

Distributed Multi-tenant In Situ Analysis Using Galaxy



Greg Abram, Paul Navrátil, David Rogers, and James Ahrens

Abstract Galaxy is a multi-tenant server platform for visualization and analysis of data distributed across many nodes of a supercomputer. It presents an interface through which multiple concurrent clients can connect to perform visualization and analysis tasks on shared data. Galaxy minimizes the memory footprint of visualization and analysis tasks to reduce the impact of analysis on the simulation. It does so by utilizing a visualization renderer that incorporates common visualization techniques directly in the rendering process. The client GUI provides a data-flow programming paradigm that enables users to connect to the Galaxy Multiserver, devise complex multi-step analytics workflows, and visualize results on their desktop.

1 Introduction

As discussed in the opening chapter, modern supercomputing-level computation calls for in situ analysis. By analyzing the data *as it is being generated*, in situ analysis avoids the necessity of saving time-step data snapshots for post-processing, a costly process that generally limits the temporal frequency at which data can be saved. However, for current systems (e.g. ParaView/Catalyst [3], VisIt/libsim [5], SENSEI [4]) these benefits bear significant costs. For example, the analysis to be performed must be known prior to the start of the computation so that the necessary changes can be made to the general-purpose simulation. Within these systems, the

G. Abram · P. Navrátil (✉)

Texas Advanced Computing Center, The University of Texas, Austin, TX, USA
e-mail: pnav@tacc.utexas.edu

G. Abram

e-mail: gda@tacc.utexas.edu

D. Rogers · J. Ahrens

Los Alamos National Laboratory, Los Alamos, NM, USA
e-mail: dhr@lanl.gov

J. Ahrens

e-mail: ahrens@lanl.gov

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
H. Childs et al. (eds.), *In Situ Visualization for Computational Science*,
Mathematics and Visualization, https://doi.org/10.1007/978-3-030-81627-8_19

distributed simulation process is instrumented with an adapter that converts the simulation's run-time data to a form palatable for analysis. This results in a monolithic simulation/analysis code that, when run, performs a specific simulation/analysis task, and requires two different customizations: to the simulation, so that the simulation provides the required internal data to the adapter, and to the analysis, to cause it to perform the specific analysis required by the science case.

Typically, such systems co-locate simulation and analysis work in the same actual processes. Consequently, simulation and analysis share memory, computation and communication resources and therefore affect the performance of one another. The available memory available to each process must be sufficient to serve the worst cases load of each phase. Unfortunately, while the footprint of many simulations can be determined, the footprint and computational cost of typical geometry-pipeline visualization algorithms is hard to predict and, in the worst case, is tied to the size of the underlying computational grid. Further, as distributed supercomputers become ever larger, system-wide synchronization becomes costly. When the workload of a task is not evenly distributed, synchronizing may cause much of the computational resources assigned the task to wait for a few overburdened participants to complete.

Given these challenges, Galaxy aims to provide the following differentiating capabilities: (1) separation of simulation and analysis activities so that multiple, concurrent and asynchronous analyses of simulation data products can attach, operate and disconnect at run-time; and (2) performance of data visualization within bounded, well-understood memory limits. Galaxy delivers these capabilities via a multiserver framework, which provides each connected tenant access to Galaxy's services for distributed data, communication, and computation; and Galaxy's filtering, analysis, and visualization methods, which are designed to limit tenant resource requirements rather than requiring full use or duplication of the input data. Galaxy replaces traditional monolithic in situ architecture with a shared distributed data- and computation-space that is accessible to concurrent data-processing activities. These activities can be organized adaptively, in real-time, into distinct workflows that can access shared data at run-time to perform different science tasks concurrently. Using Galaxy, a simulation workflow consists of one or more stages "connected" by data exposed in the shared data-space. Workflow produce data products that are stored within Galaxy's shared data-space. As activities attach to the Galaxy environment, they can access data products available within the data-space to generate additional data products and analysis results.

In the remainder of this chapter, we briefly discuss relevant related work not covered elsewhere in this book (Sect. 2). We then provide a high-level overview of Galaxy's architecture, focusing on the novel multi-tenant and in situ capabilities (Sect. 3; a description of additional Galaxy functionality is presented in [1]). We then provide a deep-dive into the algorithms and structure of Galaxy's ray tracer and direct volume renderer, a complex Galaxy analysis that can be used in situ with simulations (Sect. 4). Lastly, we provide examples of Galaxy's performance on current-generation hardware architectures (Sect. 5).

2 Related Work

A substantial body of literature exists on in situ systems, as outlined in the introductory chapter of this book. We discuss four in situ frameworks below for their relevance to Galaxy, for their large user bases, their ready availability, and their deployment at many supercomputing centers around the world. We consider these four to be divided into two subsets—*closed systems*, that are specific to a analysis/visualization tool, and *middlewares*, which connect instrumented simulations with analysis/visualization components that have appropriate interfaces available.

