Performance Analysis, Design Considerations, and Applications of Extreme-scale *In Situ* Infrastructures

Utkarsh Ayachit[†], Andrew Bauer[†], Earl P. N. Duque[§], Greg Eisenhauer[¶], Nicola Ferrier[‡], Junmin Gu[*],
Kenneth E. Jansen[‖], Burlen Loring[*], Zarija Lukić[*], Suresh Menon[¶], Dmitriy Morozov[*], Patrick O'Leary[†],
Reetesh Ranjan[¶], Michel Rasquin[**], Christopher P. Stone[††], Venkat Vishwanath[‡], Gunther H. Weber[*], Brad Whitlock[§],
Matthew Wolf[¶], K. John Wu[*], and E. Wes Bethel[*]

[*] *Lawrence Berkeley National Laboratory, email:* `ewbethel@lbl.gov`. [†] *Kitware, Inc.*
[‡] *Argonne National Laboratory.* [§] *Intelligent Light.* [¶] *Georgia Tech.* [‖] *U. Colorado Boulder.*
[**] *Cenaero and U. Colorado Boulder.* [††] *Computational Science and Engineering, LLC.*

***Abstract* – A key trend facing extreme-scale computational science is the widening gap between computational and I/O rates, and the challenge that follows is how to best gain insight from simulation data when it is increasingly impractical to save it to persistent storage for subsequent visual exploration and analysis. One approach to this challenge is centered around the idea of *in situ* processing, where visualization and analysis processing is performed while data is still resident in memory. This paper examines several key design and performance issues related to the idea of *in situ* processing at extreme scale on modern platforms: scalability, overhead, performance measurement and analysis, comparison and contrast with a traditional *post hoc* approach, and interfacing with simulation codes. We illustrate these principles in practice with studies, conducted on large-scale HPC platforms, that include a miniapplication and multiple science application codes, one of which demonstrates *in situ* methods in use at greater than 1M-way concurrency.**

1 Introduction

As Hamming observed in 1962, the purpose of computing is insight, not numbers [1]. However, the widening gap between computational capacity and I/O capacity results in an increasingly challenging scenario for gaining insight. As the amount of data computed increases, less and less of it is actually stored and analyzed. It is this concern, well documented by experts in the field (c.f., [2]), that motivates this work.

The focus of this paper is on examining issues related to solving that problem: how to enable deriving insight despite the diminishing opportunity to do so. So-called *in situ* methods, those that perform analysis and visualization on computed data while still resident in memory, have shown promise for some time, and are becoming an increasingly important part of the computational landscape as the FLOPS-to-I/O gap continues to widen. Recent work in this space includes examples of *in situ* methods being applied to various computational science problems, as well as the emergence of production-quality *in situ* software infrastructure.

In this paper, we examine and provide insight into key questions related to the design, use, and application of *in situ* methods and infrastructure. Given that computational science applications tend to be "memory hungry" and strive to achieve the best possible performance (runtime, use of system resources like memory), how much overhead is associated with use of *in situ* methods and infrastructure? How do these characteristics change over varying levels of concurrency? Is there variation in these characteristics across platforms? Is it possible to achieve some level of portability, where a given *in situ* method might be used in multiple *in situ* infrastructures, or, conversely, might it be possible for a given computational science application to make use of any number of *in situ* infrastructures with little or no code modification?

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

Our approach to answering these performance-related questions is to conduct analysis studies that use a lightweight miniapplication, multiple *in situ* infrastructures, and multiple *in situ* methods of varying computational complexity. We evaluate the performance of these codes on multiple contemporary HPC platforms and at varying levels of concurrency, taking performance measurements that provide insights into questions related to their runtime and memory overhead. Next, we extend these studies by applying the same methodology with multiple contemporary computational science codes run at high concurrency on multiple contemporary HPC platforms. The concurrency levels range from routine, from thousands to tens of thousands cores, to the extreme, where one application is run at million-way concurrency.

For the portability question, our approach centers around the idea of a lightweight, generic data interface. This interface provides the ability to bridge between simulation and *in situ* infrastructure or methods in a way that supports highly efficient, zero-copy operation. We show that this approach results in negligible performance overhead even at high concurrency, and has the potential for broad applicability.

The contributions of this paper are as follows. We present a comprehensive study showing the cost of *in situ* at scale on modern HPC platforms. For this study, we use a carefully designed miniapplication coupled with multiple *in situ* infrastructures invoking multiple *in situ* methods. The choice of miniapplication and *in situ* methods reflects design and execution patterns representative of common scientific workloads. We present a new, generic *in situ* interface, which makes it possible to instrument a simulation code once and then have it make use of any number of *in situ* infrastructures. Additionally, developers of *in situ* methods may write them once, and reasonably expect them to run in any number of *in situ* infrastructures. Finally, we demonstrate these methods and designs with three different science applications run at scale on current extreme-scale computational platforms. One of these examples is run at greater than 1M-way concurrency.

