Hybrid Image-/Data-Parallel Rendering Using Island Parallelism

Stefan Zellmann*
Bonn-Rhein-Sieg University of Applied Sciences
University of Cologne

Ingo Wald†
NVIDIA.

Joao Barbosa‡
INESC-TEC and University of Minho

Serkan Dermici§
Department of Computer Engineering
Bilkent University

Alper Sahistan¶
Department of Computer Engineering
Bilkent University

Uğur Güdükbay ‖
Department of Computer Engineering
Bilkent University

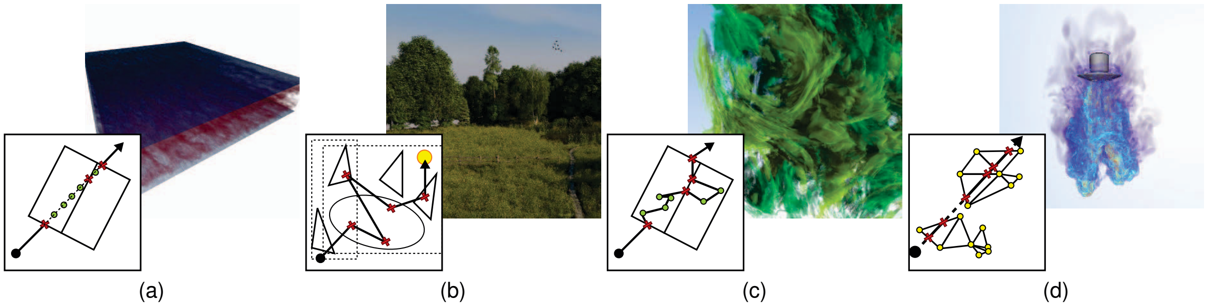


Figure 1: Four very different data-parallel, ray-based renderers. a) Sort-last absorption + emission volume renderer. b) Ray-forwarding production path tracer realized using OptiX and hardware ray tracing. c) Volumetric path tracer optimized for scientific visualization (sci-vis). d) Large unstructured element-marcher with simulation-provided bounding geometry. All four renderers operate on huge data, but have largely different scaling properties; we show how hybrid image-/data-parallel rendering with islands can benefit these renderers to scale far beyond either image or data-parallel rendering applied in isolation.

Abstract

In parallel ray tracing, techniques fall into one of two camps: *image-parallel* techniques aim at increasing frame rate by replicating scene data across nodes and splitting the rendering work across different ranks, and *data-parallel* techniques aim at increasing the size of the model that can be rendered by splitting the model across multiple ranks, but typically cannot scale much in frame rate. We propose and evaluate a *hybrid* approach that combines the advantages of both by splitting a set of $N \times M$ ranks into M *islands* of N ranks each and using data-parallel rendering within each island and image parallelism across islands. We discuss the integration of this concept into four wildly different parallel renderers and evaluate the efficacy of this approach based on multiple different data sets.

1 Introduction

Parallel rendering refers to using multiple CPUs or GPUs to work on the same rendering task simultaneously. It is motivated by either a desire for higher rendering performance or to accommodate extensive data that would not fit into a single device's memory. Generally speaking, this can be done by either replicating the model across every *rank*[1], and having different ranks render different pixels (*image parallelism*, or sort-first rendering); or conversely, by partitioning the image data across different ranks, having each rank render an image of its assigned data, and compositing the results at the end (*data-parallel* rendering, or sort-last rendering).

In some cases, (data-)parallel rendering can simultaneously achieve both of these goals. In the earlier days of graphics (and especially in scientific visualization, where graphics was often done without powerful GPUs), rendering performance was often dominated by how *many* triangles any given node had to render. Splitting a large polygonal data set across multiple nodes would reduce the data per rank *and* rendering time. This idea was somewhat counteracted by the cost of compositing the partial results of different nodes, but for large enough models, it was still worth it. Similarly, for volume rendering, at least for the more straightforward kind of ray *marching*-based volume rendering techniques, the render cost is often somewhat related to the number of voxels, AMR blocks, or unstructured elements that a given node contains, so distributing the data set across multiple nodes would simultaneously reduce memory pressure and render cost.

The link between model size and render cost has weakened with ever more advanced rendering techniques. For example, when using acceleration techniques such as empty space skipping or macro cells, volume rendering is sublinear in the number of voxels; and with modern GPUs and methods such as display lists, geometry shaders, hierarchical depth culling, deferred shading, the triangle count of a model has significantly less impact on render performance than other factors.

Today, more and more rendering is achieved via ray tracing and path tracing, and with that, this link between model size and rendering performance has weakened even further. Thanks to acceleration structures like bounding volume hierarchies (BVH)—on both CPUs and GPUs—the cost for tracing a ray is only about logarithmic in model size, and render performance is almost entirely dominated by how *many* rays a renderer requires to trace. This is obviously the case for surface rendering, but at least when adopting advanced volume rendering techniques like Woodcock tracking is similar for volumes, too.

For parallel rendering, this means that the "simultaneous" ben-

*e-mail: zellmann@uni-koeln.de
†e-mail: iwald@nvidia.com
‡e-mail: jbarbosa@macc.fccn.pt
§e-mail: serkan.demirci@bilkent.edu.tr
¶e-mail: alpersahistan@gmail.com
‖e-mail: gudukbay@cs.bilkent.edu.tr

[1] Throughout this paper, we adopt the MPI terminology of having multiple *ranks*, where a rank can be a process, a node, or a GPU, depending on context.

2022 IEEE 12th Symposium on Large Data Analysis and Visualization (LDAV) | 978-1-6654-9156-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/LDAV57265.2022.9966396

efit of distributing the model data across multiple nodes no longer exists, particularly for ray- or path-based renderers. Distributing the model can still make sense if it was otherwise too large to fit onto a single node, but doing so would likely lead to only marginal performance gains (if at all). It could easily result in significantly lower performance than single-node rendering once compositing/communication cost is considered. Consequently, such renderers today have two options: either use image-parallel rendering and benefit from a relatively easy way of increasing performance by adding more ranks—but only as long as the model can be replicated across every node; or use data-parallel rendering to render virtually any size of model—but render barely faster (and often, significantly slower) than a single node could have done if only it could have fitted the data.

In this paper, we explore the idea of what we call *island parallelism*, which aims to combine the strengths of both in a *hybrid* image- and data-parallel manner: given $R = N \times M$ ranks, we split these ranks into M *islands* of N ranks *each*; and perform data-parallel rendering *within* each island, but image-parallel rendering *across* the islands. In other words, each island would collectively render a different set of pixels just like any *rank* in traditional image-parallel rendering would, but N ranks within each island and perform data-parallel rendering for only the pixels assigned to this island. In this setup, island-parallel rendering covers a spectrum in which classical data-parallel and image-parallel rendering are the two extremes ($M = 1$ and $N = 1$, respectively). Unlike classical image-parallel rendering, it is not limited by any given model size because N can always be increased until the model fits. And unlike classical data-parallel rendering, this method does not have an obvious diminished return when adding more ranks than are required to fit the model (as long as these come in multiples of how many ranks are required to fit the model).

The core idea behind this data-replicated model is simple enough and also not entirely new. How exactly this would, however, perform in practice, and mainly when applied to ray tracing-based pipelines that are ubiquitous nowadays in both photorealistic rendering and scientific visualization—so far has not been explored rigorously and for different types of rendering techniques (e.g., Monte Carlo surface rendering, volume ray marching vs. Woodcock tracking)—in any research that we are aware of. We first sketch how we integrated this idea into several very different renderers, then evaluate these renderers on various data sets and configurations. As this evaluation shows, our hybrid model offers the flexibility to be implemented over a diverse set of rendering frameworks and computing resources while allowing for decent scalability.

2 Related Work

In this section, we review prior works on distributed rendering algorithms, which broadly fall into data-parallel sort-last techniques and their corresponding image compositing algorithms, as well as data-replicated techniques, which are—in the context of high-performance and large-scale visualization—usually realized using sort-first.

2.1 Data-Parallel Rendering

In compliance with Molnar's taxonomy [20], sort-last parallel rendering algorithms initially distribute the data to N ranks that use task parallelism to generate intermediate images. After all ranks have finished rendering, these are assembled using *compositing*.

