SAND2020-8332B

A Simulation-Oblivious Data Transport Model for Flexible In Transit Visualization

Will Usher, Hyungman Park, Myoungkyu Lee, Paul Navrátil, Donald Fussell and Valerio Pascucci

Abstract In transit visualization offers a desirable approach to performing in situ visualization by decoupling the simulation and visualization components. This decoupling requires that the data be transferred from the simulation to the visualization, which is typically done using some form of aggregation and redistribution. As the data distribution is adjusted to match the visualization's parallelism during redistribution, the data transport layer must have knowledge of the input data structures to partition or merge them. In this chapter, we will discuss an alternative approach suitable for quickly integrating in transit visualization into simulations without incurring significant overhead or aggregation cost. Our approach adopts an abstract view of the input simulation data and works only on regions of space owned by the simulation ranks, which are sent to visualization clients on demand.

1 Introduction

As discussed in the introduction chapter, two of the axes along which an in situ system can be classified are *proximity* and *access*. In situ methods can be roughly categorized

Will Usher, Valerio Pascucci
SCI Institute, University of Utah, e-mail: {will,pascucci}@sci.utah.edu

Hyungman Park
The University of Texas at Austin, Electrical and Computer Engineering, e-mail: hyungman@utexas.edu

Myoungkyu Lee
Sandia National Laboratories, e-mail: mnlee@sandia.gov

Donald Fussell
The University of Texas at Austin, Computer Science, e-mail: fussell@cs.utexas.edu

Paul Navrátil
Texas Advanced Computing Center, e-mail: pnav@tacc.utexas.edu

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

on these axes as “tightly coupled” (same process, direct access) or “loosely coupled” (different process, indirect access). Loosely coupled approaches have also been referred to as *in transit*, and we adopt this terminology throughout the chapter. A tightly coupled approach provides clear benefits by eliminating data copies or the need to synchronize between multiple processes to transfer data; however, loosely coupled approaches can provide a desirable alternative at scale [11].

A loosely coupled approach can run the simulation and visualization on distinct nodes, reducing the impact of the visualization on the simulation [8, 30]. This separation is highly desirable when scalability or resource contention is of concern, as is often the case in large-scale simulations. Furthermore, the visualization component is free to run at a different level of parallelism than the simulation, or can run in parallel to the simulation to perform additional computation [18, 8, 3, 30, 29] or enable interactive visualization [23, 25]. The visualization can also be run on demand, starting and stopping as desired in a separate process or job, based on, e.g., simulation triggers [12]. However, the application must now deal with the challenge of coordinating the two processes to transfer data from the simulation to the visualization.

When considering applying in transit visualization in practice, several challenges remain. Most off-the-shelf libraries for in situ visualization target tightly coupled use cases [9, 28]. The few libraries that do support in transit typically do so by repurposing an existing I/O API [16, 26], and perform data aggregation and redistribution using a general distributed data access strategy (e.g., DataSpaces [5], FlexIO [31], Flexpath [4]). Although such approaches fit well into simulations already using the repurposed I/O API, migrating to a new I/O API for the express purpose of enabling in transit visualization may not be desirable, especially for simulations leveraging a custom optimized I/O library, such as HACC [10].

This chapter discusses the simulation-oblivious data transport model employed by libIS [23] for in transit visualization, which lowers the bar to using in transit visualization in practice. By adopting an abstract view of the simulation data, this approach can be integrated into a range of simulations employing arbitrary mesh types. This approach is also well suited to $M : N$ configurations, where M simulation ranks communicate with N visualization ranks, allowing each task to be run in its ideal configuration. This strategy natively supports asynchronous and on-demand in transit visualization, is portable across a range of HPC or ad hoc cluster systems, and does not require significant changes to the simulation to integrate. Moreover, we discuss developments extending libIS which allow greater portability across HPC systems and simulation codes.

2 A Simulation-Oblivious Data Transport Model

Our simulation-oblivious data model treats each simulation rank as an opaque block of bytes, the interpretation of which is left up to the simulation and client. Specifically, the model does not inspect, adjust, or redistribute the input data distribution provided

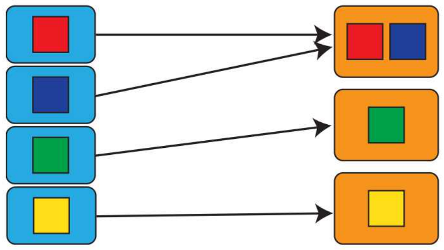


Figure 1: An 4 : 3 mapping of simulation data (left) to visualization clients (right). With $M > N$ each rank is assigned $\frac{M}{N}$ simulation states, with any remainder distributed among the clients.

by the simulation, but instead simply forward each rank's data to the assigned client. Each client receives one simulation state from each simulation rank it is assigned to process data from, allowing for $M : N$ configurations (see Figure 1).

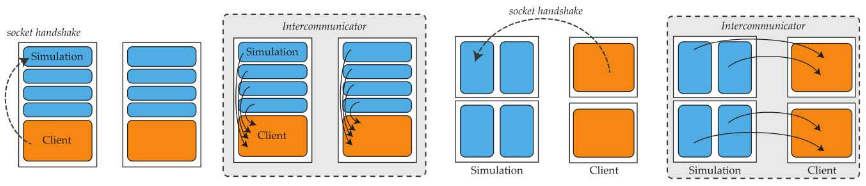
If the simulation and visualization differ in their parallelism model (i.e., MPI-only vs. MPI + threads) or scalability, an $M : N$ configuration can be used to run each in its ideal configuration. In a 1 : 1 configuration, each client is assigned a single simulation rank's data. When $M > N$ each client is assigned data from $\frac{M}{N}$ simulation ranks, with any remainder distributed among the clients. The client application is then free to merge its assigned data together, or keep it distinct. Depending on the simulation and visualization configuration, it may not be the case that the set of simulation states sent to each client forms in aggregate, e.g., a convex region in the domain. We note that this oblivious model supports only $M \geq N$, since it does not support splitting a rank's data to redistribute it. However, it is typically the case that the simulation is run at the same or higher level of parallelism than the visualization.

