

Scalable Ray Tracing Using the Distributed FrameBuffer

Will Usher^{†1,2}, Ingo Wald^{2,3}, Jefferson Amstutz², Johannes Günther², Carson Brownlee², and Valerio Pascucci¹

¹SCI Institute, University of Utah ²Intel Corporation ³Now at NVIDIA

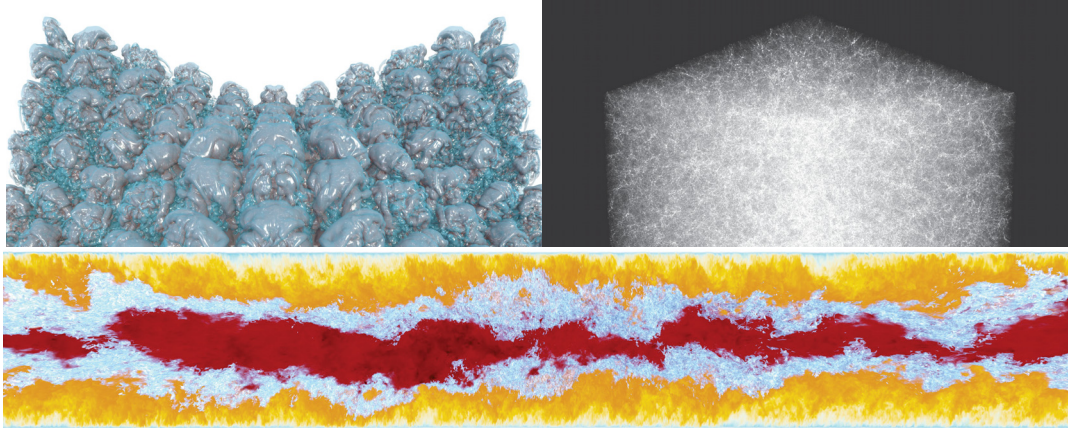


Figure 1: Large-scale interactive visualization using the Distributed FrameBuffer. Top left: Image-parallel rendering of two transparent isosurfaces from the Richtmyer-Meshkov [CDD*02] (516M triangles), 8FPS with a 2048² framebuffer using 16 Stampede2 Intel® Xeon® Platinum 8160 SKX nodes. Top right: Data-parallel rendering of the Cosmic Web [ISM*08] (29B transparent spheres), 2FPS at 2048² using 128 Theta Intel® Xeon Phi™ Knight's Landing (KNL) nodes. Bottom: Data-parallel rendering of the 951GB DNS volume [LM15] combined with a transparent isosurface (4.35B triangles), 5FPS at 4096 × 1024 using 64 Stampede2 Intel® Xeon Phi™ KNL nodes.

Abstract

Image- and data-parallel rendering across multiple nodes on high-performance computing systems is widely used in visualization to provide higher frame rates, support large data sets, and render data in situ. Specifically for in situ visualization, reducing bottlenecks incurred by the visualization and compositing is of key concern to reduce the overall simulation runtime. Moreover, prior algorithms have been designed to support either image- or data-parallel rendering and impose restrictions on the data distribution, requiring different implementations for each configuration. In this paper, we introduce the Distributed FrameBuffer, an asynchronous image-processing framework for multi-node rendering. We demonstrate that our approach achieves performance superior to the state of the art for common use cases, while providing the flexibility to support a wide range of parallel rendering algorithms and data distributions. By building on this framework, we extend the open-source ray tracing library OSPRay with a data-distributed API, enabling its use in data-distributed and in situ visualization applications.

CCS Concepts

• **Computing methodologies** → **Ray tracing**;

1. Introduction

The need for high-performance distributed parallel rendering is growing, spurred by trends in increasing data set sizes, the desire for higher fidelity and interactivity, and the need for in situ visualization. Meeting these demands poses new challenges to existing rendering methods, requiring scalability across a spectrum of memory and compute capacities on high-performance computing (HPC) resources. Whereas the growth in data set sizes demands a large amount of aggregate memory, the desire for more complex shading

and interactivity demands additional compute power. A large number of application needs fall somewhere in between these extremes, requiring a combination of additional memory and compute. Finally, in situ visualization requires the renderer to scale with the simulation, while incurring little overhead. Rendering techniques that scale well for either compute power or aggregate memory capacity are well known, but applications falling between these extremes have not been well addressed.

In large-scale rendering workloads on distributed-memory clusters, the data is typically partitioned into subregions and distributed across multiple nodes to utilize the aggregate memory available. Each node is then responsible for rendering its assigned subregion

[†] will@sci.utah.edu

of data. The partial images rendered by each node are then combined using a sort-last compositing algorithm, e.g., Parallel Direct Send [Hsu93], Binary Swap [MPHK94], Radix-k [PGR*09], or TOD-tree [GPC*17]. The IceT library [MKPH11] provides implementations of a number of sort-last compositing algorithms and is widely used in practice. However, such data-parallel renderers impose restrictions on how the data can be distributed, are susceptible to load imbalance, and are limited to local illumination effects.

At the other end of the spectrum, the master-worker architecture has been widely used to scale up compute capacity and provide interactivity for high-fidelity visualization of moderately sized data sets. Master-worker renderers distribute work image-parallel by assigning subregions of the image to be rendered by different nodes. This architecture has been used effectively in a number of ray tracers, e.g., Manta [BSP06], OpenRT [WBS02], and OSPRay [WJA*17]. While typically used for data which can be stored in memory on each node, this architecture can be used for large data sets by streaming data needed for the portion of the image from disk [WSB01] or over the network [DGBP05, IBH11]; however, these systems can suffer from cache thrashing and are tied to specific data types or formats.

Applications falling somewhere in between the extrema of only compute or memory scaling, or those seeking to go beyond common use cases, can quickly run into issues with existing approaches. For example, whereas a master-worker setup is well suited to image-parallel ray tracing, if the renderer wants to perform additional post-processing operations (e.g., tone-mapping, progressive refinement), or handle multiple display destinations (e.g., display walls), the master rank quickly becomes a bottleneck. Similarly, whereas existing sort-last compositing algorithms are well suited to statically partitioned data-parallel rendering, extending them to support partially replicated or more dynamic data distributions for better load balancing is challenging. Standard sort-last compositing methods operate bulk-synchronously on the entire frame, and are less suited to tile-based ray tracers in which small tiles are rendered independently in parallel.

In this paper, we describe the algorithms and software architecture—the “Distributed FrameBuffer”—that we developed to support distributed parallel rendering, with the goal of addressing the above issues to provide an efficient and highly adaptable framework suitable for a range of applications. The Distributed FrameBuffer (DFB) is built on a tile-based work distribution of the image processing tasks required to produce the final image from a distributed renderer. These tasks are constructed per-tile at runtime by the renderer and are not necessarily tied to the host application’s work or data distribution, providing the flexibility to implement a wide range of rendering algorithms and distribute compute-intensive image processing tasks. The DFB performs all communication and computation in parallel with the renderer using multiple threads to reduce compositing overhead. Although the DFB is flexible enough to support renderers across the spectrum of memory and compute scaling, it does not make a performance trade-off to do so. Our key contributions are:

- A flexible and scalable parallel framework to execute compositing and image processing tasks for distributed rendering;
- A set of parallel rendering algorithms built on this approach, covering both standard use cases and more complex configurations;

- An extension of OSPRay to implement a data-distributed API, allowing end users to leverage the above algorithms in practice on a wide range of different data types.

2. Previous Work

A large body of previous work has studied parallel rendering techniques for distributed-memory systems. These works can generally be classified as one of three techniques, first discussed in the context of rasterization by Molnar et al. [MCEF94]: sort-first, sort-middle, and sort-last. Sort-middle is tied to rasterization, thus we focus our discussion on sort-first and sort-last strategies. Sort-first is an image-parallel technique, where the workload is distributed across multiple ranks by subdividing the image. Sort-last is a data-parallel technique, where the workload is distributed by subdividing the 3D data, regardless of where it lies in the image. Hybrid approaches have also been proposed, which combine sort-first and sort-last techniques.

