Projecting Performance Data Over Simulation Geometry Using SOSflow and ALPINE

Chad Wood
University of Oregon
Eugene, OR, United States
cdw@cs.uoregon.edu

Matthew Larsen
Lawrence Livermore National Laboratory
Livermore, CA
larsen30@llnl.gov

Alfredo Gimenez
Lawrence Livermore National Laboratory
Livermore, CA
gimenez1@llnl.gov

Cyrus Harrison
Lawrence Livermore National Laboratory
Livermore, CA
harrison37@llnl.gov

Todd Gamblin
Lawrence Livermore National Laboratory
Livermore, CA
gamblin2@llnl.gov

Allen Malony
University of Oregon
Eugene, OR
malony@cs.uoregon.edu

ABSTRACT

The performance of HPC simulation codes is often tied to their simulated domains; e.g., properties of the input decks, boundaries of the underlying meshes, and parallel decomposition of the simulation space. A variety of research efforts have demonstrated the utility of projecting performance data onto the simulation geometry to enable analysis of these kinds of performance problems. However, current methods to do so are largely ad-hoc and limited in terms of extensibility and scalability. Furthermore, few methods enable this projection online, resulting in large storage and processing requirements for offline analysis. We present a general, extensible, and scalable solution for in-situ (online) visualization of performance data projected onto the underlying geometry of simulation codes. Our solution employs the scalable observation system SOSflow with the in-situ visualization framework ALPINE to automatically extract simulation geometry and stream aggregated performance metrics to respective locations within the geometry at runtime. Our system decouples the resources and mechanisms to collect, aggregate, project, and visualize the resulting data, thus mitigating overhead and enabling online analysis at large scales. Furthermore, our method requires minimal user input and modification of existing code, enabling general and widespread adoption.

KEYWORDS

sos, sosflow, alpine, hpc, performance, visualization, in situ

1 INTRODUCTION

Projecting application and performance data onto the scientific domain allows for the behavior of a code to be perceived in terms of the organization of the work it is doing, rather than the organization of its source code. This perspective can be especially helpful [19] for domain scientists developing aspects of a simulation primarily for its scientific utility, though it can also be useful for any HPC developer engaged with the general maintenance requirements of a large and complicated codebase [18].

There have been practical challenges to providing these opportunities for insight. Extracting the spatial descriptions from an application traditionally has relied on hand-instrumenting codes to couple a simulation's geometry with some explicitly defined performance metrics. Performance tool wrappers and direct source-instrumentation need to be configurable so that users can disable their invasive presence during large production runs. Because it involves changes to the source code of an application, enabling or disabling the manual instrumentation of a code often involves full recompilation of a software stack. Insights gained by the domain projection are limited to what was selected a priori for contextualization with geometry.

Without an efficient runtime service providing an integrated context for multiple sources of performance information, it is difficult to combine performance observations across several components during a run. Further limiting the value of the entire exercise, performance data collected outside of a runtime service must wait to be correlated and projected over a simulation's geometry during post-mortem analysis. Projections that are produced offline cannot be used for application steering, online parameter tuning, or other runtime interactions that include a human in the feedback loop. Scalability for offline projections also becomes a concern, as the potentially large amount of performance data and simulation geometry produced and operated over in a massively parallel cluster now must be integrated and rendered either from a single point or within an entirely different allocation.

The overhead of manually instrumenting large complex codes to extract meaningful geometries for use in performance analysis, combined with the limited value of offline correlation of a fixed number of metrics, naturally limited the usage of scientific domain projections for gaining HPC workflow performance insights.

1.1 Research Contributions

This paper describes the use of SOSflow [20] and ALPINE to overcome many prior limitations to projecting performance into the scientific domain. The methods used to produce our results can be implemented in other frameworks, though SOSflow and ALPINE, discussed in detail in later sections, are generalized and intentionally engineered to deliver solutions of the type presented here. This research effort achieved the following:

- Eliminate the need to manually capture geometry for performance data projections of ALPINE-enabled workflows

- Provide online observation of performance data projected over evolving geometries and metrics
- Facilitate interactive selection of one or many performance metrics and rendering parameters, adding dynamism to projections
- Enable simultaneous online projections from a common data source
- In situ performance visualization architecture supporting both current and future-scale systems

2 RELATED WORK

Husain and Gimenez's work on Mitos [7] and MemAxes [6] is motivated similarly to ours. Mitos provides an integration API for combining information from multiple sources into a coherent memoized set for analysis and visualization, and MemAxes projects correlated information across domains to explore the origins of observed performance. SOSflow is being used in our research as an integration API, but takes a different optimization path by providing a general-purpose in situ (online) runtime.

