R2E2: Low-Latency Path Tracing of Terabyte-Scale Scenes using Thousands of Cloud CPUs

SADJAD FOULADI, Microsoft Research, USA and Stanford University, USA
BRENNAN SHACKLETT, FAIT POMS, ARJUN ARORA, ALEX OZDEMIR, DEEPTI RAGHAVAN, PAT HANRAHAN, KAYVON FATAHALIAN, and KEITH WINSTEIN, Stanford University, USA

In this paper we explore the viability of path tracing massive scenes using a "supercomputer" constructed on-the-fly from thousands of small, serverless cloud computing nodes. We present R2E2 (Really Elastic Ray Engine) a scene decomposition-based parallel renderer that rapidly acquires thousands of cloud CPU cores, loads scene geometry from a pre-built scene BVH into the aggregate memory of these nodes in parallel, and performs full path traced global illumination using an inter-node messaging service designed for communicating ray data. To balance ray tracing work across many nodes, R2E2 adopts a service-oriented design that statically replicates geometry and texture data from frequently traversed scene regions onto multiple nodes based on estimates of load, and dynamically assigns ray tracing work to lightly loaded nodes holding the required data. We port pbrt's ray-scene intersection components to the R2E2 architecture, and demonstrate that scenes with up to a terabyte of geometry and texture data (where as little as 1/250th of the scene can fit on any one node) can be path traced at 4K resolution, in tens of seconds using thousands of tiny serverless nodes on the AWS Lambda platform.

CCS Concepts: • **Computing methodologies → Ray tracing**.

Additional Key Words and Phrases: ray tracing, lambda computing

ACM Reference Format:
Sadjad Fouladi, Brennan Shacklett, Fait Poms, Arjun Arora, Alex Ozdemir, Deepti Raghavan, Pat Hanrahan, Kayvon Fatahalian, and Keith Winstein. 2022. R2E2: Low-Latency Path Tracing of Terabyte-Scale Scenes using Thousands of Cloud CPUs. *ACM Trans. Graph.* 41, 4, Article 76 (July 2022), 12 pages. https://doi.org/10.1145/3528223.3530171

1 INTRODUCTION

Cloud computing platforms provide users the ability to access thousands of CPUs, featuring terabytes of aggregate memory and hundreds of gigabytes of I/O bandwidth, on demand. For example, at the time of this submission, running 1,000 virtual CPUs in the Amazon cloud for one minute costs approximately $1.30 USD. (In aggregate these CPUs feature 1.3 TB of RAM and up to 200 Gbit/s of bandwidth to shared storage.) The abundance of on-demand computing resources presents new opportunities to reduce the run time of expensive graphics jobs and increase the productivity of content creation workflows. For example, an artist might benefit from access to a supercomputer's worth of computing resources for a few minutes to quickly generate a preview of a complex scene, or a scientist might want to quickly load and visualize the results of a supercomputing-scale physics simulation. Motivated by these possibilities, we explore how elastic cloud computing resources can accelerate one challenging rendering task: achieving low latency path tracing of high-complexity scenes (e.g., scenes containing terabytes of geometry and texture) that are too large to fit on most individual render farm machines.

Today, high-performance path tracing is typically performed on CPU or GPU servers that feature many cores connected to a large, unified memory. When scenes exceed available memory size, path tracing performance drops significantly due to frequent transfer of scene geometry data from slower levels of the storage hierarchy. As a result, render farm machines are commonly packed with hundreds of gigabytes of memory, and artists painstakingly modify scene content to fit the memory footprint of these nodes.

In contrast, commercial cloud platforms are optimized to efficiently meet rapidly changing demand from a large user base (e.g., all AWS customers) that typically does not require the large, monolithic servers typical of advanced rendering. As a result, cloud platforms are most efficient when allocating their vast pool of resources in smaller denominations. It is possible for a single user to acquire large numbers of cloud computing resources in just a few seconds, provided the cloud provider can fulfill the request using many smaller nodes, each with limited CPU count, DRAM capacity, and networking bandwidth. Therefore, to achieve the goal of *low latency* job completion using cloud resources, a path tracer must be designed for *massive scale out* execution onto many distributed nodes that each hold only a small fraction of the scene at a time.

In this paper we present R2E2 (Really Elastic Ray Engine), the first system architected to perform low latency path tracing of terabyte-scale scenes using serverless computing nodes in the cloud. R2E2 is a scene decomposition-based parallel renderer that is designed to leverage the unique strengths of elastic cloud platforms (availability of many CPUs/memory in aggregate, and massively parallel access to shared storage) and mitigate the cloud's limitations (low per-node memory capacity and high latency inter-node communication). R2E2 rapidly acquires thousands of cloud CPU cores, loads scene geometry (from a pre-built scene BVH) and texture data into the aggregate memory of these nodes in parallel, and performs full path traced global illumination using an inter-node messaging service specifically designed for communicating ray data over commodity interconnect links. To balance ray tracing work across many nodes, R2E2 adopts a service-oriented design that replicates scene geometry and textures from frequently traversed scene regions onto multiple

Authors' addresses: Sadjad Fouladi, sfouladi@microsoft.com, Microsoft Research, USA, Stanford University, USA; Brennan Shacklett, bps@cs.stanford.edu; Fait Poms, fpoms@cs.stanford.edu; Arjun Arora, aarora52@alumni.stanford.edu; Alex Ozdemir, aozdemir@cs.stanford.edu; Deepti Raghavan, deeptir@cs.stanford.edu; Pat Hanrahan, hanrahan@stanford.edu; Kayvon Fatahalian, kayvonf@stanford.edu; Keith Winstein, keithw@cs.stanford.edu, Stanford University, USA.

0730-0301/2022/7-ART76
https://doi.org/10.1145/3528223.3530171

nodes based on estimates of load, and dynamically assigns work to lightly loaded nodes holding the required scene data.

We port the open source pbrt renderer [Pharr et al. 2016] to the R2E2 architecture and demonstrate that terabyte-scale scenes can be path traced at high resolution, in only tens of seconds, using thousands of tiny serverless nodes on the AWS Lambda platform. R2E2 is open-source software. The source code and the artifacts needed to reproduce the experiments are located at `https://r2e2.dev`.

2 RELATED WORK

Efficiently mapping ray tracing computations to large parallel machines is extensively studied in computer graphics. The fundamental challenge, identified by early attempts to design ray tracing algorithms for early multiprocessors and supercomputers [Cleary et al. 1986; Dippé and Swensen 1984; Kobayashi et al. 1988; Nemoto and Omachi 1986; Priol and Bouatouch 1989; Salmon and Goldsmith 1989; Scherson and Caspary 1988] (see survey by [Jansen and Chalmers 1993]) and more recent attempts to distribute ray tracing onto commodity clusters [Kato and Saito 2002; Navrátil et al. 2014; Pharr et al. 1997; Reinhard et al. 1999; Son and Yoon 2017] and modern multi-core CPUs/GPUs [Parker et al. 2010; Wald et al. 2014; Ylitie et al. 2017a], involves addressing the key conflicting goals of (1) making use of work-efficient acceleration structures, (2) ensuring good load balance of ray tracing work onto available processors, and (3) maintaining high locality to reduce stalls due to communication of scene or ray data.

The most common way to parallelize ray tracing is to use a *screen partitioning* (a.k.a. ray partitioning) approach, where all ray tracing work for a given screen region (or a given bundle of rays) is assigned to a processor. This approach is a natural extension of single processor ray tracing that is trivial to load balance, but suffers from the cost of communicating scene data when the scene working set for a screen region cannot fit in fast memories. Techniques such as lazy loading or lazy generation of scene geometry [Christensen et al. 2003; Georgiev et al. 2018] reduce working set, and increasing a machine's DRAM increases the size of scenes that can be stored in memory, however screen-partitioning schemes still perform poorly when renderers must trace incoherent rays through very large scenes.

In contrast *domain partitioning* approaches partition scene geometry into spatially adjacent regions (a cell in an octree, a treelet of a BVH) and localize this scene data with a processor. During rendering, rays must be communicated to this processor when ray traversal computations need access to this scene information. Domain partitioning allows large scenes to remain resident in the aggregate memory of multiple machines, but presents the challenges of ensuring load balance (many scene regions may not receive many rays) and the costs of frequent ray communication.

