ParRay: A Parallel Raytracer
Final Report

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 1 (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
```

The rest of the report is organized as follows: Section 2 provides an overview of ray tracing, Section 3 details our implementation, and Section 4 provides an evaluation of the performance of our implementation.

2 Ray Tracing Fundamentals

Ray tracing is a technique used in computer graphics to produce photo-realistic images. Often, an image is a scene consisting of a camera or viewport, and objects

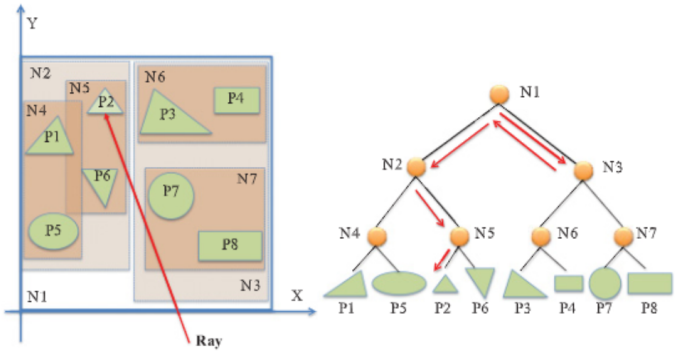

Fig. 1. Illustration of how using a bounding volume hierarchy reduces the number of intersection calculations that must be performed.

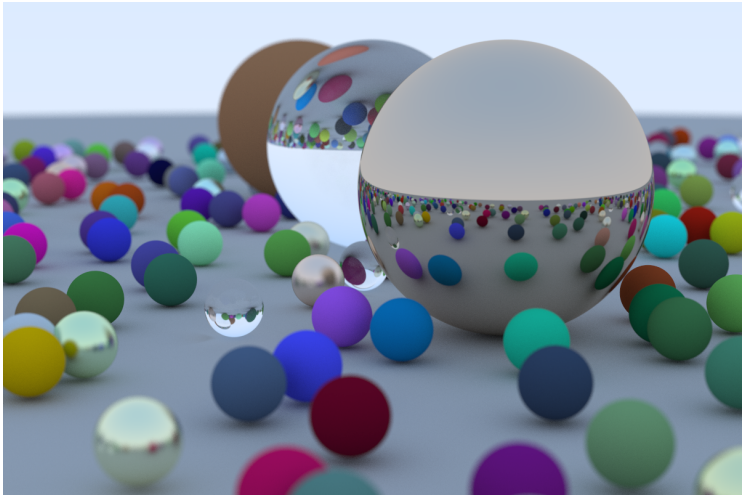

Fig. 2. An image produced by our ray tracer. This image consists of 484 spheres of varying color and material.

of varying type, color, and material. An image produced by a raytracer is from the perspective of the camera, as seen in Figure 2. To color a pixel, rays are cast from the camera through the pixel into the scene. We have implemented three different materials: diffuse, metal, and dielectric, each with different properties.

To determine which objects a ray hits, an intersection test between the ray and any of the objects it may hit is performed. The following description is adopted from [3]. Each ray has an origin and a direction. Formally, a ray can be defined as $P(t) = A + tB$, where P is the point on the ray, A is the origin of the array, and B is the unit vector direction of the ray. We use t to move P away from A in the direction of B. We define a sphere with radius r and

center C as $\|P - C\|^2 = r^2$, where P is a point on the sphere. To test whether a ray intersects a sphere, we search for t such that $t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, where $a = dot(B, B)$, $b = 2 \cdot dot(B, A - C)$, and $c = dot(A - C, A - C) - r^2$. If the discriminant $d = b^2 - 4ac$ is less than zero, then the ray does not intersect the sphere. If $d = 0$, the ray is tangent to sphere, and if $d > 0$, then the ray intersects the sphere.

2.1 Diffuse Materials

Diffuse materials take on the color of their surroundings, modulated with their own intrinsic color. When a ray hits a diffuse material, it is reflected back randomly relative to the normal of the surface. If the child ray hits an object, it is reflected back relative to the normal of the new surface. Each new object hit has decreasing effect on the color of the pixel. This happens until the ray doesn't hit any objects or a depth parameter is reached.