2 Background and Previous Work

2.1 Terminology and Concepts

For the sake of brevity, we use the phrase *scientific data management, analysis/analytics, and visualization*, and the acronym *SDMAV*, to refer to any number of data-centric operations that could include visualization; analysis; and data processing operations like transformations, compression, subsetting, indexing.

Traditionally, the approach for performing SDMAV processing is a *post hoc* process, whereby simulation output is first written to persistent storage. Then, later, the SDMAV software will read the simulation output from persistent storage then perform its task.

The opposite of *post hoc* is a process whereby SDMAV processing happens without round-trip transit to persistent storage. There are many different ways to perform this type of operation. *In situ* approaches perform SDMAV operations on data while it is still resident in the same physical memory used by the simulation. *In transit* approaches involve

some form of data movement from simulation memory to some other location, where it is then subject to SDMAV processing. "Some other location" could mean other nodes in the same system shared with the simulation, or it could mean a completely different platform. Over time, the phrase "*in situ* processing" has come to be generally accepted as an umbrella term that refers to both kinds of processing, both *in situ* and in transit.

We draw the distinction between *in situ* methods and *in situ* infrastructure. *In situ* methods are algorithms that perform some type of SDMAV processing, such as a method for statistical analysis or a method for visualization. An *in situ* infrastructure is more akin to a framework in which an *in situ* method would exist and run. Typically, but not always, the simulation code would interface to the method via a framework, which is then responsible for activities (if needed) like marshaling data movement, provisioning external resources, and so forth. This importance of this distinction will become apparent in later sections of this paper.

There are many more ways, or dimensions, that one can think about *in situ* processing that go well beyond *in situ* vs. in transit. These additional ways of thinking about the *in situ* space are summarized in a 2016 EuroVis STAR report [3], and reflect the thinking of a rather large group of SDMAV researchers [4].

2.2 Previous Work

2.2.1 Early *In Situ* Implementations

The idea of *in situ* processing is not new, though it has received renewed interest in recent years due to broad recognition of the widening gap between our ability to compute data and our ability to analyze it using traditional *post hoc* approaches [2].

The idea of generating images without first writing data to persistent storage is as old as the field of computer graphics itself. In what may be the earliest known and documented example, Zajac, 1964 [5] computes the orbital path of two bodies and generates movie frames on-the-fly through a direct-to-film process. The NCAR Graphics Library [6], originating in the 1960s, may be some of the earliest production-quality "*in situ* infrastructure and methods", and continues to be developed and used by a world-wide community today. It consists of a set of subroutine-callable methods for generating images/plots of scientific data. The NCAR Graphics Library has been widely utilized for both *in situ* and *post hoc* use cases.

2.2.2 Co-processing and Computational Steering

In the 1990s, the term "co-processing" was used to describe something very similar to what we refer to today as *in situ* processing. In a survey work in this space, Heiland and Baker, 1998 [7], referred to such technology as "co-processing systems". That report focused on systems/methods that support interactive computation, computational monitoring and steering. The systems they surveyed—CUMULVS [8], pV3 [9], AVS [10], and others—each have some visual data exploration and analysis dimension. A year later, Mulder, et al., 1999 [11] performed a similar survey that included several additional systems. These studies were useful in terms of providing an inventory of systems and their capabilities, but there was no concerted effort during this period to study the scalability of these systems nor their overhead they add to parallel simulation codes.

More recent work targets code development, execution, steering, and SDMAV processing capabilities into a single environment. The Cactus Code framework [12] consists of infrastructure for building codes (the "flesh") and then add-on components (the "thorns") that provide specific types of functionality. Cactus provides *in situ* visualization and analysis capability through this thorn mechanism. For example, one can view a Cactus-generated *in situ* visualization of simulation results while it is running by pointing a browser at a URL that is specific and unique to that simulation run. Similarly, SCIRun [13] is a problem solving environment for interactive construction and *in situ* steering of simulations.

2.2.3 Contemporary *In Situ* Frameworks

In the present day, there are four production-quality *in situ* infrastructures, GLEAN, Catalyst, Libsim, ADIOS, which are described below. In brief, these four all run at scale on modern HPC platforms, and are actively developed and supported efforts.