2.1. Data-Parallel Rendering

In sort-last, or data-parallel, rendering the geometric primitives and volumetric data are partitioned in 3D space, with each node assigned a subregion of the overall data to render. In early implementations, this subdivision was at the level of a single primitive [RK79]. Each node then renders its subregion of data to produce a partial image, which must then be combined with other nodes’ images to create the final image. Combining these partial images typically requires depth compositing the overlapping partial images to produce a correct final image. It is this second step that becomes the bottleneck at large core counts and high-resolutions, and therefore has been the focus of a large body of work (e.g., [Hsu93, MPHK94, FdR07, YWM08, KPH*10, MKPH11, GKH16, GPC*17]).

Most similar to our work in the context of data-parallel rendering is Grosset et al.’s [GKH16] Dynamically Scheduled Region-Based compositing (DSRB). DSRB divides the image into strips and constructs a per-strip blending order, referred to as a chain, based on which node’s data projects to each strip. Partial compositing for a chain can be done after receiving strips from successive nodes in the chain, overlapping compositing with rendering on other nodes. However, DSRB is restricted in the amount of rendering it can overlap with compositing, as each node renders its entire image before starting compositing; is only applicable to data-parallel rendering; and relies on a central scheduler to construct the chains.

The IceT library [MKPH11] encompasses several different compositing strategies for sort-last rendering and has been widely deployed across popular parallel scientific visualization tools. Thus, we use IceT as a primary point of comparison when evaluating our method’s performance. Although IceT was initially designed for rasterization, Brownlee et al. [BPL*12] used IceT’s depth compositing with a ray tracer inside of multiple visualization tools, though were hindered by the data distribution chosen by the tools. Wu et al. [WUP*18] employed a similar approach to integrate OSPRay into VisIt, using OSPRay to render locally on each rank and IceT to composite the image, and encountered similar difficulties.

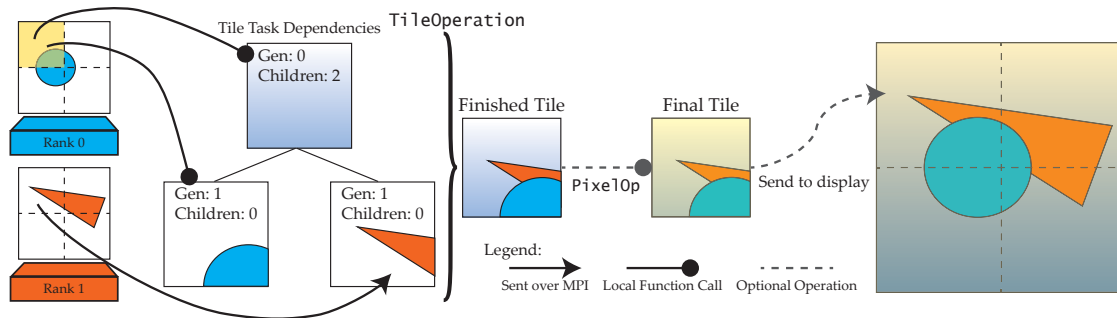


Figure 2: An example of the Distributed FrameBuffer's tile processing pipeline in a data-parallel renderer. Dependencies are specified on the fly per-tile and can be extended by child tiles. To compute the highlighted tile owned by rank 0, the owner sends a background color tile for generation 0, which specifies that two additional tiles will arrive in generation 1, potentially from different ranks. After receiving the full dependency tree, the tile operation produces the finished tile, which is tone-mapped by a pixel operation and sent to the display rank.

2.2. Image-Parallel Rendering

Image-parallel renderers assign subregions of the image to different ranks for rendering. To render large datasets, this approach is typically coupled with some form of data streaming or movement into a local cache, and is designed to exploit frame-to-frame coherence. The data movement work is amortized over multiple frames as the data rendered for a region of the image in one frame will likely be similar to that rendered in the next frame. Early rasterization-based techniques used a sort-middle algorithm, where the image was partitioned between nodes, and geometry sent to the node rendering the portion of the image it projected to [EGT90].

Image-parallel rendering lends itself well to ray tracing, as ray tracers already use acceleration structures for ray traversal which can be readily adapted to streaming and caching portions of the scene as they are traversed. Wald et al. [WSB01] used a commodity cluster for interactive ray tracing of large models, where a top-level k -d tree is replicated across the nodes and lower sub-trees fetched on demand from disk. DeMarle et al. [DGBP05] used an octree acceleration structure for rendering large volume data, where missing voxels would be fetched from other nodes using a distributed shared memory system. Ize et al. [IBH11] extended this approach to geometric data using a distributed BVH. When rendering fully replicated data, their approach avoids data movement and compositing, and can achieve 100FPS for primary visibility ray casting on 60 nodes. Biedert et al. [BMFG18] proposed an image-parallel remote streaming framework able to achieve over 80FPS from a distributed cluster to a remote client, using hardware acceleration and adaptive tile-based streaming.

2.3. Hybrid-Parallel Rendering

While image- and data-parallel rendering methods distribute work solely by partitioning the image or data, hybrid-parallel renderers combine both strategies, aiming to pick the best for the task at hand. Reinhard et al. [RCJ99] first proposed a hybrid scheduling algorithm for ray tracing distributed data, where the rays would be sent or the required data fetched depending on the coherence of the rays.

Samanta et al. [SFLS00] proposed to combine sort-first and sort-last rendering in the context of a rasterizer, by partitioning both the image and data among the nodes. Each node then renders its local data and sends rendered pixels to other nodes that own the tiles its

data touches. The tiles are then composited on each node and sent to the display node. This approach bears some resemblance to the Distributed FrameBuffer, although lacks its extensibility and support for ray tracing specific rendering effects.

Navrátil et al. [NFLC12] proposed a scheduler that combines static image and data decompositions for ray tracing, roughly similar to sort-first and sort-last, respectively. However, a key difference of their approach when compared to a sort-last rasterizer is that rays will be sent between nodes, similar to Reinhard et al. [RCJ99], to compute reflections and shadows. The static image decomposition scheduler works similar to the image-parallel algorithms discussed previously. Abram et al. [ANG*18] extended the domain decomposition model to an asynchronous, frameless renderer using a subtractive lighting model for progressive refinement. Park et al. [PFN18] extended both the image and domain decompositions, by introducing ray speculation to improve system utilization and overall performance. By moving both rays or data as needed, these approaches are able to compute global illumination effects on the distributed data, providing high-quality images at additional cost.

Biedert et al. [BWHG17] employed a task-based model of distributed rendering which is able to combine sort-first and sort-last rendering, by leveraging an existing runtime system to balance between these strategies. Although their work uses OSPRay for rendering, it is restricted to a single thread per-process and is non-interactive.

2.4. OSPRay, Embree and ISPC

Although the Distributed FrameBuffer is applicable to any tile-based rendering algorithm, we evaluate it within the context of the OSPRay ray tracing framework [WJA*17]. OSPRay provides a range of built in volume and geometric primitives used in scientific visualization, advanced shading effects, and achieves interactive rendering on typical workstations and laptops. To achieve interactive ray tracing performance on CPUs, OSPRay builds on top of Embree [WWB*14], the Intel SPMD Program Compiler (ISPC) [PM12], and Intel's Threading Building Blocks (TBB).

Embree is a high-performance kernel framework for CPU ray tracing, and provides a set of low-level kernels for building and traversing ray tracing data structures which are highly optimized for modern CPU architectures. ISPC is a single program multiple data

(SPMD) compiler, which vectorizes a scalar program by mapping different instances of the program to the CPU's vector lanes, thereby executing them in parallel. TBB provides a set of parallel programming primitives for writing high-performance multi-threaded code, similar to OpenMP.