Caliper by Boehme et al. [3] extracts performance data during execution in ways that serve a variety of uses, in much the same way our efforts here are oriented. Caliper's flexible data aggregation [4] model can be used to filter metrics in situ, allowing for tractable volumes of performance data to be made available for projections. Both ALPINE and Caliper provide direct services to users, also serving as integration points for user-configurable services at run time. Caliper is capable of deep introspection on the behavior of a program in execution, yet is able to be easily disabled for production runs that require no introspection and want to minimize instrumentation overhead. ALPINE allows for visualization filters to be compiled separately from a user's application and then introduced into, or removed from, an HPC code's visualization pipeline with a simple edit to that workflow's ALPINE configuration file. More tools like Caliper and ALPINE, featuring well-defined integration points, are essential for the wider availability of cross-domain performance understanding. SOSflow does not collect source-level performance metrics directly, but rather brings that data from tools like Caliper into a holistic online context with information from other libraries, performance tools, and perspectives.

BoxFish [8] also demonstrated the value of visualizing projections when interpreting performance data, adding a useful hierarchical data model for combining visualizations and interacting with data.

SOSflow's flexible model for multi-source online data collection and analysis provides performance exploration opportunities using both new and existing HPC tools.

3 SOSFLOW

SOSflow provides a lightweight, scalable, and programmable framework for observation, introspection, feedback, and control of HPC applications. The Scalable Observation System (SOS) performance model used by SOSflow allows a broad set of in situ (online) capabilities including remote method invocation, data analysis, and visualization. SOSflow can couple together multiple sources of data, such as application components and operating environment measures, with multiple software libraries and performance tools. These features combined to efficiently create holistic views of workflow performance at runtime, uniting node-local and distributed resources and perspectives. SOSflow can be used for a variety of purposes:

- Aggregation of application and performance data at runtime
- Providing holistic view of multi-component distributed scientific workflows
- Coordinating in situ operations with global analytics
- Synthesizing application and system metrics with scientific data for deeper performance understanding
- Extending the functionality of existing HPC codes using in situ resources
- Resource management, load balancing, online performance tuning, etc.

To better understand the role played by SOSflow, it is useful to examine its architecture. SOSflow is composed of four major components:

- **sosd** : Daemons
- **libsos** : Client Library
- **pub/sql** : Data
- **sosa** : Analytics & Feedback

These components work together to provide extensive runtime capabilities to developers, administrators, and application end-users. SOSflow runs within a user's allocation, and does not require elevated privileges for any of its features.

3.1 SOSflow Daemons

Online functionality of SOSflow is enabled by the presence of a user-space daemon. This daemon operates completely independently from any applications, and does not connect into or utilize any application data channels for SOSflow communications. The SOSflow daemons are launched from within a job script, before the user's applications are initialized. These daemons discover and communicate amongst each other across node boundaries within a user's allocation. When crossing node boundaries, SOSflow uses the machine's high-speed communication fabric. Inter-node communication may use either **MPI** or **EVPath** as needed, allowing for flexibility when configuring its deployment to various HPC environments.

The traditional deployment of SOSflow will have a single daemon instance running in situ for each node that a user's applications will be executing on. This daemon is called the **listener**. Additional resources can be allocated in support of the SOSflow runtime as-needed to support scaling and to minimize perturbation of application performance. One or more nodes are usually added to the user's allocation to host SOSflow **aggregator** daemons that combine the information that is being collected from the in situ daemons. These aggregator daemons are useful for providing holistic unified views at runtime, especially in service to online analytics modules. Because they have more work to do than the in situ listener daemons, and also are a useful place to host analytics modules, it is advisable to place aggregation targets on their own dedicated node[s], co-located with online analytics codes.

3.1.1 In Situ. Data coming from SOSflow clients moves into the in situ daemon across a light-weight local socket connection. Any

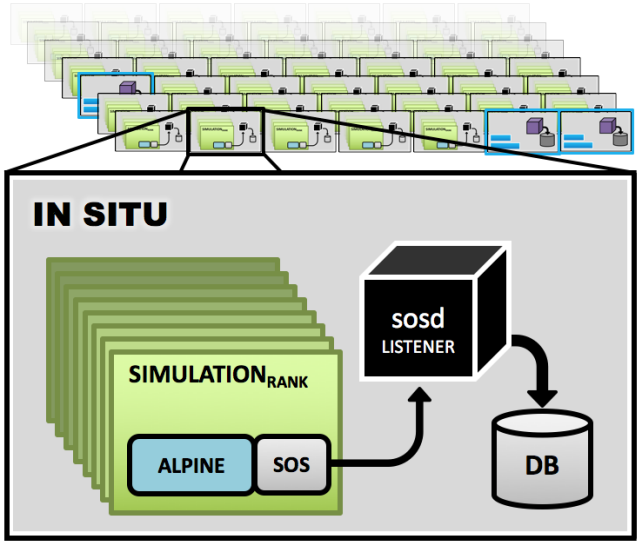

Figure 1: SOSflow's lightweight daemon runs on each node.

software that connects in to the SOSflow runtime can be thought of as a client. Clients connect only to the daemon that is running on their same node. No client connections are made across node boundaries, and no special permissions are required to use SOSflow, as the system considers the SOSflow runtime to be merely another part of a user's workflow.

