Trinity University
Digital Commons @ Trinity

Computer Science Honors Theses
Computer Science Department

5-2021

A Study in Akka-based Distributed Ray-tracing of Large Scenes

Kurt D. Hardee
Trinity University, kdavidhardee@gmail.com

Follow this and additional works at: https://digitalcommons.trinity.edu/compsci_honors

Recommended Citation

Hardee, Kurt D., "A Study in Akka-based Distributed Ray-tracing of Large Scenes" (2021). *Computer Science Honors Theses*. 59.
https://digitalcommons.trinity.edu/compsci_honors/59

This Thesis open access is brought to you for free and open access by the Computer Science Department at Digital Commons @ Trinity. It has been accepted for inclusion in Computer Science Honors Theses by an authorized administrator of Digital Commons @ Trinity. For more information, please contact jcostanz@trinity.edu.

A Study in Akka-based Distributed Ray-tracing of Large Scenes

Kurt D. Hardee

Abstract

This project creates a ray-tracing and geometry distribution framework through an actor model of parallelism, which is then expanded onto a cluster of machines to show effective data distribution across a network. This is shown to be feasible, but due to problems internal to the actor framework, as well as design failures, fails to effectively and consistently increase usable memory and generate larger ray-traces, though generally scaled well. Despite this, it compares several methods of ray organization across the geometry and shows that more complex methods generally scale better with the amount of geometry. A photometric renderer was added with very little modification, showing the generality of the geometry distribution framework, and the performance benefits of alternative serialization methods are shown to outweigh the drawbacks of more difficult implementation.

Acknowledgments

This project could not have been possible, and would have halted entirely hundreds of times over, if not for the tireless work and support from Dr. Mark C. Lewis of the Trinity University CS Department. As this project's advisor, he dealt with me at every roadblock, every overworked week. As a professor, he taught and pushed me from the first day at Trinity. He deserves all the thanks in the world for putting me where I stand today, and beyond.

Similarly, this project could not have been completed if not for the love and support of my family, my brother Jude, my sister Alex, my parents Kirsten and Randy. I thank them dearly for their help, and I dedicate this to them. They are the best friends and biggest supporters I will ever have, and could ever hope for.

Few people saw—or heard—the frustrations and tribulations of this project more directly than my friends and girlfriend. This thesis could not have happened if not for the direct support and companionship of my girlfriend, Bella, nor without the sounding-board/peanut-gallery for my grievances and complaints, my friends Kevin, Christine, Josh, Grant, Oliver, Alexander, Nathan, Long, and Duy.

A Study in Akka-based Distributed Ray-tracing of Large Scenes

Kurt D. Hardee

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation.

April 23, 2021

Thesis Advisor | Department Chair

Associate Vice President
for
Academic Affairs

Student Copyright Declaration: the author has selected the following copyright provision:

[X] This thesis is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs License, which allows some noncommercial copying and distribution of the thesis, given proper attribution. To view a copy of this license, visit http://creativecommons.org/licenses/ or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

[ ] This thesis is protected under the provisions of U.S. Code Title 17. Any copying of this work other than "fair use" (17 USC 107) is prohibited without the copyright holder's permission.

[ ] Other:

Distribution options for digital thesis:

[X] Open Access (full-text discoverable via search engines)

[ ] Restricted to campus viewing only (allow access only on the Trinity University campus via digitalcommons.trinity.edu)

A Study in Akka-based Distributed Ray-tracing of Large Scenes

Kurt D. Hardee

Contents

1 Introduction **1**

2 Background **3**

2.1 What is Ray-tracing? 3

2.2 Akka and Actors 4

2.3 Prior Work 5

3 Ray-tracing and Actorization **8**

3.1 SwiftVis2 as a Basis 8

3.2 Minimum Possible Prototype 9

3.3 Breaking up `castRay` 10

3.4 `GeometryOrganizer` Version 13

3.5 Baseline Benchmarking 15

3.6 Creating a Cluster 17

3.6.1 Basic Breakup 18

3.6.2 Loading Data and `GeometryCreator` Class 19

3.6.3 Order of Creation 19

4 Performance and Generality **21**

4.1 Benchmarking 22

4.2 Serialization 24

4.3 Photometry 28

5 Conclusion and Future Work **33**

List of Tables

3.1 Baseline benchmarking runs done with the single-machine actor ray-tracer implementation. 17

4.1 Benchmarking runs done with the nine-machine Pandora cluster ray-tracer implementation. 22

4.2 Benchmarking runs done with the sixty-seven-machine cluster ray-tracer implementation. 23

4.3 Benchmarking runs done with the Pandora cluster comparing between serializers. 27

4.4 Benchmarking runs done with the Pandora cluster comparing between schemas for photometry. 29

4.5 Benchmarking runs done with the Pandora cluster comparing between initial numbers of photons for photometry. 31

List of Figures

3.1 The `ActorSystem` hierarchy at the end of Section 3.2. This figure was originally created for [18]. 10

3.2 The `ActorSystem` hierarchy at the end of Section 3.3. This figure was originally created for [18]. 12

3.3 The `ActorSystem` hierarchy at the end of Section 3.4. This figure was originally created for [18]. 16

3.4 The Cluster hierarchy at the end of Section 3.4. This figure was originally created for [7]. 18

4.1 A broken 90 cell ray-trace, caused by Java.net failure to resolve the hostname "cs.trinity.edu". Note the black swaths caused by unloaded geometry. . . . 25

4.2 The clustered photometric rendering actor hierarchy at Section 4.3. 28

4.3 A photometric rendering of 10 cells, generated using the Some schema and 25,000,000 photons. This image originally appeared in [7]. 30

Chapter 1

Introduction

This project began with the planetary ring simulations of Dr. Mark Lewis, wherein he uses numeric simulations of rings as collections of particles, using hard-shell collisions and self-gravitation in order to effectively simulate the forces that create the ring structures observed by probes and telescopes. In order to actually observe the results of these simulations, visualization is necessary, and one of the methods used is ray-tracing. This visualization technique is embarrassingly parallel, but requires all geometry to be loaded into memory simultaneously. This relatively high memory bandwidth, combined with the fairly huge number of particles required for these kinds of numerical simulations, came to a head when Dr. Lewis was contacted by the Hayden Planetarium for its "Worlds Beyond Earth" planetarium show[15], for which they wanted a huge ray-trace. This ray-trace ultimately began to become problematic, as even on the 64 gigabytes of memory on his simulation machine, he began to run out of memory attempting to load sufficient amounts of geometry into memory. As a result of this, as well as the teaching of Akka to lower-division students, the idea of a project to attempt to use Akka Clustering to create a ray-tracer capable of distributing the data across a cluster of machines, thereby removing the memory-bound

nature of ray-tracing formed.

Later, along with exploring various implementation details, the idea of adding other visualization methods was proposed. As the main focus was on the geometry division, rather than the visualizer, this was added, and photometric renderings were added, due to their useful nature for simulated observation work, like those done with Dr. Lewis's simulations.

