Distributed Real-Time Ray Tracer

CSC 462 – Project

Michael Windels
August 10, 2019
V00854091

For Dr. Yvonne Coady's CSC 462 class at UVic (Summer 2019).
Repository: https://github.com/MWindels/distributed-raytracer

1 Introduction

This project aims to produce a distributed real-time ray tracer. The ultimate goal is to leverage a distributed system in order to produce ray-traced images at an acceptable frame rate (24-30 frames per second).

1.1 Ray Tracing in a Nutshell

Fundamentally, ray tracing is a means to project a three-dimensional scene onto a two dimensional projection plane. In the context of computer graphics, the projection plane is a screen composed of pixels. The principles behind ray tracing are very simple. For every pixel on the screen, we trace a ray from some camera point and into the scene. Then we check what (if any) objects the ray intersects. The nearest intersected object (to the camera point) is then used to inform the colour of the pixel for which the ray was traced. The position of the pixel on the screen also informs the direction in which the ray is traced. If we were to place the screen in front of the camera point, then the ray would appear to intersect the centre of the pixel in question. Figure 1 illustrates what this looks like.

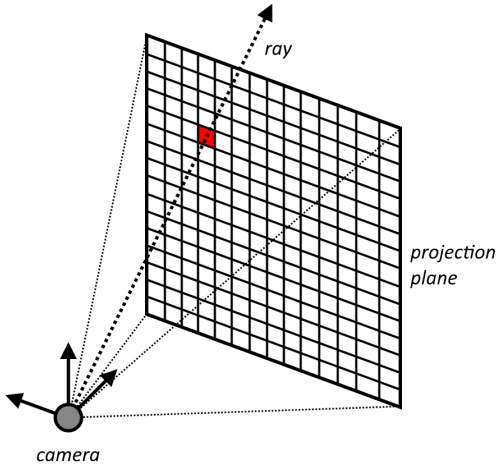

Figure 1 – Ray tracing visualized.

The traced rays are similar to rays of light. However, these rays are traced backwards because they originate, rather than terminate, in the eye (camera). Because the rays emulate photons, ray tracing is an effective means of simulating many real-world lighting phenomena. Although due to time constraints, this project limits itself to simulating shadows and approximating diffuse and specular reflection with the Phong equation[1].

1.2 Problems and Goals

Tracing an individual ray is not necessarily very computationally demanding. The complexity of tracing an individual ray is O(*f*), where *f* is the number of triangular faces (of which objects are composed) in a scene. If the faces are stored in tree-like structures, indexing into them to find faces can be reduced to O(log *f*) on average. However, tracing a ray for every pixel on a screen is very taxing. For example, a 960x540 screen necessitates the tracing of 518,400 rays for every frame.

Furthermore, to simulate shadows, additional rays must be traced from the ends of every initial ray to every light in the scene. This may increase the number of traced rays by orders of magnitude. In ray tracers which support reflection and refraction, rays might be recursively traced from the ends of other rays. This is extremely taxing to perform sequentially.

Normally one would solve this problem with a GPU, because ray tracing is immanently parallelizable. However, this project takes a different approach. This project aims to speed up ray tracing by separating the act of ray tracing from the act of drawing a frame. To accomplish this, the project distributes ray tracing to a pool of worker nodes and collects the results on a master node. The project's ultimate goal is to leverage speedup from distribution and use it to display ray-traced video.

2 System Architecture

This project's sequential and distributed implementations share many components in common. Figure 2 is a legend shared by both implementations which describes their components and the kinds of relations which exist between the components.