The in situ listener daemon offers the complete functionality of the SOSflow runtime, including online query and delivery of results, feedback, or application steering messages. At startup, the daemon creates an in-memory data store with a file-based mirror in a user-defined location. Listeners asynchronously store all data that they receive into this store. The file-based mirror is ideal for offline analysis and archival. The local data store can be queried and updated via the SOSflow API, with all information moving over the daemon's socket, avoiding dependence on filesystem synchronization or centralized metadata services. Providing the full spectrum of data collected on node to clients and analytics modules on node allows for distributed online analytics processing. Analytics modules running in situ can observe a manageable data set, and then exchange small intermediate results amongst themselves in order to compute a final global view. SOSflow also supports running analytics at the aggregation points for direct query and analysis of global or enclave data, though it is potentially less scalable to perform centrally than in a distributed fashion, depending on the amount of data being processed by the system.

SOSflow's internal data processing utilizes unbounded asynchronous queues for all messaging, aggregation, and data storage. Pervasive design around asynchronous data movement allows for the SOSflow runtime to efficiently handle requests from clients and messaging between off-node daemons without incurring synchronization delays. Asynchronous in situ design allows the SOSflow runtime to scale out beyond the practical limits imposed by globally synchronous data movement patterns.

3.1.2 Aggregation Targets. A global perspective on application and system performance is often useful. SOSflow automatically migrates information it is given into one or more aggregation targets. This movement of information is transparent to users of SOS, requiring no additional work on their part. Aggregation targets are fully-functional instances of the SOSflow daemon, except that their principle data sources are distributed listener daemons rather than node-local clients. The aggregated data contains identical information as the in situ data stores, it just has more of it, and it is assembled into one location. The aggregate daemons are useful for performing online analysis or information visualization that needs to include information from multiple nodes.

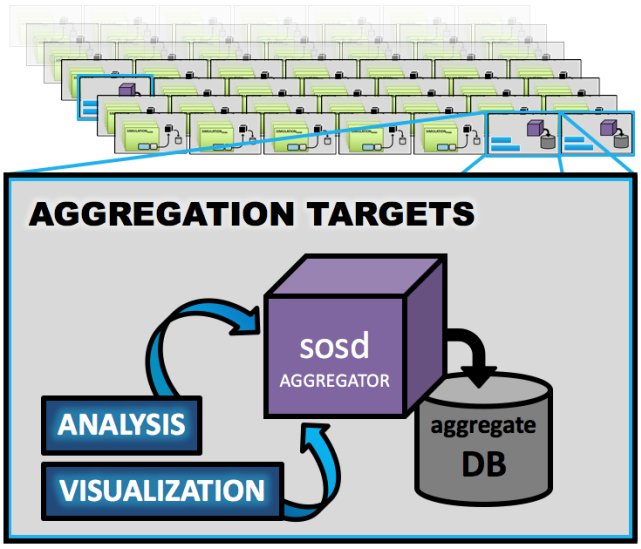

Figure 2: Co-located aggregation, analysis, and visualization.

SOSflow is not a publish-subscribe system in the traditional sense, but uses a more scalable push-and-pull model. Everything sent into the system will automatically migrate to aggregation points unless it is explicitly tagged as being node-only. Requests for information from SOSflow are ad hoc and the scope of the request is constrained by the location where the request is targeted: in situ queries are resolved against the in situ database, aggregate queries are resolved against the aggregate database. If tagged node-only information is potentially useful for offline analysis or archival, the in situ data stores can be collected at the end of a job script, and their contents can be filtered for that node-only information, which can be simply concatenated together with the aggregate database[s] into a complete image of all data. Each value published to SOSflow is tagged with a globally unique identifier (GUID). This allows SOSflow data from multiple sources to be mixed together while preserving its provenance and preventing data duplication or namespace collision.

3.2 SOSflow Client Library

Clients can directly interface with the SOSflow runtime system by calling a library of functions (libsos) through a standardized API.

Applications can also transparently become clients of SOS by utilizing libraries and performance tools which interact with SOSflow on their behalf. All communication between the SOSflow library and daemon are transparent to users. Users do not need to write any socket code or introduce any state or additional complexity to their own code.

Information sent through the libsos API is copied into internal data structures, and can be freed or destroyed by the user after the SOSflow API function returns. Data provided to the API is published up to the in situ daemon with an explicit API call, allowing developers to control the frequency of interactions with the runtime environment. It also allows the user to register callback functions that can be triggered and provided data by user-defined analytics function, creating an end-to-end system for both monitoring as well as feedback and control.

To maximize compatibility with extant HPC applications, the SOSflow client library is currently implemented in C99. The use of C99 allows the library to be linked in with a wide variety of HPC application codes, performance tools, and operating environments. There are various custom object types employed by the SOSflow API, and these custom types can add a layer of complexity when binding the full API to a language other than C or C++. SOSflow provides a solution to this challenge by offering a "Simple SOS" (ssos) wrapper around the full client library, exposing an API that uses no custom types. The ssos wrapper was used to build a native Python module for SOSflow. Users can directly interact with the SOSflow runtime environment from within Python scripts, acting both as a source for data, and also a consumer of online query results. HPC developers can capitalize on the ease of development provided by Python, using SOSflow to observe and react online to information from complex legacy applications and data models without requiring that those applications be redesigned to internally support online interactivity.