3. The Distributed FrameBuffer

At its core, the Distributed FrameBuffer (DFB) is not a specific compositing algorithm per se, but a general framework for distributed rendering applications. A renderer using the DFB specifies a set of tasks to be executed on the rendered image and per-tile dependency trees for the tasks. The tasks are parallelized over the image by subdividing it into tiles, where each tile is owned by a unique rank—the tile owner—responsible for executing tasks for that tile. If task dependencies are produced on ranks other than the tile owner the DFB will route them over the network to the owner. The tile dependency trees are specified per-tile and per-frame, allowing for view- and frame-dependent behavior.

The tile processing pipeline involves three stages (Figure 2). First, the dependency tree is constructed by the tile operation as task dependencies are received from other ranks. Once the entire tree has been received the finished tile is computed by the tile operation and passed on to any pixel operations. The final output tile is then converted to the display image format and sent to the display rank, if needed. The processing pipeline and messaging system run asynchronously on multiple threads, allowing users to overlap additional computation with that performed by the DFB. Although the focus of this paper is on using the DFB for rendering, the task input tiles are not required to be produced by a renderer.

3.1. Tile Processing Pipeline

The DFB begins and ends processing synchronously, allowing applications processing multiple frames, i.e., a renderer, to ensure that tiles for different frames are processed in the right order. Before beginning a frame, the renderer specifies the tile operation to process the tiles it will produce. Each rank then renders some set of tiles based on the work distribution chosen by the renderer. As tiles are finished, they are handed to the DFB for processing by calling `setTile`. During the frame, the DFB will compute tile operations for the tiles owned by each rank in the background and send other tiles over the network to their owner. The frame is completed on each rank when the tiles it owns are finalized, and rendering is finished when all processes have completed the frame. As each tile is processed independently in parallel it is possible for some tiles to be finalized while others have yet to receive their first inputs.

To track the distributed tile ownership, the DFB instance on each rank stores a tile descriptor (Listing 1) for each tile in the image. When `setTile` is called the DFB looks up the descriptor for the tile and sends it to the owner using an asynchronous messaging layer (Section 3.2). If the owner is the calling rank itself, the tile is instead scheduled for processing locally.

For each tile owned by the rank, the DFB stores a concrete tile operation instance in the array of descriptors. The base structure for tile operations (Listing 1) stores a pointer to the local DFB instance

```
struct Tile {
    int generation;
    int children;
    region2i screenRegion;
    int accumulationID; // Sample pass for progressive refinement
    float color[4*TILE_SIZE*TILE_SIZE];
    float depth[TILE_SIZE*TILE_SIZE];
    float normal[3*TILE_SIZE*TILE_SIZE]; // Optional
    float albedo[3*TILE_SIZE*TILE_SIZE]; // Optional
};

struct TileDescriptor {
    virtual bool mine() { return false; }
    vec2i coords;
    size_t tileID, ownerRank;
};

struct TileOperation : TileDescriptor {
    bool mine() { return true; }
    virtual void newFrame() = 0;
    virtual void process(const Tile &tile) = 0;

    DistributedFrameBuffer *dfb;
    vec4f finalPixels[TILE_SIZE*TILE_SIZE];
    Tile finished, accumulation, variance;
};
```

Listing 1: The base structures for tiles and tile operations.

and a Tile buffer to write the finished tile data to, along with optional accumulation and variance buffer tiles. The `finalPixels` buffer is used as scratch space to write the final tile to, before sending it to the display rank.

To implement the corresponding tile operation for a rendering algorithm (e.g., sort-last compositing) users extend the `TileOperation`, and specify their struct to be used by the DFB. Each time a tile is received by the DFB instance on the tile owner, the `process` function is called on the tile operation to execute the task. The `newFrame` function is called when a new frame begins, to reset any per-frame state.

When all a tile's dependencies have been received the tile operation combines the inputs to produce a finished tile, which is then passed to the DFB. The local DFB instance runs any additional pixel operations on the finished tile and converts the final pixels to the display color format, outputting them to the `finalPixels` buffer. This buffer is then compressed and sent to the display rank. In addition to the `RGBA8` and `RGBAF32` display formats, the DFB also offers a `NONE` format, which is unique in that it indicates that the display rank should not receive or store the final pixels at all. We will discuss a useful application of the `NONE` format in Section 4.4.

3.1.1. Per-Tile Task Dependency Trees

The `Tile` structure passed to `setTile` and routed over the network is shown in Listing 1. To construct the dependency tree, each rendered tile specifies itself as a member of some generation (a level in the tree), and as having some number of children in the following generation. The total number of tiles to expect in the next generation is the sum of all children specified in the previous one. Different ranks can contribute tiles with varying numbers of children for each generation, and can send child tiles for parents rendered by other ranks. There is no requirement that tiles are sent in order by generation, nor is a tile operation guaranteed to receive tiles in a fixed order. Tile operations with dependencies beyond a trivial single tile can be implemented by buffering received tiles in `process` to collect the complete dependency tree.

The interpretation and processing order of the dependency tree is left entirely to the tile operation. For example, the dependency tree could be used to represent a compositing tree, input to some

filtering, or simply a set of pixels to average together. The creation of the dependency trees by the renderer and their processing by the tile operation are tightly coupled, and thus the two are seen together as a single distributed rendering algorithm. The flexibility of the tile operation and dependency trees allows the DFB to be used in a wide range of rendering applications (Section 4).

3.1.2. Pixel Operations

Additional post-processing, such as tone-mapping, can be performed by implementing a pixel operation (`PixelOp`). The pixel operation takes the single finished tile from the tile operation as input, and thus is not tied to the tile operation or renderer. The DFB runs the pixel operation on the tile owner after the tile operation is completed to distribute the work. In addition to image post-processing, pixel operations can be used, for example, to re-route tiles to a display wall (Section 4.4).

3.2. Asynchronous Messaging Layer

To overlap communication between nodes with computation, we use an asynchronous point-to-point messaging layer built on top of MPI (Message Passing Interface). Objects that will send and receive messages register themselves with the messaging layer and specify a unique global identifier. Each registered object is seen as a global “distributed object”, with an instance of the object on each rank which can be looked up by its global identifier. A message can be sent to the instance of an object on some rank by sending a message to the rank with the receiver set as the object’s identifier.

The messaging layer runs on two threads: a thread which manages sending and receiving messages with MPI, and an inbox thread which takes received messages and passes them to the receiving object. Messages are sent by pushing them on to an outbox, which is consumed by the MPI thread. To avoid deadlock between ranks, we use non-blocking MPI calls to send, receive, probe, and test for message completion. Received messages are pushed on to an inbox, which is consumed by the inbox thread. To hand a received message to the receiving object, the inbox thread looks up the receiver by its global ID in a hash table. Messages are compressed using Google’s Snappy library [Goo] before enqueueing them to the outbox and decompressed on the inbox thread before being passed to the receiver.

In our initial implementation we also used the messaging layer to gather the final tiles to the display rank. However, unless the rendering workload is highly imbalanced, this approach generates a large burst of messages to the display, with little remaining rendering work to overlap with. This burst of messages also appeared to trigger an MPI performance issue on some implementations. As an optimization, the final tiles are instead written to a buffer, which is compressed and gathered to the display with a single `MPI_Gather` at the end of the frame.