ParaView Catalyst [14] (aka Catalyst) is an *in situ* analysis and visualization library that enables using ParaView's visualization capabilities in *in situ* workflows. Applications can use Catalyst to execute complex analysis pipelines in step with the simulation, as well as connecting with the ParaView GUI for live, interactive visualization. To minimize memory footprint, Catalyst libraries are available in various flavors, called Editions [15], that only enable components of ParaView used in the analysis pipelines.

Libsim [16, 17] is a library that makes available the full complement of features from VisIt so they may be used *in situ*. Libsim enables VisIt to connect interactively to running simulations for live exploration. Libsim can also be used directly to set up visualizations or it can use VisIt session files, which are XML files saved from the VisIt GUI, which can specify more complex visualizations. Once visualizations are set up, Libsim can save images for movie-making or it can save reduced-size data extracts for *post hoc* analysis.

ADIOS [18, 19] is an adaptive I/O service that is designed to allow applications to easily change between different I/O service providers. Only a tweak to the input parameters is needed to swap methods. This design allows for rapid conversion of *post hoc* analysis pipelines to *in situ*, in transit, or hybrid solutions by using one of the memory-to-memory "staging" methods, such as FlexPath or DataSpaces. The FlexPath transport used in this effort can support same-node, multi-node, or even multi-machine deployment configurations. Unlike Catalyst and Libsim, ADIOS does not include any of the analytics functionality itself; it marshals the memory and metadata to make such code self-describing and adaptable to new situations. As such, it can partner effectively with Catalyst, Libsim, and other analytics infrastructures to provide whatever tools the scientist currently needs.

GLEAN [20] is a flexible and extensible framework that takes application, analysis, and system characteristics into account to facilitate simulation-time data analysis and I/O acceleration. The GLEAN infrastructure hides significant details from the end user, while at the same time providing a flexible interface to the fastest path for their data and analysis needs and, in the end, scientific insight. It provides an infrastructure for accelerating I/O, interfacing to running simulations for in transit analysis, and/or an interface for *in situ* analysis with zero or minimal modifications to the existing application code base.

These contemporary *in situ* infrastructures share key traits with historical systems for "co-processing" and "computational steering". First, they all have an architecture that enables SDMAV processing but without writing data to persistent storage. In many cases, the frameworks do support writing data to persistent storage, but doing so is not a prerequisite for performing *in situ* SDMAV operations. Second, they all are "frameworks" in which individual "methods" are executed, including user-written methods. This architecture has proven useful in that it is modular, to support expansion and growth of new methods.

In this paper, we are focusing on the *in situ* infrastructure, with an eye towards gaining insight about key questions facing the community as we move forward into the exascale regime, which is characterized by increasing node-level and systemwide concurrency, shrinking per-core memory, and a widening gap between computational and I/O rates. These key questions focus on understanding the cost of using *in situ* systems in terms of performance and resource utilization, portability, scalability, and studies that include use at scale across multiple science applications.

2.2.4 Overcoming the Limitations of *In Situ* Approaches

While the idea of performing SDMAV processing *in situ* to avoid the cost of performing I/O is an attractive one, it does come with some limitations. Specifically, one typically needs to set, *a priori*, the parameters

for the SDMAV operation. If performing visualization, for example, then one would need to set the camera position, isocontouring level, and so forth before running the simulation code. If those parameters were set in a way that did not produce satisfying results, the simulation would need to be rerun with new parameters for the *in situ* SDMAV methods.

There is a substantial amount of prior work in this space going back around two decades, all of which centers around the idea of computing "explorable data products" that are much smaller than the full-resolution data, and that support varying degrees of *post hoc* interactive exploration.

Globus, 1995 [21] proposed a model where data extracts could be generated in the simulation run to reduce the size of output, then later processed to generate visuals. More recently, Ye et al., 2013 [22] focused on facilitating flow-field visualization in an *in situ* setting, where *post hoc* interactions focus on changing viewpoint, doing block cutaways, or changing the lighting or color transfer function. Ahrens et al., 2014 [23] explore extracting many images and creating animations using the Cinema system.

While this research thrust of overcoming the limits of *in situ* methods is certainly important in terms of usability, this set of topics is not the primary focus of this paper. Methods that produce "explorable extracts" will be run *in situ*, most likely using one of the infrastructures we study (§2.2.3), and future work in this area will be shaped, in part, by an understanding of the issues we study in this paper. Related, there has been little or no work at all on temporal *in situ* analysis. To that end, one of the *in situ* analysis methods we employ in our performance studies uses temporal analysis. This method, an autocorrelation calculation, performs a computation over time. It is described later in §3.3 and its performance studied in §4.1.

