Scalable Ray Tracing Using the Distributed FrameBuffer

Will Usher, Ingo Wald, Jefferson Amstutz, Johannes Günther, Carson Brownlee, and Valerio Pascucci

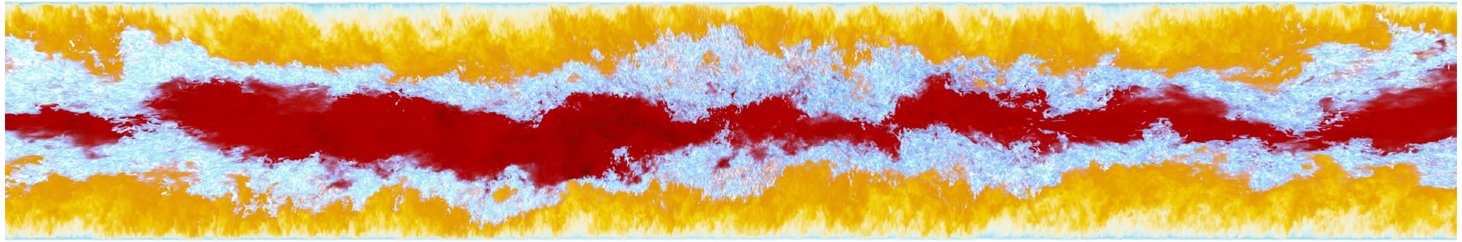
Intel Parallel Computing Centers

www.sci.utah.edu

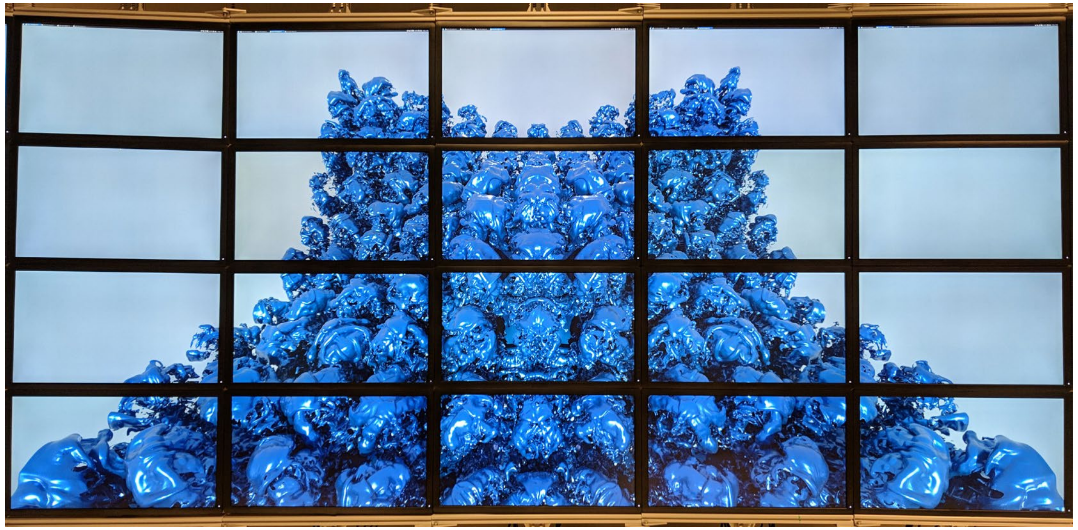
THE UNIVERSITY OF UTAH®

Challenging Rendering Problems Demand More Compute and Memory

- Expensive shading/geometry and high-resolutions require more compute for interactive rendering
- Large datasets (100+GB to TBs) cannot fit on one node
- In Situ visualization inherently requires multi-node rendering



951GB volume + 4.35B triangles at 4096x1024



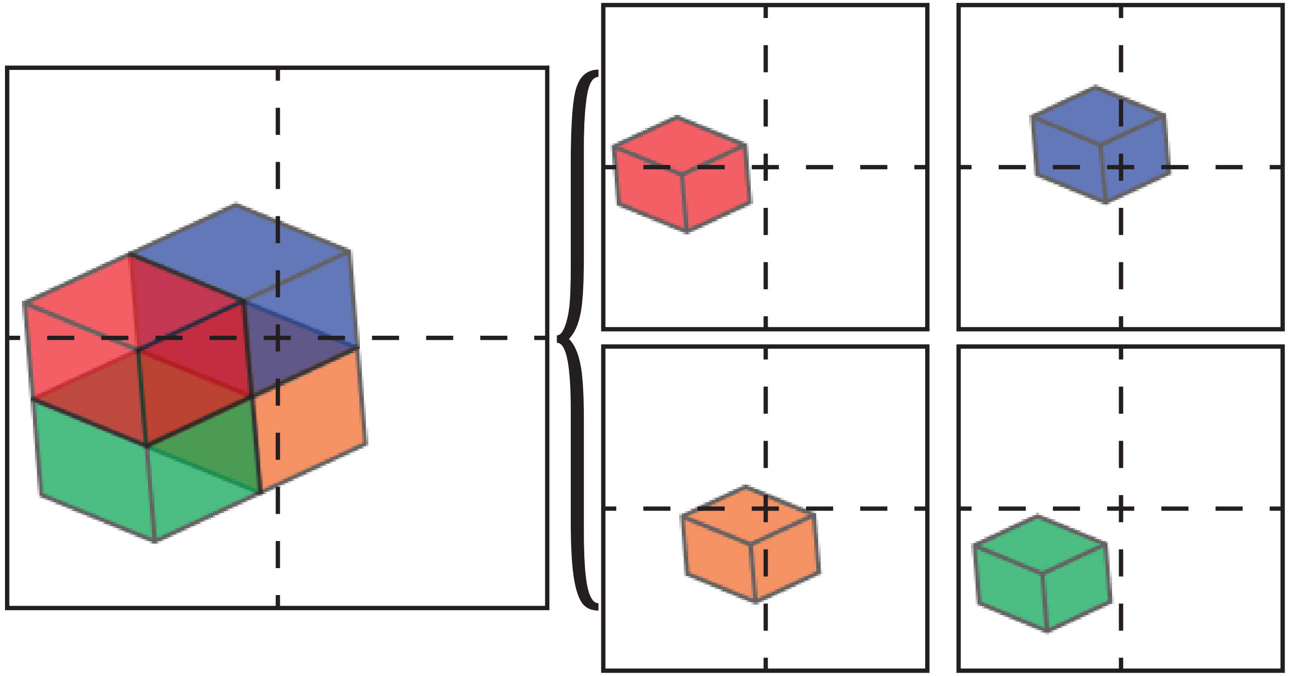
222M triangles w/ path tracing at 12800x5760

Prior Work

- Sort-last: Distribute sub-regions of **data** to render, sort partial images produced on each node
- Sort-first: Distribute sub-regions of **image** to render, sort objects before or during rendering
- Hybrid: Combine image and data work distributions

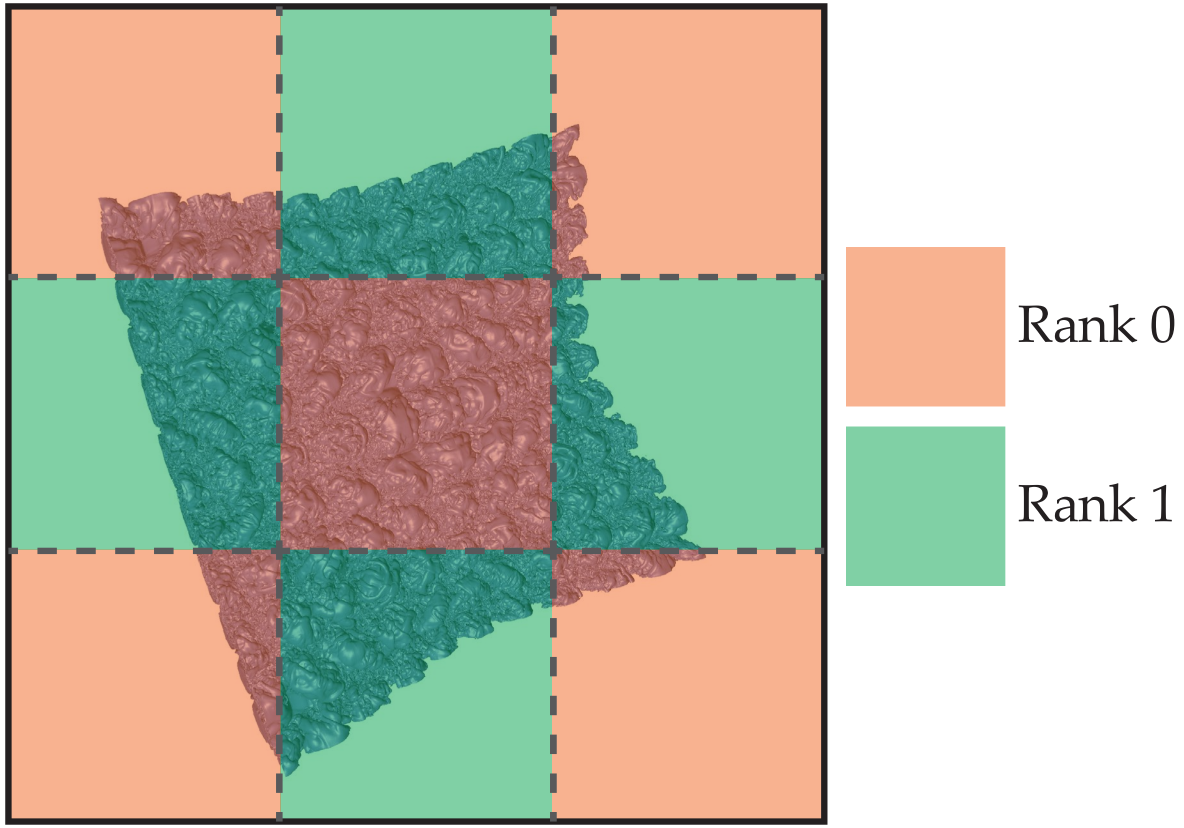
Sort-Last: Object-Space Work Distribution

- “Data-parallel” rendering
- Brick dataset and distribute among nodes, or using an existing distribution (in situ)
- Render each brick locally, composite partial images [Hsu 93; Ma et al. 94; Peterka et al. 09; Moreland et al. 11; Grosset et al. 15]



Sort-First: Image-Space Work Distribution

- "Image-parallel" rendering
- Work unit: Image tiles
- Load balancing [Wald et al. 01; Ize et al. 11]
- Large data: page from disk or network into cache on-demand [Wald et al. 01; DeMarle et al. 03, 04 & 05; Ize et al. 11]



Limitations of Prior Work

- Purpose-built solutions for each approach (sort-last, sort-first)
- Widely available software for sort-last [Moreland et al. 11] imposes restrictions on the data-distribution between nodes
- Sort-first methods can bottleneck on the master process at high-resolutions
- Widely used methods do not overlap image compositing/processing with rendering

Our Contributions

- A flexible and scalable parallel framework to run image compositing and processing tasks for distributed renderers

Our Contributions

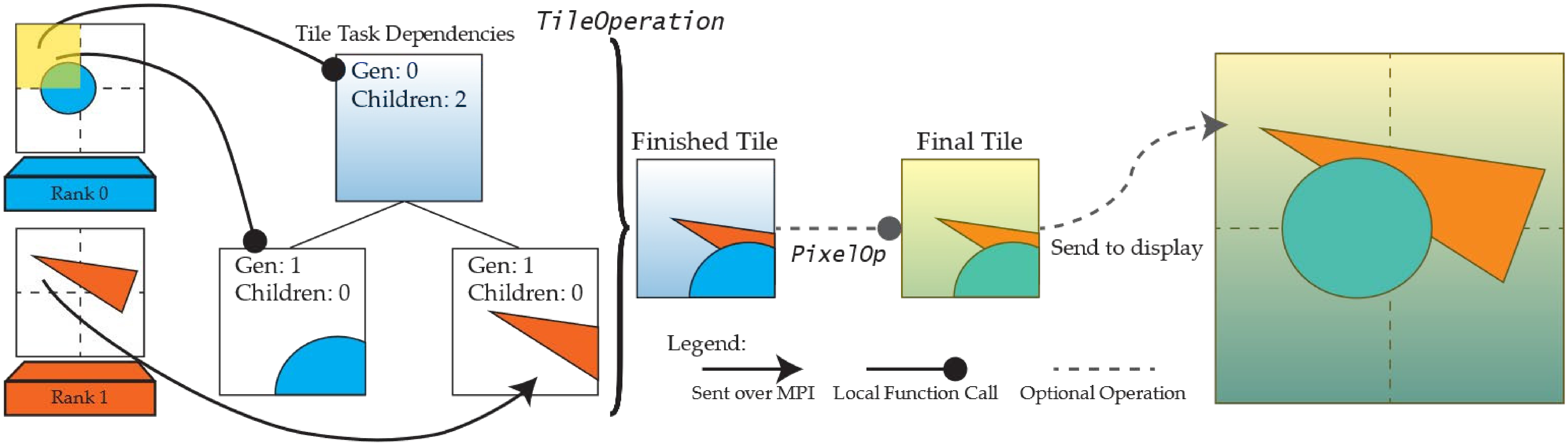
- A flexible and scalable parallel framework to run image compositing and processing tasks for distributed renderers
- A set of parallel rendering algorithms built on this approach, covering standard use cases and more complex configurations