Chapter 2

Background

2.1 What is Ray-tracing?

Ray-tracing, the visualization technique used for the majority of this project, is a visualization methodology that is embarrassingly parallel, but not embarrassingly distributable. The technique begins by establishing a point and a plane, along with loading in a geometry to ray-trace. The point, known as the eye, is the place where rays begin from, while the plane, called the view plane, sits between the eye and the geometry and contains the complete image to be drawn. The process, then, begins by sending a series of rays from the eye through the view plane and determining which of these rays hit geometry. Afterwards, a new ray is created at each spot of intersection and sent to each light source, and if these rays reach the light source without an intersection with another piece of geometry, the color is determined by the reflectivity of the geometry and the color and distance from the light source.

This system, while somewhat costly in resources, is fairly accurate and can well simulate various geometries, such as those seen in these kinds of ring simulations. There are also

large upsides to such a schema, most notably its ease of parallelism. To parallelize such a schema, all that must be done is to choose every (x,y) pixel in the view plane, and at each of these points, send a standard number of rays, then do the normal back-end calculations concurrently. This concurrency works due to the relative lack of interference between points on the view plane, meaning all these calculations do not have to take into account the results of any of the others. This trait, in which large portions of the code can be broken up into computations that do not require results from each other, and are therefore non-blocking, are referred to as "embarrassingly parallel" [16]. While the concurrency is embarrassingly clear, the clarity of data distribution is much less embarrassing. Due to the system previously established, the entire geometry to be ray-traced must be loaded into memory simultaneously, as any concurrent thread may need to access any part of it at any time, especially for the second ray-cast, which could potentially go through any other part of the geometry. As such, any schema for clustering must be able to section the data for distribution, such that rays can be cast across several machines without needing to know what specific data is held on any given machine.

This project will specifically attempt to deal with the data distribution problem, using Akka [20], an actor-based concurrency and distribution library using the JVM, and Scala, a type-safe JVM language allowing for a large amount of compile-time error handling and easy expandability.

2.2 Akka and Actors

This project, in using an actor-based concurrency library, leads to some potential roadblocks and pitfalls, as well as very clear advantages over other methodologies. Akka, specifically, uses an actor-based system to deal with safe concurrency. In this system of actors, each

class is replaced with an "actor" which contains a mailbox, which is a simple queue of pieces of data, and asynchronously sends and receives messages, and makes decisions based on messages received. The typical life-cycle of an actor is to take the first message out of the queue, do a predetermined action based on the type of the message, and potentially send one or more messages to another actor, often back to the actor who sent the message, before moving onto the next message in the mailbox. This dynamically-scheduled and asynchronous message-passing system leads to swift concurrency, and importantly, the data contained in any actor can only be accessed by the actor in relation to one message at a time, making race conditions incredibly uncommon and unlikely. They can still happen, but this is most often caused by an abusive use of an actor model, such as shared mutable data between actors.

Akka itself also has several main advantages that make it ideal for such a project. Particularly, Akka, using their base actors, has several networking expansions, most notably Akka HTML and Akka remoting, which themselves form the backbone for Akka Cluster, which uses one of several network protocols, including TCP/IP and UDP, to distribute systems of actors across a network, forming a cluster of machines. Akka Cluster will form the main part of this project's distribution methodology[20].

2.3 Prior Work

In preparing for this project, several similar endeavors were found to parallelize ray-tracing work, though relatively few with distributed data. The majority focused on concurrency, and one particular survey focused heavily on the timing results of several such attempts at concurrent ray-tracing of isosurfaces, or the exterior surfaces of geometry [12, 13]. Another focused heavily on timing results of several commercial raytracers, such as VisIt, Par-

aView, and Manta, and talked heavily about the potential advantages of GPU ray-tracers [3]. Manta, in particular, uses SIMD instructions as well as modern C++ parallelism to markedly speed up the ray-tracing problem, but has no support for data distribution [2]. GPU ray-tracers, on the other hand, in addition to being potentially much faster due to their incredibly large numbers of concurrent threads, were also proven to be completely viable, as another paper studied the useability of NVIDIA's Thrust data parallel primitives for ray-tracing, creating a completely viable ray-tracer with them [10].

The known research on these topics that come closer, however, deal with the similar problem to ours: memory constraints and visualization. Perhaps the most unique solution found was to use a new dynamic scheduling algorithm which first divides the space into domains, and duplicates and divides the domains if there is a disproportionately high concentration of rays in a single domain, then slowly moves to a domain-decomposition style of dividing the data. As such, their algorithm limits the ray memory usage and helps fit bigger ray-trace environments onto the same amount of memory [14]. This, too, however is intrinsically limited to non-distributed applications, and may be an interesting addition at a later time. Several other projects attempted, however, to distribute a ray-trace across a cluster of Linux machines, one using BVH packets and another used a system of supervisor and slave applications, however, unlike this project, these systems were very heavily specific to their local domains and were not easily adapted to other spaces, nor easily generalized for other visualization applications [4, 9]. Easily the most applicable paper, however, was a 1997 attempt by several researchers to use an actor model and C++ to distribute a rendering across a cluster of machines, as well as demonstrate its scalability to expand the cluster [8]. Despite this, their visualization was a simple rendering of a known image, and their relatively specific solution, their program was not portable or expandable in the way that a library-based solution, such as Akka, can be, nor can it be as portable as a JVM-based

solution can be.

Chapter 3

Ray-tracing and Actorization

This project began by making an actor ray-tracing library capable of ray-tracing the geometric data already available. As such, work began by breaking down the methodology used in SwiftVis2's ray-tracer packages [11]. These packages, in addition to being known to be compatible with the data, also already had many efficiency optimizations, such as the addition of kD-tree storing of particles [1].

3.1 SwiftVis2 as a Basis

SwiftVis2's ray-tracing package revolves around the `castRay` function. This function uses a known set of geometry and takes a Ray in order to return a color to be assigned to that pixel. First, the program loads in a binary file containing all the pieces of geometry into memory. Once all this is loaded, the program is passed the parameters—the eye and the view plane—from the source code, and begins the ray-trace by generating rays from the eye through each pixel of the view plane. Each of these rays is then, in parallel, sent towards the geometry and put into a function, called `castRay`, that then returns back the color to

be assigned to that pixel. This version will serve as the basis for the first version of the actorized version.

3.2 Minimum Possible Prototype

The first version within Akka was the minimum possible recreation using the actor hierarchy, simply reimplementing the RayTrace files from SwiftVis2 using Akka, relying on the `castRay` function and the Geometry primitives to get an understanding of some of the potential hurdles.

The main program created the `ActorSystem`, then proceeded to go through every pixel on the screen and send a Ray, beginning at the eye and going through that pixel, to the `RTManager`. This `ActorSystem` began by creating an `RTManager` actor which simply contained the Geometry, the light sources, and a router of `RTActor` child actors. Upon receiving a Ray, the `RTManager` sent the ray along to one of its `RTActor` children, with the router sending them in a round-robin schema. These `RTActors` then did a call to `castRay`, returning the resulting color back up to the main program to be drawn to the screen. This implementation effectively parallelizes the ray-tracing work, and could potentially work as a clustering methodology, but since `castRay` requires access to all the Geometry, every machine would need a copy of the entire Geometry, and thus it is insufficient for distributing data.