To realize the benefits of both approaches, researchers have proposed hybrid techniques that selectively adopt screen partitioning or domain partitioning strategies for different parts of the ray tracing workload [Reinhard et al. 1999]. For example, many out-of-core rendering systems (both for single machines and distributed cluster rendering setups) [Budge et al. 2009; Eisenacher et al. 2013; Navrátil et al. 2014; Pantaleoni et al. 2010; Pharr et al. 1997; Reinhard et al. 1999; Son and Yoon 2017], can be viewed as hybrid techniques that group rays by the scene region they require, then dynamically choose whether to trace those rays on a processor with the scene region already loaded into memory (domain partitioning), load the scene region onto the processor holding the rays (ray partitioning), or move both the rays and the required scene region to a new unloaded processor.

In this paper we utilize a domain partitioning approach. Specifically, we demonstrate that when targeting low latency rendering of massive scenes, the ability to rapidly acquire many nodes enables fast (constant time) scene loading via wide parallel I/O and the ability to store a massive scene in the aggregate memory of these nodes. The advantages of avoiding scene paging and the opportunity to boot enough nodes to service highly loaded scene regions mitigates common pitfalls of domain-partitioned rendering.

Scene compression and out-of-core rendering. Scene data compression techniques [Benthin et al. 2018; Mahovsky and Wyvill 2006; Ylitie et al. 2017b] allow larger scenes fit in the memory of a single machine. We aim to enable scalability to very large scenes, and view data compression as a complementary technique to the scale-out distributed rendering architecture proposed in this paper. Large-scene rendering is also the focus of out-of-core rendering methods [Burley et al. 2018; Pantaleoni et al. 2010]. These methods employ locality optimizing techniques to reduce transfer of ray or scene geometry data to and from storage. We employ similar techniques to effectively utilize a large number of cloud nodes.

Supercomputing as a service. The high elasticity and fine-grained billing of compute offerings such as AWS Lambda and Google Cloud Run have made these platforms an interesting choice for applications that seek low-latency execution through massive scale-out. In recent years researchers and practitioners have built many of such systems: ExCamera [Fouladi et al. 2017] and Sprocket [Ao et al. 2018] for low-latency video processing, PyWren [Jonas et al. 2017] for MapReduce-style data analytics, numpywren [Shankar et al. 2018] for linear algebra, gg [Fouladi et al. 2019] for software compilation and testing, and Cirrus [Carreira et al. 2019] for machine learning workflows. Our work explores the suitability of these emerging platforms for achieving low latency path tracing.

3 GOALS AND PRELIMINARIES

3.1 R2E2 Goals

Our goal is to render scenes containing hundreds of gigabytes to terabytes of geometry and texture data in *just a few minutes* (e.g., complete a full scene render in the time to get a cup of coffee). We focus on high-complexity scenes because they are costly to render and do not fit in high-bandwidth local memories of most single-machine CPU/GPU platforms. These scenes arise in settings such as film production, scientific data visualization, and engineering (e.g., high-resolution CAD models, 3D scans of large environments).

We anticipate workflows where a user *periodically* wishes to quickly render a large scene with path traced global illumination. Since it would be inefficient (in terms of machine utilization or monetary cost) to reserve high-end render farm machines for infrequent tasks (if such machines are even available to the user), we focus on the "cold start" scenario where scene data is resident in networked storage and no machines are allocated for the task. Therefore we are interested in reducing the end-to-end time, including time to acquire processing resources and load the scene from persistent storage, of the path tracing computation.

In this paper, we limit our efforts to scaling ray traversal and shading (including texturing) components of rendering. We treat BVH construction and scene partitioning (§5.1) as preprocessing steps that occur prior to R2E2 execution, and leave cloud-scale parallelization of these components to future work.

3.2 Designing for Cloud Platform Characteristics

To achieve low end-to-end render times, R2E2 must rapidly acquire large amounts processing and networking resources at the time of job creation. The result is an underlying distributed system with significantly different characteristics than the high-end servers typically used to render complex scenes. These characteristics substantially influenced the design of R2E2.

Many "small" nodes. Modern cloud platforms are capable of providing rapid access to computing resources, provided resources are requested in small denominations. For example, we implement R2E2 on top of AWS Lambda nodes that feature only 3 vCPUs and 4 GB of RAM. In benchmarks of the AWS Lambda platform, we were able to acquire 2,500 nodes (7,500 total vCPUs) in approximately 10 seconds. In comparison AWS required 125 seconds to provide eight large nodes featuring 128 vCPUs each (only 1,024 total vCPUs). As a result, R2E2 adopts a **domain decomposition based parallel renderer design** that assumes the worker nodes used for rendering can only hold a small fraction of the entire scene.

Large aggregate memory capacity. Although individual worker nodes have few resources, high overall node count results in aggregate system memory that is significantly greater than the size of the rendered scene. For example, the 2,500-node R2E2 configuration features 10 TB of aggregate memory, several times more than our largest test scene. R2E2 leverages the abundance of DRAM by **replicating geometry and texture data** for frequently traversed scene regions onto many nodes. The flexibility to provide more processing resources for frequently traversed scene regions improves workload balance, a common challenge in domain decomposition based parallel rendering.

Large aggregate I/O throughput. Using many nodes also translates into high aggregate I/O bandwidth. Each AWS Lambda worker can retrieve data from Amazon's storage service (S3) at 600 Mbit/sec, which in a 2,500-worker configuration yields 1.5 Tbit/sec of aggregate parallel I/O throughput. R2E2's domain decomposition based design **exploits massively parallel scene loading** to transfer terabyte-scale scenes and materialize them into the aggregate memory of these nodes in seconds.

High communication latency. The ability to acquire large numbers of resources rapidly comes at the cost of reduced locality of resources. Computing nodes may be distributed across a datacenter, yielding inter-node communication latencies that are both large and unpredictable. To achieve efficient ray communication in this setting,

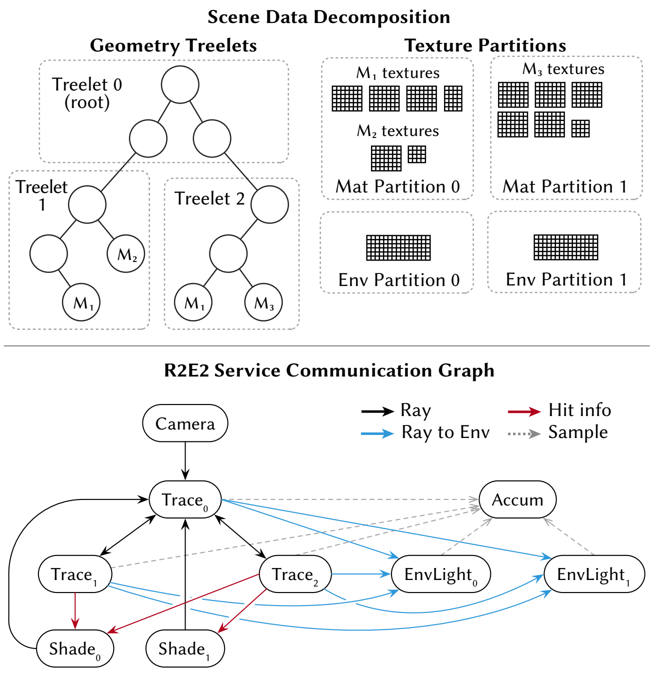

Fig. 1. Top: a BVH partitioned into three treelets. Leaf geometry references material definitions M_{1-3} whose texture data fits into two texture partitions. A large environment map is split between two additional texture partitions. Bottom: The R2E2 service communication graph resulting from this scene decomposition. In addition to camera ray generation (Camera) and render-target sample accumulation (Accum) services, there is one Trace service associated with each treelet and a Shade and EnvLight service associated with each texture partition and environment map partition respectively.

R2E2 is architected to communicate scene geometry at large granularity (100's of MBs) and implements a **customized ray messaging queue service** that batches rays to efficiently utilize communication links.

4 SYSTEM OVERVIEW

In this section, we provide an overview of R2E2's design. R2E2 maps path tracing to massively parallel cloud infrastructure using a domain partitioning approach that divides the scene's BVH and leaf geometry into *treelets* [Aila and Karras 2010] and scene texture data into *texture partitions*. Both treelets and texture partitions are sized to fit within the small memory footprint of worker nodes.

R2E2 is organized as a collection of asynchronous services that communicate via queues. These services generate camera rays (the Camera service), trace rays through a specified treelet t (the Trace_t services), perform BRDF evaluation or environment map sampling using textures in partition p (the Shade_p and EnvLight_p services), and accumulate ray radiance contributions into the frame buffer (the Accum service). Figure 1 depicts a scene partitioned into three treelets and four texture partitions (two for surface material textures and two for the environment map), along with the corresponding R2E2 service graph. There is one Trace service for each treelet and one Shade (or EnvLight) service for each texture partition.

