DOI: 10.14529/jsfi160401

In Situ Exploration of Particle Simulations with CPU Ray Tracing

Will Usher[1] ***Ingo Wald***[2] ***Aaron Knoll***[1] ***Michael Papka***[3] ***Valerio Pascucci***[1]

We present a system for interactive in situ visualization of large particle simulations, suitable for general CPU-based HPC architectures. As simulations grow in scale, in situ methods are needed to alleviate IO bottlenecks and visualize data at full spatio-temporal resolution. We use a lightweight loosely-coupled layer serving distributed data from the simulation to a data-parallel renderer running in separate processes. Leveraging the OSPRay ray tracing framework for visualization and balanced P-k-d trees, we can render simulation data in real-time, as they arrive, with negligible memory overhead. This flexible solution allows users to perform exploratory in situ visualization on the same computational resources as the simulation code, on dedicated visualization clusters or remote workstations, via a standalone rendering client that can be connected or disconnected as needed. We evaluate this system on simulations with up to 227M particles in the LAMMPS and Uintah computational frameworks, and show that our approach provides many of the advantages of tightly-coupled systems, with the flexibility to render on a wide variety of remote and co-processing resources.

Keywords: in situ rendering, parallel systems, point-based data, CPU and GPU clusters.

Introduction

In the coming era of exascale computing, simulations will produce data far in excess of what can be effectively archived in parallel file systems. Although numerous solutions exist for addressing IO bottlenecks at scale, including filtering of data attributes, sacrificing temporal or spatial resolution, or compression [3, 17], these approaches often assume *a priori* knowledge of the data and sacrifice some flexibility. Ultimately, to enable the broadest exploratory analysis of unadulterated simulation data at scale, visualization software must adapt to *in situ* use cases in which data are not stored to disk, and analysis and, potentially, rendering occur while the simulation is running. In situ visualization can add additional compute cost to the simulation which may change its performance characteristics. However, it comes with the benefit of reducing or eliminating time spent in file IO.

In situ is often used as an umbrella term encompassing numerous different approaches currently being classified in an effort led by Childs [4]. Much in situ research has emphasized sidestepping IO through specific analysis, data reduction or filtering [24, 36], optimizations to existing IO frameworks enabling scalable co-processing, streaming or offline visualization [16, 18, 33], and data forwarding mechanisms coupling simulations with production visualization software [6]. Due to the nature of large-scale simulations, most in situ approaches are designed to operate in batches. Relatively fewer in situ applications have targeted interactive use, enabling live, *exploratory* visualization from simulation. Here, one has the choice of *tightly-coupled* visualization embedded directly in the simulation code and running on the compute resource [37], or asynchronous *loosely-coupled* approaches that forward data from the simulation into separate visualization processes (e.g. [27]), either on the same machine or a different cluster. In either case, in situ rendering or analysis requires consideration of distributed data spread

[1]SCI Institute, University of Utah
[2]Intel Corporation
[3]Argonne National Laboratory

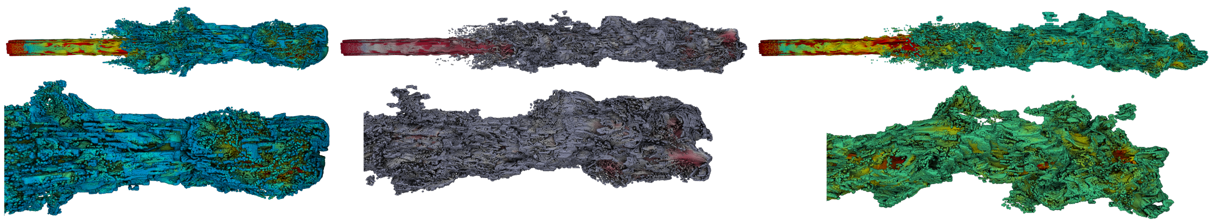


Figure 1. A coal particle combustion simulation in Uintah at three different timesteps with (left to right): 34.61M, 48.46M and 55.39M particles, with attribute based culling showing the full jet (top) and the front in detail (bottom). Using our in situ library to query and send data to our rendering client in OSPRay these images are rendered interactively with ambient occlusion, averaging around 13 FPS at 1920 × 1080. The renderer is run on 12 nodes of the Stampede supercomputer and pulls data from a Uintah simulation running on 64 processes (4 nodes). Our loosely-coupled in situ approach allows for live exploration at the full temporal fidelity of the simulation, without prohibitive IO cost

across multiple simulation nodes and potentially too large to be marshaled to a single node for processing.

In this paper we describe a system for interactive in situ exploration of particle data. Our system employs a loosely-coupled or in-transit approach, but retains many of the advantages of tightly-coupled methods. Using the OSPRay ray tracing framework for visualization [34], the system can run natively on CPUs on either compute or visualization resources, requiring no dedicated hardware for visualization. Leveraging memory-efficient approaches for particle ray tracing [35], our approach requires minimal overhead for geometry and acceleration structures. Moreover, to the best of our knowledge, this system represents the first utilization of an interactive ray tracer for in situ visualization. Our key contributions are:

- An interactive in situ rendering client that can connect and disconnect to the simulation at the user's discretion, minimizing the impact of the renderer on the simulation and enabling live exploration of the simulation state.
- A flexible in situ layer integrated with OSPRay which enables the renderer to run on the same nodes as the simulation or asynchronously on a different resource.
- Pairing in situ data query with memory-efficient ray tracing data structures and mechanisms for direct rendering and filtering of particle data.

We demonstrate the flexibility and performance of our system with two simulations, LAMMPS [25, 29] and Uintah [2]; deployed on a NUMA workstation, a visualization cluster (Maverick) and compute resource (Stampede). We evaluate the communication characteristics and scalability of our system across these different resources. Though exploratory in situ poses numerous human and logistical obstacles, it is ultimately desirable for users to be able to explore simulations as they run.