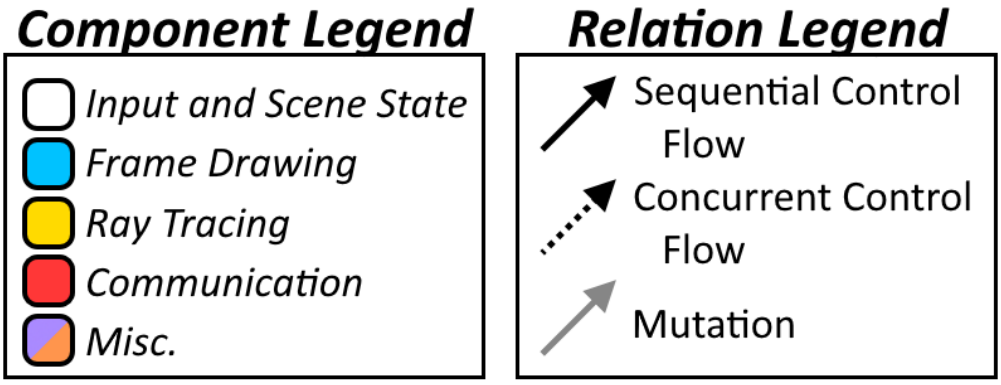

Figure 2 – Shared legends.

2.1 Sequential

The sequential (single-node) implementation is quite simple as ray tracers go. The ray tracer itself is called by a frame-drawing loop, which itself is called by an input-gathering loop. Scene data is loaded from a JSON file provided by the user, and mutated as input is collected. Figure 3 demonstrates the components of the sequential implementation and how they relate to one another.

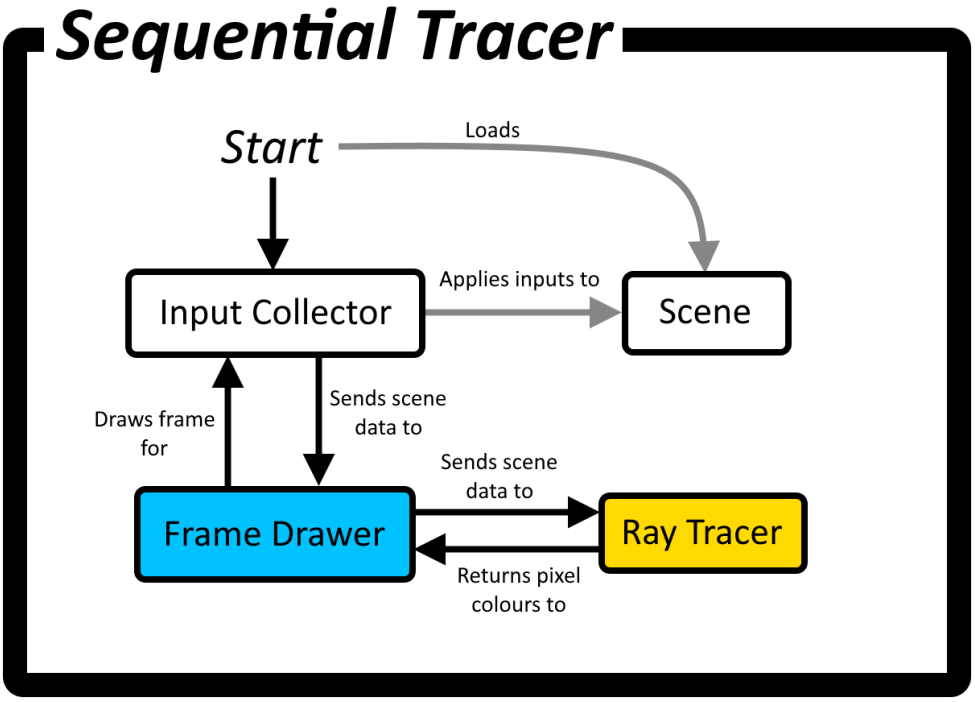

Figure 3 – The sequential implementation.

2.2 Distributed

The distributed (multi-node) implementation is decidedly more complex. At its most basic, the system is composed of a master node and a dynamically-sized pool of worker nodes. The master node is the singular node which interfaces with the user. No other nodes can assume the role of master for this reason. Behind the master node are the worker nodes. These nodes are responsible for tracing rays. To permit the system to scale with increased work loads, the number of worker nodes in the system may change at any point. Figure 4 shows the distributed implementation at its most abstract.

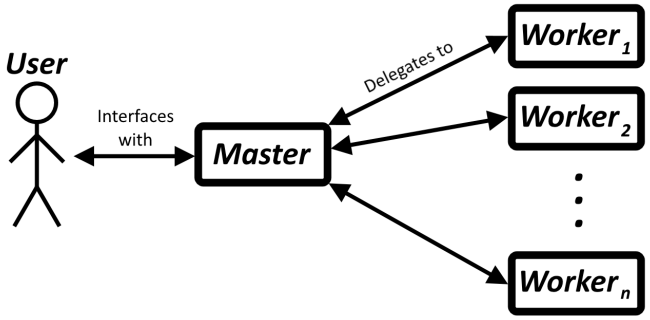

Figure 4 – The distributed implementation abstractly.