4. Rendering with the Distributed FrameBuffer

A distributed rendering algorithm using the DFB consists of a renderer, responsible for rendering tiles of the image, coupled with a

```
struct ImageParallel : TileOperation {
    void process(const Tile &tile) {
        // Omitted: copy data from the tile
        dfb->tileIsCompleted(this);
    }
};

void renderFrame(DFB *dfb) {
    dfb->begin();
    parallel_for (Tile &t : assignedTiles()) {
        renderTile(t);
        dfb->setTile(t);
    }
    dfb->end();
}
```

Listing 2: The tile operation and rendering loop for an image-parallel renderer using the DFB.

tile operation, which will combine the results of each ranks’ renderer. In the following sections we discuss a few distributed rendering algorithms built on the DFB, covering standard image-parallel (Section 4.1) and data-parallel (Section 4.2) rendering, along with extensions to these methods enabled by the DFB, specifically, dynamic load balancing (Section 4.1.1) and mixed-parallel rendering (Section 4.3). Finally, we discuss how pixel operations can be used to implement a high-performance display wall system (Section 4.4).

4.1. Image-Parallel Rendering

An image-parallel renderer distributes the tile rendering work in some manner between the ranks such that each tile is rendered once. This distribution can be a simple linear assignment, round-robin, or based on some runtime load balancing. The corresponding tile operation expects a single rendered tile as input. The DFB allows for a straightforward and elegant implementation of this renderer (Listing 2).

4.1.1. Tile Ownership vs. Work Distribution

The work distribution chosen by the renderer is not tied to the DFB tile ownership, allowing the renderer to distribute work as desired. Though it is preferable that the tile owners render the tiles they own to reduce network traffic, this is not a requirement.

This flexibility in work distribution can be used, for example, to implement dynamic load balancing. We extend the `ImageParallel` tile operation to support receiving a varying number of tiles, and the renderer to assign each tile to multiple ranks. Each redundantly assigned tile uses a different random seed to generate camera rays, thereby computing a distinct set of samples. The rendered tiles are then averaged together by the tile operation, producing a finished tile equivalent to a higher sampling rate. This approach is especially useful for path tracing, as a high number of samples are required to produce a noise-free image. Tiles with higher variance can be assigned to additional ranks, adjusting the sampling rate dynamically.

4.2. Data-Parallel Rendering

A standard sort-last data-parallel renderer decomposes the scene into a set of bricks, and assigns one brick per-rank for rendering. Each rank renders its local data to produce a partial image, which are combined using a sort-last compositing algorithm to produce an image of the entire dataset. To implement a data-parallel renderer using the DFB, we express sort-last compositing as a tile operation, and take

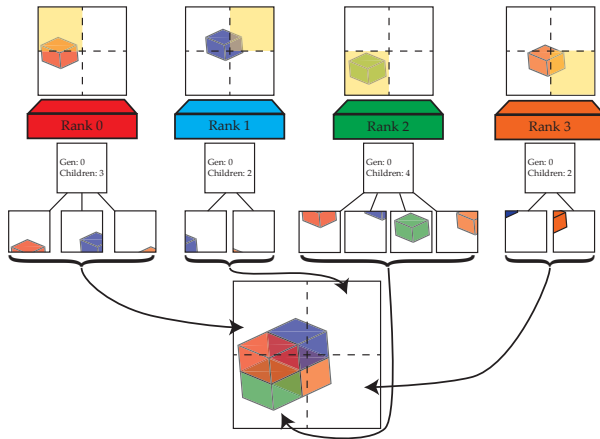


Figure 3: Tile ownership and dependency trees for a data-parallel renderer using the DFB. Each rank owns its highlighted tile, and receives input tiles from ranks whose data projects to the tile. Compositing runs in parallel to local rendering, reducing overhead.

```

1 void renderFrame(Brick local, box3f allBounds[], DFB *dfb) {
2     dfb->begin();
3     /* We touch the tiles we own and those touched by the
4        screen-space projection of our brick */
5     Tile tiles[] = {dfb->ownedTiles(), dfb->touchedTiles(local)};
6     parallel_for (Tile &t : tiles) {
7         bool intersected[] = intersectBounds(allBounds, t);
8         if (dfb->tileOwner(t)) {
9             fillBackground(t);
10            t.generation = 0;
11            t.children = numIntersected(intersected);
12            dfb->setTile(t);
13        }
14        if (intersected[local]) {
15            renderBrickForTile(t, local);
16            t.generation = 1;
17            t.children = 0;
18            dfb->setTile(t);
19        }
20    }
21    dfb->end();
22 }

```

Listing 3: The rendering loop for a standard data-parallel renderer.

advantage of the DFB's asynchronous tile routing and processing to execute the compositing in parallel with local rendering. The benefits of this approach are two-fold: the per-tile task dependencies allow to minimize compositing and communication work per-tile, and overlapping compositing and rendering reduces the additional time spent compositing after rendering is finished.

To compute a per-tile compositing dependency tree, each rank collects the bounds of the other ranks' data and projects them to the image (Figure 3). Only those ranks whose data projects to some tile will render inputs for it. Each rank is responsible for specifying the dependency information for the tiles it owns (highlighted in yellow, Figure 3). The tile owner will compute an additional "background" tile and set it as the sole member of generation 0. The background tile is filled with the background color or texture, and sets the number of ranks whose data project to the tile as the number of children.

The renderer (Listing 3) begins by determining the set of candidate tiles that it must either send a background tile for or render data to. The candidate tiles that the rank's local data may project to are found using a conservative screen-space AABB test, which is subsequently refined. For each candidate tile, the renderer computes an exact list of the ranks whose data touches the tile by ray tracing

```

struct AlphaBlend : TileOperation {
    Tile bufferedTiles[];
    int currentGen, missing, nextChildren;
    void newFrame() {
        currentGen = 0;
        missing = 1; // Expect a generation 0 tile to start
        nextChildren = 0;
    }
    void process(const Tile &tile) {
        bufferedTiles.append(tile);
        if (tile.generation == currentGen) {
            --missing;
            nextChildren += tile.children;
            checkTreeComplete();
        }
        if (!missing) {
            sortAndBlend(bufferedTiles);
            dfb->tileIsCompleted(this);
            bufferedTiles = {};
        }
    }
    // Check receipt of all children from all generations,
    // advancing currentGen as we complete generations.
    void checkTreeComplete() { /* omitted for brevity */ }
}

```

Listing 4: The sort-last compositing tile operation used by the data- and mixed-parallel renderers. It first collects the dependency tree, then sorts and blends the pixels to produce the composited tile.

the bounding boxes. The number of intersected boxes is the number of generation 1 tiles to expect as input to the tree. If the rank's local data was intersected, it renders its data and sends a generation 1 tile. To allow for ghost zones and voxels, camera rays are clipped to the local bounds of the rank's data. As with the outer candidate tile loop, the inner rendering loop is parallelized over the pixels in a tile.

After receiving the entire dependency tree, the AlphaBlend tile operation (Listing 4) sorts the pixels by depth and blends them together to composite the tile. The tile fragment sorting is done per-pixel, in contrast to the per-rank sort used in standard approaches. Sorting per-pixel allows for rendering effects like depth of field, side-by-side stereo, and dome projections. As the tile processing is done in parallel, we do not find the sorting to be a bottleneck. In the case that a rank-order sort would produce a correct image, the dependency tree can be constructed as a list instead of a single-level tree with tiles ordered back-to-front by generation. Finally, although we have discussed the data-parallel renderer with a single brick of data per-rank, it trivially supports multiple bricks per-rank, allowing for finer-grained work distributions.

4.3. Rendering Hybrid Data Distributions

A data-parallel renderer that statically assigns each brick of data to a single rank is susceptible to load imbalance, coming from factors such as the data distribution, transfer function, or camera position. To better distribute the workload, we can assign the same brick of data to multiple ranks, with each rank potentially assigned multiple bricks. Each rank is responsible for rendering a subset of the tiles the bricks it has projects to, thereby dividing the rendering workload for each brick among the ranks. Although this increases the memory requirements of the renderer, additional memory is often available given the number of compute nodes used to achieve an interactive frame rate.