As work on this prototype continues, a strange hurdle inherent to the heavily object-oriented actor methodology manifested itself: namely, the asynchronous nature of the actor system. Using the normal message-passing methodology, a message is sent and then dealt with asynchronously, with no directly returned value. Originally, attempts were made to subvert this by using the ask pattern, which returns a Scala Future which can be mapped

onto to collapse into a value when there eventually would be one, but due to its non-recommended nature by the Akka documentation, as well as the very high overhead, a new solution was decided upon. The solution is to make it further object-oriented and pass the x and y pixel coordinates with the ray to be cast, so then when messages are passed back up the tree to the image, it has all required information to draw, and all work can be done non-blocking and asynchronously. The actor hierarchy at the end of this section is shown in Figure 3.1.

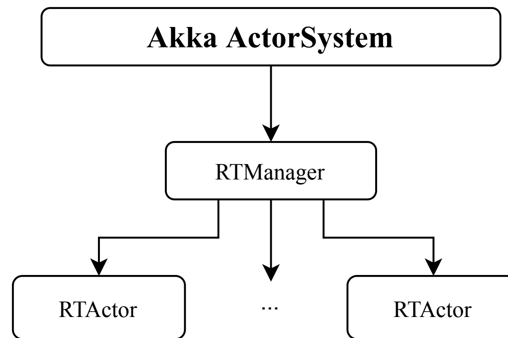

Figure 3.1: The `ActorSystem` hierarchy at the end of Section 3.2. This figure was originally created for [18].

3.3 Breaking up `castRay`

Since the goal was to move to distributed data, the first necessary step was to move away from the `castRay` function used in SwiftVis2. This function, by design, required access to the entire data, as it did every step of the raytrace, including creating new rays. As such, the first step was to break down the function into its basic steps: the first intersection, the potential creation of redirected rays to light sources, and then the potential second

set of intersections. The decision was thus made to split the monolithic hierarchy shown in Figure 3.1 into two separate trees: the first, called the GeometryManagement tree, simply contains the Geometry and casts Rays to the Geometry, and the other, named the ImageDrawer tree, which does all creation and coordination of rays.

The top-level actor in the GeometryManagement tree is the `GeometryManager`, which is roughly analagous to the `RTManager` actor from Section 3.2. Thus, the `GeometryManager`'s main focus is containing the total KD-tree of Geometry and distributing work, which it does to a new child actor: the Intersector. Like the `RTActors`, this child actor type sits in a router and is passed Rays to be cast to the Geometry in a round-robin schema. Unlike the previous iteration, however, these Intersectors do not use the `castRay` function, instead doing a simple intersection check between the Ray and the Geometry, an inherent function to the Geometry supertype, which returns back a monad, called Option, containing either None, containing no data, or Some[IntersectData], which contains all the relevant information about the intersection, such as the time and point of intersection. It then returns this monad back to the sender and works on the next Ray.

By contrast, the top-level actor in the ImageDrawer tree, the ImageDrawer actor, has no direct predecessor in the previous iteration. This actor functions as the only thread with direct access to image drawn to the screen, and thus contains all the relevant drawing data, as well as the light sources. When all the data is loaded and ray-tracing is ready to begin, the ImageDrawer creates many child actors, called PixelHandlers, with exactly one assigned to each Pixel on the screen.

These PixelHandlers each create a Ray to be cast and send the Ray to the `GeometryManager` and waits for the returned Option. Upon receiving back this Option, the PixelHandler determines the type of the monad and works accordingly: if the None is returned, indicating no intersection, the background color of the scene (usually black) is sent to the ImageDrawer to

be drawn to the screen, along with the pixel coordinates assigned to the PixelHandler. Conversely, if the monad contains a Some, and there was thus an intersection, the PixelHandler creates exactly one child actor, called the LightMerger.

The LightMerger actor begins by creating new Rays, beginning from the point of intersection of the previous step, towards each light source, and sends them to the `GeometryManager`. It then waits for responses back from the GeometryManagement tree, and for each None returned, it determines the color and intensity of the light and stores this as an RGB value in a buffer. For each Some return, it determines if it reached the light source before the intersection, and if so, does the same color calculation, and if not, it instead stores the background value. Once the buffer contains a color value for every light source, it merges the colors into a single pixel value and sends it back to the PixelHandler to be sent back to the ImageDrawer. This final hierarchy is shown in Figure 3.2.

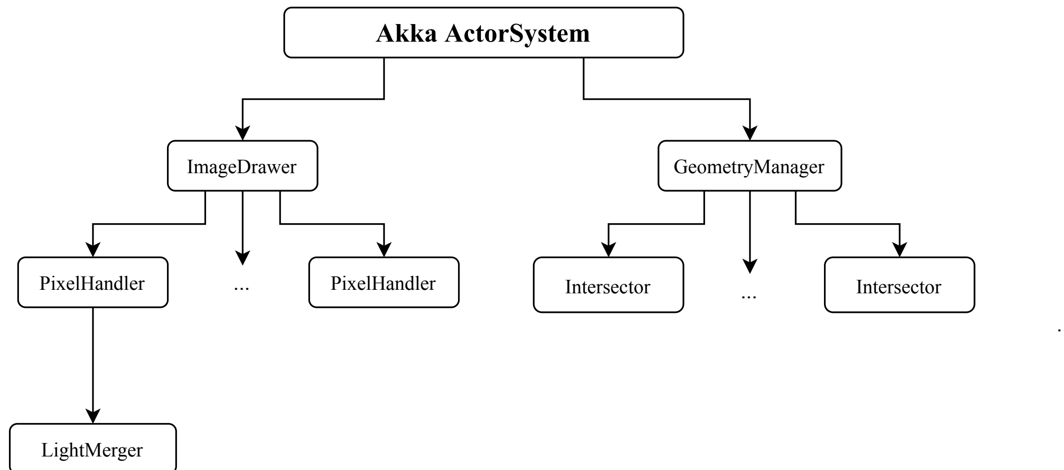

Figure 3.2: The `ActorSystem` hierarchy at the end of Section 3.3. This figure was originally created for [18].

3.4 GeometryOrganizer Version

This system, though an effective actorization of castRay, is clearly still insufficient for the ultimate goal of distributed data, as the GeometryManager and its children contain the entire file as a monolithic unit. As such, the next step was to take the GeometryManagement tree and to begin to split it up, resulting in the creation of a new top-level actor in this tree: the GeometryOrganizer.

The GeometryOrganizer is added above the GeometryManager, with the goal being to create GeometryManagers and split up Geometry, as well as organize where Rays are sent. As such, the GeometryOrganizer is the only actor in the tree with knowledge of where to access the data to be loaded, and splits it up and then creates a GeometryManager associated with each of the sections of Geometry. Then, when the ImageDrawer tree actors send Rays to be cast, instead of being sent to a GeometryManager, they are instead sent to the GeometryOrganizer, which then makes a determination of which GeometryManager (or Managers) to send the Ray to, then responds back based on which Managers were sent to, and sends one total result back to the sender actor. Three main schemata were proposed to control where the Rays were sent, and three GeometryOrganizer possibilities were created for comparison, using these three schemata: GeometryOrganizerAll, GeometryOrganizerSome, and GeometryOrganizerFew.

