secondary ray will not start before the previous ray has been tested, so the scene splits with no rays have to wait.

In order to get more rays, the secondary rays must first test with the current split, and then cast new rays recursively until all rays are out of the scope of the split. In this way, the replacement of ray data among tasks can be reduced. The ray data are not sent directly to the other tasks but output to a specific directory in HDFS to avoid the dependencies between tasks. We customize the MultipleOutputs interface (MOS) provided by Hadoop to achieve this.

The ideal situation for scheduling is tracing the rays immediately as they come out without considering the dependence on previous rays. The rays can be processed once at least, and then discarded without taking up space. Each ray is possible to contribute to the final image, once it intersects with an emitting object or shoots out of the entire scene, the light color C_l or background color C_{bg} is returned to the pixel. The actual contribution of the ray is related to the path it traced and the probability density of choosing this path. The path from the pixel to the contribution points is unique and denoted as $\{r_0, r_1, \dots, r_d\}$ where d is the depth of path terminated by Russian Roulette [1] and the intersections of rays in path are denoted as $\{x_1, x_2, \dots, x_{d+1}\}$, where x_d is the intersection of r_{d-1}. We exploit the ray coherence by merging the rays with same origin and direction and belong to the same pixel, so that subsequent rays can reuse these test results. Suppose the intersection x_d has N_d samples for the integral, and the rays generated from x_d are denoted as r_{d_i} where $i = 1,2, \dots, N_d$. If there is radiance $I(r_{d\_i})$ incoming from r_{d_i} to x_d, the total radiance outgoing to r_{d-1} can be estimated by Monte Carlo method [1] as follow:

$$I(r_{d-1}) \approx \varepsilon(r_{d-1}) + \frac{1}{N_d}\sum_{i=1}^{N_d}\frac{f_r(r_{d_i}, r_{d-1})I(r_{d_i})\cos\theta_{d_i}}{pdf(r_{d_i})} \quad (1)$$

where f_r is the bidirectional reflectance distribution function that defines how light is reflected from r_{d_i} to r_{d-1} at x_d, θ_{d_i} is the angle between the direction of r_{d_i} and the surface normal at x_d, and pdf is the probability density function for r_{d_i}. All the directions in the formula share the same origin x_d. Note that $\varepsilon(r_{d-1})$ is the emitted radiance from r_{d-1} if the point x_d is self-emitted, i.e. the contribution point, and the contribution of x_d is:

$$C(x_d) = \varepsilon(r_{d-1})\prod_{j=1}^{d}\frac{f_r(r_j, r_{j-1})\cos\theta_j}{pdf(r_j)N_j} \quad (2)$$

The factors of $\varepsilon(r_{d-1})$ is referred to as $F(r_{d-1})$ and can be stored with the ray as it is generated, so that all rays are independent for scheduling.

Algorithm 1. Tracing Job with Overlap Reduction.

```
Input:
  while slot > 0
    add S_i with most rays into List(S)
    slot=slot−1
  end while
  for each S_i in List(S)
    look up A_G to get corresponding G_i
  end for
Map(k=G_i, v=S_i ):
  while G_i are not empty
    get a ray r to test for intersection
    if r has no intersection then
    |  remove r from G_i
    |  get next non-empty grid g(r) to be traversed by r
    |  if g(r) is null then
    |    terminate r
    |    MOS.write(k= r.pixel, v=C_bg)
    |  else
    |    MOS.write(k=null, v= r, path= g(r))
    |  end if
    else  // r intersects with x
    |  terminate r and remove it from G_i
    |  if x is self-emitted then
    |    MOS.write(k= r.pixel, v=C_l × F(r))
    |  end if
    |  if the path of r is not terminated then
    |    generate secondary rays and add them to G_i
    |  end if
    end if
  end while
  tag G_i as 'DISCARD'
```

We run several Map-only jobs (see Algorithm 1) until the rays in all the groups are exhausted. Each job select splits $\{S_1, S_2, \dots, S_n\}$ with ray groups $\{G_1, G_2, \dots, G_n\}$, where n is the number of task 'slots', the units of the Hadoop cluster computing resources. We customize the InputFormat and RecordReader interfaces in Hadoop to perform the join of the two datasets on map-side. The global hierarchy of the acceleration structure A_G is replicated to each task by Distributed Cache. The ray groups G_i have been processed are tagged as 'DISCARD'.

IV. EVALUATION

In this section, we present the performance results and evaluation of our algorithm has been implemented in Hadoop. The cluster has 8 nodes consists of two types of computers equally assorted, as shown in Table 1.

