# Supercomputing Frontiers and Innovations

**2016, Vol. 3, No. 4**

## Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

## Editorial Board

# Contents

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 [1, 2], 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 [3]. Much in situ research has emphasized sidestepping IO through specific analysis, data reduction or filtering [4, 5], optimizations to existing IO frameworks enabling scalable co-processing, streaming or offline visualization [6–8], and data forwarding mechanisms coupling simulations with production visualization software [9]. 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 [10], or asynchronous *loosely-coupled* approaches that forward data from the simulation into separate visualization processes (e.g. [11]), either on the same machine or a different cluster. In either case, in situ rendering or analysis requires consideration of distributed data spread across multiple simulation nodes and potentially too large to be marshaled to a single node for processing.

[1]SCI Institute, University of Utah, Salt Lake City, USA
[2]Intel Corporation, Santa Clara, USA
[3]Argonne National Laboratory, Lemont, USA

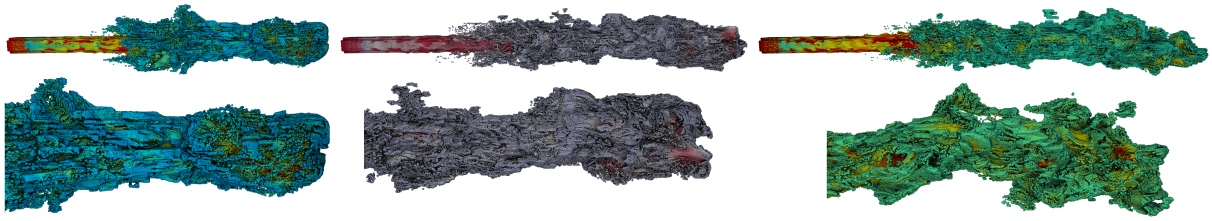


**Figure 1.** A coal particle combustion simulation in Uintah at three different timesteps with (left to right): 34.61M, 48.46M and 55.39M particles, with attribute based culling showing the full jet (top) and the front in detail (bottom). Using our in situ library to query and send data to our rendering client in OSPRay these images are rendered interactively with ambient occlusion, averaging around 13 FPS at $1920 \times 1080$. 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

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 [12], 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 [13], 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 [14, 15] and Uintah [16]; 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. [5] 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 [17, 18].

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. [17] 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 [18]. 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 [9].

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 [19]. In an effort to require no modification to existing simulation code Fogal et al. [20] 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 [17, 19] or allow modification of analysis parameters on the fly [9].

### 1.2. In Situ Visualization

In situ visualization has long been proposed as an alternative to offline visualization, first discussed by McCormick et al. [21]. 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 [23] were designed to directly integrate with the simulation and allow for computational steering. VMD [24] 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 [25]. A loosely coupled in situ renderer for weather forecast models is described by Ellsworth et al. [26] 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. [11] 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. [10] 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 [9, 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 [27]. 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 [13, 28] on CPU. On GPU, MegaMol [29] 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 [30].

### 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. [31, 32] 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 [33, 34].

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 [11]. Our own work builds on the balanced P-k-d method of Wald et al. [13], 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 [12] 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 [13], 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* [31]. 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

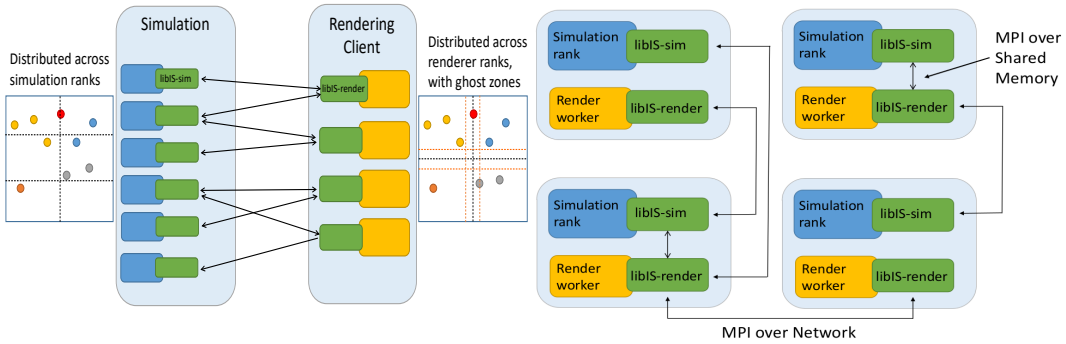



**Figure 2.** Overview of the in situ library, libIS. In our system, data are forwarded via MPI from the simulation to a distributed renderer. When the simulation and visualization are run on different resources (left) all data queries go over the network. When running on the same resource (right), data 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 [13] 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 [12]) 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-

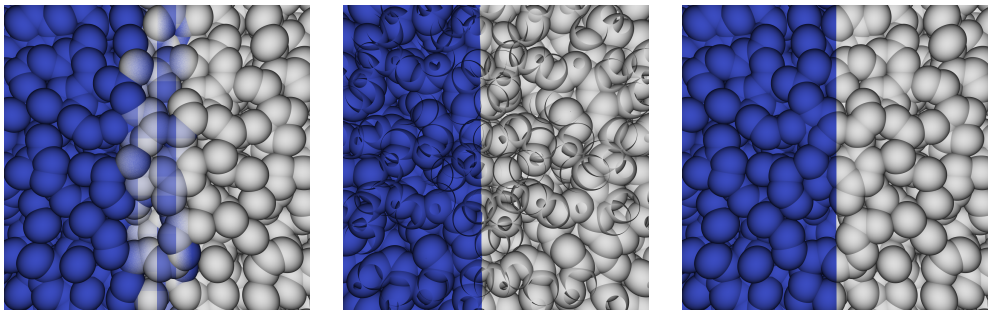


**Figure 3.** Without clipping nodes render overlapping regions, resulting in compositing artifacts (left). Always clipping rays against the node's domain (middle) incorrectly clips spheres on unshared boundaries. By extending unshared boundaries of the domain and clipping primary rays to the partially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

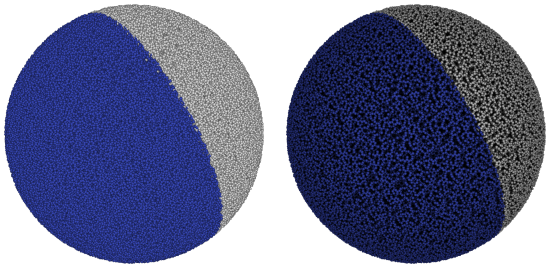


**Figure 4.** Depth perception of raycasting (left) vs. ambient occlusion (right) on a 1M atom nanosphere

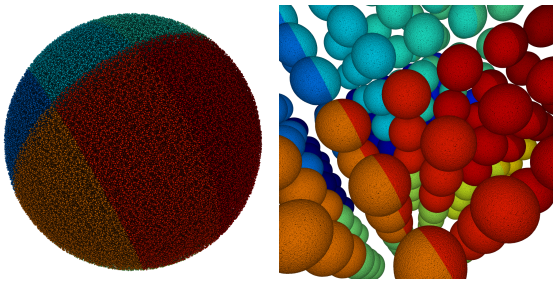


**Figure 5.** Replicated nanosphere scaling datasets used in LAMMPS with 1.05M (left) to 227M (right) atoms

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 [35], 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 [36].

**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 [37], synthetically replicated in a uniform grid to evaluate how our system scales with data size. The Uintah [16] 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.

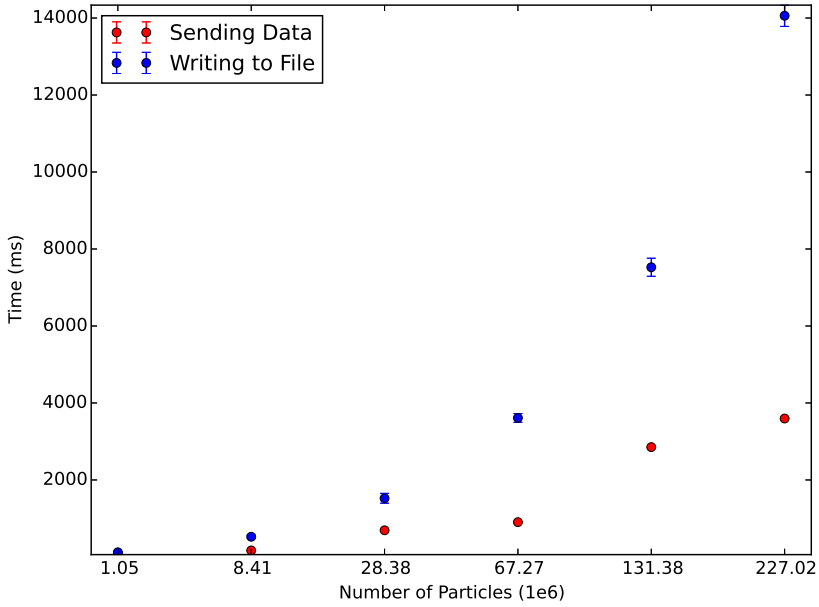
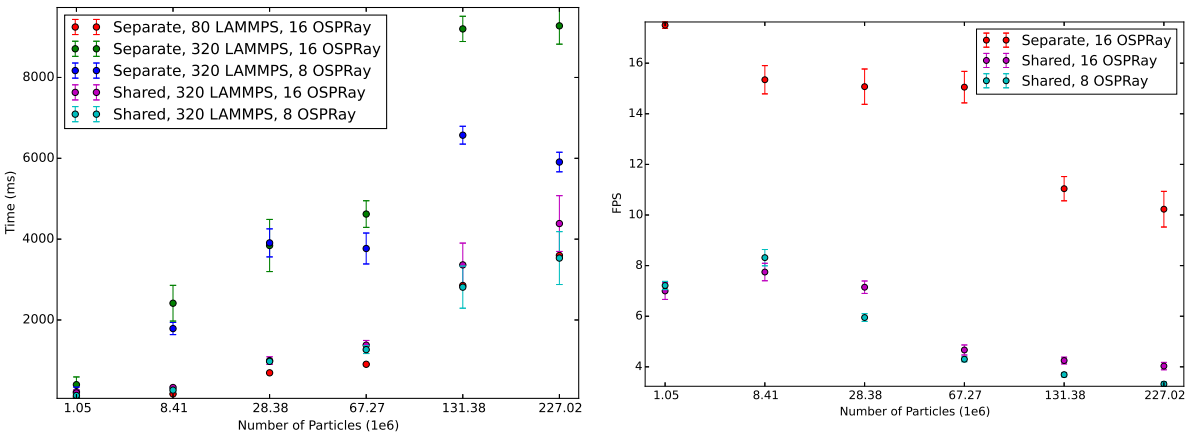



**Figure 6.** Sending data vs. writing files for a separate run with 80 LAMMPS ranks sending to 16 OSPRay ranks. We did not measure file writing beyond 67M particles as it becomes too time consuming




a) Data send times

b) Framerate with a $1004 \times 1024$ 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 [9].

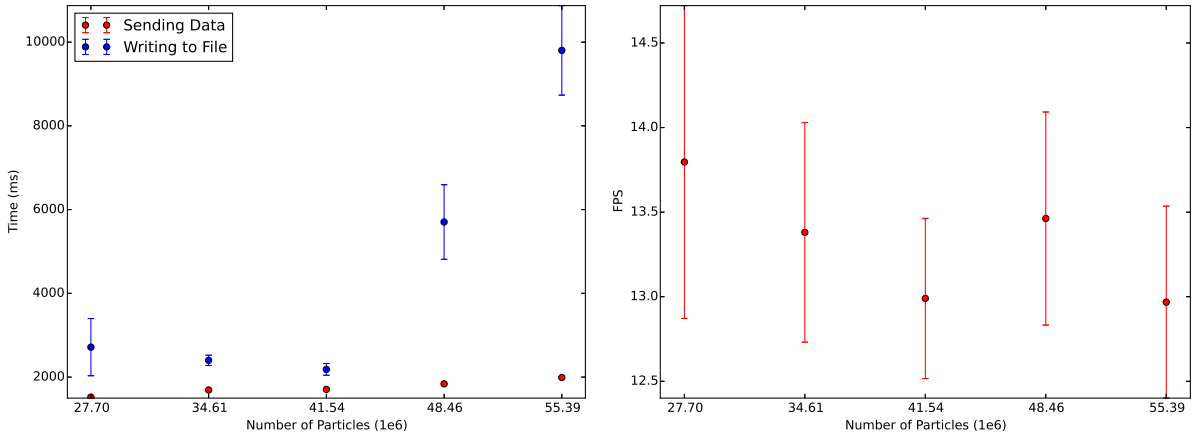
We measure scaling with data size in two configurations on Maverick. We run LAMMPS with 320 ranks (16 nodes) and the renderer on 8 or 16 of the same nodes and examine performance as the data size increases. As expected, we find that the framerate of both shared configurations is lower than the separate 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 $4\times$ 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 $1920 \times 1080$ 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 [7] and ADIOS [8] or optimized distributed-parallel compositors [33, 38], 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 [6, 33, 38] 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. Lindstrom P. Fixed-Rate Compressed Floating-Point Arrays. IEEE Transactions on Visualization and Computer Graphics. 2014;20(12):2674–2683.

2. Burtscher M, Ratanaworabhan P. pFPC: A parallel compressor for floating-point data. In: Data Compression Conference; 2009. p. 43–52.

3. Childs H. In Situ Terminology Project, https://ix.cs.uoregon.edu/ hank/insituterminology;.

4. Peterka T, Morozov D, Phillips C. 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; 2014. p. 997–1007.

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

6. Kumar S, Vishwanath V, Carns P, Summa B, Scorzelli G, Pascucci V, et al. PIDX: Efficient Parallel I/O for Multi-resolution Multi-dimensional Scientific Datasets. In: Proceedings of The IEEE International Conference on Cluster Computing; 2011. p. 103–111.

7. Vishwanath V, Hereld M, Morozov V, Papka ME. 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; 2011. p. 19:1–19:11.

8. 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. p. 1–10.

9. Fabian N, Moreland K, Thompson D, Bauer AC, Marion P, Geveci B, et al. The ParaView Coprocessing Library: A scalable, general purpose in situ visualization library. In: Symposium on Large Data Analysis and Visualization (LDAV). IEEE; 2011. p. 89–96.

10. Yu H, Wang C, Grout RW, Chen JH, Ma KL. In Situ Visualization for Large-Scale Combustion Simulations. IEEE Computer Graphics and Applications. 2010;30(3):45–57.

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

12. Wald I, Johnson G, Amstutz J, Brownlee C, Knoll A, Jeffers J, et al. OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. IEEE Transactions on Visualization and Computer Graphics. 2016;PP(99):1–1.

13. Wald I, Knoll A, Johnson GP, Usher W, Pascucci V, Papka ME. CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees. In: Proceedings of IEEE Visweek; 2015. .

14. Sandia National Labs. LAMMPS Molecular Dynamics Simulator;. Available from: `http://lammps.sandia.gov/`.

15. Plimpton S. Fast parallel algorithms for short-range molecular dynamics. Journal of computational physics. 1995;117(1):1–19.

16. Berzins M, Luitjens J, Meng Q, Harman T, Wight CA, Peterson JR. Uintah: a scalable framework for hazard analysis. In: Proceedings of the TeraGrid Conference. ACM; 2010. p. 3.

17. Peterka T, Kwan J, Pope A, Finkel H, Heitmann K, Habib S, et al. Meshing the Universe: Integrating Analysis in Cosmological Simulations. In: High Performance Computing, Networking, Storage and Analysis (SCC), SC Companion; 2012. p. 186–195.

18. 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. p. 736–740.

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

20. Fogal T, Proch F, Schiewe A, Hasemann O, Kempf A, Krüger J. Freeprocessing: Transparent in situ visualization via data interception. In: Eurographics Symposium on Parallel Graphics and Visualization (EGPGV); 2014. .

21. McCormick BH, DeFanti TA, Brown MD. Visualization in scientific computing. IEEE Computer Graphics and Applications. 1987;7(10):69–69.

22. Parker SG, Johnson CR. SCIRun: a scientific programming environment for computational steering. In: Proceedings of the ACM/IEEE conference on Supercomputing; 1995. .

23. Haimes R, Edwards DE. Visualization in a parallel processing environment. In: Proceedings of the 35th AIAA Aerospace Sciences Meeting, number AIAA Paper; 1997. p. 97–0348.

24. Humphrey W, Dalke A, Schulten K. VMD: visual molecular dynamics. Journal of Molecular Graphics. 1996;14(1):33–38.

25. Tu T, Yu H, Ramirez-Guzman L, Bielak J, Ghattas O, l Ma K, et al. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In: SC Conference, Proceedings of the ACM/IEEE; 2006. p. 12–12.

26. 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;12(5):997–1004.

27. Gross M, Pfister H. Point-Based Graphics. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2007.

28. Gribble CP, Ize T, Kensler A, Wald I, Parker SG. A coherent grid traversal approach to visualizing particle-based simulation data. IEEE Transactions on Visualization and Computer Graphics. 2007;(4):758–768.

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

30. Fraedrich R, Schneider J, Westermann R. Exploring the Millennium Run - Scalable Rendering of Large-Scale Cosmological Datasets. IEEE Transactions on Visualization and Computer Graphics. 2009;15(6):1251–1258.

31. Hsu WM. Segmented Ray Casting for Data Parallel Volume Rendering. In: Proceedings of the Symposium on Parallel Rendering; 1993. p. 7–14.

32. Ma KL, Painter JS, Hansen CD, Krogh MF. A Data Distributed, Parallel Algorithm for Ray-traced Volume Rendering. In: Proceedings of the Symposium on Parallel Rendering; 1993. p. 15–22.

33. Grosset AVP, Prasad M, Christensen C, Knoll A, Hansen C. 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. Eurographics Association; 2015. p. 67–76.

34. Moreland K, Kendall W, Peterka T, Huang J. An Image Compositing Solution at Scale. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. ACM; 2011. p. 25:1–25:10.

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

36. Ancel A, Dischler JM, Mongenet C. Load-Balanced Multi-GPU Ambient Occlusion for Direct Volume Rendering. In: Childs H, Kuhlen T, Marton F, editors. Eurographics Symposium on Parallel Graphics and Visualization. The Eurographics Association; 2012. .

37. Pol VG, Wen J, Lau KC, Callear S, Bowron DT, Lin CK, et al. Probing the evolution and morphology of hard carbon spheres. Carbon. 2014;68:104–111.

38. Rizzi S, Hereld M, Insley J, Papka ME, Uram T, Vishwanath V. 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. Eurographics Association; 2015. p. 1–10.

DOI: 10.14529/jsfi160402

# In Situ Visualization and Production of Extract Databases

***Brad J. Whitlock***[1]***, Earl P. N. Duque***[2]



Simulations running at high concurrency on HPC systems generate large volumes of data that are impractical to write to disk due to time and storage constraints. Applications often adapt by saving data infrequently, resulting in datasets with poor temporal resolution. This can make datasets difficult to interpret during post hoc visualization and analysis, or worse, it can lead to lost science. *In Situ* visualization and analysis can enable efficient production of small data products such as rendered images or surface extracts that consist of polygonal geometry plus fields. These data products are far smaller than their source data and can be processed much more economically in a traditional *post hoc* workflow using far fewer computational resources. We used the SENSEI and Libsim *in situ* infrastructures to implement rendering workflow and surface data extraction workflows in the AVF-LESLIE combustion code. These workflows were then demonstrated at high levels of concurrency and showed significant data reductions and limited impact on the simulation runtime.

*Keywords: In Situ, High Performance Computing, Visualization, Extract Database, SENSEI, Libsim, Workflow.*

## Introduction

Today's large scale simulations run on HPC systems and generate far more data than can be practically saved or analyzed. HPC system design emphasises fast computations and I/O to and from these systems is often a secondary concern, leading to an asymmetry in which computed data often cannot be written to disk without resorting to strategies that sacrifice the temporal resolution of the data (saving infrequently). Recent developments explore the use of node local storage such as *burst buffers* that give applications a fast, convenient buffer to store results while they are staged out to the main I/O system. However, such hardware is not yet commonplace and other strategies such as *in situ* computations are emerging in production software as a mechanism to manage the data problem by reducing data that must be stored. Without *in situ*, the traditional *post hoc* workflow requires large simulation data to make costly trips to and from the I/O system, slowing both the simulation and later the visualization and analysis codes invoked to process the data. *In Situ* works by integrating analysis and visualization with the simulation so these operations may take place while the data are still in main memory, and are thus far less expensive to access. The data products produced *in situ*, whether they are statistics, rendered images, or even surface geometry extracts, are often orders of magnitude smaller than the memory-resident data and can be saved out far more economically.

In situ visualization is usually inexpensive enough to be applied frequently, actually improving access to spatio-temporal data that is often not saved or is undersampled due to time and storage constraints. Sometimes *in situ* does have limitations if the data products saved during the simulation run lack enough information for analysis. For instance, *in situ* is often ideal for creating statistics and rendered images but those data products may be less useful for actual exploration of data unless many images are saved. To permit better exploration of data after the fact, *in situ* can be extremely useful in the generation of extract databases. An extract database consists of extracted polygonal geometry plus scalar and vector field data. Extract databases can drastically reduce the amount of data being saved while still providing enough information

[1]bjw@ilight.com, Intelligent Light, Rutherford, New Jersey, USA
[2]epd@ilight.com, Intelligent Light, Rutherford, New Jersey, USA

to perform useful *post hoc* visualization and data analysis. The use of extract databases enables flexibility since most of the costly data reductions occur *in situ* and the reduced datasets can be visualized on more modest compute resources. By storing the geometry and fields in an extract database, it is still possible to create derived fields, render the geometry from various viewpoints, perform surface integrations, and many other operations that would not be possible on strictly image-based data products. The potential for massive data reduction, massive time eliminated when extracting dataset features, and the ability to later visualize features of interest, as opposed to static rendered images, are what make extract databases compelling.

In this study, we implement a workflow that uses *in situ* to perform both rendering and extract database generation to highlight "interesting" features in a turbulence simulation and save them out for later analysis in a visualization tool. The combination of *in situ* to avoid the time and storage costs of massive I/O and the ability to perform further analysis or rendering on the generated data produces a powerful, streamlined workflow.

## 1. Background

AVF-LESLIE [11, 12] is a reactive flow solver used for Direct Numerical Simulation or Large Eddy Simulation (DNS/LES) investigation of canonical reactive flows. It solves the reactive multi-species compressible Navier-Stokes equations using a finite volume discretization upon a Cartesian grid. We used AVF-LESLIE to simulate an unsteady, turbulent mixing layer (TML) between two fluids. The simulation demonstrates the evolution of turbulent, braided flow structures (our features of interest) that form as the system breaks down into homogeneous turbulence. AVF-LESLIE can output results to PLOT3D or HDF5/Xdmf format for later analysis and visualization.

The quick evolution of turbulent structures requires access to many simulation time steps to produce a faithful visualization. For the TML case, each time step produced many gigabytes of data. This made it impractical to save enough time steps to produce a times series suitable for purposes such as animation. The sheer size of the saved volume datasets would quickly overwhelm available disk space and make post-processing the results require as much compute resources as the original solver to fit the solution in memory and read it back from disk in a reasonable time.

Previous integration of VisIt's Libsim library [6, 13] into solvers such as CREATE-AV$^{TM}$ Kestrel to enable *in situ* generation of surface extracts yielded good results [14]. When run on 1024 cores and saving isosurfaces of structured and unstructured grid data for helicopter geometries, the coupled Kestrel/Libsim was able to frequently output extracts while using no more then 2-3% of the solver runtime. As the simulation produced extracts, a separate visualization job processed them into images, resulting in an efficient automated workflow.

The same *in situ* workflow would be applicable to combat the challenges of isolating features of interest from AVF-LESLIE's TML flow field. Polygonal surface geometry and the fields defined on those surfaces of interest are extracted and exported to FieldView *eXtract DataBase* (XDB) files for later *post hoc* analysis and rendering using Intelligent Light's parallel FieldView [2] software. XDB files are designed to save line and mesh geometries and the associated scalar and vector fields defined on those geometries. XDB files preserve numerical precision of the stored data making it possible to perform accurate analysis such as surface integration using the extract data in lieue of the original volume data. The resulting workflow decouples feature extraction

from rendering. This enables scheduling flexibility and it lets each phase of the workflow use different amounts of compute resources.

The separation of feature extraction and actual visualization into distinct phases that run asynchronously is a feature that is shared by certain in transit infrastructures. In Transit infrastructures such as ADIOS [8] provide an I/O interface to the simulation, which then "writes" its data to other compute resources and then continues. This has the advantage of not stalling the simulation while extracting data or rendering. However, this approach requires additional compute resources to receive the simulation data and further process it, which might be a downside when operating at the limits of the compute resource. A more problematic downside of I/O based systems is that they can be too low-level. I/O interfaces such as ADIOS provide functions for writing arrays. Complicated data structures such as meshes often consist of multiple arrays to represent coordinates and connectivity. Such data structures are encoded into multiple data arrays using various conversions. By necessity, analysis routines that run on the other side of the in transit pipeline must support conventions to reassemble array data back into useful mesh structures. Paraview Catalyst [1, 5] is a powerful *in situ* infrastructure that also supports in transit. Data are exposed to Catalyst as VTK [10] datasets via user-provided adaptor code that is developed for each simulation. The adaptor code defines how simulation data are translated into VTK datasets and permits simulation data arrays to be wrapped as VTK data arrays, allowing zero-copy data passing. Once the data are represented as VTK datasets, they can be operated on by user-defined rendering and data analysis pipelines or they may be shipped to other compute resources for in transit visualization and data analysis. Though it does not have its own in transit mode, VisIt's Libsim otherwise provides similar functionality to Catalyst and provides rendering, data analysis, and extract functionality. VisIt's large set of plots and operators enable the creation of complicated visualization pipelines that can be used for rendering and data extraction. In addition, VisIt provides an export plug-in to the FieldView XDB file format, which enables analysis and rendering of extract data using FieldView.