Although a somewhat primitive data model, this simplified view provides some desirable benefits over redistribution-based approaches. Removing the restructuring process reduces computational cost and library complexity, and eases integration into simulations with more exotic mesh types or primitives that may not be supported by the chosen redistribution strategy. Moreover, this approach is able to achieve high network utilization when querying data from the simulation, as little computation is performed during the process to assign data to clients; in turn reducing the simulation time spent performing data transfers to the clients. From a practical standpoint, this approach does not rely on more complex distributed data models or libraries, and can be easily deployed on a variety of systems, making it useful for lightweight in transit integrations, or as the underlying data transport layer for in full-featured systems (e.g., SENSEI [2]).

2.1 Implementation in libIS

The libIS library[1] [23] implements our oblivious data transport model to provide a lightweight library for asynchronous in transit visualization. A simulation using

[1] `https://github.com/ospray/libIS`



(a) Shared node configuration. (b) Separate node configuration.

Figure 2: Visualization clients using libIS can be run on the same nodes as the simulation (a), and thus take advantage of shared memory data transfers at the cost of oversubscription, or separate nodes, where the applications will not impact each other at the cost of communicating over the network (b). (a) shows a 4 : 1 configuration run on two nodes, (b) shows a 2 : 1 configuration on four nodes. The figures are reproduced from our previous work [23].

libIS acts as a data server that clients can connect to and query data from. Data are transferred to clients each timestep using the model described above. The library is split into a simulation and client library: the simulation library provides a C API to support simulations written in most languages, and the client library provides a C++ API to fit well with C++ based visualization software (e.g., VTK [22]). Visualization clients using libIS can connect and disconnect as desired from the simulation, to support on-demand execution of the visualization. Furthermore, libIS does not impose requirements on where the clients are run. The clients can be run on the same nodes as the simulation, distinct nodes on the same HPC resource, or an entirely separate HPC resource. The clients can be started manually by the user or automatically using, e.g., in situ triggers [12].

Since the publication of the short paper [23], we have pursued efforts to improve portability across MPI runtimes and simulations, and to improve usability and performance. The library now includes a Fortran wrapper over the C API to ease integration into Fortran-based simulations and a fallback socket-based intercommunicator for portability across different HPC systems and MPI configurations. Moreover, this chapter presents an extensive evaluation of the performance of libIS and the impacts of libIS and visualization clients on the simulation in the supported configurations and communication modes.

2.1.1 Portable Communication Between the Simulation and Client

LibIS allows clients to connect and disconnect from the simulation as desired to enable on-demand execution. Rank 0 of the simulation spawns a background thread that listens for incoming connections on a socket. To establish an intercommunicator (Figure 2), rank 0 of the client connects to this socket and sends the simulation its hostname and port to connect back to. The client is then able to request data from the simulation and can disconnect when its analysis is complete.

The initial version of libIS used an MPI intercommunicator, and thus required the `MPI_Open_port`, `MPI_Comm_accept` and `MPI_Comm_connect` APIs to establish it. In practice these APIs are not always available. For example, Theta at Argonne National Laboratory does not provide these APIs, which motivated the implementation of an MPI multilaunch mode in our initial work on libIS. However, the MPI multilaunch mode does not support on-demand execution of the visualization, as the visualization processes must be started together with the simulation in the same `mpirun` command, using MPI's MPMD launch mode.

To portably support on-demand connection in libIS, we have implemented a socket-based fallback intercommunicator. When attempting to connect, both the simulation and client test if the `MPI_Open_port` API is available. If the API is not available, they fall back to a socket-based intercommunicator. In this case, each client opens a socket and listens for connections from the simulation. Rank 0 of the simulation broadcasts rank 0 of the client's hostname and port number to the other ranks, which each connect to client rank 0 and receive the hostname and port numbers for the other clients. The simulation ranks then connect to the remaining clients to establish an all-to-all socket intercommunicator. When setting up a connection the simulation rank sends its rank number to the client, which sends back its rank number. Each socket on a process is indexed by the rank of the process on the other end to provide a send and receive API identical to MPI. Finally, to avoid flooding each client with incoming connections from a large simulation run, we rate limit the simulation ranks' connection requests. Without rate limiting the OS would see the large number of incoming connections as a network flooding attack.

The socket intercommunicator also enables running the clients on entirely different hardware and software stacks or HPC resources. Performing the in transit visualization on an entirely distinct cluster can be useful in sensitive or time critical applications, as discussed by Ellsworth et al. [8].

2.1.2 Querying Data

After establishing the intercommunicator, the client can request to receive data from the simulation. The simulation will probe for incoming messages from rank 0 of the client after each timestep when calling `libISProcess`. The client can request a new timestep or disconnect from the simulation. If a new message is received, it is broadcast to the other simulation ranks and processed collectively.

Data are transferred from the simulation ranks to the clients using the data model and $M : N$ mapping discussed above. We compute N groups of $\frac{M}{N} : 1$ simulation to client groupings to assign data to clients. Each such group transfers data to the client rank independently and in parallel to the others. When using the MPI intercommunicator, data are transferred using point-to-point communication. When using the socket intercommunicator, data are transferred using the socket connection established on each simulation rank to the assigned client. The data transfer avoids global all-to-all communication and scales well with the number of simulation and client ranks.

LibIS does not buffer the previous timestep to send to clients that requested data during computation. Clients requesting data will wait until the current timestep finishes and the data can be sent directly from the simulation's memory. By not buffering the previous timestep, libIS does not need to keep an additional copy of the simulation state, reducing memory pressure.

```
// Set the world bounds (per-rank local/ghost bounds identical)
void libISSetWorldBounds(libISSimState *state, const libISBox3f box);

// Convenience method to set a 3D regular grid field
void libISSetField(libISSimState *state, const char *fieldName,
    const uint64_t dimensions[3], const libISDType type, const void *data);

// Convenience method to set an array of local and optional ghost particles
void libISSetParticles(libISSimState *state, const uint64_t numParticles,
    const uint64_t numGhostParticles, const uint64_t particleStride,
    const void *data);

// Set an arbitrary 1D buffer of data
void libISSetBuffer(libISSimState *state, const char *bufferName,
    const uint64_t size, const void *data);

// Call after each timestep to send data to any clients
void libISProcess(const libISSimState *state);
```

Listing 1: The libIS simulation API is used to configure a simulation state object, which stores pointers to the rank's local data. Convenience methods are provided for regular 3D fields and particles; arbitrary 1D buffers of data can be sent via the buffer API.