Given scene treelets and texture partitions, R2E2's services cooperate to perform standard fire-and-forget path tracing. The Camera service generates rays, then communicates them to the root treelet's tracing service (Trace_0). These rays may traverse to other treelets (requiring transfer to the corresponding tracing service), or hit geometry in the current treelet. After a hit, the ray is routed to the shading service containing the texture partition relevant to the hit point. Shading generates one bounce ray to continue the path and one shadow ray carrying the shading result of the current bounce. These new rays are communicated to Trace_0 to begin traversal. Ray-scene misses that require environment map sampling are sent to the appropriate EnvLight. Radiance along unoccluded shadow rays and the results of environment map sampling are sent to Accum, which accumulates samples into the render target.

Following the fire-and-forget strategy, R2E2 services are stateless with respect to rays or samples they process. For example, Trace_t services do not internally track a ray's path through the BVH; instead, the tracing services exchange not only ray origin and direction but all other information necessary to finish tracing the ray through the BVH (traversal stack, etc).

4.1 System Operation

To achieve high performance, R2E2 parallelizes its path tracing services using a heterogeneous collection of cloud storage and processing resources provides by the Amazon Web Services (AWS) platform. Figure 2 illustrates the major components of the R2E2 architecture.

Worker nodes. Upon receiving a scene rendering request, R2E2 immediately boots N worker nodes. Our implementation uses serverless functions on the AWS Lambda platform as workers (3 vCPUs per worker) because thousands of these workers can be provided by AWS Lambda in seconds. (We demonstrate up to $N = 2500$ in our experiments.) At the beginning of the ray tracing computation, R2E2 allocates worker nodes to the various Trace_t, Shade_p, and EnvLight_p services according to estimates of the amount of work the service must perform during the rendering job (see §5.3). Services allocated more workers can achieve higher processing throughput to match the expected demand. For a scaled-down example, in Figure 2, rays sent to the Trace_0 service are processed by one of five workers. Services estimated to have little work will be assigned only one worker.

Scene store. R2E2 stores treelet and texture partition data generated during scene pre-processing as read-only blobs in the AWS S3 object store. Once a worker node is assigned to a service, it immediately initiates requests to S3 to transfer the associated treelet (or texture partition) data into its local memory. Loading takes place in parallel across all worker nodes. Since each worker only requires a small amount of data from S3 (~1 GB), the cost of loading scene data into the memory of all workers is *independent of total scene size*, and typically completes in seconds.

Orchestrator. Once workers have loaded their required data, the Camera service begins generating camera rays. R2E2 executes Camera on worker nodes allocated to Trace_0, so generated rays can be processed locally using the already resident root treelet. Path tracing proceeds in parallel, with workers tracing rays for the appropriate Trace_t service and enqueuing partially traced rays or shade requests for other services. The entire computation is managed by a single orchestrator node, which tracks when phases of the computation complete (e.g., scene load) and assigns work residing in service input queues to workers allocated to each service.

Service Queue Store. Once path tracing begins, nearly all communication in R2E2 involves transferring ray (or ray hit) data to the Trace_t and Shade_p services. R2E2 implements ray communication using a set of AWS EC2 machines that manage a distributed input queue for each service. These dedicated queues allow efficient asynchronous transfers and buffering of ray data (§5.4).

Image Tile Store. R2E2 implements the Accum service by accumulating all radiance samples into render target tiles stored in S3. Accum workers hold tiles of the render target in memory, and periodically write partially completed tiles to S3 to enable incremental preview of rendering results.

5 SYSTEM IMPLEMENTATION

Efficiently tracing divergent path-traced global illumination rays through large scenes distributed across many worker nodes required addressing several key challenges.

Locality preserving scene partitioning. Because of the high cost of communication between distributed nodes, to achieve high ray tracing performance, treelet-to-treelet ray transfers during traversal must be infrequent. In section 5.1 we describe how we reduce ray communication events by adapting BVH partitioning algorithms designed to enhance GPU cache locality to the cloud setting.

Ensuring good workload balance. A critical aspect to obtaining high performance from the R2E2 architecture is generating a good allocation of worker nodes to services. A poor allocation will not only leave many worker nodes idle (wasting compute capability), but underprovision performance-critical services (creating a performance bottleneck). In Section 5.3, we describe how we use upfront profiling of the path tracing computation to derive allocation estimates that improve workload balance and performance.

Efficient ray communication. In addition to communicating ray data less often, R2E2 must transmit fine granularity ray data efficiently over commodity networking links. We find off-the-shelf cloud messaging services such as Amazon's Simple Queuing Service [Amazon Web Services 2021] deliver insufficient throughput and too high of latency for this workload, and design a custom messaging service for ray communication (§5.4).

5.1 Treelet Partitioning

To minimize communication during path tracing, R2E2 partitions the scene into disjoint subtrees called *treelets* that maximize ray–geometry locality: i.e., minimize the number of treelets (and therefore the number of transfers between Trace_t services) that a random ray must visit. This goal is analogous to the problem of optimizing BVH partitioning for GPU cache locality, and R2E2 adapts the treelet construction algorithm of [Aila and Karras 2010] for the needs of our distributed path tracing workload. We enhance the partitioning heuristic to account for the communication costs of geometry

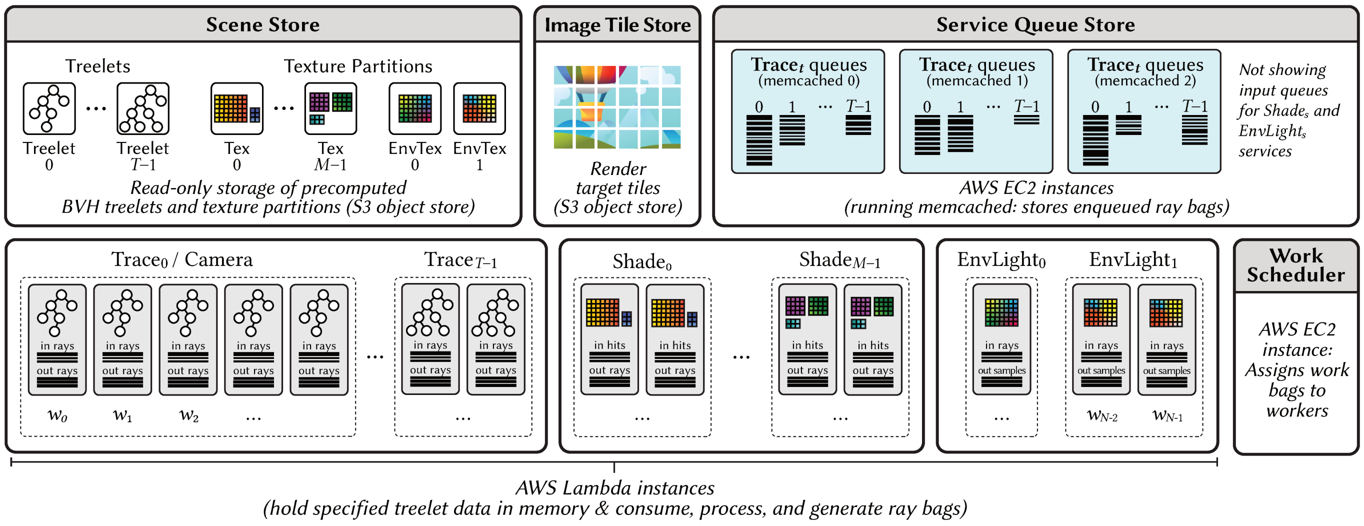
Scene Store
Treelets
Texture Partitions
Treelet 0
Treelet T–1
Tex 0
Tex M–1
EnvTex 0
EnvTex 1
Read-only storage of precomputed BVH treelets and texture partitions (S3 object store)
Image Tile Store
Render target tiles (S3 object store)
Service Queue Store
Trace_t queues (memcached 0)
Trace_t queues (memcached 1)
Trace_t queues (memcached 2)
Not showing input queues for Shade_s and EnvLight_s services
AWS EC2 instances (running memcached: stores enqueued ray bags)
Trace_0 / Camera
Trace_T–1
Shade_0
Shade_M–1
EnvLight_0
EnvLight_1
in rays
out rays
in hits
out samples
w_0 w_1 w_2 … w_N-2 w_N-1
Work Scheduler
AWS EC2 instance: Assigns work bags to workers
AWS Lambda instances (hold specified treelet data in memory & consume, process, and generate ray bags)

Fig. 2. The R2E2 system architecture configured with T scene treelets, M material texture partitions, 2 environment texture partitions, N worker nodes, and 3 memcached servers for the service queue store. R2E2 allocates services worker nodes according to estimates of how much work they will perform during a path tracing computation. Here, the heavily traversed root treelet (Trace$_0$) is assigned five workers.