These infrastructures can all form the building blocks of higher-level infrastructures. For example, Cinema [3] assembles images that are produced *in situ* into databases that can be interactively explored in a lightweight viewer. The images are produced by a simulation instrumented with Catalyst, or another suitable *in situ* infrastructure. SENSEI [4] is another higher-level infrastructure. SENSEI provides a unified interface to multiple *in situ* infrastructures, including ADIOS, Catalyst, and Libsim. SENSEI simplifies the process of *in situ* instrumentation by providing data abstractions that enable creation of a write-once simulation adaptor to expose simulation data structures to SENSEI using the VTK data model. Once inside SENSEI, the VTK data can be passed to any of the coupled *in situ* infrastructures, even combining multiple infrastructures in a single analysis. For the purposes of this analysis, we coupled AVF-LESLIE through SENSEI to enable future flexibility in connecting to infrastructures such as Catalyst or ADIOS. Once instrumented using SENSEI, we selected the Libsim analysis back end to permit the creation of FieldView XDB files. XDB files were read and processed as they were created by separate FieldView jobs running on a workstation.

## 2. Instrumentation

Producing a working instrumented version of AVF-LESLIE capable of running on a large scale HPC system required changes to several software packages.

## 2.1. VisIt / Libsim

VisIt is a visualization and analysis software package that was started at Lawrence Livermore National Laboratory in the year 2000. From the start VisIt was designed to work efficiently on large distributed-memory HPC systems. As such, VisIt's compute server runs in parallel using MPI message passing to coordinate multiple processes that each operate on a subset of the overall dataset. VisIt's compute server is architected such that it can be loaded dynamically from simulations via the Libsim library, a VisIt library that is added to simulations to enable them to perform VisIt-enabled *in situ* rendering and analysis. VisIt is normally built using shared libraries which are dynamically loaded when Libsim detects that *in situ* operations are requested. This approach works well up to a few thousand cores after which the file system may introduce delays while loading VisIt's shared libraries and plugins.

The ultimate target concurrency for this study would be in the range of tens of thousands to hundreds of thousands of cores so long delays incurred loading shared libraries would not be acceptable. To avoid such delays, we modified to VisIt's CMake-based static build process to create a statically-built version of Libsim. The static Libsim includes all VisIt libraries and plugins in a single library that simplifies addition of VisIt functionality into simulations such as AVF-LESLIE.

## 2.2. XDB Library

VisIt's FieldView XDB export capability uses an export plugin that passes VisIt's internal VTK datasets to the XDB library for writing FieldView XDB files for later consumption by FieldView. The XDB library enables the client program to create extract entities such as streamline rakes or polygonal surfaces, record relevant metadata, and define scalar and vector fields on those geometries. The format preserves the precision of the data when saved so it can be used for analyses that would have been appropriate for the original volume data. The metadata saved by the XDB format provides clues to Fieldview about how the surfaces were generated during the data extraction process. For instance, an isosurface extract will record the variable and isocontour value used to generate the surface.

Prior enhancements to the XDB export plugin in VisIt enabled support for *write groups* (described in [7]), which let VisIt perform partial data aggregation during production of XDB files. However, improved support for parallel writes was desired and the XDB library was not designed for parallel. Consequently, we undertook a redesign of the library that would decouple the data model that describes the XDB surface extracts from the methods used to write and read the data. This effort yielded a second version of the XDB library, which could read/write the XDB format while providing a future path to using parallel data transports such as ADIOS. The resulting XDB library also can represent its data objects using zero-copy constructs that support various memory layouts, a feature necessary to reduce overhead when running *in situ*. The VisIt XDB export plugin was enhanced to use the new version of the XDB library and this was the version used during instrumentation of AVF-LESLIE.

## 2.3. Integration with AVF-LESLIE

SENSEI enables simulations to integrate with multiple *in situ* infrastructures while creating just one code adaptor. An adaptor exposes simulation data structures so their data may be used by the *in situ* infrastructure. We modified previous adaptor code that had been created to

integrate directly with Libsim so AVF-LESLIE could also integrate SENSEI, opening the door to future workflows where ADIOS is used for in transit data staging to a separate "endpoint" analysis program running on an alternate set of compute nodes. At this time, we have not yet attempted using that feature but have verified that our XDB workflow continues to operate using the SENSEI infrastructure.

The SENSEI adaptor follows a pattern that requires just a few insertions of extra adaptor functions into AVF-LESLIE's main routine: an *initialize* function, a *coprocess* function, and a *finalize* function. The initialize function is used to set up SENSEI, which in turn performs the relevant initialization for its subservient *in situ* analysis infrastructures. The coprocess function passes pointers to AVF-LESLIE arrays to some book-keeping code that associates the buffer addresses and sizes with variable names. The coprocess function's role is to package the simulation's buffers (zero-copy when possible) as a VTK curvilinear grid dataset with various fields defined on its nodes. For the purposes of this study, the adaptor also calculates the vorticity field, which is not computed in the solver, yet is needed to identify vortex features of interest for *in situ* rendering and data extraction. After packaging the simulation data as a VTK dataset, the VTK dataset is passed through SENSEI to a secondary analysis adaptor that shares the dataset with an analysis infrastructure, in this case Libsim. The analysis adaptor created for Libsim accepts the VTK dataset from the SENSEI analysis adaptor and exposes that data to VisIt via Libsim function calls and adaptor callback functions. In addition, the adaptor creates a set of VisIt plots based upon commands from an extract input file and uses those plots to export the desired extract surfaces into XDB files. Plots can also be restored from a VisIt session file which is an XML file that is useful for creating visualizations that can be rendered since plot attributes and colors are preserved. The finalize function is called when AVF-LESLIE has completed its main loop and needs to clean up prior to exiting. The overall schematic for the instrumented simulation is shown in fig. 1.

## 3. Performance

In this study, AVF-LESLIE was configured to simulate unsteady dynamics of a temporally evolving planar mixing layer at the interface between two fluids, a configuration that gives rise to a temporal mixing layer (TML). This type of fundamental flow mimics the dynamics encountered when two fluid layers slide past one another and is found in atmospheric and ocean fluid dynamics as well as combustion and chemical processing. The two sliding fluid layers are subject to inviscid instabilities and can evolve from largely 2D laminar flow into fully developed, 3D homogeneous turbulent flow as shown in [9]. Visualizations of the TML flowfield in fig. 2 show isosurfaces of the vorticity field, at 10,000, 50,000, 100,000, and 200,000 time steps where the flow evolves from the initial flow field, vortex braids begin to form, wrap and then the flow breaks down leading to homogeneous turbulence, respectively.

We conducted scaling studies using AVF-LESLIE on Titan at Oak Ridge Leadership Class Compute Facility. Titan is a Cray XK7 with 18,688 compute nodes, each containing a 16-core AMD Opteron CPU, 32GB of memory, and an Nvidia Tesla K20X GPU. The scaling studies used a Cartesian grid size of $1025^3$ and physical non-dimensional domain size of $4\pi$ x $4\pi$ x $2\pi$. The size of the $1025^3$ grid was held constant during scaling, resulting in a strong scaling study that reduces the average simulation workload per core as the number of cores increases. Two types of in situ computations were performed: a rendering workflow, and an extract-based XDB workflow.

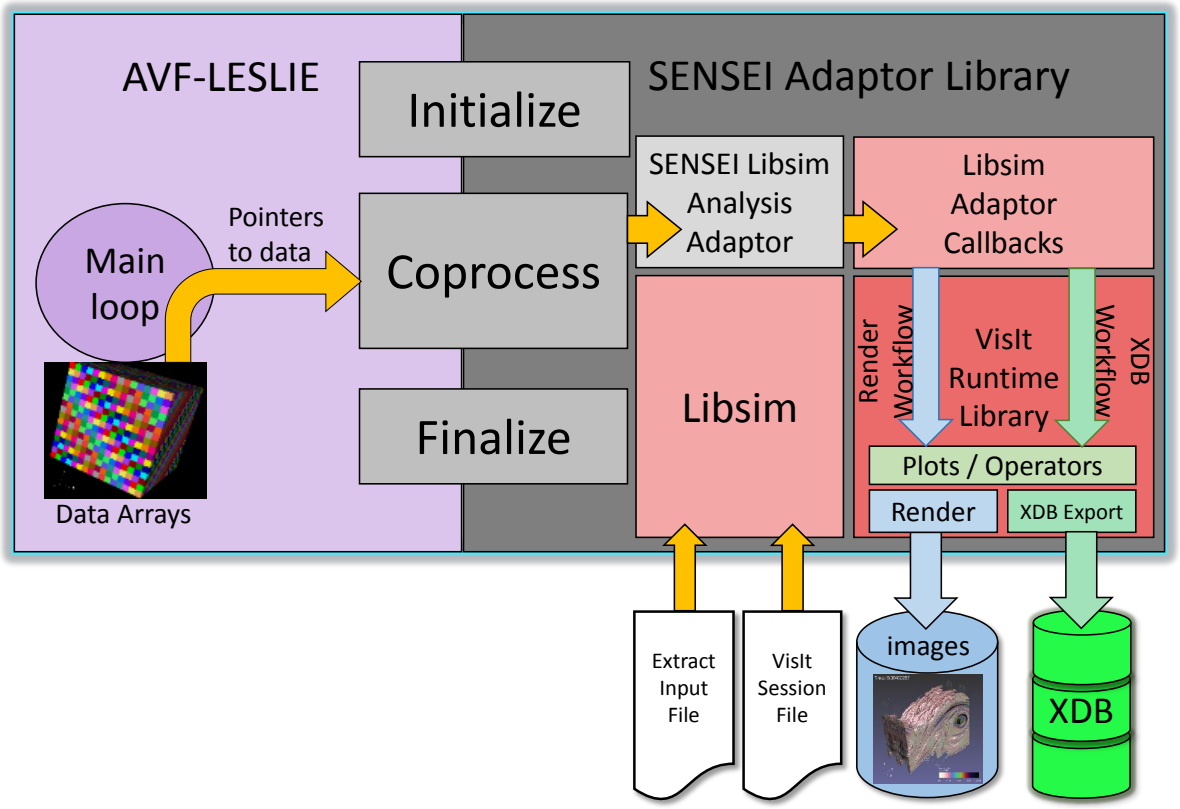



**Figure 1.** AVF-LESLIE simulation instrumented with SENSEI, VisIt/Libsim, and XDB library

### 3.1. Rendering Workflow

The rendering workflow was scaled up to 131,072 cores on Titan using all 16 cores in a compute node. The workflow demonstrated that the instrumented simulation can create and render a visualization based on the data directly from the solver memory as in [4]. In this case, the simulation was initialized using a VisIt session file, which directed Libsim to create 3 isosurfaces of the derived vorticity field and 3 planar slices. The visualization was rendered into a 1600x1600 pixel image image and saved to PNG format. Each measurement in the scaling study was performed using 100 time steps of AVF-LESLIE where *in situ* rendering was performed every 5th solver time step. Timing measurements were obtained from log files produced by the instrumented solver, which called the *MPI_Wtime()* library function around blocks of code being timed.

The timings measured time spent in the solver time step and overall time performing *in situ* rendering, which includes data extraction, rendering, image compositing, and image saving. The time spent in the solver decreased as the number of cores was increased due to the strong scaling induced by holding the grid size constant. It is worth noting that the complexity of the visualization resulted in long rendering times. The total time spent rendering is amortized over 5 solver time steps, resulting in an image saving cost on the order of 1-2 seconds when compared against the average solver time step. Also, the time spent on *in situ* increased with scale, largely due to image compositing.

Image compositing is an operation whereby full scale images from each MPI rank are combined in a tree-based reduction among all MPI ranks ultimately resulting in one MPI rank having a single image containing contributions from all of the input images on other MPI ranks.

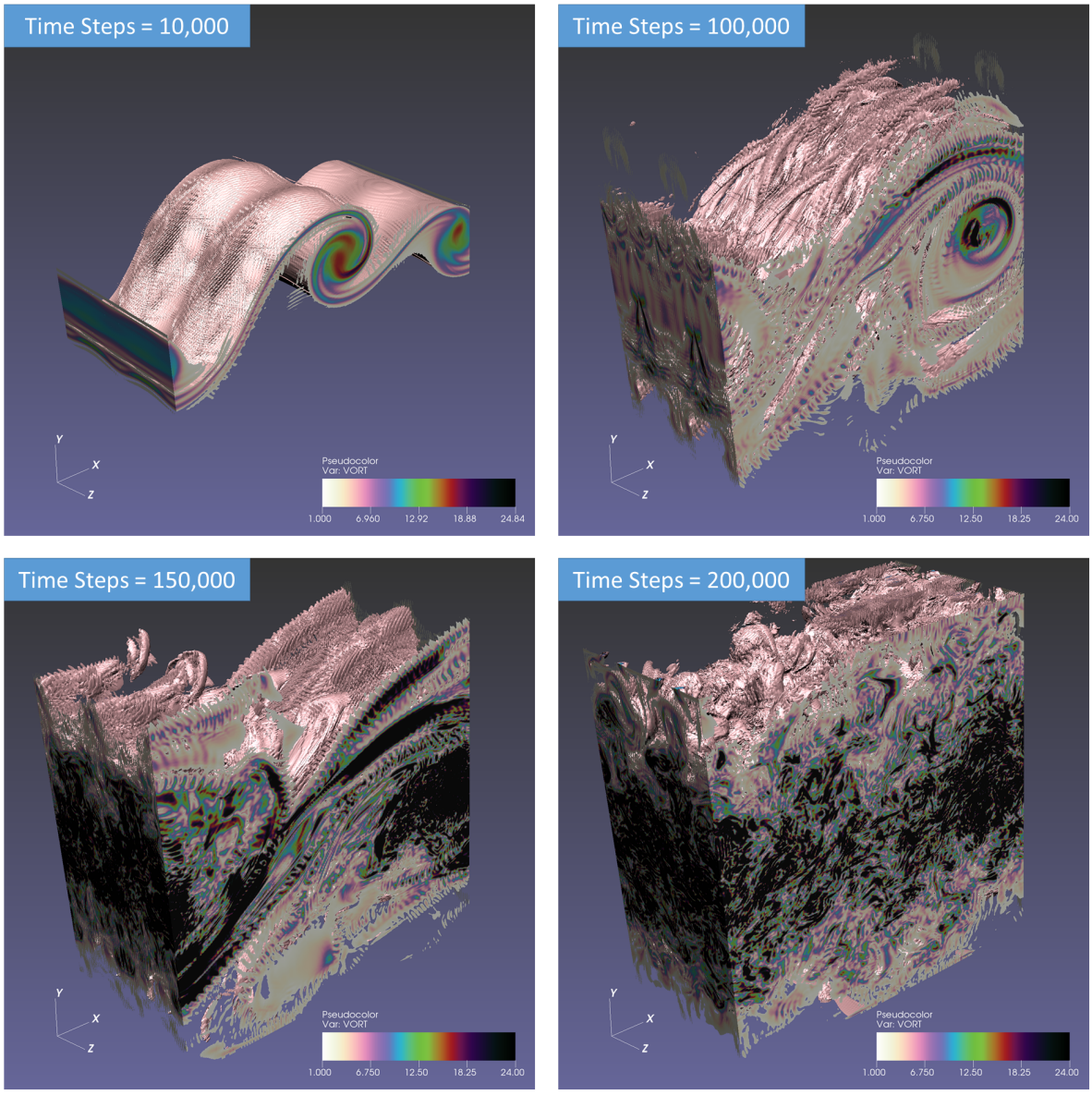



**Figure 2.** The Evolution of Temporal Mixing Layer from Initial to Vortex Breakdown

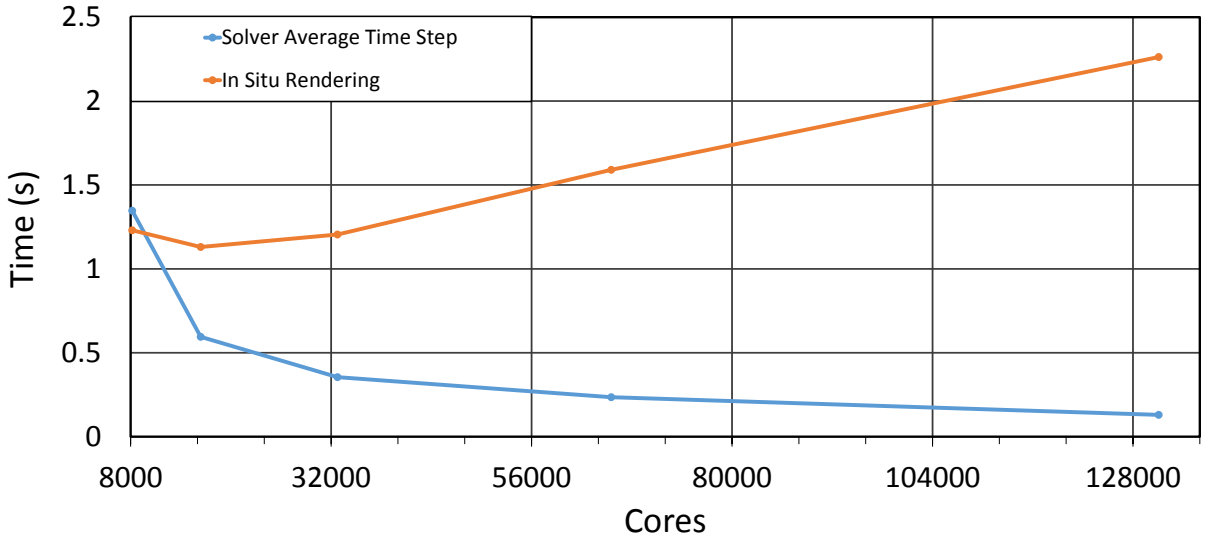



**Figure 3.** Strong Scaling Performance of AVF-LESLIE and *In Situ* Rendering Workflow for IsoSurface and Coordinate Cuts Rendered to 1600x1600 Pixel Image

Image compositing represents a different type of workload from the solver computation. As the number of processors increases, the number of images to be composited increases, as does the depth of the communication tree of processors which must communicate their images and depth buffers. Thus, image compositing can grow more expensive with larger numbers of processors as shown in fig. 3. In this study, VisIt relied on custom image compositing code which overlaps image communication with the actual pixel compositing operations. Unfortunately, the custom image compositing code appears to degrade in performance after 16K cores. In future work, Ice-T, a powerful image compositing library that is known to scale well and provides many communication optimizations could be used to further reduce the overhead associated with image compositing.

### 3.2. XDB Workflow

The XDB workflow was scaled up to 32,768 cores on Titan using all 16 cores of the allocated compute nodes. As with the rendering workflow the scaling numbers were obtained by running AVF-LESLIE for 100 time steps, with *in situ* operations every 5 solver time steps, and analysing its output logs. At each *in situ* time step, the AVF-LESLIE adaptor made Libsim function calls to create isosurfaces of vorticity and save them to a FieldView XDB file. The surface was specified using an input file to the adaptor which was interpreted and translated into Libsim function calls for creating a VisIt plot to generate an isosurface. Export to XDB was performed by partitioning the total MPI ranks with geometry into smaller write groups of 96 MPI ranks. Each write group aggregated isosurface geometry locally within the group to a single MPI rank responsible for writing a XDB file. When all groups completed writing their XDB file, the lead MPI rank wrote a FieldView layout file, which contains a list of XDB files to later read in parallel to recreate the entire surface extract geometry.

Over the course of the 100 time step runs, the time spent by AVF-LESLIE can be divided into three main categories: solver time steps, I/O, and *in situ*. The solver time steps represented the time that AVF-LESLIE spent solving and updating its physics variables. The AVF-LESLIE runs were configured to output an initial plot file in PLOT3D format, followed by additional

plot files every 100 time steps. Over the course of the short run, 2 sets of plot files were created where the size of each plot file was roughly 51842 MB as shown in tab. 1. Over the same run, the *in situ* routines created an isosurface of vorticity and wrote XDB files with the extract geometry every 5 time steps. The size of the average set of XDB files for a single *in situ* time step was between 260 MB and 266 MB, depending on the number of cores. When comparing the size of a single plot file containing volume data and a set of XDB files representing an isosurface extract, the extracts are around 200 times smaller. Given that the *in situ* extracts were written 20 times more frequently than the plot files, adding up the total size of the XDB extracts is still 10 times smaller than a single plot file.

**Table 1.** File Size Comparison

| Cores | 1 Volume File (MB) | 1 Extract (MB) | 1 Extract/ 1 Volume | 20 Extracts/ 1 Volume |
|---|---|---|---|---|
| 8192 | 51842 | 260 | 0.005 | 0.100 |
| 16382 | 51842 | 262 | 0.005 | 0.101 |
| 32768 | 51842 | 266 | 0.005 | 0.102 |
| | | | ˜200x reduction | ˜10x reduction |

The relative timings of solver time steps, I/O, and *in situ* are shown in fig. 4. The overwhelming majority of time is spent in the 2 calls to the I/O routines that write the full size PLOT3D files. That is followed by the time spent computing 100 solver time steps, followed at last by the *in situ* operations which were the least expensive in terms of time. When considering the actual overhead percentage added to the runtime of the simulation, for each 100 time steps, *in situ* added between 1.6% and 2.4% to the simulation runtime. If one was to consider an AVF-LESLIE run that performed absolutely no PLOT3D I/O then the overhead of *in situ* increases to between 21% and 28% for writing isosurface extracts every 5 solver time steps. As with any configurable operation, the overhead of *in situ* with respect to the overall runtime will depend on its frequency of use. In this case, for $1/30^{\text{th}}$ to $1/50^{\text{th}}$ the cost of full I/O, 10 times better temporal sampling was achieved and much less disk space was consumed by using *in situ* extracts. The time spent further post-processing the XDB extract files in FieldView on a 12-core MacOS X workstation with 64GB of memory was on the order of a few seconds per time step to produce rendered images.

## Conclusions

AVF-LESLIE was successfully instrumented for *in situ* using a combination of SENSEI, Libsim+VisIt, and the XDB library. The instrumented version of AVF-LESLIE can produce extract databases in the form of FieldView XDB files, which has proven to be a useful feature that greatly reduces the data that would otherwise need to be saved. The savings were measured in both time and storage costs. Extract databases took 2-3% of the solver runtime when also performing limited volume I/O and would further reduce the time spent during *post hoc* analysis. The entire size of the extract database was 10 times smaller than PLOT3D volume data while saving extracts 20 times more frequently. The surfaces in the extract database were able to be visualized on a workstation instead of requiring a visualization cluster coupled to the

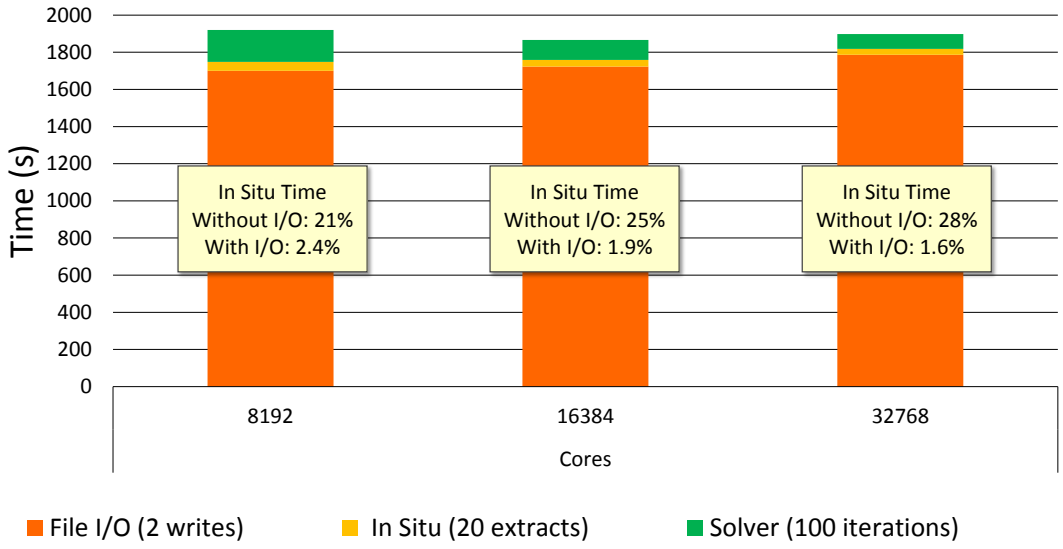



**Figure 4.** AVF-LESLIE Timings with *In Situ* Data Extraction

supercomputer. *In Situ* technologies are maturing to the point that they can begin to accelerate science and make the best use of HPC systems in spite of whatever I/O limitations might exist.

## Acknowledgments

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under award numbers DE-SC0007548 and DE-SC0012449. This research used resources of the Oak Ridge Leadership Computing Facility (OLCF).