2.1.3 The Simulation API

The simulation library provides a C API and a Fortran wrapper. Simulations begin listening for clients by calling `libISInit` and passing the port number to listen on. To make data available to clients, each rank configures a `libISSimState`, which stores pointers to the simulation data and metadata describing it (Listing 1).

The current API provides convenience wrappers for setting regular 3D fields and particles, along with a generic API to pass arbitrary 1D buffers of data. Internally, 3D fields and particles are treated the same as 1D buffers, since the data are not inspected or redistributed when sent to clients. Simulations using more complex data representations, e.g., unstructured meshes, octree AMR, block-structured AMR, etc., can use the buffer API to send the local mesh data to clients. The clients must be tailored to the simulation and know how to interpret and process the incoming data.

The `libISProcess` function is called collectively by the simulation ranks to send data to any clients that have requested to receive the latest timestep. Data are sent directly from the simulation pointers shared with libIS when configuring the simulation state to avoid data copies.

2.1.4 The Client API

The libIS client library provides a C++ API (Listing 2) to integrate well into existing visualization software, which primarily uses C++. Clients first connect to the simulation by calling `connect`, after which they can query data using the blocking or asynchronous API. Once the desired analysis has completed, the client can disconnect and exit. The client can also check if the simulation has quit by passing an optional parameter to the API calls, or explicitly checking `sim_connected`.

```
struct SimState {
    libISBox3f world, local, ghost;
    // The rank we received this data from
    int simRank;
    // The 3D fields and 1D buffers sent by the simulation
    std::unordered_map<std::string, Buffer> buffers;
    // The particles sent by the simulation, if any
    Particles particles;
};

// Connect to the simulation listening on rank 0 at host:port
void connect(const std::string &host, const int port,
    MPI_Comm ownComm, bool *sim_quit = nullptr);

// Query the next timestep, blocking until it is ready
std::vector<SimState> query(bool *sim_quit = nullptr);

// Asynchronously query the next timestep.
// The future can be monitored for completion
std::future<std::vector<SimState>> query_async(bool *sim_quit = nullptr);

// Disconnect from the simulation
void disconnect();

// Check if the simulation has terminated
bool sim_connected();
```

Listing 2: The libIS client API is used to connect to a running simulation and query data from it. Each client will receive $\frac{M}{N}$ simulation states, containing data from the simulation ranks it is assigned to receive data from.

Each client receives a vector of simulation states, containing the $\frac{M}{N}$ simulation ranks the client is assigned to receive data from. Each state contains the 3D and 1D buffers sent by the simulation, associated with their name, and any particles.

3 Example Use Cases

We evaluated the scalability and portability of libIS on two simulations, LAMMPS (Section 3.1) and Poongback (Section 3.2), using two HPC systems. We ran our benchmarks on Stampede2 at the Texas Advanced Computing Center (TACC) and Theta at Argonne National Laboratory. Both systems contain roughly similar Intel Xeon Phi Knight's Landing (KNL) nodes: KNL 7250 on Stampede2 and KNL 7230 on Theta. Stampede2 contains an additional partition of Intel Skylake Xeon (SKX) nodes.

Although the KNL nodes are similar on the two systems, the network architecture differs significantly. Stampede2 employs a fat-tree topology Omnipath network, whereas Theta uses a 3-level Dragonfly topology Cray Aries network. On Stampede2 MPI uses the Omnipath network, which can provide up to 100Gbps of bandwidth; however, sockets uses the 1Gbps ethernet network. Theta does not provide an ethernet network. Instead, sockets uses the Aries network and can achieve a peak bandwidth of 14Gbps.

The following sections compare the MPI and socket intercommunicators on Stampede2 and Theta using LAMMPS, to evaluate network performance portability and scalability with a test client application that queries data repeatedly (Section 3.1.1). We also present example use cases of in transit image database generation with LAMMPS (Section 3.1.2) and Poongback (Section 3.2.3) on Stampede2, and measure rendering performance of the client and impact on simulation performance. Finally, we provide a brief comparison against the existing redistribution-based data transfer method used in ADIOS (Section 3.3).

The in transit image database rendering benchmark is an example of using libIS in combination with an OSPRay-based [27] rendering application to render image databases, similar to those used in Cinema [1]. We modify the OSPRay mini-cinema example[2] to query data using libIS. After receiving the data mini-cinema renders a camera orbit around it using OSPRay's data-distributed API [24] for data-parallel rendering. When the image set is finished, the application queries the next timestep and repeats for a specified number of timesteps.

3.1 LAMMPS

LAMMPS [19, 20] is a large-scale molecular dynamics simulation code, which we run MPI-parallel. To integrate libIS into LAMMPS, we leverage existing mechanisms for coupling LAMMPS with other codes [21]. We build a wrapper application that behaves as the regular LAMMPS executable, with the difference that before the simulation starts, our wrapper initializes libIS and installs a fix callback to call `libISProcess` each timestep. This application is available as an example on the libIS GitHub[3]. After each timestep, the simulation will send its local and ghost particles to clients querying data.

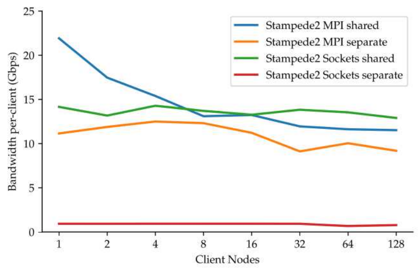
3.1.1 Performance Portability and Scalability

The weak scaling benchmark uses the Lennard Jones benchmark problem included with LAMMPS, which we replicate to store 131k particles per simulation rank. We measure the bandwidth achieved when querying data using the example client included with libIS, which simply prints out the received metadata. The benchmarks are run in a 16 : 1 configuration, where each client receives approximately 2.1M particles, which amounts to roughly 45MB of data per client after including ghost zones. Each group of 16 simulation ranks is placed on one node, and the client is run with one process per-node.