ParaView Catalyst [3] and VisIt's libsim [5] are closed systems that enable the use of pre-existing, well-developed visualization systems in situ. In each, the simulation is instrumented with an application-specific API that provides an interface to the visualization system. In general, both operate by incorporating portions of the visualization system's run-time library into the simulation process to perform the visualization. Both ParaView and VisIt's visualization engines utilize VTK [9], a class library of filters that leverage high-speed rasterization hardware (and software) by extracting renderable geometry from the input data. The resulting geometry is rendered with OpenGL and both systems use depth-compositing to combine the rendered results from many parallel processes into a single image for display/storage. In their most recent versions, both have incorporated ray tracing backends in addition to OpenGL; however, each still utilizes depth-compositing to create final images, which limits ray tracing effects to only process-local data.

Libsim is designed to interface a simulation with VisIt's GUI. By connecting the two, libsim provides an interactive view into the running simulation using the full capabilities of the VisIt runtime in a directly-integrated, synchronous manner. The user can change the visualization as it is running via the VisIt GUI. Catalyst supports the development of visualization pipelines in a pre-process step, using the ParaView GUI and representative data, generating Python scripts that capture the pipeline to be deployed in situ. When the simulation is started, the Catalyst run-time loads the Python script and, when passed data from the simulation, performs the visualization in a directly interfaced, fully synchronous manner. We note that the Python script can use VTK's serialized I/O capabilities to interface with an external visualization process. ParaView itself can then connect to the simulation using this export interface to provide an indirect, intermittent ability to interact with the simulation data as it is running.

ADIOS [6] and SENSEI [4] are middleware for in situ analysis that facilitate communication of data between source and destination activities. They are based on an externally reconfigurable I/O layer that connects the output of a source application with the inputs of the destination application. Both include an API used to instrument the simulation code to support a variety of I/O and analysis. Once the implementation costs have been paid to integrate the API within the simulation, each middleware simplifies the addition and substitution of analysis components. As middleware,

ADIOS and SENSEI do not provide analysis/visualization components. Instead, they provide an interface to other well developed systems, including ParaView and VisIt.

3 Galaxy Overview

Galaxy [1] is a parallel computation environment in which distributed parallel computation is performed by applications that intercommunicate (source code is freely available at <https://github.com/TACC/Galaxy/>). Like ADIOS and SENSEI, Galaxy serves as middleware that enables communication between activities (e.g., simulations and analysis/visualization components). However, unlike those systems, Galaxy's distributed computation model is intended for use as an algorithmic framework. Two components are central to the Galaxy architecture: a *global name-space* into which data objects are published, and an asynchronous point-to-point and broadcast messaging framework for *work messages*. These components enable any worker to cause one or all of its fellow workers to perform a parameterized operation on data by sending a work packet containing an *action*, parameters, and the global names of sources and destinations. When a work packet is processed by its recipient, the *local* data objects associated with the global names of sources and destinations are identified, and the action performed. Actions can, in turn, create and transmit additional work packets requesting work be done elsewhere in the system.

As an example, a simple data parallel algorithm can be implemented in Galaxy by subclassing the `Work` class and defining (1) its action to be the per-local-data operation to be performed, (2) its members as the source and destination object names and, (3) whatever parameters are necessary for the algorithm at hand. This work code would be invoked by a driver procedure that is parameterized with the name of a pre-existing source object, the name of a (possibly preexisting) destination object, and the other parameters. If the destination object does not already exist, the driver would use the Galaxy API to create local objects on the same workers that contain partitions of the source object. The driver would then create an instance of the `Work` object subclass and, using the Galaxy asynchronous broadcast capability, request that each worker perform the requested action on its local source data and produce results in its local destination object.

Galaxy also supports global communication. For some algorithms, the driver may need to know that the data-parallel processes have all completed (recall that the work messages are fully asynchronous). Furthermore, the distributed destination object may need to acquire some global data (e.g., the data range). These examples require the workers to communicate status/results to the driver upon completion of their data-parallel work. For the data range example, the driver could create a data structure for a response object, initializing a count variable with the total number of data-parallel workers. It would then pass the response object's address (in the driver's memory space) to the workers. A second application-specific `Work` subclass would be defined with an action method that accesses the driver's data structure, stores the associated

worker's data range, and decrements the driver's count of worker results accumulated thus far. After broadcasting the original work message, the driver would wait for the worker count to decrement zero, at which time it knows (a) that all data-parallel work is complete and, (b) it has the accumulated global data range information.