3.3 SOSflow Data

The primary concept around which SOSflow organizes information is the "publication handle" (pub). Pubs provide a private namespace where many types and quantities of information can be stored as a key/value pair. SOSflow automatically annotates values with a variety of metadata, including a GUID, timestamps, origin application, node id, etc. This metadata is available in the persistent data store for online query and analysis. SOSflow's metadata is useful for a variety of purposes:

- Performance analysis
- Provenance of captured values for detection of source-specific patterns of behavior, failing hardware, etc.
- Interpolating values contributed from multiple source applications or nodes
- Re-examining data after it has been gathered, but organizing the data by metrics other than those originally used when it was gathered

The full history of a value, including updates to that value, is maintained in the daemon's persistent data store. This allows for the changing state of an application or its environment to be explored at arbitrary points in its evolution. When a key is re-used to store some new information that has not yet been transmitted to the in situ daemon, the client library enqueues it up as a snapshot of that value, preserving all associated metadata alongside the historical value. The next time the client publishes to the daemon, current new values and all enqueued historical values are transmitted.

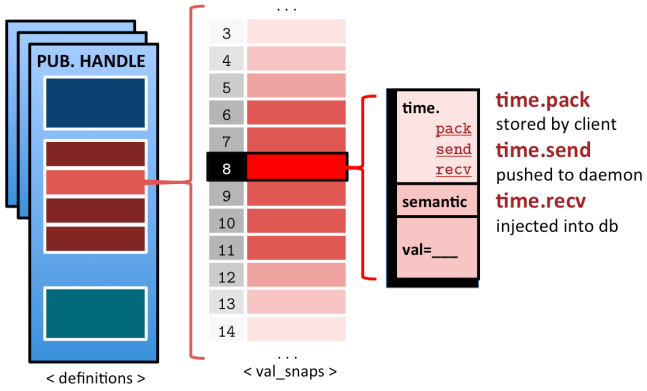

Figure 3: SOSflow retains the full history of every value.

SOSflow is built on a model of a global information space. Aggregate data stores are guaranteed to provide eventual consistency with the data stores of the in situ daemons that are targeting them. SOSflow's use of continuous but asynchronous movement of information through the runtime system does not allow for strict quality-of-service guarantees about the timeliness of information being available for analysis. This design constraint reflects the reality of future-scale HPC architectures and the need to eliminate dependence on synchronous behavior to correlate context. SOSflow conserves contextual metadata when values are added inside the client library. This metadata is used during aggregation and query resolution to compose the asynchronously-transported data according to its original synchronous creation. The vicissitudes of asynchronous data migration strategies at scale become entirely transparent to the user.

SOSflow does not require the use of a domain-specific language when pushing values into its API. Pubs are self-defining through use: When a new key is used to pack a value into a pub, the schema is automatically updated to reflect the name and the type of that value. When the schema of a pub changes, the changes are automatically announced to the in situ daemon the next time the client publishes data to it. Once processed and injected into SOSflow's data store, values and their metadata are accessible via standardized SQL queries. SOSflow's online resolution of SQL queries provides a high-degree of programmability and online adaptivity to users. SQL views are built into the data store that mask off the internal schemas and provide results organized intuitively for grouping by application rank, node, time series, etc.

SOSflow uses the ALPINE in situ visualization infrastructure described below to collect simulation geometry that it correlates with performance data.

4 ALPINE ASCENT

ALPINE is a project that aims to build an in situ visualization infrastructure and analysis targeting leading edge supercomputers.

ALPINE is part of the U.S. Department of Energy's Exascale Computing Project (ECP) [15], and the ALPINE effort is supported by multiple institutions. The goal of ALPINE is two fold. First, create a hybrid-parallel library (i.e., both distributed-memory and shared-memory parallel) that can be included in other visualization tools such as ParaView [2] and VisIt [5] thus creating an ecosystem where new hybrid-parallel algorithms are easily deployed into downstream tools. Second, create a flyweight in situ infrastructure that directly leverages the hybrid-parallel library. In this work, we directly interface with the ALPINE in situ infrastructure called Ascent [12].

Ascent is the descendant of Strawman [13], and Ascent is tightly-coupled with simulations, i.e. it shares the same node resources as the simulation. While Strawman's goal was to bootstrap in situ visualization research, the ALPINE Ascent in situ infrastructure is intended for production. Ascent includes include three physics proxy-applications out of the box to immediately provide the infrastructure and algorithms a representative set of mesh data to consume. Ascent is already integrated into several physics simulations to perform traditional visualization and analysis, and we chose to embed an SOSflow client into Ascent to eliminate the need for additional manual integration of SOSflow with Ascent-equipped simulations. Ascent uses the Conduit [10] data exchange library to marshal mesh data from simulations into Ascent. Conduit provides a flexible hierarchical model for describing mesh data, using a simple set of conventions for describing meshes including structured, unstructured, and higher order element meshes [11]. Once the simulation describes the mesh data, it publishes the data into Ascent for visualization purposes. Ascent relays the mesh data to SOSflow in the manner described below. In addition to the mesh data, we can easily add performance data that is associated with each MPI rank. Coupling the performance data with the mesh geometry provides a natural way to generate an aggregate data set to visualize the performance data mapped to the spatial region each MPI rank is responsible for.