2.2 Metal

Rather than absorb light, metal reflects it. The reflection of ray V is the ray $R = V - 2 \cdot dot(V, N) \cdot N$, where N is the surface normal of the metal sphere. If the reflected ray intersects another object, the metal at that point is given the color of the object hit by the reflected ray.

2.3 Dielectric

Dielectrics are clear materials such as water and glass. When a ray of light hits a dielectric material, it is split into two rays: a reflective ray and a refractive ray.

3 Implementation

3.1 Serial Raytracer Capababilities

Our serial ray tracer was built following the lessons in the "Ray Tracer in One Weekend" [3]. To render a scene, the ray tracer iterates over the pixels in each row and column of the image. For each pixel, a number n_r of rays are cast from the camera through the pixel into the scene. Every object in the scene is tested for intersection with the ray. As detailed in Section 2, the material of the object will determine the behavior of the ray that hits it. The color of the pixel is the average of the color of all the rays cast through the camera into the scene. The output of the ray tracer is shown in Figure 2. For an image of dimension $h \times w$, the complexity of the sequential ray tracer is $O(h \cdot w \cdot n_r \cdot n_d \cdot n_o)$, where n_r is the number of rays cast for each object, n_d is the maximum number of rays that can be generated from each ray that intersects an object, and n_o is the number of objects in the scene.

3.2 Bounding Volume Hierarchy(BVH)

The BVH builds an octree in two passes, the first pass divides the scene's 3D space (root node) into 8 child cubic spaces (child nodes) around the centroid, and assign spheres into one of the child space based on their centroid locations. Repeat this divide and assignment procedure until a child space hosts only one sphere. At the end of the first pass, calculate a bounding extent for each sphere (Figure 3).

The second pass is from bottom up. Adjust each parent node's bounding extent contain exactly its children spheres.

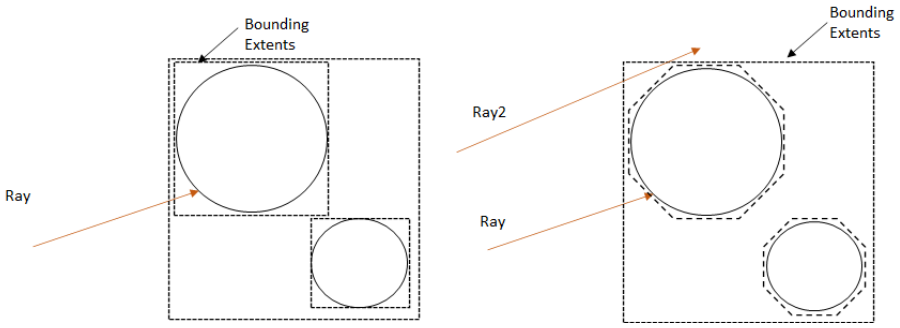

Fig. 3. 2D illustration on BVH. Left: A parent node containing two child nodes, each hosts a sphere with defined square bounding extents. Right: A parent node containing two child nodes, each hosts a sphere with an octagon bounding extents.

When ray tracing, the algorithm calculates the intersection points between a ray and the bounding extends of a node. If an intersection is detected, recursively do intersection testing of all of its 8 children until a single sphere object is identified.

To enhance detection accuracy, an octagon bounding extents is used to better approximate a sphere during intersection testing (Figure 3 Right). Now each intersection tests 3 pairs of extent bounds instead of 2 pairs, as in the case of a rectangular box. For the 3D equivalent, each intersection tests 7 pairs of extent bounds instead of 3 pairs, as in the case of a hexahedron box.

3.3 Parallelization with OpenMP

Observe that each pixel can be rendered independently from every other pixel. This makes simple parallelization with OpenMP simple: the outermost loop is annotated with a `#pragma omp parallel for` declaration. Because different parts of the scene contain more objects than others, static scheduling leads to large load imbalance between threads; therefore dynamic scheduling is used, where each thread is given a row of the output image to render at a time.

3.4 Parallelization with MPI

To enable execution across multiple nodes, additional support for MPI was included. Now, the render is split hierarchically, with each process receiving a static portion of the image to render, and the cores on each process dynamically render the output scene in parallel. For p process, each process is assigned $1/p$ of the render. After a process finishes its portion of the scene, an `MPI_Put` is used to insert the output into result buffer, which is stored on process 0.