Our Contributions

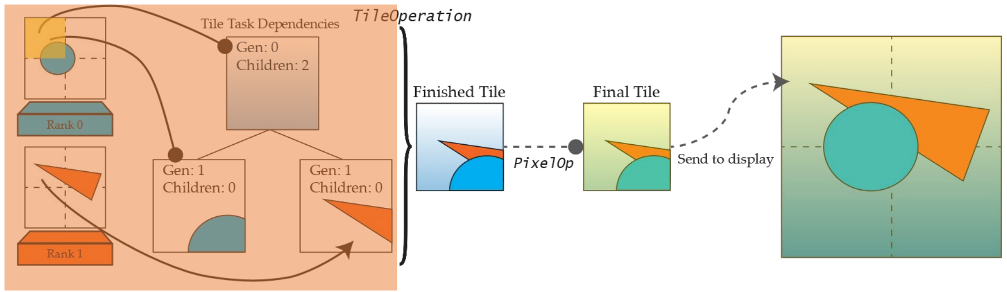
- A flexible and scalable parallel framework to run image compositing and processing tasks for distributed renderers
- A set of parallel rendering algorithms built on this approach, covering standard use cases and more complex configurations
- An extension of OSPRay to implement a distributed API, exposing the parallel rendering capabilities to end users

The Distributed FrameBuffer

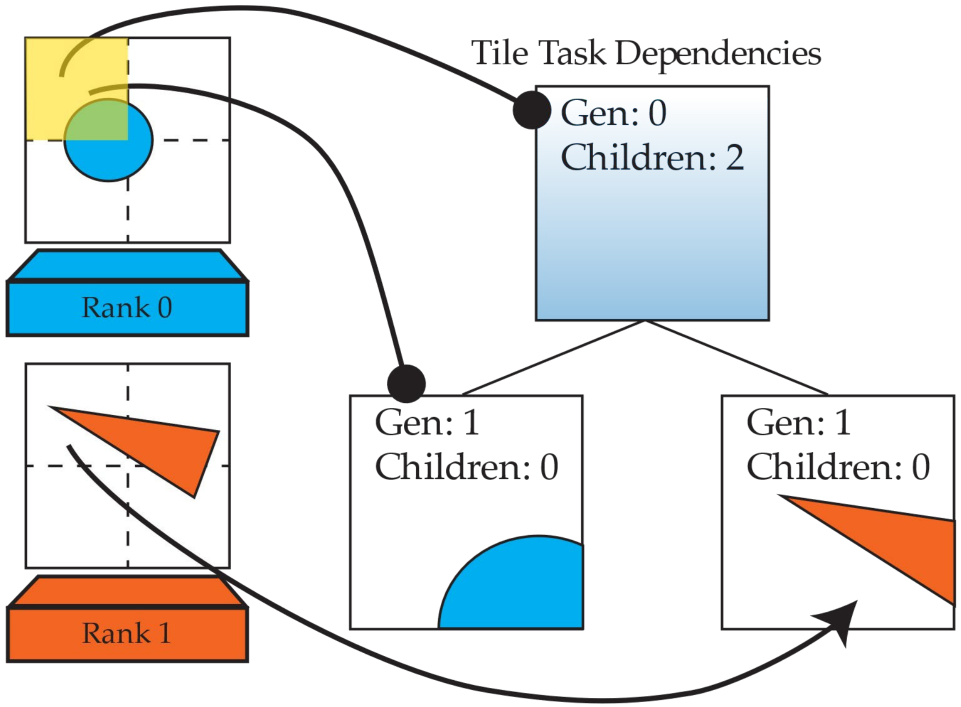
The DFB Tile Processing Pipeline



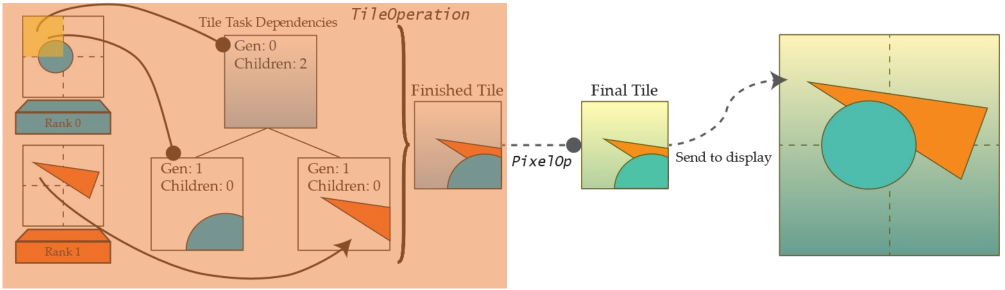
Tile Task Dependencies



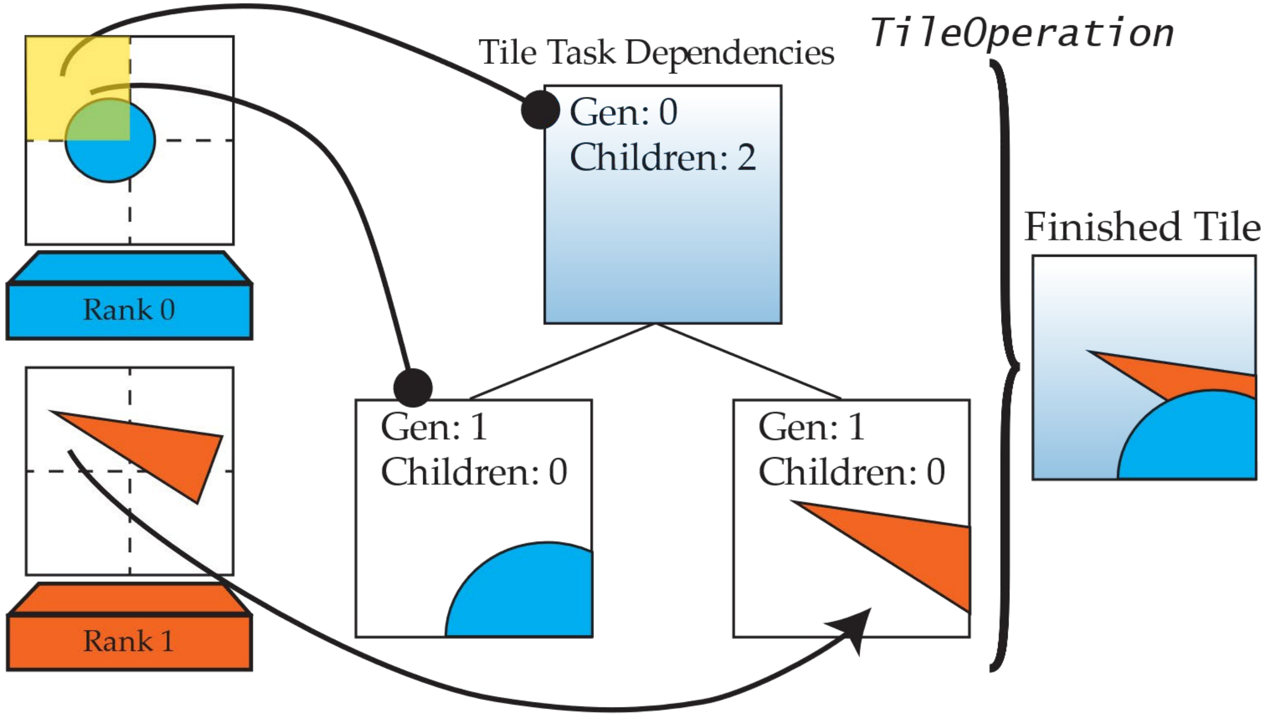
- Rendered input tiles can build a per-tile task dependency tree at runtime
- Tree construction and dependency tracking is managed by the TileOperation



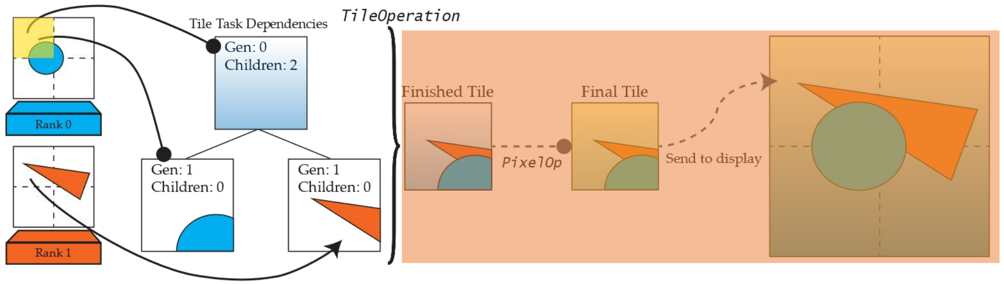
Tile Operations



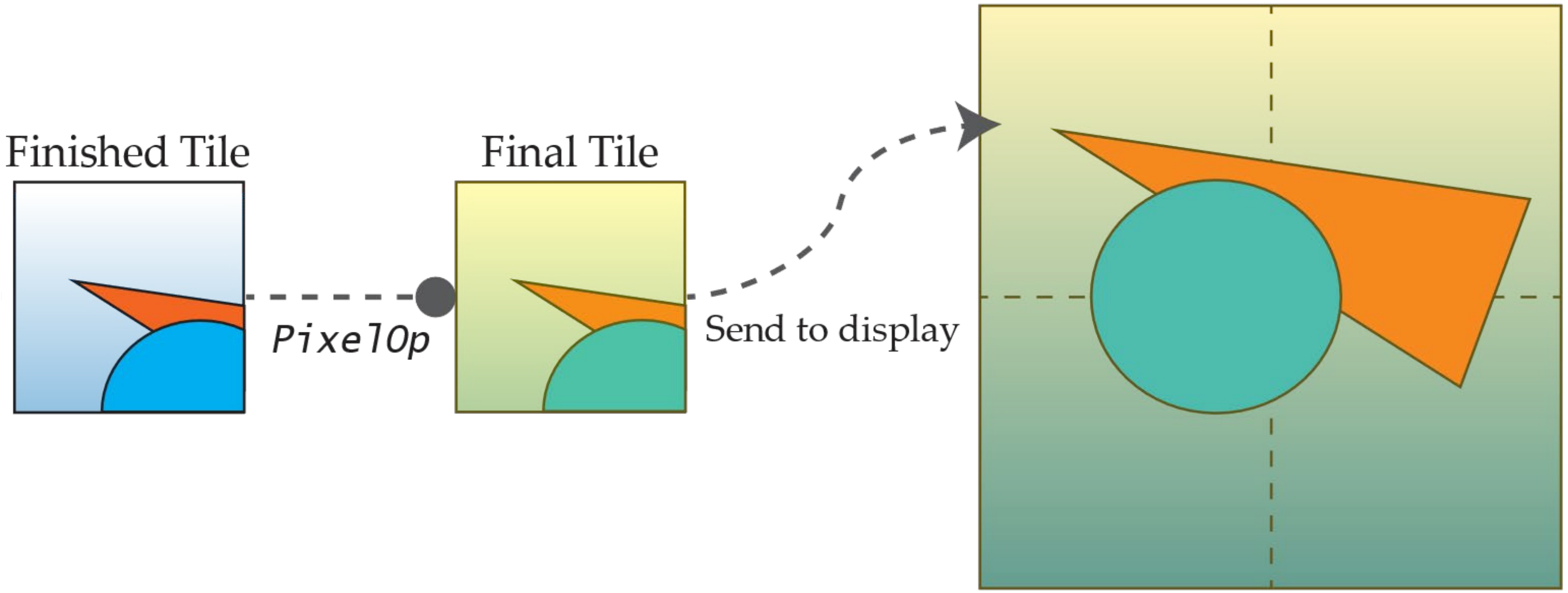
- Specifies how tiles should be combined to form the final image from the input tree
  - E.g., averaging, depth sorting, alpha-compositing



Pixel Operations



- Optional additional post-processing can be added via PixelOps
  - Tone mapping, denoising, etc.
  - Re-routing tiles to a display wall
- Independent of the Renderer and TileOperation



Asynchronous MPI Messaging Layer

- Communication runs on a background thread on each process, using non-blocking MPI
- DFB tile and pixel operations executed on background threads as dependencies are recieved
- Tile messages compressed with Snappy [Google]
- Final tiles are gathered to the master process with MPI_Gatherv

Rendering with the Distributed FrameBuffer

Anatomy of a Distributed Renderer

- Distributed Renderer = Renderer + TileOperation
- *Renderer*: Render local data to create tile task inputs
- *TileOperation*: Interpret and combine tile task inputs to make the finished image