3.1 Multi-tenancy

Galaxy is intended for use as a multi-tenant, persistent co-processing environment. In the simple examples described thus far, a single driver causes single parallel algorithm to run. However, Galaxy supports multiple algorithms running concurrently, sharing the messaging system, the global object name-space, and computational resources. Consequently, many work messages can exist simultaneously, either in work queues waiting for the necessary resources or in a running state. As detailed later in the chapter, each worker supports multiple *worker threads* which process messages concurrently. This enables Galaxy to support direct, co-located, shared-memory co-processing.

The Galaxy `Multiserver` implements a socket-based client/server interface enabling multiple external user interfaces to asynchronously attach to a running Galaxy instance, install activity-specific capabilities (e.g., libraries containing subclasses of Galaxy's `Work` class etc.) on all the workers, and run concurrently as part of the Galaxy world. `Multiserver` clients communicate with the server via a simple string-based protocol. When run as a `Multiserver`, one process of the Galaxy spawns a thread that opens a socket and listens on it. When a `Multiserver` client attaches, a client-specific thread is spawned on the recipient Galaxy process to handle communications, and a message handler is installed for a few general purpose actions.

One *pre-installed* action enables the client to send the name of a shared library to Galaxy. Galaxy's server-side creates and broadcasts a `Work` message that causes every worker process in the Galaxy world to load the library (and its dependencies). A string-based *message handler* is installed, which the client-specific server-side thread retains to process subsequent application-specific exchanges. When subsequent messages arrive, the server side thread passes the message to each installed handler in turn until one recognizes the message, enabling multiple independent Galaxy activities to be installed by any client.

As a trivial example of multi-tenancy, Galaxy includes `msdata`, a `Multiserver` client that presents a command-line interface that enables the user to interact with the Galaxy data space. For example, the code below connects to the Galaxy instance and discovers that there are no visible data objects available.

An Empty Dataspace Listing in Galaxy

```
% msdata
? list;
  datasets:
?
```

We can then start a second, possibly remote, `msdata` instance that imports a dataset into the Galaxy Dataspace:

Importing Data into a Galaxy Dataspace

```
\% msdata
? import { "Datasets": [
  {
    "name": "eightBalls",
    "filename": "eightBalls.vol",
    "type": "Volume" } ]};
?
```

This import statement causes the Galaxy world to load a dataset from the file `"eightballs.vol"`, of type `"Volume"`, and name it `"eightballs"`. The loaded dataset is now visible to both `msdata` instances:

A Populated Dataspace Listing in Galaxy

```
? list;
  datasets:
  eightballs
?
```

Two user interfaces have now connected to Galaxy and installed a shared-data capability in which they can see and access the same data space.

3.1.1 Interactive Rendering Using the Galaxy Multiserver

Galaxy includes a Multiserver client specifically for rendering datasets resident in the Galaxy data space. The *gxyviewer* application accepts a description of a rendering that includes a camera and a set of rendering operators. Rendering operators specify which datasets to render and how to render them using render-time visualization operations such as colormapping, isosurfacing and slicing. These operators are more fully discussed below.

As do all Multiserver clients, *gxyviewer* connects to the server via a socket and installs server-side code to implement a command-protocol between the client and server. In this case, *gxyviewer* implements two functions: one to transfer the initial visualization operators, and one to send camera updates to the server for rendering and to receive buffers of pixels in return. The client then creates an interactive display window, interprets user interaction events as camera transformations, and calls the server to render pixels that it then stores in its display window.

3.1.2 Non-Render-Time Visualization Algorithms

While *gxyviewer* enables the use of the Galaxy renderer directly to perform rendering-time operations (such as slicing, volume rendering, and isosurfacing), real world visualization problems may involve visualization algorithms that cannot be performed strictly at render-time. Such algorithms can themselves be implemented in the Galaxy framework; operating on Galaxy-resident data to create derived data objects.

One such algorithm is particle advection. To visualize flow fields, we implement a Runge-Kutta method to trace particles through a dataset describing a vector field. This implementation traces multiple particles (as many as there are seed points) in parallel through the partitions of the dataset, leaving partial traces in the Galaxy workers responsible for the partitions in which they pass, and using Galaxy asynchronous point-to-point transfers to move the head of particles that encounter partition boundaries to the process responsible for the neighboring partition.

3.1.3 The Galaxy GUI

Galaxy includes a user interface enabling users to build complex visualizations using a visual programming paradigm. This interface is implemented as a Multiserver client which installs a set of visualization algorithms, as described above, to create derived data using non-render-time algorithms and to orchestrate complex multi-faceted visualizations.