Ascent includes Flow, a simple dataflow library based on the Python dataflow library within VisIt, to control the execution of visualization filters. The input to Flow is the simulation mesh data, and Ascent adds visualization filters (e.g., contours and thresholding) to create visualizations. Everything within Flow is a filter that can have multiple inputs and a single output of generic types. The flexibility of Flow allows for user defined filters, compiled outside of Ascent, to be easily inserted into the dataflow, and when the dataflow network executes, custom filters have access to all of the simulation mesh data published to Ascent. We leverage the flexibility of Flow to create an SOSflow filter that is inserted at runtime. The SOSflow filter uses the data published by the simulation to extract the spatial extents being operated over by each MPI rank along, with any performance data provided. Next, we publish that data to SOSflow, and then Ascent's visualization filters execute as usual.

5 EXPERIMENTS

5.1 Evaluation Platform

All results were obtained by running online queries against the SOSflow runtime's aggregation targets (Figure 2) using SOSflow's built-in Python API. The results of these queries were used to create Vtk [17] geometry files. These files were used as input for the VisIt visualization tool, which we invoked from within the allocation to interactively explore the performance projections.

5.2 Experiment Setup

The experiments performed had the following purposes:

- **Validation** : Demonstrate the coupling of SOSflow with ALPINE and its ability to extract geometry from simulations transparently.
- **Introspection** : Examine the overhead incurred by including the SOSflow geometry extraction filter in an ALPINE Ascent visualization pipeline.

ALPINE's Ascent library was used to build a filter module outfitted with SOSflow, and this filter was used for online geometry extraction. ALPINE's JSON configuration file describing the connectivity of the in situ visualization pipeline was modified to insert the SOSflow-equipped geometry extraction filter. The SOSflow implementation used to conduct these experiments is general-purpose and was not tailored to the specific deployment environment or the simulations observed. The study was conducted on two machines, the details of which are included here —

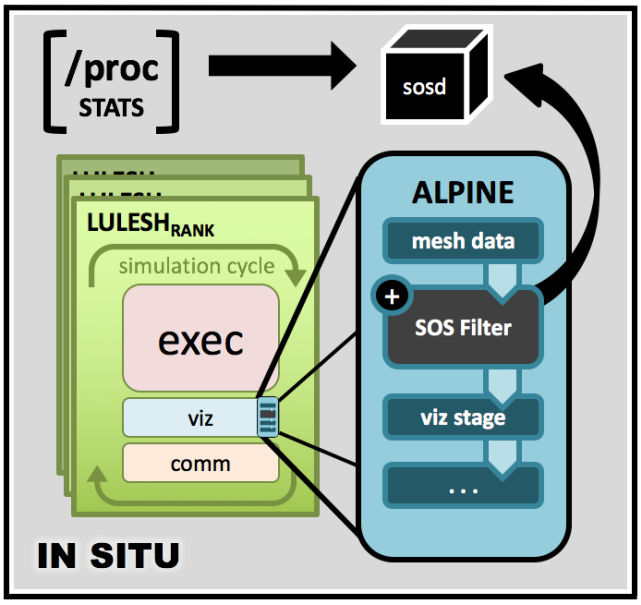

Figure 4: SOSflow collects runtime information to project over the simulation geometry.

(1) **Quartz** : A 2,634-node Penguin supercomputer at Lawrence Livermore National Laboratory (LLNL). Intel Xeon E5-2695 processors provide 36 cores/node. Each node offers 128 GB of memory and nodes are connected via Intel OmniPath.
(2) **Catalyst** : A Cray CS300 supercomputer at LLNL. Each of the 324 nodes is outfitted with 128 GB of memory and 2x Intel Xeon E5-2695v2 2.40 GHz 12-core CPUs. Catalyst nodes transport data to each other using a QLogic InfiniBand QDR interconnect.

The following simulated workflows were used —

(1) **KRIPKE** [9] : A 3D deterministic neutron transport proxy application that implements a distributed-memory parallel sweep solver over a rectilinear mesh. At any given simulation cycle, there are simultaneous sweeps along a set of discrete directions to calculate angular fluxes. This results in a MPI communication pattern where ranks receive asynchronous requests from other ranks for each discrete direction.
(2) **LULESH** [1] : A 3D Lagrangian shock hydrodynamics proxy application that models Sedov blast test problem over a curvilinear mesh. As the simulation progresses, hexahedral elements deform to more accurately capture the problem state.

5.3 Overview of Processing Steps

The SOSflow runtime provided a modular filter for the ALPINE in situ visualization framework. This filter was enabled for the simulation workflow at runtime to allow for the capture of evolving geometric details as the simulation progressed. The SOSflow runtime daemon automatically contextualized the geometry it received alongside the changing application performance metrics. SOSflow's API for Python was used to extract both geometry information and correlated performance metrics from the SOSflow runtime. This data set was used to generate sequences of input files to the VisIt scientific data visualization tool corresponding to the cycle of a the distributed simulation.