1. Background and Previous Work

1.1. In Situ Analysis

As simulations grow in scale, in situ analysis and data reduction have become popular tools in the computation-visualization workflow. Generally, these analyses are designed to operate alongside computation in batches. For example, Woodring et al. [36] sample an interesting subset of data to save during the simulation. Peterka et al. and Zhang et al. compute and save out derived data from the simulation for further analysis instead of the raw simulation timesteps, reducing IO requirements [23, 38].

Various methods have also been proposed to couple the analysis and simulation depending on how much modification of the simulation code is desired. Peterka et al. [23] integrate a distributed parallel Voronoi tessellation into the HACC cosmology simulation enabling this meshing to be performed more efficiently than as a post-process. Zhang et al. propose a less tightly coupled approach and share data between the simulation and a distributed feature tracking algorithm using an on node data staging process [38]. Fabian et al. design adaptors for integration into existing simulations which hand off to a ParaView coprocessing API, allowing for a variety of algorithms to be implemented without requiring additional modification of the simulation for each algorithm [6].

GLEAN provides a flexible framework for coupling analysis and can be called directly by the simulation or by embedding into existing higher-level IO libraries [32]. In an effort to require no modification to existing simulation code Fogal et al. [7] intercept calls to IO libraries and pass the data on to visualization tools. This approach allows for easier development of in situ analysis as no changes to the simulation code are required.

In situ batch style analysis is desirable as a way to produce analysis products at a higher spatio-temporal resolution, but the user must know *a priori* what analysis products are interesting to compute. For exploratory analysis direct in situ visualization is still desirable, and in fact some of the works mentioned previously provide a live rendering component [23, 32] or allow modification of analysis parameters on the fly [6].

1.2. In Situ Visualization

In situ visualization has long been proposed as an alternative to offline visualization, first discussed by McCormick et al. [20]. Numerous visualization and analysis tools have been written to run tightly or loosely coupled with a simulation depending on the requirements of the application and how much modification of the simulation was desired. For example, SCIRun [22] and pV3 [13] were designed to directly integrate with the simulation and allow for computational steering. VMD [15] was initially conceived as a visualization counterpart for NAMD.

Tu et al. proposed an end-to-end approach which tightly couples all components of a simulation pipeline from meshing to visualization [31]. A loosely coupled in situ renderer for weather forecast models is described by Ellsworth et al. [5] where simulation data is sent over the network to a separate rendering cluster, minimizing the impact of the rendering system on the simulation. Rizzi et al. [27] interconnect LAMMPS and vl3 in a similar manner to our work and run a one-to-one mapping of render nodes to simulation nodes. This approach allows for shared memory to be used instead of network transfers when sending data from the simulation to the renderer. Currently, vl3 relies on OpenGL and GPUs for efficient rendering and compositing.

Most similar to our work, Yu et al. [37] directly couple a software renderer to S3D for in situ rendering of mixed particle and volume data. While directly integrating the renderer into the simulation removes the IO cost of communicating between the processes, the renderer can now only provide a new frame every timestep, limiting interactivity. Relatively few in situ papers have emphasized direct in situ visualization, instead focusing on a mix of analysis and simulation steering as well as rendering [6, 22]. Our system is unique not only in that it employs CPU ray tracing for better memory-efficiency and platform portability, but in that it fosters more *direct* visualization through a lightweight rendering client which can be trivially connected or disconnected to the simulation as needed.

1.3. Particle Visualization

Point data rendering has been widely explored in graphics and visualization [10]. In this paper we are primarily interested in volumetric particle data coming from scientific simulations, where the particles fill some space and have one or more attributes (temperature, atom type, pressure, strain, etc.). Common approaches for rendering these data can be roughly split into glyph, volumetric and implicit surface approaches.

Glyph techniques represent the point with some non-singular object such as a sphere, cube, or arrow. Various techniques have been proposed for efficiently ray tracing millions of opaque sphere glyphs [9, 35] on CPU. On GPU, MegaMol [12] combines rasterization, ray casting of sphere glyphs and image space filtering to render millions of atoms. By applying LOD and out-of-core techniques Fraedrich et al. implemented an extremely fast out-of-core LOD particle renderer for real-time rendering of astro-physics data with billions of particles [8].

1.4. Data Parallel Rendering

Data parallel rendering in the context of rendering large volume datasets has been widely studied, producing a body of work covering the data distribution, rendering and compositing to form the final image. Early work by Hsu and Ma et al. [14, 19] partition the volume among the processors for rendering then composite the resulting partial images with direct send or binary swap to compute the complete frame. Recent work has focused on scaling up to large numbers of nodes where compositing becomes the bottleneck [11, 21].

While particle rendering has been long explored on the CPU and GPU, to our knowledge there has not been as much work in a distributed data setting. Recently and most similar to our work, Rizzi et al. perform in situ distributed rendering of LAMMPS data using sphere glyphs with the vl3 framework [27]. Our own work builds on the balanced P-k-d method of Wald et al. [35], which has demonstrated interactive direct ray tracing of up to 30 billion particles on a single workstation. Though the P-k-d was not originally deployed in a data-parallel setting, we show how it can be extended to distributed data applications as well. In our renderer we build on an existing data parallel renderer in the OSPRay ray tracing system (next section), subject to modifications to make it suitable for handling particle data and indirect shading effects such as ambient occlusion (see section 3.2.1).

1.5. OSPRay

OSPRay [34] is a ray tracing framework for visualization on CPUs, which provides existing visualization tools an efficient ray tracing API for rendering. OSPRay allows for interactive rendering on compute resources that do not have GPUs, and supports advanced shading effects, large models, non-polygonal primitives, etc.