2.2.5 Simplified *In Situ* Interfaces

Ours is not the first effort to look at a simplified interface to *in situ* frameworks. Recent examples include Damaris/Viz [24], *Free*processing [25] and Strawman [26]. Damaris/Viz's API has sharing semantics for arrays to be used by both the simulation and the *in situ* application safely. The allocation is done through Damaris/Viz and works most efficiently when double-buffering is used when updating the simulation's data structures during time-stepping. *Free*processing has the potential to completely avoid instrumenting a simulation code while enabling *in situ* computation. This is done by intercepting the results being written to disk and using that to construct the grids and fields. This has the potential for multiple data copies though as the simulation may make an initial data copy to prepare it for a specific file format and then another data copy from the file format to the *in situ* processing engine. Strawman supports Cartesian, rectilinear and unstructured grids and uses Conduit's [27] data model. It supports zero-copy arrays but requires a matching array layout.

Our work differs in that we are focusing on implementations that are computationally efficient and have a low-as-possible memory footprint.

Our approach to this issue (§3.2) offers some unique advantages and capabilities. We leverage the VTK data model, which simplifies the process of interfacing different infrastructures together while maintaining computational efficiency. The sophisticated data model allows *in situ* infrastructures (§2.2.3) to be chained together in interesting and desirable ways instead of each acting as individual data sinks. One of our contributions is a detailed study that examines the overhead of such an interface in terms of performance and memory footprint.

3 Implementation

3.1 Design Principles and Objective

The long-term goal of our work is to characterize the building blocks to design and refactor analysis codes on diverse *in situ* infrastructures as well as on a wide-variety of systems for diverse computational science simulations. Doing so will enable analysis algorithms to fully exploit the underlying concurrency and heterogeneity of the system. To achieve this objective, one needs to consider the parallel algorithmic patterns for our target science analysis; the implementation patterns to extract high-level of concurrency on the underlying hardware; and the *in situ* execution patterns to scale these analysis on the *in situ* infrastructures. Parallel algorithmic patterns define high-level abstractions to exploit concurrency in analysis computation for execution on a parallel machine such as structured and unstructured grids, spectral methods, particle methods, Monte Carlo methods, and graph traversal, as well as temporal dimensions of these. Implementation patterns help to realize algorithmic patterns via parallel software constructs. Common implementation patterns include SPMD, Map/Reduce, Master/Slave, Bulk Synchronous Parallel, Fork-Joins, and Task Lists. Execution patterns expose the underlying *in situ* infrastructure for *in situ* analysis. The goal is to understand the functional decomposition of the analysis and map it on to the execution pattern for improved performance. To realize this vision, we have designed an initial set of miniapplications and analyses to help us understand this spectrum.

3.2 SENSEI Generic Data Interface

There are two main challenges to using *in situ* analysis for advanced modeling and simulation workflows. First, on the simulation side, is the complexity of instrumenting simulation codes to use any *in situ* infrastructure. Presently, one has to instrument their simulation codes separately for each of the infrastructures. Each infrastructure has their own idiosyncrasies that the application developer has to endure, including mapping simulation data structures to the target infrastructure. Second, on the analysis side, analysis developers face the challenge of having to decide on the infrastructure in which to implement their analysis. It is not feasible to write analysis code once and use it in various infrastructure without modifications.

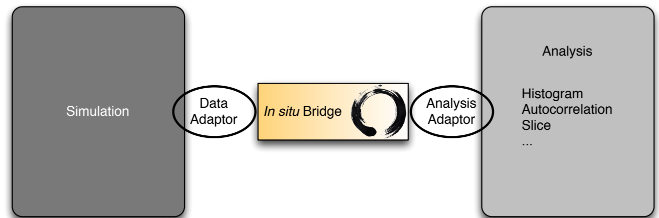

Fig. 1: The *in situ* bridge uses two adaptors (data and analysis) to hide complexity and enable *write once use everywhere* analysis.

The SENSEI generic data interface addresses both these key challenges. First, it provides application developers with a generic data interface that they then tailor for a particular use. Second, it provides analysis developers with a data model that they may use to write analysis routines. Both of these components are independent of the *in situ* infrastructure being used and hence provide both the simulation and the analysis routine isolation from which *in situ* infrastructure is being used. For example, if the application is instrumented with the SENSEI interface, application end-users can easily choose between ParaView/Catalyst and VisIt/Libsim for generating visualizations *in situ*. Furthermore, since ParaView/Catalyst and VisIt/Libsim both are treated as analysis routines under SENSEI, these visualizations can be run *in situ*, or in transit using ADIOS or GLEAN transparently.

This *write once, use anywhere* goal is only achievable when we have a mutually agreed platform for communicating the data between the simulation and analysis components – the data model. For the SENSEI interface, we selected the VTK data model [28]. The VTK data model is widely used in the scientific and engineering data analysis and visualization community, leveraged by visualization tools like ParaView [29] and VisIt [30] and hence already familiar to a broader community.