There are two static datasets rendered for evaluation: a CT scan dataset for 3D reconstruction from the National Library of Medicine, and a Townhouse scene similar to that in [7]. To unify the test cases, we have preprocessed the CT scan data with Marching Cubes and extracted 8,123,516 diffuse faces in a total size of 4038MB. The Townhouse scene has 732,104 triangles with texture coordinates and materials taking 1189MB. The two datasets are stored in 32 and 10 blocks respectively in HDFS with the default block size of 128MB. We rendered them at the resolution of 1024x1024, 5000 samples each pixel, with two area lights and camera placed at an appropriate angle and position.

We also constructed three rendering implementations with Hadoop MapReduce for comparative purposes. Each experiment was run three times and the results are averaged.

1) Demand Driven (DD). Rays are evenly distributed among Map tasks for load balancing at each iteration, and the data being rendered is loaded from HDFS as required. This implementation is similar to image-plane decomposition described in [4], which ignore data locality.

2) Scheduling Grids without Overlap Reduction (SG without OR), an original version of our algorithm. When the splits with most rays have been chosen, it requires other splits contain overlapped girds to be loaded in the task as well, and an extra Reducer to get the nearest intersection.

3) Our Algorithm (SG with OR). Set the overlap threshold as 0.2 in overlap reduction, which implies a maximum 20 percent of data require to be organized. The result of $subdiv$ in CT scan data is 10, while the Townhouse scene is 8. The smaller size of data gets relative large granularity than the scan data because of the worse order.

A. Render Time

We rendered the two datasets with the three implementations. The total time of SG with OR is less than the others for CT scan data in Fig. 4, but greater for Townhouse scene because of the pre-processing time. If rendering the same scene several times, our algorithm will win. The tracing stage of SG without OR is slower than our algorithm, because it requires more data to process and wait for the single Reducer each iterative job. Results show that the overlap reduction is worth it, even for the data with bad order like Townhouse.

The experiments were also run on 2 nodes and 4 nodes with equally assorted types of computers and configured with 0 task slot in the other nodes. Results compared with 8 nodes are shown in Fig. 5 that our algorithm gains more performance improvement when nodes are added.

B. Data Scalability

We constructed different size of datasets by surface subdivision and replication, and tested the data scalability of the implementations. As the data gets bigger, our algorithm took less rendering time than the others, as in Fig. 6. Note that the rendering time of subdivided surface data for Townhouse scene is less than raw data, because the data generated by the subdivision is stored contiguously, so that more spatial coherence can be exploited.

C. Load Status

We recorded the load status during the CT scan rendering by monitoring the cluster with Ganglia. Table 2 lists the reads from HDFS in each rendering stage of the three implementations, and the ratio of *reads_from_remote_client* to total reads. Result shows fewer HDFS reads in our algorithm since it is more focused on data locality, so that the network load is lower.

As the data being rendered are nearly equal among tasks in our algorithm, the task with more rays would be a drag, leading to imbalance of workload. We compare CPU usages and HDFS reads for each node in Fig. 7, which shows that the slow task has little impact to the overall job.

D. Fault Tolerance

The global illumination rendering of large-scale scenes often takes several hours, so the system needs to have sufficient stability. We verified the fault tolerance by randomly selecting a node to shutdown network connection. The rendering time was increased due to speculative execution and lack of running nodes, but it does not need to restart the entire work. Our algorithm has more jobs than DD, but each job takes a relatively short time. The latter job in DD has to process a large number of rays, which takes more time to test all intersections, and is more vulnerable to failure.

TABLE I. CONFIGURATIONS OF COMPUTERS

Type	Configurations			Hadoop conf.	
	CPU	*RAM*	*network*	*slots*	*heap*
A	Intel E6700 3.20GHz	2G	1Gbps	2	1GB
B	Intel i5 3.10 GHz	4G	1Gbps	4	1GB

TABLE II. LOAD STATUS IN EACH STAGE

Implementation	Stage	HDFS Reads	Remote Reads Ratio
DD	PP.	4038MB	0.55
	Tracing	261752MB	0.76
SG without OR	PP.	28266MB	0.4
	Tracing	253719MB	0.62
SG with OR	PP.	28266MB	0.4
	Tracing	171072MB	0.49

PP. is short for Pre-Processing.