Traditional works that implement data-parallel surface [7] or volume rendering [24, 27] follow this pattern exactly and require only little extra communication prior to compositing. With incoherent workloads (both in terms of data distribution and rays being traced), the need for advanced scheduling [25], caching [6], or distributed shared memory approaches [11] arises, resulting in ever more complex communication patterns.

More recent work, e.g., by Abram et al. [1], by Jaroš et al. [12], or by Wald and Parker [35], concentrated on incoherent workloads typical for production rendering with path tracing. Advancements in this field have however also led to path tracing becoming more relevant for scientific visualization [13]. Another layer of complexity is introduced by 3D models using instancing. The works by Zellmann et al. [39] and by Wald and Parker [35] have recently focused on spatial and object partitioning for the distributed rendering of instanced models.

With incoherent ray tracing and Monte Carlo path tracing workloads becoming ever more relevant for the scientific and high-performance computing communities, renderers have to concentrate even more on ray queuing and forwarding techniques, or on instancing, and by doing so, introduce potential diminishing returns from adding extra ranks due to increased communication overhead.

2.2 Parallel Compositing

While assembling final images is trivial with image-order techniques—threads that render a set of pixels at the end save their result to exclusive memory addresses—compositing intermediate images with data-parallel rendering is a challenge in itself and can exhibit undesirable scalability when not implemented carefully.

Even the simplest of techniques, such as direct-send compositing [8], subdivide image space into disjoint regions, and assign individual ranks that are responsible for compositing those regions. With direct-send, scalability issues arise because processors send their intermediate image portions to the ranks responsible for compositing at the same time. Traditional *round-based* techniques (e.g., binary swap [17]) resolve this issue by dividing the communication into *consecutive* rounds; within each round, the number of processors halves, while the number of image fragments sent per processor pair doubles, resulting in a binary tree communication pattern that can only accommodate power-of-two rank counts. To overcome this issue, round-based algorithms factorizing the number of ranks into, e.g., pairs and triplets [38], or more generally into k_i workers per $i \in M$ rounds (radix-k [26]) have become the de facto standard implemented by compositing libraries such as IceT nowadays [23].

There is an interesting parallel between radix-k [26] and our hybrid image-/data-parallel islands concept in that both techniques solve the issue of accommodating non-power-of-two rank counts; while radix-k achieves this by subdividing the communication task into rounds that execute sequentially (and within each communication group perform direct-send), island parallelism subdivides image space among communication groups (where each group can potentially use any compositing algorithm suitable, *including* direct-send). As each communication group/island holds a full copy of the data, our equivalent of the rounds in radix-k can now run in parallel, thus allowing for improved scalability but at the expense of using more memory.

2.3 Hybrid and Data-Replicated Rendering

Although the traditional sort-first formulation by Molnar [20] was data-parallel, sort-first is nowadays most often realized using data replication [3]. Moloney et al. [21] propose caching schemes replicating only parts of the data in each GPU's memory. Cambazoglu and Aykanat [4] reformulate the screen-space load-balancing problem as a hypergraph partitioning problem to achieve equal load distribution while minimizing data replication.

There also exist hybrid approaches, e.g., the distributed frame buffer method by Usher et al. [33], where the data is distributed, but the parallelization and task assignment are driven by image tiles. Another straightforward, hybrid approach proposed by Larsen et al. [14] assumes that the sequence of images rendered is known a priori and then parallelizes over the number images.

The approach by Samanta et al. [31] is closely related to ours and presents what can be considered an early implementation of our

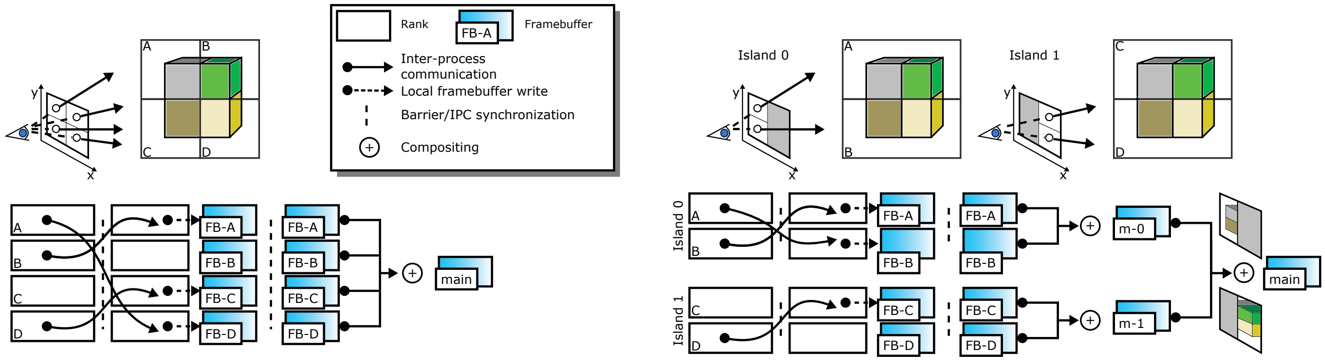


Figure 2: Left: Control flow and communication patterns of an example data-parallel renderer whose inner workings are not further specified but represent the *kind of* renderer we consider in this paper. At the top we show how the data and work are distributed to ranks (A)-(D). At the bottom, we present the communication patterns resulting from that. The left side presents a version of the renderer without island parallelism; the right side shows the same renderer with island parallelism. This renderer generates primary rays on reach rank, traces them into the data, and eventually writes a color to a local frame buffer. At the bottom, curved arrows represent inter-process communication, and vertical lines indicate synchronization barriers. One can see that inter-process communication only occurs between the *local* ranks of an island; even the final *image* compositing step requires only little synchronization since the islands write their results to exclusive addresses in the global frame buffer.

islands concept. Samanta et al. focus on rasterization and primary visibility only. Realizing that even with their coherent workloads, there are diminishing returns from adding more compute nodes than needed to fit the model, the authors propose to organize the participating ranks into K groups, each group holding a full copy of the data set. The authors complement that with a view-dependent work scheduling scheme that changes from frame to frame.

However, the workloads tackled by Samanta et al. [31] are fundamentally different from ours. With a shift to hardware-accelerated ray tracing, today's workloads are increasingly incoherent. With ray-tracing methods, we observe a $\log(n)$ or $\sqrt[3]{(n)}$ complexity per pixel rendered where n is the number of primitives/voxels, which is different from what one expects in the context of rasterization. Methods such as the one by Reinhard et al. [29] have accounted for this in terms of load balancing, but to our knowledge, data replication has not been proposed as a degree of freedom to improve the scalability of incoherent rendering workloads.

To our knowledge, Samanta et al.'s work from 2001 is the most recent one to propose k-way data replication to improve scalability; we believe this type of scheduling to become more important in the presence of the aforementioned incoherent workloads and complex communication patterns. Yet, we observe that concrete realizations of our islands concept cannot be found "out in the wild", as the state-of-the-art packages for large-scale distributed rendering, such as Paraview [2] or VisIT [5] do not implement anything similar. This motivated our desire to (re-) evaluate this concept/technique in the context of modern rendering algorithms.

3 Hybrid Image/Data Parallelism via *Islands*

The main observation that motivated this paper is that, at least for mainly ray- and path-based renderers, data-parallel rendering and image-parallel rendering operate on two completely different ends of the spectrum of parallel rendering: On one hand, data-parallel rendering can be used to scale to almost arbitrary model sizes by simply adding more work, but when scaling to node counts beyond what a given model requires, there is usually a very quickly diminishing—and possibly even negative—return of adding more nodes. Conversely, image-parallel rendering has been shown to enable almost trivial scaling of such renderers in frame rate and image quality (as in samples/paths per pixel).

The obvious alternative is to *combine* these two techniques: we take a set of $R = M \times N$ parallel resources and split them into M what we call *islands* of N ranks each. Within each island, the N ranks perform data-parallel rendering such that in its entirety, these N ranks *could* render every pixel on the screen. Suppose we now create M different such islands that are exact copies of each other. In that case, each such island could render every pixel—which means we can now use image-parallel rendering *across* these M islands (i.e., each renders only approximately one M'th of the pixels).