The simplest schema is implemented in the GeometryOrganizerAll actor. This actor, upon receiving a Ray to cast, simply sends the Ray to every GeometryManager, then creates a key and an associated buffer and moves onto the next message. Upon receiving back an Option from the GeometryManagers, the actor places this Option in the buffer associated with that key, and when the buffer reaches a length equal to the number of GeometryManagers, it filters out all the Nones. If the buffer is empty, or if all returned

Options were None, the GeometryOrganizer returns back a None to the sender. Otherwise, the Organizer, using the time values of the IntersectDatas, finds the first intersection, in order, and then returns this first Some.

The next-simplest schema is, similarly, implemented in the GeometryOrganizerSome actor. Unlike the All organizer, the Some schema requires slightly more information, and record the bounds of the Geometry loaded into each GeometryManager. As such, upon receiving a Ray to be cast, the actor does a simple intersect check with the bounds of each GeometryManager, and sends the Ray only to each GeometryManager whose bounds the Ray would enter, and thus whose Geometry could possibly be intersected by the Ray. The actor then works very similarly to the All methodology, and associates the Ray with a key and a buffer, and fills the buffer when Rays come in, but instead of waiting for the buffer to reach a length equal to the number of GeometryManagers, it instead stores the number of GeometryManagers it was initially sent to, and waits for this number instead. After reaching that length, the actor then works the same as the GeometryOrganizerAll, and returns back a None in the event of only Nones, and otherwise returns back the Some containing the first intersection.

The most complex schema, working much differently from the other two, is implemented in the GeometryOrganizerFew. The GeometryOrganizerFew begins, like the Some actor, by storing the bounds of all the Geometry for each GeometryManager. Then, upon receiving a Ray to be cast, the actor intersects each set of bounds, like in the Some schema, but unlike the Some schema, it sends the Ray only to the GeometryManager whose bounds the Ray first intersects, and then stores the key associated with the Ray along with a List of the remaining GeometryManagers whose bounds the ray intersects, ordered by the time of intersection. Then, upon receiving the Option back, it checks whether it is a Some or a None. If it is a Some, the GeometryOrganizerFew immediately sends the Some back, and

if it is a None, it instead checks the stored List and pops the head. If the head of the List is nonempty, the Ray is then sent to the Manager contained in the head, and the tail of the List is stored again. If, instead, the head of the List is empty, the actor returns back a None. In this way, the Ray is sent only to one `GeometryManager` at a time, in order, and short-circuits to send back the first Some.

These three schemata cover a wide range of potential advantages and disadvantages. In particular, if message-passing is an expensive operation, Few should theoretically be the fastest, due to its minimum sending of messages possible. Conversely, if single-threaded performance is a bottleneck, the All schema has an advantage of the absolute minimum of single-threaded work. As this will be eventually on a cluster, and messages will be passed over a network, the expectation is that the Few and Some schemata will be faster than All, with a potential edge given to the Some specifically, as it throws out a significant amount of messages that would be entirely useless, while preventing bottlenecks through a single thread. However, it is not clear that this method will be faster in a single-machine use-case, as message-passing is a fairly fast operation. In addition, SwiftVis2 was modified for the Few methodology, by adding a new type of Bounds, BoxBounds, as the initial SphereBounds posed potential problems for Few, in which non-disjoint Bounds can cause erroneous drawings. The final actor hierarchy, including the `GeometryOrganizer` is shown in Figure 3.3.

3.5 Baseline Benchmarking

In order to provide a consistent and scalable benchmark framework, the ray-tracer was given the ability to load in files containing geometry and offset them in space. This was done by the use of the GeometryCreator class, which was implemented much later and will

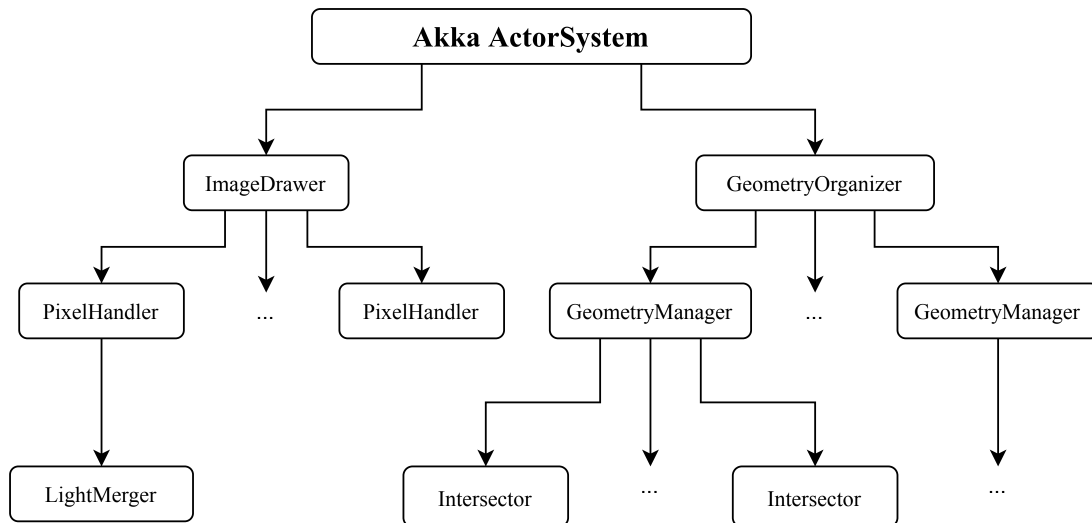

Figure 3.3: The `ActorSystem` hierarchy at the end of Section 3.4. This figure was originally created for [18].

be discussed in Section 4.2. The result, combined with a new view, allows the long, narrow slices of geometry to be tiled into a field that is perfectly square and should view all pieces of geometry equally. However, this square view is only accurately square when there are $10n^2$ pieces of geometry, thus the setup was only tested in these intervals.

This non-distributed implementation was tested using Pandora00, in order to allow for consistency, and the clustered implementations were later tested using the Pandora machines as a cluster. Each machine contains a Intel Xeon E5-2683v4 CPU, with 16 physical cores and 32 threads, as well as 32 gigabytes of memory. All test runs were done three times, and then the results were recorded. Table 3.1 records the mean and standard deviation of each. As 90 files proved too large for the single machine, only the results of 10 and 40 are listed.