The responsibilities of the master node include accepting user input, storing scene state, drawing frames to the screen, and keeping tabs on the system's workers. The master does not trace rays itself. Instead, the master uses a gRPC service to delegate the tracing of rays to worker nodes. The master uses another gRPC service to discover new workers. Every time a new worker is discovered, they are

sent some (immutable) scene information, and their IP and contact port is stored in a worker pool. This information is accessed by frame coordinators when new frames are being drawn. Every frame has an associated frame coordinator who is responsible for drawing the frame. However, because network latency can introduce hundreds of milliseconds of delay, frame coordinators must be initialized concurrently. This ensures smooth frame rates, but it forces each coordinator to synchronize with the next coordinator. That synchronization keeps the frames from being displayed out of order. Figure 5 illustrates the components of the master node.

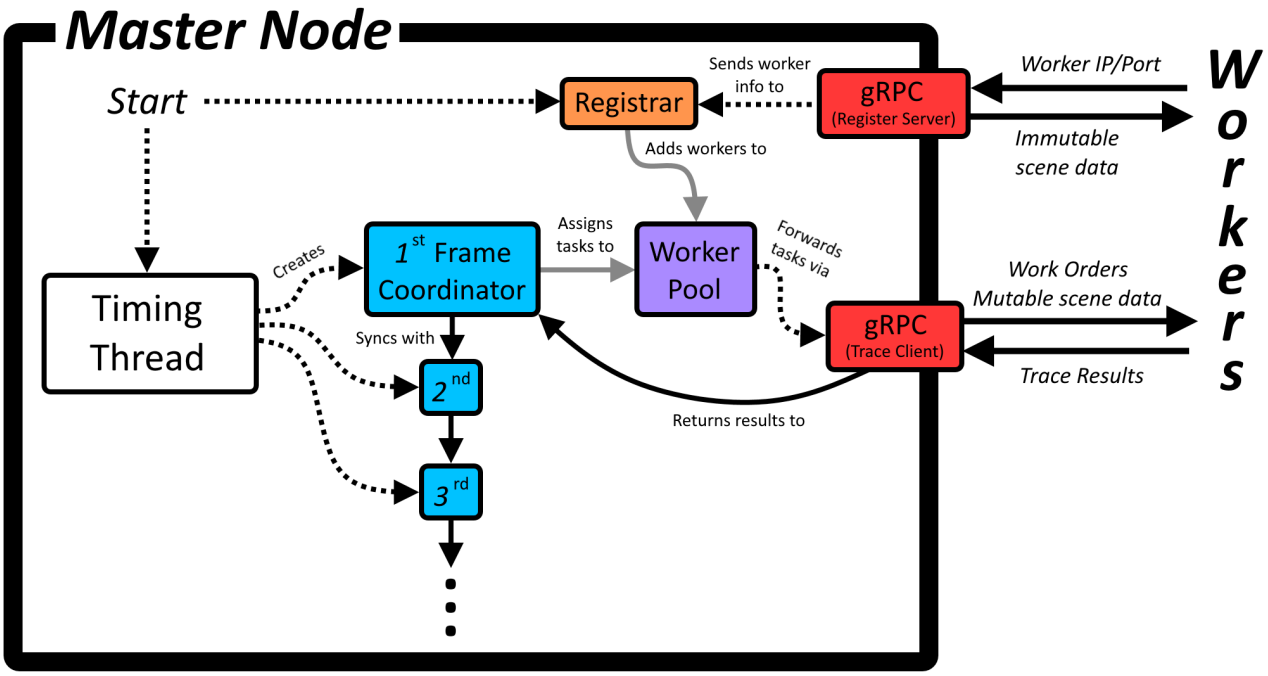

Figure 5 – The master node in detail.