3.5 Dynamic Load Balancing

Observe that the naïve use of MPI has the same problem as static scheduling in OpenMP: load imbalance. While there is no load imbalance within a process, load imbalance between processes persists. To overcome this, we implemented a dynamic load balancing mechanism for our ray tracer. The load balancer takes the form of a coordinator process. This process is responsible for assigning rows of the render to the other processes. When a worker process has finished its portion of the render, it will place the result into the result buffer via `MPI_Put`, and will request more work from the coordinator process. To balance the overhead of this mechanism with load imbalance between the processes, the coordinator assigns rows in guided scheme. That is to say that the coordinator will assign $C_i = R_i/P$ rows to each process, where P is the number of worker processes, and R_i is the number of rows remaining in the render.

Notice that the cost to assign work to processes is relatively high, requiring two messages: the request and the response. To mitigate some of this overhead, the coordinator process can assign no less than $2 \cdot t$ rows to each process, where t is the number of threads each process has for rendering.

4 Evaluation and Discussion

4.1 Experimental Methodology

We evaluate performance of our ray tracing implementation on two machines, detailed in Table 1. Unless stated otherwise, each experiment was performed for 4 trials for an image of size 1200×800, with 500 samples taken for each pixel, with a maximum tracing depth of 10. We report the mean value of each experiment, and each experiment was performed with exclusive access to the node on which it was performed.

Platform	Model	Cores	Clock	Memory
PLAT1	2× E5-2670 (Sandy Bridge)	2×10	2.5 GHz	64 GiB
PLAT2	2× Epyc 7542 (Zen)	2×32	2.9 GHz	256 GiB

Table 1. Platform details. Numbers given are for a single node on each platform.

4.2 Single Thread Performance of Simple Tracing and BVH

Table 4.2 summaries the difference in running times to render a scene with varying sphere counts using a simple brute force iteration (BFI) versus BVH.

At low object counts (8 spheres), BVH takes longer time than BFI, due to the number of extra vector multiplications (around 14 vector multiplications plus solving one quadratic equation) incurred to explore each level of the octree; BFI requires only one quadratic equation solving per sphere.

The benefit of BVH starts to emerge at higher object counts where the brute force tracing running time increases linearly. At object count 4097, BFI tracing takes 10 times as much duration to render the same scene. It's worth noting that at high object counts, a large portion of spheres fall outside of the camera view port. BVH ignores those out of view spheres, whereas BFI tests all of them. This makes running the BFI at high object count is prohibitly slow.

Number of Spheres	BVH	Simple Tracing
8	1154s	668s
20	921s	2138s
4097	44min	567min

Table 2. Rendering times for BVH vs Simple tracing on different number of spheres. The run time is collected on `PLAT1` using 1 core.

4.3 Parallel Ray-tracing Time Performance

Figure 4 shows the runtime performance for ParRay on `PLAT1` with (MPI-LB) and without (MPI) load balancing. We can see a substantial reduction on runtime with increasing rank used, and a speedup of 22.8× over the single-rank execution is observed for MPI, while a speedup of 70.6× is observed for MPI-LB. We observe the advantage of load balancing, especially with higher rank counts: MPI-LB exhibits a maximum 3.17× speedup over MPI.

Figure 5 plots runtime performance on `PLAT2` scaling from 4 to 192 cores. It can be seen that ParRay achieves near-perfect speedup up to 64 cores, but when additional nodes are utilized parallel efficiency decreases. Interestingly, MPI-LB has a slight negative effect on execution time, with a minimum speedup of 0.84, and maximum of 1.0. This is consistent with the data observed in Figure 4: load balancing has little effect on low rank counts. This can be explained by the large portion of the image that is rendered by each rank, where each rank is more likely to have uniform load.