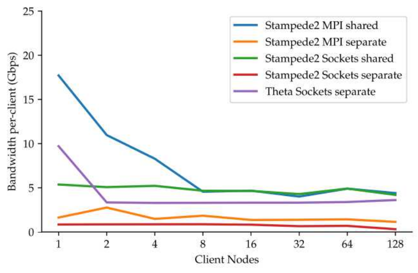
The benchmark is run on 1 to 128 client nodes, corresponding to 16 : 1 to 2048 : 128 configurations, using the SKX nodes on Stampede2 (Figure 3a) and the KNL nodes

[2] `https://github.com/Twinklebear/mini-cinema`

[3] `https://github.com/ospray/libIS`



(a) Bandwidth per client on Stampede2 SKX.



(b) Bandwidth per client on KNL.

Figure 3: Bandwidth per client rank achieved in the weak scaling benchmark. We find that the independent communication strategy employed by libIS achieves good weak scaling and high network utilization. Shared node runs are able to leverage shared memory for improved bandwidth.

on Stampede2 and Theta (Figure 3b). In the shared node configuration, the client and simulation are run across the entire set of nodes, using the same resources. In the separate node configuration, we split the allocation of nodes in half between the simulation and client. The benchmark records bandwidth per client, and we compare the shared and separate configurations and the MPI and socket intercommunicators.

We find that the independent communication mode employed by libIS weak scales well, and can nearly saturate the 1Gbps ethernet network when using sockets on Stampede2. When using MPI over Omnipath on Stampede2, libIS is not network bound, and averages 11% network utilization on SKX and 2% on KNL. When using sockets on Stampede2, libIS averages 88% network utilization on SKX and 74% on KNL. When using sockets on Theta, libIS averages 30% network utilization. The relatively low network utilization achieved with MPI could potentially be resolved by parallelizing the data transfer from the simulation ranks to their clients. Although each group of simulation ranks and clients communicates in parallel, the simulation ranks within a group send their data to the client in serial.

When run in the shared node configuration, shared memory can be used for higher bandwidth by both the MPI and sockets intercommunicators, and we find the sockets intercommunicator achieves performance similar to MPI. The shared node configuration avoids the practical challenges of queuing and running a second job on the HPC system and is likely to be a common use case for in transit visualization. The ability of both MPI and sockets to use shared memory for higher bandwidth is especially promising for performance portability.

3.1.2 In Transit Image Database Rendering

For the image database rendering example, we use the Rhodopsin benchmark input and run LAMMPS in the same $16:1$ configuration as before. OSPRay uses an MPI + Threads model and is run with a single rank per node, whereas LAMMPS is run MPI-parallel with 16 ranks per node. The Rhodopsin benchmark stores 32k particles per simulation rank, resulting in each client rank receiving 512k particles in the $16:1$ configuration. After accounting for ghost zones, this corresponds to roughly

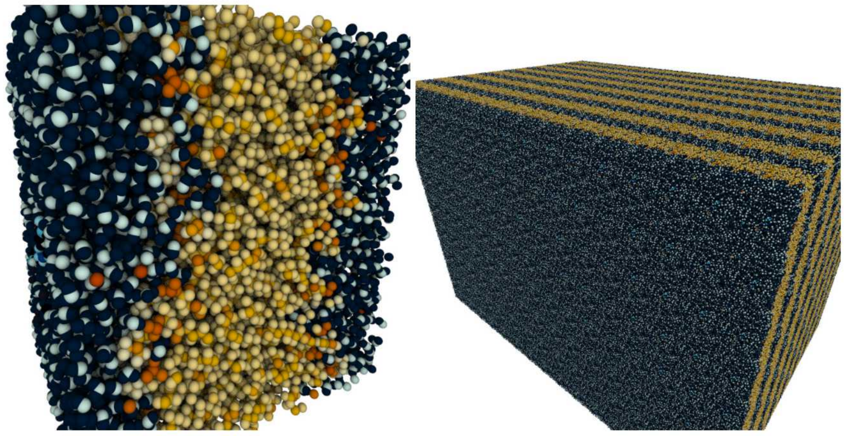
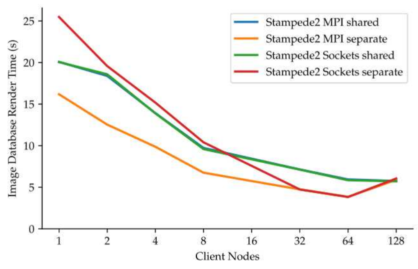
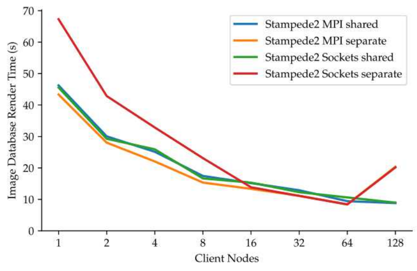


Figure 4: The LAMMPS Rhodopsin benchmark rendered with ambient occlusion. Left: the simulation data on a single rank. Right: replicated in our weak scaling benchmark for 1024 ranks. By using the ghost particles already computed by LAMMPS, our in transit renderer is able to compute ambient occlusion effects using only the local data available on each rank.



(a) Stampede2 SKX.

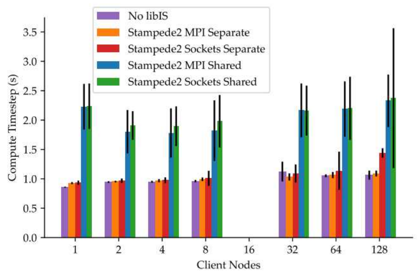


(b) Stampede2 KNL.

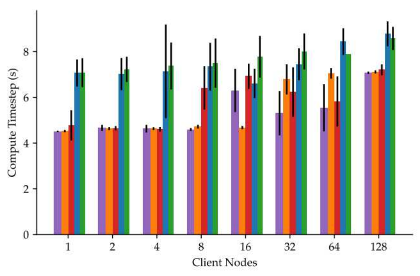
Figure 5: Camera orbit render times on SKX and KNL for each timestep. We find that the image database render task scales well, with the shared node configuration slightly impacted by the simulation. We find a performance decrease at 128 clients in the separate configuration, which may be due to a different network placement for the 256 node job impacting compositing performance.