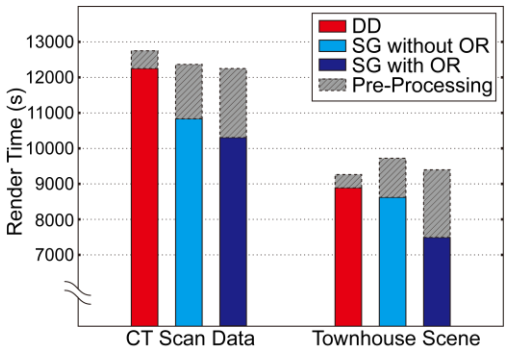

Figure 4. Rendering time of the three implementations for the two datasets.

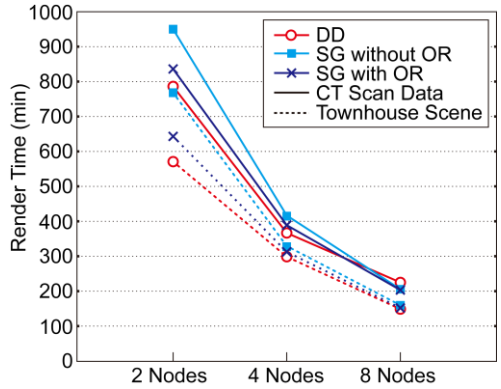

Figure 5. Comparison of rendering time for different nodes.

V. Conclusions and Future Work

In this paper we presented an improved method of Monte Carlo ray tracing in distributed memory for large-scale rendering. We studied the structure and distribution of the scene data in the storage fully, and reduced the remote data access by building an efficient acceleration structure with the consideration of coherence. Our rendering system is implemented in a Hadoop cluster and the large scene data are processed in splits by MapReduce framework, which ensures scalability and reliability. The result shows that our implementation takes less rendering time than other methods.

The future work is to better control the workflow of the MapReduce jobs, and parallel process the rays in one task by using GPUs, which are beyond the current framework.

Acknowledgment

This work was supported in part by the NSFC (Grant No.61202102, 61173036) and the Growth Young Teacher Program of Hunan University.

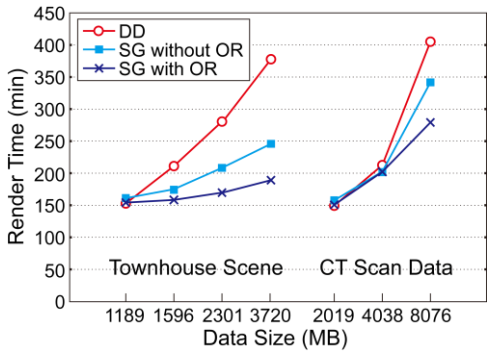

Figure 6. Comparison of rendering time for different data size.

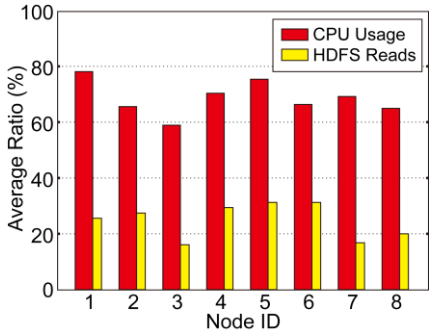

Figure 7. The average usage of CPU and the ratio of HDFS reads to total in tracing stage of our algorithm.

References

[1] M. Pharr and G. Humphreys, Physically based rendering: From theory to implementation: Morgan Kaufmann, 2010, pp. 679-730.

[2] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, and D. Luebke, "Optix: a general purpose ray tracing engine," in ACM Transactions on Graphics (TOG), 2010, p. 66.

[3] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, and P. Hatcher, "Large data visualization on distributed memory multi-GPU clusters," in Proceedings of the Conference on High Performance Graphics, 2010, pp. 57-66.

[4] 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.

[5] A. Bialecki, M. Cafarella, D. Cutting, and O. O'MALLEY, "Hadoop: a framework for running applications on large clusters built of commodity hardware," Wiki at http://lucene. apache. org/hadoop, vol. 11, 2005.

[6] D. E. DeMarle, C. P. Gribble, S. Boulos, and S. G. Parker, "Memory sharing for interactive ray tracing on clusters," Parallel Computing, vol. 31, pp. 221-242, 2005.

[7] T. Kato and J. Saito, "Kilauea: parallel global illumination renderer," in Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, 2002, pp. 7-16.

[8] 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, 1997, pp. 101-108.

[9] L. Northam, R. Smits, K. Daudjee, and J. Istead, "Ray tracing in the cloud using MapReduce," in High Performance Computing and Simulation (HPCS), 2013 International Conference on, 2013, pp. 19-26.

[10] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, pp. 107-113, 2008.