ParRay: A Parallel Raytracer
Project Milestone

Zane Fink[1] and Jianfeng Qiu[1]

University of Illinois at Urbana-Champaign
{zanef2,jqiu13}@illinois.edu

1 Introduction

Rendering, a fundamental component of computer graphics, is the process of displaying a three-dimensional scene in a two-dimensional image. This process enables many applications such as simulation, animated movies, special effects, and video games. Today, the mainstream approach to rendering in computer graphics is ray tracing. Ray tracing enables scenes to be rendered with ultra-realism , but comes with substantial computational cost.

A ray is a vector with a magnitude and direction. In computer graphics, rays originate from a light source such as the sun or a light bulb. The path of each ray is followed until it intersects with an object, where the object and the ray can interact in different ways depending on the object's composition. For example, the object can cause the ray to reflect or refract.

A simple ray-tracing algorithm is shown below (taken from Deng et al. [1]. Observe the $O(m \cdot n)$ complexity, where m is the number of pixels being rendered, and n is the number of objects in the scene. Because of the scaling of this algorithm, substantial research effort has been focused on reducing the complexity. One such algorithm uses a bounding volume hierarchy (BVH) [2] to reduce the complexity to $O(mlogn)$. An example BVH and the manner in which it reduces the number of intersection calculations is shown in Figure **??** (figure from Deng et al [1]).

```
for each pixel P on the display
      Create ray R passing from camera through P
      initialize NearestT to INFINITY and NearestPrimitive to NULL
      Color C = TraceRay(R,S,NearestT)
      Shade the current pixel with C
end for
```

Thus far, we have completed a ray tracer with support for parallelization using OpenMP. A more detailed description of work completed follows.

2 Current Progress

Our implementation currently consists of a ray tracer that is capable of rendering different types of materials such as diffuse, reflective, and refractive objects, and

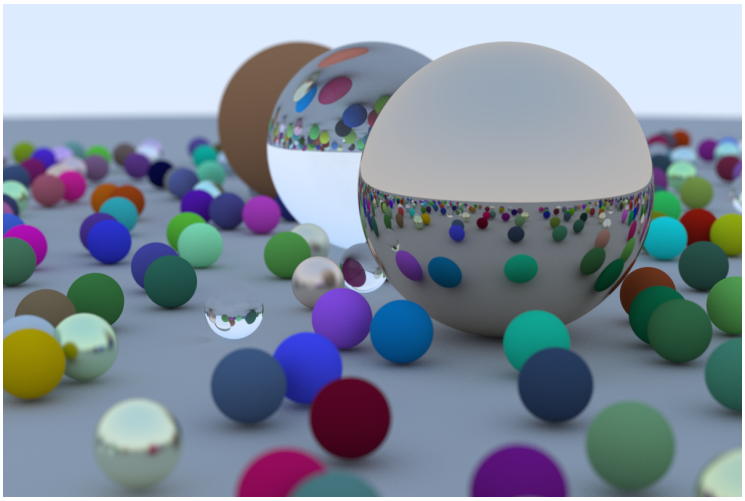

Fig. 1. An image produced by our ray tracer. This image consists of 484 spheres of varying color and material.

is equipped with a movable camera with depth of field capabilities. To learn how ray tracing works, we used the book "Ray Tracing in One Weekend" [3]. The sequential sampling algorithm is described in the code snippet below. An example image produced by our ray tracer can be seen in Figure 1. To parallelize the rendering, we dynamically assign the rendering of the pixels in a row of the image to a core.

```
for each row of the output image
    for each column of the output image
      for each sample of the total samples per pixel
            emit a ray to the scene and determine the color
            accumulate color sample
      end for
      set output buffer[row][column] = sampled color average
    end for
end for
```

To evaluate the performance of our ray tracer, we executed the render for the scene shown in Figure 1 with $1, 2, 4, 8$, and 12 threads. These experiments were performed on an Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 24 total physical cores spread across 2 NUMA nodes, each with 12 cores. In these experiments, we reduce the number of rays cast for each pixel in the output from 500 to 50, reducing the algorithm's runtime for ease of experimentation. The effect this has on output quality can be seen in Figure 2.

The results of the experiments can be seen in Table 2. Given that ray tracing is an algorithm described as "embarrassingly parallel", we anticipated higher

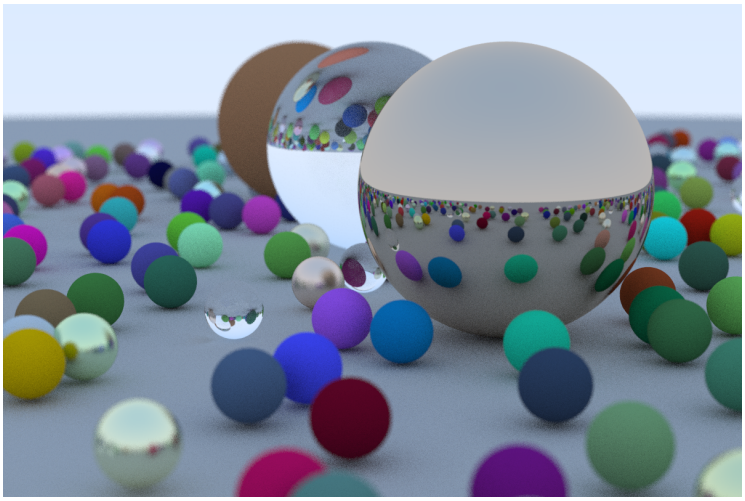

Fig. 2. The image produced by our raytracer when 50 rays are cast for each pixel. Close inspection of the image reveals graininess that is not present when 500 rays are cast for each pixel.

Number of Cores	Runtime (s)	Speedup
1	372.986	1
2	267.516	1.39
4	198.159	1.88
8	187.991	1.98
12	162.687	2.29

Table 1. Rendering runtime vs. the number of cores used to perform the rendering. Also shown is the speedup over one core.

parallel efficiency; more experimentation is required to assess the cause of poor scaling and make improvements.

3 Future Work

In our current implementation, each ray that is cast checks if it has intersected with every object in the scene. To reduce the number of intersection checks that must be performed, acceleration structures such as bounding volume hierarchies (BVH) have been introduced [2]. Our BVH implementation is almost finished, but we are still working out some bugs. We expect that the BVH will result substantially speed up our ray tracer.

In addition to finishing our BVH implementation, we want to enable our ray tracer to scale beyond a single node using MPI. From the runtime results shown in Table 2, we can see that different parallelization strategies to increase the

scalability of our implementation. For example, we can increase the granularity of the scene decomposition. More experimentation is needed to identify the scaling bottlenecks in the current implementation. The results of this experimentation, and the additional strategies we employ, will be in our final report.

References

1. Deng, Y., Ni, Y., Li, Z., Mu, S., Zhang, W.: Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques. ACM Comput. Surv. **50**(4) (Aug 2017). https://doi.org/10.1145/3104067, https://doi.org/10.1145/3104067
2. Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., Manocha, D.: Fast bvh construction on gpus. Computer Graphics Forum **28**(2), 375–384 (2009). https://doi.org/https://doi.org/10.1111/j.1467-8659.2009.01377.x, https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01377.x
3. Shirley, P.: Ray tracing in one weekend (December 2020)