20MB of data per-client. The additional ghost particles are used to compute ambient occlusion to provide better depth cues (see Figure 4). The benchmark renders an orbit of 80 camera positions around the data, and does so for 50 timesteps. OSPRay's asynchronous rendering support is used to reduce the total time to render the set of by rendering eight images in parallel. We measure the total time to render all 80 frames for each timestep at 1024^2 pixels, and compare the shared and separate configurations using the MPI and sockets intercommunicators (Figure 5).

We find that at low node counts, the difference in the total render time between the MPI and socket intercommunicators is unexpectedly large. The different communicators affect only the data query with libIS, which is not timed in the benchmark, and we would expect to observe similar performance. At higher node counts, the MPI and socket modes converge to similar render times, with the separate node configuration achieving slightly better performance than the shared node one, as



(a) Stampede2 SKX.



(b) Stampede2 KNL.

Figure 6: The impact of the image database rendering client on LAMMPS performance. Bars show the average time to compute each timestep in the different modes, with standard deviation shown as error bars. As expected, the shared node configuration significantly impacts performance, whereas the separate node configuration has relatively little effect. On 16 SKX nodes the LAMMPS simulation became unstable, and we were unable to run benchmarks in that configuration. The plots share the same legend.

expected. The shared node configuration runs both the simulation and renderer on the same nodes, and some performance degradation of both is to be expected. On KNL each configuration achieves roughly similar performance, potentially due to the larger number of available cores for the renderer, and the relatively weaker per-core performance compared to SKX. We note that the overall render time decreases in the benchmark, which is a result of each client's local data projecting to a smaller region of the image as the data set is scaled up.

3.1.3 Impact on Simulation Performance

Finally, we measure the impact on simulation performance for each configuration by comparing the time taken to compute each timestep with and without the mini-cinema client running (Figure 6). In the separate node configuration, the simulation and client are run on distinct nodes, and do not contend for resources. Thus, the simulation is impacted only by the time spent sending data to the clients. On the Stampede2 SKX nodes, libIS has a negligible impact when using MPI or sockets, though do observe an outlier at 128 nodes where a larger impact when using sockets is observed. On the KNLs the simulation is 4% slower on average when using MPI, with a similar impact observed for sockets, though we do observe an outlier at 16 nodes where a greater impact when using sockets is measured. The greater impact on KNL can be attributed to the lower single-core performance compared to SKX, which increases the time spent transferring data to the clients.

The shared node configuration runs the simulation and client on the same nodes, potentially leading to significant resource contention and impacting simulation performance. On SKX simulation takes 82% longer when using MPI and 104% longer when using sockets. On KNL the simulation takes 44% longer when using MPI and 44% longer when using sockets. The reduced impact on KNL is likely attributable to the larger number of cores available, which may reduce resource contention for

Table 1: Weak scaling configurations for the Poongback benchmark, targeting roughly 1.4GB of volume data per client rank.

Client Ranks	Total Voxels	Total Volume Size (GB)
1	384×768×576	1
2	768×768×576	3
4	768×768×1152	5
8	1536×768×1152	11
16	1536×768×2304	22
32	3072×768×2304	43
64	3072×768×4608	87
128	6144×768×4608	174

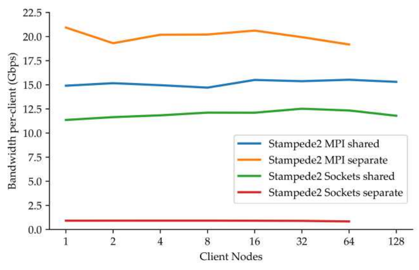
processors, and the availability of MCDRAM, which in this configuration is large enough to hold both application's data.

3.2 Direct Numerical Simulation of Turbulent Channel Flow with Poongback

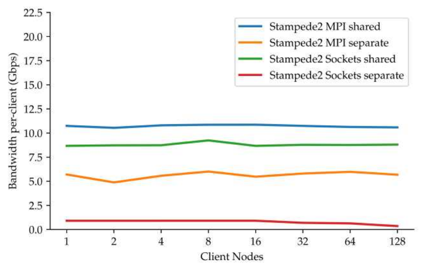
We simulate a large-scale turbulent channel fluid flow using Poongback [13, 14, 15, 17], and use libIS to transfer data to the mini-cinema clients as before. Poongback is a computational fluid dynamics (CFD) solver for direct numerical simulation (DNS) of incompressible turbulent channel flows written in Fortran. The simulation generates a 3D volume data set and runs MPI-parallel. To integrate libIS into Poongback, we use libIS's Fortran wrapper. Minimal modifications to the simulation code are required to integrate in transit visualization through libIS. First, before the Poongback simulation begins, we initialize libIS and configure the libIS simulation state on each rank with the bounds of its local volume data. After each timestep we call the `libISProcess` wrapper to send the data on to any clients requesting the current timestep. Poongback stores its data row-major, and does not require a transpose to be done after receiving data on the client.

3.2.1 Evaluation Setup

We evaluate libIS with Poongback using mini-cinema for in transit image database rendering, and record network utilization, rendering performance, and the impact on simulation performance. The Poongback benchmarks are run on Stampede2 SKX and KNL nodes, using a weak scaling benchmark for Poongback. To create the weak scaling benchmark, we proportionally increase the simulation dimensions based on the number of clients, to maintain roughly 1.4GB of volume data per client (Table 1). The simulation and clients are run in a 32 : 1 configuration. Poongback is run MPI-parallel with 32 ranks per node, whereas the mini-cinema clients are run using MPI + Threads with one rank per node. In our benchmark we compare both



(a) Bandwidth per client on Stampede2 SKX.



(b) Bandwidth per client on Stampede2 KNL.

Figure 7: Bandwidth per client rank achieved in the Poongback weak scaling benchmark. We find that the independent communication strategy employed by libIS achieves good weak scaling, and performs well when transferring the larger portion of data sent by each Poongback rank.

the shared and separate configurations and the MPI and socket intercommunicators. When using the shared node configuration, the simulation and application are run on the same nodes, whereas in the separate configuration half the nodes are assigned to the simulation and half to the client. The benchmarks are run up to an aggregate of 128 SKX nodes and 256 KNL nodes. For shared node runs, we scale from a 32 : 1 configuration up to a 4096 : 128 one on SKX and KNL. For separate node runs, we scale from a 32 : 1 configuration up to a 2048 : 64 one on SKX and a 4096 : 128 one on KNL.