One way of viewing image- and data-parallel rendering is as opposite ends of a spectrum; another is as two orthogonal axes that together form a 2D plane. In the latter case, the $R = N$ axis for $M = 1$ is purely data-parallel rendering where the model is split into $N = R$ parts, and each rank works for potentially every pixel. The $R = M$ axis for $N = 1$ is purely image-parallel rendering where the model is not split at all, but the image is split into $M = R$ different regions that get assigned to different nodes. In that view, hybrid parallelism is the *super-set* that spans the entire plane between these two axes: the model is split into N parts for data-parallel rendering, and the image into M regions for image-parallel rendering. This formalism also generalizes prior works, such as that by Samanta et al. [31], which concentrated on a particular, view-dependent work distribution.

Island parallelism is a powerful *concept* as described above, but not actually an *algorithm* that can quickly and obviously be thrown into any given renderer: how to integrate it depends on how exactly the renderer works. Similarly, the benefits of applying it will depend on how exactly the renderer works, as a renderer whose dominant cost is pixel count or paths per pixel will be affected differently than one for which this is not the case.

To introduce some notation used throughout the paper, let us consider the imaginary renderer in Fig. 2. The renderer presents a template for the type of renderers we focus on in this paper: it works off an a priori data distribution into $N = 4$ parts (how these parts were derived, for this imaginary renderer, remains unspecified). The renderer generates a set of camera rays on each rank and renders its portion of the data in parallel. During rendering, the ranks participating in the rendering process may or may not communicate with each other, and if they *do* communicate, it may or may not be necessary to use barriers or other blocking operations to synchronize the ranks. Finally, all ranks are synchronized, composite the intermediate images present in their *local frame buffers*, and send the result to the display rank. Again, how exactly this compositing operation works is unspecified and depends on the concrete implementation.

On the right, Fig. 2 shows how the same workload gets distributed across $M = 2$ islands; inter-process communication and synchronization now only occur among ranks within an island; the islands finally assemble the final image using sort-first compositing.

In the remainder of this paper, we will look into four very different data-parallel renderers that present concrete realizations of

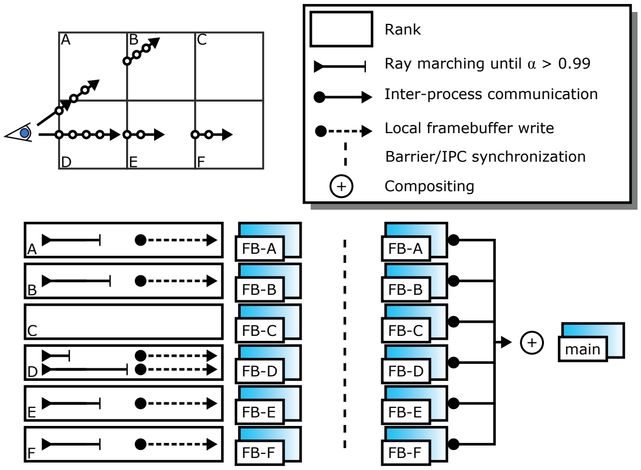

Rank
Ray marching until α > 0.99
Inter-process communication
Local framebuffer write
Barrier/IPC synchronization
Compositing
A
B
C
D
E
F
FB-A
FB-B
FB-C
FB-D
FB-E
FB-F
main

Figure 3: Control flow and communication patterns of a simple absorption plus emission ray marcher for structured volumes as presented in Section 4. Top: the data set is distributed spatially among processors (A)-(F). Rays are marched with equidistant steps, and when the ray terminates—either because opacity reached a threshold or the ray left the node's bounds—the result is written to the local frame buffer. Bottom: the corresponding communication pattern. All rays are marched in parallel. Potential divergence occurs due to different step counts through the density. When all rays finished marching, results are combined using sort last compositing (bottom right). Barrier synchronization is only needed for compositing.

the imaginary template renderer introduced in Fig. 2. The renderers were implemented as part of various projects over the last few years and differ wildly in how they realize the data-parallel rendering and compositing operations. We describe how we added the islands paradigm for these four renderers. We then perform scalability studies and discuss why some renderers are affected differently than others.

4 Renderer 1: Sort-last Structured Volume Ray Marching

The first renderer we focus on can be viewed as the equivalent of a traditional, MPI-parallel, sci-vis direct volume renderer and is representative of the implementations found in, e.g., ParaView [2] or VisIT [5]. The renderer supports absorption plus emission with on-the-fly gradient shading, all of which can be realized by marching coherent camera rays through the volume in equidistant steps.

To this end, the renderer can be viewed as a baseline, which is later—at least conceptually, the other renderers derive from entirely different projects and are also maintained by different teams—extended to support complex 3D models and shading modes.

4.1 Implementation

The renderer's control flow and communication patterns are presented in Fig. 3. The implementation is based on the framework used by Wald et al. [36]. However, we deliberately deactivate the space skipping component, leaving us with a renderer that marches camera rays generated in a CUDA kernel through a volumetric density stored in a 3D GPU texture. We terminate rays early if they leave the scene bounding box or accumulate up to 99% opacity.

To realize data-parallelism, we split the volume and corresponding 3D texture into one brick per rank using a kd-tree builder and the split-middle heuristic. The bricks include ghost cell layers to support trilinear interpolation and gradient shading. The ranks compute intermediate images in parallel and use IceT [22] for sort-last compositing. To determine the correct order to composite the local frame buffers required by IceT, we use the kd-tree from before.

4.2 Extension to Island Parallelism

Implementing island parallelism within this framework is straightforward. To distribute the data among ranks, we first perform an `MPI_Comm_split` of the global world communicator into as many communicators as we have islands; each island has as many ranks as there are bricks.

Each rank first generates primary rays for the whole viewport to distribute the work, including those regions that we might potentially know are never overlapped by the brick assigned to this rank. Inside the CUDA kernel, we group the threads into tiles and directly terminate those whose tiles are inactive. Tiles are, however, just one possible way to distribute the work, and the other renderers use different work distribution schemes. As each rank renders full-size images, the images composited with IceT will span the whole viewport.

4.3 Experimental Setup

To evaluate this renderer, we use the turbulent channel flow simulation direct numerical simulation (DNS) data set from [15], which is $10240 \times 7680 \times 1536$ voxels in size, and with four bytes per voxel amounts to 450 *GB* total. As this data set, including ghost layers, saturates a significant portion of the available memory on a typical GPU cluster, we also run benchmarks with a downsampled version of $5120 \times 3840 \times 768$ floating point voxels.

Since the form of the data set resembles that of a sheet, we test with two different viewpoints (cf. Fig. 6); one that zooms in on the data but will result in culling a significant portion of the bricks to the left and right, and another one that is zoomed out but contains significant amounts of white space.

5 Renderer 2: Hardware-accelerated Production Path Tracing

Our second renderer deviates mainly from the traditional high-throughput, coherent workload pattern and is a data-parallel path tracer for *production-style* content; i.e., for content that is primarily triangle-based, but with instantiation, with materials, textures, light sources, with reflection, refraction, and multi-bounce indirect illumination. This is the same renderer used in [35].

5.1 Original Renderer without Island-Parallelism

A complete discussion of this data-parallel path tracer is beyond this paper's scope, so we will limit ourselves to only a brief, high-level description (also cf. Fig. 4). At its highest level, this renderer builds on *ray forwarding* (i.e., it sends rays to the node(s) that have the data that these rays need to interact with), and it operates in two distinct phases. In the first phase, this renderer generates primary rays that will eventually become *paths* and traces these into the scene in a wavefront manner. Ranks maintain a set of *ray queues* containing rays that need processing on this rank. Primary rays get generated by the ranks that own the closest spatial region for that ray and get put into the ray queue.

5.1.1 Stage 1: Path Tracing with Ray Forwarding

In each step, the renderer traces its rays into its local rank's geometry using CUDA [18] and OptiX [19]. After that, it determines whether a ray needs forwarding to another rank for further processing. It either shelves the ray for later shading (if no forwarding is required) or puts it into an output queue from where it can be forwarded. After all of the rays are locally processed, the ranks collaboratively exchange rays based on where each ray wants to get forwarded to until all rays are ready for shading. A shading stage then shades all rays, generates secondary and shadow rays as required, and iterates back to the tracing stage until all paths have been traced to completion.