By contrast, the workers are much more simple. When a worker starts up, it enters a registering state. The worker will attempt to contact a master node and provide it with its working port. Once it succeeds, the worker will store the immutable scene data it received from the master, and begin accepting work orders. Work orders contain the following information: a screen offset position, a width and height, and the mutable scene state for the order's frame. This information is used to trace rays, and the results are passed back to the master via gRPC. The worker never mutates any of the scene's data. This gives the master a monopoly on the scene's state, and allows us to handily sidestep issues of distributed consensus. Figure 6 shows the internals of the worker node.

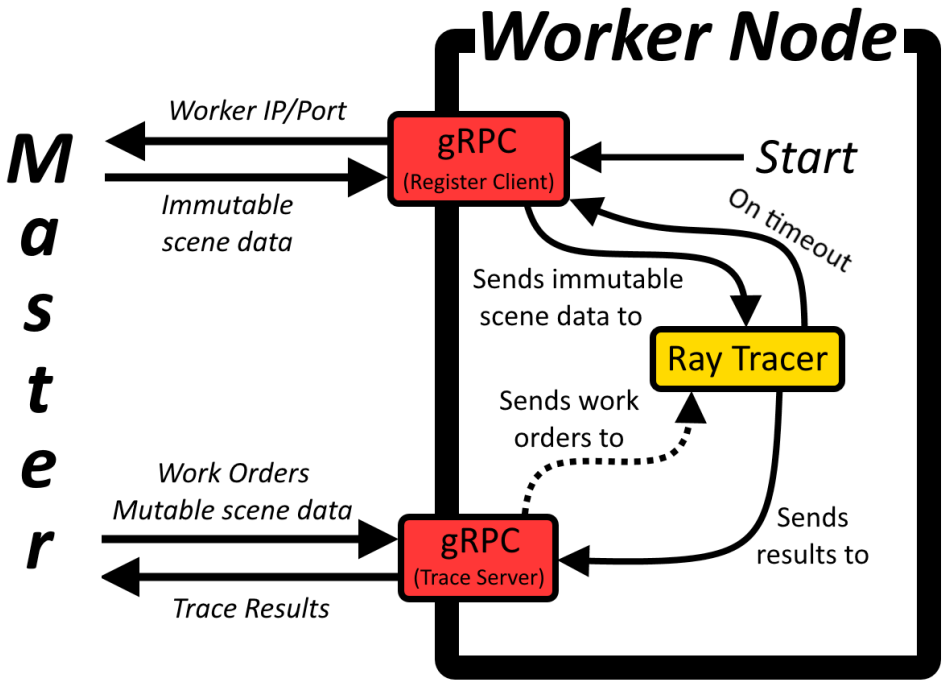

Figure 6 – A worker in detail.

2.3 Languages and Libraries

Both the sequential and distributed implementations are written in Go. Go's goroutines enable excellent concurrency control, and as Figure 5 demonstrates, this is very important for the distributed master. In Go, gRPC (which this project makes use of) also uses goroutines to spin off the RPC service handlers[2]. This enables additional concurrency, even on the worker nodes. Because the distributed implementation uses gRPC, it also makes use of protocol buffers. Protocol buffers help specify the formats of the RPCs exchanged between the master and workers.

Both implementations also share several libraries. For drawing pixels on the screen, both implementations use Go bindings to the SDL2 library[3]. SDL2 enables faster pixel manipulation than vector graphics libraries and interfaces like OpenGL. In a ray tracer it is necessary to have quick pixel manipulation, because every pixel is coloured individually by a traced ray. This project also makes use of a Wavefront OBJ file reader written in Go[4]. This library is used to load the 3D models of objects included in a scene. The library was also modified for this project in order to parse ambient and specular reflectivities from the material files associated with an object. The project also uses an implementation of R-Trees in Go[5]. R-Trees enable sub-linear (O(log *n*) on average[6]) indexing into (triangular) face and object data in our scenes. This library was further modified to allow the user to perform arbitrary intersection checks with the bounding boxes of the tree's nodes (rather than just box-box intersections tests). This was necessary to check whether rays intersected any of the faces or objects in the tree.

