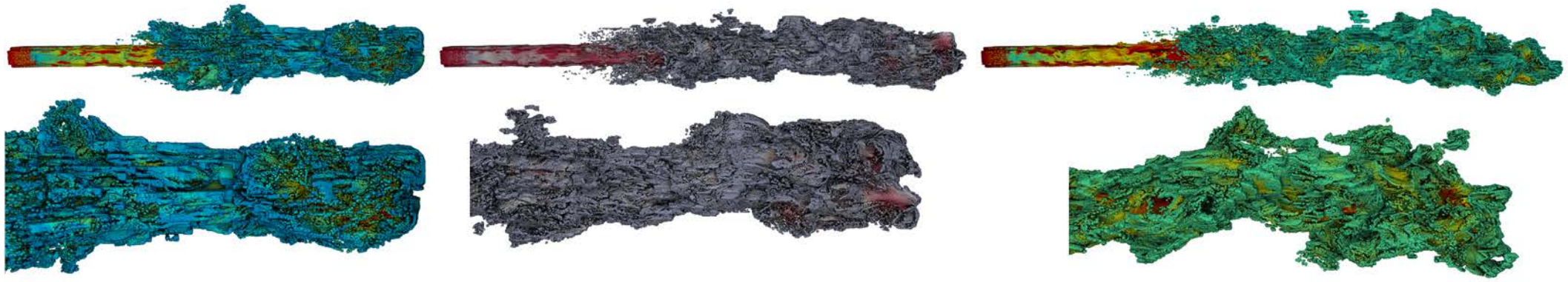


In Situ Exploration of Particle Simulations with Ray Tracing

WILL USHER, INGO WALD, AARON KNOLL, MICHAEL PAPKA, VALERIO PASCUCCI

CARBON CAPTURE MULTIDISCIPLINARY SIMULATION CENTER
SCI www.sci.utah.edu
THE UNIVERSITY OF UTAH®

Related Work

ParaView Catalyst

- May be able to plug pvOSPRay into the VTK pipeline run by Catalyst
- Our module for particle data can also be loaded in pvOSPRay and used in ParaView w/o Catalyst

VisIt LibSim

Rizzi et al.'s interactive vl3 client for LAMMPS [Rizzi et al. 15]

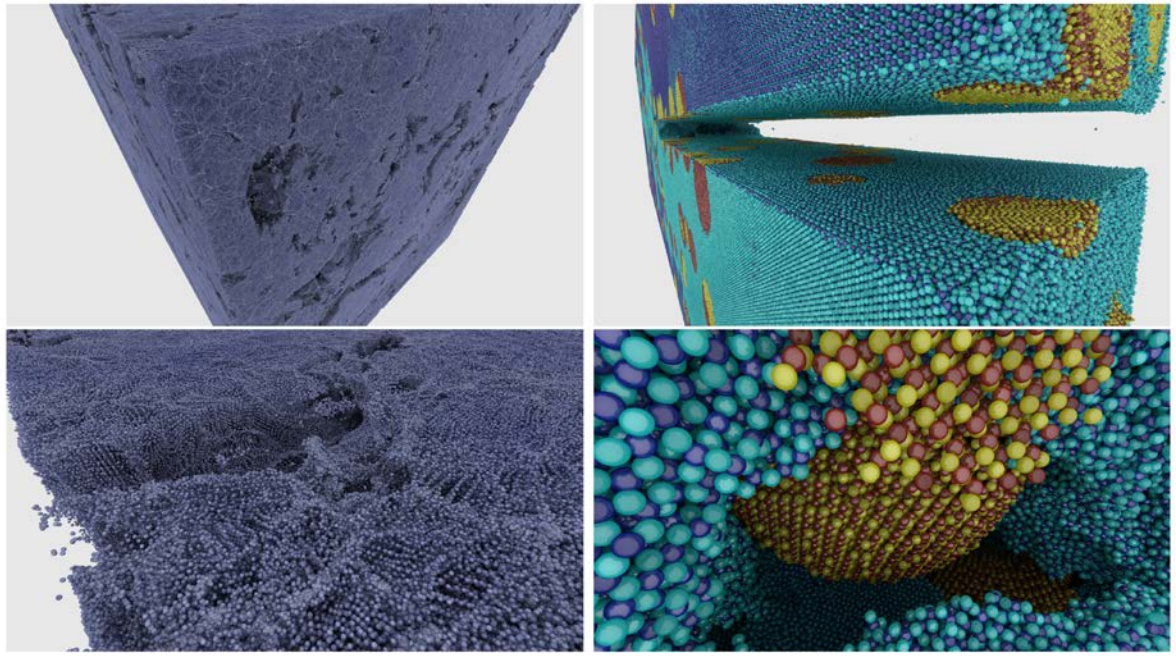
Ellsworth et al.: stream weather forecast data to a rendering cluster and send rendered video frames to clients [Ellsworth et al. 06]

OSPRay and P-k-d Trees

OSPRay is a ray tracing framework for visualization built on Embree and ISPC

Provides tools for writing a distributed renderer

The P-k-d module [Wald et al. 15] provides efficient acceleration structure for ray tracing large particle data



[Wald et al. 15]