Stock OSPRay already supports efficient techniques for rendering large numbers of particles [35], and provides some infrastructure for MPI-parallel and data-parallel volume ray casting. Though a complete description of OSPRay's data-parallel renderer is beyond the scope of this paper, in general its implementation distributes "bricks" of volume data as well as "tiles" of the frame buffer among the render nodes. Each node then iterates over all the bricks of volume data it owns, and renders all the tiles required for this brick. Each such brick-specific tile is then sent to the node that owns the corresponding region of the frame buffer, which composites it with the other bricks' tiles computed by other nodes. Conceptually this rendering technique is similar to Hsu et al.'s *Segmented Ray Casting* [14]. As a sort-last compositing approach, this technique is well suited to large data bricks because it communicates neither data nor rays, but is inherently designed for simple shading of primary rays, not ray tracing.

2. In Situ Data Handling and Live Connection

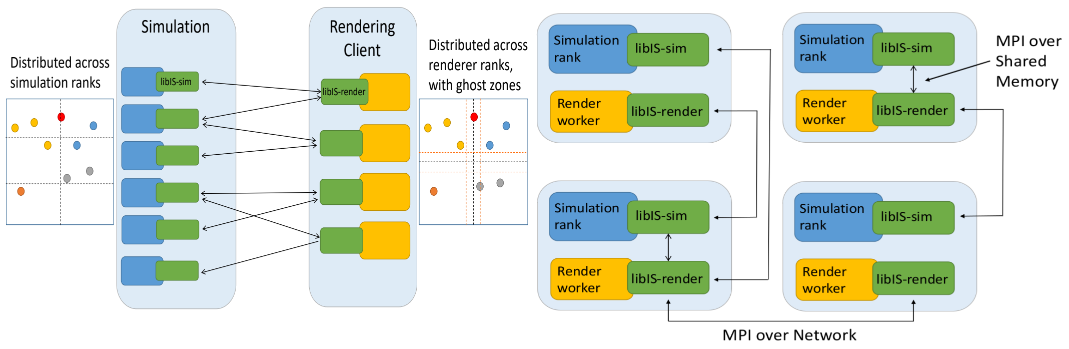


Figure 2. Overview of the in situ library, libIS. In our system, data are forwarded via MPI from the simulation to a distributed renderer. When the simulation and visualization are run on different resources (left) all data queries go over the network. When running on the same resource (right), data can be forwarded locally from one process to another on the same physical machine via shared memory or transferred over the network

To access timesteps as they are produced by the simulation a visualization client queries data through a library linked into the simulation code and client code, which communicate over MPI. By extracting a lightweight, simple to use API from these libraries we make it easy to integrate them into existing simulation and rendering codes.

The loose coupling of the simulation-side and render-side libraries to each other allows for a variety of configurations of the simulation and in situ client processes. The client can be run on the simulation nodes, a separate vis cluster or on a single workstation, as illustrated in Figure 2. Since the rendering client and simulation can have different scaling qualities this flexibility allows for each to be run in the most favorable configuration, and lets us adjust the simulation's data layout to be suitable for distributed rendering.

With a simple sockets handshake mechanism, end users can easily connect to and disconnect from the simulation at will, allowing them to easily connect and interactively explore the simulation at any time. This approach may be preferable to tightly-coupled in situ methods that require visualization parameters and output to be specified a priori before running a batch job. Moreover, when no OSPRay clients are connected to the simulation, no data are communicated over the networkl; employing the in situ library effectively incurs no cost when not in use.

2.1. Simulation-side Library

The simulation side library libIS-sim acts as a spatially queryable server of the most recent timestep from the simulation, allowing clients to pull blocks of data as desired. The library exports two C–callable functions, making it available for integration into simulations written in almost any language. The simulation first initializes the library by calling `ospIsInit` which will perform some one time setup for MPI communication in the library and launch a background polling thread to watch for new clients. The simulation can then call `ospIsTimestep` each timestep, passing the list of particles for the current timestep when data is ready to be sent to clients.

The first rank on the client side connects to the polling thread on the first rank of the simulation over sockets and sends its MPI port name, which will be used to uniquely identify the client. The simulation side library maintains a list of clients who have requested a timestep during computation, and when a timestep is ready will connect to each client so it can request parts of the data.

When the simulation is ready to send a timestep it traverses the list of clients and responds to queries. Using the port name stored previously, the simulation and client either set up a new MPI communicator or reuse a previously created one. The library then sends each client the simulation world bounds so it can request regions of data in the simulation space. Each client process sends back a list of boxes bounding the regions it wants particle data within. To satisfy the query each simulation rank finds which particles it has in these boxes and sends them to the client. In order to reduce the amount of data transferred the simulation only sends the particle positions and the single particle attribute being displayed in the renderer.

2.2. Rendering-side Library

The rendering-side library libIS-render acts as a counterpart to the simulation library, allowing render processes to request regions of the current timestep from the simulation. The rendering side library provides a single function, `ospIsPullRequest`, which is called when the client wants to request a new timestep from the simulation. As mentioned previously the client will send its MPI port name and wait for a connection back from the simulation, either on a new communicator or one created previously. This allows for easy disconnection from the simulation by simply not requesting a new timestep, and easy reconnection since a client just needs to send its new MPI port name over to request data.

After getting the world bounds from the simulation the client process computes a grid that partitions the world among the ranks. Each rank finds the bounds of its boxes in the world and requests them the simulation, getting back a list of particles to render. Each node may also extend its bounds by some ghost region, resulting in overlap between ranks. The ghost region size can be zero if no overlap is needed. However, in our renderer some duplication is required to properly render particles at the boundary of two nodes and compute ambient occlusion. After getting the list of particles from the simulation `ospIsPullRequest` returns the list of boxes and the particles contained within to the caller.