To minimize effort and memory overhead when mapping memory layouts for data arrays from applications to VTK, we enhanced the

VTK data model to support arbitrary layouts for multicomponent arrays. VTK now natively supports the commonly encountered *structure-of-arrays* and *array-of-structures* layouts. This allows for mapping data arrays from application codes to the VTK data model without additional memory copying (*zero-copy*).

Besides the data model, the other components that comprise the SENSEI interface are simple and quite light weight. Fig. 1 shows the main components of the SENSEI interface. The *data adaptor* provides a mapping between simulation data structures and the VTK data model. The *analysis adaptor* passes the data described in form of VTK data objects to any analysis code, doing any necessary transformations. The *in situ* bridge is a simple mechanism to assemble the analysis workflow, i.e., to initialize the data adaptor and execute selected analysis routines.

To instrument an application with SENSEI, one provides a concrete implementation for the *data adaptor* API. The *data adaptor* API provides the analysis code with access to mesh and attributes arrays as needed. By providing an API that encourages lazy mapping to VTK data model for the mesh and attribute arrays, the *data adaptor* avoids any work to map simulation data to VTK data when not needed. Thus when no analysis is enabled, the SENSEI instrumentation overhead is almost nonexistent.

To add an analysis routine to SENSEI, one provides a concrete implementation for the *analysis adaptor* API. The analysis adaptor is provided an instance of the *data adaptor* that it may use to gain access to the simulation data through VTK data model.

Finally, the *in situ* bridge is simply an API and the corresponding implementation that the application developer implements to pass data and control to SENSEI during the application execution. A typical bridge implementation will initialize the data adaptor and one or more analysis adaptors during the initialization phase of the simulation; then for each time step pass the current simulation data arrays and any other metadata to the data adaptor and call *execute* on the analysis adaptors.

The *analysis adaptor* is also the mechanism for the SENSEI interface to connect with the different *in situ* infrastructures. For example, an analysis adaptor may use ADIOS to save the data out to an ADIOS BP file, or it may serve as a ParaView/Catalyst-based adaptor that starts up ParaView/Catalyst to process the data using ParaView/Catalyst data processing pipelines, including rendering.

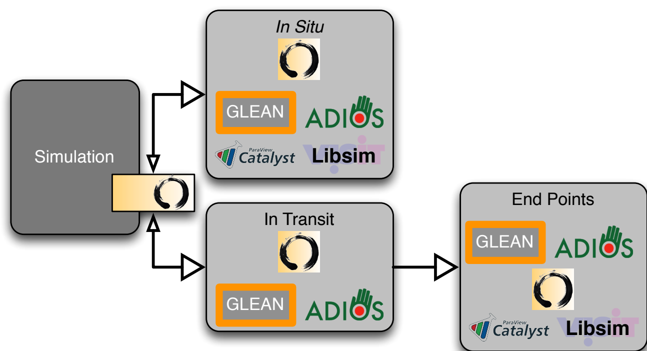

Fig. 2: The SENSEI generic data interface creates several possibilities for *in situ*, in transit, in flight and hybrid analysis. In enables a developer to instrument a simulation code once, then have access to multiple *in situ* infrastructures.

Allowing additional *in situ* infrastructures to be coupled via the SENSEI generic data interface, as depicted in Fig. 2, provides a number of analysis techniques to map to future high-performance computing architectures.

The current limitations of the SENSEI interface are an incomplete data model and an immature analysis adaptor specification. The SENSEI interface will truly be simpler when more complex simulation data structures easily map to the SENSEI data model through the data adaptor. Although this study examined several analysis and visualization use cases, this is just the tip of an iceberg of analysis techniques, and the adaptor infrastructure must grow to accommodate the requirements of the others.

3.3 Mini Application and *In Situ* Analysis Methods

As a prototypical data source, we implemented a miniapplication, an MPI code in C++, that simulates a collection of periodic, damped, or decaying oscillators. Placed on a grid, each oscillator is convolved with a Gaussian of a prescribed width. The oscillator parameters are specified as the input, which is read and broadcast from the root process. The user also specifies the time resolution, duration of the simulation, and the dimensions of the grid, partitioned between the processes using regular decomposition. The code iteratively fills the grid cells with the sum of the convolved oscillator values; the computation on each rank takes $O(mN^3)$ per time step, where m is the number of oscillators and N^3 is the size of the subgrid on the rank. The computation is embarrassingly parallel; optionally, the ranks may synchronize after every time step, but this synchronization is off in the experiments below.