Rendering such a configuration with a standard compositing approach is either difficult or not possible, as the compositing tree and blending order is set for the entire framebuffer by sorting the ranks [MKPH11]. However, the DFB's per-tile dependency trees

```

void renderFrame(Brick local[], box3f allBounds[], DFB *dfb) {
    dfb->begin();
    Tile tiles[] = {dfb->ownedTiles(), dfb->touchedTiles(local)};
    parallel_for (Tile &t : tiles) {
        bool intersected[] = intersectBounds(allBounds, t);
        if (dfb->tileOwner(t)) {
            // Listing 3, lines 9-12
        }
        parallel_for (Brick &b : local) {
            if (tileBrickOwner(b, t) && intersected[b]) {
                // Listing 3, lines 15-18
            }
        }
    }
    dfb->end();
}

```

Listing 5: The rendering loop of the mixed-parallel renderer. The DFB allows for an elegant extension of the data-parallel renderer to support partially replicated data for better load-balancing.

allow renderers to change which ranks contribute tiles for each image tile. This enables a direct extension of the data-parallel renderer discussed previously into a mixed-parallel renderer, which balances image and data parallelism to achieve better load balance.

To develop the mixed-parallel extension, we introduce the concept of a “tile-brick owner”. Given a dataset partitioned into a set of bricks and distributed among the ranks with some level of replication, the renderer must select a unique rank among those sharing a brick to render it for each image tile. The rank chosen to render the brick for the tile is referred to as the “tile-brick owner”. Thus we can take our data-parallel renderer and modify it so that a rank will render a brick for a tile if the brick projects to the tile and the rank is the tile-brick owner (Listing 5). The task dependency tree and tile operation are the same as the data-parallel renderer; the only difference is which rank renders the generation 1 tile for a given brick and image tile.

Our current renderer uses a round-robin assignment to select tile-brick ownership, however this is not a requirement of the DFB. A more advanced renderer could assign tile-brick ownership based on some load-balancing strategy (e.g., [FE11]), or adapt the brick assignment based on load imbalance measured in the previous frame. The strategies discussed for image-parallel load balancing and work subdivision in Section 4.1.1 are also applicable to the mixed-parallel renderer. For example, two ranks sharing a brick could each compute half of the camera rays per-pixel, and average them together in the tile operation to produce a higher quality image.

The mixed-parallel renderer supports the entire spectrum of image- and data-parallel rendering: given a single brick per-rank it is equivalent to the data-parallel renderer; given the same data on all ranks it is equivalent to the image-parallel renderer; given a partially replicated set of data, or a mix of fully replicated and distributed data, it falls in between.

4.4. Display Walls

The DFB can also be used to implement a high-performance display wall rendering system by using a pixel operation to send tiles directly to the displays (Figure 4). Tiles will be sent in parallel as they are finished on the tile owner directly to the displays, achieving good utilization of a fully interconnected network. Moreover, when rendering with the NONE image format, the image will not be gathered to the master rank, avoiding a large amount of network

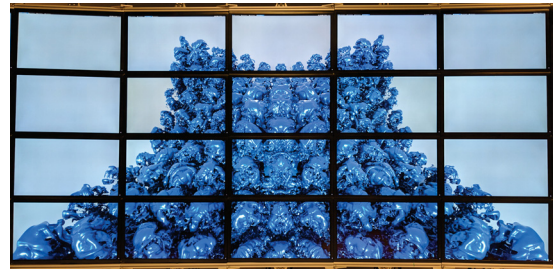


Figure 4: A prototype display wall system using DFB pixel operations to send tiles in parallel from an image-parallel path tracer.

communication and a common bottleneck. As pixel operations are not tied to the rendering algorithm or tile operation, this method can be used to drive a display wall with any of the presented renderers.

4.5. Implementation

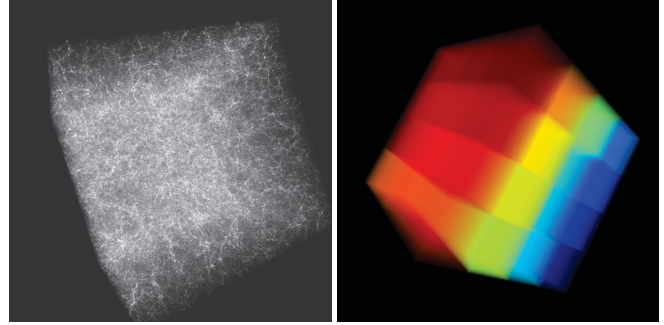
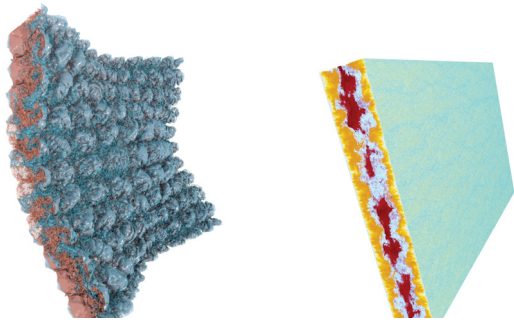
We implement the Distributed FrameBuffer and the presented rendering algorithms in OSPRay’s MPI module, using Intel TBB for multi-threading and ISPC [PM12] for vectorization. The underlying implementation of the `MPIDevice` provided by OSPRay [WJA*17] for image-parallel rendering has been significantly improved by this work, although it is exposed to users in the same manner as before. Users can continue to run existing OSPRay applications with `mpirun` and pass the `--osp:mpi` argument to the application, and OSPRay will replicate the scene data across a cluster and render it image-parallel using the rendering algorithms described in Sections 4.1 and 4.1.1.

5. A Data-Distributed API for OSPRay

The OSPRay API was originally designed for a single application process passing its data to OSPRay. Although OSPRay may offload the data in some way to other ranks, this is done without the application’s awareness. This API works well for applications that do not need to specify the data distribution; however, it is not applicable to those that do, e.g., ParaView and VisIt. Maintaining an API that is familiar to users while extending it to a data-distributed scenario poses some challenges. Furthermore, we would like to seamlessly support existing OSPRay modules, which have added new geometries [WKJ*15, VSW*17, WWW*19] and volumes [RWCB15, WBUK17], in a data-distributed setting.

We implement the data-distributed API through the addition of a new OSPRay API backend, the `MPIDistributedDevice`. As in single process rendering, each rank sets up its local geometries and volumes independently and places them into one or more `OSPModel` objects. However, instead of a single model per-scene, the application must create one model for each disjoint brick of data on the rank. Each brick may contain any combination of geometries and volumes, including ones provided by user modules. To allow applications to pass OSPRay information about the data distribution, the distributed device extends the `OSPModel` with two additional parameters: a required integer ID, and an optional bounding box.

The ID is used to determine if two ranks have the same brick of data and can share the rendering work using the mixed-parallel



(a) R-M transparent isosurfaces. (b) DNS with transparent isosurfaces. (c) 5^3 Cosmic Web subset. (d) Synthetic benchmark volume.

Figure 5: The data sets used in our benchmarks. (a) Two transparent isosurfaces on the Richtmyer-Meshkov [CDD*02], 516M triangles total. (b) A combined visualization of the 451GB single-precision DNS [LM15] with two transparent isosurfaces, 5.43B triangles total. (c) A 5^3 subset of the 8^3 Cosmic Web [ISM*08], 7.08B particles rendered as transparent spheres. (d) The generated volume data set used in the compositing benchmarks, shown for 64 nodes. Each node has a single 64^3 brick of data.

renderer. A typical data-parallel application with a single model per-rank could simply use the MPI rank as the ID, while an application with a hybrid data distribution would have a list of models and assign a unique ID for each shared brick of data. An MPI-parallel application can even use the distributed API for image-parallel rendering by specifying the same data and ID on each rank.

The bounding box parameter can be used to override the model's computed bounds, if the model contains additional ghost geometries or voxels that should be hidden from camera rays. An additional set of ghost models can also be passed to the renderer, containing data visible only to secondary rays. The bounding box parameter and ghost models allow applications to support local shading effects such as ambient occlusion, or compute shadows and reflections on the replicated data in the scene.