4.4 Parallel Speedup and Efficiency Evaluation

Parallel implementation of BVH is done with an MPI/OMP hybrid approach as mentioned in Section 3. In order to explore the effect of MPI communication on

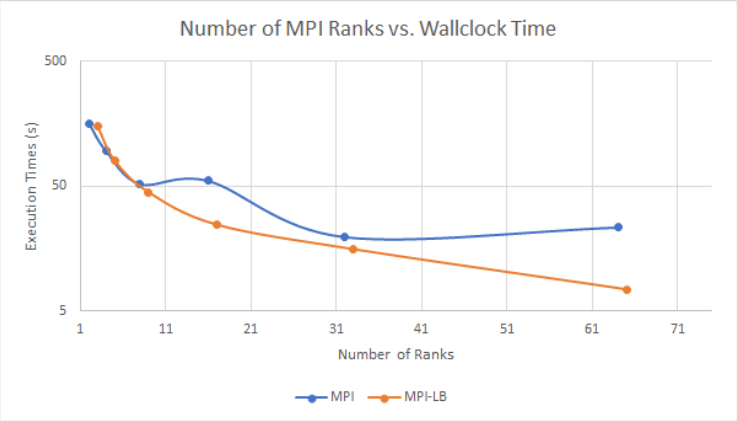

Fig. 4. Runtime vs. number of MPI ranks used for execution with load balancing (MPI-LB), and without (MPI) on `PLAT1`. MPI-LB uses one additional rank for the work distributor. Also shown is the ideal runtime, which assumes perfect parallel efficiency. Each rank uses 2 cores for rendering, totalling 128 cores.

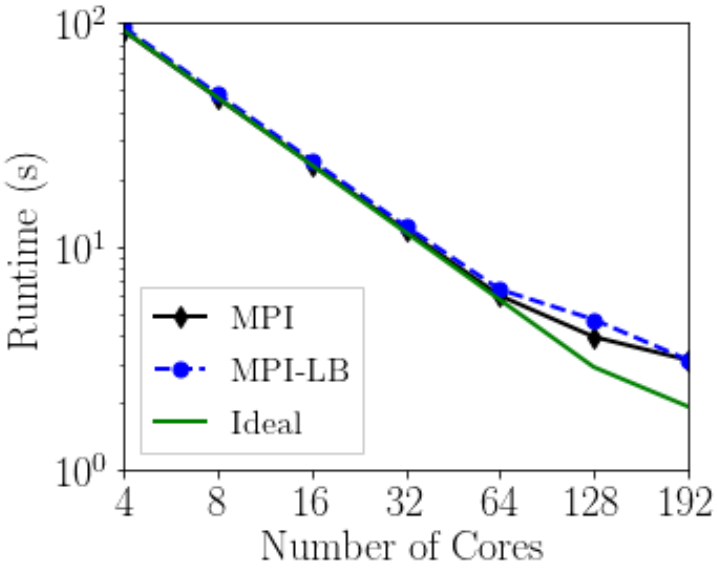

Fig. 5. Runtime vs. number of cores used for execution with load balancing (MPI-LB), and without (MPI) on `PLAT2`. The extra rank used by MPI-LB is not shown. One node has 64 cores, totalling 3 nodes, one MPI rank per node.

performance, this experiment limits each MPI rank could spawn only two OMP threads. Therefore a node with 24 virtual cores can host 12 ranks.

The speed up is calculated according to the equation

$$Speedup = SequentialTime/ParallelTime$$

Theoretically speaking, each ray from the origin tracing at one target pixel is independent of other rays and other target pixel. Therefore, the ideal speedup is linear, assuming no communication cost.

Given the same input data set, the speedup for MPI implementation without load balancing is close to the ideal when the number of cores is below 20. According to 12, a node consists of 20 CPU cores, and core count under 20 could be sharing the same physical memory, hence the speedup scales well with core number. At node count larger than 20, MPI ranks span across multiple nodes, speedup deviates from the ideal case due to extra inter-node communication.

Load imbalance also contributes to the weak scale-ability at high node count, the overall task parallel run time is as fast as the slowest MPI rank. When analysing load imbalance in Section 4.5 , it is notice that at high MPI rank count, the overall task execution time is affected by a couple of CPUs that run slowly. The slow down in the CPUs could be random, and they could be eliminated by repeating the test several times.

Dynamic balancing would have a positive impact by removing this bottle neck and enabling the BVH ray tracer to have a strong scale-ability up to 64 ranks (128 cores). Section 4.5 will further discuss load balancing.