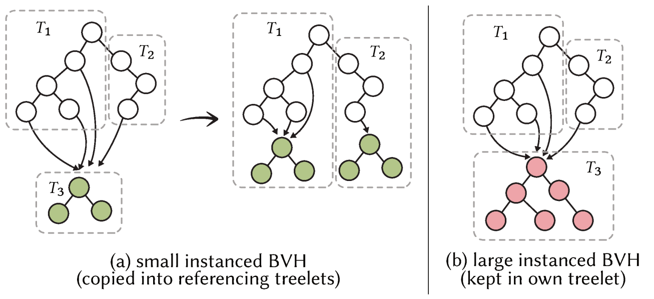
T1
T2
T3
(a) small instanced BVH (copied into referencing treelets)
(b) large instanced BVH (kept in own treelet)

Fig. 3. (a) To avoid inter-service ray transfers R2E2 duplicates instance geometry (grey nodes) into referencing treelets when instances are small and the cost of this duplication is low. (b) Instanced objects that have large amounts of geometry are kept in their own treelets since duplication is costly and the communication cost of ray transfer is amortized over a large amount of traversal work in the instance.

instancing (extensively used in real-world production scenes) and scale-up target treelet size from a few KB (targeting GPU L1 caches) to ~1 GB to fill the memory of cloud workers.

Prior to treelet construction, R2E2 performs a standard acceleration structure build that accounts for instanced geometry: first, R2E2 constructs a distinct BVH for each instance-able object in the scene. Next it constructs a "top-level" BVH for the overall scene that references the object's BVH for each object instance. In a shared-memory renderer, this standard setup reduces footprint by sharing geometry and BVH structures across all instances of an object. However, naively applying the treelet partitioning algorithm of [Aila and Karras 2010] to each of these BVHs (both top level and instance BVHs) will result in substantial inter-treelet communication when scenes contain many instances of low triangle-count objects (e.g., blades of grass).

If instance geometry is always placed in its own treelet, checking a ray against the instance's geometry requires two ray transfers: a first transfer from the worker holding the treelet in the scene's BVH to the worker that holds the treelet containing the instance's geometry, and a reverse transfer so the ray can continue traversal in the scene. This double transfer is repeated *for each instance* of the object, forcing the same ray to visit workers holding the instance's geometry multiple times. For small objects, the cost of these ray transfers outweighs the footprint reduction benefits of sharing geometry across instances. Figure 3(a) provides an example where a region of the BVH contains three instances of an object, so a ray will be transferred from treelet T1 to instanced object's treelet (T3) three times.

To mitigate this issue, R2E2 duplicates the geometry of small objects into any treelet that instances those objects. (In Figure 3(a) T3 is copied into both treelets that reference it.) This increases total scene memory footprint, as a given object may be included in multiple treelets[1]; however, this duplication removes two ray transfers every time the instance appears in the scene. R2E2 can access a large amount of total memory across many cloud workers, so the performance benefit from a reduction in ray transfers outweighs the data loading and memory footprint costs of duplicating instanced geometry across workers. We find that using a heuristic that elects to duplicate objects less than half the maximum treelet size worked well. Figure 3(b) shows an example where an instance featuring large amounts of geometry is not duplicated into referencing treelets.

The greedy BVH partitioning strategy of [Aila and Karras 2010] aims to construct treelets that yield high ray locality: these are treelets that approach the maximum treelet footprint, and have large surface area. However, geometry that cannot be fit in these desirable treelets ends up in a large number of low-surface area, low memory

[1]Note that within a treelet, a small object may still be instanced multiple times; the only duplication of geometry is between treelets, not within them.

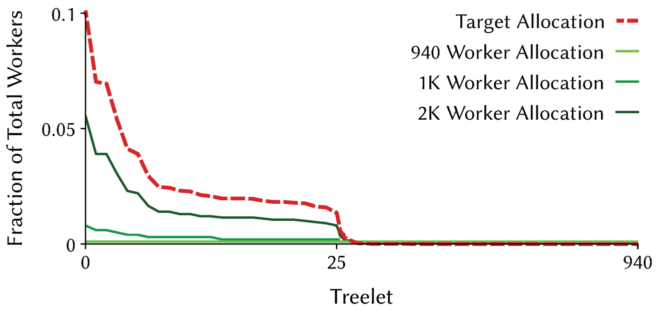

Fig. 4. The fraction of workers R2E2 assigned to each service is proportional to the service's estimated load. Red line: target allocation for Trace$_t$ services, as determined by estimates of load from a low-resolution render of a 940 treelet scene (treelets sorted by most-to-least load). Green lines: actual allocations for R2E2 configurations using increasing numbers of workers. At low worker counts, many lightly used services (right side of graph) are over provisioned as they must have at least one worker to execute. This results in under provisioning of highly used services (left side of graph). Configurations with more workers more closely approximate the target allocation.

footprint treelets. To generate scene partitions that "fill up" worker node memories, R2E2 groups these small disjoint treelets together into new treelets that approach the maximum size. This makes efficient use of available worker node memory and also improves network utilization since grouping small, rarely visited, treelets together increases the number of rays sent to workers holding these parts of the scene. Elsewhere in the paper we use the term "treelets" to refer to chunks of geometry that may include many disjoint parts of the BVH.

5.2 Texture Partitioning

In contrast to geometry partitioning, constructing texture partitions in R2E2 is simple. Our current implementation uses textures in the Ptex texture format [Burley and Lacewell 2008], which divides surface texture data into into small per-face textures. R2E2 forms texture partitions by adding individual Ptex texture images from a single Ptex file (from a single object) into a texture partition until the partition fills. Packing also includes face texture images adjacent to the chosen images. When forming partitions, R2E2 iterates over an object's faces in a BFS order to maintain surface locality of the Ptex face texture images residing in the same R2E2 partition. After inserting all face texture images for an object, R2E2 selects the next scene object and continues to add Ptex face images until the current partition fills.

While our current implementation is based on Ptex, the R2E2 architecture only requires that the texture data assigned to a single service fits in the memory of a worker. Any texture representation capable of partitioning texture data into worker-sized chunks could be used with R2E2 (e.g., UDIM).

5.3 Allocating Workers to Services

R2E2's path tracing performance depends heavily on provisioning services with enough workers so that the service's execution can keep up with the rate of arriving requests. For example, the scene's root treelet service must process orders of magnitude more rays than treelets containing rarely traversed geometry. R2E2 adopts a static worker allocation policy where each service's allocation is determined prior to the start of path tracing and not changed for the duration of the computation. Initial experiments showed that estimating load using heuristics such as treelet surface area poorly approximate actual load, so R2E2 employs an up-front scene profiling step to gather statistics about workload distribution.

R2E2's profiling phase performs a low resolution, low path depth rendering of the full scene using one worker per service (Our implementation renders a 960×540 image with 1 spp and maximum path depth of 5.) Profiling records the total number of rays processed by each service as well as the average computation time for processing requests.

Whether execution is compute-limited or bandwidth-limited varies on a per-service basis (treelets requiring more traversal steps are more likely to be compute bound), so we estimate the time each service s spends doing work during the profiling run (W_s) as the maximum of the average time spent per-request performing computation ($W_{s,\text{comp}}$) and communicating data to the queue store ($W_{s,\text{comm}}$):

$$W_s = \text{num requests processed by } s \times \max(T_{s,\text{comp}}, T_{s,\text{comm}})$$

where $T_{s,\text{comm}}$ is estimated as:

$$T_{s,\text{comm}} = \text{sizeof}(\text{input data} + \text{output data}) \ / \ \text{link bandwidth}$$

R2E2 allocates workers to treelet services proportionally to W_s. However, each service must be allocated at least one worker (otherwise the service could not execute). One result of requiring all parts of the scene to be memory resident is that rarely used services (e.g., rarely traversed treelets) are *over-provisioned*, even if they are allocated only one worker. As scene sizes grow in relation to the total number of worker nodes, a greater fraction of the workers are used to hold rarely visited treelets, preventing critical treelets from receiving sufficient processing resources. R2E2's ability to correctly provision services grows with the number of workers because the additional workers are allocated to heavily used services in need of greater throughput. Figure 4 illustrates this behavior for a scene containing 940 treelets, many of which need less than one worker of processing throughput. As the number of workers is increased, R2E2's allocation (green lines) begins to more closely approximate the target allocation (red line).