3. Rendering

Using the library described above we implement an interactive in situ particle renderer as an OSPRay module. The module makes use of the memory-efficient balanced P-k-d tree [35] to render the particle and extends it to make this acceleration structure suitable for data parallel rendering. OSPRay is primarily designed for static geometry that is updated externally through the OSPRay API and difficulties arise in this case where the geometry is internally responsible for querying the next timestep and updating itself.

3.1. Rendering Client

The rendering client is designed for a distributed environment and is split into two parts: an OSPRay geometry module responsible for querying and rendering data, and an OSPRay scene graph module which instantiates this geometry in OSPRay's interactive viewer.

The master process is an interactive viewer which displays frames from the render workers, and allows the user to move the camera and edit transfer functions. We add a new node to the scene graph used by this viewer which instantiates the geometry provided by our geometry module. This node also launches a background thread to receive the updated world bounds from the first worker process so we can synchronize when the workers switch to the new data.

The geometry module uses libIS-render to pull data from the simulation. When the geometry is first added to the scene we perform a blocking query to get an initial timestep, then continue to query asynchronously. After each node has built a P-k-d tree for the first timestep it spawns a background thread to request data from the simulation at some user set frequency. This thread pulls a new timestep and builds a P-k-d tree on the particles to prepare it to be swapped with the current data. Once each worker's data is ready the first rank informs the master process running the viewer that the geometry in the world should be updated by sending it the new world bounds, at which time the viewer re-commits the geometry, synchronizing when the workers swap to the new data. This approach is similar to ping-pong buffer and texture strategies in GPU renderers, which take care to avoid trampling data that is in flight in the rendering pipeline.

3.2. Rendering on a Shared-Memory Workstation

If a workstation with enough memory is available, a simple option is to run the renderer on it and pull data from a simulation running on the same machine or a remote cluster. Since OSPRay internally uses multiple threads for rendering, only two processes are required: one to display the interactive viewer, and another to poll for updates from the simulation and render the data. If the simulation is running on the same workstation as the renderer MPI will use shared memory to transfer data between the processes, minimizing data transfer cost.

3.2.1. Data-Distributed Rendering

OSPRay includes functionality for writing a distributed renderer and includes a data distributed volume renderer (described in [34]) the design of our renderer is motivated by this volume renderer and takes a similar form. While the P-k-d tree is well suited to rendering particles on a single node, it provides no functionality for data parallel rendering across multiple processes. Since our particle data is volumetric in nature we can take a similar approach to volume rendering and render "bricks" of P-k-d trees. This fits well with libIS-render, which returns a list of blocks that the node is responsible for rendering. We build a P-k-d on each of these bricks and then proceed similarly to a distributed volume renderer. Each node renders its bricks of particle data and then performs sort-last compositing using OSPRay's distributed framebuffer to compute the final image.

Since particles are represented with sphere glyphs it's possible for a node's glyphs to overlap with another node's domain resulting in non-disjoint regions of data, leading to incorrect com-

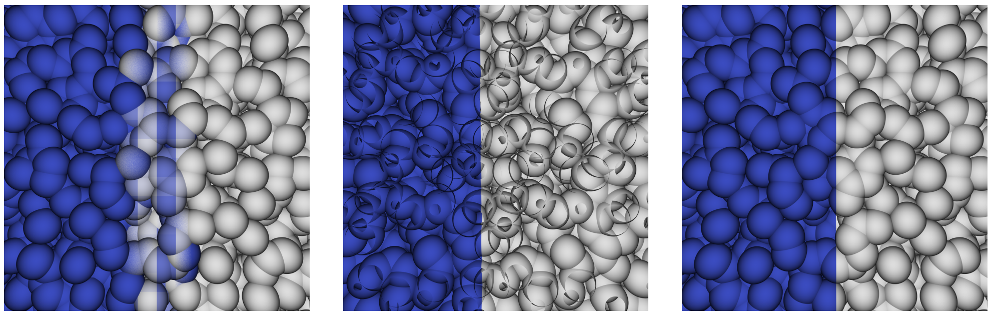


Figure 3. Without clipping nodes render overlapping regions, resulting in compositing artifacts (left). Always clipping rays against the node's domain (middle) incorrectly clips spheres on unshared boundaries. By extending unshared boundaries of the domain and clipping primary rays to the partially extended bounds both compositing and clipping artifacts can be avoided (right)

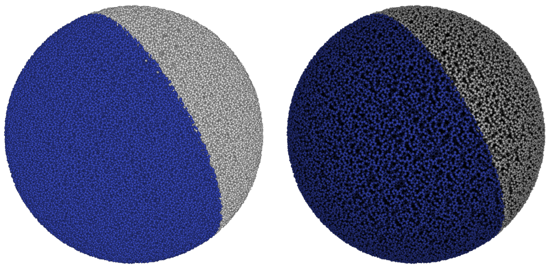


Figure 4. Depth perception of raycasting (left) vs. ambient occlusion (right) on a 1M atom nanosphere

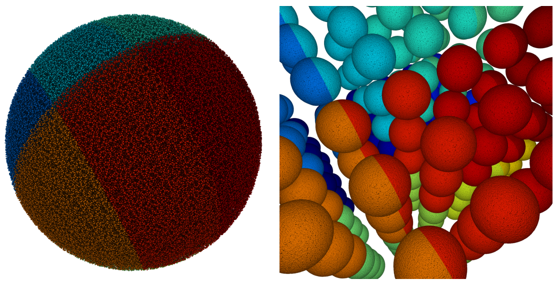


Figure 5. Replicated nanosphere scaling datasets used in LAMMPS with 1.05M (left) to 227M (right) atoms

positing as seen in fig. 3. This is corrected by clipping primary rays against the node's domain to only find hits within the assigned domain. If a sphere straddles the split between two workers each is responsible for rendering just the piece in their domain.