2.4 Building the System

Both the sequential and distributed implementations are available on Github at the following link:

https://github.com/MWindels/distributed-raytracer

To build and run this system yourself, you will need the Go compiler. You will also need to install packages for SDL2. Installation instructions for a variety of operating systems can be found at:

https://github.com/veandco/go-sdl2#requirements

Then, use `go get github.com/mwindels/distributed-raytracer/...` to install the system and all its dependencies. To build parts of the system, call `make` (with the appropriate target) on the repository's makefile. On Windows operating systems, you will also need to include the `SDL2.dll` file with the master and sequential executables.

3 Testing and Tradeoffs

In this section, we compare and contrast the performance, scalability and reliability of the project's sequential and distributed implementations. Each subsection's tests use frames per second as their primary metric, because achieving speedup is this project's primary concern. Every test in this section used the same scene featuring the Stanford bunny, composed of 4968 triangular faces[7]. Furthermore, the master and sequential nodes in every test were the same two-core Asus T300 Chi.

3.1 Testing Platform

Google Compute Engine (GCE) was used to test the distributed implementation. GCE was invaluable to this project, because it provided worker nodes with which to test the distributed implementation. For testing, it was crucially important that these worker nodes were separate from one another. If compute resources were shared between workers, then we would not get the parallelism that we need in order to speed up ray tracing. For this reason, containers would not have sufficed. The workers in these tests were part of managed instance groups created using GCE. This ensured worker separation, and also enabled failures and partitions to be simulated by deleting members of the group.

3.2 Performance

To compare the performance of the sequential and distributed ray tracers, tests used a 320x240 screen, and the camera was rotated around the Stanford bunny at (roughly) one unit of distance away. The distributed ray tracer used 24 workers, each with four virtual CPUs. None of these workers were redundant (they each drew a unique part of the screen). Based on these tests, it appears that the distributed ray tracer handily outperforms the sequential ray tracer under normal network conditions. In a typical test, the distributed implementation achieved a mean of 25.16 frames per second, and a median of 26.69 frames per second. Out of 1328 total frames, only 20 were dropped due to workers taking too long to produce their results. By contrast, the sequential implementation achieved a mean of 4.29 frames per second, and a median of only 2.67 frames per second. Figures 7 and 8 show the distributions of frames per second over the duration of the tests. Note that Figure 8 contains fewer frames because the frames took longer to produce.