## References

1. ParaView Web Site, http://www.paraview.org, June 2010.

2. FieldView 16, http://www.ilight.com, September 2016.

3. James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H. Rogers, and Mark Petersen. 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*, SC '14, pages 424–434, Piscataway, NJ, USA, 2014. IEEE Press.

4. Utkarsh Ayachit, Andrew Bauer, Earl P. N. Duque, Greg Eisenhauer, Nicola Ferrier, Junmin Gu, Kenneth Jansen, Burlen Loring, Zarija Lukić, Suresh Menon, Dmitriy Morozov, Patrick O'Leary, Michel Rasquin, Christopher P. Stone, Venkat Vishwanath, Gunther H. Weber, Brad J. Whitlock, Matthew Wolf, K. John Wu, and E. Wes Bethel. Performance

Analysis, Design Considerations, and Applications of Extreme-scale *In Situ* Infrastructures. In *Proceedings of SC16*, Salt Lake City, UT, USA, November 2016. *To appear.*

5. Andrew C. Bauer, Berk Geveci, and Will Schroeder. *The ParaView Catalyst User's Guide v2.0.* Kitware, Inc., 2015.

6. Henry R. Childs, E. Brugger, Brad J. Whitlock, Jeremy S. Meredith, Sean Ahern, Kathleen Biagas, Mark C. Miller, Gunther H. Weber, Cyrus Harrison, David Pugmire, Thomas Fogal, Christophe Garth, Allen Sanderson, E. Wes Bethel, Marc Durant, David Camp, Jean M. Favre, Oliver Rubel, and Paul Navratil. Visit: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*, pages 357–368. Chapman and Hall, 2012.

7. Earl P. N. Duque, Brad J. Whitlock, Steve M. Legensky, Christopher P. Stone, Reetesh Ranjan, and Suresh Menon. The impact of in situ data processing and analytics upon scaling of cfd solvers and workflows. In *27th International Conference on Parallel Computational Fluid Dynamics*, Montreal, Canada, 2015.

8. Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

9. Ralph W Metcalfe, Steven A Orszag, Marc E Brachet, Suresh Menon, and James J Riley. Secondary instability of a temporally growing mixing layer. *Journal of Fluid Mechanics*, 184:207–243, 1987.

10. Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics.* Kitware, Inc., fourth edition, 2004. ISBN 1-930934-19-X.

11. Thomas M. Smith and Suresh Menon. The structure of premixed flame in a spatially evolving turbulent flow. *Combustion Science and Technology*, 119, 1996.

12. Christopher P. Stone and Suresh Menon. Open loop control of combustion instabilities in a model gas turbine combustor. *Journal of Turbulence*, 4, 2003.

13. Brad J. Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109. Eurographics Association, 2011.

14. Brad J. Whitlock, James R. Forsythe, and Steve M. Legensky. In Situ Infrastructure Enhancements for Data Extract Generation. In *54th AIAA Aerospace Sciences Meeting, SciTech 2016*, pages 1–12, January 2016.

DOI: 10.14529/jsfi160403

# In situ, steerable, hardware-independent and data-structure agnostic visualization with ISAAC

*Alexander Matthes*[1,2], *Axel Huebl*[1,2], *René Widera*[1], *Sebastian Grottel*[2], *Stefan Gumhold*[2], *Michael Bussmann*[1]



The computation power of supercomputers grows faster than the bandwidth of their storage and network. In particular, applications using hardware accelerators like Nvidia GPUs cannot save enough data to be analyzed in a later step. There is a high risk of losing important scientific information. We introduce the in situ template library ISAAC which enables arbitrary applications like scientific simulations to live visualize their data without the need of deep copy operations or data transformation using the very same compute node and hardware accelerator the data is already residing on. Arbitrary meta data can be added to the renderings and user defined steering commands can be asynchronously sent back to the running application. Using an aggregating server, ISAAC streams the interactive visualization video and enables user to access their applications from everywhere. *Keywords: HPC, in situ, visualization, live rendering, petascale, particle-in-cell, C++11, CUDA, Alpaka, FOSS.*

## Introduction

Supercomputers are getting faster every year. 2016 the National Supercomputing Center in Wuxi released its Sunway TaihuLight with over 93 PFLOP/s peak performance in the LINPACK Benchmark [23]. The system provides over 1.3 petabytes main memory enabling scientists having a glimpse at the upcoming exascale area.

Unfortunately neither the mass memory, nor the network interconnect, not even the system's internal interconnect (e.g. PCIe) are fast enough to be able to transfer or save all data created in high performance computing (HPC) applications, e.g. scientific simulations, running on such systems without unacceptable latencies. Consequently only a small subset of the processed data can be analyzed afterwards with the classical post processing approach which means loosing maybe important scientific data.

One solution is to analyze the data on the same system they are produced on while they are produced, called in situ processing. The benefits are the reduction of the system's bandwidth bottlenecks, especially the network disk bandwidth, and the instantaneous feedback from the HPC application. It becomes apparent that analyzing the data of the application without going in situ will become impossible in the age of exascale computing [19]. However, in situ processing creates new challenges: A connection between the application and the in situ process must be enabled, maybe even the application may need to be changed. Furthermore, using live in situ analysis, the user cannot just queue an application and forget about it until it is done, analyzing it sometime afterwards. The scientist must be connected to the application while it's running to steer the analysis process. Data not been analyzed at this point are gone. Last but not least HPC clusters are not suited for a direct connection of users observing and steering from the outside. You cannot decide or know beforehand, which compute nodes your job will be executed on. Additionally, most of the time computation nodes are not accessed from outside the cluster. For most systems, a direct connection is not possible at all.

[1]Helmholtz-Zentrum Dresden – Rossendorf, Dresden, Germany
[2]Technische Universität Dresden, Dresden, Germany

In this paper we present ISAAC, an open-source, in situ visualizing and steering library which addresses the stated isssues. Not only does ISAAC enable arbitrary applications to create volume renderings of their scientific data, it also establishes an easy to use way of passing and receiving messages from and to the application. Using C++ metaprogramming techniques no deep copy of already existing data is needed. The visualizing algorithm can work on the original data of the application itself, but is still independent of domain-specific layouts, alignments or shifts of the underlying data.

Right now six different architectures are present in the world wide top 10 supercomputers, including the well-known X86, Sparc, or PowerPC CPU architectures and hardware accelerators like Nvidia Tesla and Intel Xeon Phi. The previously mentioned TaihuLight introduces a new architecture SW26010. To be able to interact with all these different architectures without loosing performance portability, ISAAC uses Alpaka, an open-source library for the zero overhead abstraction of highly parallel CPUs and hardware accelerators [36]. Alpaka abstracts from different parallel programming models like CUDA, OpenMP and std::thread to a single programming model, similar in structure to CUDA, enabling host CPUs to be abstracted as hardware accelerators. In the future support for OpenACC, Intel Thread Building Blocks or AMD HIP is also envisaged. With this approach, ISAAC can concentrate on a single programming model while still being able to run on every hardware of the top 10 systems. In this paper we will concentrate on one of these systems, Piz Daint, to test the performance of ISAAC in a real-world situation.

For this we have used ISAAC to visualize the open-source plasma simulation PIConGPU running on GPUs [5, 6]. For the evaluation 4096 Nvidia Kepler GPUs of the Piz Daint cluster are used. We show that ISAAC is capable of visualizing a highly parallel petascale application and scales for the upcoming exascale era.

## 1. Related Work

In situ processing, especially visualizing, is a well-investigated topic, not only presenting some theoretical thoughts, but also with a lot of ready to use libraries and tool sets.

Besides visualizing the application data, other data compressions are possible. Lakshminarasimhan et al. showed that a lossy compression of 85% of scientific data is possible with a guaranteed correlation of $> 0.99$ with the original data [14]. The libraries zfp and zplib have similar approaches for lossy and lossless compressing of floating point arrays and 2D and 3D fields to decrease the amount of data to save or transfer [15, 16]. Another method is to filter the data before saving or transferring it for classical post processing using only a representative sample of the whole dataset [31].

The literature furthermore distinguishes the distance of computation and visualization [2]. Tightly coupled in situ processing uses the very same hardware, whereby loosely-coupled processing uses dedicated but with a fast interconnect connected visualizing nodes. To differentiate these two, the last is also called in-transit processing [2]. The tight coupled approach has the benefit of reusing the simulation data without the need of copying [10, 29, 33]. Internal application data and functions can be used for a faster visualization as shown from Yu et al. [32]. Apart from that, application and visualization then compete for the same resources. Load balancing may be needed for the most efficient usage of the hardware [11]. Hybrid systems first filter and compress data before the loosely-coupled visualization [26].

Hardware accelerators in particular often only have limited memory compared to the host system which is already fully used by the HPC application. The analysis of application data

often requires creation of consecutive transformations of the raw data, often stored in a new temporary buffer. Beside stealing memory from the HPC application, this approach also does not allow CPUs to cache data before the next analysis step. Most of the cycles are used writing or reading data to and from memory. Moreland et al. describe a much faster approach defining a chain of so called *worklets* describing the transformations which are executed at every timestamp from the raw data to the final transformation without storing it in temporary buffers [24].

An intrinsic challenge of in situ is that users must to be connected to the HPC application to be able to steer it. Kageyama et al. use a different approach called Light-Field-Rendering, in which the application is visualized from dozens or even thousands of different directions and the videos saved automatically. Using modern compression algorithms like H264 the produced videos are much smaller than the raw application data and much easier to be saved to hard disk. Later the user can analyze the visualization choosing and interpolating the correct video based on the viewing angle [13].

Ahrens et al. implemented a similar solution called ParaView Cinema. Based on options set right before the run, like camera position and angles, iso surface thresholds, cutting planes, and similar visualization parameters, a database with millions of images is created live in situ. Although this database is multiple terabytes in size, it is still smaller than saving the raw scientific data directly. The post processing application can then query for specific images similar to the approach of Kageyama et al., but as no videos but an image database is created, the iso surface images can consist, for example, of normals and the hit values instead of colors, enabling the user to adjust coloring and lighting interactively [1].

ParaView [7] and VisIt [8] are two of the most used tools for scientific visualization. As they are open-source, other groups were able to extend them to enable in situ rendering [26]. The library Libsim implements the tightly coupled approach for VisIt [30], whereas two in-transit approaches exist for ParaView. As most scientific applications are able to write to the de facto standard for scientific data, HDF5, the library ICARUS implements an HDF5 driver which sends the data to a ParaView visualization server instead of being written to a hard disk [3]. The built-in library ParaView Coprocessing of ParaView uses a similar approach, but defines a C-like interface like Libsim which needs to be used from the HPC application [9].

Besides these general solutions, some attempts have been made to add in situ visualization to already existing application codes directly [17, 25]. The benefit is in the possibility to directly reuse the existing data structures. In particular applications running on hardware accelerators don't need to deep copy the data to the host interface of the generic in situ libraries [28]. Although being very fast these solution have the disadvantage of being very hard to port to another application. In fact one would just rewrite the visualization code.

## 2. System design

The goal of ISAAC is to combine the benefits of a generic library solution with the speed of custom-built in situ visualizations. For this ISAAC is implemented as a header only library meaning that the included C++-header does not only contain the interface but the whole library code and thus does not need to be linked against in the end. With this approach we can use the same types defined and used by the HPC application and the underlying hardware whereby no data conversion is needed at all.

In its current version ISAAC provides the following visualization features

- volume rendering,

- iso surface rendering,
- clipping planes,
- freely definable transfer functions,
- data interpolation,
- zero data copy capability with the ability to directly operate on the simulation data,
- freely configurable data transformation using functors and
- a server-client architecture allowing for simulation steering and remote visualization.

## 2.1. Design concept

The idea behind ISAAC is to work on the simulation data in-memory. This means that the simulation data structure is transformed to be suitable for visualization without the need for copying the data. This is important to reach optimum performance for current multi- and many-core compute hardware. On such systems, the fastest memory with the highest bandwidth and the lowest latency is usually small compared to the overall memory available in a node. Deep copies between levels in the memory hierarchy, for example from the accelerator memory to the main memory of a node, usually come with a degradation in performance.

With its zero copy philosophy, ISAAC operates directly on the simulation data at the highest bandwidth and lowest latency available while keeping memory consumption at a minimum. It makes extensive use of C++ template metaprogramming techniques in order to transform data by reindexing during compile time and abstracting parallel computations using the library Alpaka.

With this, ISAAC can define its rendering algorithm completely independent from the application. Instead of defining a C library interface like Libsim or ParaView Coprocessing, ISAAC defines a type independent interface which the application needs to feed with accessors to the real data which handle the alignment and pitch, but also can perform a reordering of the domain dimensions. The accessor itself can be removed at compile time (zero overhead abstraction).

Instead of deciding on which data transformations to apply at runtime using a large loop which includes all data sources and visualization features and selecting them based on user input, ISAAC unrolls this loop, compiling all possible combinations of features during the generation of the final executable. This is important to enable hardware-dependent optimization at compile time and for keeping branch divergence in the code to a minimum. Using the functor chain concept introduced later, this enables the user to choose arbitrary visualization features at runtime while providing for optimum code performance.

To be able to interact with arbitrary application code, ISAAC abstracts the HPC application describing a cuboid-shaped 3D volume, in which each compute node has its own disjoint, also cuboid-shaped local part of the global volume. Applications often use copies of other local volumes of other nodes called *ghost regions* or *guards*. These regions are exploited from ISAAC for interpolation over node borders, but are not part of the disjoint volume and thus not directly accessed in the ray casting. The data calculated in the application are abstracted as *sources*. A source assigns a scalar value or a vector for every regularly spaced position in the global volume, defining a field this way. Thereby every compute node only needs to know the assignment for its local sub volume.

On the local volume ISAAC performs a ray casting over all sources on every node. The local images of all nodes are then composited. Afterwards the application gets back the focus and ISAAC returns for all except for the first node on which the image is compressed and sent to a

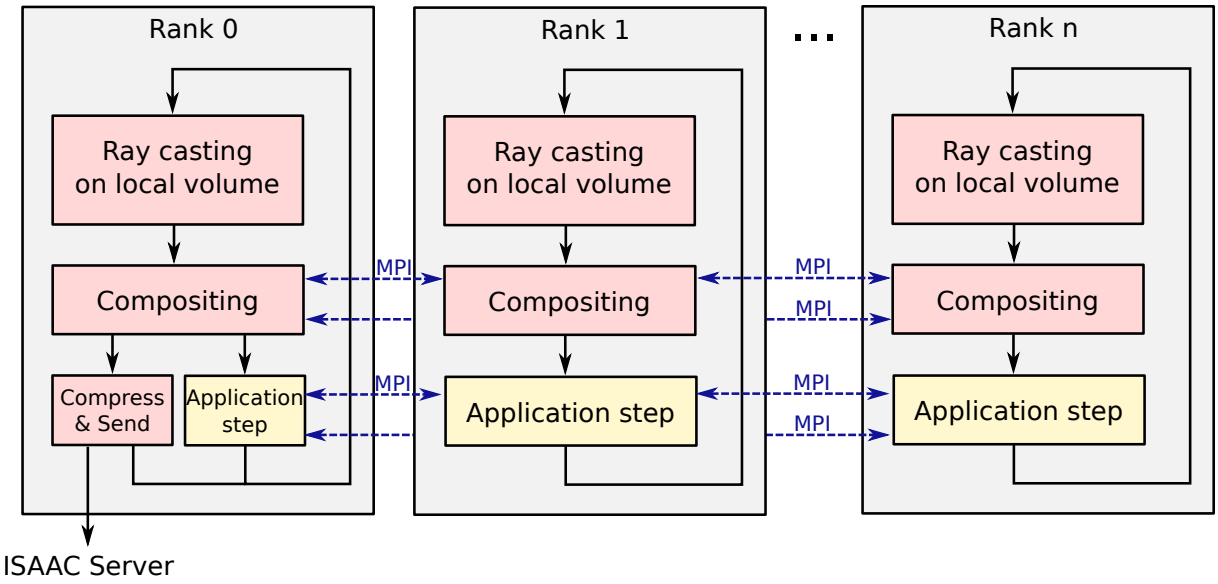



**Figure 1.** The full ISAAC concept. First a local ray casting is done. The resulting images are composited using the interconnect. The whole image is only on the first node in the end. Afterwards the application gets back the focus on every node, but on the first node the image is also compressed and sent to the ISAAC server in parallel

central streaming server in the background as seen in figure 1. The details will be described in the following sub sections.

## 2.2. Class–based interface

Listing 1 shows a local description of a source. An application defines such a class for every source. It consists of three static, constant attributes and three member functions. The attribute `f_dim` defines the dimension of the vector field called *feature dimension*. ISAAC supports vectors with up to 4 dimensions. A vector field with more than four dimensions could be implemented as two or more sources with smaller dimensions sorted in semantic groups. The data accessor `operator[]` is used to assign a value for an integer index position `nIndex` in the local volume of the compute node. In this example the accessor does not have to read from memory at all. Even an analytic description of a source is possible. This example source, for example, defines a homogeneous scalar field which is 42 everywhere.

ISAAC works on a 3D location domain, but it is still possible to define sources for 2D applications by simply ignoring the z component of the position. If possible application wise, past timesteps could be mapped to the z component using time as the third dimension.

Although the ray casting algorithm does only work on the local volume it might be handy to access the most outer region of a neighbor volume residing on another device or compute node for interpolation between the integer positions. ISAAC itself does not communicate these guards or ghost regions between the nodes, but can use it if the underlying application transparently provides those. The flag `has_guard` deactivates a border check if interpolation is activated to tell ISAAC that such a region exists outside the local volume.

Right before ISAAC starts the ray casting for each source the feedback function `update` is called which can also be used to forward frame–dependent information (like the time step) to the sources. This function shall be used to prepare the accessors to deliver the correct fields

```
struct TSource
{
  //Feature dimension of the source. 1 => scalar field
  static const size_t f_dim = 1;
  //Option, whether the algorithm may read data outside the devices local volume
  static const bool has_guard = true;
  //Non persistent data need to be copied before rendering
  static const bool persistent = true;

  //Name of the source
  static std::string getName()
  {
    return std::string("Test Source");
  }

  //Function to be called for every source right before the rendering starts
  void update(bool enabled, void* pointer) {}

  //Accessor for the field data of the source
  float_vec<f_dim> operator[] (const int_vec<3>& nIndex) const
  {
    float_vec<f_dim> result = { 42 };
    return result;
  }
};
```

**Listing 1.** Definition of a source in ISAAC

when the rendering begins. ISAAC is primary meant to visualize already existing fields which are called persistent. But an application can also create secondary fields just for the visualization at this point. As some applications have only one temporary buffer for such fields, but can have more than one source using it, it is also possible to tell ISAAC to make a copy of this field using the data accessor by setting the flag `persistent` to false, so that the update functions of the following sources are safely able to reuse the temporary buffer.

For every scan point of the ray casting, the volume rendering algorithm iterates over all sources, checks for the `has_guard` and `persistent` flags, uses the `[]operator` (subscript operator) to get a date from the source, applies a functor chain (see next sub section), classifies it by color and opacity, and finally uses these data to calculate a global color and opacity for the scan point to be used in the classical front to back ray cast algorithm. Figure 2 shows all renderings loops and the metaprogramming optimizations as a flow chart.

Iterating over all sources for every scan point for every pixel of the resulting image and checking for flags in every round would add an enormous looping and comparison overhead. To avoid this, all sources need to be defined at compile time and the iteration is done via C++ metaprogramming. The whole loop over all sources is evaluated and unrolled at compile time and does not produce overhead at runtime. Furthermore, the `static const` flags can be evaluated at compile time and the compiler can optimize the data access having detailed informations about the data accessors and their arithmetical connection. Also if different sources use the same or very close data in the background, the compiler can optimize the data access such that it is read only once although it may be abstracted differently sourcewise.

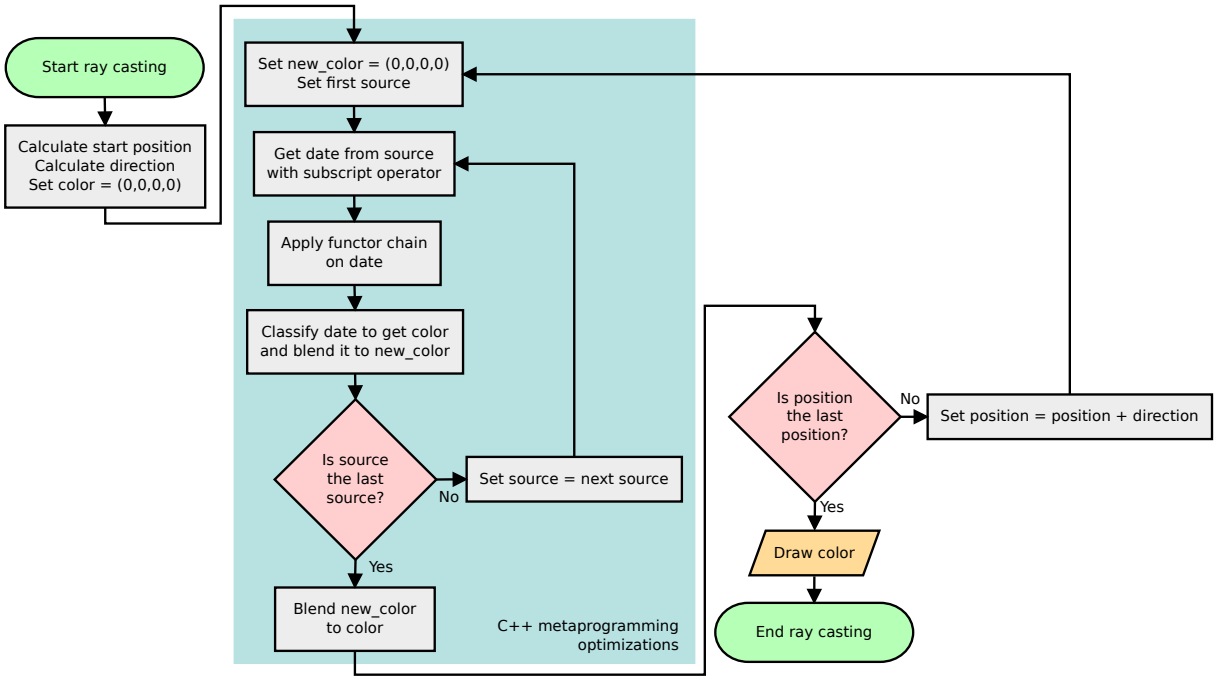



**Figure 2.** Simplified flow chart of the ray casting of ISAAC. The inner loop over all sources is evaluated at compile time

However, looping over all sources at compile time has the big disadvantage that every source is touched, even if it may not be of current interest at all. Even a classification assigning the opacity zero to every position in the local domain would still read the source data despite it being discarded in the end. To avoid this, $2^n$ different render functions are created at compile time for every combination of activated sources, where $n$ is the total number of sources. Although the rendering functions are generated at compile time, ISAAC chooses the right function at runtime based on interactive user settings. Deactivated sources are not touched at all, but we still have the benefit of looping over all activated sources at compile time.

Although this approach is quite fast, the number of rendering functions needed grows exponentially especially for big $n$. However, in the field $n$ should not be bigger than 10 for most applications resulting in a manageable amount of 1024 rendering functions. If an application defines more sources, at this time it is only possible to preselect the most interesting ones at compile time.

In the future ISAAC could be improved in such a way that $n$ sources may be defined, but only $m$ activated at the same time which can be calculated with the binomial coefficient $\binom{n}{m} = \frac{n!}{m!\cdot(n-m)!}$ instead of with $2^n$. For $n = 20$ and only three sources of interest at the same time ($m = 3$) this would resolve to $\binom{20}{3} = 1140$ instead of $2^{20} = 1048576$ render functions.

To use ISAAC, an application has to define the mentioned classes for every source, enqueue them in a C++ metaprogramming sequence and pass this list with some additional options as template parameters to the main ISAAC visualization class providing an optimized solution for the specific application without ISAAC needing to know anything about the application at all. An instance of this class can then be used for creating visualizations of the HPC application. Every time a frame shall be sent to the user, only the render method of the object needs to be called. ISAAC gets the focus from the application, creates the rendering and returns afterwards.

With this approach the developer has total control over ISAAC and no unpredictable behavior is happening in the background to interfere with the application, see also figure 1.

## 2.3. Functor chains

ISAAC works on the original application data to avoid any deep copy in the main memory. However, not all application data is suited for a direct visualization. The transfer function from volume date to color and opacity needs a value range to work on. Of course this range could be set as `static const` option like for guard and persistence, but sometimes even the application users cannot estimate what the maximum range may be, as it can depend on run time parameters or non-linear processes.

Furthermore, the classification works on scalar values, but often applications have vector fields, too, and it is not obvious before running the application and actually seeing the data which dimension or transformation from a vector value to a scalar value fits best.

As solution to this problem, ISAAC uses an approach similar to the worklets of Moreland et al. [24]. We define a small set of very basic functions, called *functors*, like multiplication or addition with a constant vector or scalar value, calculating the length, or summarizing all vector elements to one scalar value. These functors can be concatenated with the pipe sign `|` to more complex functions called *functor chains* successively applying functors to a date. At the moment functor chains are limited to work on one source only. Table **??** shows the predefined functors of ISAAC. Notice that the functor may change the domain of the input vector or scalar value to a vector of a different dimension or a scalar value.