To aid depth perception of the simulation data we render the data with ambient occlusion, which adds local shadowing effects. Ambient occlusion is especially useful in large or dense particle datasets where it becomes challenging to determine the position of particles relative to each other [30], also see fig. 4. In the case of distributed data some particles on a node may need data from another node to properly determine this occlusion term. Since the shadowing effect is local distant particles don't shadow each other so we can solve this without introducing much overhead by adding small ghost regions to each node, similar to Ancel et al.'s approach for volume data [1].

Raycasting: In extremely memory constrained environments, where the small overhead needed to compute ambient occlusion on the data is not available, we can fall back to raycasting. With raycasting each worker only needs data for those particles whose glyph is at least partially in their domain, requiring less data duplication.

4. Results

We evaluate our in situ data library and renderer on the Stampede and Maverick clusters at TACC in several configurations, rendering data in situ from LAMMPS and Uintah simulations. The LAMMPS simulations consist of a thermal annealing study of a carbon nanosphere [26], synthetically replicated in a uniform grid to evaluate how our system scales with data size. The Uintah [2] simulation is a Lagrangian fluid dynamics code studying combustion of coal particles in a boiler system. Since our renderer can be run directly on the simulation nodes or a separate visualization cluster compare rendering performance and time spent sending data in these configurations.

4.1. Experimental Setup

The synthetic data are grids of nanospheres in LAMMPS; a single nanosphere is 1.05M atoms to examine the scalability of our system we test grids up to $6 \times 6 \times 6$ tiled nanospheres for a total of 227M atoms (fig. 5). In Uintah we use a particle injection simulation which injects approximately 57M particles per second and start from checkpoints in the simulation which range from 27.70M particles at $t = 0.12$ to 55.39M at $t = 0.24$.

Maverick has 132 nodes with two Intel Xeon E5-2680 v2 Ivy Bridge processors per node for a total of 20 cores, each node has 256GB of memory. On Maverick we use the Intel 15.0.3 compiler and Intel MPI 5.0.3.

Stampede has 6400 nodes with two Xeon E5-2680 Ivy Bridge processors per node for a total of 16 cores, each node has 32GB of memory. On Stampede we use the Intel 15.0.2 compiler and Intel MPI 5.0.2.

4.1.1. Separate Nodes

Running the renderer on a separate allocation of nodes allows for improved rendering and simulation performance compared to sharing nodes, and is ideal for exploring the simulation over a long period. In this configuration the simulation data must be sent over the network or high-speed interconnect to the renderer but this remains much faster than writing to disk.

We measure time spent sending data from LAMMPS to the rendering client at various sizes of the nanosphere grid data sets and compare this to time spent writing the same data to a file using LAMMPS "custom/mpiio" output mode in fig. 6 or sending it to our renderer in a shared node configuration, fig. 7a. Our in situ library allows for rendering each timestep of the data at a fraction of the cost of writing files at an equivalent frequency, especially as the data size increases. For this comparison we changed the "custom/mpiio" output mode to dump raw binary data equivalent to what we send our in situ client, instead of the LAMMPS's current method of serializing the data to ASCII, which would give an unfair data and work comparison.

We also evaluate send vs. write performance in Uintah running the particle injection simulation on 64 ranks and sending data to 12 render nodes on Stampede, fig. 8a. Uintah writes one binary file per process along with some XML metadata files describing the contents and locations of these binary files. The time spent writing files or sending data is comparable at lower particle counts, however as the data size grows the parallel file system becomes overloaded and file IO performance decreases, while our performance remains relatively flat.

4.1.2. Shared Nodes

Running the renderer on the same nodes as the simulation does impact the performance of both the simulation and the renderer, but only requires a single interactive node to display the viewer (or none when rendering to a file). This provides a non-intrusive way of checking in on a running simulation, particularly if the viewer will not be run for a long period of time, but rather connected occasionally to check in on the simulation state.