As a simple analysis routine, we compute the histogram of the data. At any given time step, the processes perform two reductions to determine the minimum and maximum values on the grid. Each processor divides the range into the prescribed number of bins and fills the histogram of its local data. The histograms are reduced to the root process. The only extra storage required is proportional to the number of bins in the histogram.

As a prototypical time-dependent analysis, we compute autocorrelation. Given a signal $f(x)$ and a delay t, we find $\sum_x f(x)f(x+t)$. Starting with an integer time delay t, we maintain in a circular buffer, for each grid cell, a window of values of the last t time steps. We also maintain a window of running correlations for each $t' \leq t$. When called, the analysis updates the autocorrelations and the circular buffer. When the execution completes, all processes perform a global reduction to determine the top k autocorrelations for each delay $t' \leq t$ (k is specified by the user). For periodic oscillators, this reduction identifies the centers of the oscillators. Each MPI rank performs $O(tN^3)$ work per time step, where N^3 is the size of its subgrid, and maintains two circular buffers, each of size $O(tN^3)$.

This particular miniapplication and these *in situ* analysis methods are representative of some, but not all, common design and execution patterns (§3.1). This approach, of using miniapplications representative of larger, more complex workloads, is a commonly accepted practice in studying the performance of workloads on HPC systems (c.f., [31]).

4 Results

The objective for the experiments we run is to shed light on the performance impact that *in situ* infrastructures and methods will have on codes. These results help to shed light on understanding "the cost of *in situ*" from several different vantage points. We are interested in the potential runtime and memory footprint impact of using *in situ* methods/infrastructures when used with a focused, limited-complexity miniapplication (§4.1), as well as when used with contemporary science application codes run at extreme scale on modern architectures (§4.2).

4.1 Mini Application Study

4.1.1 Methodology and Test Configurations

Objectives. The high level objective of the miniapplication tests in this section is to provide insight into the "cost of *in situ* processing." To do so, we run the miniapplication in several different configurations, described below, with/without an *in situ* workload, and with/without an *in situ* infrastructure and *in situ* workload.

Measurement methodology. To measure the impact, or overhead, of *in situ* methods and infrastructure, we are using two metrics in these tests: runtime and memory footprint. Runtime is measured as elapsed

wall-clock time. Memory footprint is measured as the memory high water mark. In the case of parallel runs, the memory high water mark is the sum of the high water marks from all MPI ranks. We collect both measures for the various test configurations (below) and use them as the basis for gaining insight into the cost of *in situ* processing.

Platform and Problem Configuration. We ran these miniapplication tests on Cori Phase I at the National Energy Research Scientific Computing Center (NERSC). Cori Phase I is a Cray XC system based on Intel Haswell processors. It contains 1,630 compute nodes, each with two 2.3 GHz 16-core Haswell processors and 128 GB of memory. It utilizes the Cray Aries high speed *dragonfly* topology interconnect, and has a Lustre file system with 30 PB of disk and greater than 700 GB/second I/O bandwidth.

The miniapplication tests use a weak scaling configuration: at 812 (~1K), 6496 (~6K) and 45440 (~45K) cores. The amount of work per core is the same as we go from 1K to 6K, but increases by about 100K degrees of freedom per core at the 45K level. The reason for the imbalance is due to an operational limit on Cori, where we were restricted to 1420 nodes/45K cores, but performed the amount of work per core originally planned for the 50K-core configuration.

Application/*In Situ* configurations. The miniapplication test configurations, listed below, show the various combinations of the oscillator miniapplication (§3.3), *in situ* the ParaView/Catalyst (ParaView v4.4.0, Catalyst v5.0.1 with patches, which are in v5.1.0), VisIt/Libsim (v2.11.0), and ADIOS (v1.9) infrastructures (§2.2.3), different *in situ* analysis methods (§3.3), and with/without use of the SENSEI data interface.

- ***Original*:** miniapplication with no SENSEI interface and no I/O. In some test configurations, we do perform *in situ* analysis, but that coupling is done directly via subroutine call and does not use any *in situ* interface. The distinction of with vs. without analysis will be called out when needed in the subsections that follow.
- ***Baseline*:** miniapplication with SENSEI interface enabled, but no *in situ* analysis or I/O. This configuration is useful in measuring the overhead of the SENSEI data interface in isolation from other processing.
- ***Histogram*:** miniapplication with the SENSEI interface enabled, and connected directly to an *in situ* histogram calculation, but without any of the *in situ* infrastructures.
- ***Autocorrelation*:** miniapplication with the SENSEI interface enabled, and connected directly to an *in situ* autocorrelation calculation, but without any of the *in situ* infrastructures.
- ***Catalyst-slice*:** miniapplication with SENSEI interface enabled, and connected to Catalyst, which performs *in situ* rendering of a 2D slice from a 3D volume, then writes the image to disk.
- ***Libsim-slice*:** miniapplication with SENSEI interface enabled, and connected to Libsim, which performs *in situ* rendering of a 2D slice from a 3D volume, then writes the image to disk.
- ***ADIOS-FlexPath*:** miniapplication with SENSEI interface enabled, and connected to the ADIOS FlexPath *in situ* infrastructure. Within this miniapplication/*in situ* infrastructure combination, we further refine the configuration in §4.1.4 to include *in situ* workloads for histogram, autocorrelation, and Catalyst-slice.