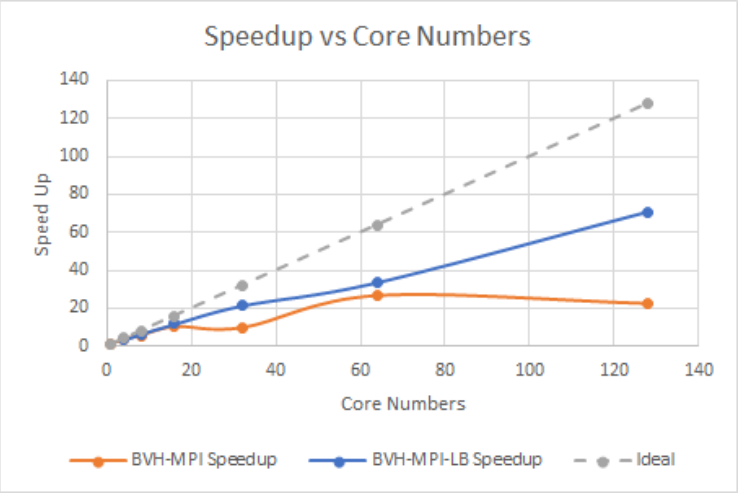

Fig. 6. Speedup vs number of cores. The running time is collected on `PLAT1`. Data set consists of 1600 spheres of random sizes, materials and locations. In each run, one MPI rank spawns 2 OpenMP threads, the NUMA nodes number is adjusted to avoid over-subscription.

The parallel efficiency is calculated by the equation

$$ParallelEfficiency = SequentialTime/(CoreCount * ParallelTime)$$

The ideal efficiency is 1 given the unit of work tracing one ray from the origin is independent of others. The following Figure 7 depicts an efficiency drop with higher MPI ranks count. This can be explained by higher number of inter-rank communication. The workload imbalance has a similar negative impact on efficiency as it does on speedup.

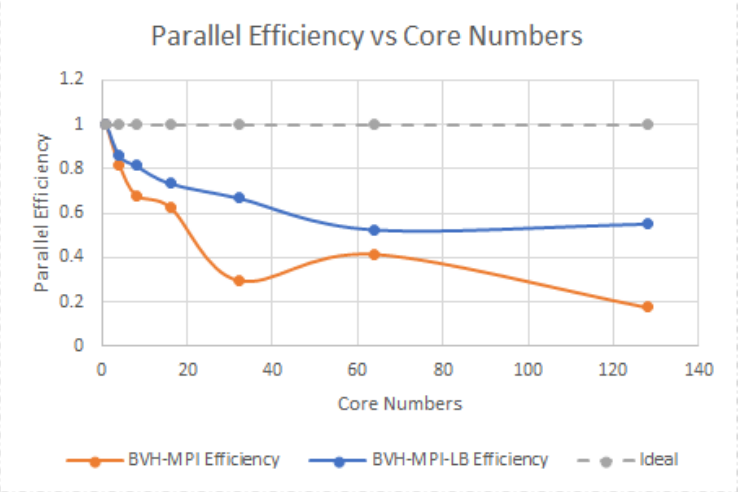

Fig. 7. Efficiency vs number of cores. The running time is collected on `PLAT1`. Dataset consist of 1600 spheres of random sizes, materials and locations. In each run, one MPI rank spawns 2 OpenMP threads, the NUMA nodes number is adjusted to avoid over-subscription.

4.5 Impacts of Load Balancing

It can seen from Fig 2, spheres are not evenly distributed in the output viewport. For example, upper rows in the image is relatively empty compared to the lower part of the image where spheres are numerous and scattered all over the floor. Additionally, rays shooting at the middle region can reach many more other spheres due to the large reflective metal ball and glass ball.

As work are divided by row and distributed to each core, CPUs working at the upper few rows would finish much faster than those tracing the middle region. Figure 8 shows the run time distribution among 64 MPI ranks. To emphasize the workload distribution, normalized runtime is used, which is defined by the following ratio.

$$NormalizedRuntime = CPURuntime/OverrealRenderTime$$

Without load balancing (BVH-MPI), CPU 1-20 render the lower rows of the scene and has medium workload; CPU 21-40 render the center rows and has the highest workload; lastly CPU 41-64 render the upper rows and sit mostly idle.