This first stage does all the tracing, shading, and ray forwarding. During this stage, rays can make *image contributions*. For example, a ray gets shaded when a shadow ray realizes that it is not occluded.

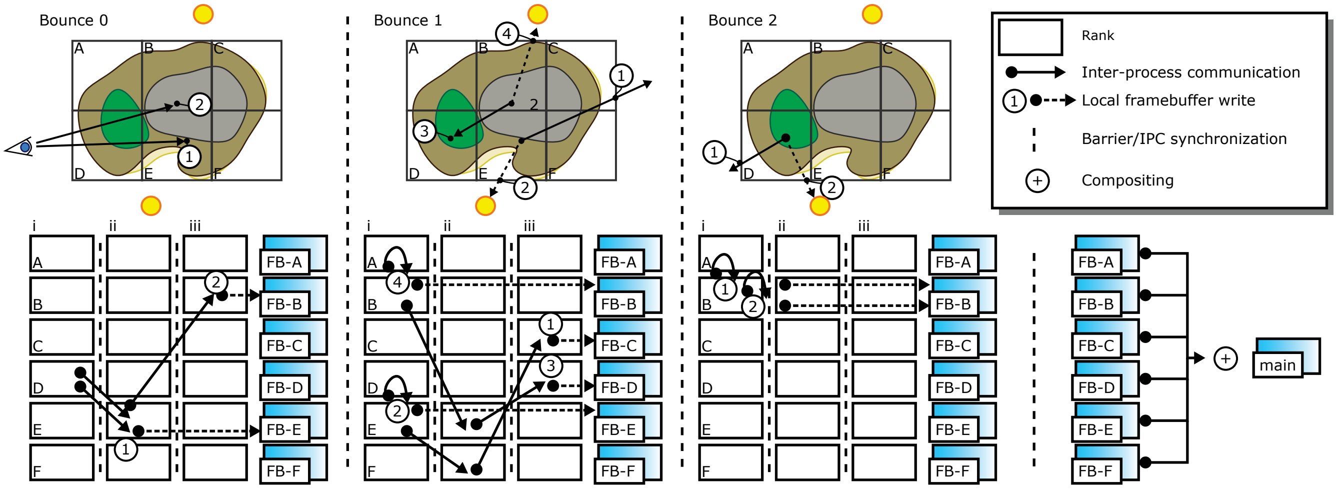


Figure 4: Control flow and communication patterns of a typical ray-forwarding path tracer as the ones described in Section 5 and Section 6. The illustration shows two paths with three bounces, including BRDF sampling and next event estimation (NEE). Top: spatial distribution of the scene among processors (A)-(F). Solid arrows represent BRDF rays, dashed arrows represent NEE rays, and circles with numbers indicate geometry or boundary hits. Bottom: the corresponding communication pattern (solid arrows). Rays bounce and are forwarded between ranks multiple times ((i)-(iii)). Geometry or light interactions on a local processor result in partial/local frame buffer writes (dashed arrows). The algorithm exhibits inter-process communication and the need for synchronization during and in-between bounces. Local frame buffers are eventually combined using compositing (bottom right).

As illustrated in Fig. 4, irrespective of which rank *started* tracing a ray through a particular pixel, any other rank can potentially contribute to this pixel, depending on where this ray (or its descendants) gets forwarded to. This renderer combines image contributions from different ranks by having each rank keep a full-sized frame buffer that the first stage's operations can atomically add their image contributions to. The final image then is the *sum* of what we call *partial* frame buffers. The renderer's second phase then adds these partial frame buffers, which this implementation does via a parallel direct-send as, e.g., described by Grosset et al. [9].

5.1.2 Stage 2: Adding the Per-Rank Images

Assuming the scene was split into N parts, the renderer would use $R = 1+N$ MPI ranks: one for a master that runs the viewer and controls the worker, and N ranks to perform data-parallel rendering. Upon startup, the renderer executes an `MPI_Comm_split` that splits the initial ranks into two groups—one for the master and one for the N workers—with an intercommunicator between them. In the path tracing stage, the workers operate exclusively on the worker group, realizing ray exchange using one call to `MPI_Allgather` to tell each rank how many rays it can expect to get sent, followed by an `MPI_Alltoall` to move the rays to other workers.

In the image adding stage, the workers first horizontally split the frame buffer into N parts; i.e., each part comprises a range of consecutive, entire scan lines, with N parts total. Using an `MPI_Allgather` on the worker communicator, each rank performs direct-send by moving its $N-1$ regions to the other ranks and receiving the $N-1$ other ranks' pixel contents for its regions. After adding its pixel content, the worker is also responsible for tone mapping. Finally, each rank sends its composited scan lines to the display rank using an `MPI_Isend` on the master/workers intercommunicator, using the scan line's `y` coordinate as the MPI tag. The display rank sets up a matching `MPI_Irecv` for the scan lines using the same tags and then uses `MPI_Waitall` until all scan lines are received.

5.2 Extension to Island Parallelism

To extend this framework to island parallelism, we chose to interleave the final frame buffer's scan lines among the M distinct and interleaved sets of scan lines (where M is the desired number of islands) so that scan lines 0, M, and $2M$ go to Island 0, scan lines 1, $M+1$, and $2M+1$ go to Island 1, and so on. As a result, each island operates on what is essentially a frame buffer of one M'th the height of the real frame buffer, but within this frame buffer, it can behave almost as if there were no island parallelism.

Operations like frame buffer resizing have to be modified to compute the right size of the *island* frame buffer, but ray queue management can then operate on that (smaller) frame buffer in the same way as before. Ray generation has to know that each pixel (x,y) in the island frame buffer corresponds to pixel $(x, I+yM)$ in the final frame buffer (where I is the index of the island that this rank is in); but other than that can again operate as before. Ray traversal and shading operate just as earlier (they only operate on ray queues), and even the entire ray exchange step can remain as is as long as each island gets its own communicator. To do this, we run a second `MPI_Comm_split` on the worker's communicator that splits that into M groups (one per island) and then have the workers operate on that communicator rather than the parent worker's communicator. This way, islands can operate entirely autonomously, with the communication code not even needing to know that other islands even exist.

For the image-adding stage, too, changes are minimal: we again use the island communicator for the parallel direct send stage, meaning each island can run its image-adding stage completely orthogonally. In the final step, the master can perform its `MPI_Irecv` operations as it did before. The only thing that changes is that the workers need to translate each sent scan line's tag back to the y coordinate of the final image (i.e., replacing tag y with tag $I+yM$).

5.3 Experimental Setup

To evaluate this renderer, we focus on traditional ray tracing/production content, in the form of the PBRT landscape scene. The scene comprises 23 K different plant models that are spread out across a base mesh using instancing; the total triangle count is 24 M, but due to instancing has a geometric complexity of 3.1 B triangles [28]. The scene contains surfaces with several different physical materials and exhibits complex lighting from an HDRI environment map. We split the scene into up to 44 parts (further splitting the model results in the surface area heuristic (SAH) based splitter generating empty parts) that we distribute across ranks and islands.

6 Renderer 3: Volume Path Tracing

The next renderer we will look at is targeted toward data-parallel volume path tracing rather than surface rendering. It explicitly aims for high-quality volume path tracing with shadows and scattering but is primarily designed for scientific visualization data.

While this renderer is different from the one in the previous section in many core routines, it still shares many similarities regarding high-level structure. The communication patterns are fundamentally the same as shown in Fig. 4: the renderer uses the same communicators for MPI display master and workers, uses ray forwarding and queuing, and accumulates arbitrary pixel contributions in partial frame buffers on the rendering stage; similarly, compositing is realized using parallel direct-send, like with the surface path tracer. However, the renderer's approach to managing and integrating its data is fundamentally different. Unlike the surface renderer, this renderer does not support next event estimation or any form of path splitting. However, paths can still have multiple bounces and/or stochastically turn into shadow rays if and when desired. Instead of surface data with instances, similar to the one from Section 4, this renderer operates on bricks of structured volume data with ghost layers.