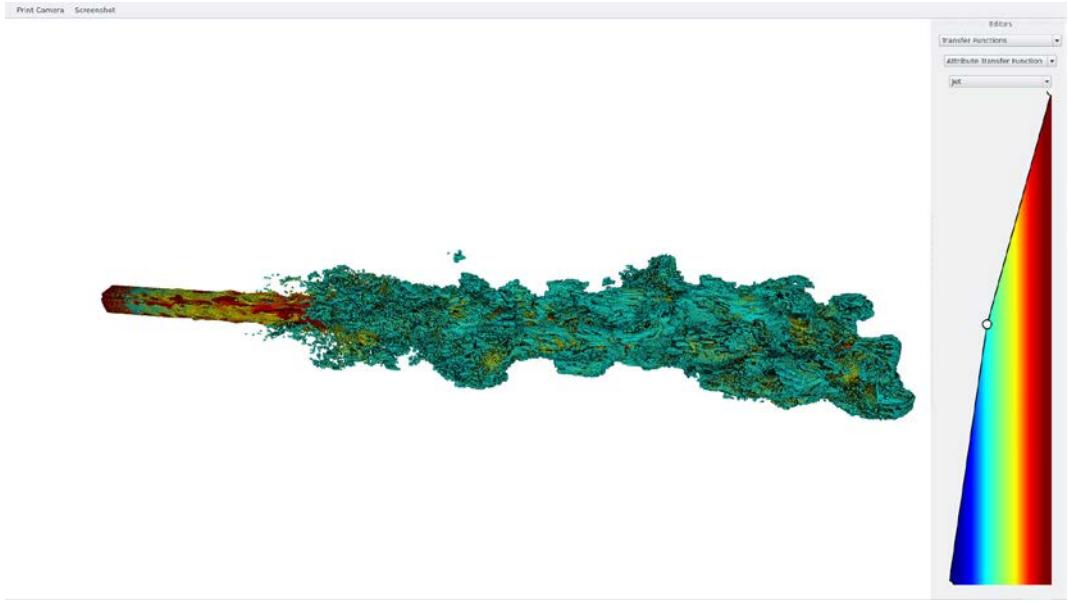
Overview

Communication w/ sim performed by a simulation side server library and renderer side client library

Can connect and disconnect to the simulation at the user's discretion

Requires only CPUs, can run on any compute or visualization HPC resource

Evaluate performance and flexibility with Uintah and LAMMPS simulations



In Situ Library Design

Provide two libraries which communicate simulation data over MPI:

- libIS-sim: Two C-callable functions making for easy integration into existing simulations
- libIS-render: Provides a single function to query a new time step from the simulation

Use a sockets handshake to allow for easy connection & disconnection, when no clients are connected, there is effectively no cost

Simulation-side Library

Simulation acts as a spatially query-able server of the most recent time step via libIS-sim

Two functions to call:

`ospIsInit`: setup MPI in the library and spawn the socket listening thread

`ospIsTimestep`: send particle data for this time step to any clients who've connected

- Open a new MPI communicator to client or re-use existing one

Renderer-side Library

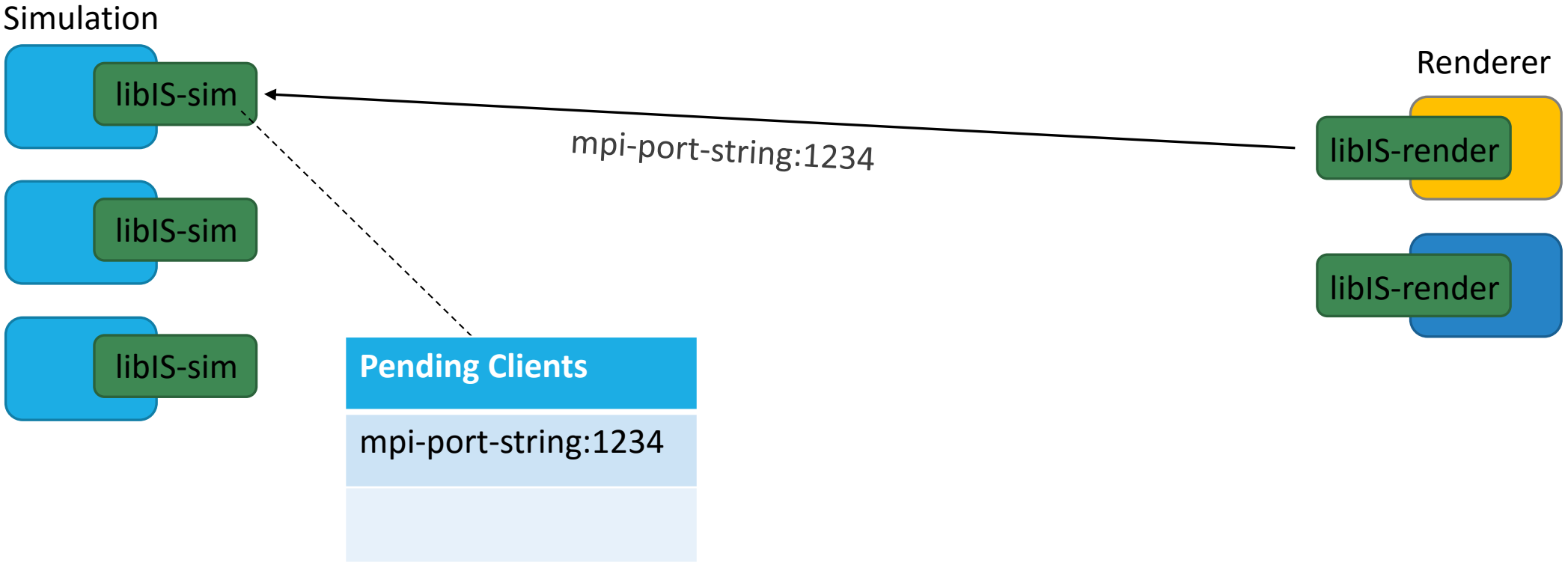
`ospIsPullRequest`: Query libIS-sim to request particles in a worker's domain and return bricks of particles assigned to this worker

Register request to get data by sending our MPI port name to the simulation and do MPI_accept to set up a new communicator