What is most immediately obvious about the single-machine implementation of the ray-tracer is the consistent speed of the Few schema. This was faster, by a significant margin,

Times [sec]	10 cells	40 cells
All	221.7157 ± 2.9281	727.5848 ± 15.9751
Some	320.7971 ± 12.4618	593.1772 ± 3.9132
Few	205.5791 ± 2.3950	412.0922 ± 9.3044

Table 3.1: Baseline benchmarking runs done with the single-machine actor ray-tracer implementation.

in both sizes of ray-trace. Meanwhile, the All and Some implementation go back and forth, with each being faster in one case. On the small scale, the All manages times close to the Few implementation, but as more geometry is loaded in, it sends more unnecessary messages and begins to slow down considerably, seeing almost exactly linear performance increases. As messages become serialized and sent across a network, this is only likely to worsen the performance of All. On the other hand, the performance increases for the other two are sub-linear, with both seeing nearly exactly twice as much compute time for a four times as much increase in geometry. This is likely due to the constant amount of pixels drawn, as the image is still 1200x1200 in resolution regardless, and ray-tracers scaling primarily by resolution. What is interesting is that in both cases, Few was faster than the Some implementation, despite scaling very similarly. This will especially be something to focus on when moving to a clustered implementation, as the slow network should only cause the Few implementation, with its minimum messages, to become even faster compared to the other two.

3.6 Creating a Cluster

The next major step was the actual creation of the cluster, using the actor hierarchy established in the previous chapter. Akka makes this relatively simple by allowing Actors to exist on any machine on the cluster and treating them nearly identical to how they exist in

a non-clustered ActorSystem.

3.6.1 Basic Breakup

The first step of creating the cluster is to create the hierarchy of the cluster itself. To this end, we create two types of Actor to encapsulate all the Actors on a single machine. The first of these Actors is the `FrontendNode`, whose job is to contain all the data on the head node of the cluster. In doing so, it contains all the `ImageDrawer` tree, but in addition, the head node must control all the other nodes, and thus it also must contain the `GeometryOrganizer`. Just as with the single-machine implementation, the `GeometryOrganizer` continues to have three implementations based on the All, Few, and Some schemata.

The other main node type in the cluster is the `BackendNode`, which is much simpler. The `BackendNode` contains any number of `GeometryManagers`, one per file assigned to it, as well as all its child Intersector actors. In this way, the most possible memory can be allocated on each machine. This cluster hierarchy can be seen in Figure 3.4.

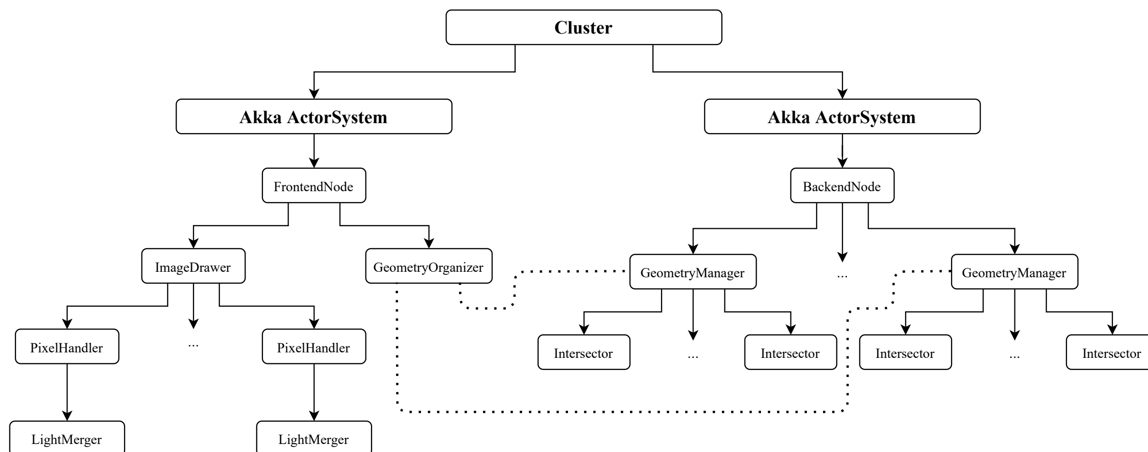

Figure 3.4: The Cluster hierarchy at the end of Section 3.4. This figure was originally created for [7].

3.6.2 Loading Data and `GeometryCreator` Class

In order to prevent potential network bottlenecks and memory usage on the `FrontendNode`, the choice was made for each `BackendNode` to asynchronously load their own Geometry. However, hierarchically the `GeometryManagers` should be directed by the `GeometryOrganizer` to load their Geometry. In order to do this, the `GeometryOrganizer` assigns each `GeometryManager` an offset and a number during creation. In doing so, the `GeometryOrganizer` was then capable of simply sending a function that, by taking that number and offset as parameters, each `GeometryManager` can find and load its own Geometry. Initially this was done in the form of a simple function, however this later caused issues with serialization, so a new container interface was created to deal with this, called the `GeometryCreator`, which has subclass implementations that can be serialized and passed as messages, but contain an apply method that does the same as the original function. The serialization issues that caused this necessity will be discussed later, in Chapter 5.3.

3.6.3 Order of Creation

Creating an Akka cluster is a difficult balancing act and requires a protocol of handshaking messages, sent back and forth, in order to establish the differing nodes' locations in the cluster. This can be done a number of ways, but the way this was accomplished in this project begins with the creation of each `BackendNode`.

Each machine with a `BackendNode` is spun up with a command, and then left to sit. Each machine has, in its configuration, the IP address and port on which to communicate with every other machine. Once all of the wanted `BackendNodes` are created, one `FrontendNode` is started on the head node in the cluster. Each of these `BackendNodes` recognizes that a new node has entered the cluster, and once they recognize it as a `FrontendNode`, they send

a message to the `FrontendNode` in order to register themselves with it. Following this, the `FrontendNode` waits for every `BackendNode` to register with it. Once all the `BackendNodes` are registered, it passes references to the `BackendNodes` to its `GeometryOrganizer`. This `GeometryOrganizer` then, using a round-robin schema, sends messages to each of the `BackendNodes` to create a `GeometryManager` for each file, up to the number of files defined in its configuration, with the given number and offset. Each of these new `GeometryManagers` then send messages registering with the `GeometryOrganizer`, who sends them the `Geometry-Creator` with which to load their data. Once these `GeometryManagers` complete loading their data, they send a message back to the `GeometryOrganizer` containing their bounds. Once all the `GeometryManagers` have returned back their bounds, the `GeometryOrganizer` finally sends a message to the `ImageDrawer` to begin drawing the image. From this point on, the ray-tracing can be done nearly seamlessly as it was being done prior, as nothing must be modified in the `ImageDrawer` tree, which only communicates with the `GeometryOrganizer`, and the `GeometryOrganizer` now has references to the `GeometryManager` actors that it can send messages to, exactly as before. Now that this handshaking is done, the cluster is fully functional and ready to be benchmarked.