The renderer implements Woodcock tracking [37] with an isotropic phase function. For that, instead of tracing rays into an OptiX acceleration structure, it uses a macro cell hierarchy as proposed by Günther et al. [10], using Digital Differential Analyzer (DDA) traversal over macro cells to reduce the number of rejection samples taken to account for null collisions. Each time a sample gets rejected, we simply go on. Otherwise, we schedule this sample for shading, at which point the ray stochastically samples how the ray will interact with the volume at the position where the collision occurred (e.g., it can become a shadow ray to a sampled light source). In addition to Woodcock tracking, this renderer supports implicit iso-surface ray tracing integrated into the same macro cell DDA traversal.

6.1 Extension to Island Parallelism

As discussed in the previous sections, we did the same to integrate hybrid parallelism into this renderer. Though the code bases are different due to different ray types and operations on top of those rays, the basic technique of interleaving scan lines across islands, particularly the per-island compositing and merging at the master, is conceptually identical.

6.2 Experimental Setup

This renderer targets scientific visualization rendering as the main application. To evaluate this, we chose two very different models (of also different sizes), as shown in Fig. 6: the *chameleon* data set consisting of $1024 \times 1024 \times 1080$ voxels represented with 32 bit floating point precision, which, in this example, is rendered with both Woodcock tracking (using a relatively "spiky" transfer function) and implicit iso-surfaces; at these settings, the model contains a lot of (post-transfer function) "empty" space where rays can travel across multiple ranks. As a second example, we chose the same turbulent flow DNS data from Section 4 with the same viewpoints and sizes of $10240 \times 7680 \times 1536$ and $5120 \times 3840 \times 768$.

7 Renderer 4: Element-marcher with Sort-last "Deep" Compositing

The fourth and final renderer we are looking at is an element-marcher with a sort-last "deep compositor" designed to render the two versions of the Fun3D Mars Lander Retropropulsion study data set, which is also used by Wald et al. [34]. The "Small" version of this data set is pre-partitioned into 72 non-convex, unstructured meshes with a total of 798 million unstructured elements. The "Huge" one comprises 552 meshes with 6.38 billion elements total. This renderer can be seen as a modern, interactive, and GPU-targeted re-design of the basic ideas already described by Ma et al. [16]. A high-level overview is given in Fig. 5. The implementation extends the one by Sahistan et al. [30] to support data-parallel rendering using MPI. It uses a combination of three techniques. First, it utilizes hardware-accelerated triangle ray tracing using OptiX to determine where a given ray will enter and exit (i.e., ray segment) the non-convex mesh(es) that each rank stores, including the boundary face mesh that the ray enters. Then, it employs *element marching* to step through the unstructured mesh and take (at least) one sample per element. Every time the marcher retires a segment (i.e., when the ray leaves the mesh on a boundary face), the renderer registers a *fragment* comprised of an RGBA color and a depth Z, which gets stored in a CUDA buffer during rendering. Fully transparent fragments get rejected. The fragments get stored in a list that we call the *deep frame buffer*, which contains a variable number of fragments per pixel.

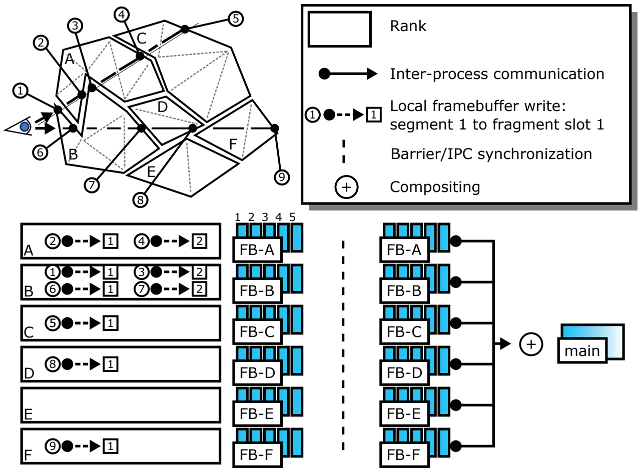

Rank
Inter-process communication
① ●--▶ 1 Local framebuffer write: segment 1 to fragment slot 1
Barrier/IPC synchronization
⊕ Compositing
A
B
C
D
E
F
1
2
3
4
5
6
7
8
9
1 2 3 4 5
FB-A
FB-B
FB-C
FB-D
FB-E
FB-F
main

Figure 5: Control flow and communication patterns of an unstructured element ray marcher with non-convex bounding geometry as described in Section 7. Top: each processor (A)-(F) is assigned one *cluster* containing unstructured elements; primary rays are marched through the density and intersected with their cluster, resulting in *segments*, and *fragments* that represent the integrated color and depth of the segment. Bottom: each segment is integrated, in parallel per ray and sequentially in visibility order; each fragment is then written to a sequential slot in the "deep" frame buffer. Bottom right: at the end, fragments are composited using deep frame buffer compositing.

After the rendering stage, the fragments assigned to one pixel are generally scattered across multiple ranks; these ranks now exchange their fragments in a way that is essentially a generalization of direct-send. Instead of sending one fragment per rank per pixel, the variable list of fragments is sent using a `MPI_Alltoall`. Since the fragment count is not known at the receiving ranks, the ranks first exchange a list with a prefix sum computed over the fragment counts, also using `MPI_Alltoall`, to determine which slots the later exchanged fragments will go to. After that exchange, each rank merges and sorts its per-pixel fragment lists, composites them in front-to-back order, and performs tone mapping. Each rank then sends its final pixel to the master for final display.

We chose to include this renderer because it operates very differently from the others. In particular, unlike the surface- and Woodcock-based renderers, it does not have some automatic built-in "early ray termination" that aims at quickly finding a surface or volume interaction at which the ray can terminate traversal. Instead, each segment gets integrated separately and cannot know whether other ranks might generate closer fragments that will occlude this segment. Also, the number of samples taken per ray depends much more on how much of the total model a given rank has. Partitioning the model across more ranks will automatically mean that each rank has fewer elements in total and, on average, fewer fragments it will generate for a typical ray.

7.1 Extension to Island Parallelism

Despite being different in nature from the other three renderers, applying the concept of hybrid parallelism is strikingly similar. In the first phase, each rank renders fragments by tracing one ray per pixel. As before, we interleave scan lines and adjust the pixel coordinate during ray generation; each rank now renders $M\times$ fewer scan lines (into a correspondingly smaller frame buffer), but other than that, this stage will work exactly as before: rays get generated, traced, and integrated, and fragments will get stored just as before. For the compositing stage, the steps performed inside each island are exactly as before. As such, we again `MPI_Comm_split` the workers into M separate island communicators and then perform the same compositing operations as before.

7.2 Experimental Setup

We will exercise this renderer on two differently sized data sets to be able to scale up the number of islands and ranks flexibly. Similar to the DNS data set, the huge lander occupies a significant portion of the memory available on our test system and allows us to add just one additional island. Therefore, we also test with the small lander, which will enable us to scale our benchmarks up in both the number of islands and the number of GPUs per island. Since both models already come pre-partitioned based on how Fun3D partitioned the models for its simulation, we cannot easily repartition them. Instead, to test scalability, we have each rank pick a number of bounding meshes/clusters in a round-robin fashion, where rank 0 is assigned the $0th$, Nth, $2Nth$ boundary mesh, rank 1 is assigned boundary meshes $1, N+1, 2N+1$, and so on.

8 Evaluation

This section evaluates our hybrid image-/data-parallel island renderers using the experimental setups described above. For the evaluation, we migrated the renderers to run on the RTX partition of the Frontera supercomputer at the Texas Advanced Computing Center (TACC) [32]. In the following, we provide a brief overview of the hardware and software infrastructure provided by this system.

8.1 Frontera RTX System Overview

On the hardware side, Frontera's RTX partition comprises 90 GPU nodes, each of which is equipped with four NVIDIA Quadro RTX 5000 GPUs. A compute job can allocate a maximum of 22 nodes, amounting to 88 GPUs. Each GPU is equipped with 16 GB GDDR6 memory. The interconnect uses Mellanox's HDR technology with 100 Gb/s bandwidth for inter-node communication.

On the software side, Frontera's RTX partition supports a regular CUDA 11 plus OptiX 7 workflow, which allows us to run all our renderers on this system. The `mvapich2-gdr` MPI suite installed on the system is CUDA-aware and *GPUDirect*-enabled [40].