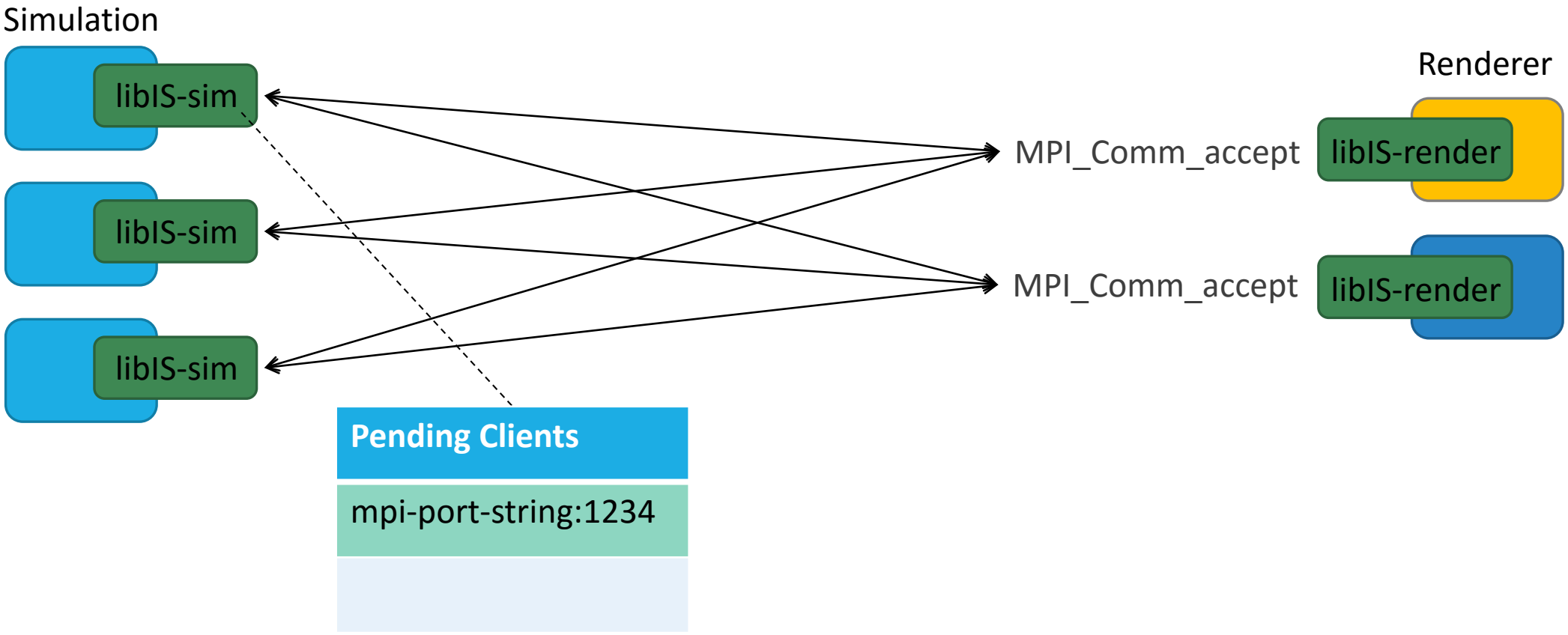
The renderer specifies the data layout for correct compositing, *not* the simulation