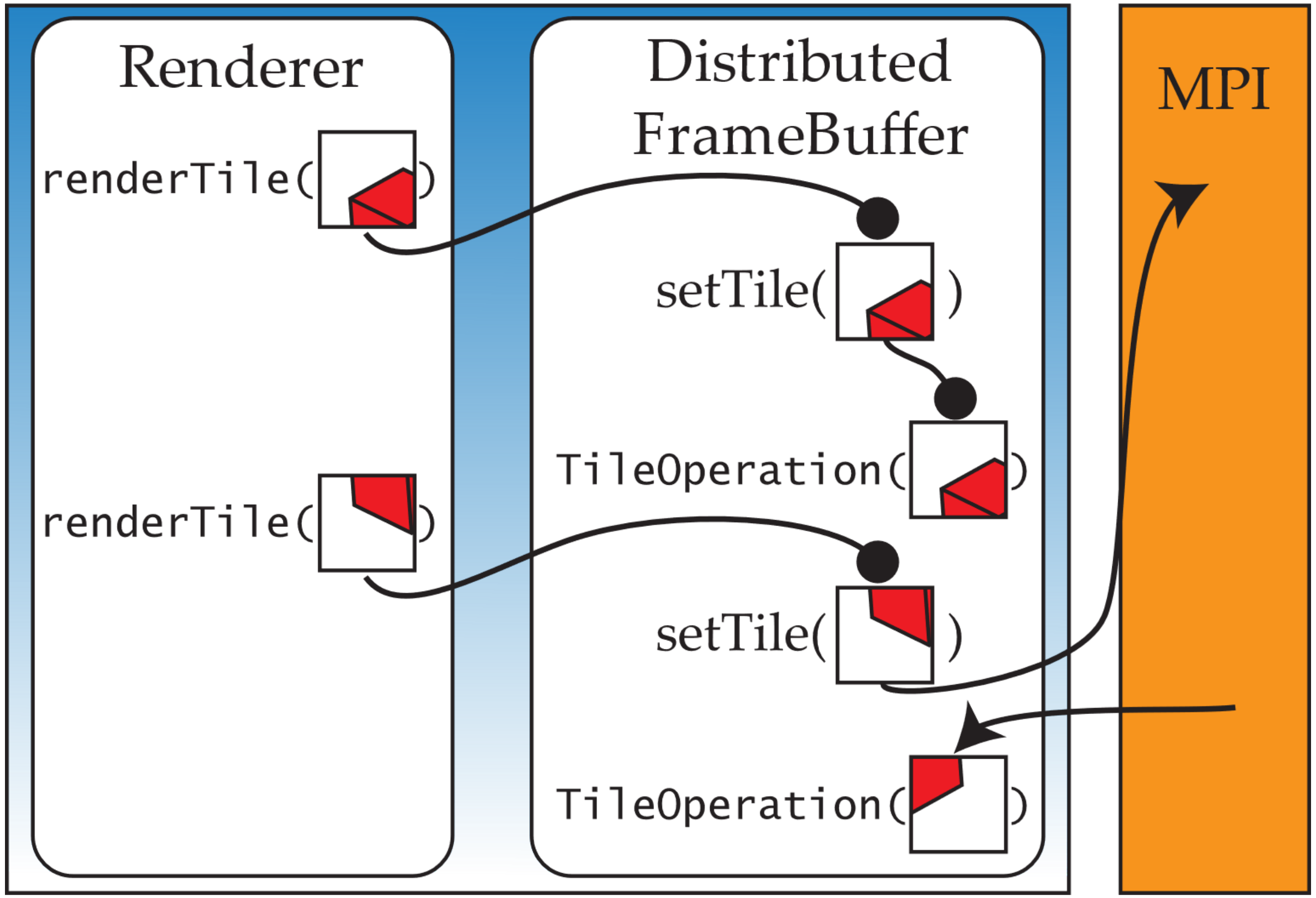
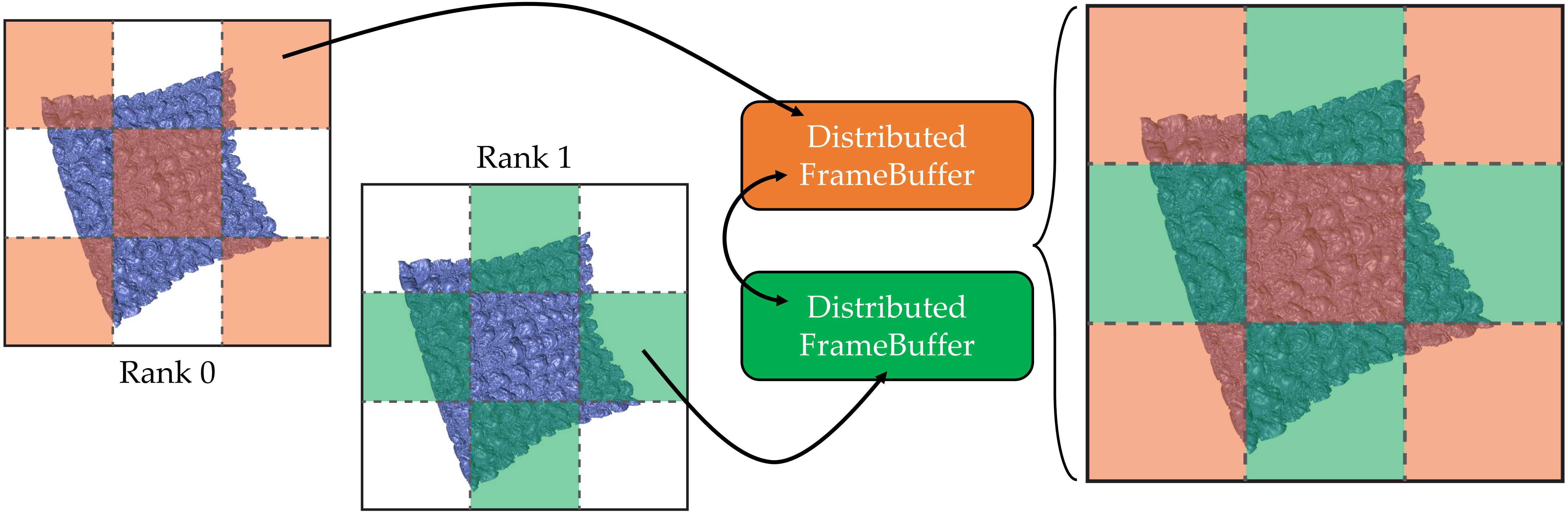


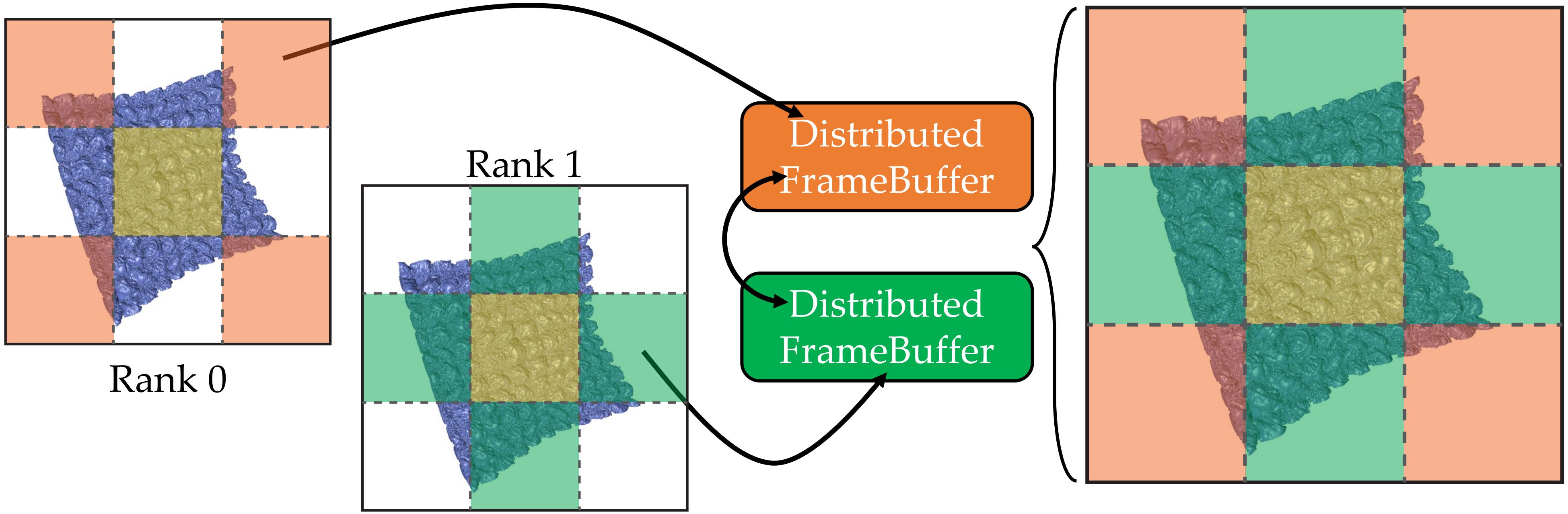
Image-Parallel Rendering

- Renderer: Assign each tile to be rendered by a unique process
- TileOperation: Expect a single input tile, forward to output



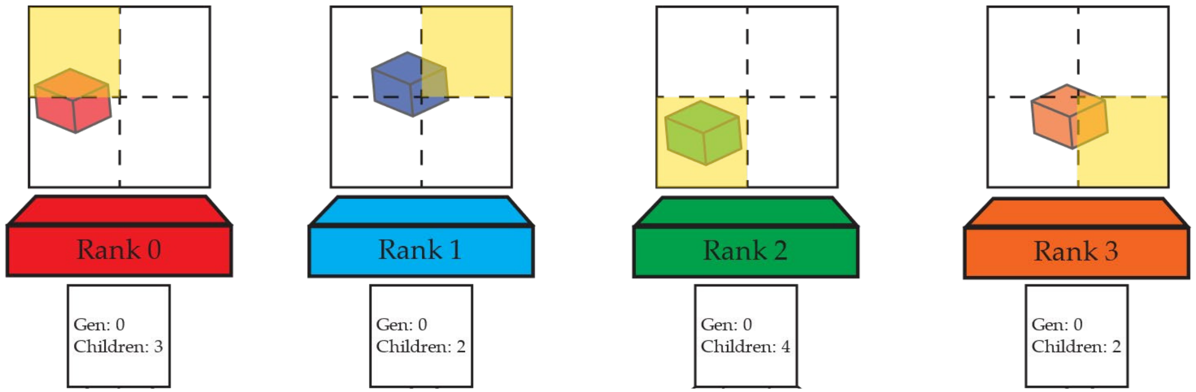
Load Balancing Image-Parallel Rendering

- Renderer: Assign each tile to be rendered by **one or more processes**
- TileOperation: Expect a **varying number** of input tiles **and average them together**



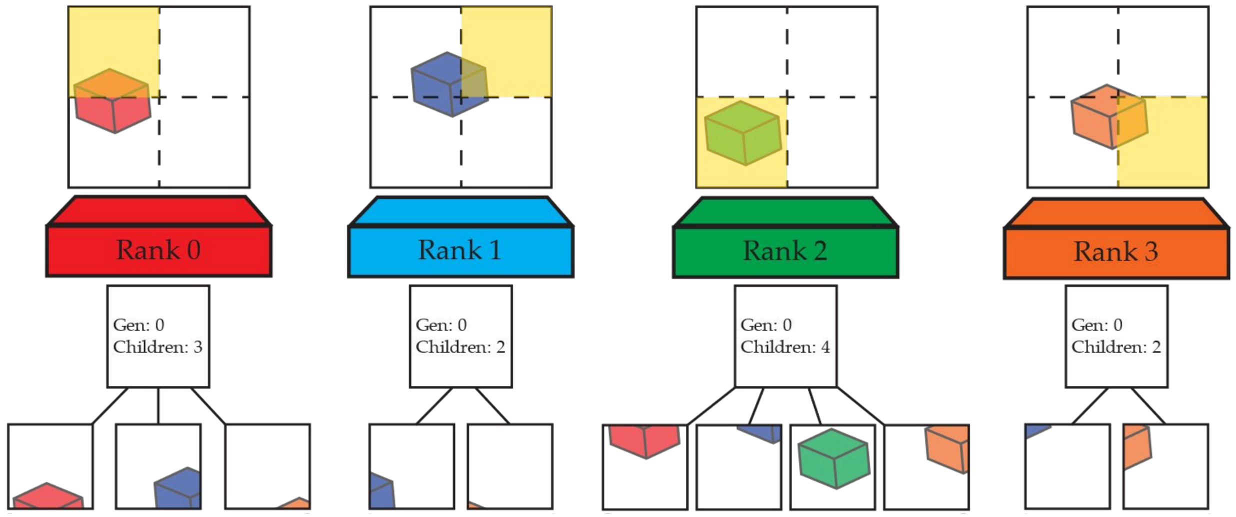
Sort-Last Data-Parallel Rendering

- Renderer: Render local data for tiles it touches, and a background tile for the tiles owned by the process