Chapter 4

Performance and Generality

For bench-marking, all ray-traces were at a resolution of 1200x1200 pixels. In addition, all photometry was done with a single light source, sending out the listed number of photons. For standardization, a cluster was created using the Pandora machines, which were previously described in Section 3.5. This cluster, referred to afterwards as the "Pandora cluster," had Pandora00 running a `FrontendNode` and the other eight Pandora machines running a `BackendNode` each. One other cluster, only used for some work, was created using Pandora00 for a `FrontendNode`, along with `BackendNodes` on each of the 25 Janus machines, 21 Xena machines, and 20 Titan machines. All three labs of machines have the same amount of memory, 16 gigabytes, and only differ in the CPU. The Xena machines are the newest, with Intel Core i7-8700 processors with 6 cores and 12 threads, while the Janus machines have an Intel Core i7-7700s and the Xena machines have Core i7-6700s, each with 4 cores and 8 threads. This cluster was ultimately unsuccessful, as will be discussed later, and based on this it was only used for a very limited subset of benchmarks.

4.1 Benchmarking

The initial benchmarks, run on the Pandora cluster, are tabulated in Table 4.1.

Time [sec]	**10 Cells**	**40 Cells**
All	258.689 ± 11.250	911.643 ± 103.095
Some	277.586 ± 32.984	403.194 ± 45.241
Few	282.191 ± 10.126	324.981 ± 23.242

Table 4.1: Benchmarking runs done with the nine-machine Pandora cluster ray-tracer implementation.

This system, despite having eight times more computing power, was unable to complete a ray-trace of 90 cells, the next step up. Though this seems counterintuitive at first, one of the largest reasons for this is the serialization of messages. During serialization, messages have at least three copies in memory, thus taking up three times the space, and due to the high number of messages, this adds up immensely. Another major reason is the fact that, on startup, the `ImageDrawer` and `PixelHandlers` send millions of messages effectively instantly, and Akka has an outbox for all messages going across a network. Due to this, the outbox was being overfilled almost instantaneously and would throw away messages, so the size of the outbox was required to be much larger than originally intended. This outbox, thus, reserves a large amount of memory as well. Despite this, the result is disappointing, but what remains true is the advantages of the All schema in small ray-traces. As with the local version, it ended up faster than either of the others in the smallest version - though within the margin of error for the Some schema - but scales almost directly linearly with number of cells. By comparison, the Some and Few implementations scale by fairly small margins between the two data points. In addition, all clustered attempts were slower than their unclustered counterparts which would seem to indicate that despite the increased overall computational power, the cost of networking across the relatively slow one-gigabit

network is higher. Overall scaling comparisons, however, are nearly impossible to make with such a small sample size, so the next attempt was to enlarge the cluster and try for 90 cells.

The first attempts were made by creating clusters across the Janus, and later the Janus and Xena machines, though both of these attempts did not pan out as hoped, as the lower memory per machine meant that even the combination of the two could not accomplish 90 cells. The big cluster was then created, adding the Titan machines, for a total of 67 machines in the single cluster. A small table of results of this cluster are shown in Table 4.2.

Time [sec]	**90 Cells**
All	DNF
Some	631.291 ± 238.799
Few	458.531 ± 109.2741

Table 4.2: Benchmarking runs done with the sixty-seven-machine cluster ray-tracer implementation.

Originally, management of such a large cluster was going to done through the use of Bash scripts and a Fat JAR containing all the source code in an executable format, generated with the Scala Build Tool. Upon actually attempting to run any of the main classes in the Fat JAR, however, it was soon found that due to unknown issues with Akka and SBT, the main classes in SBT were unable to correctly load many aspects of the configuration files, including the default configurations, and would not run in any manner. A solution to this problem was never found, and seems to be long-lasting, as a question on the official Akka Boards showing the same fault was never answered, despite being two years old. As a result, the fallback method was to run SBT manually using Bash scripts, but as SBT is not intended to be run on the same directory simultaneously, this led to continuous issues, varying from

the understandable, such as SBT being unable to find the main class, to the less clear, such as the wrong configuration being spontaneously loaded and used, to the completely unintelligible, such as the underlying Java.net failing to be able to correctly resolve the hostname of "cs.trinity.edu." The last of those errors led to totally unusable ray-traces, such as the one depicted in Figure 4.1 As a result of all these issues, for each timing run necessary for benchmarking, there was an average of 12 failed runs. Even despite all this, the data here proves the capability of this system to expand across a heterogeneous cluster of machines, and do so widely, with the majority of the issues seemingly stemming from our apparent misuse and abuse of SBT. While the Some methodology does appear to be slower, the variance was so large they are well within each other's margin of error. What is, however, interesting is the complete failure of the All schema to complete a successful raytrace, still running out of memory in this huge cluster. This is very likely due to the creation of so many messages, along with their serialization copies, causing the `FrontendNode` to run out of memory despite having the most memory of any machine in the cluster.

4.2 Serialization

One of the more interesting findings during the creation of this project was the alternative serialization methods. Serializing data is a necessary step for any networked project, as the data must be put into a standard format before being sent across a network, often in the form of JSON or a binary file. For this project, this means that every individual message must be serialized and then deserialized when being sent across a network, and prevailing wisdom has held that the standard Java serializer is relatively slow and inefficient. Akka, in particular, recommends the use of a serializer called Jackson[5], and this project allows for a good potential comparison of these serializers. As well, we chose to add a third serializer

Figure 4.1: A broken 90 cell ray-trace, caused by Java.net failure to resolve the hostname "cs.trinity.edu". Note the black swaths caused by unloaded geometry.

in for comparison, called Kryo[19], due to it being the recommended serializer for Apache Spark[6], the other main clustering framework in Scala.

The process of setting up each serializer varied, as different serializers have different capabilities and require different setup to work correctly. The least modification, by far, is the Java serializer. The Java serializer only requires the inheritance of the `Serializable` superclass on any class to serialize, as well as all case classes are serializable automatically. Because all of the messages are already case classes or case objects, this means that in order to use the Java serializer, no modification of any kind is necessary. By comparison, both Kryo and Jackson require significant code modification to work.

Correctly setting up Jackson begins by binding Jackson to an interface to be serialized. This is done in the Akka configuration, and this interface is a trait in Scala, containing no data at all. This trait is then inherited by all messages to be serialized, and if all data in messages was in standard types, that would be sufficient. However, Jackson has issues with polymorphism, and in order to correctly serialize any polymorphic type, JSON tags must be put into the code to specify all possible subtypes of that polymorphic type. This ended up being a significant hurdle, as the SwiftVis2 `Geometry` supertype has dozens of individual implementations, and editing SwiftVis2 to add the JSON tags necessary for this project was not a reasonable option. As such, the idea of container classes was used. For each type of `Geometry` that could be sent, an equivalent container class was created that contained all the information of the `Geometry`, with a companion object to allow for the ability to transfer the data between the container class and the real type conveniently. These container classes were then given the necessary JSON tags, and were able to correctly allow Jackson serialization.

Kryo serialization, unlike Jackson, recommends manually binding each individual class to be serialized in the configuration, rather than use an interface. As such, the classpath of

each message was added to the configuration, but we soon discovered that Kryo is unable to correctly serialize lambda functions. Lambda functions are used heavily in the original `Geometry` primitives, so the container classes were then required to remove all lambda functions from these `Geometry`, as well as the creation of the `GeometryCreator` class. This class encompasses the function we had been sending, but with it as an apply method rather than a lambda function, Kryo is able to serialize it without any problems. With these modifications made, all three serializers worked and could be changed between with no code modification. Ray-tracing was done with all three, using the 40 cell maximum of the Pandora cluster, and the results are tabulated in Table 4.3.