Can't guarantee when sharing nodes w/ sim that all transfers use shared memory

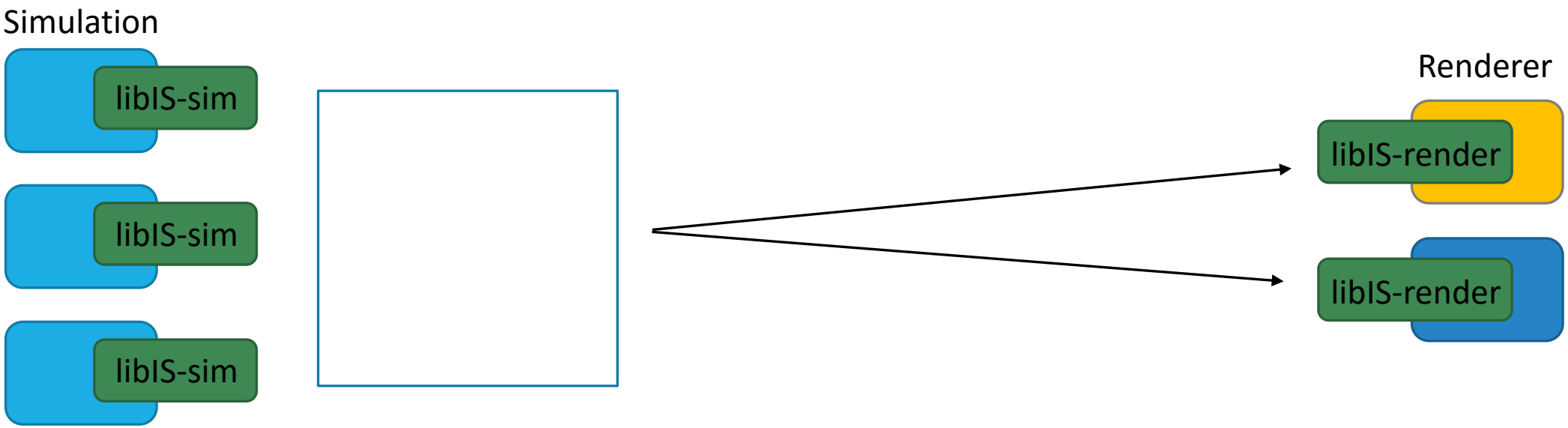
Time Step Query



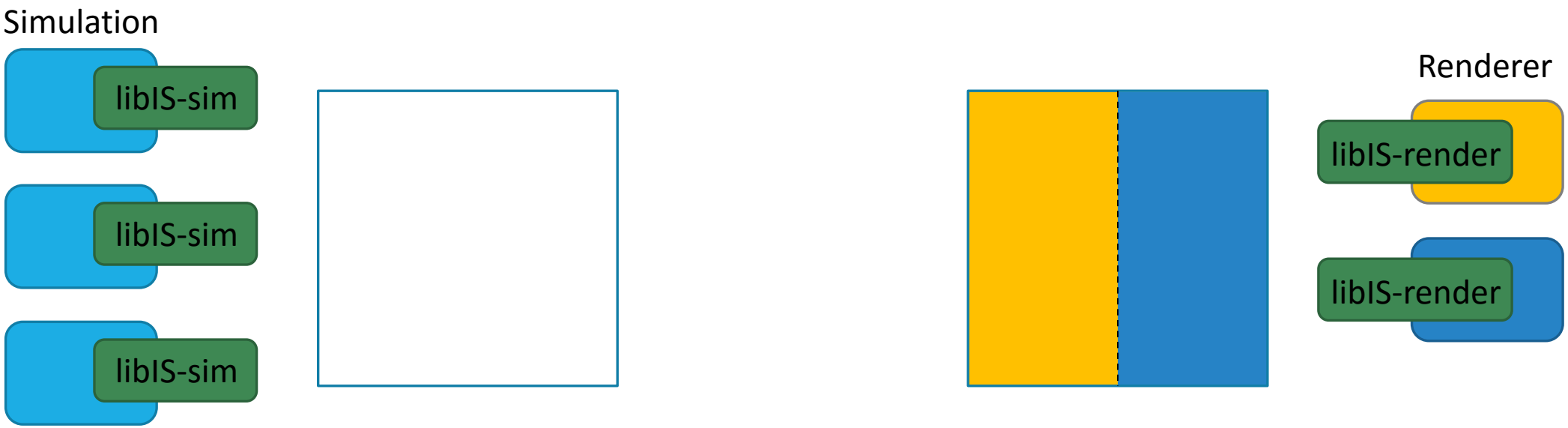
Time Step Query



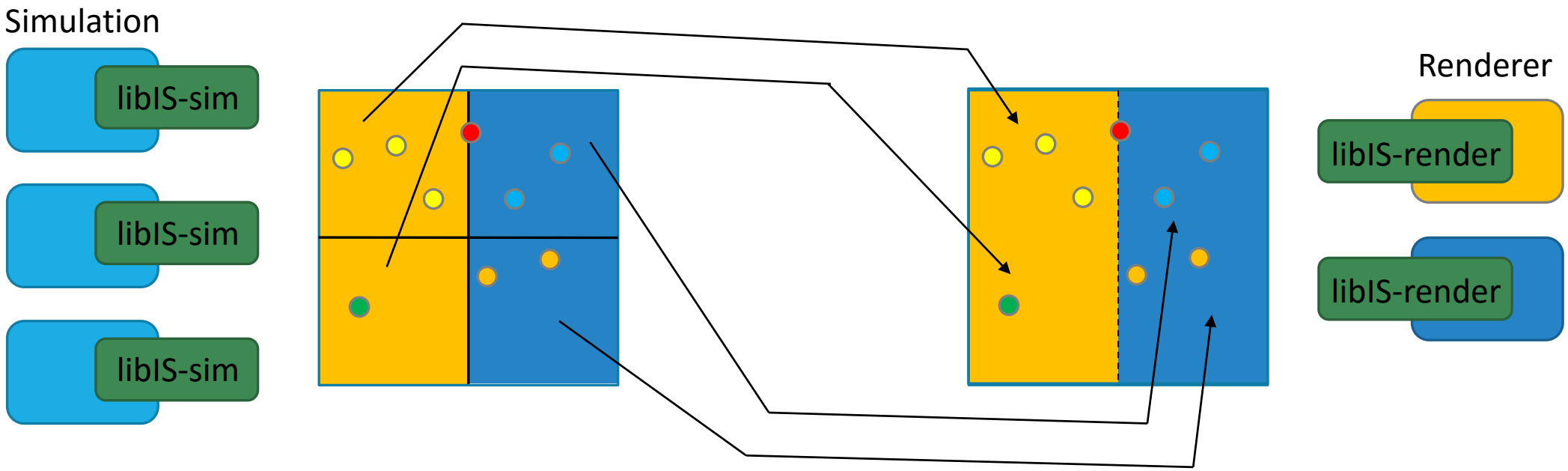
Time Step Query



Time Step Query



Time Step Query



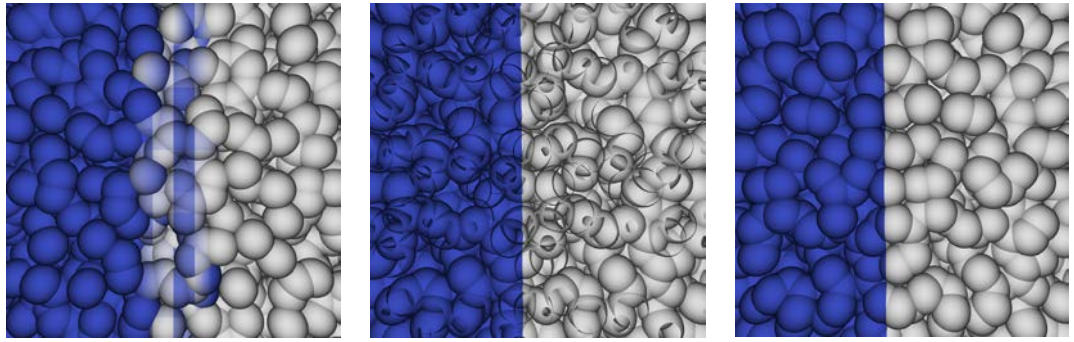
Rendering Client in OSPRay

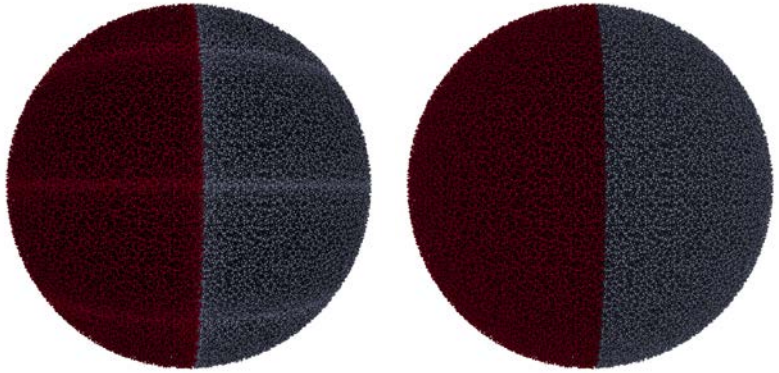
Similar to distributed volume rendering

- World partitioned into convex, disjoint regions assigned to nodes to render
- Build a P-k-d tree on the particles in each region
- Ray trace as normal and use OSPRay's distributed framebuffer for sort-last compositing