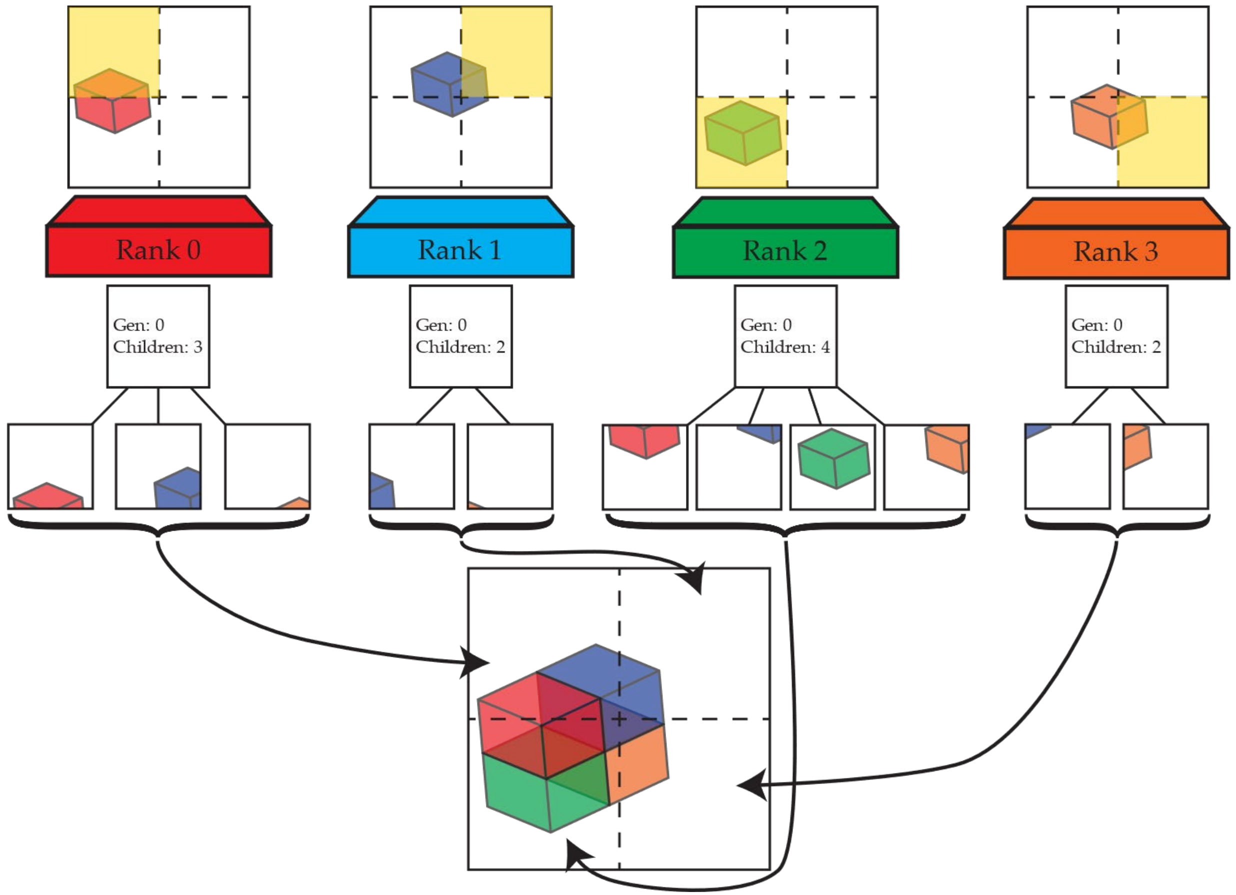
Sort-Last Data-Parallel Rendering

- Renderer: Render local data for tiles it touches, and a background tile for the tiles owned by the process
- TileOperation: Collect generation 0 and generation 1 tiles, sort and blend fragments of generation 1 tiles



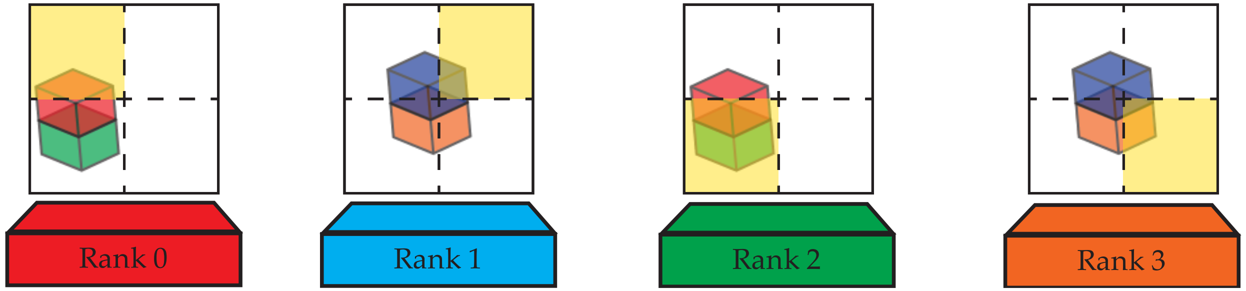
Sort-Last Data-Parallel Rendering

- Renderer: Render local data for tiles it touches, and a background tile for the tiles owned by the process
- TileOperation: Collect generation 0 and generation 1 tiles, sort and blend fragments of generation 1 tiles



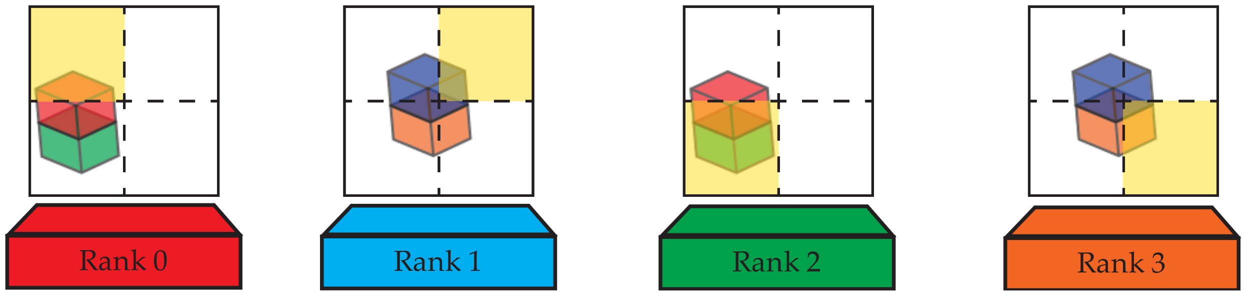
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes



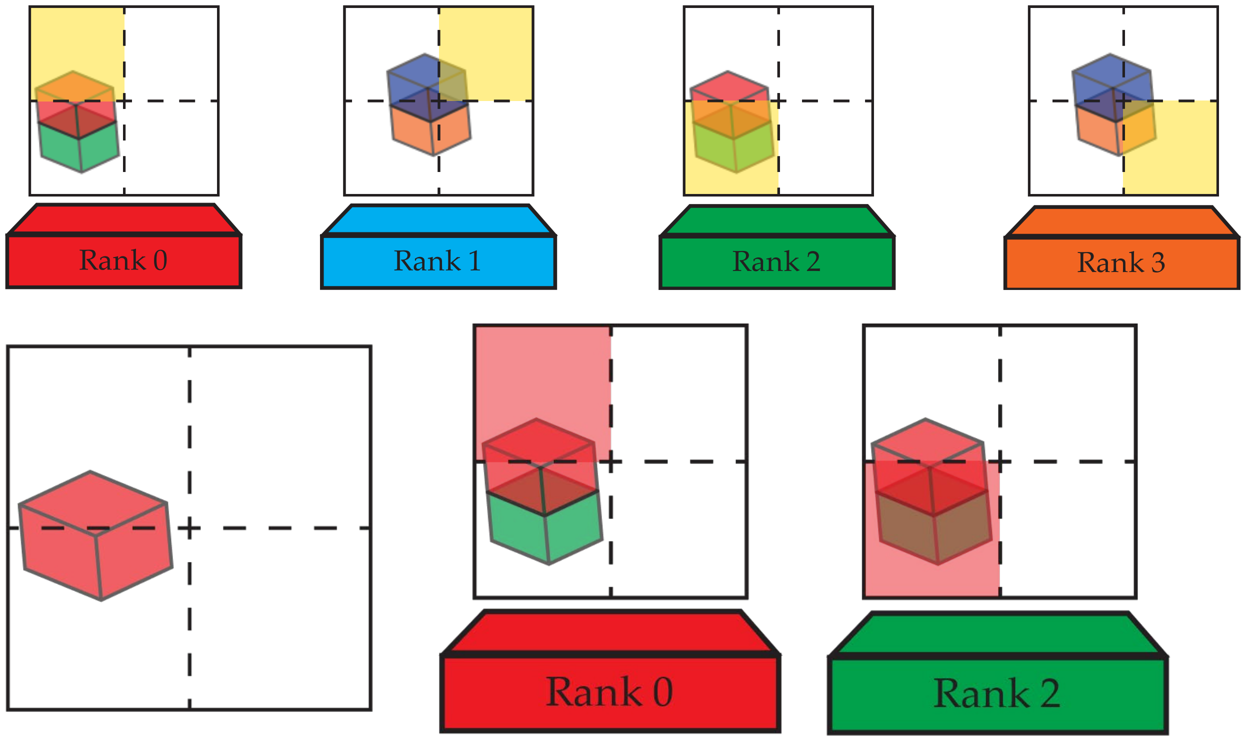
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile



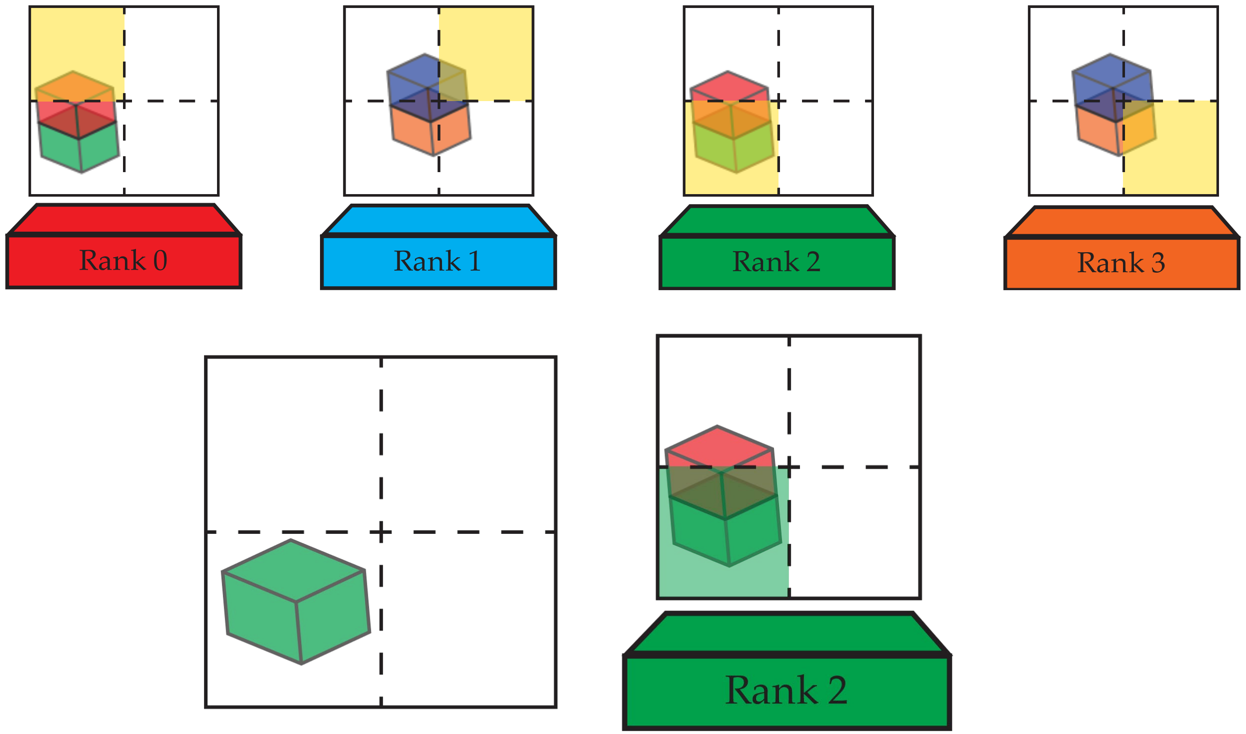
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile



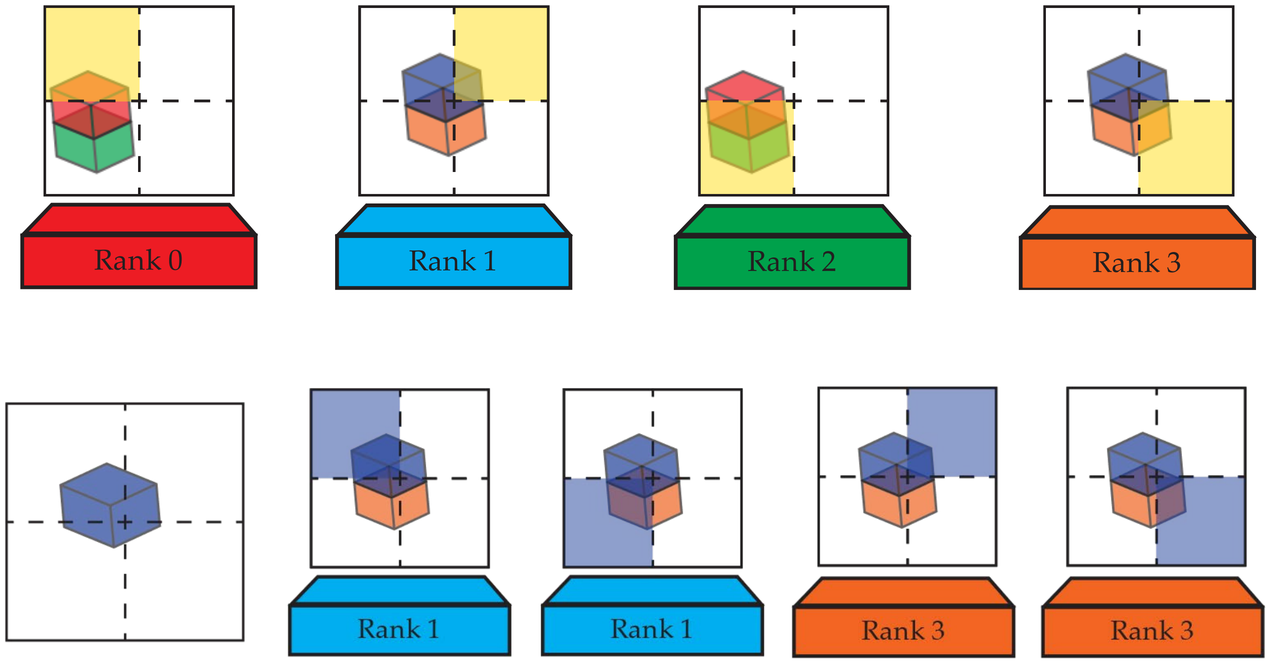
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile



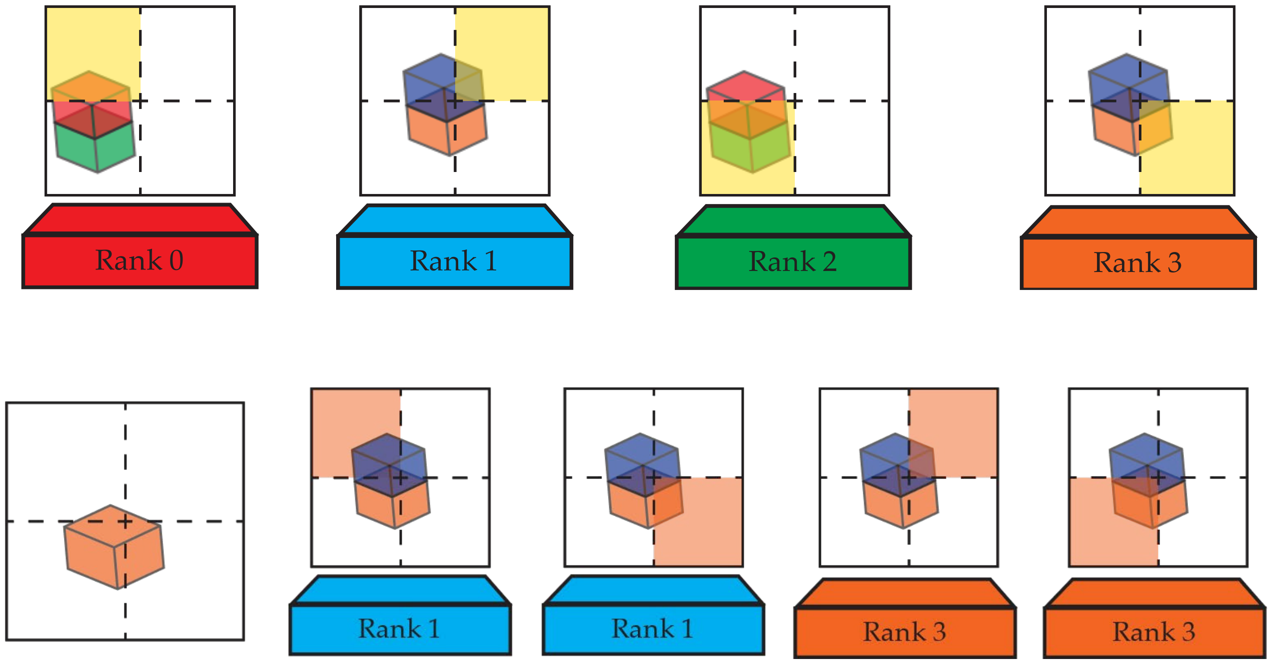
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile



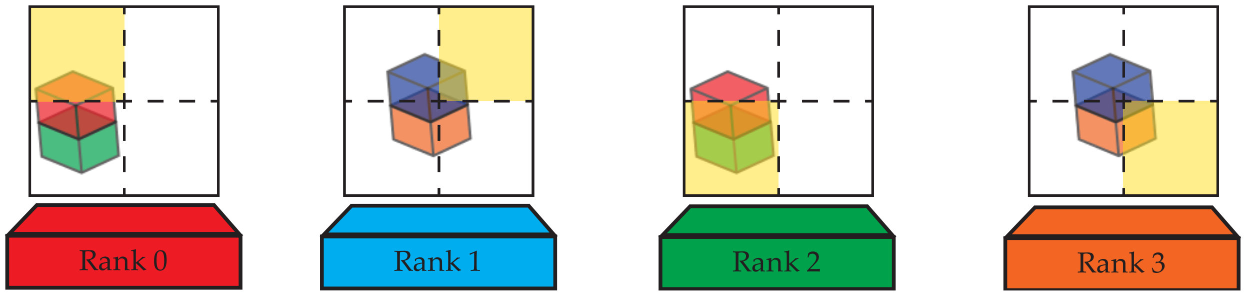
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile



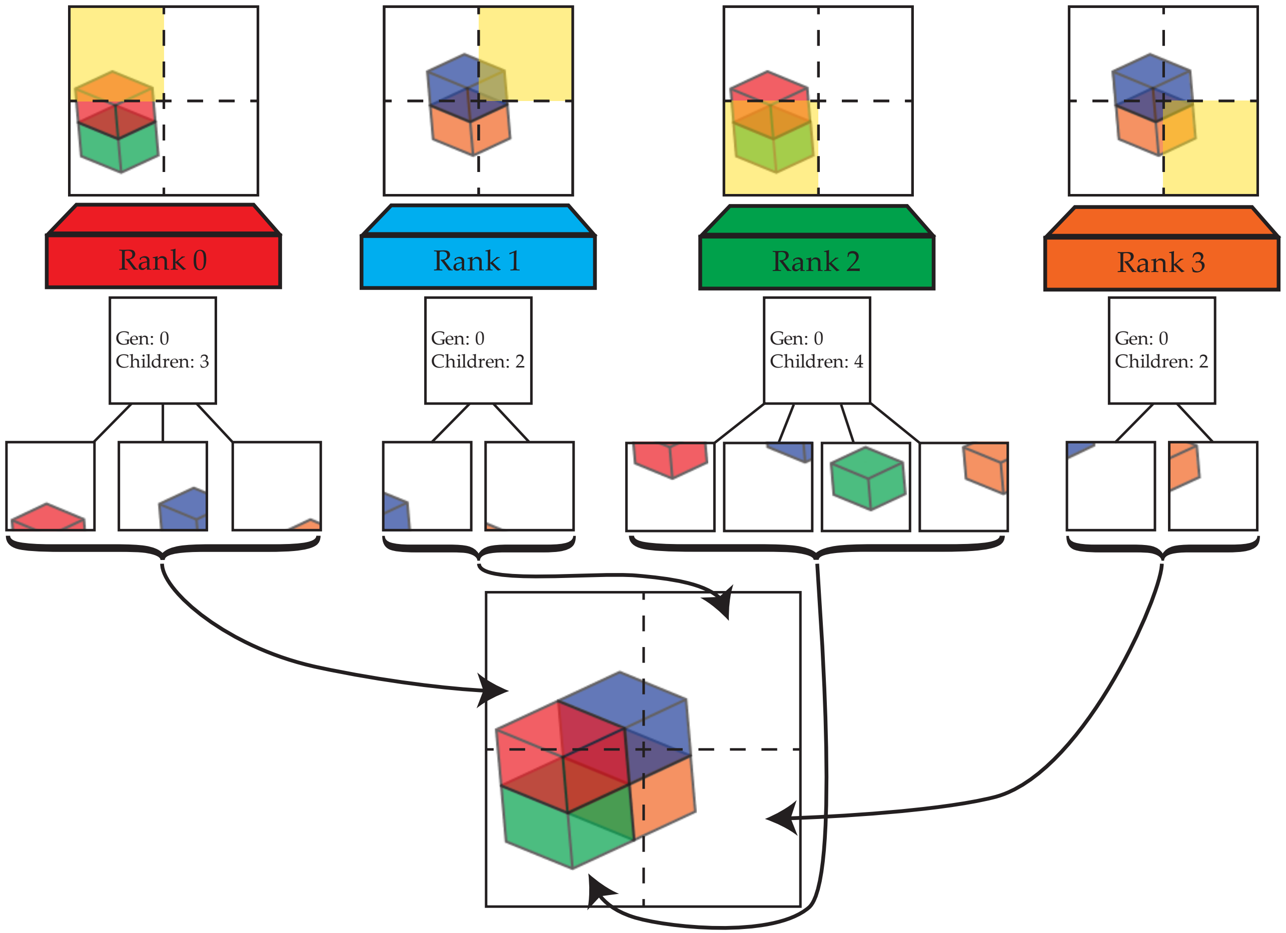
Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile
- TileOperation: Identical to typical sort-last cases



Rendering Hybrid Data Distributions

- Load balance data-parallel rendering by partially replicating data among nodes
- Renderer: Assign a unique process among those sharing each to render it for each tile
- TileOperation: Identical to typical sort-last cases



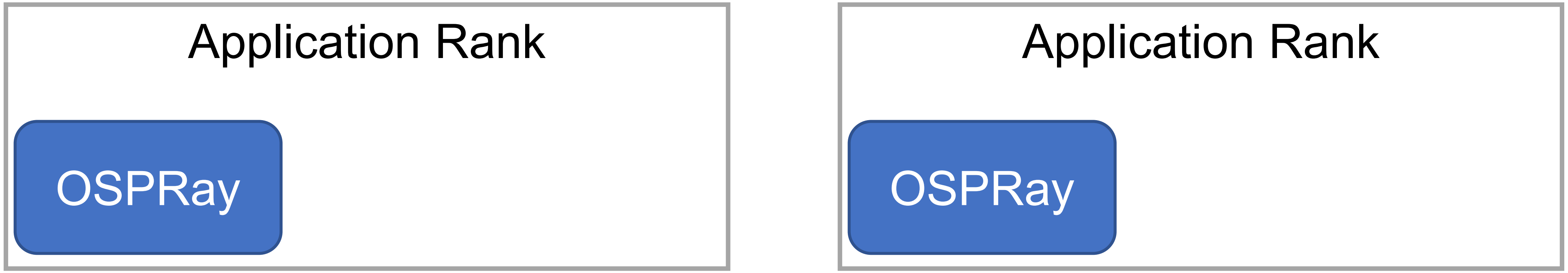
Scalable Support for Display Walls

- Route tiles directly to the display wall from the tile owner in the pixel operation
- Skip bottleneck of aggregating large images to the master

A Data-Distributed API for OSPRay

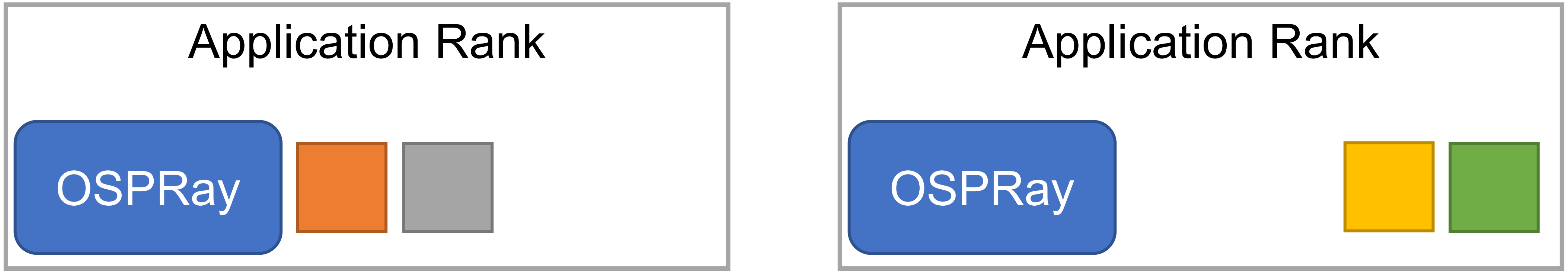
Distributed Rendering Support in OSPRay

- Extend OSPRay with a new MPIDistributedDevice API backend
- Distributed data abstracted as a set of OSPModels (bricks) with possible replication
- Supports existing OSPRay geometry and volume modules rendered as local data, composited with DFB