3.2.2 Performance Portability and Scalability

We measure the average data transfer bandwidth achieved on each client over 50 timesteps (Figure 7). When using MPI in the separate node configuration, data are transferred over the 100Gbps Omnipath network, and libIS achieves an average of 20% network utilization on SKX and 6% on KNL. When using sockets in the separate node configuration, data are transferred over the 1Gbps ethernet network, and libIS achieves 90% network utilization on SKX and 78% on KNL. As discussed previously, parallelizing the data transfer within each set of simulation ranks and client could improve network utilization when using MPI, especially for the larger aggregate data transfer performed for the Poongback simulation.

In the shared node configuration, each set of 32 : 1 ranks is run on the same node, ensuring communication is local to each node. As a result, data can be transferred using shared memory instead of over the network to achieve higher bandwidth. A notable exception is on SKX (Figure 7a), where higher bandwidth is achieved in the separate configuration than in the shared one when using MPI. This result is counter to our results on KNL with Poongback and on KNL and SKX with LAMMPS, and warrants further investigation.

Overall, we achieve higher bandwidth on SKX than KNL, likely due to the higher per-core performance of SKX. Similar to our findings with LAMMPS in Section 3.1.1,

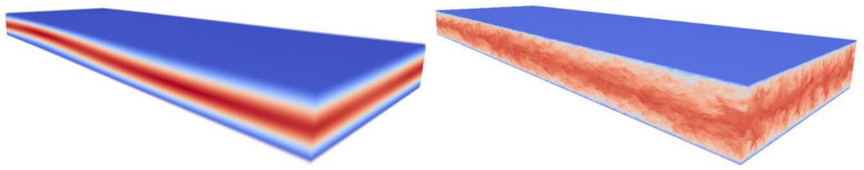


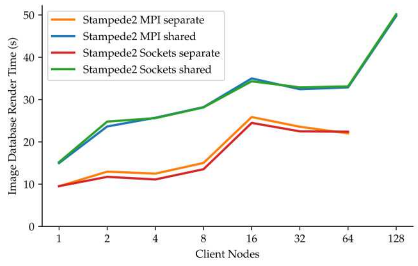
Figure 8: Images of the Poongback turbulent channel-flow simulation rendered using our mini-cinema libIS client. Both volumes are $6144 \times 768 \times 4608$ voxels in double-precision floating-point values (174GB). Left: The simulation state at timestep 0, the initial condition used in our weak scaling benchmarks. Right: A checkpoint of the same simulation (fully developed turbulence).

MPI achieved higher bandwidth than sockets, as MPI is able to leverage the faster Omnipath network. However, in the shared configuration both can leverage shared memory and the gap between the two narrows. In terms of overall weak scalability, the independent data transfer strategy employed by libIS scales well with the number of clients, with per client bandwidth remaining nearly constant across each scaling run.

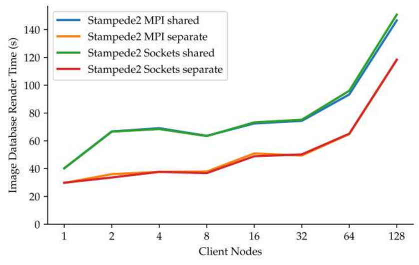
3.2.3 In Transit Image Database Rendering

For each of the 50 timesteps queried during the benchmark, the mini-cinema client renders an 80-position camera orbit around the data set. Each image is rendered at a 1024^2 resolution, with one sample per pixel and a volume sampling rate of one sample per voxel. In the largest run with 128 client ranks, the application queries and renders a total of 8.7TB of volume data over the course of the 50 timesteps. We disable the asynchronous rendering functionality of mini-cinema to reduce the impact of the client in the shared node configuration, and disable it in the separate configuration for consistency.

The time taken to render each camera orbit is shown in Figure 9. In contrast to the results observed with LAMMPS, the overall rendering time increases as additional client ranks are added. We believe this to be due to the differing data distributions of the two simulations. Compared to LAMMPS, where each simulation rank has a cube of data, Poongback partitions its data using a pencil decomposition [13]. After multiple pencil-pencil data transposes, the set of regions assigned to each client is a group of these x-pencils along the axis of flow, so that each pencil spans the entire x-axis of the data set. The set of regions assigned to each client is a group of these x-pencils, where each pencil spans the entire x axis of the data set. The set of regions project to a large number of pixels for most of the viewpoints in the orbit. The number of pixels covered does not reduce significantly as more ranks are added, leading to a significant amount of rendering and compositing work. One measure that could be taken to alleviate the compositing work is to merge the 32 regions assigned to each client rank into a single OSPRay rendering region, allowing them to be rendered and composited as a single brick, instead of 32 separate ones. This optimization would reduce the compositing work by a factor of 32, allow for faster local rendering, and provide a meaningful performance improvement.

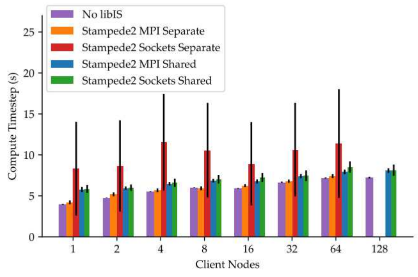


(a) Stampede2 SKX.

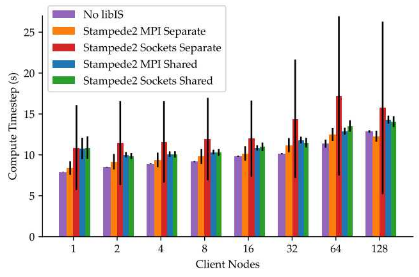


(b) Stampede2 KNL.

Figure 9: Camera orbit render times on SKX and KNL for each timestep in our weak scaling benchmark. We achieve better total rendering performance on SKX, and in the separate node configurations, as the shared node configuration oversubscribes the nodes. In contrast to the LAMMPS results, we find poorer scaling at higher node counts, potentially due to the differences in data distribution and compositing workload.



(a) Stampede2 SKX.



(b) Stampede2 KNL.

Figure 10: The impact of the image database rendering client on Poongback performance. Bars show the average time to compute each timestep in the different modes, with standard deviation shown as error bars. Overall, the time-consuming socket-based communication in the separate configuration incurs the most overhead. By not rendering multiple frames in parallel, the renderer's impact on the simulation is reduced in the shared configurations.