Particles whose sphere glyphs overlap the border between workers are clipped at the border with each worker only rendering the portion of the glyph in its region

Since AO is a local effect we just introduce small ghost regions, like AO for distributed volumes [Ancel et al. '12]



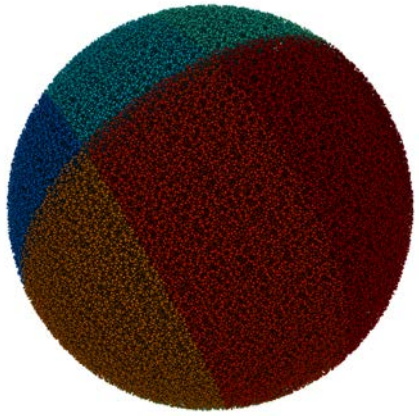


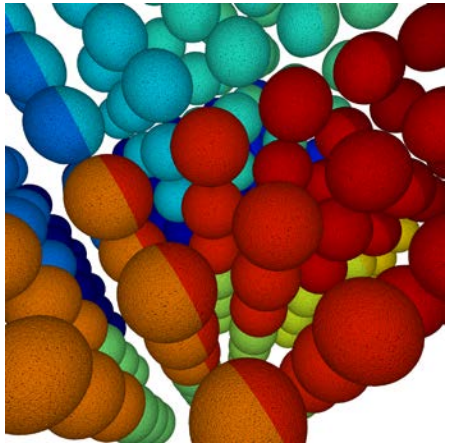
Evaluation

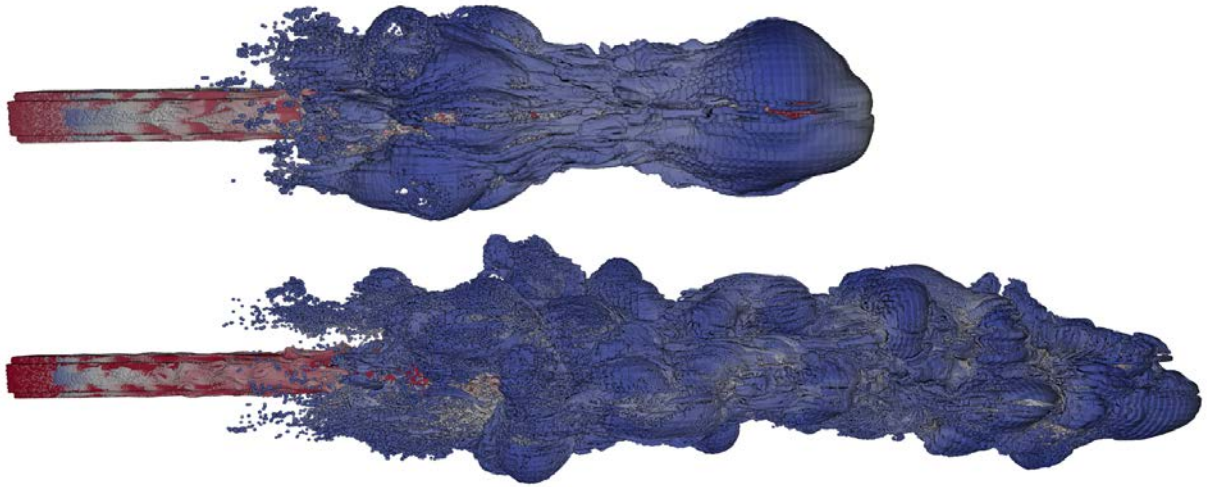
Evaluate with LAMMPS and Uintah simulations on Maverick and Stampede at TACC

LAMMPS: Generated scaling test data sets by replicating ananosphere (1.05M atoms) into grids up to 6x6x6 (227M)

Uintah: A coal particle combustion simulation from Josh McConnell, particles are injected over time, from 34.61M to 55.39M