Overview of Tests. We begin in §4.1.2 with a test that measures cumulative performance impact of the SENSEI interface over the course of an entire run where we perform an autocorrelation computation using two configurations: a subroutine-called version of the autocorrelation computation, and the *Autocorrelation* implementation. Next in §4.1.3, we go into more detail by using several different application configurations, and report both one-time (startup) costs as well as per-timestep costs. Similarly, in §4.1.4, we look at one-time (startup) and per-timestep costs, but using an ADIOS-based in transit configuration. We compare the cost of *post hoc* and *in situ* runs in §4.1.5.

4.1.2 Measuring Impact of SENSEI Interface

The focus of this test is to measure the impact to the miniapplication of the SENSEI generic data interface over the course of an entire run. This result gives a high-level view of the overall impact to the application of using the SENSEI interface over the course of the run.

We compare the runtime and memory footprint of a the *Original* configuration with subroutine-called autocorrelation, and *Autocorrelation* configurations at varying levels of concurrency in a weak-scaling study. Looking at time-to-solution (Fig. 3), and memory footprint (Fig. 4), we see no measurable difference between the two.

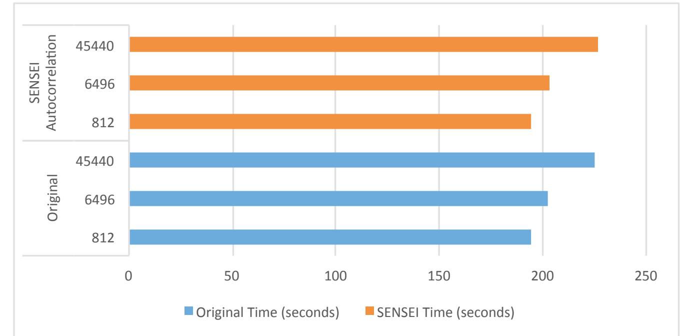

Fig. 3: Time to solution for the 1K, 6K and 45K runs comparing the *Original* and *Autocorrelation* test configurations. These results show comparable runtimes for the two configurations.

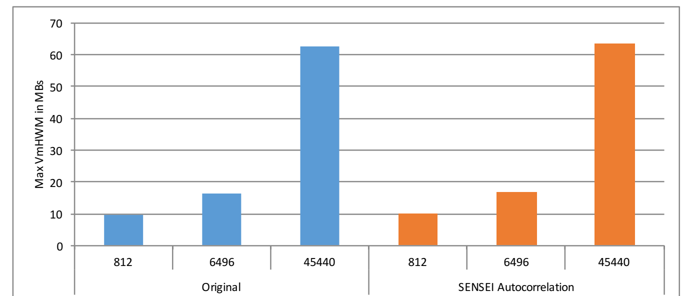

Fig. 4: Memory footprint for the 1K, 6K, and 45K configurations comparing the *Original* and *Autocorrelation* test configurations. The results show comparable memory footprint for the two configurations.

These results confirm the desired operation of a zero-copy data interface, which in this case is straightforward due to the fact both the miniapplication and the autocorrelation code are working with structured grids. These results could be different in cases where some additional effort is required to map from the source/simulation data model into the generic data model (§3.2).

4.1.3 Miniapplication, Libsim, and Catalyst SENSEI-enabled Test Configurations

In this section, we examine performance more deeply by looking at multiple *in situ* miniapplication configurations. Due to some key differences, the *ADIOS-FlexPath* in transit configuration is the subject of §4.1.4. Our analysis consists of looking at runtime performance in terms of one-time costs (Fig. 5) and recurring costs (Fig. 6), and memory utilization (Fig. 7).