**Table 1.** Five predefined functors in ISAAC for addition, multiplication, length, summarization and exponentiation. Functors may change the domain of the vector or scalar value and may have constant arguments

| Predefined functors | `add(v)` | `mul(v)` | `length` | `sum` | `pow(v)` |
|---|---|---|---|---|---|
| Functionality | Adds the vector `v` | Multiplies the vector `v` | Calculates the length | Summarizes all vector components | Exponentiates component-wise with `v` |
| Domain change | $\mathbb{R}^n \to \mathbb{R}^n$ | $\mathbb{R}^n \to \mathbb{R}^n$ | $\mathbb{R}^n \to \mathbb{R}$ | $\mathbb{R}^n \to \mathbb{R}$ | $\mathbb{R}^n \to \mathbb{R}^n$ |

A functor chain could be as an example `mul(2,3,4) | add(1) | length`. This exemplary function takes the raw date from the source, multiplies with vector $\begin{pmatrix}2\\3\\4\end{pmatrix}$, adds the vector $\begin{pmatrix}1\\1\\1\end{pmatrix}$ and calculates the length of the resulting vector returning a scalar value in the end. If no reduction to a scalar value is done explicitly, the first component of the resulting vector is used by default for the following classification in the ray casting.

If only selected vector components are of peculiar interest, the functor `sum` can be used in this way: `mul(0,1,0) | sum`. In this case, the first and third component get extinguished and afterwards all values are summarized to a scalar value representing the second component of the original vector.

Most of the needed transformations for sources can be described with this functors. If a functor is missing, it can easily be added to the code at compile time. Internally the functors are just classes which need to implement four versions of a method for all four possible feature dimensions, e.g. using templates. A needed dimension reduction (or enhancement) can be

expressed via the return type of the methods. These classes are enlisted in a C++ metaprogramming sequence and parsed at compile time. Without changing ISAAC itself, an application developer can add a functor; useful for a specific application domain.

The functor chains can be defined at run time and are executed right after reading the data from the source. The result of the functor chain execution is then used in the classification. However, checking for every functor of all sources in the ray casting loop would, again, be slow because of the lookup overhead – and useless as the functor chains do not change while rendering the image. Unfortunately it is not possible to create an own render function for every functor chain combination as every source can have a different functor chain. If we limit the maximum count of functors usable per source to $c$, this would create $f^c$ different combinations per source and $f^{cn}$ combinations of combinations for all $n$ sources. Even without involving the already mentioned $2^n$ render functions, if we choose the quite small values $f = 4$, $c = 3$, and $n = 3$ we would already have 262144 render functions. With $n = 5$ it would be over one billion.

Instead, we compile one function for every possible combination of functor chains for the total number of functors $f$ and user defined maximum functor chain length $c$. At run time, if the functor chain is changed, the corresponding function pointer of the precompiled function is chosen and set up for the source. Only $4 \cdot f^c$ functions need to be generated. The factor 4 comes from the four different feature dimensions possible in ISAAC. For the mentioned examples $n = 3$ and $n = 5$ this solves to 256 or 4096 needed versions, whereby $n > c$ is questionable anyway as this would mean that one functor is used twice which should not be needed. But even with this unrealistic number this can be handled by the compiler in finite time.

Using function pointers the compiler cannot do cross-function optimization at this point anymore. We tried to exploit as much compile time optimization as possible but had to draw a line at the point that is reached here with thousands of already fused functor chains being created. In use, further integration of the functor chains into the rendering function would have only a small or even no impact on the rendering time, but increase the compile time radically.

CUDA-specific optimization makes sense for the functor chains nevertheless. Alpaka abstracts the hardware accelerators in such a way that all available parallelization layers and performance gainers are part of the interface, but ignored for hardware not supporting them. ISAAC uses this to store the functor chain function pointers and the parameters of the functors in constant memory if the hardware supports this. Especially on Nvidia GPUs this improves the performance of the rendering noticeably.

## 2.4. Parallel Rendering

A typical cluster design can be seen in figure 3. The heart of a cluster consists of a high number of compute nodes doing the main computation tasks. Among themselves they shall be connected over a high performance network like InfiniBand. Sometimes they are also connected to the head node via such a network, but often only a slower connection, like Ethernet, is available. The task of cluster head nodes is to manage the jobs running on the cluster and sometimes compile the cluster applications before distributing them to the compute nodes. Often the cluster itself is just a small part of a bigger IT infrastructure of a company or research site. To access this network from the outside, login servers are used. However, the connection inside the global site network is only using Ethernet most of the time which is fast enough for most activities, but cannot handle the raw data produced in a modern cluster. ISAAC is designed to work well in those environments.

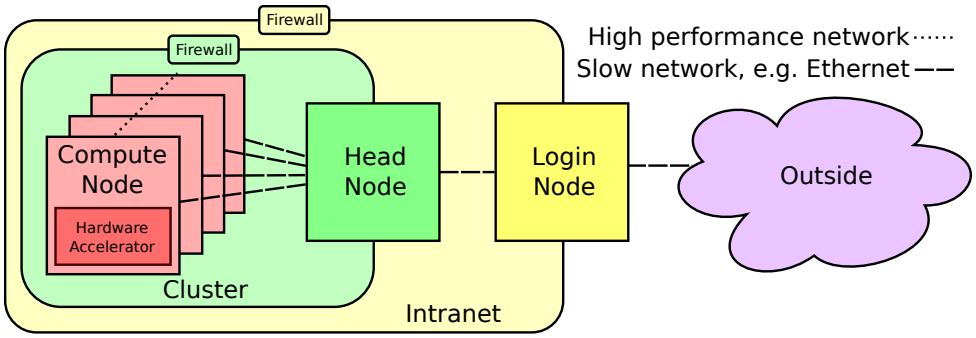



**Figure 3.** Simplified design of a typical cluster with multiple firewalls. The only connection between the compute nodes and the outside (e.g. the Internet) is over the cluster's head and login nodes. A high performance network can only be assumed between computation nodes. Green shows the cluster, yellow the intranet of the site, and purple the outside

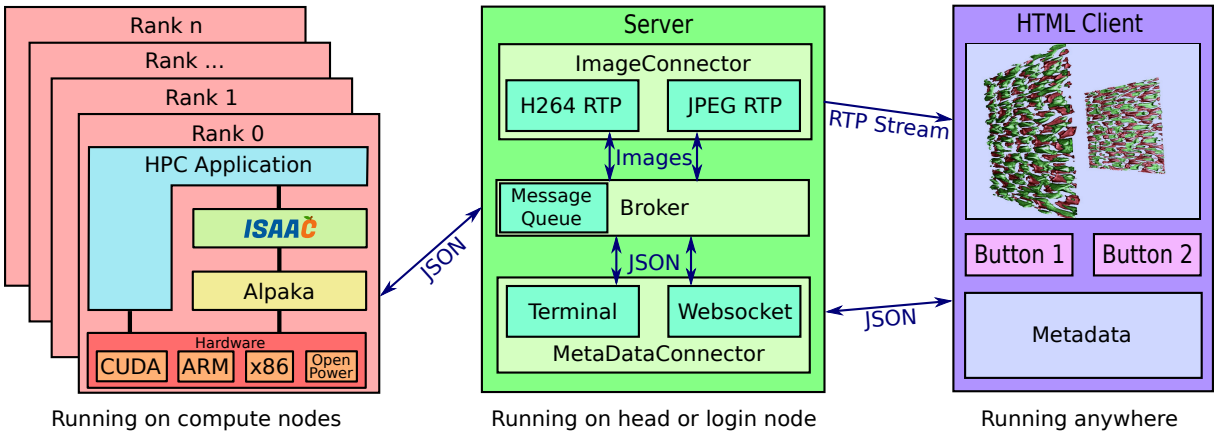



**Figure 4.** ISAAC consists of a library running integrated with the application on the compute nodes, a server running on one the head or login nodes, and an arbitrary amount of clients running anywhere. With this a connection between the compute nodes and the outside of the cluster can be established. Furthermore, the server can compress the data for slower networks

After the local image of every local domain is rendered, ISAAC composites one big image out of it using the compositing library IceT [27]. IceT composites the renderings over MPI using balancing techniques like binary swap [18] or 2–3 swap [34]. These algorithms ensure that the network load and the computing load are well distributed over the whole cluster.

IceT supports collection of the composited image on a node not involved in a rendering of a sub-domain at all. As the merged image needs to leave the cluster sooner or later, we considered collecting the final image on the head node. However, two problems occurred with this idea: First of all IceT works with MPI by default, whereas the head node is not part of the clusters MPI world and often of a different architecture. This could be solved with extending the open-source library IceT, but would mix MPI and Ethernet socket calls in IceT. Second, the user allocates time on the compute nodes but not the head node. It is not meant for being an important part of the HPC application itself.

Instead, the image is collected in the very first compute node (MPI rank 0). However, it is not good practice to let computers outside the cluster connect to the compute nodes ensured by internal firewalls. Considering this, ISAAC introduces a central server as seen in figure 4. Besides forwarding and managing connections of ISAAC sessions on the cluster and to an arbitrary amount of clients, it also has the possibility to compress the images and, e.g., to create streams.

At the moment the ISAAC server supports creating RTP streams with H264 and JPEG encoded videos and RTMP for streaming services like Twitch or Youtube, but defines a simple interface to add other or future streaming services, too.

## 2.5. Steering arbitrary HPC applications using open standards

The connection between the server and the ISAAC library uses TCP/IP. To be able to exchange library or server or to extend the behavior of both, every message is formatted in the easy to use and well documented *JavaScript Object Notation* (JSON) [4].

The server is designed in such a way that both the streaming service and the client connected to the server can be exchanged. We provide a ready to use HTML5 reference client which uses Websockets to establish a TCP/IP connection to the server. Again, JSON is used for easy extensibility. As JSON directly defines a JavaScript object, using the received objects in HTML5 and sending feedback back to applications is straightforward. The client can easily be adapted to the specific needs of an application.

All JSON commands sent and understood by the ISAAC library and the server are documented [20]. Additionally a `metadata` object is defined which can be used from the application as the root object for arbitrary information which will be added to an image after it is created but before it is sent to the clients. ISAAC does not care for the content of this object, but ensures that the resulting JSON object is still valid by removing objects with the same name as every node can add its own metadata which are merged by ISAAC before being sent to the server. Nevertheless, it is possible to gather information on all instances of the application binary running on the cluster using JSON arrays which are just concatenated by ISAAC.

A client can send arbitrary information or requests to a running application in the same way. The application gets ready-to-use JSON objects with the steering data of all connected clients. With this the application can be paused at a state of interest, to investigate the data, modify the functor chains or even change the application parameters themselves, e.g. decreasing the step width in a numerical integration. Additionally, unsuccessful application runs can be identified early and corrected or restarted with better settings. Especially as jobs often need a long waiting time on a cluster before they are started, an application restart inside the job can decrease waiting times and improve researchers productivity.

## 2.6. Exploiting heterogeneous systems

As the root rank and the server are often not connected with a high performance network, sending a raw bitmap takes more time than compressing the image and sending the smaller image. Because of this, a JPEG compression (with adjustable quality) is done on the root rank. The JPEG image itself is then included as string in the JSON message to the server using base64 encoding. This again adds an overhead of 33%, but ensures maintainability and extensibility as everything is still JSON. Furthermore adding a second binary channel would make the server more complex.

For applications using hardware accelerators we noticed that although the acceleration device itself is working to capacity, the host idles most of the time. For example, the GPU-accelerated plasma simulation PIConGPU used for the evaluation uses only one CPU per GPU for busy waiting for events of other ranks. As most compute nodes have more CPU cores than dedicated GPUs, this leads to a poor utilization of CPU resources.

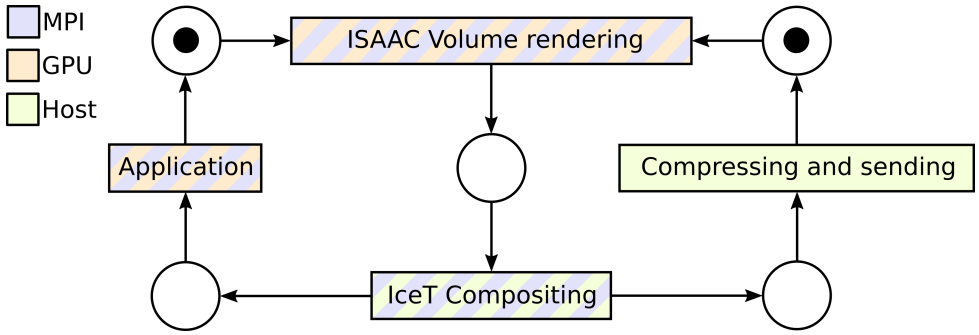



**Figure 5.** As ISAAC and the HPC application work on the same hardware accelerator and both use MPI, they cannot be executed in parallel. Only compressing and sending the final image is invariant of the application and MPI and can run in background while the application continues. Only if both concurrent tasks finish, the next frame can be rendered

To exploit this, ISAAC tries to do as much CPU tasks in parallel as possible. Unfortunately most of ISAAC's CPU tasks involve MPI. Even concurrent rendering can need changed scene settings propagated over MPI beforehand. The MPI standard does in fact define ways to use it on the same rank in different threads, but even if vendor support is given, the performance is worse than without threading. We assume that this is because of synchronization overhead between the concurrent MPI calls. Because of this, only the compressing and sending of the image is done in background on the root rank while the application continues.

The Petri net in figure 5 depicts this. The scene setting propagation, the volume rendering and the IceT compositing need to pause the application, but the compression and sending can be done in parallel to the application. Only if both concurrent tasks have finished, the transition to render the next frame is activated again. If an application step is much faster than compressing and sending the image, a solution is to call ISAAC only every $n$th frame which may result in a lower framerate but increases the utilization of the whole system.

## 3. Evaluation

To demonstrate that ISAAC works well together with existing applications running on modern HPC systems (especially those equipped with hardware accelerators), the plasma simulation PIConGPU running on Nvidia GPUs was visualized and steered with ISAAC. The simulation code defines a plugin interface which enables user defined code to extend the simulation without the need of changing the core code. Just three new classes needed to be written: two for the two possible field types of PIConGPU and one plugin class. Besides rendering a live visualization, some meta data, like the particle count, are also sent to the clients and the plugin listens to those enabling them to pause or even exit it.

### 3.1. Run Time on a Petascale System

We ran PIConGPU and ISAAC on the supercomputer Piz Daint of the Swiss National Supercomputing Centre on up to 4096 Nvidia Tesla K20X GPUs [22]. The cluster itself consists of 5272 GPUs, but because of availability and single user scheduling policies only 4096 were accessible for these tests. Each GPU has a single precision theoretical peak performance of 3.95 TFLOP/s. All used 4096 GPUs together reach a peak performance of $\sim 16.2$ PFLOPS/s. PIConGPU is memory bound, but still capable to use over 12% of the single precision peak

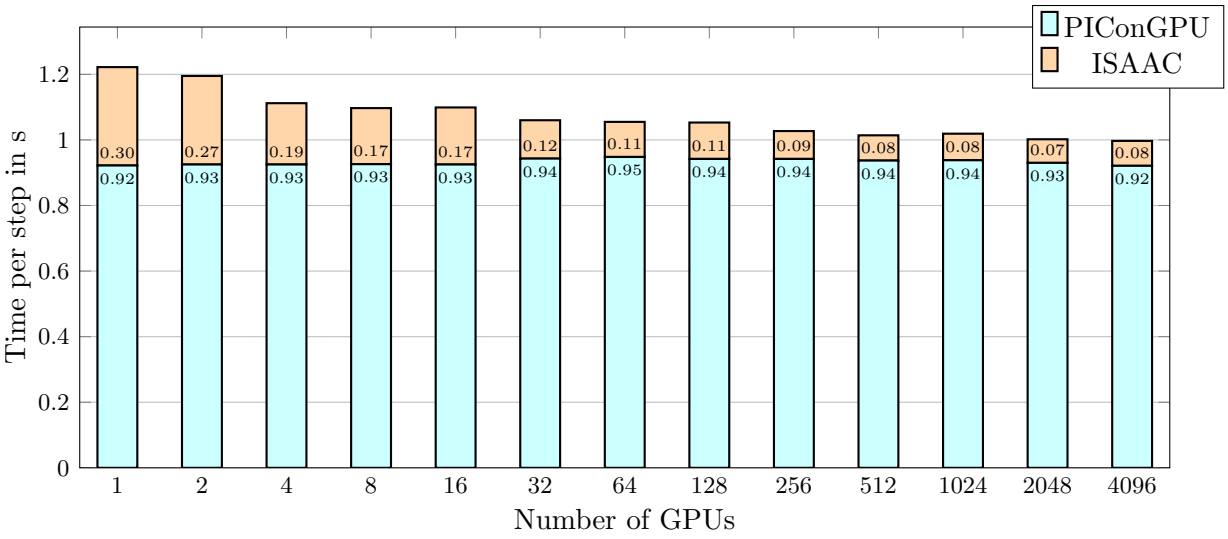



**Figure 6.** PIConGPU with ISAAC running on up to 4096 Nvidia Tesla K20X GPUs ($\sim$ 16.2 PFLOPS/s theoretical peak performance, single precision) on Piz Daint simulating a Kelvin-Helmholtz instability. Every GPU node simulates a volume of $128^3$. Adding more GPUs increases the global, simulated volume. PIConGPU shows perfect weak scaling [6], while ISAAC gets faster as the resolution of the image per GPU shrinks and thus the number of needed ray casting rays

performance on the Kepler architecture [35]. On the investigated sub set of Piz Daint this means $\sim$ 1.9 PFLOP/s are actually executed.

A Kelvin-Helmholtz instability is simulated. The simulation was parameterized with the Boris pusher, Esirkepov current solver, Yee field solver, trilinear-interpolation in field gathering, three spatial dimensions (3D3V), 128 cells in each dimension, electron and ion species with each sixteen particles per cell, and quadratic-spline interpolation (TSC) [12]. PIConGPU was compiled with nvcc 7.0 using the compiler flags `--use_fast_math --ftz=false -Xcompiler=-pthread -m64 -O3 -g`. With increasing the number of GPUs the local domain stays the same, but the global domain grows (weak scaling). The resolution of the ISAAC rendering was $1920 \times 1080$ (Full HD), interpolation was activated, two sources (electric field and current density) were rendered with complex functor chains including a square root operation, and 26 different view angles chosen such that well and poorly cacheable memory accesses happened.

Figure 6 shows the results of the runs. PIConGPU showed perfect weak scaling and always needed around 0.93 seconds per time step independent from the number of GPUs. ISAAC on the other hand decreased the run time depending on the count of GPUs. The reason is that with a growing number of GPUs, the local image size decreases. There is not a large difference between 512 and 4096 GPUs anymore: the rendering time itself reaches its minimum and the execution time is dominated by the render preparation and the IceT compositing which cannot be decreased easily anymore.

To show the effect of the 26 different camera angles, table **??** shows the average, best, and worst runtime for 1, 8, 64, 512, and 4096 GPUs. The worst value may be up to three times slower than the best for some cases. This is because of the previously mentioned inefficient caching as the ray casting algorithm may move perpendicular to the cache lines using only a few or even only one element of them.

Figure 7 shows the ISAAC run time compared to the total run time of the simulation in dependence of number of GPUs used. It shows that ISAAC is capable of visualizing a petascale

**Table 2.** Average, minimum, and maximum runtime for 1, 8, 64, 512, and 4096 GPUs. Due to inefficient caching for some view angles the worst value may be up to three times slower than the best

| Number of GPUs | 1 | 8 | 64 | 512 | 4096 |
|---|---|---|---|---|---|
| Average runtime | 300 ms | 171 ms | 107 ms | 77 ms | 76 ms |
| Minimum runtime | 212 ms | 88 ms | 61 ms | 51 ms | 53 ms |
| Maximum runtime | 369 ms | 229 ms | 131 ms | 98 ms | 102 ms |

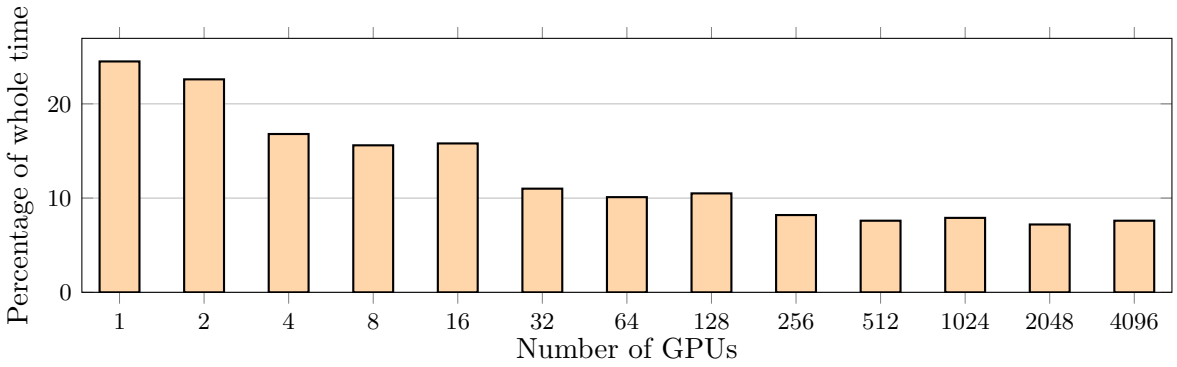



**Figure 7.** Percentage of ISAAC run time compared to the total run time of the simulation run. With decreasing number of GPUs, ISAAC settles down at $\sim 8\%$

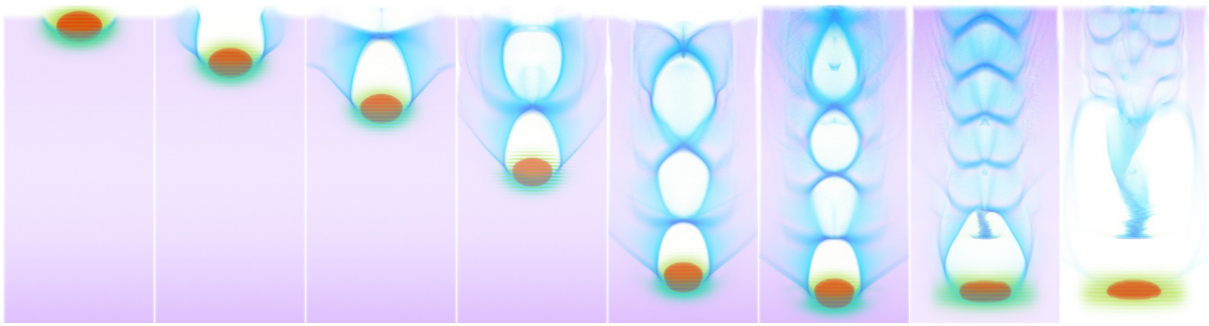


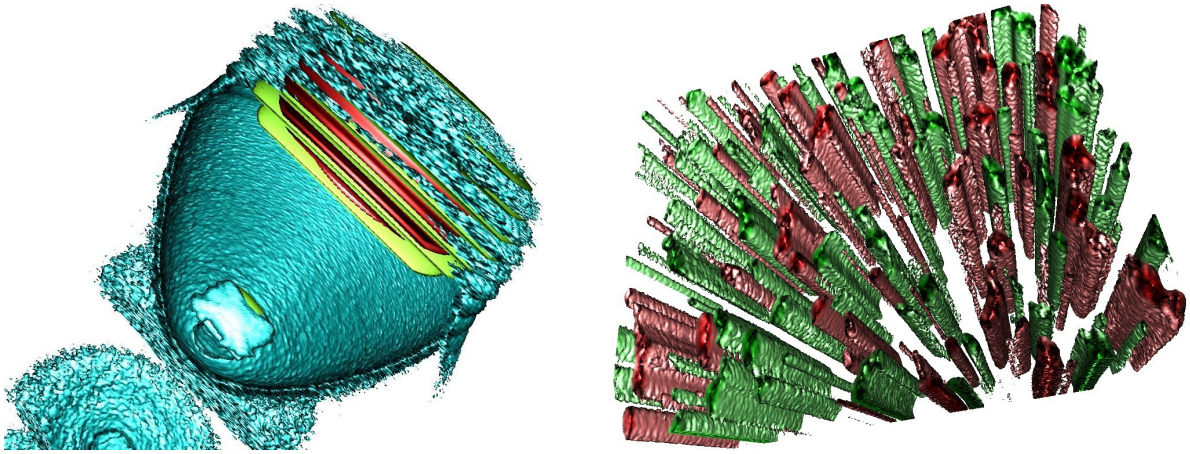
**Figure 8.** Time series of a laser wakefield accelerator

application running on modern hardware accelerators without interrupting it for a too long time: From 256 GPUs on, ISAAC needs less than 10% of the total run time and finally settles down at $\sim 8\%$.

The parameters of ISAAC were thereby set up quite conservatively. Enabling interpolation creates better visualization if iso surface rendering is used, but may be deactivated for a visualization as glowing gas for performance reasons. The chosen resolution shows that Full HD (resolution of $1920 \times 1080$) renderings are possible, but most of the time smaller resolutions will fit the needs anyway and will be chosen as other client elements also need to fit on the screen. Accordingly the run time can still be decreased easily if needed.

## 3.2. PIConGPU Renderings

This section will give some examples of PIConGPU renderings and why they are useful for the users.