Figure 1 shows the Galaxy GUI in use. Datasets from an asteroid-strike simulation have been placed in the Galaxy data space by another client. The visualization shown accesses three of these datasets, describing pressure, density and the gradient of density. It uses a sampling algorithm to select seed points at a density level of interest, creating a temporary particles dataset. The particles and the original gradient vector

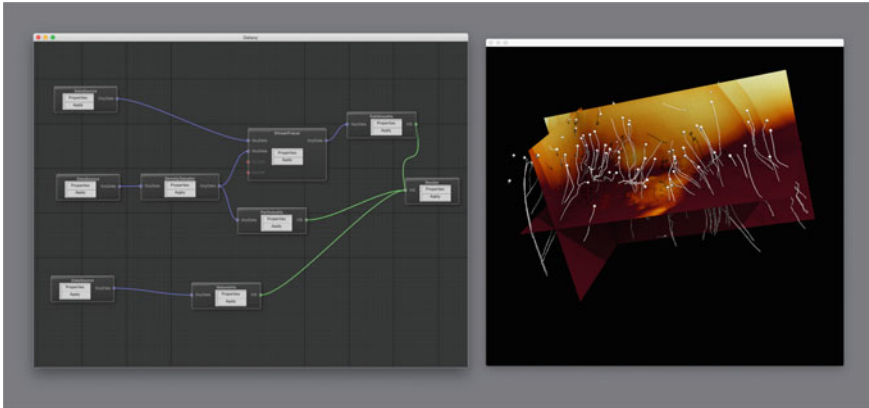


Fig. 1 Galaxy User Interface on the author’s laptop using the Stampede2 supercomputer to visualize data

field are received by the stream tracer, which traces streamlines that represent the density gradient at the sample points. The particles and pathlines and the original volumetric pressure field are passed through filters that enable the user to control how they will be rendered - notably the radius of the spheres and stream tubes and, in the case of the pressure field, the location of slicing planes and color maps to apply. These results are then received by a Renderer filter, which owns an interactive rendering window and, as above with `gxyviewer`, calls Galaxy to render images.

3.2 *Using Galaxy In Situ*

Galaxy can perform *direct, co-located* in situ analysis of simulation data. A simple approach to enable this entails implementing a simulation as a Galaxy client application. In this manner, a simulation would run independently of other clients (such as the analysis client). The simulation client would publish one or more objects intended for access by other clients to a global dataset directory, and analyses can browse the global dataset directory to discover data for analysis. Galaxy incorporates an event architecture (based on the observer pattern) so that upstream publishing clients can signal that data objects (or other global capabilities) have been updated. Downstream clients, having registered interest in data objects, will then be notified when the objects have changed.

In order to work in this manner, preexisting simulations would need to be instrumented to work in the Galaxy environment. While this code transformation may be non-trivial, we note that a client can receive its own private MPI communicator, enabling it to run largely unchanged, interacting with the Galaxy environment only when it explicitly needs to export published data.

Alternatively, Galaxy and a simulation can cooperate in an indirect manner by incorporating a communications client that presents an external parallel interface for separate distributed-parallel applications to attach to and to transmit data. In this approach the application is instrumented with an I/O API that negotiates and sends data to the communications client, and need not run inside the Galaxy computational framework.

We have implemented a simple example of this approach as the SocketConnector Galaxy client using a simple sockets-based protocol. This enables an external application (e.g., simulation) to connect to a Galaxy Multiserver instance, to initialize and publish data objects, and to periodically transfer time-step data to update the Galaxy-side data using parallel process-to-process communication. We note that sockets are simply a convenient communications mechanism that enable cooperation between distributed-parallel applications on the same or different servers; alternatives exist for special cases such as co-located processes using shared memory, or using a single MPI world with separate communicators to enable MPI point-to-point communications.

Galaxy enables different clients to communicate through a common Multiserver instance using events on data objects. Specifically, the server-side components of the DataSource filters on the left side of Fig. 1 attach *observers* on the data objects selected by the GUI user. When these data objects are updated (by a client interfacing with an external simulation), the observers are triggered, causing a message to be passed to the remote GUI client process. This then triggers the data-flow execution of the visualization and updates derived data objects and, ultimately, the rendered result in the GUI display window.

This is in fact what is happening in Fig. 1. A demonstration client (*simsim*, included in the Galaxy distribution) is given two timesteps as VTK `.vti` files and, according to command-line arguments, periodically interpolates time steps in between the two given input datasets and transfers the data to Galaxy using the Socket-Connector in situ interface.

4 The Galaxy Ray Tracing Engine

The original motivating problem behind the development of Galaxy was to investigate the use of ray tracing and ray tracing-based direct volume rendering (DVR) techniques for large scientific data applications. We present a deep discussion of this application here to highlight performance and implementation considerations when using Galaxy components for in situ analysis or other computation.

4.1 Performance in Space and Time

In visualization contexts, ray tracing and DVR offer significant advantages over geometry rasterization systems. Ray tracing and DVR techniques invert the basic loop of many visualization algorithms: where geometry-based systems iterate through the input data elements to determine how they contribute to the output image (at cost $O(N)$ time), ray-based systems iterate through the sample space, asking what input elements contribute to each ray, at cost $O(XYK)$ time, where XY reflects the resolution of the sample space, and K the cost of performing the algorithm along the ray.