Dynamic allocation alternatives. An alternative to static worker allocation is to dynamically adjust a service's worker allocation mid-render-job according to load. The challenge of dynamically allocating workers to services is that it incurs the cost of loading new data (treelets or textures). Reloading a service's required input data not only occupies node bandwidth (preventing ray processing during this time), but there is a substantial latency before the load is complete (~15 sec for the 1 GB treelets). Given our low latency goals, there is little time to adjust treelet allocation during the path tracing computation. Decreasing treelet size to gain more flexibility for reallocation increases ray tracing time since smaller treelets increase the number of high-latency treelet-to-treelet ray transfers. In our experiments we were unable to design a dynamic allocation policy that yielded lower job completion times than the static policy described above. In a latency-prioritized setting, the benefits of using large treelets to reduce inter-node transfers, and "pre-loading" scene data efficiently en masse at the start of computation outweigh the costs of imperfect worker allocation.

5.4 Inter-Service Message Queues

R2E2's communication mechanisms must realize high link utilization when the worker is heavily loaded (since most Trace_t workers are bandwidth bound). Importantly, inter-service communication should have low latency, since the completion time of long ray paths (and therefore the completion time of rendering) is determined by the product of communication latency and the number of inter-service transfers on a ray path. For simplicity, in this section we describe the implementation of communication of ray data between Trace_t services. R2E2 uses the same queuing system communicate hits and samples to the shading and environment light sampling services.

As illustrated in Figure 2 R2E2 implements a queuing service (Service Queue Store) that is based on communicating ray data[2] from workers to and from a small number of proxy servers that buffer enqueued ray state until it can be processed by a worker. This centralized solution avoids the complexity of point-to-point communication between thousands of workers while still maintaining higher throughput and lower latency communication than general-purpose queuing services offered by cloud providers [Amazon Web Services 2021].

The state of a single ray (~1 KB or less) is too small for efficient transmission between cloud nodes. Therefore, to achieve efficient data transfer, workers communicate with the service queue store at the granularity of large batches of rays, which we call *ray bags*. After a Trace_t worker traces a ray through its local treelet, it sorts the ray into local buffers based on the required destination service. When the worker accumulates a large batch of rays for the same target Trace_t service, it compresses the bag then sends it via a single message to the ray store. Empirical tests show that 1 MB transfers are needed to sustain maximum communication throughput between nodes, so our implementation sends ray bags when their pre-compression size reaches 1 MB. We observe that for the scenes used for evaluation in Section 6, ray bag compression reduces bandwidth requirements by a factor of 2.4. Bag assembly, compression, and transmission occur asynchronously with ray tracing (one thread on the worker is used for communication tasks).

Workers also receive rays to process from the service queue store at the granularity of bags. The R2E2 orchestrator tracks the number of bags enqueued for each service, as well as the number of bags currently assigned to that service's workers. As rendering proceeds, the orchestrator assigns ray bags enqueued for a service to the least loaded worker allocated to that service.

The service queue store is implemented by a collection of nodes running the `memcached` in-memory key-value store [Fitzpatrick 2004] (rays bags are objects inserted and removed from `memcached`). These nodes are configured with higher memory capacity (5.25 GB) than workers to provide adequate storage for in-memory ray buffering. We find that a ratio of one ray store server for every 10 R2E2 worker nodes is sufficient to maintain high communication bandwidth across the system.

[2]R2E2 service input queues may store ray state, hit state, or sample information depending on the service involved, but we refer only to ray state to simplify exposition.

Table 1. Scene statistics for Moana-XL and Terrace scenes. Both scenes are sized to require approximately 1 TB of memory.

	Moana-XL	Terrace
Triangles	2,465,363,112	2,455,042,403
BVH Nodes	4,238,464,161	2,963,036,146
Size on disk	177 GB (geometry) 178 GB (texture)	99 GB (geometry) 138 GB (texture)
Size in memory	1.1 TB	0.9 TB
Lights	353	1
Instancing	Yes	No
Treelet count	940	684
Avg. treelet size	0.4 GB ± 0.2	0.3 GB ± 0.2

6 EVALUATION

We evaluate R2E2 in terms of its wall-clock performance (we seek low latency job completion) and its scalability to large scenes and many worker nodes. We demonstrate that R2E2 is capable of rendering terabyte-scale scenes significantly faster (up to 8.6×) than a high-end shared memory server (§6.2). We also show that R2E2 scales to increasingly large scenes simply by adding more workers to the system (§6.3, §6.4).

6.1 Experimental Setup

Scenes. We created scenes with increasing complexity by duplicating the island geometry and textures from the Moana Island Scene [Walt Disney Animation Studios 2018]. We created scenes with one (Moana-XS), two (Moana-S), four (Moana-M), six (Moana-L), and eight (Moana-XL) copies of the island geometry positioned on a 4×2 grid. Each duplicated copy of the island has its own unique set of instances and associated geometry so that within each island, instancing is still present. Moana-XL's geometry and texture data occupies approximately 1 TB when materialized in memory. Moana rendering uses the scene's original BXDFs and textures (including the environment light), but does not include volumetric effects.

We also evaluate on Terrace, a high-resolution terrain populated with detailed objects. Terrace is also sized to require approximately 1 TB of memory. Both scenes are divided into treelets and texture partitions that each occupy at most ~2 GB of memory, less than the total DRAM available on each worker node to ensure adequate space is available for ray-state storage and other runtime state. Table 1 provides statistics for both scenes.

In our experiments all images are rendered at 4K resolution (3840×2160) and with a maximum path depth of five. To account for view dependence in results, we render the Moana and Terrace scenes from four and two viewpoints respectively (Figure 5) and report average performance from these timings. A full breakdown of these results by the viewpoint can be found in the supplementary materials.

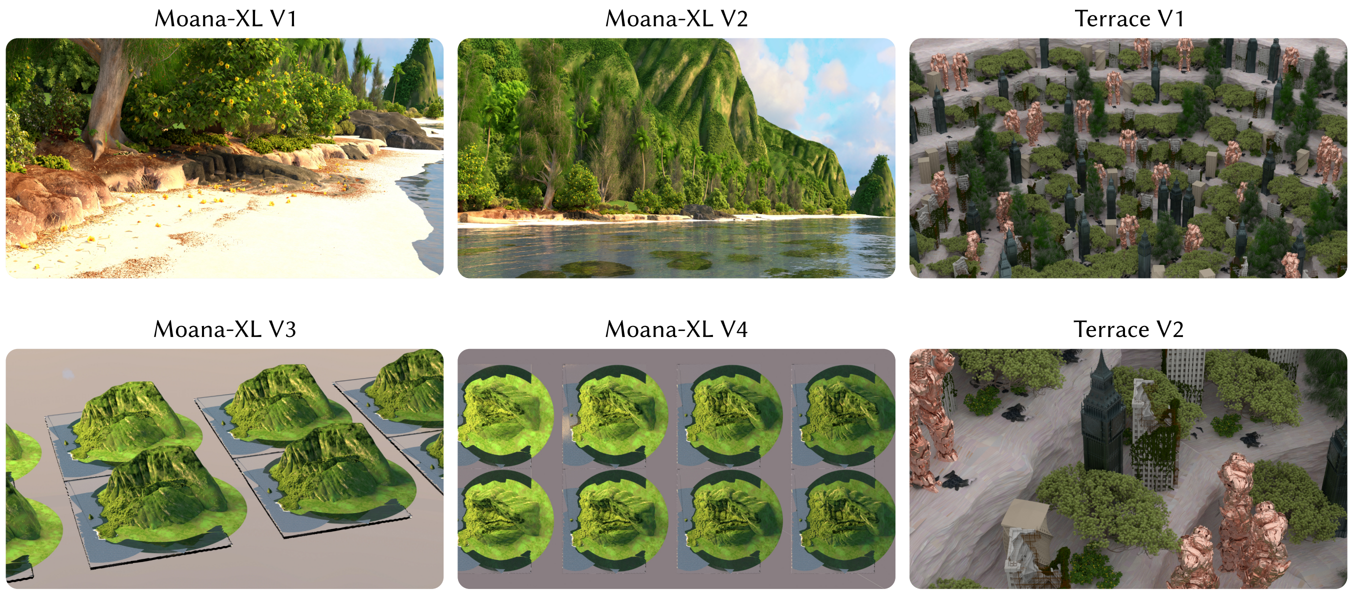

Fig. 5. Example viewpoints used to render Moana and Terrace scenes in our experiments.

`R2E2` *configuration.* We evaluate `R2E2` running on AWS Lambda workers equipped with three virtual CPUs (vCPUs), 4 GB of RAM, and up to 600 Mbit/s networking [Fouladi et al. 2019; Wawrzoniak et al. 2021]. We configure `R2E2` to use one `memcached` node (for ray queueing) for every 10 workers. These nodes are Amazon Elastic Compute Cloud (EC2) `c5n.large` instances with 2 vCPUs, 5.25 GB of RAM, and up to 25 Gbit/s networking.