(a) Laser wakefield Accelerator (b) Two stream instability

**Figure 9.** Laser wakefield accelerator and two stream instability with iso surface rendering

Figure 8 shows a time series of a so called laser wake field accelerator. A laser pulse (red and green) is ionizing and penetrating a gas. The electric-magnetic field of the laser pulse pushes the electrons (purple) off their ions (not shown). The moving electrons create a current which is shown in blue. As ions and electrons are seperated now, a plasma is created. Behind the laser pulse a bubble without electrons is created which is thereby positively charged as only inert ions remain. An electron can now be injected in this region as it can be seen in the both last images of the series, in which electrons from the back border enter the zone which is called self injection.

A physicist can see quite soon whether the simulation parameters show the expected phenomena, how long it takes until a self injection is happening, how the shape of the wake looks like, and how it changes over time.

Beside the visualization as glowing gas as seen in figure 8, ISAAC also supports visualization with iso surfaces as seen in figure 9. In (a) the bubble behind the laser pulse can be seen directly as 3D object and even the self injection and its shape is easily recognizable. The picture (b) shows a so called two stream instability, in which plasma flows in opposite directions creating long standing filaments and magnetic fields. While the simulation is running, the user can see the tubes appear, disappear and change their shape. The simulation can be paused at any time, the visualization parameters changed, a record of data started in the simulation or more (meta) data requested to be sent to the client. Going back in time would be an option, too, if the simulation supports this, e.g. with loading a slightly older checkpoint.

Using different colors and color maps may also help understanding the behavior of the simulation. Figure 10 shows a Kelvin-Helmholtz instability, where plasma flows in two streams side by side with different speeds, whereas vortexes are created. The green and red color identify both electrons, but with every color showing a different stream. The vortex effect can easily be spotted and observed over time. Furthermore, the electric field created by the moving electrons is shown in blue.

## Conclusion

In this paper we presented ISAAC, an open-source library for the in situ visualization and steering of applications running on HPC computers using multi-core CPUs or hardware accel-

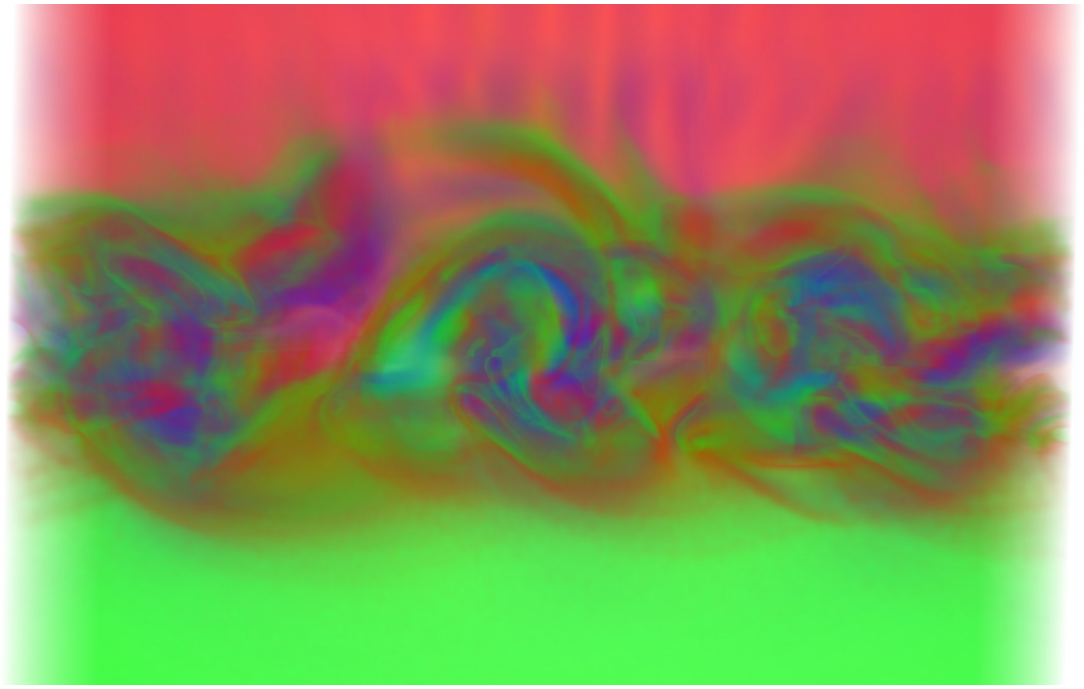


**Figure 10.** Kelvin-Helmholtz instability

erators like Nvidia GPUs. It is designed to work on the original application data structures and types without the need of deep copies or data format conversions. It provides volume rendering visualization as well as iso surface rendering and supports clipping planes, free transfer functions, and data interpolation for the fine tuned control of the output image.

Beside live rendering we showed that it is possible with ISAAC to ship arbitrary meta data packed in the open JSON format and to send feedback back to the application. A central server manages connections from supercomputer applications and clients and forwards messages between them as well as creates streams out of the raw pictures reducing the needed bandwidth. The central server enables exploration of these applications even from outside the site network.

To demonstrate the capabilities of ISAAC on a real world application, we added it to the world fastest particle-in-cell code PIConGPU [6] and ran it on the supercomputer Piz Daint. We showed that ISAAC is capable of visualization such a petascale simulation with only using 8% of the compute time for high quality Full HD images. We provided examples of how ISAAC's live visualizations help scientists on three different plasma effects simulateable with PIConGPU.

In the future, ISAAC must be tested on other large-scale high performance compute systems and hardware platforms. We expect the performance of ISAAC on a system to mainly depend on the scalability and performance of the Alpaka and IceT libraries used.

*ISAAC is open-source under the GNU Lesser General Public License (LGPL) Version 3+ [21]. This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 654220. Travel costs for the presentation on ISAAC at the ISC Workshop On In situ Visualization 2016 were supported by the Nvidia GPU Center of Excellence Dresden. Furthermore we want to thank the PIConGPU contributors for their help including and evaluating ISAAC with their simulation.*

## References

1. James Ahrens, Sébastien Jourdain, Patrick O'Leary, John Patchett, David H Rogers, and Mark Petersen. 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*, pages 424–434. IEEE Press, 2014.

2. Janine C Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, et al. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9. IEEE, 2012.

3. John Biddiscombe, Jerome Soumagne, Guillaume Oger, David Guibert, and Jean-Guillaume Piccinali. Parallel computational steering and analysis for hpc applications using a paraview interface and the hdf5 dsm virtual file driver. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 91–100. Eurographics Association, 2011.

4. Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/html/rfc7159`, 29 February 2016. [Online; accessed September 2, 2016].

5. Heiko Burau, Renée Widera, Wolfgang Honig, Guido Juckeland, Alexander Debus, Thomas Kluge, Ulrich Schramm, Tomas E Cowan, Roland Sauerbrey, and Michael Bussmann. Picongpu: A fully relativistic particle-in-cell code for a gpu cluster. *IEEE Transactions on Plasma Science*, 38(10):2831–2839, 2010.

6. Michael Bussmann, Heiko Burau, Thomas E Cowan, Alexander Debus, Alex Huebl, Guido Juckeland, Thomas Kluge, Wolfgang E Nagel, Richard Pausch, Felix Schmitt, et al. Radiative signatures of the relativistic kelvin-helmholtz instability. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 5. ACM, 2013.

7. Andy Cedilnik, Berk Geveci, Kenneth Moreland, James P Ahrens, and Jean M Favre. Remote large data visualization in the paraview framework. In *EGPGV*, pages 163–170, 2006.

8. Hank Childs. Visit: An end-user tool for visualizing and analyzing very large data. 2013.

9. Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Gevecik, Michel Rasquin, and Kenneth E Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

10. Oliver Fernandes, David S Blom, Steffen Frey, Alexander H Van Zuijlen, Hester Bijl, and Thomas Ertl. On in-situ visualization for strongly coupled partitioned fluid-structure interaction. In *Coupled Problems 2015: Proceedings of the 6th International Conference on*

*Computational Methods for Coupled Problems in Science and Engineering, Venice, Italy, 18-20 May 2015*. CIMNE, 2015.

11. Robert Hagan and Yong Cao. Multi-gpu load balancing for in-situ visualization. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, 2011.

12. Roger W Hockney and James W Eastwood. *Computer simulation using particles*. CRC Press, 1988.

13. Akira Kageyama and Tomoki Yamada. An approach to exascale visualization: Interactive viewing of in-situ visualization. *Computer Physics Communications*, 185(1):79–85, 2014.

14. Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *European Conference on Parallel Processing*, pages 366–379. Springer, 2011.

15. Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

16. Peter Lindstrom and Martin Isenburg. Fast and efficient compression of floating-point data. *IEEE transactions on visualization and computer graphics*, 12(5):1245–1250, 2006.

17. Kwan-Liu Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.

18. Kwan-Liu Ma, James S Painter, Charles D Hansen, and Michael F Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

19. Kwan-Liu Ma, Chaoli Wang, Hongfeng Yu, and Anna Tikhonova. In-situ processing and visualization for ultrascale simulations. In *Journal of Physics: Conference Series*, volume 78, page 012043. IOP Publishing, 2007.

20. Alexander Matthes. ISAAC JSON Commands. `http://computationalradiationphysics.github.io/isaac/doc/json/index.html`. [Online; accessed September 2, 2016].

21. Alexander Matthes. ISAAC Website. `http://computationalradiationphysics.github.io/isaac/`. [Online; accessed September 6, 2016].

22. Alexander Matthes, Axel Huebl, René Widera, Sebastian Grottel, Stefan Gumhold, and Michael Bussmann. PIConGPU and ISAAC software and results bundle for Supercomputing frontiers and innovations submission 2016. `https://doi.org/10.5281/zenodo.163494`, 2016.

23. Hans Werner Meuer, Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. June 2016 — TOP500 Supercomputer Sites. `https://www.top500.org/lists/2016/06/`, June 2016. [Online; accessed September 2, 2016].

24. Kenneth Moreland, Utkarsh Ayachit, Berk Geveci, and Kwan-Liu Ma. Dax toolkit: A proposed framework for data analysis and visualization at extreme scale. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 97–104. IEEE, 2011.

25. Liu Ning, Gao Guoxian, Zhang Yingping, and Zhu Dengming. The design and implement of ultra-scale data parallel in-situ visualization system. In *Audio Language and Image Processing (ICALIP), 2010 International Conference on*, pages 960–963. IEEE, 2010.

26. Marzia Rivi, Luigi Calori, Giuseppa Muscianisi, and Vladimir Slavnic. In-situ visualization: State-of-the-art and some use cases. *PRACE White Paper; PRACE: Brussels, Belgium*, 2012.

27. Sandia National Laboratories. Sandia National Laboratories: IceT Documentation. `http://icet.sandia.gov/documentation/index.html`, January 2011. [Online; accessed September 5, 2016].

28. John E Stone, Kirby L Vandivort, and Klaus Schulten. Gpu-accelerated molecular visualization on petascale supercomputing platforms. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, page 6. ACM, 2013.

29. Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R O'hallaron. From mesh generation to scientific visualization: An end-to-end approach to parallel supercomputing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 91. ACM, 2006.

30. Brad Whitlock, Jean M. Favre, and Jeremy S. Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. 2011.

31. Jonathan Woodring, J Ahrens, J Figg, Joanne Wendelberger, Salman Habib, and Katrin Heitmann. In-situ sampling of a large-scale particle simulation for interactive visualization and analysis. In *Computer Graphics Forum*, volume 30, pages 1151–1160. Wiley Online Library, 2011.

32. Hongfeng Yu, Tiankai Tu, Jacobo Bielak, Omar Ghattas, Julio López, Kwan-Liu Ma, David R O'hallaron, Leonardo Ramirez-Guzman, Nathan Stone, Ricardo Taborda-Rios, et al. Remote runtime steering of integrated terascale simulation and visualization. 2006.

33. Hongfeng Yu, Chaoli Wang, R Grout, J Chen, and K Ma. A study of in situ visualization for petascale combustion simulations. Technical report, Citeseer, 2009.

34. Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Massively parallel volume rendering using 2–3 swap image compositing. In *2008 SC-International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2008.

35. Erik Zenker, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E Nagel, and Michael Bussmann. Performance-portable many-core plasma simulations: Porting picongpu to openpower and beyond. *arXiv preprint arXiv:1606.02862*, 2016.

36. Erik Zenker, Benjamin Worpitz, René Widera, Axel Huebl, Guido Juckeland, Andreas Knüpfer, Wolfgang E Nagel, and Michael Bussmann. Alpaka-an abstraction library for parallel kernel acceleration. *arXiv preprint arXiv:1602.08477*, 2016.

DOI: 10.14529/jsfi160404

# Preparing for In Situ Processing on Upcoming Leading-edge Supercomputers

*James Kress*[1], *Randy Michael Churchill*[2], *Scott Klasky*[3], *Mark Kim*[3], *Hank Childs*[4], *David Pugmire*[3]



High performance computing applications are producing increasingly large amounts of data and placing enormous stress on current capabilities for traditional post-hoc visualization techniques. Because of the growing compute and I/O imbalance, data reductions, including in situ visualization, are required. These reduced data are used for analysis and visualization in a variety of different ways. Many of the visualization and analysis requirements are known a priori, but when they are not, scientists are dependent on the reduced data to accurately represent the simulation in post hoc analysis. The contributions of this paper is a description of the directions we are pursuing to assist a large scale fusion simulation code succeed on the next generation of supercomputers. These directions include the role of in situ processing for performing data reductions, as well as the tradeoffs between data size and data integrity within the context of complex operations in a typical scientific workflow.

*Keywords: Scientific Visualization, In Situ Methods, Data Staging Methods, Data Reductions, High Performance Computing.*

## Introduction

As leading-edge supercomputers get increasingly powerful, scientific simulations running on these machines are generating ever larger volumes of data. However, the increasing cost of data movement, in particular moving data to disk, is increasingly limiting the ability to process, analyze, and fully comprehend simulation results [1], hampering knowledge extraction. Specifically, while I/O bandwidths regularly increase with each new supercomputer, these increases are well below corresponding increases in computational ability and data generated. Further, this trend is predicted to persist for the foreseeable future.

Given this reality, many large-scale simulation codes are attempting to bypass the I/O bottleneck by using in situ visualization and analysis, i.e., processing simulation data when it is generated. A key question for in situ analysis is whether there is a priori knowledge of which visualizations and analyses to produce. If this a priori knowledge exists, then in situ processing is often superior to post hoc processing on leading-edge supercomputers, since it avoids disk usage. However, it is not guaranteed that the required visualizations and analyses are known a priori. That is, a domain scientist may need to explore the data in an interactive fashion, or unanticipated analysis may be required. With this research, we consider the model where in situ processing is used to reduce the data to a form small enough that it can be saved to disk and later read back for post hoc exploration.

Data reductions under this model could take on many different forms. A few examples of data reduction types could be compression (lossy or lossless), summary data (vector fields, min/max in a region, etc.), and reduced precision data formats. Importantly, data reductions should consider the types of analysis that will be done in the future, as to not introduce errors or artifacts that are not expected. This means that data reductions should be done within a set of known error bounds that is acceptable to the researchers doing the post hoc analysis.

[1]University of Oregon, Eugene, USA & Oak Ridge National Laboratory, Oak Ridge, USA; kressjm@ornl.gov
[2]Princeton Plasma Physics Laboratory, Princeton, USA; rchurchi@pppl.gov
[3]Oak Ridge National Laboratory, Oak Ridge, USA; {klasky, kimmb, pugmire}@ornl.gov
[4]University of Oregon, Eugene, USA & Lawrence Berkeley National Laboratory, Berkley, USA; hank@cs.uoregon.edu

With this work, we consider in situ data reduction in the context of a cutting-edge fusion simulation code, XGC1. We collaborate closely with XGC1 domain scientists and have been considering which reductions will be appropriate for this code as their ability to store data further and further decreases. We consider two distinct approaches, both of which we believe will be necessary for successfully maintaining meaningful analysis and visualization as XGC1 simulations are run on the next generation of supercomputers. In both cases, the techniques we consider keep in mind the balance between reduction and integrity. The contribution of the paper, then, is the description of the directions we pursue for XGC1 to succeed on the next generation of supercomputers, and our results to date in these directions. XGC1 contains two different types of data — particle data and mesh-based field data — and we consider techniques for both. For particle data, we consider sub-selection of particles and how to carry this out in a meaningful fashion. For mesh-based field data, we consider reduced precision and its effects.

In the remainder of this paper we discuss related work in Section 1, briefly discuss XGC1 in Section 2, present two example problems motivating in situ data reductions in Sections 3 & 4, and conclude with a discussion of areas of continuing and future research.

# 1. Related Work

Our work builds off of the momentum from the in situ movement, as well as past visualization work within XGC1. We present the related work for in situ data reduction in three sub-categories: (1) in situ visualization, (2) XGC1 visualization, and (3) HPC data compression.

## 1.1. In Situ Visualization

Visualization algorithms are particularly sensitive to I/O bandwidth [3, 4], causing the community to turn to in situ techniques to alleviate this growing problem. There has been significant work and successes with the in situ visualization paradigm. For instance, ParaView Catalyst coprocessing [7] and VisIt LibSim [33] are frameworks that are tightly-coupled to the simulation, *i.e*, the visualization runs at scale with the simulation. Visualization and analytics can be performed during the transport of the simulation data to the I/O layer as well. Three examples of this loosely coupled approach are Nessie [26], GLEAN [32], and ADIOS [18]. For a more thorough overview of the three loosely-coupled in situ visualization frameworks, we refer the reader to [23].

## 1.2. XGC1 Visualization

Early work on production visualization for XGC1 mainly focused on addressing the immediate data needs of scientists during the course of a simulation run. One example of this was an online dashboard that was developed for XGC1 simulation monitoring called eSimon [30]. This dashboard was launched in conjunction with each simulation run, and was responsible for performing common visualization and analysis tasks in XGC1. First, the dashboard was responsible for creating and updating plots of approximately 150 different variables every 30 seconds and plotting 65 different planes from the live simulation. At the conclusion of a run, the dashboard would automatically output movies of each of these plots of interest for quick review. In addition this dashboard catalogued simulation output, allowing users to search for and retrieve data of interest, without having to locate and search through simulation output files. Finally, this dashboard was available to scientists anywhere in the world through their internet browsers, making the data quickly and readily available.

More recent work has focused on expanding the visualization capabilities and opportunities for XGC1 through the utilization of in situ methods. For example, they utilized the features of ADIOS and EAVL [28], and demonstrated the effectiveness of loosely coupled in situ visualization for large scale simulation codes using a workflow consisting of ADIOS, data staging, and EAVL. In that work, they focused on the performance, scalability, and ease of use of visualization plugins that were used on the output of the XGC1 simulation code. One component of this study looked at optimizing the parallel rendering pipeline in situ, and gave insight into getting high performing renderings in continuing studies.

Following this effort, work shifted towards researching in transit visualization opportunities for XGC1 on the wide area network [27]. This research looked at data coupling and near-real-time analysis and visualization between two geographically separated sites using ADIOS and its ICEE [5] transport method. This capability is important when considering future use cases for visualization and analysis, when both simulation and experimental data need to be used and streamed together.

### 1.3. HPC Data Compression

Data compression strategies for HPC data are typically split into two categories, lossless and lossy compression. Typical lossless compression techniques include Huffman encoding [10], LZ77 [34], Gzip [8] and Fpzip [15]. Huffman encoding is an entropy-based compression method that identifies common occurrences in the data and assigns them unique codes. The LZ77 algorithm uses a sliding window to search for repeated sequences and then replaces them with a single copy that occurred previously in the data. Gzip is a general purpose compression technique for compressing any type of data, and makes use of both Huffman encoding and the LZ77 algorithm. Fpzip is a lossless compression method that is focused on floating point data. To improve the compression ratio, a variety of lossy compression methods have been developed over the years, including ISABELA [13] and ZFP [14]. ISABELA applies a pre-conditioner to the data, and then fits the data with B-spline interpolation to achieve very impressive compression rates for floating point data. ZFP is a fixed-rate compression scheme that encodes structured scientific data into $4 \times 4 \times 4$ blocks using a lifting scheme. Although designed for encoding scientific codes, ZFP was used for lossy disk storage compression for full-3D seismic waveform tomography, and reduced the strain-field disk storage by at least an order of magnitude [16].

Work in mesh decimation and simplification is a well researched topic, and Luebke [19] provides an overview of the various strategies including simplification and view-dependent methods. Also included are the approaches for each strategy, including sampling, decimation, vertex merging, topology driven, and error minimization techniques.

## 2. XGC1 Overview

XGC1 [2] is a 5D gyrokinetic ion-electron particle in cell (PIC) code used to study fusion of magnetically confined burning plasmas. XGC1 is used in particular to study turbulence in the outer region of the plasma called the *edge*. The simulation proceeds by computing the interactions of a very large number of particles (ions and electrons), and depositing them onto a finite element mesh at each time step. The mesh consists of a number of 2D planes positioned uniformly around the toroidal shape of the tokamak, as shown in Figure 1. At each time step, the particles, which interact within the toroidal space of the mesh, are statistically deposited as scalar fields onto the mesh. This deposition step provides a statistical view of simulation, and also helps optimize the simulation runtime.

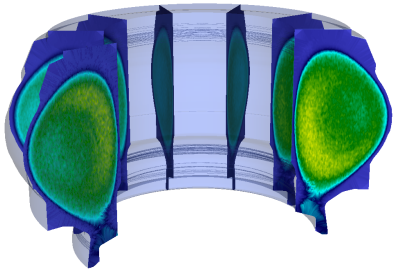


**Figure 1.** Example of a mesh in XGC1. Its planes are equally spaced around the central axis of the tokamak

## 3. Data Reduction in Staging

In situ methods allow access to all of a simulations output at each simulation time step. This is a powerful technique that is enabling new types of analysis and finer spatiotemporal resolutions than ever before. However, this newfound access to all of the simulation's data brings with it challenges in how to efficiently process that much data. With our workflow, we will show that in situ data summaries of a very large number of particles is possible using a data staging environment, enabling a new type of analysis for the simulation scientists.

This portion of our study used the particle data from XGC1. Particle data is the largest output dataset from this simulation. This data can be very large, generally ranging from over 900 GB to nearly 20 TB per time step. Furthermore, in order to get good bulk particle velocity vector fields as the particles move around the tokomak, we were required to access the particles at each time step of the simulation. Since this analysis routine is still being developed however, we did not run the simulation at full scale, instead we used smaller test scale run on 256 processors with 100,000 particles per core. Our final particle files were 4 GB. We do however relate our experience from this smaller run to how larger in situ runs will need to be handled in our conclusion.

### 3.1. Analysis Workflow

Our workflow consists of three primary elements: (1) the simulation code, (2) a data transfer system to move data from the simulation to the visualization nodes, and (3) an efficient parallel visualization library. This study was conducted on the Sith cluster at the Oak Ridge Leadership Computing Facility. Each of the 39 nodes in Sith contains four 2.3 GHz 8 core AMD Opteron processors and 64 GB of memory, configured with an 86 TB Lustre file system for scratch space. The components of the workflow that interact with XGC1 are described below, followed by a description of how the components of the workflow interact.

#### *3.1.1. ADIOS Data Staging*

The Adaptable I/O System (ADIOS) [17] is a componentization of the I/O layer accessible via a posix-style interface. The ADIOS API abstracts the operation away from implementation, allowing users to compose their applications independent of the underlying software and hardware. This capability, along with the functionality of DataSpaces [6], allows this same API to support read and write operations to and from the memory space of visualization staging nodes.

The loosely coupled paradigm in ADIOS and DataSpaces provides for a clean interface and separation from XGC1 that provides ease of use and fault tolerance. Also, this method allows the resource requirements for the visualization and staging tasks to be tailored for specific purposes.

For this study, we utilized two nodes on Sith to launch eight staging servers to handle the data connections from both the simulation and visualization nodes. Staging servers are used to store in memory the data coming from the simulation until the visualization routines call for it. This server configuration gave the best performance, and is easily scaled as more simulation or visualization nodes are added to the workflow.

#### *3.1.2. Visualization Library*

We designed our visualization routines as flexible, light weight plugins. Our plugins are based on the Extreme-scale Analysis and Visualization Library (EAVL) [20]. EAVL was developed to address three primary objectives: update the traditional data model to handle modern simulation codes, investigate the efficiency of I/O, computation and memory on an updated data and execution model, and explore visualization algorithms on next-generation architectures. EAVL defines more flexible mesh, and data structures which more efficiently supports the traditional types of data supported by de-facto standards like VTK, but also allows for efficient representations of non-traditional data. Examples of non-traditional data includes graphs, mixed data types (e.g. molecular data, high order field data, unique mesh topologies (e.g. unstructured adaptive mesh refinement and quad-trees)). EAVL uses a functor concept in the execution model to allow users to write operations that are applied to data. The functor concept in EAVL has been abstracted to allow for execution on either the CPU or GPU, and the execution model manages the movement of data to the particular execution hardware.

For this study, we ran our visualization routines on four nodes, with four processes per node. The number of visualization nodes can readily be scaled up as the data size increases so as not in slow down the simulation as each time step is processed.

#### *3.1.3. Workflow Composition*

Our visualization and analysis workflow combines three separate elements: the XGC1 simulation, the staging servers, and the visualization libraries. The workflow is launched as three separate binaries, with resources for each partitioned as follows: (1) 256 XGC1 processors; (2) 8 Staging servers; and (3) 16 visualization processors. Data flows through the workflow for every XGC1 time step as follows:

- When a new time step is ready from XGC1, it is immediately sent to our staging servers and subsequently consumed by the visualization routine. The visualization routine does a parallel read of the restart file, with each process taking $1/nProcs$ of the data. As consecutive time steps become available they are consumed.
- Next, the visualization routine statically maps particle ID's based on the visualization rank.
- Once each rank has the correct particles, a vector is computed between time step $n$ and $n+1$. These vectors are then averaged onto an unstructured grid. Since the XGC1 mesh is very finely resolved, we use a coarser version of the mesh in this depositing step.