In the important case of structured grids (and composite grids consisting of structured subgrids) interpolating the dataset at an arbitrary point is $O(1)$. When the grid is unstructured, a ray-based system will first create an acceleration structure (at cost $O(N \log N)$ time), which is amortized across all uses of the grid; once this is created, interpolation is of cost $O(\log N)$. In the following we demonstrate this advantage in several cases.

- Rather than extract a triangulated surface first, slicing planes can be rendered by a ray tracer by intersecting the ray with the equation of the plane, then interpolating the result.
- Volume rendering is implemented in a DVR system by iterating along the ray (at cost roughly $O(N^{\frac{1}{3}})$). Thus the cost of evaluating each ray is $K = O(N^{\frac{1}{3}} K')$ where $K' = O(1)$, for structured grids, and $O(\log N)$ for unstructured grids for a total cost of $O(XY N^{\frac{1}{3}})$ or $O(XY N^{\frac{1}{3}} \log N)$.
- Similarly, isosurfaces can be rendered by iterating along the ray looking for intervals that contain the isovalue. The performance analysis is the same as above, for volume rendering.
- Other algorithms that cannot be implemented as local operations along a ray can be rendered in a two-pass manner as in geometry-pipeline systems, substituting standard ray tracing in place of geometry-based rasterization. While this will necessitate the creation of an acceleration structure, we note that the cost of this will be $O(K \log K)$ where K is the size of the derived dataset, and K is likely much less than N . Once this acceleration structure is available, the rendering cost reverts to $O(XY \log K)$.

We note that current versions of both ParaView and VisIt incorporate ray tracing back-ends. However, they simply substitute ray tracing to perform geometry-based rasterization; each still relies fully on geometry pipelines and intermediate derived datasets.

4.2 Adapting Galaxy to Be a Sampling Engine

The Galaxy ray tracing engine can be thought of as an abstract engine for processing rays as they pass through data on a massively parallel system. In most cases, we want

to use this engine to render the data, as described above. However, we can use this engine for more abstract operations, including sampling. The simplest way to think about this is that the sampling engine sends rays through a dataset, and produces a new dataset of samples (particles).

A sampling operation is an abstraction of the following operations, which are normally done in the context of rendering the data:

- Intersection of ray with data. Because the engine processes intersections with data, we can write abstract operations evaluated instead of standard intersection operations. For example, we can use rays to find places in the data that have certain data values, gradients, or multi-variate properties.
- Action taken when an intersection occurs. In the case of sampling, we typically would like to take a sample of a data when we detect an intersection or a "hit". Thus, we can produce a dataset of samples, with each sample being taken when a ray intersects with something in the data.
- Ray propagation. When a ray creates a sample, we can decide to stop the ray, change its path based on data, or propagate other rays from that point in space. This gives the engine control over how to continue sampling the space. This sampling capability is a powerful option for exploring properties of the data in a new way.

4.3 Ray Tracing Distributed Simulation Data

In the general case, ray tracing distributed data is difficult. Any arbitrary ray will encounter surfaces that may be distributed anywhere across the distributed system; therefore the *closest* intersection requires gathering information from each process that contains data that *might* intersect the data. An arbitrary region of space may contain surfaces that might be anywhere on the distributed system.

Ray tracing systems generally handle this issue by using a global acceleration structure, such as a bounding volume hierarchy (BVH). If the full dataset is not present on each node, the BVH will necessarily contain only bounding volumes of surfaces (rather than the surfaces themselves) and these bounding volumes will often overlap.

Tracing an arbitrary ray on a given process is done by testing the ray against the global BVH to identify data that *might* contain the next hit point. The ray is then tested against that data (either by transmitting the data to the process containing the data or by requesting the data from that process so that the test be performed locally). The results are then compared to find the actual closest hit points. Much work is being done to efficiently perform this task, notably including *speculative* ray tracing [7], in which each ray is traced against the data in each process.

Fortunately, simulation-based data offers us a simplifying fact: simulations are generally based on *spatially partitioned* data; each cooperating process is assigned a compact region of the computation space and, to the greatest extent possible, handles

its data independently of other partitions using minimal regions (ghost zones) of overlap with neighboring processes.

This means that anything that may affect a ray in a partition subspace is resident within the process responsible for the subspace. Importantly, this is also true of any data *derived* from a partition of data in a typical data-parallel process will also be resident on the node responsible for the partition, modulo an exchange of results near boundaries with the neighbors across the boundaries.

Given this property, the tracing of a ray with the computational space can be performed by breaking it into an *ordered set of non-overlapping intervals* based on its intersection with the partition boundaries, and that the intersection of the ray within each interval can be processed on the node responsible for the corresponding partition completely independently of data resident elsewhere on the system.