6. Results

We evaluate the performance of the Distributed FrameBuffer on the rendering algorithms described in Section 4, using our implementations within OSPRay. The benchmarks are run on two HPC systems, the Texas Advanced Computing Center's *Stampede2*, and Argonne National Laboratory's *Theta*, on a range of typical image- and data-parallel rendering use cases (Figure 5). We also perform a direct comparison of our sort-last compositing implementation using the DFB against IceT for a typical data-parallel use case. To measure performance as the rendering workload varies, the benchmarks are taken while rendering a rotation around the data set. Unless otherwise stated, we plot the median performance for the benchmarks, with the median absolute deviation shown as error bars. These measures are more robust to outliers, giving some robustness against influence from other jobs on the system. All benchmarks are run with one MPI rank per-node, as OSPRay uses threads on a node for parallelism.

Stampede2 and *Theta* consist of 4200 and 4392 Intel® Xeon Phi™ KNL processors respectively. *Stampede2* uses the 7250 model, with 68 cores, while *Theta* uses the 7230 model with 64 cores. *Stampede2* contains an additional partition of 1736 dual-socket Intel® Xeon Phi™ Platinum 8160 SKX nodes. Although the KNL nodes of both machines are similar, the network interconnects differ significantly, which can effect the performance of communication

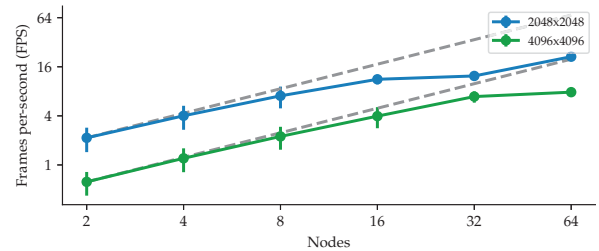


Figure 6: Image-parallel strong-scaling on the R-M transparent isosurfaces data set on Stampede2 SKX nodes. The image-parallel renderer using the DFB scales to provide interactive rendering of expensive, high-resolution scenes.

in the DFB. *Stampede2* employs an Intel Omni-Path network in a fat-tree topology, while *Theta* uses a Cray Aries network with a three-level Dragonfly topology.

6.1. Image-Parallel Rendering Performance

To study the scalability of the DFB and the image-parallel rendering algorithm described in Section 4.1, we perform a strong scaling benchmark using OSPRay's scientific visualization renderer. We use VTK to extract two isosurfaces from the Richtmyer-Meshkov volume, which are rendered with transparency and ambient occlusion (Figure 5a). We measure strong-scaling on *Stampede2* SKX nodes at two image resolutions (Figure 6). Although the renderer begins to drop off from the ideal scaling trend as the local work per-node decreases, this could potentially be addressed by employing the work-subdivision and load-balancing strategies discussed in Section 4.1.1.

6.2. Data-Parallel Rendering Performance

To study the scalability of the DFB when applied to the standard data-parallel rendering algorithm in Section 4.2, we run strong scaling benchmarks with two large-scale data sets on *Stampede2* and *Theta*. On *Stampede2* we render a combined visualization of the DNS with transparent isosurfaces (Figure 5b), and on *Theta* we render the 5^3 Cosmic Web subset (Figure 5c). We find that our data-parallel renderer using the DFB is able to provide interactive frame rates

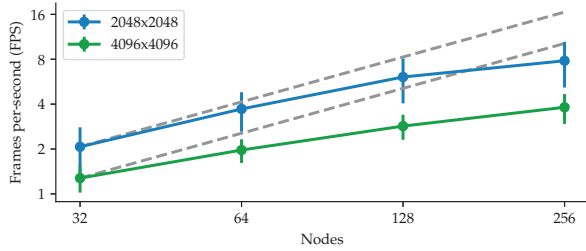
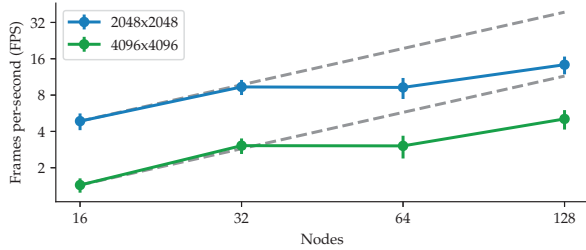
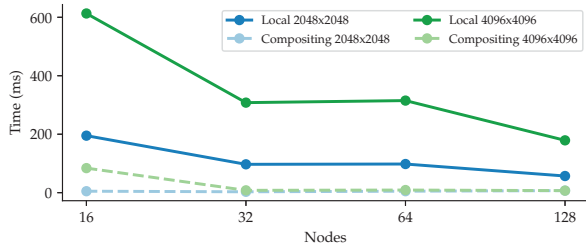


Figure 7: Data-parallel strong-scaling on the Cosmic Web data set on Theta. We find close to ideal scaling at moderate image sizes and node counts, with somewhat poorer scaling at very high resolutions.



(a) Overall rendering performance.



(b) Timing breakdown of local rendering and compositing overhead.

Figure 8: Data-parallel strong-scaling on the DNS with isosurfaces on Stampede2 KNLs. The lack of scaling from 32 to 64 nodes is attributable to a poor local work distribution (b), which can be partially addressed by using our mixed-parallel renderer.

for these challenging scenes, and scale up performance with more compute.

On the Cosmic Web we observe good scaling from 32 to 64 nodes (Figure 7). Although performance begins to trail off the ideal trend beyond 128 nodes, absolute rendering performance remains interactive.

On the DNS we find near ideal scaling from 16 to 32 nodes (Figure 8a); however, we observe little change from 32 to 64 nodes, although we see improvement again at 64 to 128 nodes. To find the cause of the bottleneck at 64 nodes, we look at a breakdown of the time spent rendering the rank's local data and the compositing overhead incurred by the DFB (Figure 8b). Compositing overhead refers to the additional time the compositor takes to complete the image, after the slowest local rendering task has completed [GKH16]. In this case we find that the bottleneck is caused by the local rendering task not scaling, which could be addressed by employing a hybrid data distribution or the work-splitting techniques discussed previously.

6.2.1. Compositing Performance Comparison with IceT

To perform a direct comparison with IceT for data-parallel rendering, we use a synthetic data set (Figure 5d), and modify our data-parallel renderer to support using IceT for compositing. The IceT renderer follows the same code-path as our data-parallel renderer to render its assigned brick of data, then hands the framebuffer off to IceT for compositing. We found IceT's automatic compositing algorithm selection to give the best performance, and use this mode throughout the benchmarks.

In terms of overall scalability and performance, our approach scales better than, or at least similar to, IceT, while achieving better absolute rendering performance (Figures 9a to 9c). When comparing timing breakdowns (Figures 9d to 9f) we find that, as expected, local rendering times are similar, and the performance difference is due to the differing compositing overhead. It is important to note that some of the absolute difference in overhead is due to IceT's synchronous design, which makes it unable to overlap compositing with rendering. We can consider a hypothetical IceT implementation which does overlap compositing and rendering by subtracting the local rendering time from the compositing overhead, and find that the DFB still achieves similar or superior compositing performance. Furthermore, we observe that when comparing the scaling trends of the two approaches, the DFB scales similar to, or better than, IceT. Although a rigorous comparison is difficult due to the different HPC systems used, the DFB follows similar scaling trends as Grosset et al.'s DSRB [GKH16], while providing greater flexibility.