The resulting vector field is then written to disk for further analysis. It is important to note that this step is a major data reduction. As shown in Table 1, by performing the vector computation in situ, we are reducing the amount of data written to disk by between 95 and 476,190 times. This reduction factor is based on the output particle size of our Test-Scale Run, and the output size of a Large-Scale Run.

### 3.2. Results

Using the workflow setup described in the previous section, we are able to successfully create effective bulk plasma particle velocity vector fields for XGC1. We are in the early stages of the analysis

**Table 1.** Showing the output particle size in relation to the actual data size being written to disk at each simulation time step by performing the particle vector analysis in situ

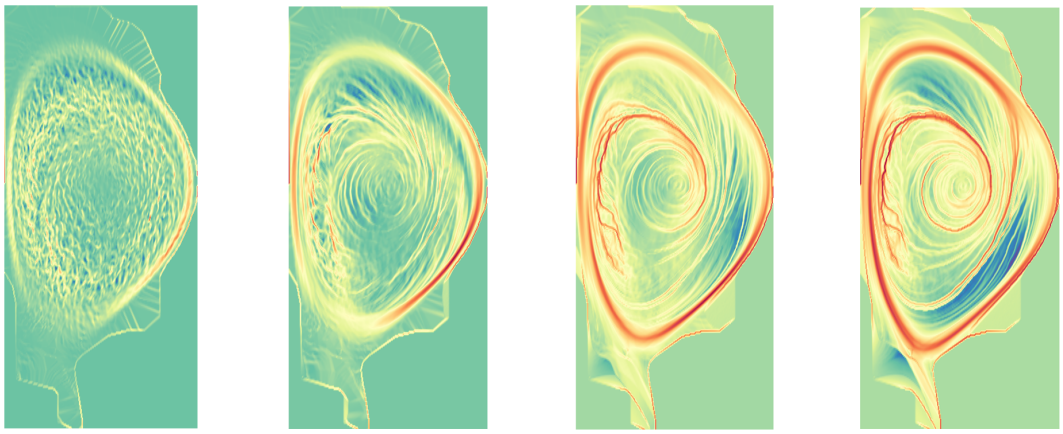
| XGC1 Sim Run Size | Particle Size | Mesh Size | Reduction |
|---|---|---|---|
| Test-Scale Run | 4 GB | 42 MB | 95x |
| Large-Scale Run | 20 TB | 42 MB | 476,190x |

of these new fields, so our main highlighted results are that this analysis is now feasible, as well as the timings that we have gathered from our different analysis runs. A few of the analysis routines that we plan to explore with the physicists using this data include streamlines, FTLE (Figure 2), Poincaré plots [29], and more.

From the runs we performed to create the effective bulk plasma particle velocity vector fields, we measured the amount of time used by the visualization routine, XGC1, and staging. Timings are an important metric for this simulation, as additional time impacts to the simulation are carefully scrutinized for production runs. From the timings that we gathered, we found that the impacts of the analysis routine on the XGC1 simulation are minimal. The only impact to the simulation occurs during the particle write at each time step. We are currently writing at each time step in order to achieve maximum temporal resolution with our vectors, but this number can be tuned with future runs as we perform further analysis.

We performed timings of the particle write step of XGC1 using the two currently possible mechanisms for accessing the particles for vector analysis: staging and file based. As shown in Table 2, we found that by staging the particles, we were able to get a 3.2x time reduction with staging versus writing the particles to disk. This meant that the simulation was impacted less, as it was paused for a shorter period of time with staging versus the file based method. It is important to note that the time to write the file to disk will exponentially increase as the run is scaled up to production sizes, while we believe that staging times will rise at a much lower rate.

Another important data point that we found is that we are able to easily complete the analysis portion of the pipeline during the time that XGC1 uses to compute a new time step, using only on average 35%



**(a).** Advection time 0.10 **(b).** Advection time 1.0 **(c).** Advection time 5.0 **(d).** Advection time 10.0

**Figure 2.** A slice of an FTLE plot using the effective bulk plasma particle velocity vector field. The shorter advection times demonstrate smaller scale features (plots a and b), while longer advection times bring larger scale features to light (plots c and d). The red values correspond to areas where the flow tends to separate, and blue is where the flow stays together

**Table 2.** Runtime of a single XGC time step using our test-scale run size when either sending data to staging or writing it to a file. The time variance is due to staging being much faster than the file based method, impacting the simulation less

| **XGC1 Simulation Time step Time (sec)** | | **XGC1 Simulation Particle Write Time (sec)** | | |
|---|---|---|---|---|
| *Staging* | *File* | *Staging* | *File* | *Reduction* |
| 96.6 | 102.8 | 1.8 | 5.8 | 3.2x |

of that time. This is good news as we progress towards using production runs, as we should be able to scale the visualization nodes in order to keep up with each simulation time step, and even add other analysis tasks to the workflow to fill in idle analysis periods.

## 4. Data Reduction Through Reduced Precision

In this example, we motivate and explore the implications of using reduced representations of data for analysis, and the impacts on preservation of information, and the associated errors.

The errors associated with data reduction techniques, where $f$ is the original data, and $\tilde{f}$ is the reduced data, are typically defined as follows:

$$E = |f(x,t) - \tilde{f}(x,t)| \tag{1}$$

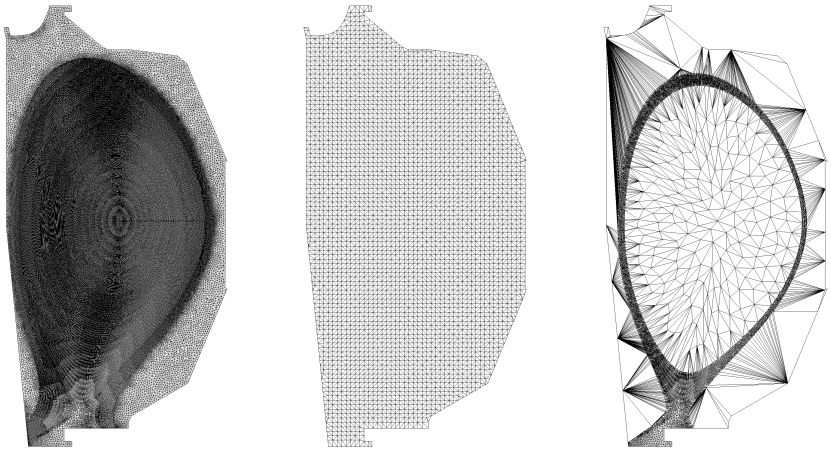
However, scientists will typically require derived quantities or operations, $G(f)$, on their data, and these derived quantities are not always known a priori. And so the errors that scientists care about becomes:

$$E = |G(f(x,t)) - G(\tilde{f}(x,t))| \tag{2}$$

These post-simulation analyses span a large space of possibilities. The simplest cases involve analysis and visualization of additional simulation variables, additional slice planes, subsets of data, or isovalues for contouring, as examples. More complicated examples involve transformations of the data, for example, changes of coordinate systems, different mappings, or analysis in different spaces such as the Fourier Transform. Derived variables can involve simple mathematical operations such as sum, difference, product, or division, or more complicated operations such as gradient, curl, divergence, and norms. More complex operations can include things such as feature detection and tracking, or particle advection for streamlines, pathlines, Poincaré plots, and Lagrangian Coherent Structures.

While there are a number of ways to reduce the size of scientific data, in this motivating example we focus on two lossy data reduction methods. The first is the floating point precision of the variable data, and the second is the spatial resolution of the underlying mesh. We also consider a combination of these two methods. In order to understand these methods in practice, we have applied these methods to primary variables from a simulation, derived variable calculations, feature detection, and more complex analysis operations.

Our work with reduced precision arose from our collaborations with the SIRIUS [11] project. The SIRIUS project is researching methods for the management and layout of large scientific data across the memory hierarchy of an HPC system. This layout might include breaking the data up into separate, dependent pieces. For example, 3 digits of precision in fast memory, and the rest of the precision in lower levels of the memory hierarchy. If needed, for particular operations, the extra precision can be combined with the lower precision representation.



**(a).** Full mesh.

**(b).** Uniform decimation of the full mesh.

**(c).** Adaptive decimation of the full mesh, preserving detail along the edge of the plasma.

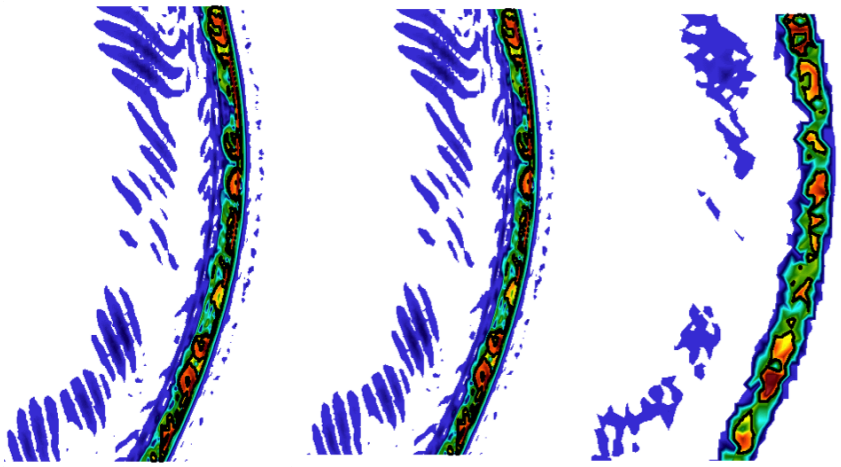
**Figure 3.** Different types of decimation for the simulation mesh in XGC1

Scientific simulations are typically done using double precision values, which are represented with 64 bits. While this level of precision is required for solving the equations in a simulation, it is typically not required for basic analysis and visualization operations. For example, when mapping the values of a variable through a color map, the resolution of the color map is often quite small compared to the range of the floating point values. The resulting pixels are then a quantized representation of the actual values. Because of this, very similar, or, in some cases, identical images can be produced from greatly reduced data. However, as stated previously, one under-explored area is the implications of these reduced precision data when more complex operations are applied to the data.

The spatial resolutions of the meshes for scientific simulations are determined by the convergence requirements of the underlying mathematics. This resolution may be appropriate for downstream processing of analysis and visualization operations, or it may be overkill. In this paper we explore two different types of mesh decimation: uniform and adaptive (see Figure 3). Adaptive decimation reduces the resolution of the mesh in a manner consistent with the underlying science. For example, in Figure 3(c), which is from an XGC1 fusion simulation, the plasma profiles near the edge region have sharp gradients, and so the resolution is higher in the edge region of the mesh to capture the fine-scale physics in this region. On the other hand, using uniform decimation, the mesh is reduced without this underlying knowledge. To adaptively decimate the mesh, we use a scheme based on quadric decimation [9]. For the error metric that drives the decimation, we use the proximity of mesh points to the edge region of the tokamak.

### 4.1. Results

To explore these ideas, we have been working with the XGC1 fusion simulation code. In this work, we are focusing on the field variables on the unstructured mesh in XGC1. The *primary* variables we are examining include the scalar potential ($\phi$), and the magnetic field ($B$). We are interested in examining *derived* variables computed using mathematical expressions on these *primary* variables, as well as particle tracing through the magnetic field.



**(a).** Full data with 48 features

**(b).** Reduced precision data with same 48 features

**(c).** Reduced precision and mesh decimation with 13 features

**Figure 4.** Feature detection of reduced representations of a primary variable

Figure 4 shows an example of feature detection on the scalar potential ($\phi$) in XGC1. In Figure 4(a), an edge detection algorithm has detected a set of 48 features from the full data set. In Figure 4(b), the same set of 48 features are identified using a 3 digit precision (5$X$ data reduction) version of $\phi$.

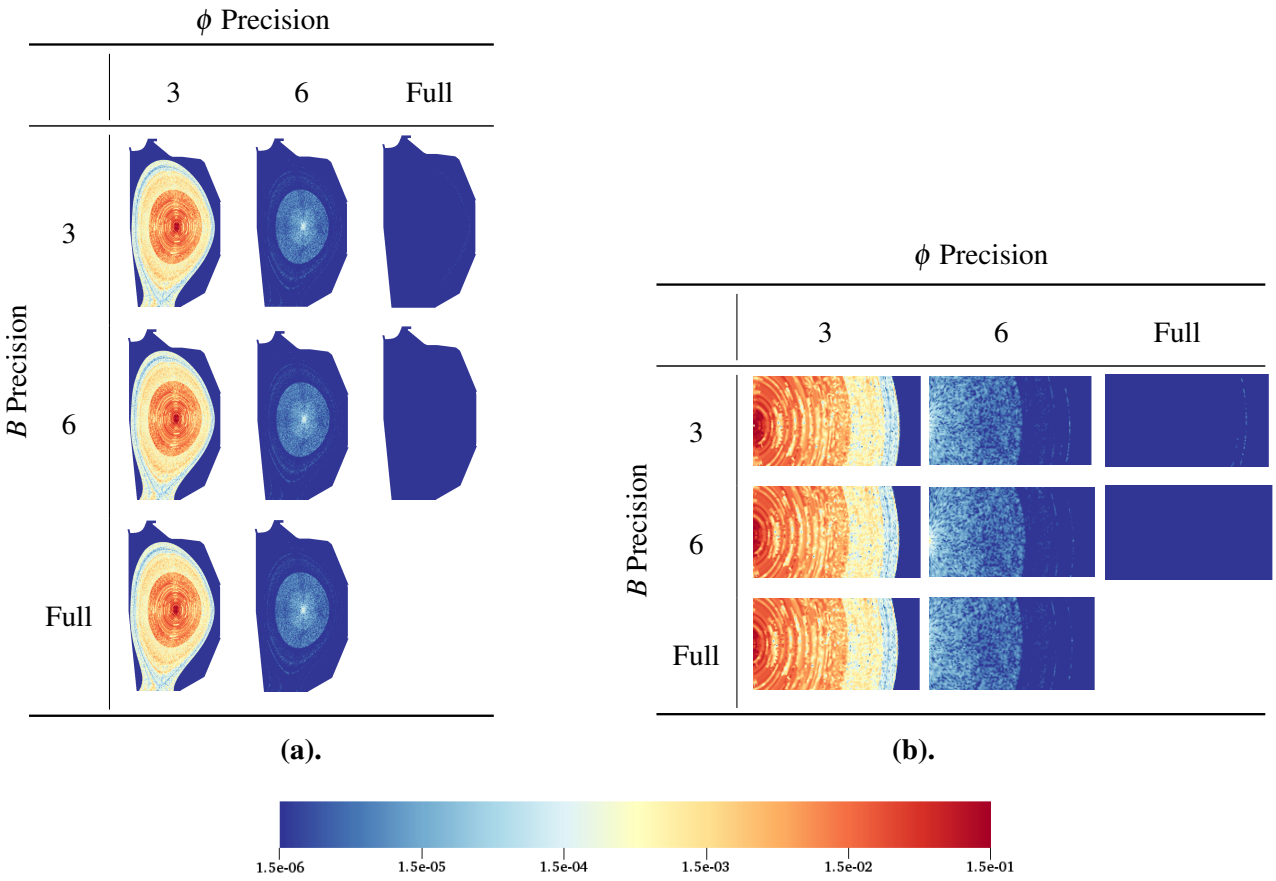



**Figure 5.** Relative error for fluid velocity using differing amounts of precision. Full cross section is shown in (a), and a zoomed in section in shown in (b)

In Figure 4(c), 3 digits of precision, and a $20X$ adaptively reduced mesh results in 13 features. The 3 digits of precision ($5X$ reduction) and $20X$ reduced mesh results in a total data reduction of $100X$. The question of whether or not the reduced data in Figure 4(c) provides an adequate representation of the underlying physics is something that we are in the process of evaluating with our collaborators in the XGC1 application team.

As an example of a derived variable, periodically scientists want to examine the perpendicular velocity of the plasma driven by the electric field ($V_F$), which is defined as follows:

$$V_F = \frac{\nabla\phi \times B}{|B|} \tag{3}$$

The relative error in the derived quantity of varying levels of precision in the primary variables is shown in Figure 5. Note that in this particular case, the error is clearly dependent on the precision of the variable $\phi$. The precision of $B$ has a much smaller impact on the errors in the derived variable. This leads to important insights about the relationship between errors and data reductions on variables, and can result in more efficient use of very limited data resources in an HPC system.

Relying solely on preserving scientific features in the primary variables can be problematic, as illustrated in Figure 6. In this example, by looking only at the reduced precision and decimated meshes for primary variables, $\phi$ and $B$, the main features appear to be preserved. However, when computing derived variables, in this case the fluid velocity, $V_F$, the features are *not* preserved using a uniform decimation scheme. This highlights the imperative to understand the implications of reduced data representations

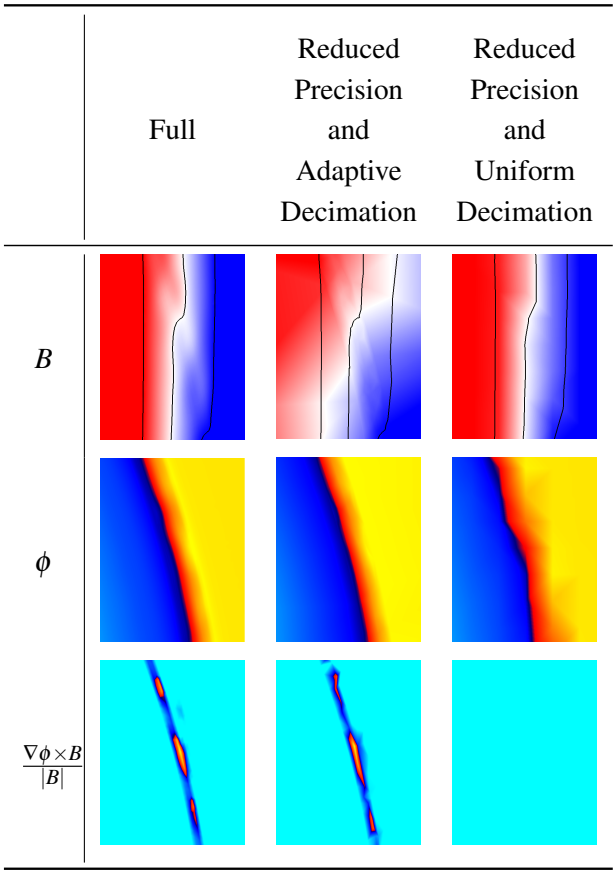



**Figure 6.** An example where features are preserved in the primary variables across different types of data reduction, but are lost when the derived variable is computed

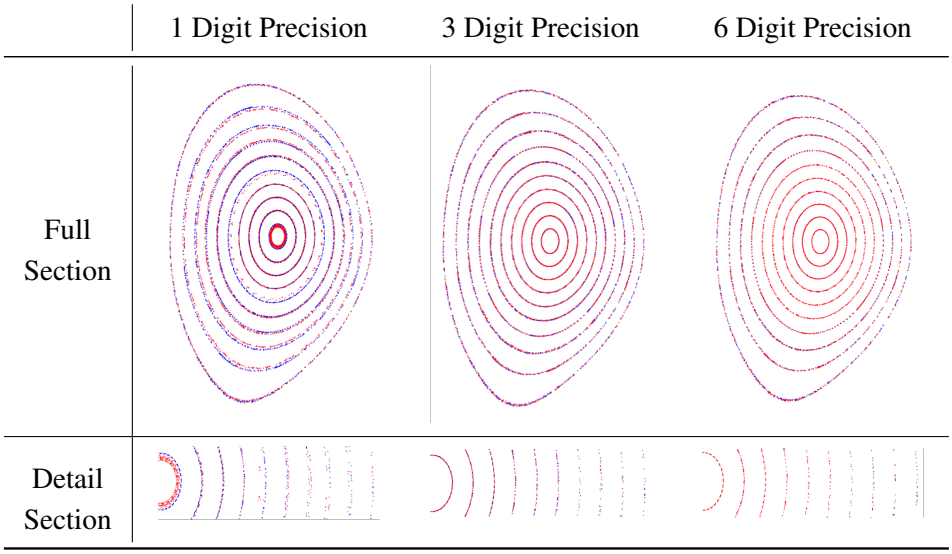



**Figure 7.** Poincaré plots using differing levels of precision in the magnetic field. The original data are shown in blue, and the reduced data are shown in red

on down stream analysis and visualization operations, and the unanticipated implications of reduced on data.

The magnetic field, $B$, is a fundamental driving mechanism in a fusion device. There are a number of ways to analyze and visualize the magnetic field in a fusion simulation, but one of the most common is by using particle advection based techniques. In particle advection methods, a set of massless particles are traced through the mesh by the vector field. As the magnetic field in a tokamak is cyclical, one common analysis technique is the Poincaré, or puncture plot. Each particle is traced through the vector field, and each time it intersects a plane perpendicular to the field, the intersection point is marked. Since each particle travels along a magnetic surface, after the particle has circulated enough, a 2D representation of the magnetic surface can be seen.

The images in Figure 7 show Poincaré plots of several reduced versions of the magnetic field. The full data are shown in blue, and the reduced data are shown in red. The first row shows the entire Poincaré plot, and the second row shows a zoomed in section. As shown in the far left column, a single digit of precision produces results with significant errors, particularly near the center of the plasma. However, using only three digits of precision produces results that are very good visual approximations of the full data.

## Conclusions and Future Work

The growing disparity between compute and I/O on HPC systems will require simulation codes to radically alter how results are output and analyzed, and how scientific information is obtained. This transformation will require simulations to compute and output analysis and visualizations that are greatly reduced in size and complexity. These reduced outputs include a wide variety of data, including results that are more easily analyzed like images, movies, plots and graphs, as well as outputs that require post-hoc processing to understand, such as data summaries, and mesh-based data. At times, additional downstream processing is required in order to fully understand output results.

In situ and in transit processing methods will play a critical role for both categories of outputs mentioned above. For situations where a priori knowledge is available, these outputs should be computed in situ and output to disk. For situations where a priori knowledge is not available, and post-hoc analysis,

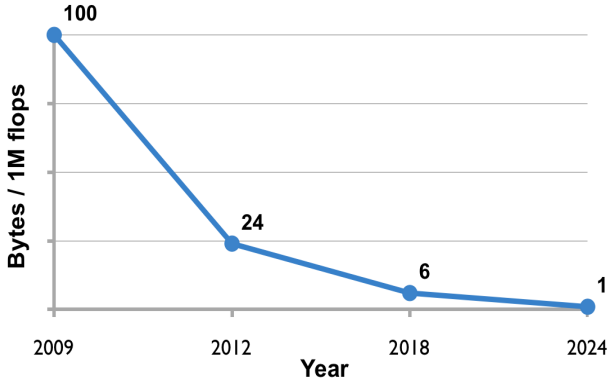



**Figure 8.** Current and expected I/O in bytes per 1 Million FLOPS. Data derived from forecasts on past, current and future HPC systems [12, 25, 31]

visualization, and processing are required, in situ and in transit methods will play a crucial role in producing integrity preserving reduced data that can be analyzed later. Because all the data are available, in situ methods play a crucial role in performing data reductions with specified error bounds. These error bounds should be valid for the simulation variables which are saved, as well as for post-hoc operations on data that are anticipated on the scientific workflows associated with the simulation.

While the growing disparity between compute and I/O is a reality, there is still significant I/O capability in current and future HPC systems, and efforts should be taken to ensure that a maximum amount of information is saved. The challenge is determining the proper set of data to be saved. As shown above in Figure 8, the expected trend for I/O is one byte per one million floating points operations on an exascale computer. The challenge then becomes to make sure that each byte written accurately represents the information produced from the one million floating point operations. In situ methods will play a critical role in determining the proper byte to be output. In transit methods, where data staging is used, will also play an important role in providing asynchronous computational capability. Additionally, because data staging nodes use a separate set of resources, analysis and visualization algorithms that require communication can operate on reduced data much more efficiently, and with a limited impact on the simulation.

Because of the breadth of issues involved, solutions to these problems will require collaborative research between a number of disciplines, including applied mathematics, analysis, visualization and middleware.

From a visualization perspective, a better understanding is required of the errors bars associated with operations on data, and how these errors propagate through visualization and analysis workflows. The data models for visualization tools needs to be expressive enough to represent data in new and unique forms. For example, native operations on data streams with compressed, or reduced precision data, efficient operations for variables that are on different meshes.

To address the widening of the memory hierarchy on HPC systems, projects like SIRIUS, in conjunction with middleware systems like ADIOS, are working to optimize the placement of data. SIRIUS aims to place the most valuable data in memory locations that are easily accessible, and data that are less important are pushed down the memory hierarchy, and eventually to long term storage systems, such as tape. Collaborations with this type of system would involve reduced, approximated representations in faster memory and the ability to pull up increasingly more accurate representations of the data as needed. Analysis and visualization processes should work seamlessly in these types of environments to operate on, and meet the error bars required by scientists. As such, developing the appropriate interfaces between these different layers is an important research direction.