We find similar results as those observed with LAMMPS when comparing the relative rendering performance of the different configurations and communicators. Although we find little impact of the communication method used (MPI vs. sockets) on render time, we find a significant impact on performance when using the shared configuration. The shared configuration can lead to contention between the simulation and client, and impacts the performance of both.

3.2.4 Impact on Simulation Performance

Finally, we measure the impact of connecting the mini-cinema client in the different configurations on simulation performance (Figure 10). We find that the overhead is primarily determined by the ratio of compute time and data transfer time for each timestep. Depending on how large this ratio is, the simulation becomes either compute bound or communication bound. Thus, the impact on simulation time is related to the network performance measured in Section 3.2.2. We also observe a much higher standard deviation in the overhead, which is likely tied to varying network performance when using sockets, while MPI and local sockets are able to provide more consistent performance. Moreover, when not rendering multiple frames in parallel, the rendering client has little impact on the simulation in the shared configuration.

When using the socket communicator in the separate mode, the simulation becomes communication bound, and must wait for the data transfer to complete before advancing, thereby impacting performance. The sockets separate configuration has the greatest impact on simulation performance of the different modes, increasing compute time by 78% on SKX and 34% on KNL. The impact is greater on SKX as faster nodes lead to a 1.7× faster simulation, exacerbating the impact of the simulation becoming bound by the data transfer. In the other configurations, the impact is less severe. The MPI separate mode increases simulation time by 4% and 6%, on SKX and KNL, respectively; the MPI shared mode by 18% and 14%; and the sockets shared mode by 22% and 17%.

3.3 Comparison to Existing Libraries

To compare data transfer performance against existing restructuring-based techniques we provide a brief comparison against the widely used ADIOS [16] I/O library (version 2.5.0). ADIOS repurposes its existing I/O API to support in transit visualization, allowing users to simply change the I/O "engine" being used by the simulation and client. ADIOS adopts a lightweight data model, processing one-, two-, or three-dimensional arrays of primitive types which are passed to the I/O engine. However, in contrast to libIS's oblivious data model, ADIOS supports redistributing the data to the clients. Each client specifies a starting offset within an array and number of elements to be read, as if reading from a file. ADIOS will then make the requested data available on that rank. Though this provides a transparent transition from a file-based pipeline, in an in transit scenario this requires some form of data redistribution, potentially adding overhead to the data transfer.

We write a test application which generates 131k particles per rank and run the simulation and client in a 16 : 1 configuration, matching the Lennard Jones benchmark configuration in Section 3.1.1. Each particle attribute (x, y, z, type) is passed to ADIOS as a global array. Clients request the subregion of this array corresponding to their assigned set of simulation ranks. To transfer data we use the "SST" engine, which supports $M : N$ data transfer and will make use of RDMA enabled networks where available.

In both the shared and separate configurations ADIOS achieves similar data transfer speeds, averaging 0.5Gbps per client on SKX and KNL when run with 16 clients on Stampede2. In contrast, libIS averages 13Gbps and 11Gbps respectively on SKX, and 5Gbps and 1.5Gbps respectively on KNL (see Figure 3). The potential need for more global communication among ranks to perform data redistribution comes at a cost compared to our independent model, though is convenient for applications.

SENSEI [2], which provides adapters to bridge between the multitude of in situ frameworks to ease portability, could select between ADIOS or libIS for data transport, depending on the application's needs. For example, if the visualization task work on subblocks of data instead of requiring redistribution, libIS can be used for increased bandwidth. If this is not the case, ADIOS could be used to redistribute the data to meet the client's needs.

4 Conclusion

We have presented a simulation-oblivious data transport model for in transit visualization. Although a relatively primitive model, this approach is desirable in a number of scenarios. When time spent by the simulation in data aggregation and redistribution is of concern, our oblivious approach eliminates this step to enable high bandwidth communication, in turn reducing impact on simulation performance. This approach also imposes no restrictions on the simulation runtime configuration or data structures, supporting $M : N$ configurations and arbitrary buffers of data.

From a practical standpoint, our implementation in libIS provides an easy-to-use C API that can be integrated into simulations with minimal changes to the simulation code. The data model does not require advanced networking functionality, and can use MPI or sockets to transfer data between the simulation and client. Although the sockets intercommunicator may not be able to take advantage of some high-speed interconnects, it ensures portability across different HPC or ad hoc clusters. The ease of use, flexibility, and portability of libIS also make it a useful building block for more full-featured libraries (e.g., SENSEI [2]), where libIS can serve as a base data transport layer.

In our evaluation we compared the separate and shared node configurations, and the MPI and socket intercommunicators on two HPC systems using two simulation codes. We found that the independent communication model employed by libIS weak scales well across the systems and network architectures. The fast data transfer to clients achieved by libIS means the simulation performance is not severely impacted when running the visualization on separate nodes. The library also allows clients to connect and disconnect as desired, which is suitable for spurious and on-demand visualization tasks.

Through our evaluation we have found further areas for improvement in libIS's networking model. When sending data to a client rank, each of the assigned $\frac{M}{N}$ simulations ranks sends its data serially to the client. This serialization of the data transfer leads to underutilizing the higher bandwidth network architectures, and makes it more likely for the simulation compute time to become bound by the libIS data transfer time, as observed with Poongback. Parallelizing the data transfer within each set of simulation ranks and client would further improve network utilization, and alleviate these issues.

In practice, we recommend using the separate node configuration for long-lived visualization tasks, and the shared node configuation for short-lived on-demand tasks. Short-lived tasks do not impact the simulation performance for a long period, and from a practical standpoint it may not be possible to start an additional job quickly enough to run the task in time. Oversubscribing the nodes for a short period to run the task is a better option than blocking the simulation until the task starts, or missing the desired timestep entirely. Potential future changes to HPC job schedulers to enable co-scheduling [6, 7] could make it easier to run on-demand tasks on separate nodes to reduce the impact of the visualization.