Each input file contained the geometric extents of every simulation rank, the portion of the simulated space that each part of the application was working within. Alongside that volumetric descriptions for that cycle, SOSflow integrated attribute dictionaries of all plottable numeric values it was provided during that cycle, grouped by simulation rank. Performance metrics could then be interactively selected and combined in VisIt with customizable plots, presenting an application rank's state and activity incident to its simulation effort, projected over the relevant spatial extent.

5.4 Evaluation of Geometry Extraction

Our experiments were validated by comparing aggregated data to data manually captured at the source during test runs. Furthermore, geometry aggregated by ALPINE's Ascent SOSflow filter was rendered and visually compared with other visualizations of the simulation. Projections were inspected to observe the simulation's expected deforming of geometry (LULESH) or algorithm-dependent workload imbalances (KRIPKE). Performance metrics can be correlated in SQL queries to the correct geometric regions by various redundant means such as pub handle GUID, origin PID or MPI rank, simulation cycle, host node name, SOSflow publish frame, and value creation timestamps. Aggregated performance metrics projected over the simulation regions were compared to metrics reported locally, and required to be identical for each region and simulation cycle.

5.5 Evaluation of Overhead

Millisecond-resolution timers were added to the per-cycle *execute* method of the SOSflow Alpine geometry extraction filter. Each rank tracked the amount of time it spent extracting its geometry, packing the geometry into an SOSflow pub handle, and transmitting it to the runtime daemon. Every cycle's individual time cost was computed and transmitted to SOSflow, as well as a running total of the time that Alpine had spent in the SOSflow filter. From a region outside the timers, the timer values were packed into the same SOSflow publication handle used for the geometric data. Timer values were transmitted at the end of the following cycle, alongside that cycle's geometry. The additional transmission cost of these two timer values once per simulation cycle had no perceivable impact on the performance they were measuring.

6 RESULTS

Geometry was successfully extracted (Figures 5, 6, 7, and 8) with minimal overhead from simulations run at a variety of scales from 2 to 33 nodes. The side-by-side introspection of the behavior of

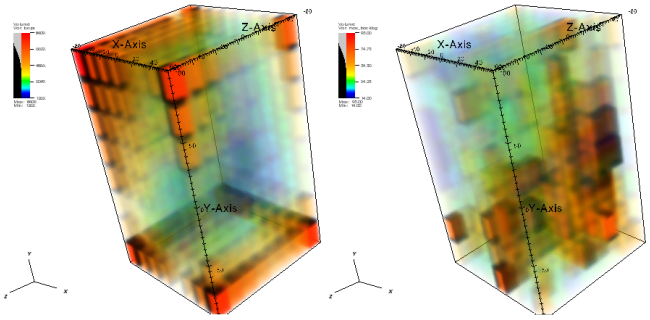

Figure 5: Loops (left) and maximum backlog (right) from one cycle of 512 KRIPKE ranks distributed to 32 nodes.

KRIPKE (Figure 5) are a good example of the value this system provides to developers. The amount of work loops and the backlog of requests for computation are correlated negatively, with ranks operating in the center of the simulation space getting through less loops of work per cycle, since they are required to service data requests in more directions than the ranks simulating the corners regions. The directionality of energy waves moving through he simulated space can also be observed, with more work piling up where multiple waves are converging. A developer can quickly assess the behavior of their distributed algorithm by checking for hot-spots and workload imbalances in the space being simulated.

6.1 Geometry Extraction and Performance Data Projection

Aggregated simulation geometry was a precise match with the geometry manually recorded within applications, across all runs. After aggregation and performance data projection, geometry from all simulation ranks combined to create a contiguous space without gaps or overlapping regions, representative of the simulated space subdivided by MPI rank.

6.2 Overhead

The inclusion of the ALPINE Ascent filter module for SOSflow had no observable impact on overall application execution time, being significantly less than variance observed between experimental runs both with and without the filter. The filter module is executed

at the end of each simulation cycle, from the first iteration through to the simulation conclusion. Manual instrumentation was added to the SOSflow filter to measure the time spent inside the filter's *execute* method, where all simulation geometry and performance metrics were gathered for our study.

When gathering *only the simulation geometry*, filter execution never exceeded 2ms per simulation cycle. We collected performance information for our projections by reading from the /proc/[pid] files of each rank. These readings were made from within the SOSflow filter, and published to SOSflow alongside the collected geometry. Collecting 31 system metrics and application counters added additional overhead, but the filter time but did not exceed 4ms for any of the projections shown in this paper. The filter's execution time was logged as a performance metric alongside the other in situ performance data, and is visualized for LULESH in Figure 7.

7 CONCLUSION

Services from both SOSflow and ALPINE were successfully integrated to provide a scalable in situ (online) geometry extraction and performance data projection capability.

7.1 Future Work

Workflows that use the ALPINE framework but have complex irregular meshes, feature overlapping "halo regions", or that operate over non-continuous regions of space within a single process, may require additional effort to extract geometry from, depending on the organization of spatial descriptions they employ. ALPINE uses the Vtk-m [16] library for its operations over simulation mesh data. The addition of a general convex hull algorithm to Vtk-m will simplify the task of uniformly describing any spatial extent[s] being operated on by a process using ALPINE for its visualization pipeline.