We have demonstrated the advantage of these techniques through our work with the particle and mesh data in the XGC1 fusion code. We have focused on techniques for reducing the data in ways that preserve information for the scientists. For the particles we have shown how in transit processing can be used to compute the bulk velocity field, which is a reduced representation of the particle data. For mesh variables we have shown that care must be taken to reduce data in ways that preserve scientific information. We have focused on eddy's and features in the potential and fluid velocity field, and magnetic field analysis using streamlines and Poincaré methods.

In Section 3 we have shown how data staging can be used to perform data reductions on particle data in XGC1. Data staging proves useful in two distinct ways. First, the non-trivial data reduction task can be performed asynchronously, and not interfere with the simulation. Second, the communication required to derive the bulk velocity field can be done much more efficiently on a smaller set of nodes. We have developed these techniques using small runs of XGC1 on a moderately sized cluster. These small runs allowed us to operate on all of the simulation particles. A full production run of XGC1 on an HPC system would produce significantly more particles than could processed in a staging environment. For a full production run a subset of particles would be required that is statistically equivalent to the entire set of particles. In particular, given a total of $N$ simulation particles, we need a query that will return a set of $M$ (where $M << N$) particles uniformly distributed within the spatial extents of the simulation. Because the particles move at each time step, the representative set of particles must be periodically recomputed. As a result we will need methods for determining the quality of the particle subset, and deciding when it is required to recompute. We are currently collaborating with the ADIOS project to develop such particle queries in order to facilitate production runs on HPC systems. We are also working to determine the appropriate interface between the visualization tools and the middleware for these types of operations.

In Section 4 we have shown how reduced representations of XGC1 data can be used in visualization, and the unforseen issues that arise on derived quantities. Resolution of these issues will require a significantly better understanding of data reductions, and the propagation of errors. To formalize these ideas, we must understand the mathematical implications of these reductions, and be able to provide error bounds on the use of these reduced data under a variety of operations. These include simple operations like sum, difference, product and division, as well more complex operations like cross product, gradient, and curl. With a better understanding of the relationship between errors on input data, and operations in complex workflows, scientists can accurately specify the requirements for their analyses, and be confident in the results derived using reduced data.

From an analysis and visualization perspective, we need data models that provide the flexibility to operate on reduced data in a zero-copy paradigm. Tradeoffs exist between converting reduced data streams to floating point data and performing on-the-fly conversions as data are used. We are currently exploring how the data model in VTK-m [21] [22] [24] can be used to address these issues and explore the tradeoffs of reduced data size and efficiency of computations, particularly as it relates to computations on accelerators.

We have shown the clear benefits to using an adaptive decimation technique for XGC1. We are actively exploring techniques for other codes, as well as more generalized methods. When performing operations with variables on different meshes, collaborative research with applied mathematics is needed on how best to calculate the derived quantities in a way that minimizes the error. Solutions might be to interpolate from one mesh to another, or to derive a new mesh that meets the error requirements.

Finally, all of this work takes place in context of a middleware system that manages and coordinate the movement of data within the HPC system. We are actively collaborating with the ADIOS and SIRIUS

projects that provide methods to manage data across the the memory hierarchy, as well as the data staging capabilities in order to provide asynchronous processing. We are also collaborating with these projects to explore the proper interfaces between analysis and visualization components, and the middleware system components.

## Acknowledgements

*This work was supported by the Scalable Data Management, Analysis, and Visualization (SDAV) which is funded by the DOE Office of Science through the Office of Advanced Scientific Computing Research (Contract No. 12-015215).*



## References

1. Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, J Ahrens, EW Bethel, H Childs, et al. Scientific discovery at the exascale. *Report from the DOE ASCR 2011 Workshop on Exascale Data Management*, 2011.

2. CS Chang, S Ku, PH Diamond, Z Lin, S Parker, TS Hahm, and N Samatova. Compressed ion temperature gradient turbulence in diverted tokamak edgea). *Physics of Plasmas (1994-present)*, 16(5):056108, 2009.

3. Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel. Extreme scaling of production visualization software on diverse architectures. *IEEE Comput. Graph. Appl.*, 30(3):22–31, May 2010.

4. Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel. Visualization at extreme scale concurrency. In E. Wes Bethel, Hank Childs, and Charles Hansen, editors, *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, Boca Raton, FL, 2012.

5. Jong Y Choi, Kesheng Wu, Jacky C Wu, Alex Sim, Qing G Liu, Matthew Wolf, C Chang, and Scott Klasky. Icee: Wide-area in transit data processing framework for near real-time scientific applications. In *4th SC Workshop on Petascale (Big) Data Analytics: Challenges and Opportunities in conjunction with SC13*, 2013.

6. Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing*, 15(2):163–181, 2012.

7. Nathan Fabian, Kenneth Moreland, David Thompson, Andrew Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

8. Gzip compression. `http://www.gzip.org`.

9. Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In *Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99)*, VISUALIZATION '99, pages –, Washington, DC, USA, 1999. IEEE Computer Society.

10. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

11. Scott Klasky, Hasan Abbasi, Mark Ainsworth, Qing Liu, Jay Lofstead, CArlos Maltzahn, Manish Parashar, and Feyi Wang. Sirius: Science-driven data management for multi-tiered storage. *Proceedings of the XXVII IUPAP Conference on Computational Physics*, December 2015.

12. Kalyan Kumaran. Introduction to Mira. `https://www.alcf.anl.gov/files/bgq-perfengr.pdf`. Visited June 20, 2016.

13. Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par'11, pages 366–379, Berlin, Heidelberg, 2011. Springer-Verlag.

14. P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, Dec 2014.

15. P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, Sept 2006.

16. Peter Lindstrom, Po Chen, and En-Jui Lee. Reducing disk storage of full-3d seismic waveform tomography (f3dt) through lossy online compression. *Computers & Geosciences*, 93:45 – 54, 2016.

17. Qing Liu, Jeremy Logan, Yuan Tian, Hasan Abbasi, Norbert Podhorszki, Jong Youl Choi, Scott Klasky, Roselyne Tchoua, Jay Lofstead, Ron Oldfield, Manish Parashar, Nagiza Samatova, Karsten Schwan, Arie Shoshani, Matthew Wolf, Kesheng Wu, and Weikuan Yu. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

18. Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08, pages 15–24, New York, NY, USA, 2008. ACM.

19. David P. Luebke. A developer's survey of polygonal simplification algorithms. *IEEE Comput. Graph. Appl.*, 21(3):24–35, May 2001.

20. Jeremy S Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30. The Eurographics Association, 2012.

21. Jeremy S Meredith, Sean Ahern, Dave Pugmire, and Robert Sisneros. EAVL: the extreme-scale analysis and visualization library. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 21–30. The Eurographics Association, 2012.

22. Jeremy S. Meredith, Robert Sisneros, David Pugmire, and Sean Ahern. A distributed data-parallel framework for analysis and visualization algorithm development. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 11–19, New York, NY, USA, 2012. ACM.

23. Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, et al. Examples of in transit visualization. In *Proceedings of the 2nd international workshop on Petascale data analytics: challenges and opportunities*, pages 1–6. ACM, 2011.

24. Kenneth Moreland, Christopher Sewell, William Usher, Lita Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, Matthew Larsen, Chun-Ming Chen, Robert Maynard, and Berk Geveci. VTK-m: Accelerating the Visualization Toolkit for Massively Threaded Architectures. *IEEE Computer Graphics and Applications (CG&A)*, 36(3):48–58, May/June 2016.

25. Lucy Nowell. Science at extreme scale: Architectural challenges and opportunities, 2014. `http://www.mcs.anl.gov/~hereld/doecgf2014/slides/ScienceAtExtremeScale_DOECGF_Nowell_140424v2.pdf`.

26. R. A. Oldfield, P. Widener, A. B. Maccabe, L. Ward, and T. Kordenbrock. Efficient data-movement for lightweight i/o. In *2006 IEEE International Conference on Cluster Computing*, pages 1–9, Sept 2006.

27. David Pugmire, James Kress, Hank Childs, Matthew Wolf, Greg Eisenhauer, Randy Churchill, Tahsin Kurc, Jong Choi, Scott Klasky, Kesheng Wu, Alex Sim, and Junmin Gu. Visualization and analysis for near-real-time decision making in distributed workflows. In *High Performance Data Analysis and Visualization (HPDAV) 2016 held in conjuction with IPDPS 2016*, May 2016.

28. David Pugmire, James Kress, Jeremy Meredith, Norbert Podhorszki, Jong Choi, and Scott Klasky. Towards scalable visualization plugins for data staging workflows. In *Big Data Analytics: Challenges and Opportunities (BDAC-14) Workshop at Supercomputing Conference*, November 2014.

29. Allen Sanderson, Guoning Chen, Xavier Tricoche, David Pugmire, Scott Kruger, and Joshua Breslau. Analysis of recurrent patterns in toroidal magnetic fields. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1431–1440, 2010.

30. Roselyne Tchoua, Jong Choi, Scott Klasky, Qing Liu, Jeremy Logan, Kenneth Moreland, Jingqing Mu, Manish Parashar, Norbert Podhorszki, David Pugmire, et al. Adios visualization schema: A first step towards improving interdisciplinary collaboration in high performance computing. In *eScience (eScience), 2013 IEEE 9th International Conference on*, pages 27–34. IEEE, 2013.

31. Patrick Thibodeau. Coming by 2023, an exascale supercomputer in the U.S. `http://goo.gl/qKTa8q`. Visited June 20, 2016.

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

33. Brad Whitlock, Jean M Favre, and Jeremy S Meredith. Parallel in situ coupling of simulation with a fully featured visualization system. In *Proceedings of the 11th Eurographics conference on Parallel Graphics and Visualization*, pages 101–109. Eurographics Association, 2011.

34. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

DOI: 10.14529/jsfi160405

# Analysis of CPU Usage Data Properties and their possible impact on Performance Monitoring

***Konstantin S. Stefanov***[1]***, Alexey A. Gradskov***[1]



CPU usage data (CPU user, system, iowait etc. load levels) are often the basic data used for performance monitoring. The source of these data is the operating system. In this paper we analyze some properties of CPU usage data provided by the Linux kernel. We examine the kernel source code and provide test results to find which level of accuracy and precision one may expect when using CPU load level data.

*Keywords: performance monitoring, sensor properties, sampling rate, CPU usage, CPU load level.*

## Introduction

Today supercomputers show very impressive performance for their peak values and with some benchmarks like LINPACK [1]. Other benchmarks like HPCG [2] as well as real-world applications produce much worse results, often not reaching even 10% of peak performance.

Performance monitoring is one of the methods used to evaluate applications while they are running and determine the obstacles to higher sustained performance of applications. The idea of performance monitoring is to collect metrics which describe the state of the application being run. These data are collected for all compute nodes running the given application. Some performance monitoring approaches [3] try to analyze metrics obtained from components which are shared for the whole system and correlate those data to specific applications.

The source of metrics data, which we call sensors, may be hardware like performance counters in modern CPU, or software, like various data provided by the operating system such as CPU usage data, load average, memory usage etc. Some sensors may be somewhere on the border between software and hardware like InfiniBand interface counters, which are maintained by InfiniBand card firmware.

There are questions about the properties of such sensors and their suitability for performance monitoring in different modes. Of course all those sensors are widely used for a long time for performance monitoring and give useful data which lead to useful results in application analysis. But as performance monitoring systems evolve we may encounter some limitations of such sensors which may lead to their unsuitability for new approaches or new modes of usage. For example SuperMon [4] can achieve up to 6000 Hz sampling rate while reading Linux kernel data from `/proc` [5] filesystem. But do we need such sampling rate and are the results obtained at such a high rate reliable? Will such high rate affect the precision of the data?

This question is not widely discussed for every type of data used for performance monitoring. When performance counters were introduced in processors, hardware performance counters were analyzed [6–9] from the point of accuracy, predictability, reproducibility and so on. Paper [10] compares two modes of using performance counters and compares the results obtained in these modes. A discussion about Load Average data in Linux kernel aroused on mailing lists [11]. This discussion resulted in some patches on kernel source code to make Load Average results more accurate, but it is not clear if today kernel load average data are accurate enough, and we found no such analysis for sensors other than performance counters and load average.

[1]M.V. Lomonosov Moscow State University, Moscow, Russia

CPU usage data are obtained from Linux kernel and are widely used for performance monitoring. In this paper we try to analyze Linux kernel source code and make some testing to evaluate CPU usage data properties which are vital for analyzing application behavior.

The paper is organized as follows. Section 1 describes how the CPU usage data in their conventional form are obtained from the kernel. Section 2 gives results of analysis of Linux kernel source code in parts which relate to CPU usage data. Section 3 provides results of testing supporting the results which were given in Section 2. The last section contains the conclusion.

## 1. How CPU usage data are obtained

The Linux kernel gives CPU usage data in `/proc/stat` file. For every active CPU in the system the kernel gives the amount of time, measured in 1/100 ths of a second, that the system spent in different modes of execution [5] since boot. These different modes are: user mode (running user processes), user mode with low priority (nice), system mode (running kernel), idle, iowait (idle while waiting for IO request to complete), irq (processing interrupts), softirq (processing software interrupts) and a few other modes related to virtualization. To obtain CPU usage in the form we are accustomed to with `top` or other utilities (we call it CPU load level hereafter), one should take the difference in one mode values between two successive measurements and divide it by the sum of such differences for all modes. If $T_m^i$ is the time spent in $m$-th mode at time moment $i$ (these values are given in `/proc/stat`), than the CPU load level $L_m$ for the mode $m$ is

$$L_m = \frac{T_m^i - T_m^{i-1}}{\sum\limits_m (T_m^i - T_m^{i-1})}$$

One can also try to get CPU load level by dividing $T_m^i - T_m^{i-1}$ by the time difference between $i$ and $i-1$ time moments, but as obtaining precise time is quite an expensive operation involving system call, this method is usually not used.

One consequence of this calculation method is that CPU usage values are discrete in nature. The number of different values they can take depends on sampling interval. The more time passes between successive samples, the more levels CPU usage value can take. `top` utility gives us CPU usage percent with precision of 1 decimal place (1000 possible different values in range from 0 to 100%). The data supplied by the kernel which are used for calculations are measured in 1/100 ths of a second; to have real precision of 1/10 th of percent for CPU load level value we should take the measurements more rarely than once in 10 seconds. With more frequent sampling the precision of CPU load level will be less than 1 decimal place.

## 2. Source code analysis

We examined the Linux kernel source code to find how these values (time spent in different modes) are calculated.

These per-CPU values (and per-process values for time spent in user and system mode, too) are updated during timer interrupt processing. Timer interrupt frequency is a parameter set during kernel compilation (it is named `HZ`). The most common values for this parameter are 1000 and 250 (timer interrupt is raised 1000 or 250 times per second, respectively). When executing the timer interrupt handler (generally it is executed on every CPU with some exceptions described later), the kernel finds the mode in which the given CPU was before switching to interrupt

handler and which process was running. The whole tick is accounted to the mode and to the process which was active before the CPU received timer interrupt. The modes which were active in period between timer interrupts are not accounted in any way.

Internally, CPU usage times are calculated in kernel as numbers of 1/`HZ` second time intervals. When the results are returned to the user, they are scaled to 1/100 ths of a second. Such rescaling may introduce some rounding errors but they are not expected to be high.

When `NO_HZ` [12] kernel compilation parameter is active (true by default for modern SMP kernels), timer interrupts are not delivered to idle CPUs. When CPU usage values are requested, idle and iowait times are calculated at the moment of the request by finding the time since the given CPU became idle. When the CPU comes out of idle state, the values for idle and iowait are calculated and saved to accounting data structures.

The outcome of this calculation method is that the CPU usage times for user, system, and nice modes are updated only on timer interrupts and that some frequent changes from running to idle or between other modes may pass unnoticed by the accounting code.

# 3. Experiments

## 3.1. Measuring interval between CPU usage data changes

Our first experiment was designed to prove that CPU usage times for user, system, and nice modes are updated only on some periodic events. To check this we performed a test which was constantly requesting CPU usage time and calculated time which passed between successive CPU usage changes. The results are presented in fig. 1. Time interval in milliseconds between

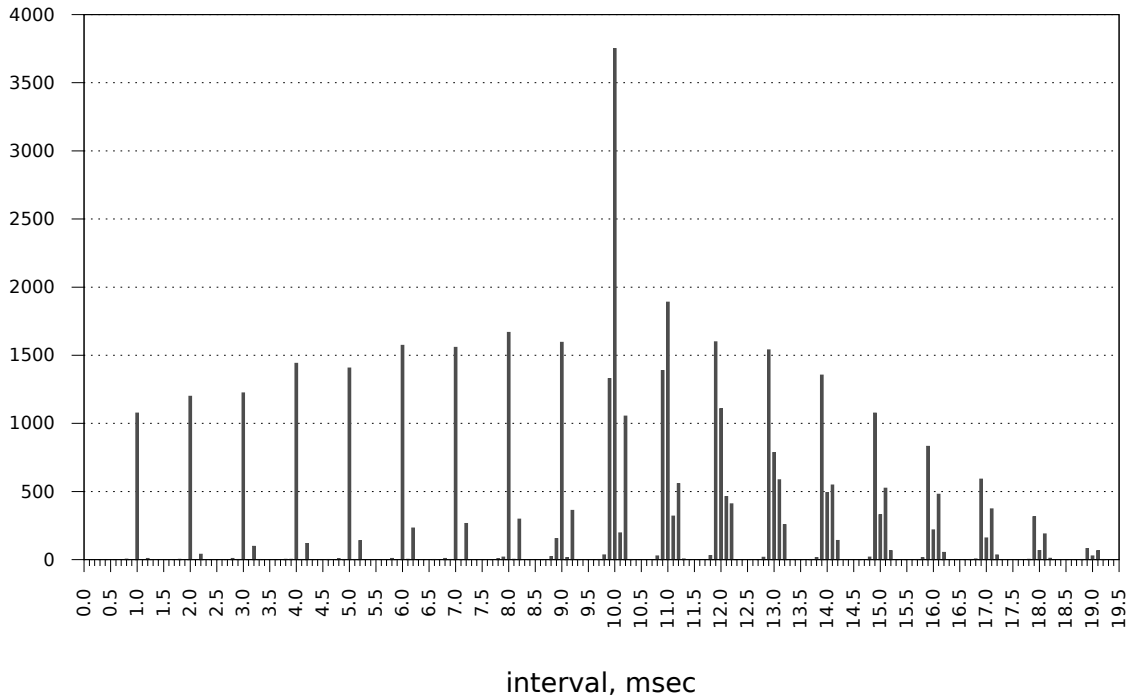



**Figure 1.** Distribution of time between CPU usage values increments for user, system, and nice modes

CPU usage increments is on X-axis, and number of increments that occurred at these intervals is shown on Y-axis. We see that most of CPU usage values updates happen at intervals that are

multiples of 1 millisecond, which is 1/`HZ` second (`HZ`=1000 for OS installed on our test machine). The peak at 10.0 milliseconds (1/100 th of a second) is caused by the fact that the data exported to user is rescaled to units of 1/100 ths of a second.

The load for the test was produced by two threads both bound to the same CPU. One thread was mostly iterating in an empty loop and sometimes performed `nanosleep` [13] system call with incorrect input parameters, thus creating a bit of system mode load. `nanosleep` system call delays the calling process for a time measured in nanoseconds, but in fact `nanosleep` resolution is rougher than nanosecond. Argument to `nanosleep` is a structure with separate members for seconds and nanoseconds to sleep, so when the nanoseconds member is set to a value higher than $10^9$, the value is incorrect and the call returns immediately, quickly switching to system mode and back. The second thread was mostly performing `nanosleep` call with incorrect parameter thus creating mostly system mode load and sometimes making some iterations in an empty loop. This mix of user mode and system mode load on single CPU made system update values for user, and system modes happen frequently.

When we changed the test to look for intervals between increments of CPU usage data for all modes (thus including idle, iowait, irq, softirq and virtualization-related modes into consideration), the results changed, see fig. 2. CPU usage data increments happen mostly at intervals that are multiples of 0.2 millisecond, which definitely can't happen only on timer interrupts as they are known to happen at intervals of 1 millisecond.

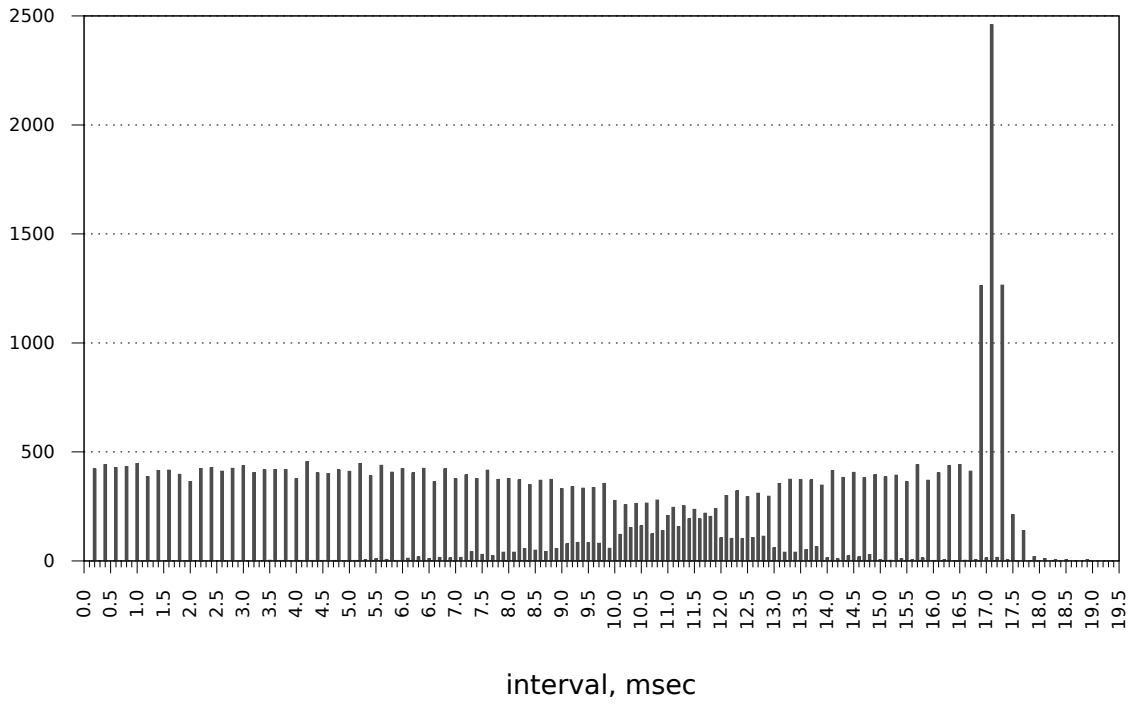



**Figure 2.** Distribution of time between CPU usage values increments for all modes

The load for this test was produced by a thread with mixed user mode load and idle state. The thread performed an empty loop (for user mode load) and made `nanosleep` call to introduce some idle time for the CPU to allow idle CPU usage values to be incremented.

We can't explain why the intervals between CPU usage value increments are the multiples of 0.2 milliseconds. We propose that this value is somehow connected with hardware timer resolution, but this requires further research.

### 3.2. Estimating the accuracy of CPU usage data

Our second experiment tries to estimate the error introduced by the fact that CPU state (for user and system mode) is examined only when timer interrupt occurs and no changes between interrupts are accounted for. The test pseudocode is shown in fig. 3

```
for (int j = 0; j < 10000000; ++j)
  nanosleep(&delay, NULL);
```

**Figure 3.** Test pseudocode

The test is just a `nanosleep` call in a loop. We use different values for the delay. To have zero-length delay we use an incorrect value so `nanosleep` returns immediately. Thus we can measure the time needed for performing all the work except sleep itself. The results are shown in fig. 4 and fig. 5, and the data with some uninteresting points omitted are given in tab. 1.

Run time for different delays is shown in fig. 4. The 'Real' line is the total run time for the test measured with the `time` [14] Linux utility. The 'Estimated' line is the requested delay length multiplied by the number of iterations. Both lines are parallel for delays greater than 200 microseconds, so we may assume that the real delay introduced by `nanosleep` is quite accurate for such delays. For delays less than 200 microseconds the real delay seems to be less accurate, but still it is approximately equal to the requested delay.