With load balancing (BVH-MPI-LB), rows of work are dynamically allocated. Each CPU is busy most of the time, computing resources utilization is optimized. As described in Section 4.5, work is distributed to idle CPU, leading to faster job completion.

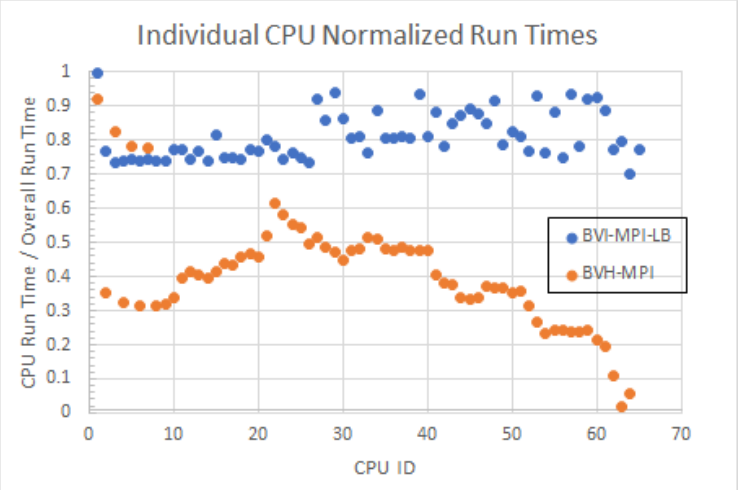

Fig. 8. Normalized runtime of each core. The running time is collected on `PLAT1`. Data set consists of 1600 spheres of random sizes, materials and locations. 64 MPI ranks,each rank spawns 2 OpenMP threads, 8 nodes are used to host 16 cores (2threads*8rank) each

4.6 Tiled OMP Experiment

This experiment explores square tile instead of rows decomposition, as illustrated in Figure 9. The experiment is setup with OMP threads running on a Intel(R) i7-6700 quad-core processor at 3.4GHz. BVH is used as the acceleration algorithm for the

The rendering time of different tile sizes is charted in Figure 10. The tile size has no observable impact on overall job running time.

The absent of spacial locality may lead to the observed result. When a ray hits a reflective surface, it is redirected to another sphere that may located outside of the current tile (as illustrated in Figure 9). The redirected target probably does not locate in adjacent memory blocks.

Another explanation is the repetitive sampling (500 samples) of each pixel, likely through the same set of spheres, diminishes the overhead of the initial

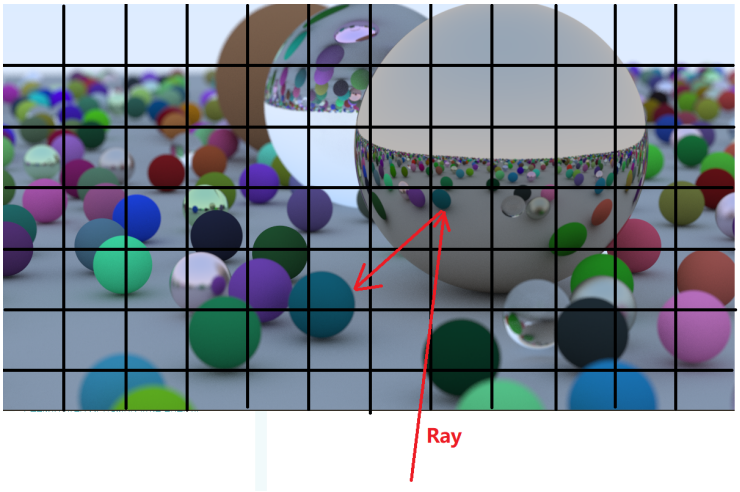

Fig. 9. Tile Decomposition of Ray Tracing Target

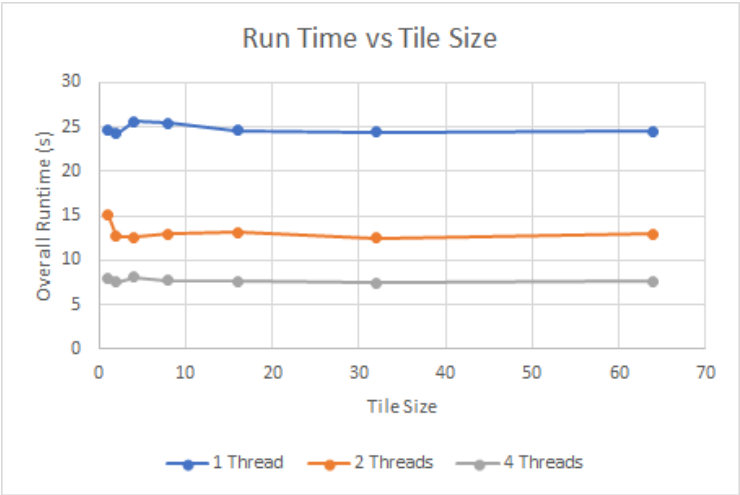

Fig. 10. Rendering time of tile decomposition with varying sizes.