Acknowledgments

This work is supported in part by the Intel Graphics and Visualization Institute of eXcellence at the Scientific Computing and Imaging Institute, University of Utah and at the Texas Advanced Computing Center at the University of Texas at Austin. This work is supported in part by NSF: CGV Award: 1314896, NSF:IIP Award: 1602127, NSF:ACI Award: 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375 and NSF:OAC Award: 1842042. The software development of PoongBack is supported by NSF PetaApps grants: OCI-0749223 and NSF PRAC Grant 0832634. 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 TACC for providing HPC resources that have contributed to the research results reported in this paper. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

References

1. Ahrens, J., Jourdain, S., O'Leary, P., Patchett, J., Rogers, D.H., Petersen, M.: An Image-Based Approach to Extreme Scale In Situ Visualization and Analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (2014)
2. Ayachit, U., Whitlock, B., Wolf, M., Loring, B., Geveci, B., Lonie, D., Bethel, E.W.: The SENSEI Generic In Situ Interface. In: Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization, ISAV '16 (2016)
3. Bennett, J.C., Abbasi, H., Bremer, P.T., Grout, R., Gyulassy, A., Jin, T., Klasky, S., Kolla, H., Parashar, M., Pascucci, V.: Combining In-Situ and In-Transit Processing to Enable Extreme-Scale Scientific Analysis. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (2012)
4. Dayal, J., Bratcher, D., Eisenhauer, G., Schwan, K., Wolf, M., Zhang, X., Abbasi, H., Klasky, S., Podhorszki, N.: Flexpath: Type-Based Publish/Subscribe System for Large-Scale Science Analytics (2014-05)

5. Docan, C., Parashar, M., Klasky, S.: DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. Cluster Computing (2012)
6. Dorier, M., Dreher, M., Peterka, T., Wozniak, J.M., Antoniu, G., Raffin, B.: Lessons Learned from Building In Situ Coupling Frameworks. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (2015)
7. Dorier, M., Yildiz, O., Peterka, T., Ross, R.: The Challenges of Elastic In Situ Analysis and Visualization. In: Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV '19. Denver, Colorado (2019)
8. Ellsworth, D., Green, B., Henze, C., Moran, P., Sandstrom, T.: Concurrent Visualization in a Production Supercomputing Environment. IEEE Transactions on Visualization and Computer Graphics (2006)
9. Fabian, N., Moreland, K., Thompson, D., Bauer, A., Marion, P., Geveci, B., Rasquin, M., Jansen, K.: The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library. In: Symposium on Large Data Analysis and Visualization (LDAV) (2011)
10. Habib, S., Pope, A., Finkel, H., Frontiere, N., Heitmann, K., Daniel, D., Fasel, P., Morozov, V., Zagaris, G., Peterka, T., et al.: HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures. New Astronomy (2016)
11. Kress, J., Klasky, S., Podhorszki, N., Choi, J., Childs, H., Pugmire, D.: Loosely Coupled In Situ Visualization: A Perspective on Why It's Here to Stay. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, ISAV 2015 (2015)
12. Larsen, M., Woods, A., Marsaglia, N., Biswas, A., Dutta, S., Harrison, C., Childs, H.: A Flexible System for In Situ Triggers. In: Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV '18. Dallas, Texas (2018)
13. Lee, M., Malaya, N., Moser, R.D.: Petascale Direct Numerical Simulation of Turbulent Channel Flow on Up to 786K Cores. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC'13 (2013)
14. Lee, M., Moser, R.: Direct Numerical Simulation of Turbulent Channel Flow up to $Re_\tau \approx 5200$. Journal of Fluid Mechanics (2015)
15. Lee, M., Ulerich, R., Malaya, N., Moser, R.D.: Experiences from Leadership Computing in Simulations of Turbulent Fluid Flows. Computing in Science Engineering (2014)
16. Lofstead, J., Zheng, F., Klasky, S., Schwan, K.: Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In: IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009 (2009)
17. Malaya, N., McDougall, D., Michoski, C., Lee, M., Simmons, C.S.: Experiences Porting Scientific Applications to the Intel (KNL) Xeon Phi Platform. In: Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (2017)
18. Moreland, K., Oldfield, R., Marion, P., Jourdain, S., Podhorszki, N., Vishwanath, V., Fabian, N., Docan, C., Parashar, M., Hereld, M.: Examples of In Transit Visualization. In: Proceedings of the 2nd International Workshop on Petascal Data Analytics: Challenges and Opportunities (2011)
19. Plimpton, S.: Fast Parallel Algorithms for Short-Range Molecular Dynamics. Journal of computational physics (1995)
20. Sandia National Laboratories: LAMMPS Molecular Dynamics Simulator. URL `lammps.sandia.gov`
21. Sandia National Laboratories: Coupling LAMMPS to Other Codes (Accessed Jan. 2020). URL `lammps.sandia.gov/doc/Howto_couple.html`
22. Schroeder, W., Martin, K., Lorenson, B.: The Visualization Toolkit, 4 edn. Kitware (2006)
23. 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, ISAV'18 (2018)
24. Usher, W., Wald, I., Amstutz, J., Günther, J., Brownlee, C., Pascucci, V.: Scalable Ray Tracing Using the Distributed FrameBuffer. Computer Graphics Forum (2019)

25. Usher, W., Wald, I., Knoll, A., Papka, M., Pascucci, V.: In Situ Exploration of Particle Simulations with CPU Ray Tracing. Supercomputing Frontiers and Innovations (2016)
26. Vishwanath, V., Hereld, M., Morozov, V., Papka, M.E.: Topology-Aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM (2011)
27. 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)
28. Whitlock, B., Favre, J.M., Meredith, J.S.: Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In: Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization, EGPGV '11 (2011)
29. Zanúz, H.C., Raffin, B., Mures, O.A., Padrón, E.J.: In-Transit Molecular Dynamics Analysis with Apache Flink. In: Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization - ISAV '18. Dallas, Texas (2018)
30. Zhang, F., Lasluisa, S., Jin, T., Rodero, I., Bui, H., Parashar, M.: In-Situ Feature-Based Objects Tracking for Large-Scale Scientific Simulations. In: High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion (2012)
31. Zheng, F., Zou, H., Eisenhauer, G., Schwan, K., Wolf, M., Dayal, J., Nguyen, T.A., Cao, J., Abbasi, H., Klasky, S., Podhorszki, N., Yu, H.: FlexIO: I/O Middleware for Location-Flexible Scientific Data Analytics (2013)