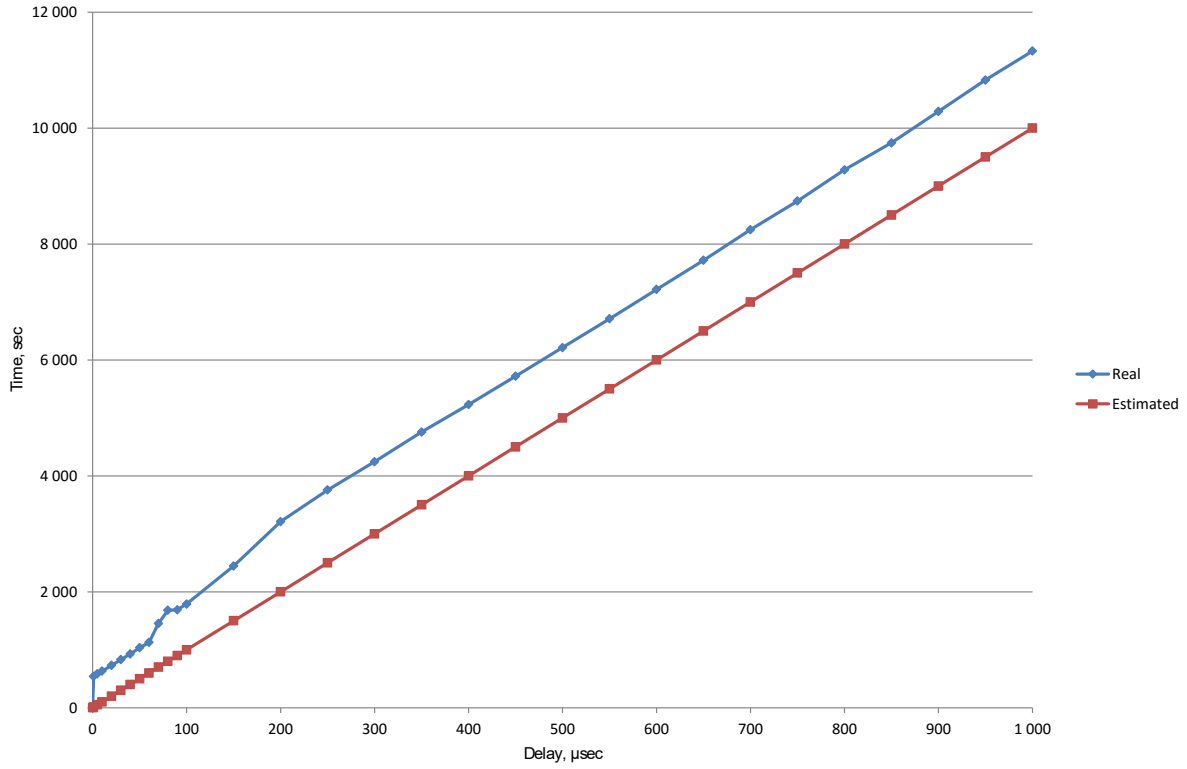



**Figure 4.** Real and estimated runtime for the delay test

But the values measured for the times the test program spent in user and system mode during execution are not so accurate (see fig. 5). We expected beforehand that the values for user and system time will be the same as in a no-delay run (0.29 and 0.32 seconds, respectively), as user mode and system mode work done by the processor seem to be independent of the delay

value. But the results are not so trivial. For the delays greater than 200 microseconds user time is approximately constant, but for the lesser delays the values are a bit chaotic. For system time the results are even more strange. System time is approximately constant for the delays in range from 200 to 700 microseconds, but has unpredictable values outside that range.

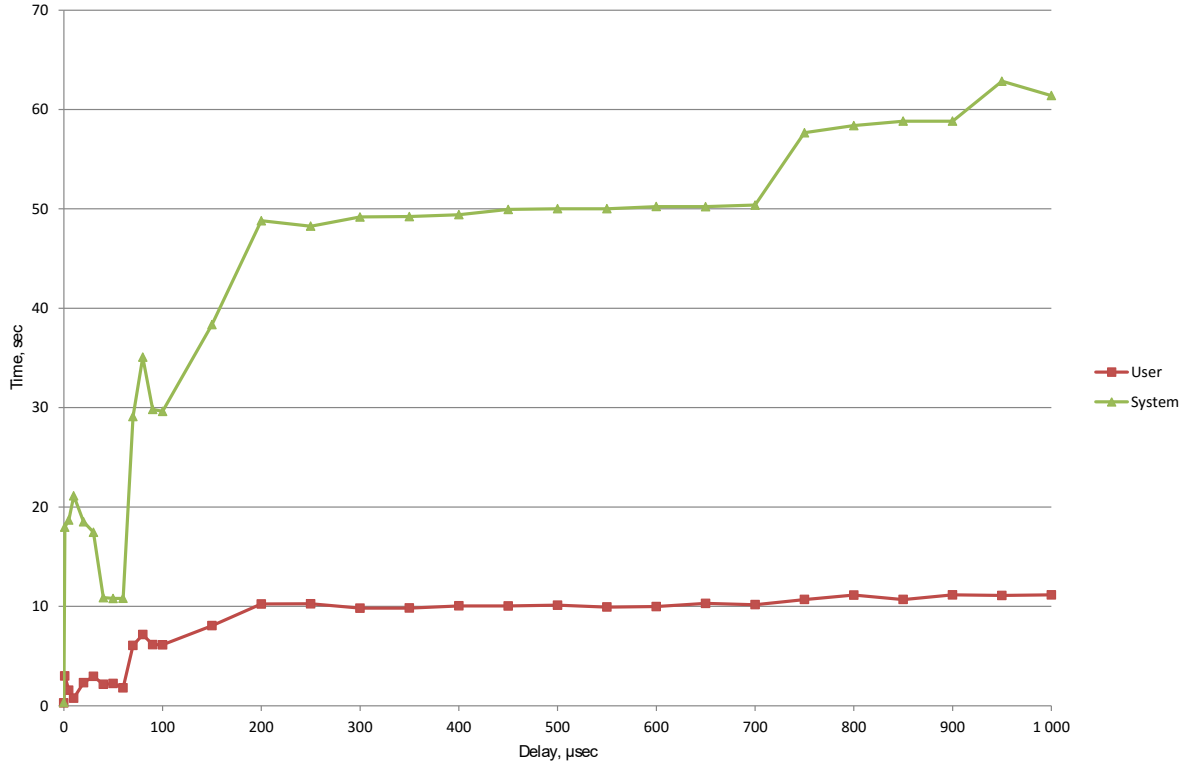



**Figure 5.** User and system mode times for the delay test

The only explanation we see is that some of the delays are somehow synchronized with timer interrupts, and as check for the CPU mode is done on timer interrupts, the results for user and system time depend on the fraction of delays which overlap with timer interrupts.

For example, if we compare a no delay run and a run with 90-microseconds delay, the same loop with the same system call is accounted 30 times more for user time when inserting a real sleep, more than 7 times for system time. For this example we chose a delay value to have the overaccounting effect high. But how high is the probability that such synchronization occurs in real world applications? Of course this question demands further research, but at least scheduling events are done in timer interrupt handler, and it is known that in networks seemingly unconnected independent events tend to synchronize [15]. As HPC applications use network for communication, we may expect similar effect as well.

## Conclusion and Future Work

We analyzed CPU usage data provided by the Linux kernel and how CPU load level is calculated based on these data. The result is that to have the precision of CPU load level percentage of 1 decimal place (when CPU load level is measured in percent of full load) one should sample CPU usage data no more frequently than once every 10 seconds. CPU usage data are not continuously updated, they are updated on timer interrupt which occurs HZ (common values are 250 or 1000) times per second. When calculating accounting data for user, system,

**Table 1.** The results of the delay test

| Delay, $\mu$sec | Run time, sec | User time, sec | System time, sec |
|---|---|---|---|
| 0 | 0.62 | 0.29 | 0.32 |
| 1 | 541.05 | 2.99 | 17.98 |
| 5 | 586.24 | 1.55 | 18.69 |
| 10 | 631.19 | 0.77 | 21.13 |
| 20 | 730.32 | 2.31 | 18.51 |
| 30 | 831.06 | 2.96 | 17.46 |
| 40 | 927.86 | 2.15 | 10.88 |
| 50 | 1036.01 | 2.23 | 10.8 |
| 60 | 1128.32 | 1.8 | 10.81 |
| 70 | 1454.09 | 6.07 | 29.1 |
| 80 | 1681.41 | 7.16 | 35.08 |
| 90 | 1690.68 | 6.15 | 29.82 |
| 100 | 1789.82 | 6.13 | 29.62 |
| 150 | 2446.33 | 8.05 | 38.35 |
| 200 | 3211.62 | 10.24 | 48.81 |
| 700 | 8246.95 | 10.17 | 50.38 |
| 750 | 8741.05 | 10.69 | 57.66 |
| 1000 | 11329.95 | 11.16 | 61.4 |

and nice modes, only the state of the system at the moment of the interrupt is examined, no changes in between the interrupts are accounted. Our experiments show that the same amount of work may be measured very differently when delays are introduced between work periods.

Our future task is to run more elaborate tests and find real application examples when delays between periods of work (calculations) affect the accuracy of CPU load level measurements. We think that it will be especially interesting if the delays are done by waiting for communications which is quite a common case for HPC applications.

*The reported study was supported by the RFBR research project No. 16-07-01121.*



## References

1. Dongarra JJ, Moler CB, Bunch JR, Stewart GW. LINPACK User's guide. Society for Industrial and Applied Mathematics; 1979. Available from: `http://epubs.siam.org/doi/book/10.1137/1.9781611971811`.

2. Dongarra J, Heroux MA, Luszczek P. HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems. Knoxville, Tennessee: lectrical Engineering and Computer Sciente Department, Knoxville, Tennessee; 2015. Available from: `http://www.eecs.utk.edu/resources/library/file/1047/ut-eecs-15-736.pdf`.

3. Kluge M, Hartung M. Mapping of RAID Controller Performance Data to the Job History on Large Computing Systems. In: 2014 International Workshop on Data Intensive Scalable Computing Systems. New Orleans, Louisiana, USA; 2014. p. 73–80. Available from: http://conferences.computer.org/discs/2014/papers/7038a073.pdf.

4. Sottile MJ, Minnich RG. Supermon: a high-speed cluster monitoring system. In: Proceedings. IEEE International Conference on Cluster Computing. IEEE Comput. Soc; 2002. p. 39–46. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=1137727.

5. proc(5) - process information pseudo-file system;. Available from: http://linux.die.net/man/5/proc.

6. Korn W, Teller PJ, Castillo G. Just how accurate are performance counters? In: Conference Proceedings of the 2001 IEEE International Performance, Computing, and Communications Conference (Cat. No.01CH37210). IEEE; 2001. p. 303–310. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=918667.

7. Weaver VM, McKee SA. Can hardware performance counters be trusted? In: 2008 IEEE International Symposium on Workload Characterization, IISWC'08. vol. 08; 2008. p. 141–150.

8. Weaver V, Dongarra J. Can hardware performance counters produce expected, deterministic results. Proceedings of Third Workshop on Functionality of Hardware Performance Monitoring. 2010;Available from: http://icl.cs.utk.edu/news_pub/submissions/fhpm2010_weaver.pdf.

9. Weaver VM, Terpstra D, Moore S. Nondeterminism and Overcount in Hardware Counter Implementations. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). Austin, TX: IEEE; 2013. p. 215–224. Available from: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6557172.

10. Moore SV. A Comparison of Counting and Sampling Modes of Using Performance Monitoring Hardware. In: Computational Science ICCS 2002. Springer Berlin Heidelberg; 2002. p. 904–912. Available from: http://link.springer.com/10.1007/3-540-46080-2_95.

11. Smythies D. Linux reported load averages, for example from top and uptime commands, can be incorrect; 2012. Available from: http://www.smythies.com/~doug/network/load_average/.

12. NO_HZ: Reducing Scheduling-Clock Ticks;. Available from: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

13. nanosleep(2): high-resolution sleep;. Available from: https://linux.die.net/man/2/nanosleep.

14. time(1) - time a simple command or give resource usage;. Available from: https://linux.die.net/man/1/time.

15. Floyd S, Jacobson V. The synchronization of periodic routing messages. IEEE/ACM Transactions on Networking. 1994 apr;2(2):122–136. Available from: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=298431.

DOI: 10.14529/jsfi160406

# Parallel algorithm for 3D modeling of monochromatic acoustic field using integral equations

*Mikhail S. Malovichko*[1], *Nikolay I. Khokhlov*[1], *Nikolay B. Yavich*[1], *Michael S. Zhdanov* [2]



We present a parallel algorithm for solution of the three-dimensional Helmholtz equation in the frequency domain using volume-integral equations. The algorithm is applied to seismic forward modeling. The method of integral equations reduces the size of the problem by dividing the geologic model into the anomalous and background parts, but leads to a dense system matrix. Tolerable memory consumption and numerical complexity were achieved by applying an iterative solver, accompanied by an effective matrix-vector multiplication operation, based on the fast Fourier transform. We used OpenMP to speed up the matrix-vector multiplication, while MPI was used to speed up the equation system solver, and also for parallelizing across multiple sources. Practical examples and efficiency tests are presented.

*Keywords: integral equations, acoustics, seismics, MPI, OpenMP.*

## Introduction

The integral equations method (IE) is a well-known wavefield modeling technique [2, 9]. It has a number of attractive properties. It requires discretization for the anomalous volume only. It drastically reduces the size of a problem in many applications. The IE formulation does not require boundary conditions. It also can be naturally used to compute the Fréchet derivatives and for this reason it is very attractive for inverse problems.

The IE method is based on the discretization of the original integral equation and results in a complex-valued dense matrix. In a straightforward implementation, the computational burden becomes prohibitive for large three-dimensional problems, encountered in seismology. Yet some studies, many of them for the two-dimensional solution, have been reported [4–6, 8, 11]. Recently, the full IE method, accompanied by an iterative solver, was applied to the visco-acoustic three-dimensional modeling [1]. The authors of [1] have implemented Green's integral operator for the free space background model via the Fourier transform (this idea itself is quite old). An effective matrix-free implementation allowed them to apply the BiCGStab method [10] to the resulting system of linear equations with the cost per iteration of $O(NlogN)$, where $N$ is the number of model cells. They showed that a realistically large acoustic model can be simulated almost as effectively with the IE method, as it is for finite-different schemes.

In this study, we design a parallel version of the solver for the half-space host medium, and study the efficiency of parallelization.

## 1. Problem formulation

We consider a 3D model consisting of a half-space host medium and an anomalous volume D, confined within the lower half-space. The host medium has piece-wise constant density and acoustic velocity: $\rho_0$ and $c_0$ for the upper half-space; $\rho_b$ and $c_b$ for the lower half space. The anomalous volume has the same background density $\rho_b$, and arbitrary distribution of velocity $c = c(r)$, where $r$ is the position vector. Let $\omega$ be the circular frequency. Assuming a lossless

[1]Moscow Institute of Physics and Technology, Moscow, Russia
[2]The University of Utah, Salt Lake City, USA

medium, we define the anomalous velocity as $c_a = c - c_b$, wavenumber as $k = \omega/c$, background wavenumber as $k_b = \omega/c_b$ and parameter $\chi = 1/c^2 - 1/c_b^2$. The pressure response $p$ to a point source at a given frequency $\omega$ satisfies to the Helmholtz equation, which can be transformed to the following integral equation:

$$p - \mathcal{G}[\omega^2 \chi p] = p_b, \tag{1}$$

where $p_b$ is the background field, $\mathcal{G}$ is the integral operator, defined for any scalar field $f$,

$$\mathcal{G}[f] = \int\limits_D g(r|r')f(r')dV', \tag{2}$$

where $g$ is the background Green's function.

Let the anomalous volume be covered by $N = N_x N_y N_z$ cubical cells. After discretization we receive the following matrix equation:

$$Au = b, \tag{3}$$

where $u = (p_1..p_N)$ is the vector of $N$ unknown values approximating $p$ in the center of each cell, $b = (p_{b,1}..p_{b,N})$ are known values of the background field in each cell, $A$ is the scattering matrix,

$$A = (I - GX), \tag{4}$$

where $I$ is the identity matrix, $X = diag(\chi_1..\chi_N)$ is a diagonal matrix formed by contrasts for each cell, $G$ contains integrals of Green's function over cells,

$$G_{mn} = \int\limits_{D_n} g(r_m|r')dV', \quad m, n = 1..N. \tag{5}$$

Matrix $A$ is a dense complex-valued non-hermitian matrix with real-valued spectrum. Its condition number substantially improves at lower frequencies since in this case matrix spectrum has a weak dependence on the grid step size and velocity model. These properties make the IE equation system more attractive for iterative solution than that of the finite-difference method.

To solve system (3) we use the unpreconditioned BiCGStab iterative solver. An effective implementation of the matrix-vector multiplication is critical for iterative solutions.

In case of a layered background medium Green's function $g$ has lateral symmetry, i.e.

$$g(r|r') = g(x - x', y - y', z, z'). \tag{6}$$

Thus, for interior points ($r \in D$),

$$I_{nml} = \sum_{k=1}^{N_z} \left( \sum_{j=1}^{N_y} \sum_{i=1}^{N_x} G(x_n - x_i, y_m - y_j, z_l, z_k) f(x_i, y_j, z_k), \right) \tag{7}$$

where $I_{nml}$ is the value of $\mathcal{G}[f]$ for a cell located at $(x_n, y_m, z_l)$. The expression inside the outer parenthesis is essentially a 2D convolution for given $k$ and $l$. Being implemented with 2D FFT, it takes $O(N_x N_y log(N_x N_y))$. This should be performed $N_z^2$ times (for every $k$-$l$ combination). Finally, the total complexity for matrix-vector multiplication in the case of a layered host model is $O(N N_z log(N_x N_y))$. The memory requirements is $O(N N_z)$, though this amount can be reduced to $O(N)$ at the expense of increased running time if the values of $G$ are calculated on-the-fly.

## 2. Parallelization

Our code exploits three levels of parallelism: parallel execution of the matrix-vector product, parallel system solution, and data decomposition across multiple acoustic sources (Figure 1).

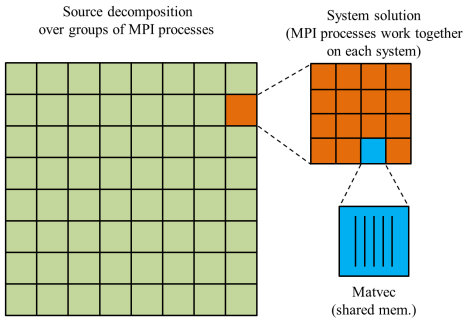



**Figure 1.** Layers of parallelism on a hybrid cluster

The BiCGStab solver requires four inner products and two matrix-vector products per iteration. The inner products are easily parallelized, though in distributed-memory systems some collective communication is required. In this study we have used the standard MPI collective communication routines. This overhead is negligible for our typical tasks.

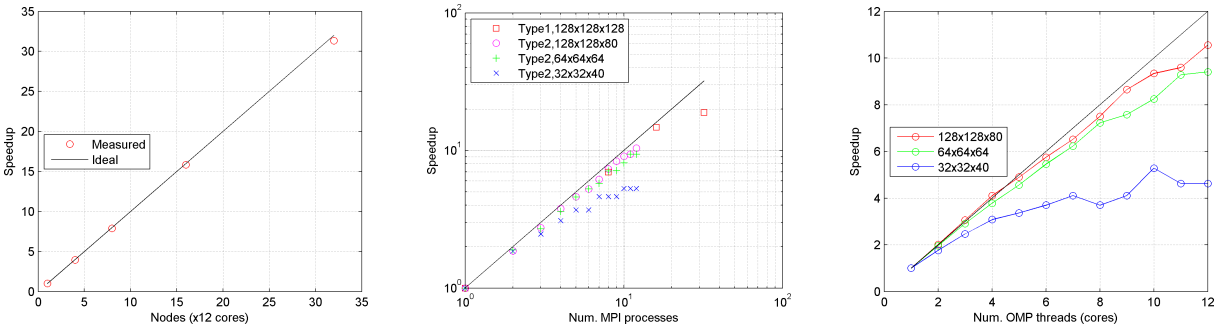
The matrix-vector multiplications are easily parallelized within a single computational node with OpenMP. On a distributed-memory system, this operation becomes the most problematic part, because each model cell is connected with all other cells through the integral operator. No matter what distribution scheme is selected (splitting the matrix into strips across MPI processes with subsequent assembling the resulting vector on a single MPI process, or splitting the matrix into vertical strips and subsequent reduction of resulting vectors on a single MPI process), when the matrix-vector product is computed, it must be sent to all other MPI processes to update their copies. It can be expected that this level scales to $O(logP)$ where $P$ is the number of processes that communicate during the system solutions [7]. This may create huge network traffic for large $P$.

The highest level of parallelism is the distribution of different seismic sources across several MPI groups. In our tasks a forward problem typically has to be solved for many sources (several thousand) for the same model. Naturally, the available pool of MPI processes is divided into several groups. All processes in a group have their copies of model parameters and simultaneously work on a subset of sources, processing them one by one.

## 3. Numerical tests

In this section we present the results of numerical experiments. We used two types of hardware. All tests, involving several computing nodes, such as those shown in Figure 2a and Figure 2b (labels "Type 1"), have been performed on a hybrid cluster. Each node consisted of twelve-core Intel Xeon (Westmere X5660) processors running at 2.8GHz and was equipped with 24Gb RAM and QDR Infiniband interconnect. All tests on a shared-memory system (Figure 2b, labels "Type 2", and Figure 2c) have been performed on a workstation with a single twelve-core Intel Xeon E5-2620 processor running at 2.10GHz.

The highest level of parallelism, i.e. the distribution of seismic sources across different MPI groups, should scale almost linearly in $K$, where $K$ is the number of equally-sized MPI groups. In our tests, which involved a moderate number of nodes, the overhead can be neglected. For




(a) (b) (c)

**Figure 2.** Scalability tests. (a) - distribution of sources across MPI groups, (b) - system solution with several MPI processes, (c) - matrix-vector parallelization with OpenMP threads

the first test we selected a model with 2;097;152 cells (128 cells in each direction). There were 32 seismic sources, located at the surface. The size of a single MPI group was constant and set to 1. We ran computations with 1, 4, 8, 16, and 32 nodes, i.e. each node had to solve 32, 8, 4, 2, and 1 forward problems, respectively. The resulting speedup (Figure 2a) confirmed, that the task decomposition has produced a good acceleration.

In the next test, only one forward problem (a single source) was solved. The iterative solver ran in parallel on varying number of MPI processes. The curve labeled "Type 1" (Figure 2b) was obtained on a cluster, on which one MPI process was running on a single node with 12 cores. The curves named "Type 2" have been obtained on a single 12-core node, where each core was running one MPI process. For this test we used four different models, in which the number of cells varied from 40;960 cells to 2;097;152 cells (see the plot legend). The speedup rapidly deteriorates as the number of processes per system grows. For the largest model, the computational time dominates the communication time until 16 process per system. For smaller models the speedup began to deteriorate at lower $K$.

In the third test, we studied the performance of the matrix-vector multiplication leveraged with the shared-memory parallelization. There were three runs with three models of different sizes (Figure 2c). The resulting speedup curves revealed fairly good scaling versus the number of cores. The speedup for the smallest model suffered from the fact that the background field $p_b$ was computed on the master thread. However, for the larger models where the matrix-vector operations dominated, the efficiency was above 86%.

## Conclusion

We have designed a parallel solver for 3D frequency-domain modeling of an acoustic field. The equation system is solved iteratively; the matrix-vector product is performed via FFT. It makes the computational complexity and memory requirements tolerable for realistically large problems. We have extended this approach by parallelizing the code at three levels: the distribution of seismic sources across different groups of MPI processes, solution of the equation system with several MPI processes, and parallelization of the matrix-vector multiplication over OpenMP threads.

We have studied the efficiency of all three levels of parallelism. The parallelization of the system solution has been found to be the most difficult part, because the system matrix is dense. This limited the speedup of this level. Presumably, any algorithms, based on the volume

integral equations would have similar problems. On the other hand, coarse-grained (distribution of sources) and fine-grained parallelism (matrix-vector product) have good scalability. We expect, that the presented approach might be quite effective on CPU-GPU clusters, especially in case of multi-frequency and multi-source simulation, which are often encountered in seismic applications.

The presented results confirm that the integral-equation modeling can be applied to realistic problems, and is quite promising for seismic applications involving multiple sources/frequencies.

*This research was supported by the Russian Science Foundation, project No. 16-11-10188.*



## References

1. Abubakar, A., T.M. Habashy. Three-dimensional visco-acoustic modeling using a renormalized integral equation iterative solver. //Journal of computational physics, vol. 249, pp.1-12, 2013.

2. Aki, K. and P. G. Richards. Quantitative seismology. 1980. W. R. Freeman and Co.

3. Aminzadeh, F., Brac, J., Kunz, T. 3-D Salt and Overthrust Models. SEG/EAGE 3-D Modeling Series No.1., Soc. Explor. Geophysicists, Tulsa, 1997.

4. Freter, H. An Integral Equation Method for Seismic Modelling// in Inversion Theory and Practice of Geophysical Data Inversion, vol. 5, 1992. Vieweg+Teubner Verlag.

5. Fu, Li-Yun. Numerical study of generalized Lippmann-Schwinger integral equation including surface topography. //Geophysics, vol. 68, no. 2, pp.665-671, 2003.

6. Fu, Li-Yun, Yong-Guang Mu, Huey-Ju Yang. Forward problem of nonlinear Fredholm integral equation in reference medium via velocity-weighted wavefield function. //Geophysics, vol. 62, no. 2, pp.650-656, 1997.

7. Grama, A., A. Gupta, G. Karypis, V. Kumar Introduction to parallel computing, 2nd ed. Addison Wesley, 2003.

8. Johnson, S.A., Y. Zhou, M.J. Berggren, M.L. Tracy. Acoustic Inverse Scattering Solutions by Moment Methods and Back Propagation // Conference on inverse scattering: theory and applications. 1983. SIAM.

9. Morse, P. M. and Feshbach, H. Methods in theoretical physics. 1953, McGraw-Hill Book Company, inc.

10. Saad, Y. Iterative methods for sparse linear systems, 2nd ed., SIAM, Philadelphia, PA, USA. 2003.

11. Zhang, Rongfeng , Tadeusz J. Ulrych. Seismic forward modeling by integral equation and some practical considerations. //SEG Technical Program Expanded Abstracts 2000, chp. 593, pp. 2329-2332, SEG, 2000.