A CUDA-aware MPI implementation will recognize when CUDA device pointers are being passed to MPI functions. When an MPI implementation is also *GPUDirect*-enabled, and processes from two different nodes connected via Mellanox communicate with each other, the communication will go directly through the fabric without a detour through the host memory.

This architecture allows the path tracers to implement ray forwarding with direct memory access from GPU to GPU; the deep compositing algorithm from Section 7 can also make use of this by directly exchanging the fragments between devices using DMA-enabled `MPI_Alltoall` calls. Only the simple structured volume baseline renderer from Section 4 cannot use any such optimization, as IceT does not support local frame buffers stored in CUDA device memory.

8.2 Scalability Study

Based on the experimental setups outlined in Sections 4.3, 5.3, 6.2 and 7.2, we evaluate scalability by splitting the respective models into bricks/parts, or use the clusters present in the data to distribute the data across nodes; when running the benchmarks, we manipulate two degrees of freedom: the number of overall GPUs used (we increase this number in at least multiples of four, as a node on Frontera contains four GPUs), and the number of islands; as we are restricted to 88 GPUs total, and as some of the parts or bricks/clusters cannot be split beyond a certain point/exceed certain size limits, not all our benchmarks can be run in all theoretically possible combinations.

The exemplary scalability plots in Fig. 7 contain *two* types of curves that are of significance to us: one that is drawn explicitly and another that can be observed when following the contours of the data points. For that, let us consider an idealized renderer—representative, e.g., of how a surface raycaster with a bounding volume hierarchy accelerator and no inter-process communication overhead at all would perform in the limit and that, per pixel, performs $\log(n)$ work in the number of primitives n. A scalability plot for that idealized renderer is presented in Fig. 7. We can interpret these plots in two ways: in Fig. 7a, each of the colored curves represents weak scalability, where the model/data set is split into smaller pieces and more GPUs are devoted per island, resulting in the typical diminished return mentioned before. For a renderer with more complicated communication and dependent on certain implementation constants, we expect the scalability to be different, if not negative, within one island.

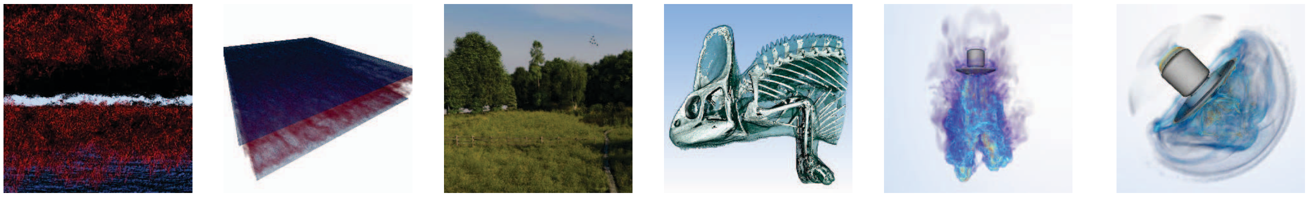
An alternative way to read the plots is indicated in Fig. 7b, where we show the same plot as in Fig. 7a, but with contour lines added that illustrate *island scalability*. Whenever we decide to double the number of GPUs devoted to the rendering task, we can either increase the number of GPUs per island or add another island; while the former decision is reflected by following the colored lines, the latter is reflected by going to the right on the contour lines.

We hypothesize that for the workloads we take under consideration, we will observe similar behavior as with this idealized renderer, where scalability from adding more GPUs to existing islands results in diminishing or even negative returns. In contrast, the performance from adding more islands ideally increases systematically and at a higher monotonic rate. In the following, we will test this hypothesis with our four renderers.

8.3 Results

We present the results of our scalability study in Fig. 8. Fig. 6 provides an overview of the data sets used and Sections 4.3, 5.3, 6.2 and 7.2 discuss the choice of data set size and composition. We deliberately choose data sets of different sizes, such as the huge DNS or NASA Mars Lander data sets that saturate almost half of the available GPU memory. Smaller versions of the data sets, or data sets that draw their complexity from instancing, allow us to scale up higher in the number of GPUs and the number of islands. While some of the data sets allow us to collect data points at a fine granularity, others can only be run on certain configurations with a fixed number of GPUs, e.g., the lander data sets with their fixed number of clusters or data sets where the partitioners cannot create splits with finer granularity. We render 1K × 1K images, as we consider this to represent the rendering configuration one would usually use in a typical in-situ/in-transit or remote rendering session.

We observe several effects. Firstly, our tests generally confirm that, for the kind of workload under consideration, island scalability is higher than scalability from adding extra GPUs to existing islands. We also observe that the renderers with little communication overhead during rendering—i.e., the structured ray marcher and the unstructured element-marcher—show diminishing returns or plateaus, while the path tracing renderers, particularly when exercised on larger data sets, exhibit negative returns. This is likely at-



DNS View 1 DNS View 2
Huge: 10240 × 7680 × 1536
Small: 5120 × 3840 × 768

PBRT Landscape
23 K Instances
24 M Triangles

Chameleon
1024 × 1024 × 1080

Small Lander Huge Lander
Huge: 6.13 B tets, 284 K pyramids, 255 M wedges
Small: 766 M tets, 47 K pyramids, 31 M wedges

Figure 6: Data sets used for our evaluation. From left to right, top to bottom: two views of the DNS turbulent flow data set; the PBRT landscape data set, with a geometric complexity through instancing of 3.1 B triangles; the Chameleon CT data set, with direct volume rendering and implicit ISO surface for the bone structure; the Small and Huge NASA Mars Lander, two unstructured data sets with complex, simulation-provided bounding geometries.

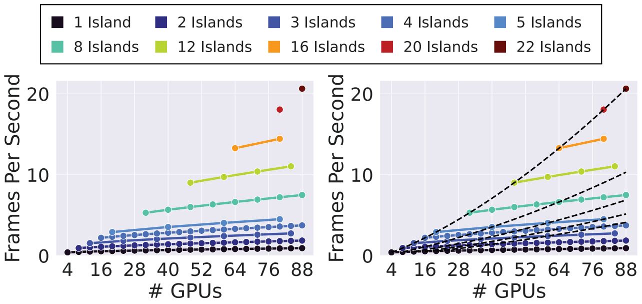


Figure 7: Two ways to read our scalability plots for a hypothetical renderer performing $\log(n)$ work per pixel and n primitives. Left: the colored curves represent scalability within M islands. Right: we add contour lines to that plot that represent *island scalability*: the gain in performance when, instead of doubling the number of GPUs per island, we create *another* island with the same number of GPUs.

tributable to the communication overhead for ray forwarding, which imposes barrier synchronization before and after compositing and for each reflective or shadow ray bounce. Here, islands help relax this scalability issue inherent to these renderers significantly.

Another effect we observed during our study, confirmed by the island scalability curves eventually flattening out, is reduced GPU utilization. For the structured ray marcher, e.g., we found that image sizes below 800 × 800 pixels cause the GPUs to become underutilized, as the latency incurred by threads performing memory accesses cannot be effectively hidden behind compute. With islands, the number of pixels that a single GPU renders decreases, and so does GPU utilization. With 22 islands/88 GPUs and 1K × 1K images, e.g., each GPU is responsible for rendering roughly 110 × 110 pixels. We expect each renderer to be susceptible to this issue. We believe this to be another reason for island scalability to diminish eventually.

In Fig. 8, we have aimed at plotting the performance for all kinds of possible combinations of M islands and N ranks per island, and as predicted, have seen that adding more islands will generally increase performance more than adding more ranks for the same number of islands. Ultimately (and as an indirect continuation of the arguments we made in Section 3 and Fig. 7) this suggests that the "ideal" performance for any number of available ranks should always be where the user chooses the smallest possible number of ranks per island that can still represent the model, and invest any additional resources into adding more islands—we will call this configuration the "ideal" island configuration for a given number of ranks. For any renderer and number of ranks, we can then also compare the performance of this ideal island configuration to the performance that a pure data-parallel configuration (i.e., the same renderer with a single island) would have achieved.