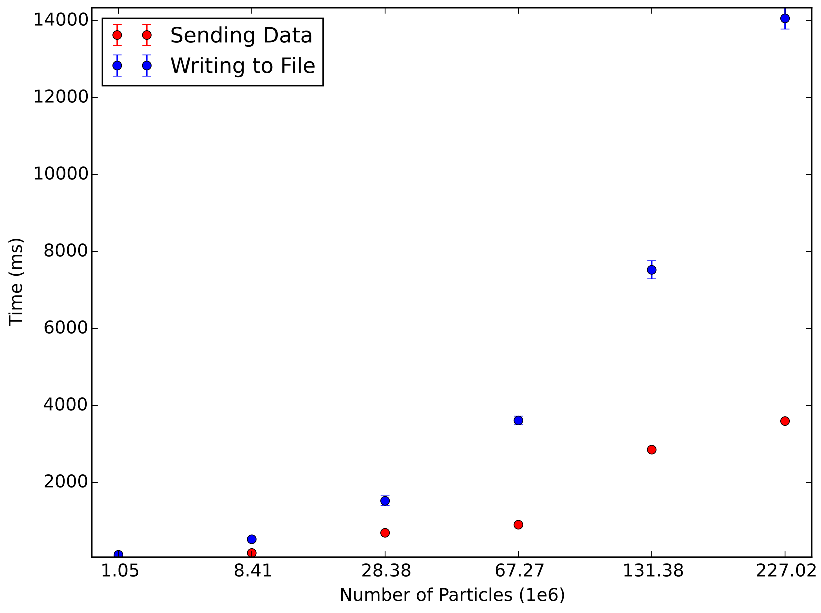
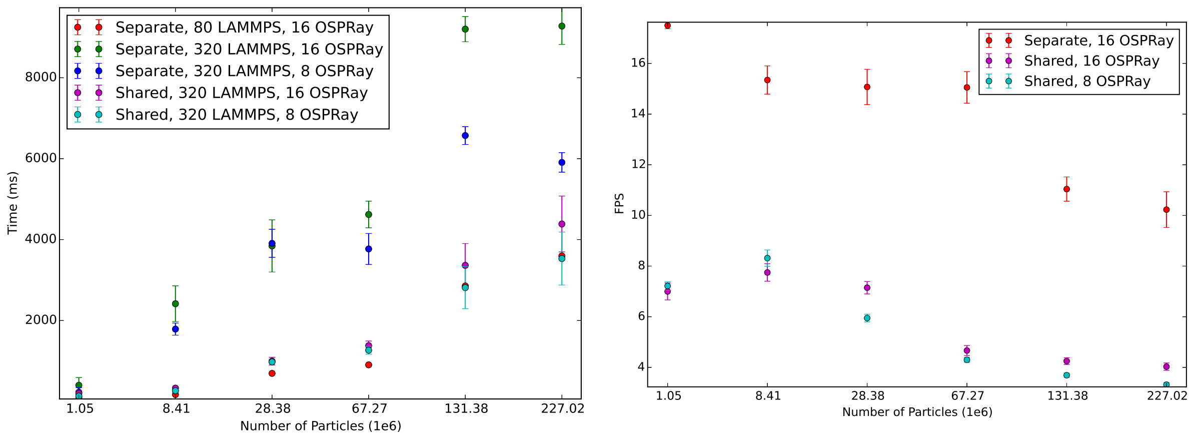


Figure 6. Sending data vs. writing files for a separate run with 80 LAMMPS ranks sending to 16 OSPRay ranks. We did not measure file writing beyond 67M particles as it becomes too time consuming



a) Data send times

b) Framerate with a 1004 × 1024 framebuffer

Figure 7. Performance of sending data and rendering for the replicated nanosphere data. Depending on the layout of ranks in a shared configuration it's likely that MPI will use shared memory to transfer data to our renderer, reducing send time compared to separate runs at the same number of nodes, as seen here. Although rendering performance is impacted when sharing nodes it remains interactive and scales reasonably well with data size

To launch the renderer on the simulation nodes we request a single visualization node, then use an MPI separate app/worker invocation to spawn OSPRay worker processes on the simulation nodes and a single viewer process on the interactive node. In this configuration, there is the possibility that shared memory will be used instead of network transfers to send data from the simulation to the client, though this is not currently guaranteed by our library.

Ideally, clients should pull data from local simulation processes when the simulation and renderer are run on the same nodes, however in practice this poses some challenges. When coupling to arbitrary simulations there is no guarantee that the simulation's data partitioning is suitable for distributed rendering. Sort-last compositing requires that each rank renders a convex, disjoint subregion of the data and there's no guarantee that the simulation has partitioned the particles in this manner. Further difficulty is introduced in handling $m \neq n$ simulation to renderer ranks, requiring a merging or scattering approach to distribute data evenly to the render processes while preserving the disjoint and convex requirements. Depending on the configuration of the simulation and renderer we may request data from local simulation ranks and in this case MPI will use shared memory to transfer the data (see fig. 2(b)). Even in the worst case where all requests go over the network this is still faster than writing to disk [6].

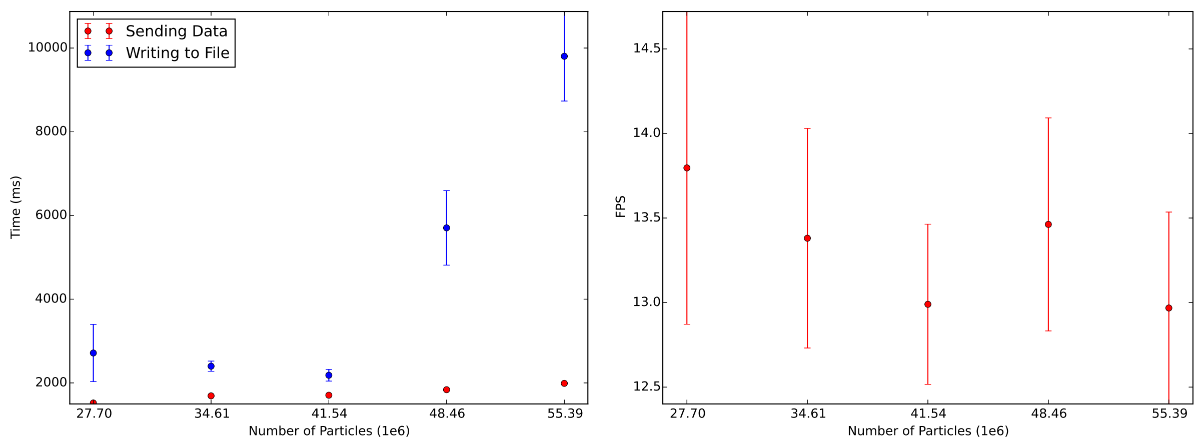
We measure scaling with data size in two configurations on Maverick. We run LAMMPS with 320 ranks (16 nodes) and the renderer on 8 or 16 of the same nodes and examine performance as the data size increases. As expected, we find that the framerate of both shared configurations is lower than the separate configuration (fig. 7b), but remains generally interactive.