A fleet of 250 `memcached` servers can be started in 30 sec on Amazon EC2. We did not include this boot time in our measurements, because the servers could be booted in parallel with the execution of the upfront profiling (our current implementation does not feature this optimization). However, the cost of running the fleet during the rendering job is included in all `R2E2` cost estimates.

pbrt-treelet baseline. We benchmark `R2E2`'s performance against a version of `pbrt` that requires the entire scene to be materialized in memory during ray tracing, but is modified to use the same treelet structures as those used by `R2E2` (`pbrt-treelet`). This modification reduces `pbrt`'s traversal performance, but was necessary to allow `pbrt` to handle terabyte-sized BVHs. We run `pbrt-treelet` on an EC2 `x1.32xlarge` instance with 128 virtual CPUs, 1.9 TB of RAM and a 25 Gbit/s network link. We view this configuration as representative of a large server in a production render farm. Typical time to provision and boot the machine from EC2 is ~70 seconds; however, we do not include this launch time in our results, yielding a best-case execution for this baseline. `pbrt-treelet` stores scene geometry on remote storage (S3), and transfers it over the network, and materializes it in memory before path tracing starts. We parallelize deserialization and object materialization to maximize `pbrt-treelet`'s scene loading performance.

6.2 End-to-End Performance

Figure 6 compares the end-to-end performance of `R2E2` and `pbrt-treelet` renders of the Moana-XL and Terrace scenes. `R2E2` is configured to use 2,500 Lambda workers (a total of 7,500 vCPUs and 10 TB of RAM), accompanied by 250 `memcached` servers for ray communication. In this configuration, `R2E2` is 3.5–7.8× faster than `pbrt-treelet`'s single-machine in-memory path tracer. We average rendering performance over the views shown in Figure 5. The results include the upfront profiling time.

A significant factor in the performance speedup of `R2E2` comes from the speed of loading the scene: time to transfer data over the network, deserialize the data, and (in the case of geometry) construct a traceable BVH in memory. By using the aggregate network throughput and compute of thousands of workers that each load and materialize a small subset of the scene, `R2E2` finishes scene transfer 4.9× (Moana-XL) and 2.9× (Terrace) faster than `pbrt-treelet`. `R2E2` achieves an average aggregate bandwidth usage of ~150 Gbit/s (up to 1.2 Tbit/s peak) during the initial scene load, compared to the 25 Gbit/s available to `pbrt-treelet`'s single machine.

Massive parallelism also results in `R2E2` outperforming `pbrt-treelet` by 5.7–9.8× (Moana) and 5.1–6.8× (Terrace) in just the path tracing portion of the computation. While `R2E2`'s ray throughput is higher in aggregate than `pbrt-treelet`, `R2E2`'s ray throughput per-vCPU is lower than `pbrt-treelet`'s due to the costs of internode communication and workload imbalance across large numbers of workers. Recall that after scene deserialization, `pbrt-treelet` requires no further network I/O as it can access the entire scene in unified shared memory, reducing the performance cost of tracing each ray.

In all rendering configurations except Terrace 64 spp, `R2E2` *completes the entire path tracing job before* `pbrt-treelet` *begins tracing*

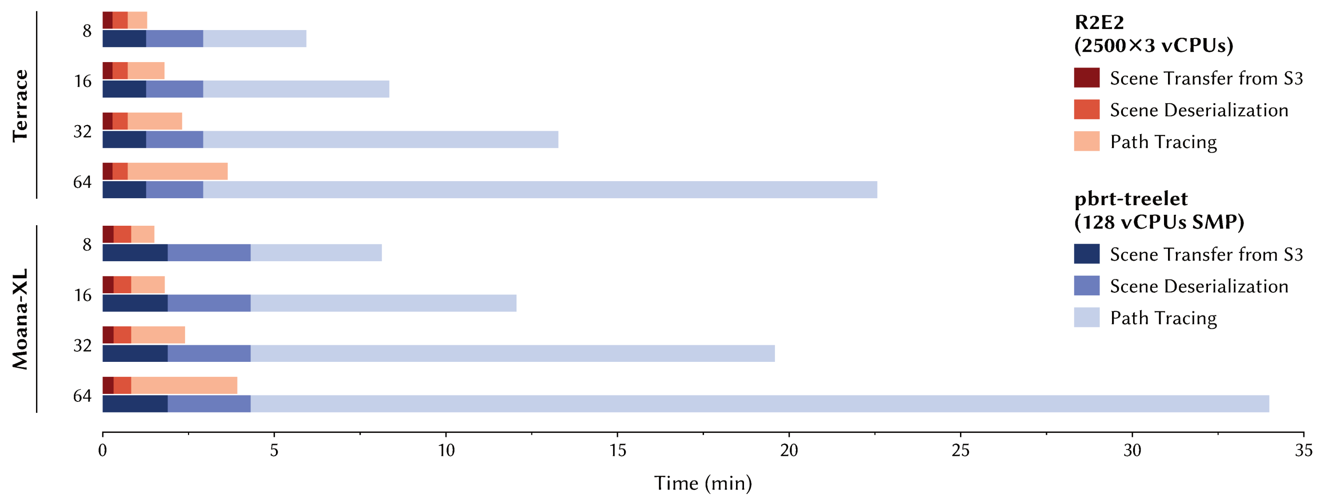

Fig. 6. By parallelizing scene transfer and computation onto 2,500 AWS Lambda workers (7,500 total vCPUs), R2E2 reduces the end-to-end time to load and path trace terabyte-scale Moana-XL and Terrace scenes by 3.8–8.6× compared `pbrt-treelet` running on a high-end 128 vCPU server with 1.9 TB of memory. R2E2's massive parallelism results in faster path tracing performance and also high throughput transfer and deserialization of scene data into the 10 TB aggregate memory of worker nodes. R2E2 is able to complete a 32 spp Moana-XL rendering job (including time to acquire and boot worker nodes) in 146 seconds, barely more than the time it takes the single 128 vCPU server to transfer scene data from shared storage. The time for the upfront profiling stage is included in the results. A breakdown of the results by the viewpoints can be found in the supplementary materials.

Table 2. R2E2 reduces job completion time compared to `pbrt-treelet`, but uses 2,500 worker nodes (nearly 60× more vCPUs) to do so. However, cost increases are only 4.8–8.1× since `pbrt-treelet` must use more expensive computing resources with access to large shared memories. Results include the upfront profiling stage.

	Moana-XL				Terrace			
SPP	8	16	32	64	8	16	32	64
`pbrt-treelet`	$1.81	$2.68	$4.35	$7.56	$1.32	$1.86	$2.95	$5.02
R2E2	$11.52	$14.82	$20.92	$36.77	$9.77	$14.99	$20.39	$34.32
Cost Ratio	6.4×	5.5×	4.8×	4.9×	7.4×	8.1×	6.9×	6.8×
Speedup	5.3×	6.5×	8.0×	8.6×	3.8×	4.0×	5.1×	5.8×

a single ray (`pbrt-treelet` is still loading and deserializing scene data). This implies that an alternative strategy of accelerating `pbrt-treelet` by parallelizing it across multiple large 128-vCPU machines using image-decomposition rendering would not outperform R2E2 on these jobs, regardless of the number of machines used. (Recall that in an image decomposition renderer, all machines must load the scene.) Also, it can be slower to acquire large numbers of terabyte-memory-scale SMT machines as they are less readily available from cloud providers.

6.2.1 Monetary Cost. Beyond the raw speedup achieved by R2E2, another important aspect to the practicality of the system is the cost effectiveness of leveraging 7,500 vCPUs through AWS Lambda compared to `pbrt-treelet`'s 128-vCPU machine. Despite using nearly 60× more compute resources during the path tracing part of the computation, R2E2 incurs only a 4.8–8.1× increase in monetary cost (Table 2). This is because the Lambda vCPUs run for shorter time (the R2E2 computation finishes faster than that of `pbrt-treelet` and only a small number of Lambda workers run during the initial profiling phase) and cost 32% less than vCPUs on the EC2 instance. While on-demand Lambda workers are generally rented at a price premium due to their fast startup times, the vCPUs of the EC2 instance are even more expensive due to the large 2 TB unified memory system on the machine. Further efficiency optimizations, such as sizing AWS Lambda workers on a per service basis (using the smallest worker type that can fit a service's input data in memory), or using bulk-launch APIs to rapidly launch small memory footprint traditional VMs (as opposed to higher priced Lambda workers) could notably reduce the cost of rendering using R2E2.