4.4 Ray Tracing with Asynchronous Work Messages

Based on the observations above, we have designed an algorithm for ray tracing/DVR in-situ simulation data with global illumination based on the asynchronous passing of rays from process to process as the ray traverses the partitioned computational space. This process is begun by an initial one-to-all `CameraMessage` that carries camera information to every node. Each recipient compares the camera information against its local partition to find exactly those rays that enter the computation space through their local partition. These are bundled onto `RayPacketMessage` objects and queued for processing locally.

When a process processes a `RayPacket`, it knows that the rays in that packet all begin in the local partition. The rays are processed against local data looking for *events* that cause (potentially temporary) termination of the ray (e.g., surface hit points or, importantly, the far, exiting surface of the partition) and each ray is ‘tagged’ to denote its terminating event type. When all the rays in the packet have terminated (as far as the local partition is concerned) the tracing process concludes. Once the local ray tracing of a packet of rays completes, the rays in the packet are examined to determine what occurred in the tracing process.

Rays that struck exiting boundaries of the partition are bundled into new `RayPackets` and sent to the nodes responsible for the neighboring partitionings. Rays that strike surfaces or acquire full opacity are then handled. Two new empty messages are initialized: a `FrameBufferMessage`, to carry results to a remotely hosted frame buffer, and a new `RayPacket` to hold any generated secondary rays.

If a processed ray is a *primary* ray that strikes a surface, we determine material properties, evaluate a lighting model, and place the result in the `FrameBufferMessage`. If the lighting model calls for shadows and/or ambient occlusion, secondary rays are generated and placed in the secondary-ray `RayPacket` (note that these secondary rays will necessarily begin in the current partition); *shadow* rays are tagged to terminate at light sources, and *ambient occlusion* rays are tagged to terminate at some specific distance.

If a processed ray is a *secondary* ray, its reason for terminating is examined. Secondary rays that terminate early indicate that light *was not* received at the hit point. Those secondary rays that don't terminate early represent light that *was* received at the hit point. We note here that we use a *subtractive* lighting model: primary rays are lit as if they are unshadowed and receive full ambient light, and shadows are *subtracted* from the pixel buffer if it is determined that the surface point is unlit. *Early terminated* secondary rays are appended to the `FramebufferMessage` and the remainder are dropped.

4.5 Asynchronous Rendering in Galaxy

In the previous section we have outlined the basic distributed Galaxy ray tracing algorithm that moves rays between processes as the rays pass through partitions of the computation space. One process, responsible for accumulating the final image, receives `FramebufferMessages` and adds the contents of each to the frame buffer. The image updates progressively over time, with primary Phong-lit contributions tending to come in first, and shadows and ambient occlusion contributions coming later. In an interactive session, this provides a form of incremental refinement. As the user moves the camera, new frames are initiated, causing the many rendering worker processes to spawn new primary rays, and also to ignore any further in-flight ray- or framebuffer-messages. When we need to know that an image has been completed (for example, to write a file), we need to know that all ray- and frame-buffer messages have been retired. In a distributed system this is non-trivial. Each ray- and framebuffer-message must be fully accounted for; a message sent from node to node is the responsibility of the sender until a receipt message is received from the recipient.

Galaxy uses a binary-tree message-based algorithm to verify that all messages have been retired. A subtree is determined to be idle if there are no ray- or framebuffer messages "alive" in the subtree. When a subtree enters or exits the idle state, it sends a notification message to its parent. These messages cause the parent to consider its own state. Its next state will be a combination of its children's last received state and any live messages it is working on. Again, when its state changes, a notification message is passed upward. When the root node enters the idle state, it knows (a) that it itself does not own any live messages and (b) that when last informed, each child subtree was also idle. Unfortunately, strange conditions may occur. An exchange of message ownership from a leaf of one subtree to a leaf of the other may cause one subtree to enter the idle state and the other to exit it; however, the idle notification from one subtree may reach the root before the not-idle notification arrives from the other subtree. This necessitates a final downward pass through the tree when the root process enters the idle state. This is effectively a synchronization point of the process, though it may be performed either using low-level synchronous communication (e.g., MPI collectives) or asynchronous point-to-point messages.

4.6 Visualization Specification with Galaxy

The capabilities described above support much of what is required to do simple ray tracing: for each ray, find the closest surface hit point, determine appropriate pixel based on surface properties, materials and the lighting environment, and store the results into a frame buffer. Then, transfer sets of rays across processors to evaluate them with respect the global data of the scene.

However, for visualization, we want to do more than simply intersect rays against pre-existing surfaces; we want to do processing *along the ray*, including integration for volume rendering, the intersection of rays with planes that slice volumetric data, and the procedural visualization of isosurfaces. Thus Galaxy requires the camera, a lighting environment, a destination frame buffer, the objects to be displayed, and a specification for how to visualize each data object. The visualization of an object is, in fact, independent of the object (potentially, multiple visualizations might be simultaneously operating on the same object). For this reason, Galaxy visualizations do not annotate objects with parameters (isovalues, clipping planes, colormaps, etc.) but instead refer to constituent datasets indirectly, through a Galaxy object referred to as a `Vis` object, subclassed for the several currently-supported data types: volumes, surfaces, particle sets and pathline sets.