compulsory misses. As a result, cache misses are insignificant compared to computation time for each pixel.

5 Correctness Evaluation

To demonstrate the correctness of our parallel agorithm, we first rendered the scene shown in Figure 2 one each of our configurations: sequential, parallel with OpenMP, parallel with MPI + OpenMP, and finally parallel with MPI +

OpenMP where load balancing is included. Then, we created the output image shown in Figure 11 in the following way.

The output for each version was compared with the output of the sequential version. Each pixel in Figure 11 is the sum of the magnitude of the differences of that pixel between each implementation and the pixel in the reference implementation R. That is to say that each pixel i, j in the verification image V is calculated as:

$$V_{ij} = \sum_{F \in P} |R_{ij} - F_{ij}| \tag{1}$$

Where P is the set containing the different parallel implementations. It can be seen that if V_{ij} is large, then the color of that pixel varies substantially across implementation. In Figure 11, white pixels indicate no change in pixel color, and darker pixels indicate greater change. Little variation between implementations is observed. This variation is expected due to the random nature of certain portions of the rendering process, e.g. each child ray has a random direction with a maximum difference between one child rays of the radius of the unit circle.

Fig. 11. Image demonstrating correctness of different parallel implementations of the ray tracer. Darker pixels indicate greater divergence across different implementations. Note that a border was added to the image to differentiate it from the page.

6 Conclusion

Ray tracing is a fundamental operation in computer graphics used to generate photo-realistic images. Given its computational complexity, ray tracing complex

scenes is intractable on a single core. The highly-parallelizable nature of ray tracing in combination with acceleration structures such as bounding volume hierarchies make it an attractive target for acceleration. In this report, we have detailed ParRay, a distributed-memory parallel ray tracer capable of rendering scenes consisting of thousands of objects of different materials. ParRay is scalable, with demonstrated parallel efficiency of 0.62 on 192 cores.

While ParRay delivers high-performance, it is unsuitable for real-time computer graphics. A factor limiting further speedup is limited parallelism in row-based decomposition of the rendering. If greater speedup is to be achieved, one could achieve better strong scaling performance by implementing parallelism at the ray level, rather than the pixel level. For example, a sampling depth of 500, means at least a 500× increase in the number of data parallel objects on which work may be performed. Additionally, ray-tracing's embarrassingly parallel nature makes it suitable on GPUs, and in particular on recent architectures form NVIDIA that feature hardware acceleration for real-time ray tracing.

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)

A Appendix Campus Cluster CPU Information

```
CPU INFO
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              20
On-line CPU(s) list: 0-19
Thread(s) per core:  1
Core(s) per socket:  10
Socket(s):           2
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               62
Model name:          Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz
Stepping:            4
CPU MHz:             1200.103
CPU max MHz:         3300.0000
CPU min MHz:         1200.0000
BogoMIPS:            4999.77
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            25600K
NUMA node0 CPU(s):   0-19
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts
acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts
rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx
smx est tm2 ssse3 cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic popcnt tsc_deadline_timer aes xsave
avx f16c rdrand lahf_lm epb ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid fsgsbase smep
erms xsaveopt dtherm ida arat pln pts spec_ctrl intel_stibp flush_l1d
Run Benchmarks
Run Benchmarks-single threaded
```

Fig. 12. CPU information