Finally, we evaluate the portability of our approach by comparing the KNL runs on Stampede2 (Figures 9b and 9e) and Theta (Figures 9a and 9d). The slightly different KNLs on each system will have a minor effect on performance; however any significant differences are attributable to the differing network architectures and job placement strategies. On Stampede2 we observe a rather bumpy scaling trend where, depending on the image size, we see a temporary decrease in the compositing performance at certain node counts. On Theta we observe a smoother trend, with better absolute compositing performance. We found that disabling message compression on Theta gave better performance, while on Stampede2 we encountered MPI messaging performance issues at 16 nodes and up without it. Thus, we leave compression as an option to users which is enabled by default at 16 nodes. In our benchmarks we disable compression on Theta, and enable it at 16 nodes and up on Stampede2. IceT uses a custom image compression method, which is not easily disabled.

6.3. Hybrid Data Distribution Rendering Performance

To measure the impact of partial data replication on load balance, we look at the per-frame overall time on the DNS with isosurfaces data set on Stampede2 (Figure 10). The volume is partitioned into as many bricks as there are ranks, with bricks redundantly assigned to ranks based on the available memory capacity. When using 64 KNLs there is enough memory to store two bricks per-rank, with 128 KNLs we can store up to four. The rendering work for each brick will be distributed among two or four ranks, respectively. The redundant bricks are distributed using a simple round-robin assignment. A brick distribution based on, e.g., some space filling curve or runtime tuning, could provide additional improvement.

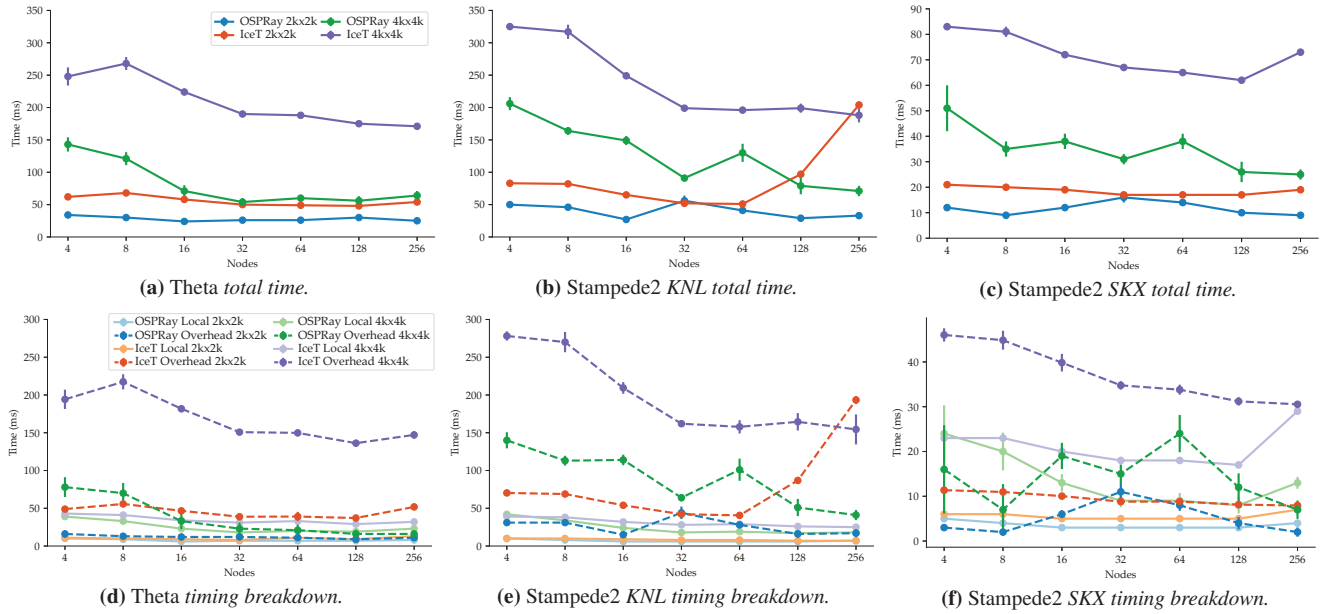


Figure 9: Compositing benchmark performance comparison of the DFB and IceT on the synthetic data set. We find that our approach achieves better, or at least similar, scaling as IceT, while providing faster absolute rendering times. In the timing breakdowns (d-f), we observe this difference is due to the DFB achieving a significant reduction in compositing overhead.

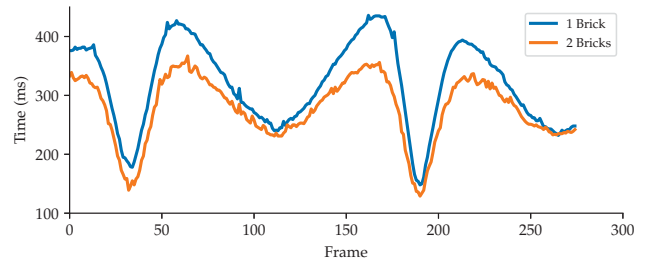
In both the 64 and 128 node runs the two brick per-node configuration provides a consistent improvement over no replication. This improvement is more pronounced for camera positions with greater load imbalance. With four bricks per-node, there are larger fluctuations in rendering performance, though at times we do find improvement over the two brick configuration. These larger fluctuations could be due to increased memory traffic, which is alleviated as data is cached in the KNL MCDRAM. This theory is further supported by the sharp spikes in performance, when new data must be fetched from RAM.

7. Conclusion

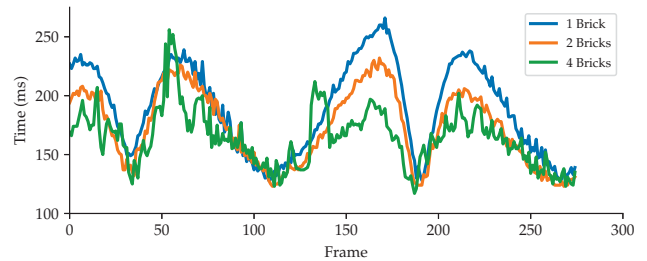
We have presented the Distributed FrameBuffer, an asynchronous, distributed image processing and compositing framework primarily targeted at rendering applications. By breaking the image processing operations into a set of per-tile tasks with independent dependency trees, the DFB simplifies the implementation of complex distributed rendering algorithms. Moreover, the DFB does not trade performance for this flexibility and we report performance competitive with specialized state-of-the-art algorithms. Our data-distributed API extension to OSPray has already been used successfully in practice for in situ visualization [URW*18].

We have merged our implementation of the DFB, the rendering algorithms presented, and the data-distributed API into OSPray, and released them in version 1.8. While prior work integrated OSPray into VisIt [WUP*18] by using OSPray's single-node rendering API and IceT for compositing, this can now be done using the distributed API directly. Compared to results reported on prior versions of OSPray [ANG*18] our work provides significant performance improvements.

However, the DFB and rendering algorithms presented are not



(a) Per-frame render time on 64 Stampede2 KNLs, at 4096×4096



(b) Per-frame render time on 128 Stampede2 KNLs, at 4096×4096

Figure 10: Improving load-balancing on the DNS with isosurfaces with partial data-replication in the mixed-parallel renderer. Sharing rendering between two nodes (two bricks per-node) gives a consistent improvement, between four tends to give further improvement.

without limitations. The rendering algorithms presented support only local lighting effects computed with the data available on a rank. Although approaches to compute global illumination on distributed data by sending rays between nodes [ANG*18, PFN18] could be implemented in the DFB, it is unclear how well a naive implementation would perform, or if extensions to the DFB would

be required. We leave this exciting avenue of research as future work.

In our evaluation we observed large differences in MPI performance and network behavior between *Stampede2* and *Theta*. Although we expose the use of compression as an option for users to tune as needed, it would be worthwhile to investigate self-tuning strategies for the DFB to automatically adapt to such architectural differences.