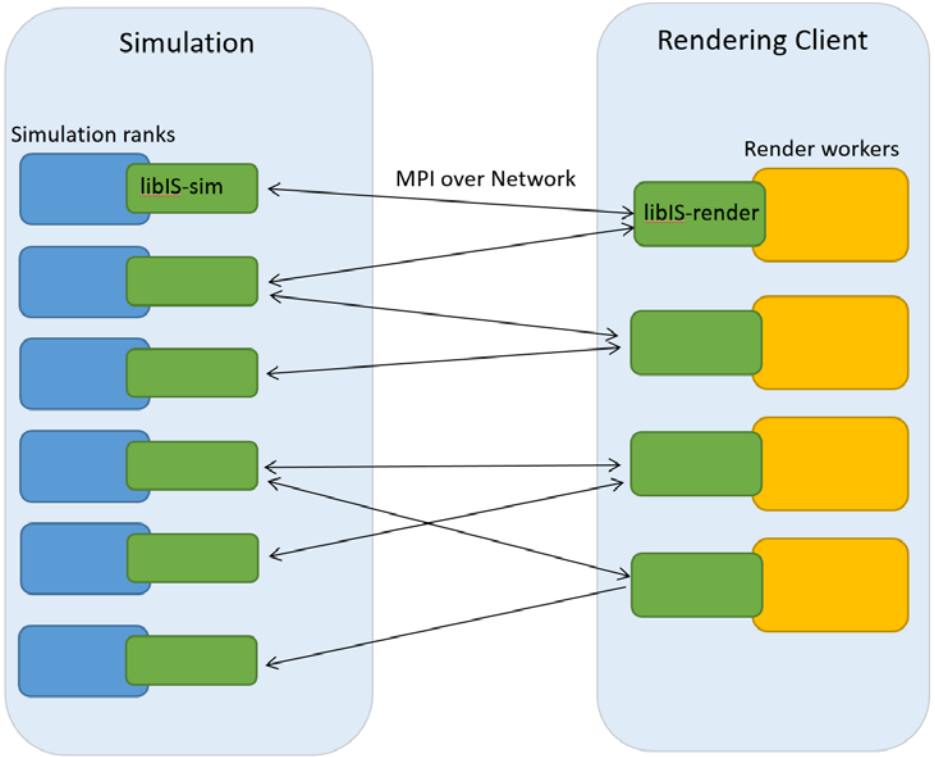






Separate Nodes

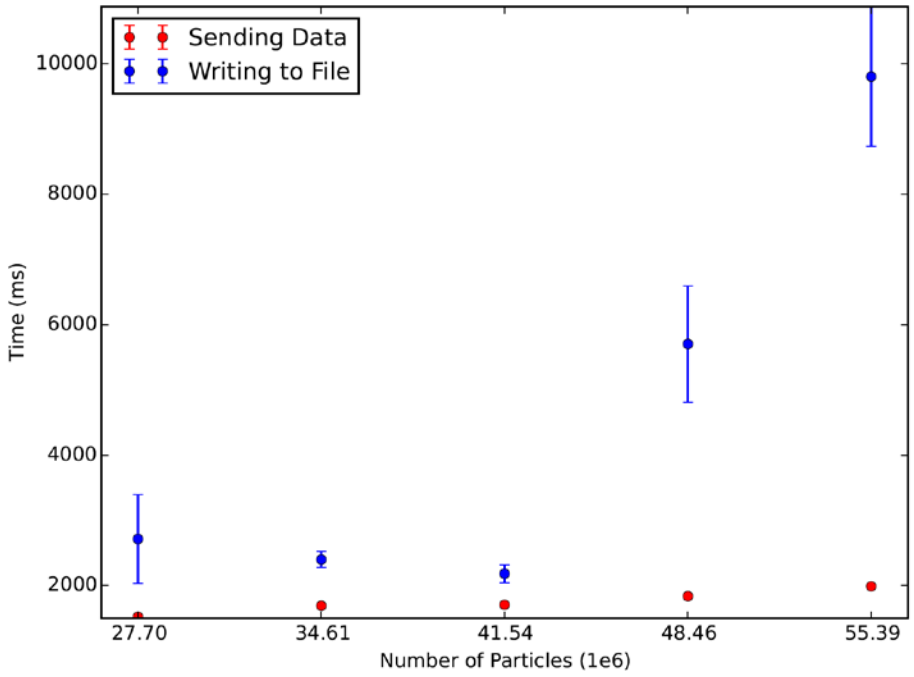
Better rendering and simulation performance vs. sharing nodes but data must go over network