- **SurfaceVis** The simplest data object to render are surfaces. Surfaces consist of a triangle mesh with data values associated with vertices. The `SurfaceVis` object refers to a surface object and associates a color map with it.
- **ParticleVis** Particles are 3D points with data values associated. Particles are rendered in Galaxy as procedural spheres, where both the radius and color of each is determined by the data value. Thus, the `ParticleVis` also associates a colormap with a particles data object, but also includes a linear map of data to radius: (v_0, R_0) and (v_1, R_1) . Particles with values less than or equal to v_0 will be radius R_0 , values between v_0 and v_1 result in a linear interpolation between R_0 and R_1 , and remaining particles will have radius R_1 .
- **PathLineVis** Pathlines are sequences of 3D points with associated data values. Like Particles, Pathlines assign color via a colormap and determine stream-tube radius using a (v_0, R_0) – (v_1, R_1) linear map.
- **VolumeVis** Volumes have the most degrees of freedom when it comes to visualization. A volume can be volume rendered; a `VolumeVis` therefore includes a boolean flag. If the flag is true, both a colormap and opacity map are required and the volume is volume rendered. If the volume is *not* being volume rendered it can be visualized as sets of isosurfaces and slicing planes. A `VolumeVis` object therefore also manages a set of isovalues and planar equations. These also require a colormap; all surfaces rendered using the same `Vis` object use the same colormap.

4.7 Galaxy Ray Processing

As described, `RayList` processing involves tracing rays as they pass through a region of space. This is implemented, at the lowest level, in ISPC [8]—an Intel parallel computing language that effectively vectorizes kernels over multiple rays simultaneously. This ray processing procedure receives a set of rays and a set of `Vis` objects that identify the objects to visualize and how to visualize them. It assumes that the rays begin inside the local subspace, either at an entry boundary surface, or fully inside it.

The first action intersects the rays with the exiting surfaces of the subspace. This determines the longest interval of the ray that *might* be traced locally. It then intersects the rays with the explicit surfaces (e.g., triangle sets) and procedurally defined implicit surfaces (e.g., particle spheres, pathline tubes, volume slicing planes) and if any are hit, updates the farthest point of the ray interval to the hit point. If any `VolumeVis` objects request isosurfaces or volume rendering, the processing iterates along the ray from entry to closest hit point, accruing volume contributions and searching for isosurface crossings along the way. If an integration interval traps an isovalue, a linear interpolation is done to approximate the hit point.

4.8 Galaxy and Cinema

Cinema [2] is a means by which limited interactive exploration of the dataset can be performed post-hoc by building a database of rendered images using visualization parameter sweeps (e.g. camera rotation, isovalue iteration) and combining the resulting images to mimic direct data exploration. Galaxy supports the efficient generation of Cinema databases by rendering many visualization settings simultaneously on each data timestep. Each of these visualizations is defined by rendering context; thus, each ray list, with its reference to a global rendering context, contains all the information necessary to process it. Since multiple rendering contexts (each including a reference to a destination frame buffer) can co-exist, it is possible to launch any number of visualizations simultaneously. Since Cinema databases very often contain many viewpoints, this serves to load-balance; rays will enter the computational space from many different sources, and will therefore engage different sets of worker processes. We present performance results for Cinema database generation in Sect. 5.

5 Galaxy Performance

This section presents Galaxy performance on representative datasets across two current hardware configurations. A deeper evaluation of Galaxy performance compared to Intel OSPRay is presented in [1].

We evaluate Galaxy’s performance using two currently-deployed advanced computing resources at the Texas Advanced Computing Center (TACC): Stampede2, with Intel Xeon Platinum 8160 “Skylake” processors and an Intel Omni-Path (OPA) 100-Gb/s fat-tree interconnect; and Hikari, with Intel Xeon E5-2690 v3 “Haswell-EP” processors and a Mellanox EDR 100-Gb/s fat-tree interconnect. Each Stampede2 Skylake node has 192 GB of RAM and two 24-core processors with two hardware hyperthreads per core (96 total threads). Each Hikari Haswell node has 64 GB of RAM and two 12-core processors with two hardware hyperthreads per core (48 total threads).

For each hardware configuration, we simulate tightly-coupled in situ analysis by rendering a camera orbit of 500 frames, each rendered at 1080p resolution. We render fifty frames simultaneously across the available hardware resources, with new frames initiated asynchronously as prior frames are completed. We perform these tests using three datasets: a volumetric simulation data of a deep water asteroid impact from Los Alamos National Laboratory ($1380 \times 840 \times 720$ grid of floats; 3.184 GB); a geometric isosurface extraction of a limestone karst core sample from Florida International University (5.8M vertices comprising 11.6M triangles; 0.266 GB); and an n-body dark matter “cosmic web” particle simulation from the Enzo team at National Center for Supercomputing Applications (1.07B particles; 12.2 GB).