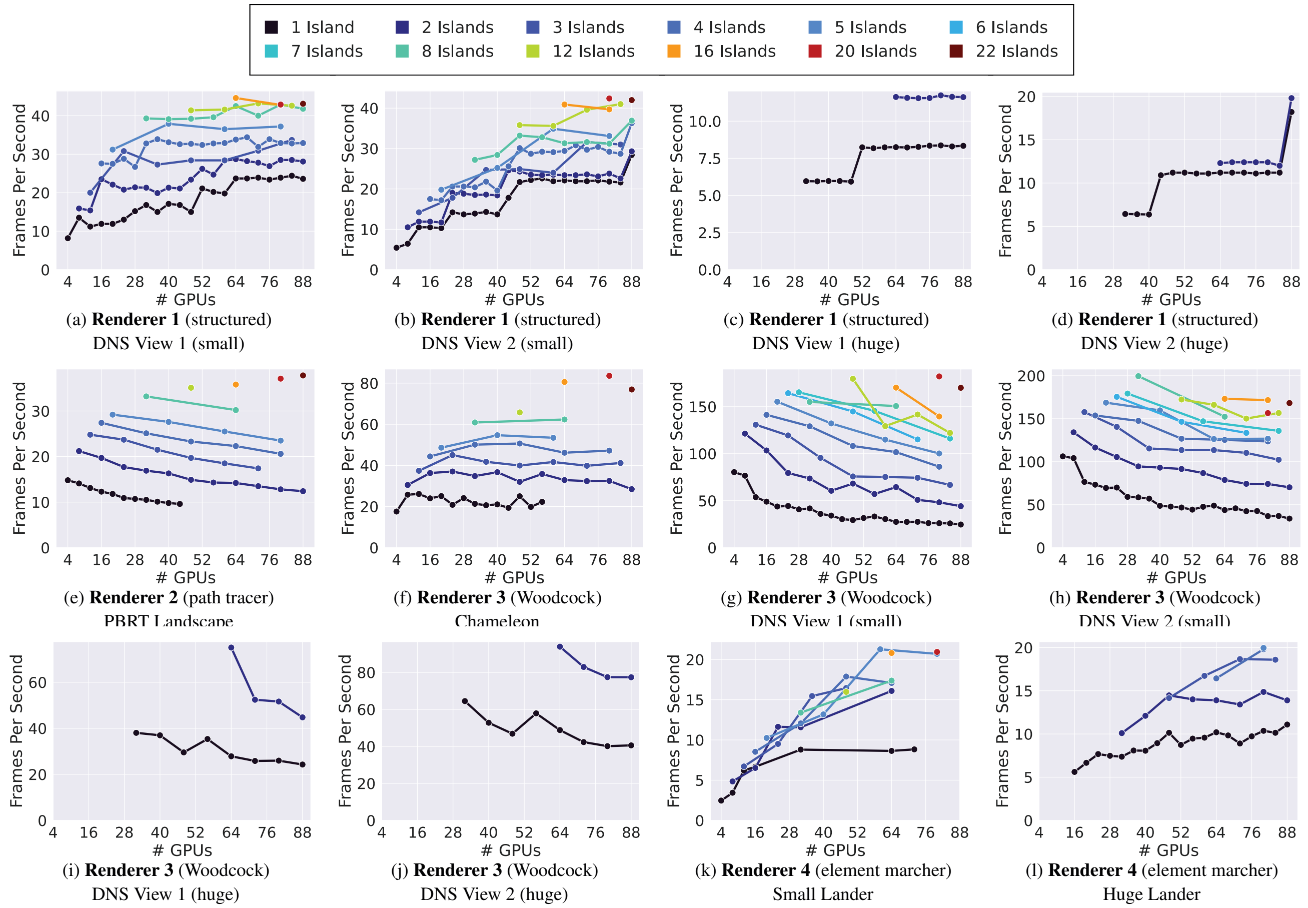
We present the result of this experiment in Fig. 9: as can be seen, even with island parallelism these scalability graphs are not perfectly linear; this is the result of multiple factors, including tail end effects (some pixels being much more expensive than others), impact of compositing cost, starvation due to increasingly fewer pixels per island, and many others (a full discussion for each renderer is beyond the scope of this paper). However, this still starkly contrasts with today's state of the art (pure data parallelism): the island-parallel configuration is always significantly faster than the purely data parallel configuration, often by several multiples.

9 Discussion

Concepts like our island parallelism, though compellingly simple and scalable from medium-sized to large data sets have not seen much attention from the scientific visualization community in recent years—nor is this concept implemented by major visualization packages. We have shown that the paradigm behind this is all the more relevant, the less scalable the underlying renderer is. For example, an idealized renderer like that in Fig. 7 with $\log(n)$ work complexity in the number of primitives will only see a performance improvement of $\log(n/2)$ from doubling the number ranks without adding more islands. A theoretical *rasterizer* whose computational complexity is linear in the number of primitives would instead see a performance improvement proportional to $n/2$.

We expect this effect to become more pronounced in the future, as we assume that $\log(n)$ work complexity for a parallel ray tracer is still overly enthusiastic. We have shown that in reality, path tracing or ray forwarding renderers exhibit much worse scalability. We expect these effects to become more critical due to the gap between compute and memory performance. The larger our data sets get, the more critical become optimizations comprising adaptive mesh refinement, space skipping, acceleration structures, or early ray termination. A typical slicing volume renderer using 3D textures and rasterization hardware, for example, cannot easily support early ray termination because the partially accumulated transmittance is not available in the fragment shader that samples the volume texture. For a renderer like that, the worst-case *and* average runtime would not only theoretically, but also in practice, be approximately the same, because no rays terminate early; a typical ray marcher, which is the de factor for absorption + emission nowadays, would still have the same *upper* bound proportional to the number of voxels, but in practice, many rays will *on average* terminate much earlier.

We thus expect typical modern renderers to exhibit average scalability in the number of primitives much worse than what was observed in the past with typical rasterizers. We expect this trend to continue in the future. All the more important become concepts that allow for better scalability beyond the number of primitives adopted by the scientific visualization and rendering communities. We see our paper as a first, overdue step in this direction.



(a) **Renderer 1** (structured) DNS View 1 (small)

(b) **Renderer 1** (structured) DNS View 2 (small)

(c) **Renderer 1** (structured) DNS View 1 (huge)

(d) **Renderer 1** (structured) DNS View 2 (huge)

(e) **Renderer 2** (path tracer) PBRT Landscape

(f) **Renderer 3** (Woodcock) Chameleon

(g) **Renderer 3** (Woodcock) DNS View 1 (small)

(h) **Renderer 3** (Woodcock) DNS View 2 (small)

(i) **Renderer 3** (Woodcock) DNS View 1 (huge)

(j) **Renderer 3** (Woodcock) DNS View 2 (huge)

(k) **Renderer 4** (element marcher) Small Lander

(l) **Renderer 4** (element marcher) Huge Lander

Figure 8: Scalability of the four renderers when increasing the number of GPUs and/or number of islands. (a-d) Renderer 1: sort-last structured volume ray marcher from Section 4. (e) Renderer 2: hardware-accelerated production path tracer from Section 5. (f-j) Renderer 3: volume path tracer from Section 6. (k-l) Renderer 4: element-marcher with sort-last deep compositor from Section 7.

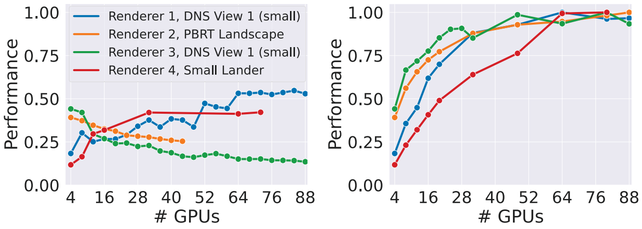


Figure 9: Comparison of purely data parallel rendering vs “ideal” islands-parallelism, across different renderers and renderer configurations, over increasing number of compute resources. Left: pure data parallelism over R ranks (normalized to the performance of the smallest number of ranks that can render the given model). Right: Performance for the same number of ranks, always split into as many islands as is possible for the given size of model (normalized by the same factor). Even perfect island-parallelism does not always scale perfectly, but always significantly better than pure data parallelism.