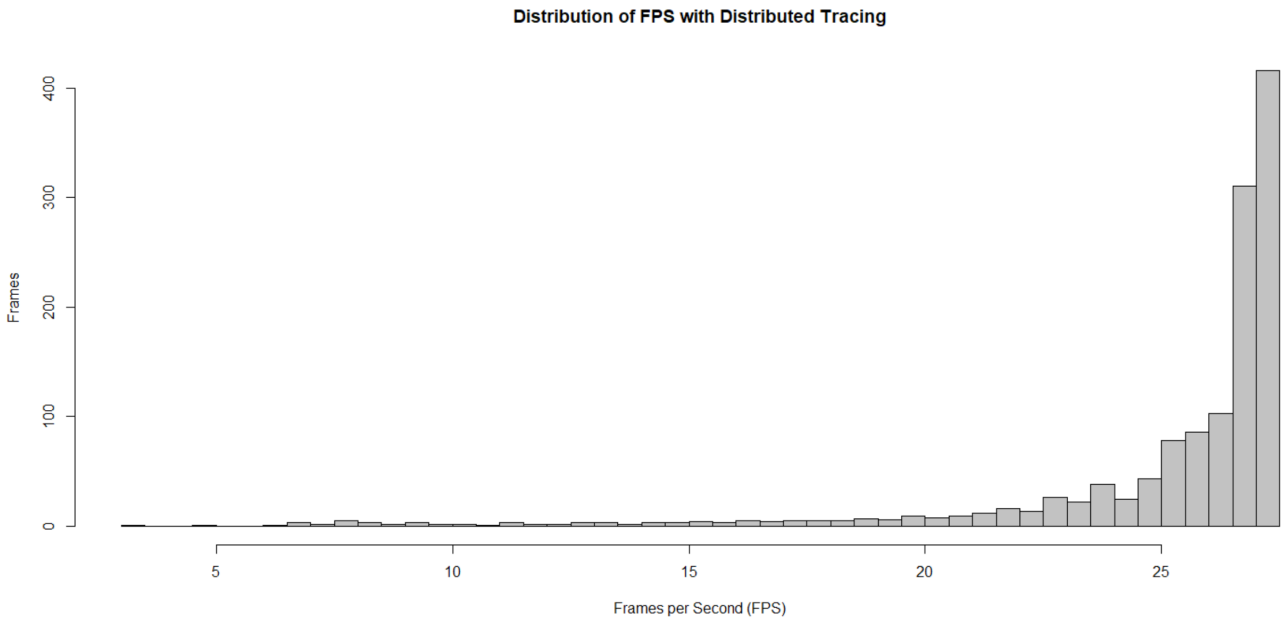

Figure 7 – Distribution of FPS in the distributed test.

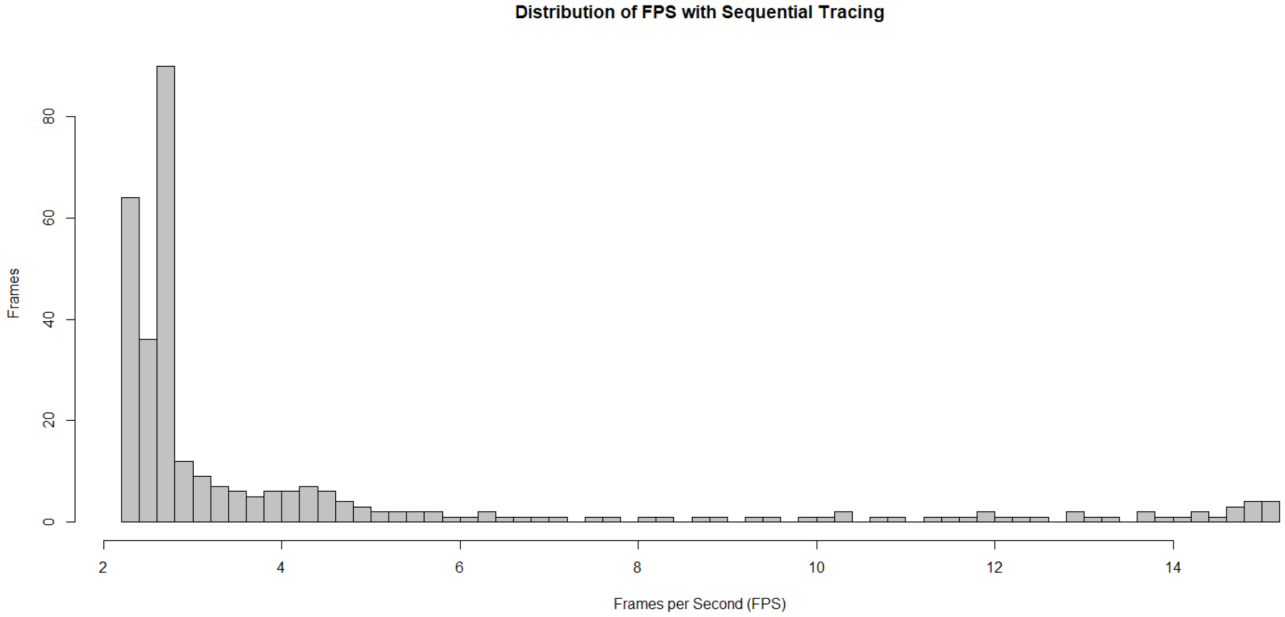

Figure 8 – Distribution of FPS in the sequential test.

Although the distributed implementation tends to outperform the sequential implementation, both ray tracers experience slow-down as they approach objects in the scene. This is likely because the objects take up more screen space as they are approached. Therefore, more rays intersect with the bounding boxes of the object's R-Trees (containing their faces). Thus, and thus more intersection checks must be performed.

3.3 Scalability

The distributed ray tracer's performance edge comes from its ability to scale, which the sequential implementation fundamentally lacks. This scaling is achieved by adding more workers to the worker pool. The same scaling can also be achieved by adding more powerful workers to the pool. Table 1 shows how as the number of (virtual) CPUs in the worker pool increases, the mean and median frames per second also increase. The values in Table 1 were derived from a series of tests with a 320x240 screen, and no redundant workers.

vCPUs / Workers	*Mean FPS*	*Median FPS*	*Frame Drop Rate*
(1 per worker) 24 / 24	*21.48*	*22.53*	*3.39 %*
(2 per worker) 48 / 24	*23.82*	*24.70*	*2.15 %*
(4 per worker) 96 / 24	*23.32*	*24.72*	*21.63 %*

Table 1 – Frames per Second (FPS) as the workers scale vertically.