As Fig. 2 shows, Galaxy achieves interactive rendering rates across node counts. We note increasing performance for the Skylake configuration as node count increases, though with sub-linear increase from 64 to 128 nodes likely due to an exhaustion of parallel work available. In the Haswell configuration, we attribute the relative performance degradation at higher node counts due to relative inefficiencies of the Mellanox interconnect versus the OPA interconnect: while both have the same

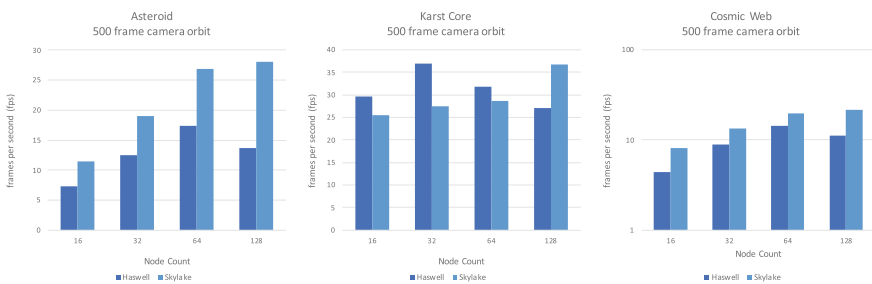


Fig. 2 Frames per second (fps) when rendering a 500 frame camera orbit for *Asteroid*, *Karst Core*, and *Cosmic Web* datasets (higher is better). Galaxy achieves interactive rendering rates for both Haswell and Skylake architectures

nominal bandwidth and latency, OPA can sustain higher messaging rates, particularly for the relatively small message sizes used by Galaxy, which enables the Skylakes on Stampede2 to sustain performance at higher node counts.

6 Conclusion

In this chapter, we have presented the `Galaxy Multiserver`, which provides multi-tenant distributed data filtering and analysis for a variety of in situ visualization scenarios. Galaxy can use ray tracing operations both for rendering and for data filtering, leveraging the same hardware-optimized components for multiple steps in a data analysis workflow. We have demonstrated that Galaxy provides interactive performance for distributed rendering tasks on current-generation hardware. Since Galaxy utilizes vendor-provided libraries for its internal tasks (currently the OSPRay and Embree libraries for Intel Xeon-optimized instructions), it can seamlessly benefit from vendor-provided improvements for current and future architectures. We plan to continue developing Galaxy's interfaces, both for direct use and for integration into third-party applications that already support ray tracing, such as ParaView and VisIt.

Acknowledgements This work was funded in part by a DOE ASCR program award, Dr. Laura Biven, program manager; US NSF award ACI-1339863; and an Intel Graphics and Visualization Institute of eXcellence award.

References

1. Abram, G., Navrátil, P., Grossett, P., Rogers, D., Ahrens, J.: Galaxy: asynchronous ray tracing for large high-fidelity visualization. In: 2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV), pp. 72–76 (2018)
2. Ahrens, J., Jourdain, S., O'Leary, P., Patchett, J., Rogers, D., Petersen, M.: An image-based approach to extreme scale in situ visualization and analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pp. 424–434. IEEE Press, Piscataway, NJ, USA (2014). <https://doi.org/10.1109/SC.2014.40>
3. Ayachit, U., Bauer, A., Geveci, B., O'Leary, P., Moreland, K., Fabian, N., Mauldin, J.: Paraview catalyst: enabling in situ data analysis and visualization. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, pp. 25–29 (2015)
4. Ayachit, U., Whitlock, B., Wolf, M., Loring, B., Geveci, B., Lonie, D., Bethel, E.W.: The sensei generic in situ interface. In: 2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), pp. 40–44. IEEE (2016)
5. Kuhlen, T., Pajarola, R., Zhou, K.: Parallel in situ coupling of simulation with a fully featured visualization system. In: Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV), vol. 10, pp. 101–109. Eurographics Association Aire-la-Ville, Switzerland (2011)
6. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: 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, pp. 15–24 (2008)

7. Park, H., Fussell, D., Navrátil, P.: Spray: Speculative ray scheduling for large data visualization. In: 2018 IEEE 8th Symposium on Large Data Analysis and Visualization (LDAV), pp. 77–86 (2018)
8. Pharr, M., Mark, W.R.: ispc: A SPMD compiler for high-performance CPU programming. In: 2012 Innovative Parallel Computing (InPar), pp. 1–13. IEEE (2012)
9. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit, 4th edn. Kitware (2006)