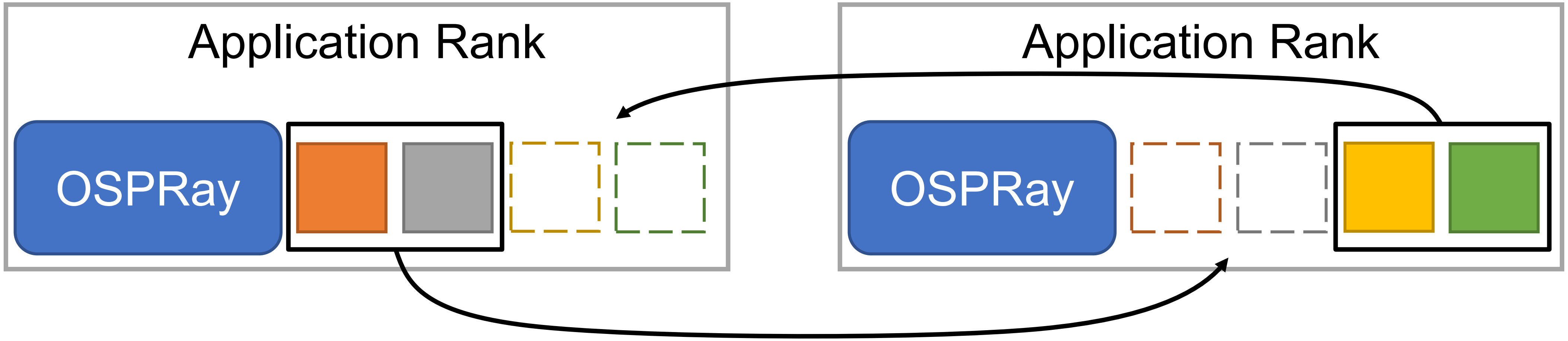
Distributed Rendering Support in OSPRay

- Extend OSPRay with a new MPIDistributedDevice API backend
- Distributed data abstracted as a set of OSPModels (bricks) with possible replication
- Supports existing OSPRay geometry and volume modules rendered as local data, composited with DFB



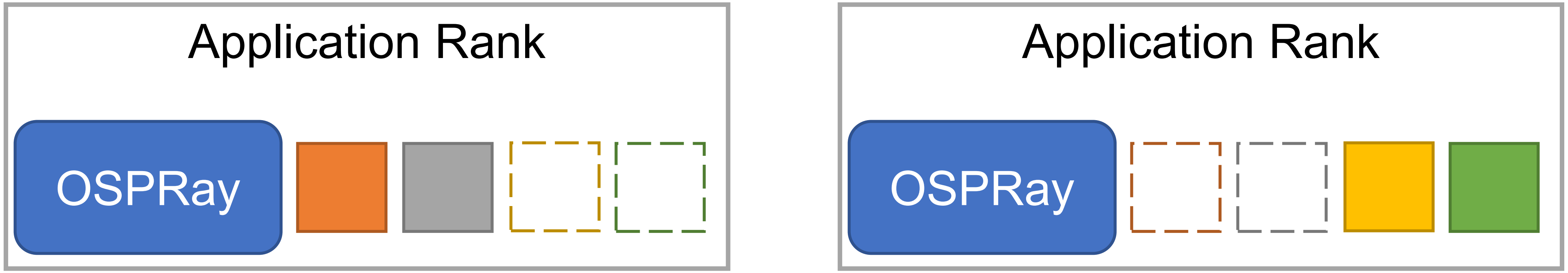
Distributed Rendering Support in OSPRay

- Extend OSPRay with a new MPIDistributedDevice API backend
- Distributed data abstracted as a set of OSPModels (bricks) with possible replication
- Supports existing OSPRay geometry and volume modules rendered as local data, composited with DFB



Distributed Rendering Support in OSPRay

- Extend OSPRay with a new MPIDistributedDevice API backend
- Distributed data abstracted as a set of OSPModels (bricks) with possible replication
- Supports existing OSPRay geometry and volume modules rendered as local data, composited with DFB



Results

Benchmark Configurations

- Run on:
  - TACC *Stampede2* KNL & SKX
  - ANL *Theta* KNL
- Similar KNL nodes, but very different networks
- Benchmarks use one process per-node and threads for on-node parallelism
- All benchmarks render a rotation around the dataset

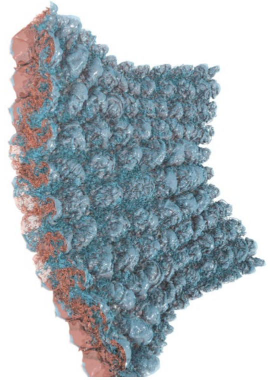
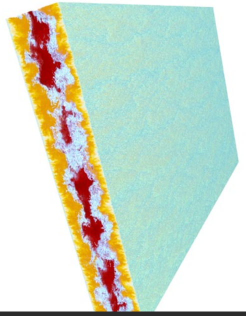
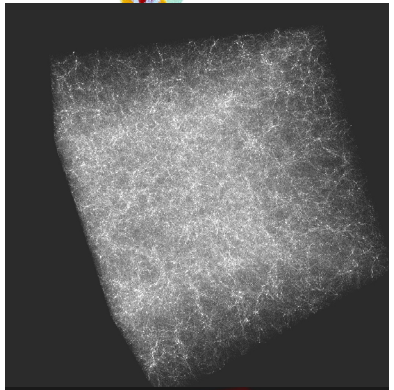
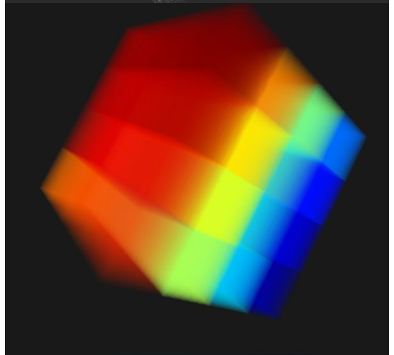
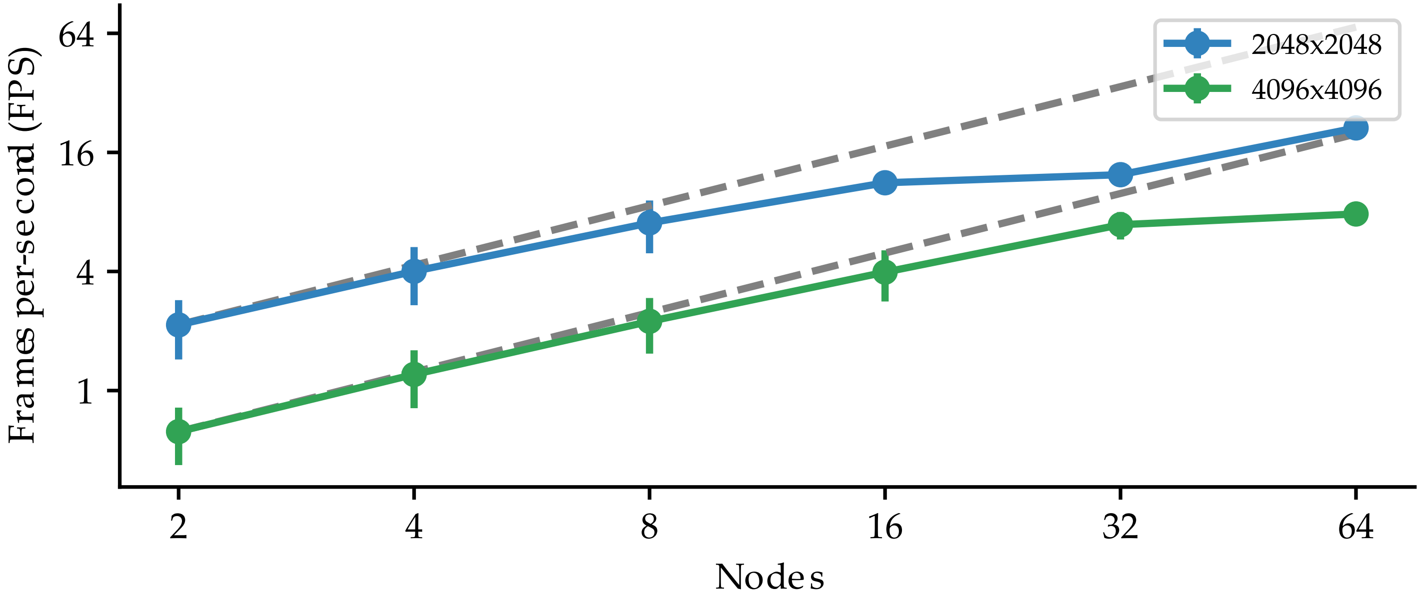
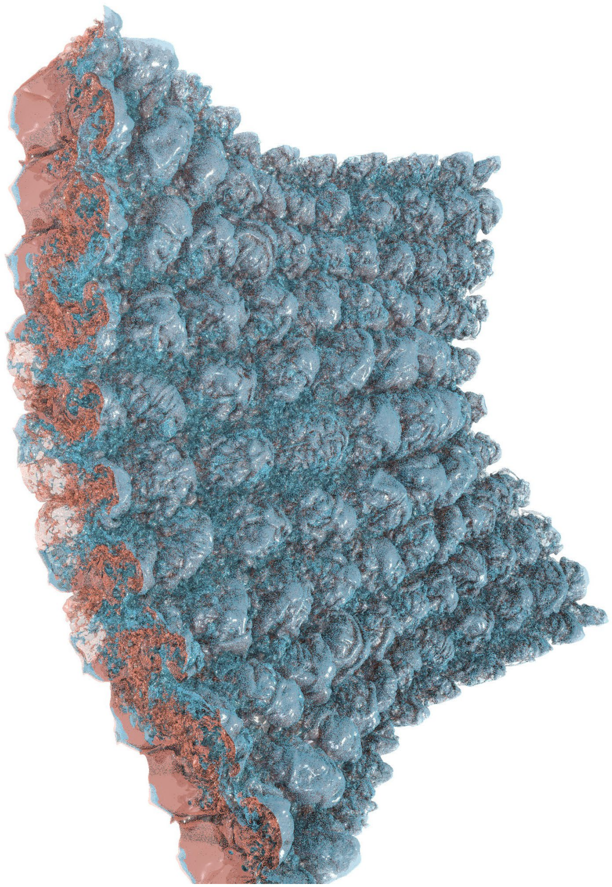





Image-Parallel Rendering Scalability

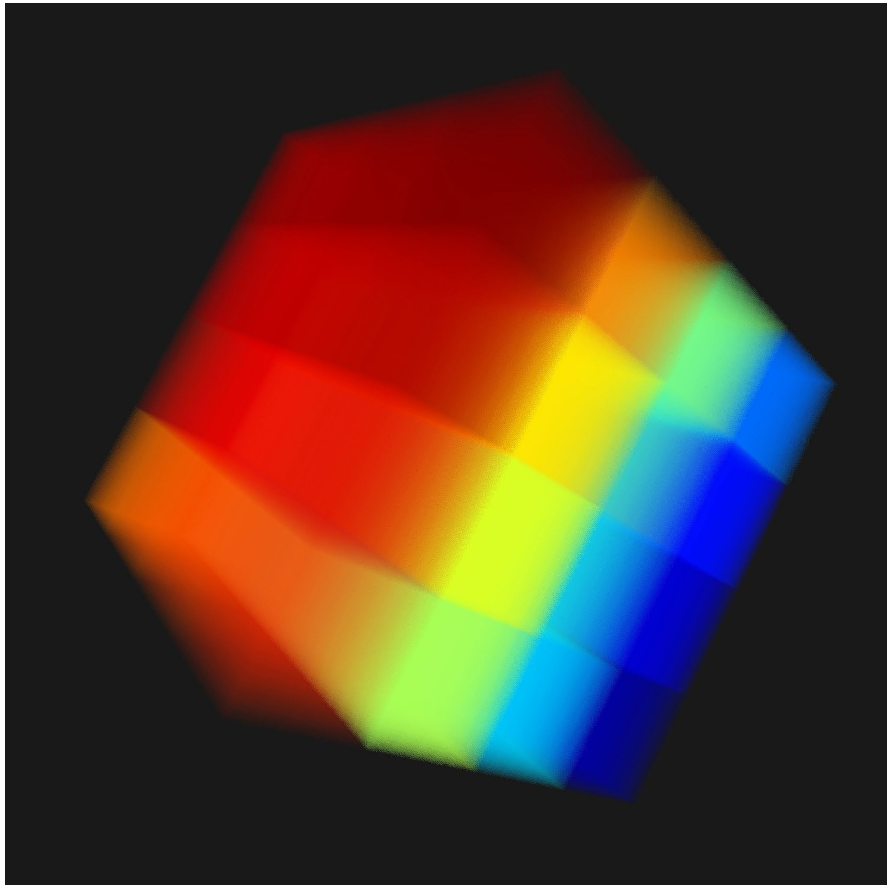
- Two transparent isosurfaces on the Richtmyer-Meshkov, 516M triangles
- Shadows and ambient occlusion
- *Stampede2* SKX