The VisIt UI can be extended to support additional interactivity with the SOSflow runtime. UI elements to submit custom SQL queries to SOSflow would enhance the online data exploration utility of VisIt. SOSflow's interactive code steering mechanisms allow for feedback messages and payloads to be delivered to subscribing applications at runtime. With some basic additions to the VisIt UI, these mechanisms could be triggered by a VisIt user based on what they observe in the performance projections, sending feedback to targeted workflow components from within the VisIt UI.

While the geometry capture and performance data projection in this initial work has a scalable in situ design, the final rendering of the performance data into an image takes place on a single node. Future iterations of this performance visualization work will explore the use of in situ visualization techniques currently employed to render scientific data from simulations [14]. These emerging in situ rendering technologies will allow for live views of performance data projected over simulation geometry at the furthest extreme scales to which our simulations are being pressed.

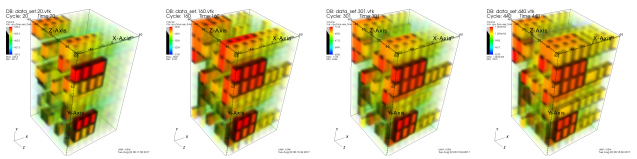

Figure 6: Cumulative user CPU ticks during 440 cycles of 512 KRIPKE ranks on 32 nodes.

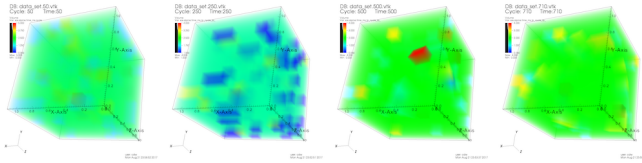

Figure 7: Filter execution (1-4ms) over 710 LULESH cycles.

ACKNOWLEDGMENTS

The research report was supported by a grant (DE-SC0012381) from the Department of Energy, Scientific Data Management, Analytics, and Visualization (SDMAV), for "Performance Understanding and Analysis for Exascale Data Management Workflows."

Part of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-737874).

REFERENCES

[1] [n. d.]. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.

[2] James Ahrens, Berk Geveci, and Charles Law. 2005. Paraview: An end-user tool for large data visualization. *The Visualization Handbook* 717 (2005).

[3] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for*. IEEE, 550–560.

[4] David Böhme, David Beckingsdale, and Martin Schulz. 2017. Flexible Data Aggregation for Performance Profiling. *IEEE Cluster* (2017).

[5] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. 2012. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. CRC Press/Francis–Taylor Group, 357–372.

[6] Adolfo Alfredo Gimenez, Todd Gamblin, Ilir Jusufi, Abhinav Bhatele, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. 2017. MemAxes: Visualization and Analytics for Characterizing Complex Memory Performance Behaviors. *IEEE Transactions on Visualization and Computer Graphics* (2017).

[7] Benafsh Husain, Alfredo Giménez, Joshua A Levine, Todd Gamblin, and Peer-Timo Bremer. 2015. Relating memory performance data to application domain data using an integration API. In *Proceedings of the 2nd Workshop on Visual Performance Analysis*. ACM, 5.

[8] Katherine E Isaacs, Aaditya G Landge, Todd Gamblin, Peer-Timo Bremer, Valerio Pascucci, and Bernd Hamann. 2012. Exploring performance data with boxfish. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 1380–1381.

[9] AJ Kunen, TS Bailey, and PN Brown. 2015. *KRIPKE-a massively parallel transport mini-app.* Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[10] Lawrence Livermore National Laboratory. 2017. Conduit: Simplified Data Exchange for HPC Simulations. (2017). https://software.llnl.gov/conduit/

[11] Lawrence Livermore National Laboratory. 2017. Conduit: Simplified Data Exchange for HPC Simulations - Conduit Blueprint. (2017). https://software.llnl.gov/conduit/blueprint.html

[12] Matthew Larsen, James Aherns, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. 2017. The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman. In *Proceedings of the In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization Workshop (ISAV2017)*. ACM, New York, NY, USA.