More interesting is comparing the time spent sending data in the different configurations, fig. 7a. While we don't guarantee that our library will pull data from the simulation processes on the same node we find that when sharing nodes we spend much less time sending data than with equivalent process counts on separate nodes. This indicates that we do still gain from shared memory transfers even though we have no strict enforcement they will be used. Since OSPRay is run with one rank per node there are 20 LAMMPS ranks sharing the same node and it's likely that at least some of the data it will render is on one of these local ranks.

The separate run with 80 LAMMPS ranks performs better than the shared runs up to 131.38M particles where it falls only slightly behind the 320 LAMMPS to 8 OSPRay shared configuration. In this separate run, there is less communication overall between the render processes and LAMMPS as there are fewer LAMMPS processes to talk to. This reduces bandwidth needs and communication overhead compared to even the shared runs which coordinate with 4× as many simulation processes.

5. Summary and Discussion

We have presented a system for interactive in situ visualization of large particle simulations, suitable for general CPU-based HPC architectures. Our system is loosely coupled, allowing for simple connection and disconnection to the rendering client, and is easy to integrate with simulations. Running our system on the same nodes as the simulation offers some of the benefits of tightly coupled systems, i.e. using shared memory instead of network for data transfer. Using the P-k-d trees to represent particles in the we can classify and directly render the data efficiently with low memory overhead. We also adapt local shading effects (specifically, ambient occlusion) to be suitable for OSPRay's compositor for distributed rendering of particle data. To demonstrate the scalability and effectiveness of our system compared to traditional IO approaches we study its performance on data sets up to 227M particles. Our system enables exploration at the



a) Data send times b) Framerate with a 1920 × 1080 framebuffer

Figure 8. Performance of streaming data to our in situ renderer for the Uintah particle injection on Stampede with 64 simulation ranks streaming to 12 render nodes. In a) initially we're only marginally faster than writing to disk but as the data size grows the file system gets overloaded and write performance drops, while our in situ data streaming remains relatively flat. Rendering performance decreases a very small amount as we increase the data size

same spatio-temporal resolution as the simulation due to significantly reduced IO costs compared to saving data to disk. The code for libIS and our rendering client in OSPRay is available on Github: github.com/Twinklebear/in-situ-particles.

The HPC batch workflow itself remains the greatest barrier to adoption of interactive, exploratory in situ visualization. For many scientific users, lack of familiarity with general-purpose visualization software, paired with potentially high memory overhead and the need for dedicated GPU HPC resources, make interactive in situ techniques too impractical for everyday use. The practical effect of the high cost of IO has been far less data archived to disk. Our system aims at making interactive in situ visualization practical, by providing a lightweight library and rendering framework that let the user explore a simulation at runtime without interruption, on either a compute or visualization resource. Though our system is not yet competitive at scale with state-of-the-art IO forwarding frameworks like GLEAN [33] and ADIOS [18] or optimized distributed-parallel compositors [11, 28], this work shows that CPU ray tracing can provide an interactive in situ solution for a range of mid-size simulations.

5.1. Future work

An important limitation of our system compared to tightly coupled solutions is that we do not guarantee that when sharing nodes with the simulation the library will pull data from these local simulation ranks. This is challenging in the general case as we have no knowledge of how the simulation data is distributed across the nodes but it is reasonable to assume some spatial coherence. To this end, we would like to examine methods to preferentially pull data from local simulation ranks while keeping the data layout on each render node suitable for data parallel rendering.

We would also like to explore in situ rendering of mixed simulation data, simulations in Uintah often contain both volume and particle data and supporting these mixed simulations would make our situ library and renderer of use to a broader set of applications. Moreover, we would like to incorporate techniques of scalable IO forwarding frameworks and compositing-based renderers [11, 16, 28] into OSPRay along with view dependent data querying to effectively ray trace larger simulations at scale.

6. Acknowledgements

This work was supported in part by NSF: CGV: Award:1314896, NSF CISE ACI-0904631, DOE/Codesign P01180734, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375, and PIPER: ER26142 DE- SC0010498. Additional support comes from the Intel Parallel Computing Centers program and the Argonne Leadership Computing Facility. This material is also based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375.

We thank Joe Insley, Kah Chun Lau and Larry Curtiss at ANL for the nanosphere simulation, Tony Saad and Josh McConnell at the University of Utah CCMSC for the Uintah simulation, Mark West at Intel for generous loans of hardware, Paul Navrátil, Jeff Amstutz and Carson Brownlee for help running at TACC and with OSPRay, and Alan Humphrey and John Holmen for help integrating into Uintah.

References

1. Alexandre Ancel, Jean-Michel Dischler, and Catherine Mongenet. Load-Balanced Multi-GPU Ambient Occlusion for Direct Volume Rendering. In Hank Childs, Torsten Kuhlen, and Fabio Marton, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2012.

2. Martin Berzins, Justin Luitjens, Qingyu Meng, Todd Harman, Charles A Wight, and Joseph R Peterson. Uintah: a scalable framework for hazard analysis. In *Proceedings of the TeraGrid Conference*, page 3. ACM, 2010.

3. Martin Burtscher and Paruj Ratanaworabhan. pFPC: A parallel compressor for floating-point data. In *Data Compression Conference*, pages 43–52, 2009.

4. Hank Childs. In situ terminology project, https://ix.cs.uoregon.edu/ hank/insituterminology.

5. D. Ellsworth, B. Green, C. Henze, P. Moran, and T. Sandstrom. Concurrent Visualization in a Production Supercomputing Environment. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):997–1004, 2006.

6. N. Fabian, K. Moreland, D. Thompson, A.C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K.E. Jansen. The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library. In *Symposium on Large Data Analysis and Visualization (LDAV)*, pages 89–96. IEEE, 2011.

7. Thomas Fogal, Fabian Proch, Alexander Schiewe, Olaf Hasemann, Andreas Kempf, and Jens Krüger. Freeprocessing: Transparent in situ visualization via data interception. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2014.