The mean and median in the last row of Table 1 were likely impacted by the camera getting too close to the Stanford bunny. This is corroborated by the last row's heightened drop rate. Because the workers were tracing rays on a two second timeout, the high drop rate indicates that workers were taking longer than normal to trace their rays. In spite of this, the mean and median are still quite close to those of the previous test. This indicates that the additional vCPUs still had an impact in spite of the increased proximity to the bunny.

Although Table 1 suggests that vertically scaling the workers has a positive affect on speedup, Table 2 demonstrates that vertical scaling at the expense of horizontal scaling does not increase speedup. This is because as the number of workers decreases, each remaining worker is forced to trace rays for more of the screen. Although the remaining workers are stronger, they end up getting saddled with more work.

vCPUs / Workers	*Mean FPS*	*Median FPS*	*Frame Drop Rate*
(8 per worker) 96 / 12	*24.06*	*25.17*	*3.28 %*
(16 per worker) 96 / 6	*26.45*	*25.72*	*0.00 %*
(32 per worker) 96 / 3	*23.51*	*24.20*	*3.35 %*
(96 per worker) 96 / 1	*22.43*	*22.75*	*5.93 %*

Table 2 – Frames per Second (FPS) as the workers scale at the expense of the pool's scale.

3.4 Reliability

Unlike the sequential ray tracer, the distributed tracer is susceptible to partitions and worker failures. In the worst case, the master could even be partitioned from all of the workers. If this happened, the master would be forced to skip every frame until the partition heals. However, the

master does still have a means to cope with partial worker failures. These are “redundant workers.” Redundant workers are workers assigned to trace the same rays (on the same frame) as another worker. In the event that one of the workers fails to deliver their results, a redundant worker can return its results for the same pixels. To keep the frames from being delayed, these redundant workers are sent their work orders concurrently with the other workers.

Figure 9 demonstrates a test which started with 24 workers (with four vCPUs each). However, during the test the worker pool was halved twice. First to twelve workers, then to six. The test used a 160x120 screen, and every worker had one redundant worker alongside them.