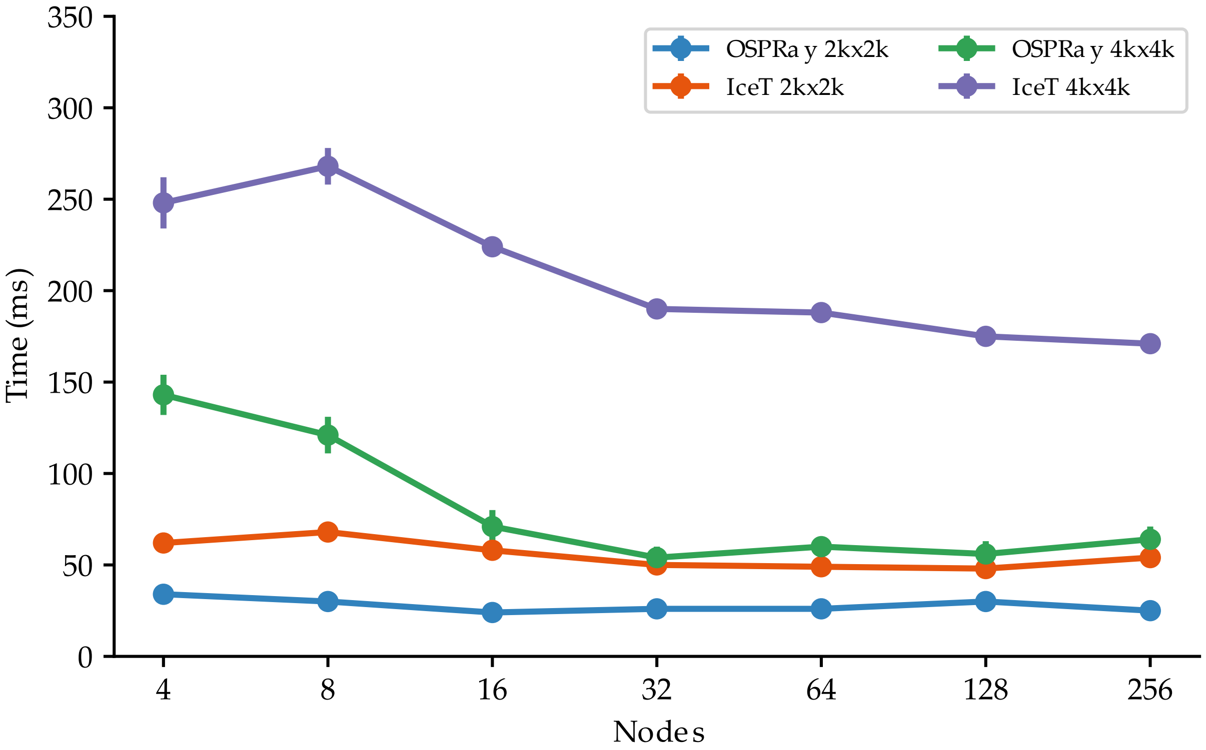


Data-Parallel Rendering: Compositing Benchmark vs. IceT

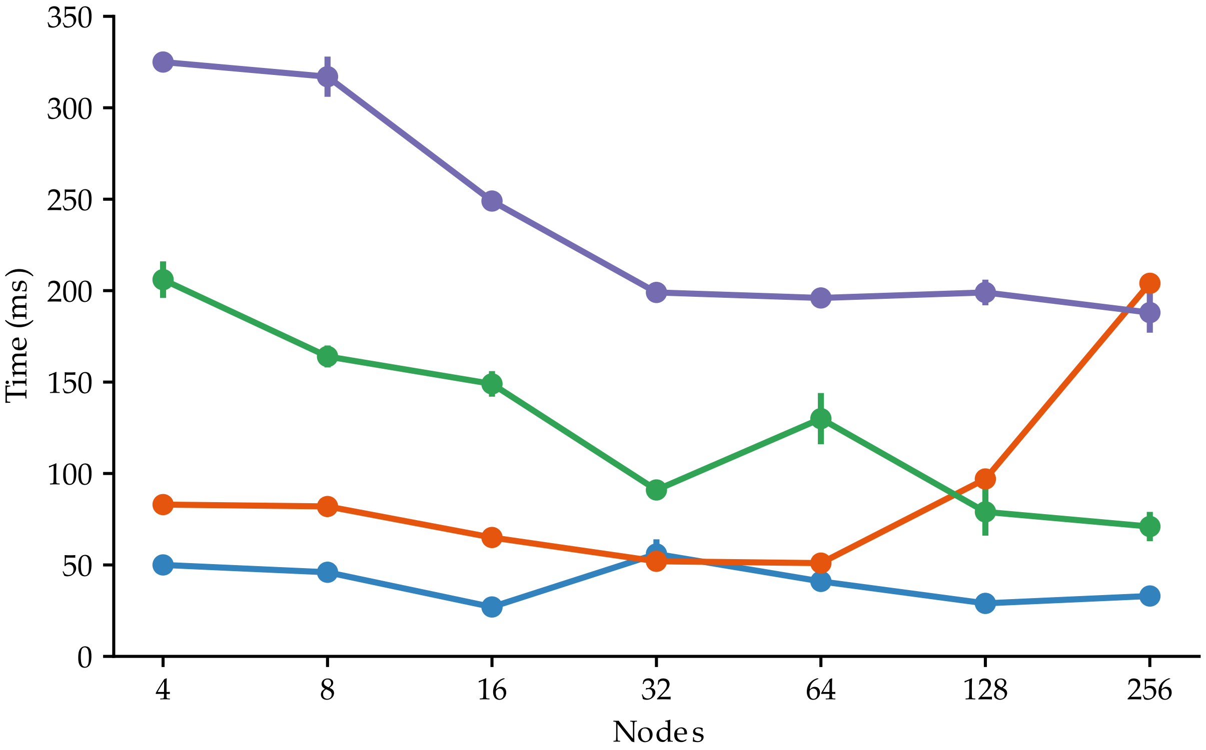
- Modify data-parallel renderer in OSPRay to use IceT for direct comparison
- Synthetic dataset with 64^3 volume brick per-node
- Also allows comparison between network architectures, job schedulers, system differences



IceT Comparison: Overall Performance

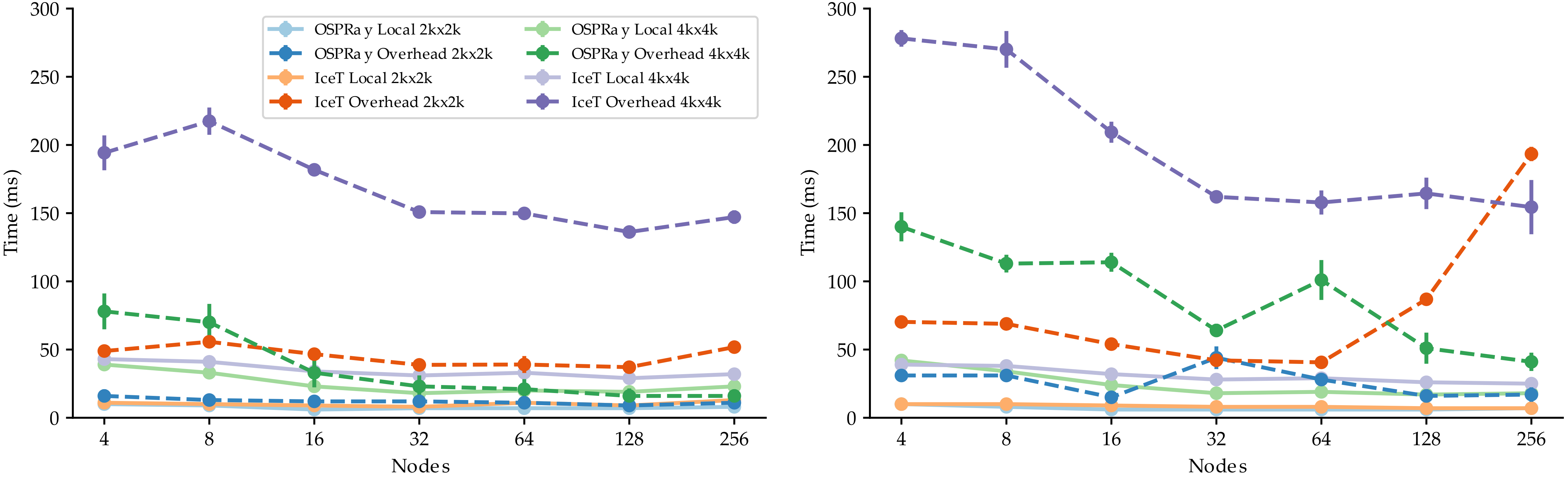


Theta KNL



Stampede2 KNL

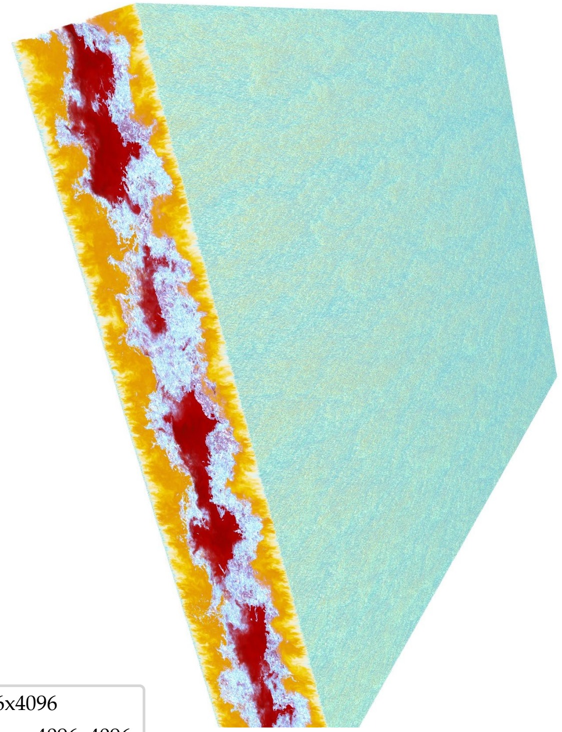
IceT Comparison: Timing Breakdown

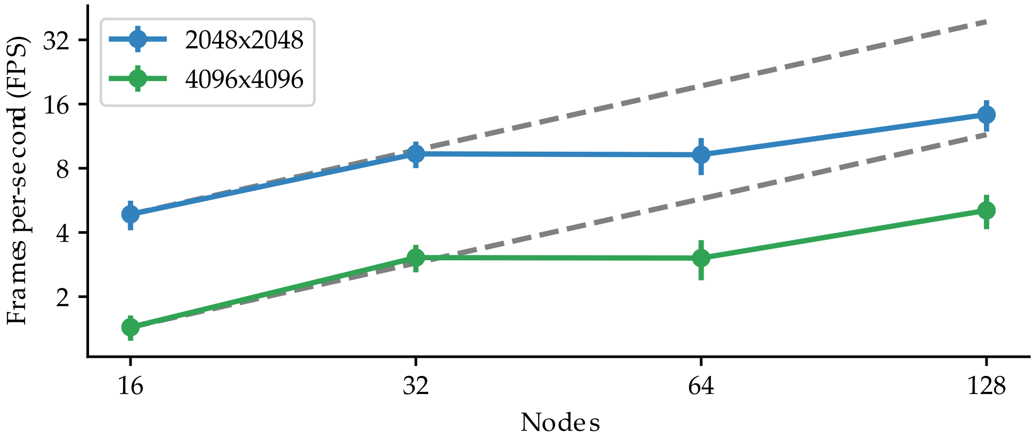


Theta KNL *Stampede2* KNL

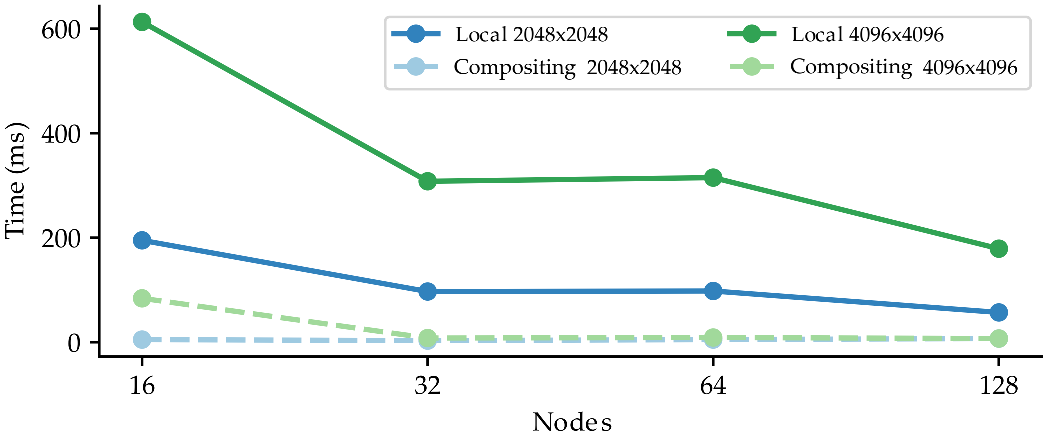
Data-Parallel Rendering: DNS Volume with Isosurfaces