Times [sec]	**Java**	**Jackson**	**Kryo**
All	911.6 ± 103.1	304.6 ± 15.2	369.6 ± 0.4
Some	403.2 ± 45.2	243.7 ± 3.9	217.4 ± 1.3
Few	325.0 ± 23.2	218.1 ± 20.5	191.1 ± 1.3

Table 4.3: Benchmarking runs done with the Pandora cluster comparing between serializers.

Benchmarks comparing the serializers are going to favor differences seen between message-heavy methodologies, such as the All schema, more than they favor the methods that send fewer messages, such as Few. Despite this, there are noticeable increases in speed, no matter the schema, between the Java serializer and the other two. Despite its low-modification ease-of-use, the Java serializer is about 1.5x slower in the less message-heavy schemas, and nearly triple the compute time in the All schema. This seems to support the conventional wisdom, as either alternative sees significant speed benefits. Between Jackson and Kryo, the compute times are fairly comparable, with Kryo being slightly slower in some instances and slightly faster in others, but what is consistently clear is that Kryo is significantly more consistent than Jackson or Java. With its consistency, these times vary less and we see very narrow standard deviations. Based on this, Kryo seems like the best choice, as it is

competitive with Jackson, yet is much less prone to outlier results.

4.3 Photometry

The final major expansion of this project was to add the ability for photometric renderings with as little modification to the GeometryManagement code as necessary. The final implementation is depicted in Figure 4.2.

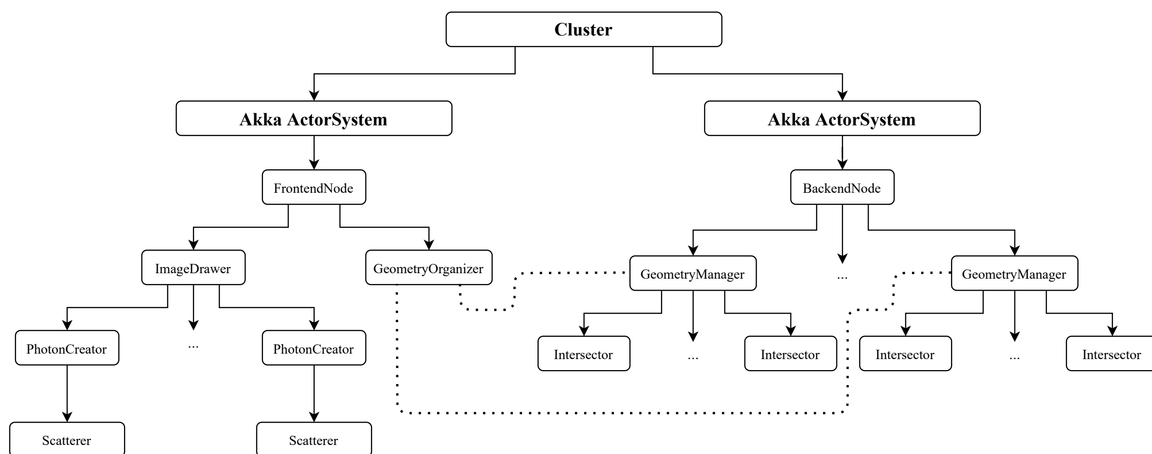

Figure 4.2: The clustered photometric rendering actor hierarchy at Section 4.3.

The `ImageDrawer` is kept from the ray-tracer frontend, and similar to its original function, it handles all modification to the image. Instead of `PixelHandlers`, however, it creates PhotonCreators based on the number of light sources and threads. These PhotonCreators generate a number of photons in the form of rays, each beginning at the light source and randomly towards the geometry. It then waits for a response back from the `GeometryOrganizer`, and if it receives no hit, does nothing. If there is a hit, however, it creates a new subactor, called the Scatterer, which does a calculation to determine which pixel on the screen that intersect location is, as well as how far, and then sends the color

to be assigned to that pixel back up the hierarchy. Unlike with the ray-tracer, however, this color is added to the colors already there rather than assigned in isolation, and then normalized based on the number of changed pixels. This methodology is heavily adapted from the SwiftVis2 photometry code, written by two other researchers under Dr. Lewis[17]. Specifically, this was adapted due to its use of SwiftVis2 geometry primitives, just as this project has. In total, the only modification to any of the GeometryManagement code was to allow the `GeometryOrganizer` to return back the total bounds of the loaded geometry, which it can create trivially by combining the bounds from all its `GeometryManagers`. While this was not directly necessary, it allows the `PhotonCreators` to only send photons into the bounds of the geometry, thus limiting useless photons somewhat. As such, this GeometryManagement data distribution setup is more generally useful as a ray-casting engine, rather than specific to ray-tracing. A 10-cell photometric rendering is depicted in Figure 4.3.

While a local actor-based implementation of the photometric rendering code was made as a proof of concept, the clustered photometry was run across the Pandora cluster, first to compare the three schemas. The results of this work are tabulated in Table 4.4. They use 6.4 million photons as a standard, as well and 40 simulation cells. All runs were done using the Kryo serializer, both for its relative speed and consistency.

Times [sec]	**6.4 million photons, 40 cells**
All	492.205 ± 2.492
Some	84.844 ± 4.688
Few	154.929 ± 9.139

Table 4.4: Benchmarking runs done with the Pandora cluster comparing between schemas for photometry.

These results differ from the ray-tracer comparisons, as the Some implementation significantly outpaced the Few implementation. To some extent, this can be explained, as unlike

Figure 4.3: A photometric rendering of 10 cells, generated using the Some schema and 25,000,000 photons. This image originally appeared in [7].

in the ray-tracer, where a top-down view is used and thus, most rays enter only a single geometry cell, the point light is to the left side and sending rays into geometry from that angle, thus the majority of rays enter the bounds of several cells. Based on these results, as well as the results from [18] showing that the angle at which you view the geometry can heavily change the results between the schemas, seem to indicate that the advantages of the Few schema are more dependent on ray-casting orientation than the others, which should be continued to be investigated with more testing.

The other major scaling factor that can be tested for photometric renderings is the number of photons originally sent out. The larger the number of photons sent, the higher resolution and definition the final image becomes, thus an optimal image has an incredibly high number of photons, generally in the billions of photons. For speed and standardization, testing was done using the Some schema, the Kryo serializer, and incremented the number of initial photons by a factor of ten until it began to fail. The results are tabulated in Table 4.5.

Times [sec]	**Some, Kryo, 40 cells**
128,000	3.280 ± 0.230
1.28 million	28.017 ± 0.470
12.8 million	329.774 ± 32.540

Table 4.5: Benchmarking runs done with the Pandora cluster comparing between initial numbers of photons for photometry.