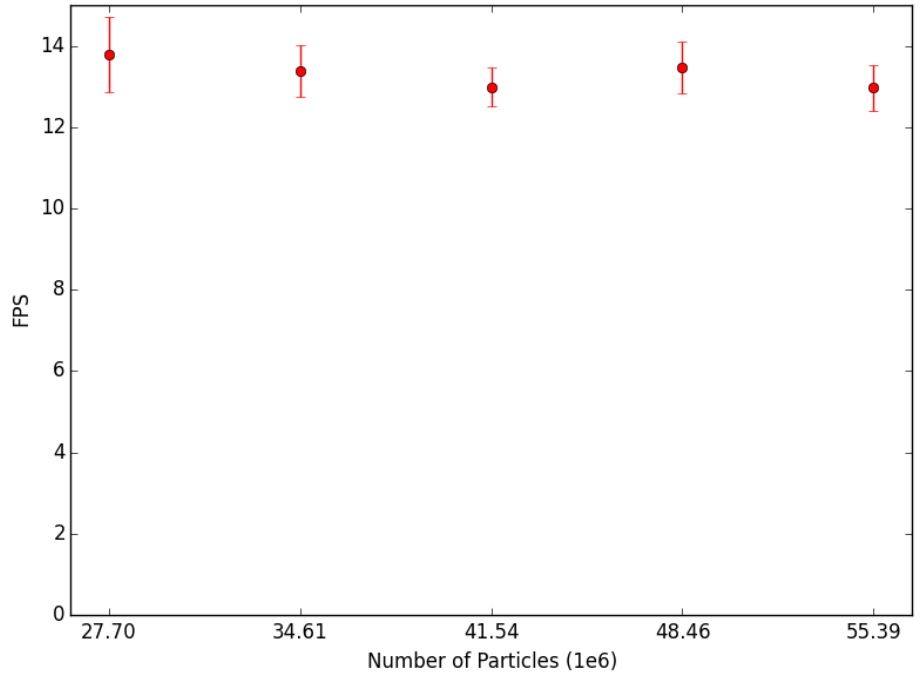
Separate Nodes: Uintah on Stampede

Comparison vs. UDA output: binary files per process + XML metadata

64 Uintah ranks (4 nodes), sending to 12 OSPRay render nodes on Stampede



Sending data vs. writing files



Framerate vs. data size, 1920x1080 framebuffer

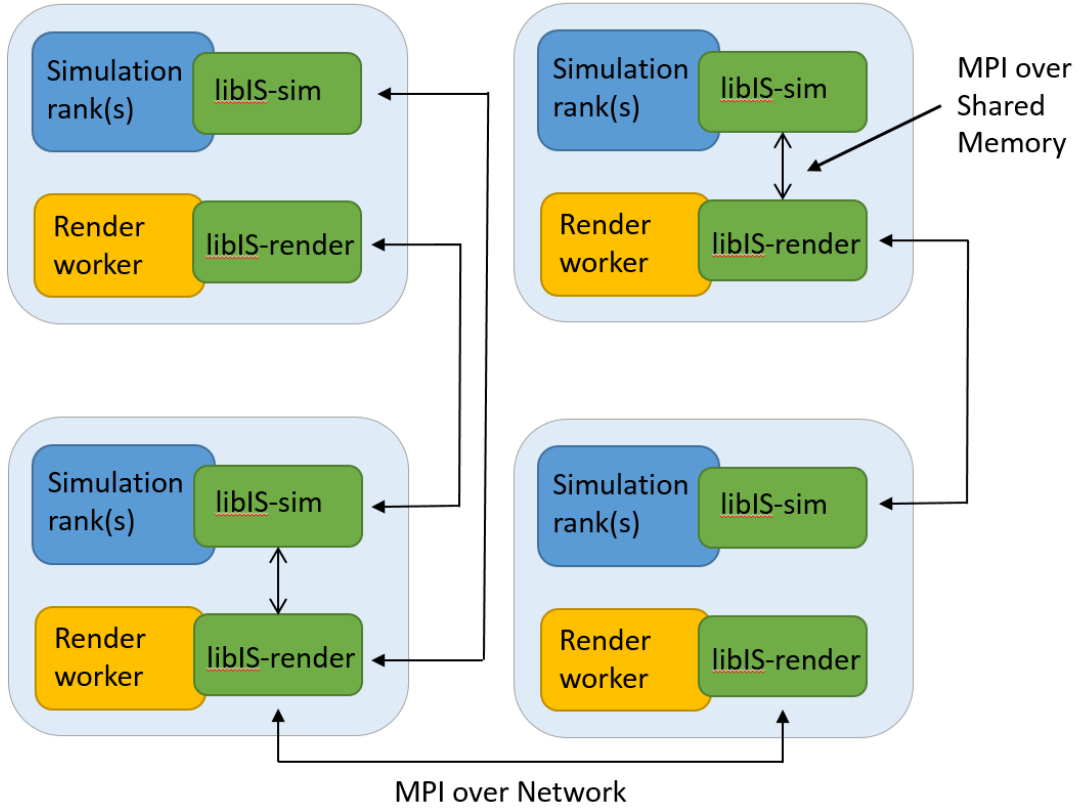
Shared Nodes

Shared memory may be used to transfer data

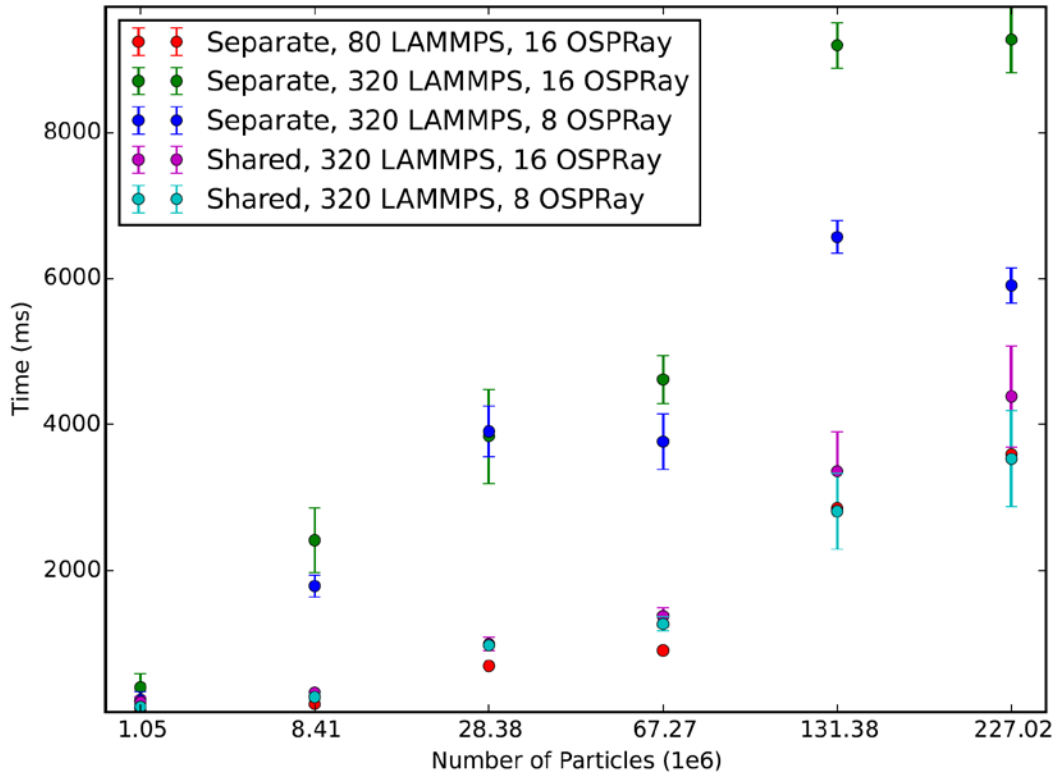
- This is not guaranteed

Only need to allocate at most one extra vis node (or none if rendering to a file)