6.3 Scalability with Scene Size

A major goal of R2E2 is to render scenes with arbitrary complexity, so we study the scalability of R2E2 across varying scene sizes. Specifically, using the same 2,500-worker configuration from Section 6.2, we evaluate R2E2 on Moana-XS (~125 GB in-memory), Moana-S (~250 GB), Moana-M (~500 GB), Moana-L (~750 GB) and Moana-XL (~1,000 GB). All scenes are rendered from view V2.

Overall, end-to-end speedup over `pbrt-treelet` increases as scene size increases (Figure 7a), demonstrating the benefits of the R2E2 architecture for larger, terabyte-scale scenes. Speedup also increases with sample count, since the availability of more rendering work facilitates more efficient parallelization (more work per worker yields better workload balance and larger ray bags during transfers).

Figure 7(b,d) clearly demonstrate how scene loading speedup increases with scene size. By design, the time for R2E2 to load scene

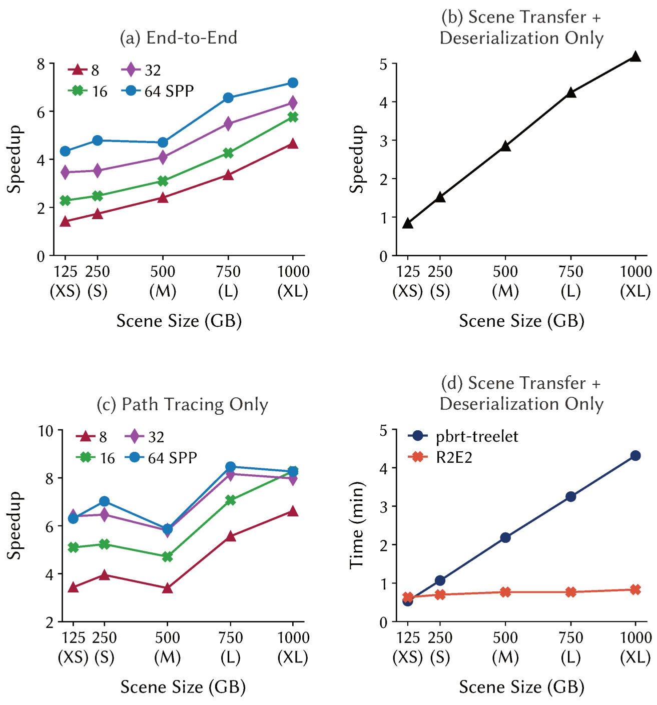

Fig. 7. (a) R2E2's end-to-end speedup over pbrt-treelet increases with scene size (all results are from Moana view V1). (b,d) Unlike pbrt-treelet, R2E2's scene load time is independent of scene size, so loading speedup increases linearly with scene size. (c) R2E2 achieves higher ray tracing performance than pbrt-treelet in all configurations, with speedup increasing at higher sample counts. All R2E2 results use 2,500 workers.

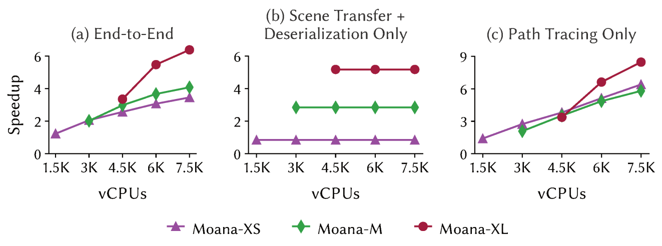

Fig. 8. R2E2 scales to thousands of workers. (Lines for different scenes start at different points on the X axis because scene size dictates the minimum number of workers needed to hold the entire scene.) (a) R2E2 speedup over pbrt-treelet increases with worker count, since the ray tracing portion of the computation is accelerated by additional workers. (b) Additional workers do not reduce scene loading times since each worker loads approximately the same amount of scene data independent of worker count. (c) Additional workers provide R2E2 more flexibility to balance ray tracing work across workers, resulting in superlinear scaling for the larger Moana-XL scene. All results are rendered at 32 spp from view V2.

geometry is independent of the total scene size. Each worker always loads a single treelet or geometry partition. Conversely, pbrt-treelet's single machine must transfer and deserialize an increasingly large amount of geometry as the scene grows. Therefore, as the scene size increases, pbrt-treelet's transfer and deserialization time increases linearly, while R2E2's remains constant. Linear speedup with scene size would continue for even larger scenes than Moana-XL, provided R2E2 utilized more workers to maintain enough aggregate memory to store the entire scene.

6.4 Scalability with Increasing Worker Nodes

Figure 8 shows that the speedup achieved by R2E2 scales well with increasing number of workers, particularly for larger scenes. By design, adding more workers does not reduce the scene load time (Figure 8(b)), because each worker always loads a constant amount of data (one treelet). Therefore increasing speedup is the result of using additional workers to accelerate the path tracing component of the workload (Figure 8(c)). For path tracing, all scenes show nearly linear, or in the case of Moana-XL, super-linear speedup as the number of workers increases. Super linear scaling occurs because as the number of workers grows, R2E2's worker allocation to Trace$_t$ services more closely approximates the target allocation determined by the profiling phase (Figure 4). Additional workers not only add new compute and bandwidth to the system, but can also reduce inefficiencies that arise due to insufficient worker capacity to process heavily loaded Trace$_t$ services.

6.5 Runtime Breakdown

Figure 9 shows a detailed breakdown of R2E2's behavior over a 64 spp render of Moana-XL (view V3). The breakdown shows both network bandwidth consumed per second (blue line) as well as the rate of path completion (paths/sec, red line) over the duration of the computation (in aggregate, across all workers). The computation begins using 940 Lambda workers to complete the profiling render in 30 seconds. Only a small number of rays are traced during this stage, so profiling time is dominated by time to transfer and deserialize the scene. After profiling completes, R2E2 launches 2,500 Lambda workers for the full render, network utilization quickly reaches a peak of over 1.2 Tbit/s as these workers transfer treelet data from S3. Once path tracing begins, network utilization reaches a steady state of 350 Gbit/s for the next 50 seconds while rays are traced. Path tracing reaches a peak rate of 9 million paths completed per second.

7 DISCUSSION

Our work demonstrates that it is possible to construct a "supercomputer on the fly" from many small elastic cloud nodes, and use these resources to reduce end-to-end job latency when path tracing terabyte-scale scenes. Further work should continue to explore how additional components of high-quality rendering systems, such as BVH and treelet construction, can be mapped onto widely available, parallel cloud platforms.

R2E2 is a proof of concept for how a fine-grained communication-heavy, data-locality-sensitive workload can be mapped onto cloud platforms intended for applications featuring many lightweight, independent tasks. While we are aware of ways that further performance engineering could improve R2E2 to realize significantly more efficient use of today's AWS Lambda platform (increasing performance per unit cost), we are most interested in using our experiences to influence the design of future elastic cloud platforms

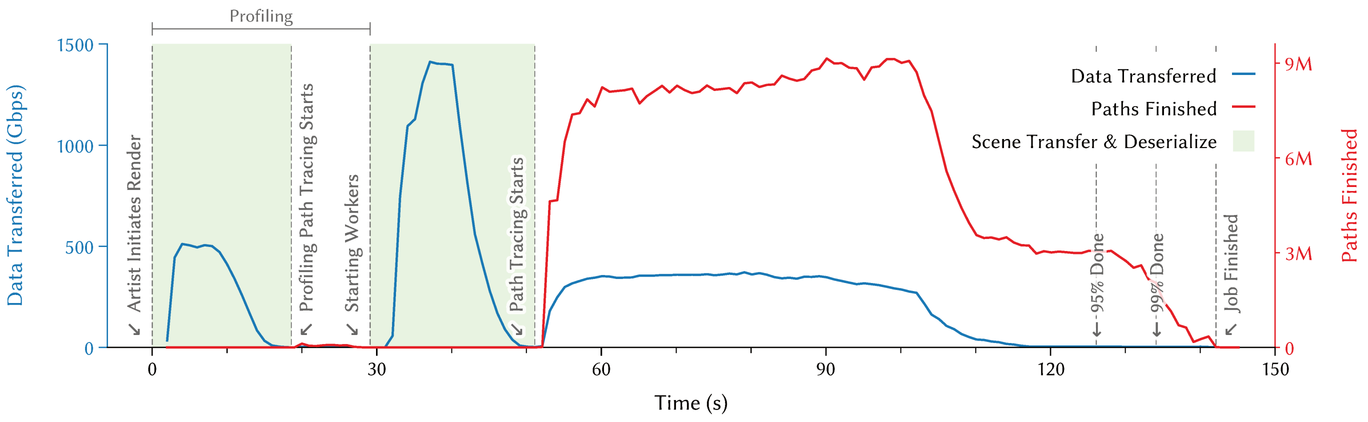

Fig. 9. Detailed view of R2E2 behavior during a 64 spp rendering of Moana-XL (view V3). The total render time, including the initial profiling render, is 142 seconds. The lines show network bandwidth utilized (blue line) and paths completed (red line) in aggregate across all 2,500 Lambda workers. In this computation, R2E2 realizes a peak network bandwidth of nearly 1500 Gbit/s and a peak path completion rate of 9 million paths per second.

to better meet the needs of large-scale graphics computations. For example, the ability to inform the cloud platform of an application's locality needs, or introspect cloud-provided resources to understand the topology of allocated nodes might allow more efficient scheduling of path-tracing-like applications. In general, we view our experiments with AWS Lambda today as performing a similar role as early GPGPU work that made unexpected use of the OpenGL graphics pipeline, and established the requirements for later GPU compute mode execution.