The scaling factor, interestingly, is very close to exactly linear. for each tenfold increase in particles, there is a near-tenfold increase in compute time. This fits into the expectation, as the main determinant for the scaling of a photometric rendering is the number of photons, but disappointingly, this cluster begins to fail at the next step up, though this seems to be primarily due to the number of simultaneous messages sent on startup, rather than a

genuine limit of the system. This shortcoming is reflected in the increased variance in the 12.8 million runs. A potential solution for this drawback would be repeated smaller runs, using the modified image as a starting point, though the system does not yet support this behaviour.

Chapter 5

Conclusion and Future Work

This project, worked on over the course of two years, is both a success and a failure.

In terms of the written goals, i.e., the use of Akka Cluster to do a distributed-data ray-trace, the project was a success. Beginning by creating a local version using the `castRay` function from SwiftVis2, followed by the creation of the ImageDrawer and GeometryManagement trees, the addition of the `GeometryOrganizer`, and finally the creation of the `FrontendNode` and `BackendNodes`, the system follows the ray-tracer and continues to improve its capabilities with each commit to GitHub. The local ray-tracer worked remarkably well, in particular, with relatively good compute times and maximum number of cells loaded, as well as remarkable scaling relative to the number of cells of geometry, scaling well below linearly. The actor model of Akka made the project remarkably interesting and varied, as the asynchronous nature makes problem-solving different. As well, the networked nature of such a project brought many interesting problems, such as the serializers and creation of the container classes and `GeometryCreator`.

Where the success of this as a project begins to break down is in the results. The main purpose behind this project was to potentially use the cluster work for data distribution

and to overcome the single-machine memory bottlenecks, and to that goal, it failed. Due to the memory overhead of Akka, especially the massive outbox memory overhead necessary due to this method, as well as the serialization of messages taking large amounts of memory, the final stable cluster was unable to do ray-traces larger than the single-machine implementation. Despite that, the design of a large, distributed cluster of machines running Akka and effectively managing huge ray-traces is laid out in the results. The main factor preventing the creation of such a large, stable cluster is the inability to create a Fat JAR to run, and instead having to rely on the reliability of SBT, a tool never designed for parallel access of a single directory, especially not by the number of machines we wished for it to.

The results seem to indicate that there are several ways to move forward, both in schema and in serializer. For the schema, the Some methodology, as well as the Few, provided similar results, with the Few edging out in the ray-tracer's speed, and the Some edging out in the photometry's speed. Based on this, significantly more tests are necessary, with varying light sources and view angles, in order to see the consistency of those results. Despite that, the All methodology's significantly higher scaling factor makes it relatively infeasible for our purposes, but in small-scale ray-traces could prove useful. The serializers, on the other hand, see significant benefits from alternatives, such as Jackson and Kryo, over the Java serializer, but due to the modifications of types necessary to get them running, may prove difficult for uses that pull in code or types from libraries that are not built with Jackson or Kryo themselves. Despite this, the container classes provided enough usability that they are worth using for their performance benefits.

There is a large amount of future work to be done on this project. The most obvious is continued testing, especially over a heterogeneous cluster and with differing views. As well, if the problems with the Fat JAR are solved, comparing the results of a small, high performance cluster with those of a larger cluster of lower performance machines would be

interesting. The largest refactor and potentially worthwhile change would be the implementation of packeting messages. As this system sends each message out individually, the reserved size of the outbox queue for each machine has to be massive, tens of thousands of times larger than the default implementation, and in doing so it significantly impacts the memory usage of the program. If, instead, messages were batched or packeted, in sets of 100 or 1000, the size of the outbox would not be so necessarily large, allowing for this parameter to be scaled down. In addition, the other large drawback of this system is the lack of resiliency. When geometry is lost, or a machine in the cluster fails, there is no load-balancing or reloading of that geometry done, and instead the visualisation is left with an erroneous result. The last interesting possible extension is, based on the `KDTreeGeometry` type in SwiftVis2, which has the capability duplicate itself by duplicating only the reference to the root node, thus allowing the duplication of geometry in code without duplicating it in memory, thus allowing more to be drawn with less memory reserved.

Bibliography

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[2] James Bigler, Abe Stephens, and Steven G Parker. Design for parallel interactive ray tracing systems. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 187–196. IEEE, 2006.

[3] Carson Brownlee, John Patchett, Li-Ta Lo, David DeMarle, Christopher Mitchell, James Ahrens, and Charles Hansen. A study of ray tracing large-scale scientific data in parallel visualization applications. In *Proceedings of the Eurographics Workshop on Parallel Graphics and Visualization, EGPGV*, volume 12, pages 51–60, 2012.

[4] David E DeMarle, Steven Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed interactive ray tracing for large volume visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003.*, pages 87–94. IEEE, 2003.

[5] FasterXML. Jackson. `https://github.com/FasterXML/jackson`.

[6] The Apache Software Foundation. Apache spark. `https://spark.apache.org/`.

[7] Kurt D. Hardee and Mark C. Lewis. Clustered visualization of large scenes using actors. In *27th International Conference on Parallel & Distributed Processing Techniques & Applications*, 2021.

[8] Alan Heirich and James Arvo. Parallel rendering with an actor model. In *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, pages 115–125. Citeseer, 1997.

[9] Thiago Ize, Carson Brownlee, and Charles D Hansen. Real-time ray tracer for visualizing massive models on a cluster. In *EGPGV*, pages 61–69, 2011.

[10] Matthew Larsen, Jeremy S. Meredith, Paul A. Navrátil, and Hank Childs. Ray tracing within a data parallel framework. In *2015 IEEE Pacific Visualization Symposium (PacificVis)*, pages 279–286. IEEE, 2015.

[11] Dr. Mark C. Lewis. Swiftvis2. https://github.com/MarkCLewis/SwiftVis2.

[12] Kwan-Liu Ma and Steven Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Computer Graphics and Applications*, 21(4):72–83, 2001.

[13] Ali Meligy. Parallel and distributed visualization: The state of the art. In *2008 Fifth International Conference on Computer Graphics, Imaging and Visualisation*, pages 329–336. IEEE, 2008.

[14] Paul A Navrátil. Dynamic scheduling for large-scale distributed-memory ray tracing. 2012.

[15] American Museum of Natural History. Planetarium show: Worlds beyond earth — amnh. https://www.amnh.org/exhibitions/permanent/hayden-planetarium/worlds-beyond-earth.

[16] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly parallel search. In Christian Schulte, editor, *Principles and Practice of Constraint Programming*, pages 596–610, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Elizabeth Ruetschle, Lance Ellis, and Mark C. Lewis. Photometric rendering of dust and freed regolith in ringsimulations. In *19th International Conference on Scientific Computing*, 2021.

[18] Elizabeth M. Ruetschle, Kurt D. Hardee, and Mark C. Lewis. Distributed ray tracing of large scenes using actors. In *2020 International Conference on Computational Science and Computational Intelligence*, volume 7. IEEE CPS (+ IEEE Xplore, Scopus, ...), 2020.

[19] Esoteric Software. Kryo. `https://github.com/EsotericSoftware/kryo`.

[20] LightBend Staff. Akka. `https://akka.io`.