- DNS single-precision volume 10240x7680x1536 (451GB)
- Two transparent isosurfaces, 5.43B triangles total
- *Stampede2* KNL





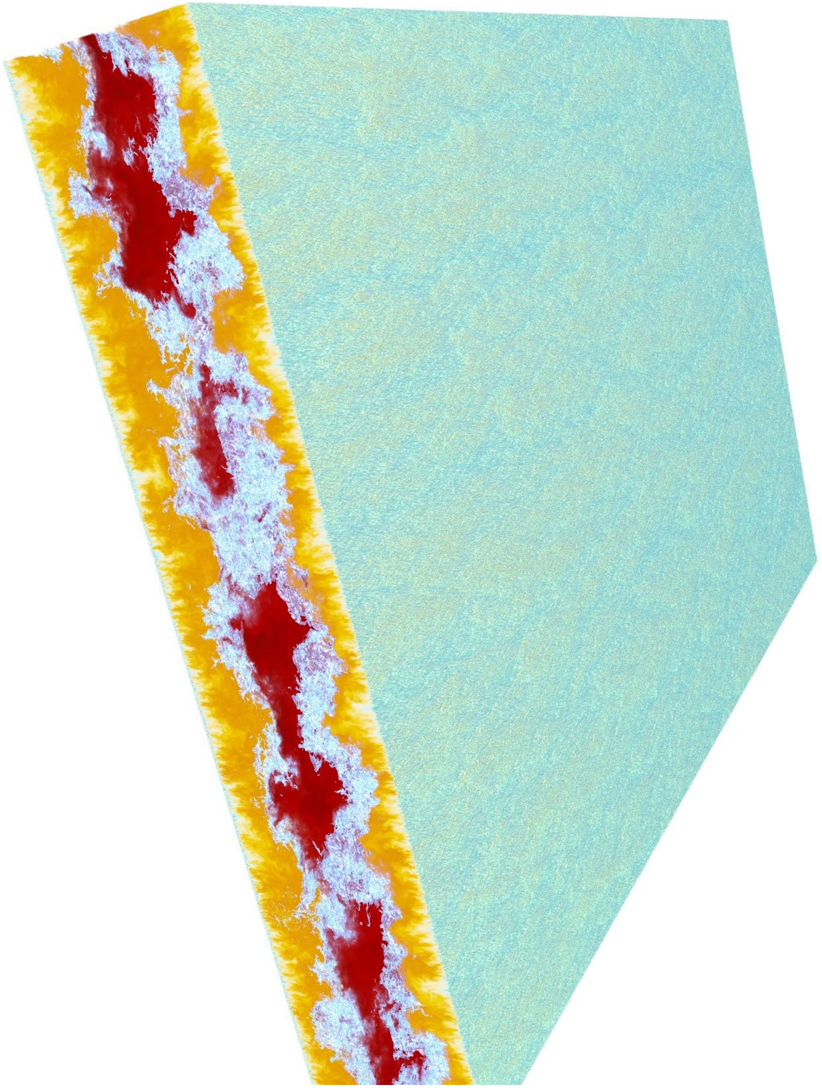
Overall Performance



Timing Breakdown

Hybrid-Parallel Rendering Performance

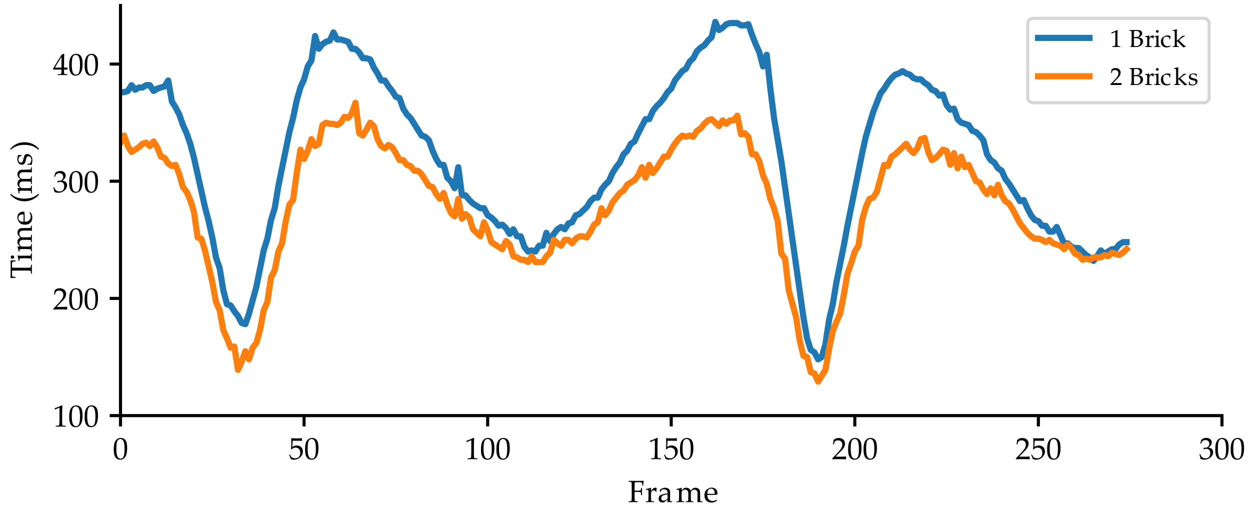
- Partially replicate bricks of data among nodes to improve load balancing



Hybrid-Parallel Rendering Performance

- Partially replicate bricks of data among nodes to improve load balancing
- 64 nodes: 2 bricks per/node

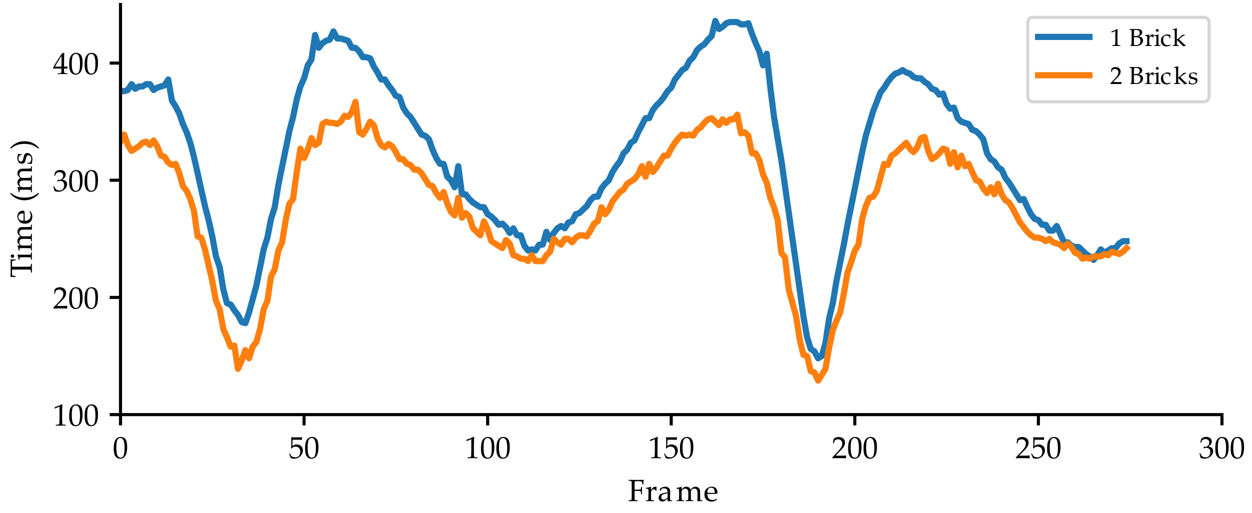
64 KNL Nodes



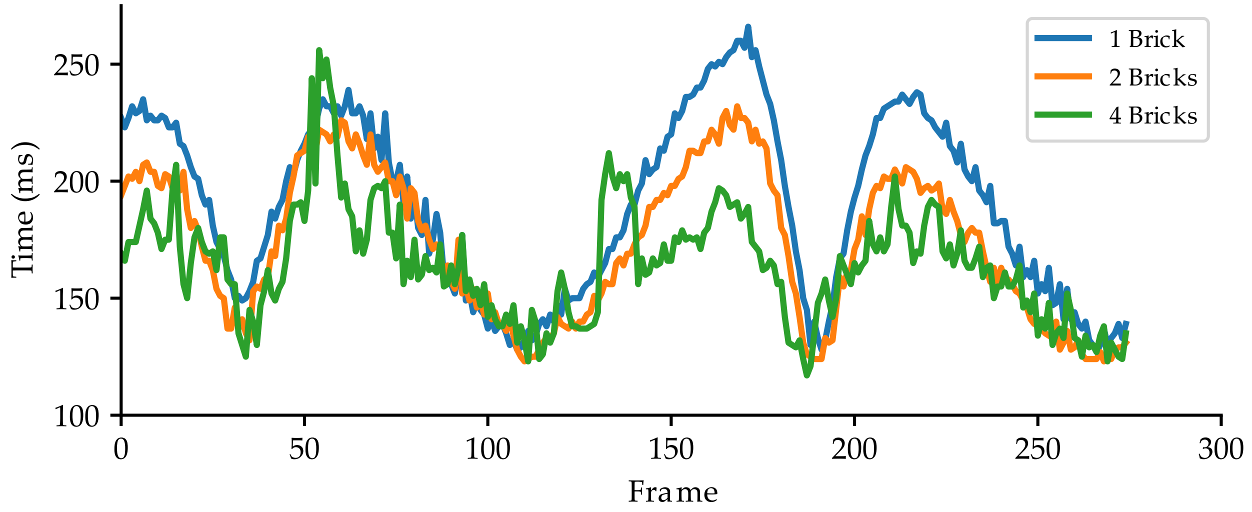
Hybrid-Parallel Rendering Performance

- Partially replicate bricks of data among nodes to improve load balancing
- 64 nodes: 2 bricks per/node
- 128 nodes: 2 or 4 bricks per/node

64 KNL Nodes

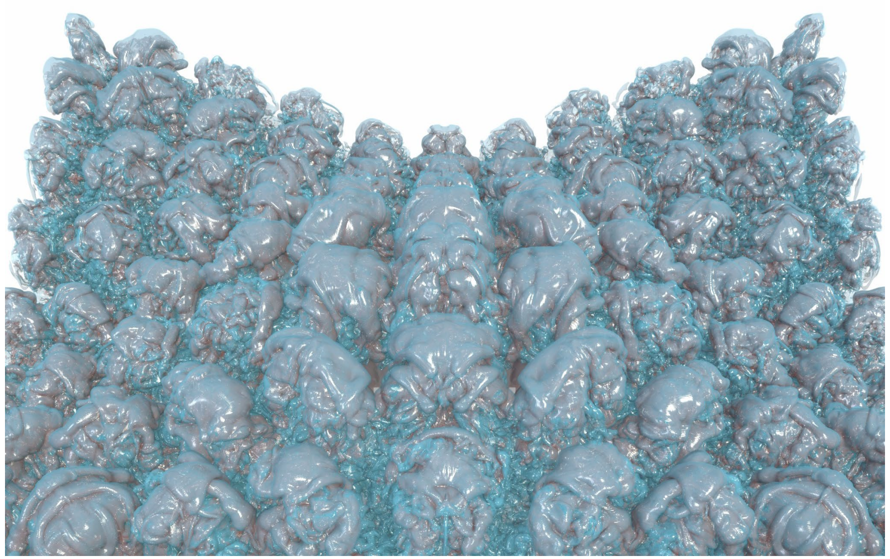


128 KNL Nodes



Thanks!

- DFB and Distributed API out now in OSPRay 1.8.0!



We would like to thank Damon McDougall and Paul Navrátil for assistance investigating MPI performance at TACC and Mengjiao Han for help with the display wall example. 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.

Intel Parallel Computing Centers

www.sci.utah.edu

THE UNIVERSITY OF UTAH®