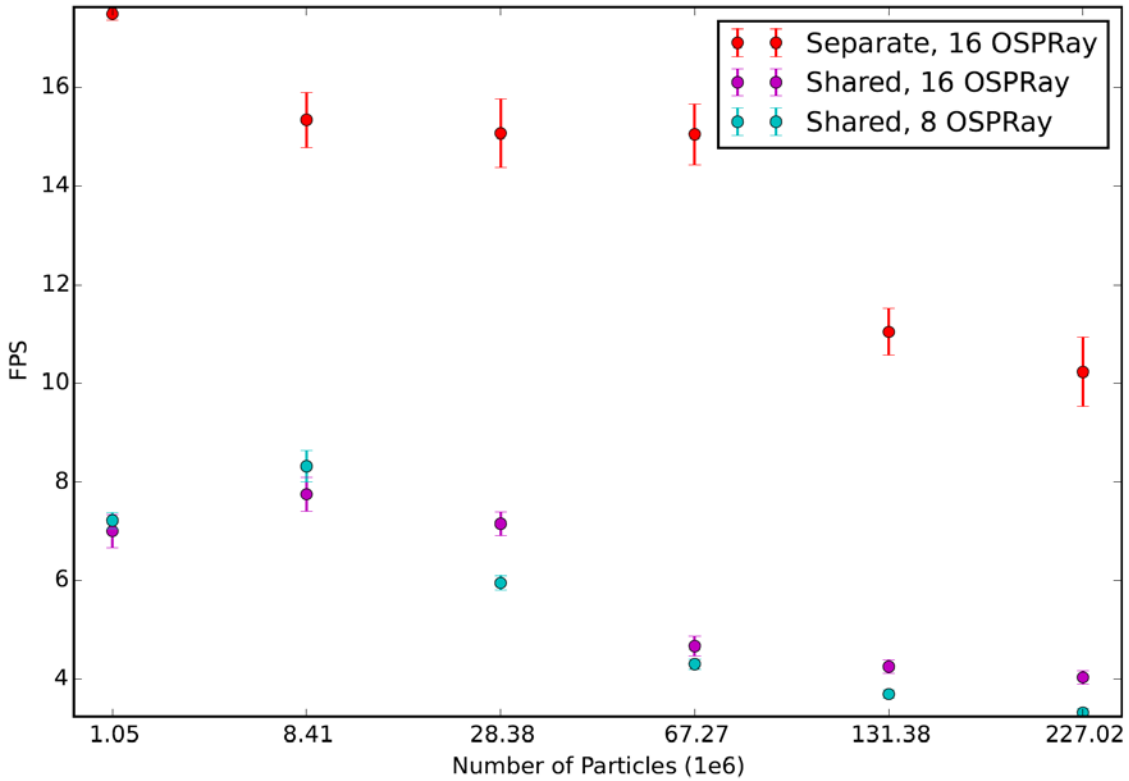
Best for occasionally checking in on a long running simulation



Shared vs. Separate: LAMMPS on Maverick



Shared & Separate send times vs. data size



Shared & Separate framerate vs. data size, 1004x1024 framebuffer

Limitations

When sharing nodes w/ simulation we don't guarantee data transfers will use shared memory

- Key limitation when compared to tightly-coupled work
- Can't assume that the simulation data distribution is suitable for compositing, however it's reasonable to assume some spatial coherence

Have not evaluated our system at very large node counts or data sizes

HPC batch job workflow makes interactive techniques difficult to use in practice, we hope our connect/disconnect functionality and shared node option lowers barriers to use

Future Work

Support for volume data enabling in situ exploration of mixed volume & particle simulations with our renderer

View dependent data querying and techniques from scalable IO frameworks

Incorporate techniques from large scale compositing-based renderers to scale to higher node counts

Thank You!

University of Utah PSAAP II Center, CCMSC DE-NA0002375

Intel Parallel Computing Center at the University of Utah

will@sci.utah.edu

@_wusher

Simulation-side Library

```
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    // Duplicate the MPI communicator in libIS
    ospIsInit(MPI_COMM_WORLD);
    // Particles is an array of [x, y, z, attrib] for example
    std::vector<float> particle = initial_conditions();
    for (size_t step = 0; step < NUM_STEPS; ++step){
        step_simulation(particles, step);
        // Tell libIS a timestep is ready for clients
        // Pass total # of floats in particle array, ptr to particle data
        // and stride for each particle
        ospIsTimeStep(particles.size(), particle.data(), 4);
    }
    return 0;
}
```

Renderer-side Library

```
int main(int argc, char **argv){
    MPI_Init(&argc, &argv);
    vec3i grid(2, 2, 2);
    float ghostSize = 0.01;
    // Connect to the simulation at argv[1] port argv[2] (logged by libIS-sim)
    DomainGrid *domains = ospIsPullRequest(MPI_COMM_WORLD, argv[1],
                                           atoi(argv[2]), grid, ghostSize);
    // Domains are owned by ranks of the renderer and distributed, each rank
    // is assigned a set of cells in the grid to render, getMine returns such a
    // block along w/ the particles and its bounds
    for (size_t i = 0; i < domains->numMine(); ++i){
        render(domains->getMine(i));
        // Perform compositing of the worker's results
    }
    return 0;
}
```