Acknowledgments

We would like to thank Damon McDougall and Paul Navrátil of the Texas Advanced Computing Center for assistance investigating MPI performance at TACC, and Mengjiao Han for help with the display wall example. The Cosmic Web and DNS datasets were made available by Paul Navrátil, the Richtmyer-Meshkov is courtesy of Lawrence Livermore National Laboratory. This work is supported in part by the Intel Parallel Computing Centers Program, NSF: CGV Award: 1314896, NSF:IIP Award: 1602127, NSF:ACI Award: 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375 and NSF:OAC Award: 1842042. This work used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility supported under Contract DE-AC02-06CH11357. The authors acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing HPC resources that have contributed to the research results reported in this paper.

References

- [ANG*18] ABRAM G., NAVRÁTIL P., GROSSET A. V. P., ROGERS D., AHRENS J.: Galaxy: Asynchronous Ray Tracing for Large High-Fidelity Visualization. In *2018 IEEE Symposium on Large Data Analysis and Visualization* (2018). 3, 10
- [BMFG18] BIEDERT T., MESSMER P., FOGAL T., GARTH C.: Hardware-Accelerated Multi-Tile Streaming for Realtime Remote Visualization. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018). 3
- [BPL*12] BROWNLEE C., PATCHETT J., LO L.-T., DEMARLE D., MITCHELL C., AHRENS J., HANSEN C.: A Study of Ray Tracing Large-Scale Scientific Data in Parallel Visualization Applications. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012). 2
- [BSP06] BIGLER J., STEPHENS A., PARKER S. G.: Design for Parallel Interactive Ray Tracing Systems. In *2006 IEEE Symposium on Interactive Ray Tracing* (2006). 2
- [BWHG17] BIEDERT T., WERNER K., HENTSCHEL B., GARTH C.: A Task-Based Parallel Rendering Component For Large-Scale Visualization Applications. In *Eurographics Symposium on Parallel Graphics and Visualization* (2017). 3
- [CDD*02] COHEN R. H., DANNEVIK W. P., DIMITS A. M., ELIASON D. E., MIRIN A. A., ZHOU Y., PORTER D. H., WOODWARD P. R.: Three-dimensional simulation of a Richtmyer-Meshkov instability with a two-scale initial perturbation. *Physics of Fluids* (2002). 1, 8
- [DGBP05] DEMARLE D. E., GRIBBLE C. P., BOULOS S., PARKER S. G.: Memory sharing for interactive ray tracing on clusters. *Parallel Computing*, 2 (2005). 2, 3
- [EGT90] ELLSWORTH D., GOOD H., TEBBS B.: Distributing display lists on a multicomputer. *SIGGRAPH Comput. Graph.* (1990). 3
- [FdR07] FAVRE J. M., DOS SANTOS L. P., REINERS D.: Direct Send Compositing for Parallel Sort-Last Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2007). 2
- [FE11] FREY S., ERTL T.: Load balancing utilizing data redundancy in distributed volume rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011). 7
- [GKH16] GROSSET A. P., KNOLL A., HANSEN C.: Dynamically scheduled region-based image compositing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2016). 2, 9
- [Goo] GOOGLE: Snappy. <https://github.com/google/snappy>. 5
- [GPC*17] GROSSET A. V. P., PRASAD M., CHRISTENSEN C., KNOLL A., HANSEN C.: TOD-Tree: Task-Overlapped Direct Send Tree Image Compositing for Hybrid MPI Parallelism and GPUs. *IEEE Transactions on Visualization and Computer Graphics* (2017). 2
- [Hsu93] HSU W. M.: Segmented ray casting for data parallel volume rendering. In *Proceedings of the 1993 Symposium on Parallel Rendering* (1993). 2
- [IBH11] IZE T., BROWNLEE C., HANSEN C. D.: Real-Time Ray Tracer for Visualizing Massive Models on a Cluster. In *Eurographics Symposium on Parallel Graphics and Visualization* (2011). 2, 3
- [ISM*08] ILIEV I. T., SHAPIRO P. R., MELLEMA G., MERZ H., PEN U.-L.: Simulating Cosmic Reionization. *arXiv:0806.2887 [astro-ph]* (2008). [arXiv:0806.2887](https://arxiv.org/abs/0806.2887). 1, 8
- [KPH*10] KENDALL W., PETERKA T., HUANG J., SHEN H.-W., ROSS R. B.: Accelerating and Benchmarking Radix-k Image Compositing at Large Scale. *Eurographics Symposium on Parallel Graphics and Visualization* (2010). 2
- [LM15] LEE M., MOSER R. D.: Direct numerical simulation of turbulent channel flow up to $Re_\tau=5200$. *Journal of Fluid Mechanics* (2015). 1, 8
- [MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications* (1994). 2
- [MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An Image Compositing Solution at Scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (2011). 2, 6
- [MPHK94] MA K.-L., PAINTER J. S., HANSEN C. D., KROGH M. F.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications* (1994). 2
- [NFLC12] NAVRÁTIL P. A., FUSSELL D., LIN C., CHILDS H.: Dynamic scheduling for large-scale distributed-memory ray tracing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012). 3
- [PFN18] PARK H., FUSSELL D., NAVRÁTIL P.: SpRay: Speculative Ray Scheduling for Large Data Visualization. In *2018 IEEE Symposium on Large Data Analysis and Visualization* (2018). 3, 10
- [PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009). 2
- [PM12] PHARR M., MARK W. R.: ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing (In-Par), 2012* (2012). 3, 7
- [RCJ99] REINHARD E., CHALMERS A., JANSEN F. W.: Hybrid scheduling for parallel rendering using coherent ray tasks. In *Proceedings of the 1999 IEEE Symposium on Parallel Visualization and Graphics* (1999). 3
- [RK79] ROMAN G.-C., KIMURA T.: A VLSI Architecture for Real-Time Color Display of Three-Dimensional Objects. *Proceedings of IEEE Micro-Delcon* (1979). 2
- [RWC15] RATHKE B., WALD I., CHIU K., BROWNLEE C.: SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization* (2015). 7

- [SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (2000). 3
- [URW*18] USHER W., RIZZI S., WALD I., AMSTUTZ J., INSLEY J., VISHWANATH V., FERRIER N., PAPKA M. E., PASCUCCI V.: libIS: A Lightweight Library for Flexible In Transit Visualization. In *ISAV: In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization* (2018). 10
- [VSW*17] VIERJAHN T., SCHNORR A., WEYERS B., DENKER D., WALD I., GARTH C., KUHLEN T. W., HENTSCHEL B.: Interactive Exploration of Dissipation Element Geometry. In *Eurographics Symposium on Parallel Graphics and Visualization* (2017). 7
- [WBS02] WALD I., BENTHIN C., SLUSALLEK P.: *A Flexible and Scalable Rendering Engine for Interactive 3D Graphics*. Tech. rep., Saarland University, 2002. 2
- [WBUK17] WALD I., BROWNLEE C., USHER W., KNOLL A.: CPU Volume Rendering of Adaptive Mesh Refinement Data. In *SIGGRAPH Asia 2017 Symposium on Visualization* (2017). 7
- [WJA*17] WALD I., JOHNSON G. P., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRÁTIL P.: OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* (2017). 2, 3, 7
- [WKJ*15] WALD I., KNOLL A., JOHNSON G. P., USHER W., PASCUCCI V., PAPKA M. E.: CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees. In *2015 IEEE Scientific Visualization Conference (SciVis)* (2015). 7
- [WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001*. 2001. 2, 3
- [WUP*18] WU Q., USHER W., PETRUZZA S., KUMAR S., WANG F., WALD I., PASCUCCI V., HANSEN C. D.: VisIt-OSPRay: Toward an Exascale Volume Visualization System. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018). 2, 10
- [WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics* (2014). 3
- [WWW*19] WANG F., WALD I., WU Q., USHER W., JOHNSON C. R.: CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data. *IEEE Transactions on Visualization and Computer Graphics* (2019). 7
- [YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2–3 swap image compositing. In *SC-International Conference for High Performance Computing, Networking, Storage and Analysis* (2008). 2