8. Roland Fraedrich, Jens Schneider, and Rudiger Westermann. Exploring the Millennium Run - Scalable Rendering of Large-Scale Cosmological Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1251–1258, 2009.

9. Christiaan P Gribble, Thiago Ize, Andrew Kensler, Ingo Wald, and Steven G Parker. A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics*, (4):758–768, 2007.

10. Markus Gross and Hanspeter Pfister. *Point-Based Graphics.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

11. A. V. Pascal Grosset, Manasa Prasad, Cameron Christensen, Aaron Knoll, and Charles Hansen. TOD-tree: Task-overlapped Direct Send Tree Image Compositing for Hybrid MPI Parallelism. In *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization*, pages 67–76. Eurographics Association, 2015.

12. S. Grottel, M. Krone, C. Muller, G. Reina, and T. Ertl. Megamol – A Prototyping Framework for Particle-based Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):201–214, 2015.

13. Robert Haimes and David E Edwards. Visualization in a parallel processing environment. In *Proceedings of the 35th AIAA Aerospace Sciences Meeting, number AIAA Paper*, pages 97–0348, 1997.

14. William M. Hsu. Segmented Ray Casting for Data Parallel Volume Rendering. In *Proceedings of the Symposium on Parallel Rendering*, pages 7–14, 1993.

15. William Humphrey, Andrew Dalke, and Klaus Schulten. VMD: visual molecular dynamics. *Journal of Molecular Graphics*, 14(1):33–38, 1996.

16. S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout. PIDX: Efficient Parallel I/O for Multi-resolution Multi-dimensional Scientific Datasets. In *Proceedings of The IEEE International Conference on Cluster Computing*, pages 103–111, 2011.

17. P. Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014.

18. J. Lofstead, Fang Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich IO methods for portable high performance IO. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2009.

19. Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. A Data Distributed, Parallel Algorithm for Ray-traced Volume Rendering. In *Proceedings of the Symposium on Parallel Rendering*, pages 15–22, 1993.

20. Bruce Howard McCormick, Thomas A DeFanti, and Maxine D Brown. Visualization in scientific computing. *IEEE Computer Graphics and Applications*, 7(10):69–69, 1987.

21. Kenneth Moreland, Wesley Kendall, Tom Peterka, and Jian Huang. An Image Compositing Solution at Scale. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 25:1–25:10. ACM, 2011.

22. Steven G Parker and Christopher R Johnson. SCIRun: a scientific programming environment for computational steering. In *Proceedings of the ACM/IEEE conference on Supercomputing*, 1995.

23. T. Peterka, J. Kwan, A. Pope, H. Finkel, K. Heitmann, S. Habib, J. Wang, and G. Zagaris. Meshing the Universe: Integrating Analysis in Cosmological Simulations. In *High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion*, pages 186–195, 2012.

24. Tom Peterka, Dmitriy Morozov, and Carolyn Phillips. High-performance computation of distributed-memory parallel 3D Voronoi and Delaunay tessellation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 997–1007, 2014.

25. Steve Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of computational physics*, 117(1):1–19, 1995.

26. Vilas G Pol, Jianguo Wen, Kah Chun Lau, Samantha Callear, Daniel T Bowron, Chi-Kai Lin, Sanket A Deshmukh, Subramanian Sankaranarayanan, Larry A Curtiss, William IF David, et al. Probing the evolution and morphology of hard carbon spheres. *Carbon*, 68:104–111, 2014.

27. S. Rizzi, M. Hereld, J. Insley, M.E. Papka, T. Uram, and V. Vishwanath. Large-scale co-visualization for LAMMPS using vl3. In *Symposium on Large Data Analysis and Visualization (LDAV)*, pages 141–142. IEEE, 2015.

28. Silvio Rizzi, Mark Hereld, Joseph Insley, Michael E Papka, Thomas Uram, and Venkatram Vishwanath. Large-scale parallel visualization of particle-based simulations using point sprites and level-of-detail. In *Proceedings of the 15th Eurographics Symposium on Parallel Graphics and Visualization*, pages 1–10. Eurographics Association, 2015.

29. Sandia National Labs. LAMMPS Molecular Dynamics Simulator.

30. M. Tarini, P. Cignoni, and C. Montani. Ambient Occlusion and Edge Cueing for Enhancing Real Time Molecular Visualization. *IEEE Transactions on Visualization and Computer Graphics*, pages 1237–1244, 2006.

31. T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K. l. Ma, and D. R. O'Hallaron. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In *SC Conference, Proceedings of the ACM/IEEE*, pages 12–12, 2006.

32. V. Vishwanath, M. Hereld, and M. E. Papka. Toward simulation-time data analysis and I/O acceleration on leadership-class systems. In *Symposium on Large Data Analysis and Visualization (LDAV)*, pages 9–14. IEEE, 2011.

33. Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E. Papka. Topology-aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 19:1–19:11, 2011.

34. I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, and P. Navratil. OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):1–1, 2016.

35. I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka. CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees. In *Proceedings of IEEE Visweek*, 2015.

36. J. Woodring, J. Ahrens, J. Figg, J. Wendelberger, S. Habib, and K. Heitmann. In-situ Sampling of a Large-Scale Particle Simulation for Interactive Visualization and Analysis. *Computer Graphics Forum*, 30(3):1151–1160, 2011.

37. Hongfeng Yu, Chaoli Wang, R.W. Grout, J.H. Chen, and Kwan-Liu Ma. In Situ Visualization for Large-Scale Combustion Simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

38. F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, and M. Parashar. In-situ Feature-Based Objects Tracking for Large-Scale Scientific Simulations. In *High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion*, pages 736–740, 2012.