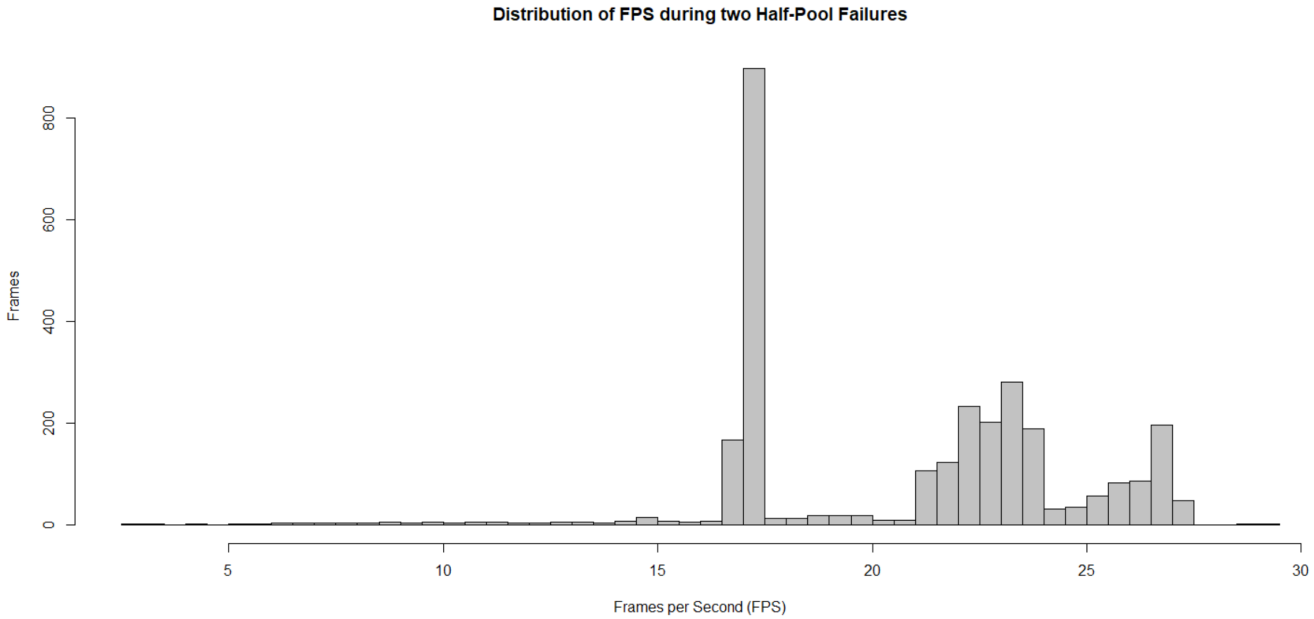

Figure 9 – The distributed ray tracer during two half-pool failures.

Notice the almost trimodal distribution of FPS values in Figure 9. Near 26.5 FPS is the first mode, when all 24 workers were running. Then at around 22.5 FPS is the second mode after twelve workers failed. Finally, at approximately 16.5 FPS is the third mode, once six more workers had failed. This third mode is largest (in part) because the test continued to run for a while after the second mass failure. The other two modes also likely have a little overlap due to variance in the proximity between the camera and the Stanford bunny. Although losing workers has a noticeable impact on frame rate, as long as some workers continue to function, the system continues to function.

4 Future Work

During one test which used a single worker with a shared vCPU, the worker crashed and burned after a few dozen work orders were issued. After the desired number of frames per second was reduced from 30 to one, the worker was able to trace rays just fine. It turned out that the master had been bombarding it with too many work orders for it handle. In order to improve the distributed ray tracer's reliability, it may be worthwhile to devise a means to slow down the master when workers are being overwhelmed. Furthermore, one could also dynamically set timeouts on workers corresponding to their capacity to trace rays, and the number of rays they need to trace.

Other tests also revealed that it is extremely inefficient for workers to trace each ray in their own goroutine. This may be because thousands of goroutines were being spawned, and their overhead was too great. Future work should try to discover whether there exists a happy medium between sequentially tracing each ray in a work order, and concurrently tracing each ray. For example, batching the rays of a work order and tracing the batches concurrently.

Future work could also focus on creating a better screen-partitioning algorithm. The current algorithm simply bisects the screen recursively until no workers are left to fill each half. This can result in increased work loads for some workers who get parts of the screen that are full of objects, and negligible work loads for others who get to trace through empty space. A better screen-partitioning algorithm might sample the screen at regular intervals, and use the presence or absence of objects to decrease or increase the size of screen partitions respectively.

5 Conclusion

Tracing a ray is simple, tracing hundreds of thousands of rays is difficult. By distributing ray tracing among a pool of worker nodes, we leverage parallelism to speed up the process. With sufficient workers, we can even achieve video-like frame rates. However, distribution brings with it reliability issues. The network becomes the arbiter of your frame rate, or whether you have frames at all. In practice though, we mitigate the uncertainties of the network through redundancy. Though there is much that could be done to improve this distributed real-time ray tracer, it still successfully speeds up ray tracing.

6 References

[1] The Phong equation:
https://en.wikipedia.org/wiki/Phong_reflection_model.
[Accessed August 10, 2019].

[2] Concurrency in gRPC:
https://github.com/grpc/grpc-go/blob/master/Documentation/concurrency.md#servers.
[Accessed August 10, 2019].

[3] Go SDL2 bindings: https://github.com/veandco/go-sdl2. [Accessed August 10, 2019].

[4] Go Wavefront OBJ parser: https://github.com/MWindels/gwob. [Accessed August 10, 2019].

[5] Go R-Tree: https://github.com/MWindels/rtreego. [Accessed August 10, 2019].

[6] R-Tree performance: https://en.wikipedia.org/wiki/R-tree. [Accessed August 10, 2019].

[7] The Stanford bunny:
https://github.com/McNopper/OpenGL/blob/master/Binaries/bunny.obj.
[Accessed August 10, 2019].