Finally, we hope R2E2 encourages others in the graphics community to consider how building graphics applications for massive scale out execution on elastic platforms might enable new interactive (or low latency) visual computing experiences or new content creation workflows that are simply not possible on the highest end single-machine CPU and GPU platforms today.

ACKNOWLEDGMENTS

We thank the ACM SIGGRAPH reviewers for their helpful comments and suggestions. We are grateful to Matt Pharr, Solomon Boulos, Feng Xie, and Marc Brooker for conversations and feedback. This work was supported in part by NSF grants 2045714, 2039070, 2028733, and 1763256, DARPA contract HR001120C0107, a Sloan Research Fellowship, and by Google, Huawei, VMware, Dropbox, Amazon, and Meta Platforms.

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*. 113–122.

Amazon Web Services. 2021. Simple Queue Service (SQS). https://aws.amazon.com/sqs/.

Lixiang Ao, Liz Izhikevich, Geoffrey M Voelker, and George Porter. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.

Carsten Benthin, Ingo Wald, Sven Woop, and Attila T. Áfra. 2018. Compressed-Leaf Bounding Volume Hierarchies. In *Proceedings of the Conference on High-Performance Graphics* (Vancouver, British Columbia, Canada) *(HPG '18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 4 pages. https://doi.org/10.1145/3231578.3231581

Brian Budge, Tony Bernardin, Jeff A. Stuart, Shubhabrata Sengupta, Kenneth I. Joy, and John D. Owens. 2009. Out-of-core Data Management for Path Tracing on Hybrid Resources. *Computer Graphics Forum* (2009). https://doi.org/10.1111/j.1467-8659.2009.01378.x

Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. 2018. The Design and Evolution of Disney's Hyperion Renderer. *ACM Trans. Graph.* 37, 3, Article 33 (jul 2018), 22 pages. https://doi.org/10.1145/3182159

Brent Burley and Dylan Lacewell. 2008. Ptex: Per-Face Texture Mapping for Production Rendering. In *Proceedings of the Nineteenth Eurographics Conference on Rendering* (Sarajevo, Bosnia and Herzegovina) *(EGSR '08)*. Eurographics Association, Goslar, DEU, 1155–1164. https://doi.org/10.1111/j.1467-8659.2008.01253.x

Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*. 13–24.

Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. 2003. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum* 22, 3 (2003), 543–552. https://doi.org/10.1111/1467-8659.t01-1-00702

J G Cleary, B M Wyvill, G M Birtwistle, and R Vatti. 1986. Multiprocessor Ray Tracing. *Comput. Graph. Forum* 5, 1 (March 1986), 3–12. https://doi.org/10.1111/j.1467-8659.1986.tb00263.x

Mark Dippé and John Swensen. 1984. An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis. *SIGGRAPH Comput. Graph.* 18, 3 (Jan. 1984), 149–158. https://doi.org/10.1145/964965.808592

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. 2013. Sorted Deferred Shading for Production Path Tracing. In *Proceedings of the Eurographics Symposium on Rendering* (Zaragoza, Spain) *(EGSR '13)*. Eurographics Association, Goslar, DEU, 125–132. https://doi.org/10.1111/cgf.12158

Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5.

Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 475–488.

Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 363–376.

Iliyan Georgiev, Thiago Ize, Mike Farnsworth, Ramón Montoya-Vozmediano, Alan King, Brecht Van Lommel, Angel Jimenez, Oscar Anson, Shinji Ogaki, Eric Johnston, Adrien Herubel, Declan Russell, Frédéric Servant, and Marcos Fajardo. 2018. Arnold: A Brute-Force Production Path Tracer. *ACM Trans. Graph.* 37, 3, Article 32 (aug 2018), 12 pages. https://doi.org/10.1145/3182160

F.W Jansen and A.G Chalmers. 1993. Realism in real time?. In *4th Eurographics Workshop on Rendering*. 27 – 46.

Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451.

Toshi Kato and Jun Saito. 2002. "Kilauea" - Parallel Global Illumination Renderer. In *Eurographics Workshop on Parallel Graphics and Visualization*, D. Bartz, X. Pueyo, and E. Reinhard (Eds.). The Eurographics Association. https://doi.org/10.2312/EGPGV/EGPGV02/007-016

Hiroaki Kobayashi, Satoshi Nishimura, Hideyuki Kubota, Tadao Nakamura, and Yoshiharu Shigei. 1988. Load balancing strategies for a parallel ray-tracing system based on constant subdivision. *The Visual Computer* 4 (07 1988), 197–209.

J. Mahovsky and B. Wyvill. 2006. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. *Computer Graphics Forum* (2006). https://doi.org/10.1111/j.1467-8659.2006.00933.x

P. A. Navrátil, H. Childs, D. S. Fussell, and C. Lin. 2014. Exploring the Spectrum of Dynamic Scheduling Algorithms for Scalable Distributed-MemoryRay Tracing. *IEEE Transactions on Visualization and Computer Graphics* 20, 6 (2014), 893–906.

K. Nemoto and T. Omachi. 1986. An Adaptive Subdivision by Sliding Boundary Surfaces for Fast Ray Tracing. In *Proceedings of Graphics Interface and Vision Interface '86* (Vancouver, British Columbia, Canada) *(GI '86)*. Canadian Man-Computer Communications Society, Toronto, Ontario, Canada, 43–48. http://graphicsinterface.org/wp-content/uploads/gi1986-9.pdf

Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. 2010. PantaRay: Fast Ray-Traced Occlusion Caching of Massive Scenes. *ACM Trans. Graph.* 29, 4, Article 37 (July 2010), 10 pages. https://doi.org/10.1145/1778765.1778774

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Trans. Graph.* 29, 4, Article 66 (jul 2010), 13 pages. https://doi.org/10.1145/1778765.1778803

Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation.* Morgan Kaufmann.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 101–108.

Thierry Priol and Kadi Bouatouch. 1989. Static load balancing for a parallel ray tracing on a MIMD hypercube. *The Visual Computer* 5 (1989), 109–119.

Erik Reinhard, Alan Chalmers, and Frederik W. Jansen. 1999. Hybrid Scheduling for Parallel Rendering Using Coherent Ray Tasks. In *Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics* (San Francisco, California, USA) *(PVGS '99)*. IEEE Computer Society, USA, 21–28. https://doi.org/10.1145/328712.319333

J. Salmon and J. Goldsmith. 1989. A Hypercube Ray-Tracer. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications - Volume 2* (Pasadena, California, USA) *(C3P)*. Association for Computing Machinery, New York, NY, USA, 1194–1206. https://doi.org/10.1145/63047.63073

Isaac D. Scherson and Elisha Caspary. 1988. Multiprocessing for ray tracing: a hierarchical self-balancing approach. *The Visual Computer* 4 (1988), 188–196.

Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: Serverless linear algebra. *arXiv preprint arXiv:1810.09679* (2018).

Myungbae Son and Sung-Eui Yoon. 2017. Timeline Scheduling for Out-of-Core Ray Batching. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 11, 10 pages. https://doi.org/10.1145/3105762.3105784

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph.* 33, 4, Article 143 (jul 2014), 8 pages. https://doi.org/10.1145/2601097.2601199

Walt Disney Animation Studios. 2018. Moana Island Scene (v1.1). https://www.disneyanimation.com/resources/moana-island-scene/.

Mike Wawrzoniak, Ingo Müller, Rodrigo Fraga Barcelos Paulus Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *11th Annual Conference on Innovative Data Systems Research (CIDR'21)*.

Henri Ylitie, Tero Karras, and Samuli Laine. 2017a. Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3105762.3105773

Henri Ylitie, Tero Karras, and Samuli Laine. 2017b. Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs. In *Proceedings of High Performance Graphics* (Los Angeles, California) *(HPG '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 13 pages. https://doi.org/10.1145/3105762.3105773