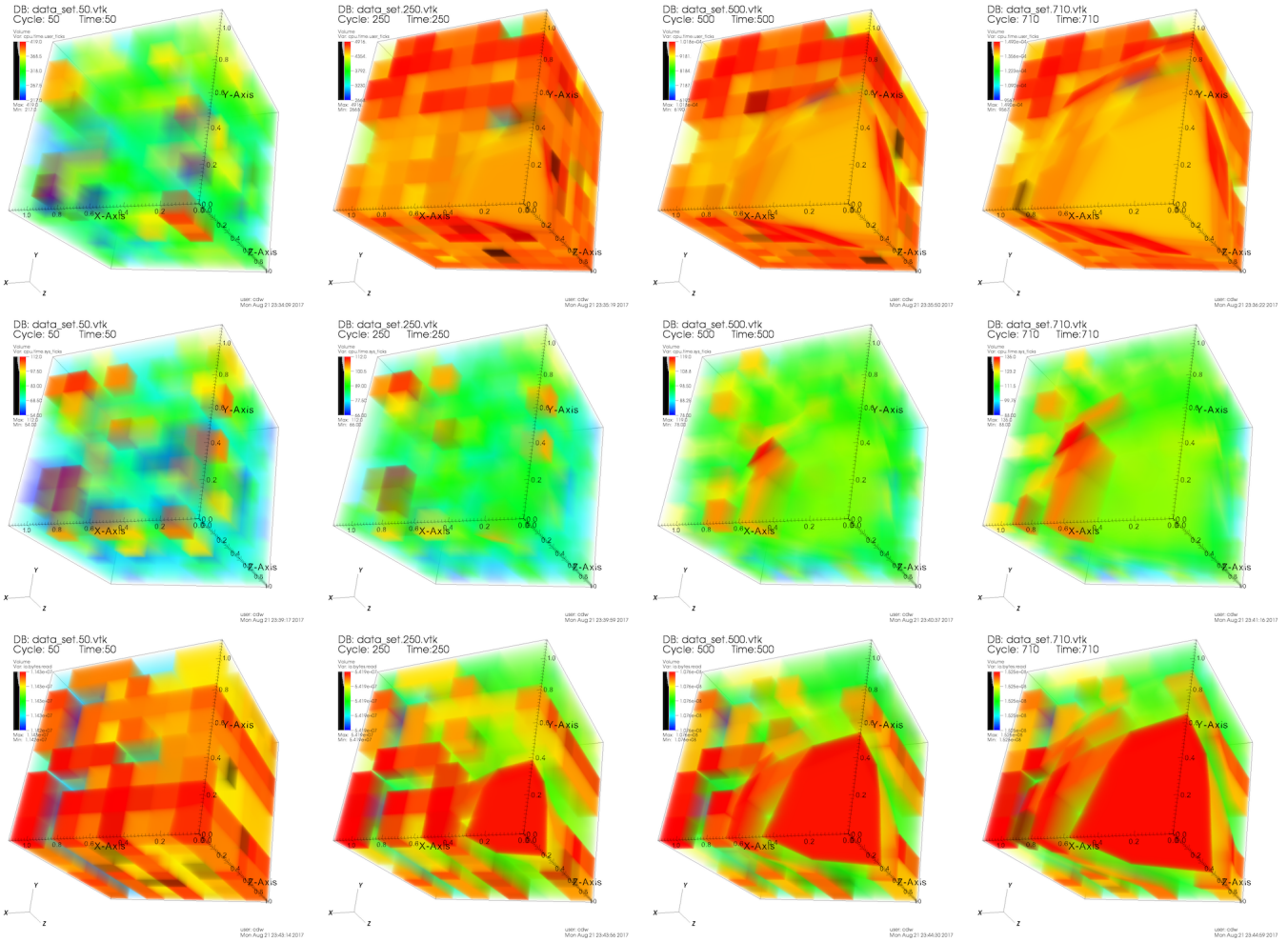

Figure 8: Many metrics can be projected from one run. Here we see (top to bottom) user CPU ticks, system CPU ticks, and bytes read during 710 cycles of 512 LULESH ranks distributed across 32 nodes.

[13] Matthew Larsen, Eric Brugger, Hank Childs, Jim Eliot, Kevin Griffin, and Cyrus Harrison. 2015. Strawman: A Batch In Situ Visualization and Analysis Infrastructure for Multi-Physics Simulation Codes. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV2015)*. ACM, New York, NY, USA, 30–35. https://doi.org/10.1145/2828612.2828625

[14] Matthew Larsen, Cyrus Harrison, James Kress, David Pugmire, Jeremy S Meredith, and Hank Childs. 2016. Performance modeling of in situ rendering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 24.

[15] Paul Messina. 2017. The Exascale Computing Project. *Computing in Science & Engineering* 19, 3 (2017), 63–67.

[16] Kenneth Moreland, Christopher Sewell, William Usher, Li-ta Lo, Jeremy Meredith, David Pugmire, James Kress, Hendrik Schroots, Kwan-Liu Ma, Hank Childs, et al. 2016. Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE computer graphics and applications* 36, 3 (2016), 48–58.

[17] Will J Schroeder, Bill Lorensen, and Ken Martin. 2004. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware.

[18] Martin Schulz, Abhinav Bhatele, David Böhme, Peer-Timo Bremer, Todd Gamblin, Alfredo Gimenez, and Kate Isaacs. 2015. A Flexible Data Model to Support Multi-domain Performance Analysis. In *Tools for High Performance Computing 2014*. Springer, 211–229.

[19] Martin Schulz, Joshua A Levine, Peer-Timo Bremer, Todd Gamblin, and Valerio Pascucci. 2011. Interpreting performance data across intuitive domains. In *Parallel Processing (ICPP), 2011 International Conference on*. IEEE, 206–215.

[20] Chad Wood, Sudhanshu Sane, Daniel Ellsworth, Alfredo Gimenez, Kevin Huck, Todd Gamblin, and Allen Malony. 2016. A scalable observation system for introspection and in situ analytics. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*. IEEE Press, 42–49.