10 Conclusion

We have presented *island parallelism*, a concept to scale rendering workloads beyond the typical mere data- or image-parallel paradigms omnipresent in the scientific visualization community. Distributing the workload in a hybrid fashion among compute nodes by adding data replication as an additional degree of freedom has recently seen little to no attention in the community. At the same time, returns from data-parallel scaling diminish—are potentially even negative—in the presence of modern-day ray-tracing renderers. Hence, the pressure imposed by scalability issues becoming more pronounced is a trend that will continue in the foreseeable future. At the same time, mere image parallelism does not allow for data distribution and only scales for small or medium-sized data. We have thoroughly evaluated island parallelism in the context of several MPI-based GPU ray tracing renderers that are representative of what is nowadays used by the scientific and rendering communities. Our results confirm our hypothesis that scalability in the number of primitives is diminishing and hybrid data/image parallelism with data replication allows scaling far beyond what is possible with either of the two parallelization paradigms when used in isolation. We argue that when more scientific visualization pipelines switch to ray tracing-based rendering, due to the different scalability and communication patterns, concepts like our islands will become more relevant in the future.

Acknowledgments

We acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—grant no. 456842964. We also thank the Ministry of Culture and Science of the State of North Rhine-Westphalia for supporting the work through the PROFILBILDUNG grant PROFILNRW-2020-038C. Access to the Frontera supercomputer, as well as technical support, was graciously granted by Paul Navrátil and his team at the Texas Advanced Computing Center (TACC). The Mars Lander data set is courtesy Pat Moran with NASA. We finally thank Steven G. Parker for contributing ideas and for helpful discussion.

References

[1] G. Abram, P. Navrátil, P. Grossett, D. Rogers, and J. Ahrens. Galaxy: Asynchronous ray tracing for large high-fidelity visualization. In *Proceedings of the IEEE 8th Symposium on Large Data Analysis and Visualization*, LDAV '18, 2018.

[2] J. Ahrens, B. Geveci, and C. Law. ParaView: An end-user tool for large-data visualization. In C. D. Hansen and C. R. Johnson, eds., *Visualization Handbook*. 2005.

[3] E. Bethel, G. Humphreys, B. Paul, and J. Brederson. Sort-first, distributed memory parallel visualization and rendering. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, 2003.

[4] B. B. Cambazoglu and C. Aykanat. Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. *IEEE Transactions on Parallel and Distributed Systems*, 18(1), 2007.

[5] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. 2012.

[6] D. E. DeMarle and A. C. Bauer. In situ visualization with temporal caching. *Computing in Science Engineering*, 23(3), 2021.

[7] S. Eilemann, M. Makhinya, and R. Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE Transactions on Visualization and Computer Graphics*, 15(3), 2009.

[8] S. Eilemann and R. Pajarola. Direct send compositing for parallel sort-last rendering. In J. M. Favre, L. P. Santos, and D. Reiners, eds., *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '07, 2007.

[9] A. V. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. Hansen. TOD-Tree: Task-Overlapped Direct send tree image compositing for hybrid MPI parallelism and GPUs. *IEEE Transactions on Visualization and Computer Graphics*, 23(6), 2017.

[10] T. Günther, A. Kuhn, and H. Theisel. MCFTLE: Monte Carlo rendering of finite-time Lyapunov exponent fields. *Computer Graphics Forum*, 35(3), 2016.

[11] T. Ize, C. Brownlee, and C. D. Hansen. Real-time ray tracer for visualizing massive models on a cluster. In T. W. Kuhlen, R. Pajarola, and K. Zhou, eds., *Proceedings of the 11th Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '11. Eurographics Association, 2011.

[12] M. Jaroš, L. Říha, P. Strakoš, and M. Špeťko. GPU accelerated path tracing of massive scenes. *ACM Trans. Graph.*, 40(2), 2021.

[13] A. Knoll, G. P. Johnson, and J. Meng. Path tracing RBF particle volumes. In A. Marrs, P. Shirley, and I. Wald, eds., *Ray Tracing Gems II: Next Generation Real-Time Rendering with DXR, Vulkan, and OptiX*. 2021.

[14] M. Larsen, K. Moreland, C. R. Johnson, and H. Childs. Optimizing multi-image sort-last parallel rendering. In *Proceedings of the IEEE 6th Symposium on Large Data Analysis and Visualization*, LDAV '16, 2016.

[15] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *Journal of Fluid Mechanics*, 11(4), 2015.

[16] K.-L. Ma. Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '95, 1995.

[17] K.-L. Ma, J. Painter, C. Hansen, and M. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4), 1994.

[18] NVIDIA. CUDA Toolkit. Available at `https://developer.nvidia.com/cuda-toolkit`, Accessed: 27 March 2022.

[19] NVIDIA. NVIDIA OptiX Ray Tracing Engine. Available at `https://developer.nvidia.com/optix`, Accessed: 27 March 2022.

[20] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4), 1994.

[21] B. Moloney, M. Ament, D. Weiskopf, and T. Moller. Sort-first parallel volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 17(8), 2011.

[22] K. Moreland. IceT users' guide and reference. Technical report, Sandia National Laboratories, 2011.

[23] K. Moreland, W. Kendall, T. Peterka, and J. Huang. An image compositing solution at scale. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[24] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2001.

[25] P. A. Navrátil, D. S. Fussell, C. Lin, and H. Childs. Dynamic scheduling for large-scale distributed-memory ray tracing. In H. Childs, T. Kuhlen, and F. Marton, eds., *Eurographics Symposium on Parallel Graphics and Visualization*, 2012.

[26] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09. Association for Computing Machinery, New York, NY, USA, 2009.

[27] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham. End-to-end study of parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of the International Conference on Parallel Processing*, 2009.

[28] M. Pharr, W. Jakob, and G. Humphreys. Scenes for pbrt-v3. `https://pbrt.org/scenes-v3`, Accessed: 22 June 2022.

[29] E. Reinhard, A. Chalmers, and F. Jansen. Hybrid scheduling for parallel rendering using coherent ray tasks. In *Proceedings 1999 IEEE Parallel Visualization and Graphics Symposium*, pp. 21–28, 1999.

[30] A. Sahistan, S. Demirci, N. Morrical, S. Zellmann, A. Aman, I. Wald, and U. Güdükbay. Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. In *Proceedings of the IEEE Visualization Conference-Short Papers*, VIS '21, 2021.

[31] R. Samanta, T. Funkhouser, and K. Li. Parallel rendering with k-way replication. In *Proceedings IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, 2001.

[32] TACC. Frontera System Architecture - GPU Nodes. `https://frontera-portal.tacc.utexas.edu/user-guide/system/#gpu-nodes`, Accessed: 23 June 2022.

[33] W. Usher, I. Wald, J. Amstutz, J. Günther, C. Brownlee, and V. Pascucci. Scalable ray tracing using the distributed framebuffer. *Computer Graphics Forum*, 38(3), 2019.

[34] I. Wald, N. Morrical, and S. Zellmann. A memory efficient encoding for ray tracing large unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 28(1):583–592, 2022.

[35] I. Wald and S. G. Parker. Data parallel path tracing with object hierarchies. In *Proceedings of High Performance Graphics*, HPG '22, 2022. (to appear, preprint available under arXiv:2204.10170).

[36] I. Wald, S. Zellmann, and N. Morrical. Faster RTX-accelerated empty space skipping using triangulated active region boundary geometry. In M. Larsen and F. Sadlo, eds., *Eurographics Symposium on Parallel Graphics and Visualization*, 2021.

[37] E. R. Woodcock, T. Murphy, P. J. Hemmings, and T. C. Longworth. Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Proceedings of the Conference on Applications of Computing Methods to Reactor Problems*. Argonne National Laboratory, 1965.

[38] H. Yu, C. Wang, and K.-L. Ma. Massively parallel volume rendering using 2–3 swap image compositing. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '08, 2008.

[39] S. Zellmann, N. Morrical, I. Wald, and V. Pascucci. Finding efficient spatial distributions for massively instanced 3-D models. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '20, 2020.

[40] J. Zhang, X. Lu, C.-H. Chu, and D. K. Panda. C-GDR: High-performance container-aware GPUDirect MPI communication schemes on RDMA networks. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, IPDPS '19. IEEE, 2019.