Fig. 5 shows the one-time onetime costs for initializing the simulation and analysis, as well as finalization. Here, we see that the simulation initialization is negligible. The analysis initialization is minimal, although the *Libsim-slice* shows a ~3.5 second initialization time on the 45K run. This overhead currently represents per-rank configuration file checks and can be removed with very little effort. The only finalization timing that is non-negligible is for the autocorrelation

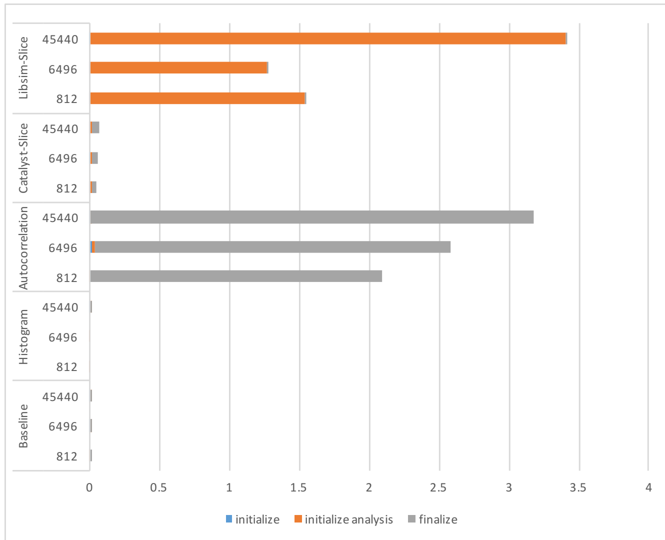

Fig. 5: The onetime costs of various *in situ* analysis and visualization use cases including: simulation initialize (blue), analysis initialize (orange), and finalize (grey). (b) The per time step costs of various *in situ* analysis and visualization use cases including: simulation (blue) and analysis (orange).

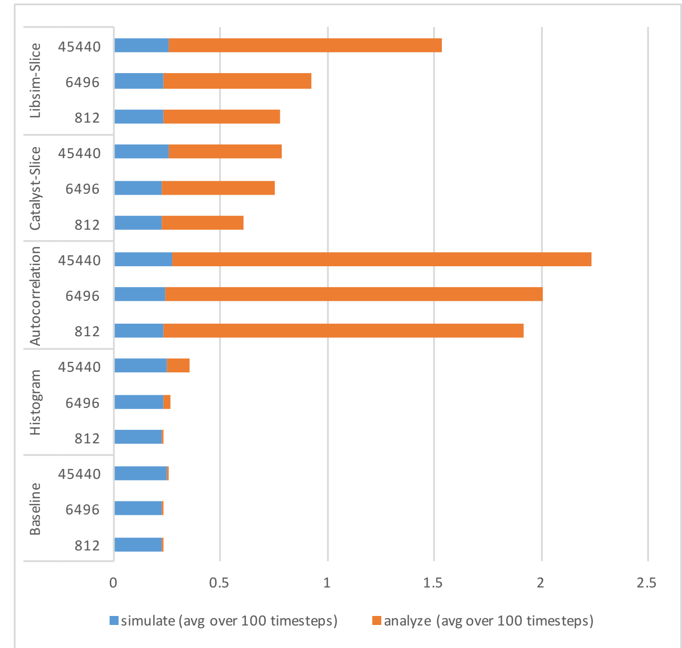

Fig. 6: The per time step costs of various *in situ* analysis and visualization use cases including: simulation (blue) and analysis (orange).

output and is due to a reduction operation that happens at the end of the computation.

In Fig. 6, we examine the per-timestep, recurring runtime measures. (As a forward reference, we later examine the overall performance metrics of these applications over their entire run in Fig. 12, where we compare *post hoc* and *in situ* configurations.) In these figures, we see that the portion of the chart labeled "simulation," which is the oscillator miniapplication, exhibits nearly perfect weak-scaling runtime performance.

For the *Catalyst-slice* and *Libsim-slice* configurations, we are extracting a 2D slice from a 3D volume, then rendering the result using a pseudocoloring, or heatmap technique. Rendering is a two-stage process. First, only those ranks whose domains intersect the slice plane will extract and render the slice geometry. Second, there is a costly compositing operation that involves communication of image-sized buffers among a hierarchical set of ranks to ultimately produce a final composite image on a single rank, which then writes the image to disk. Catalyst and Libsim use different compositing algorithms, but both perform essentially the same task, and produce images of resolution 1920x1080 and 1600x1600 respectively. There are differences in the scaling characteristics between these two algorithms visible in Fig. 6. These differences, while noticeable in these charts, are not of significant concern because scalable compositing is a challenging problem that can require significant tuning to optimize [32]. We have not endeavored to perform any specific tuning for compositing optimization as part of this study.

Looking at the memory utilization in Fig. 7, we see the memory footprint at startup and at the high-water memory mark for each use case. The startup executable footprint is essentially the equivalent to the *Baseline* simulation runs for each of the *in situ* analysis and visualization use cases. In contrast, the high-water memory mark varies once again based on the specific *in situ* analysis. This high-water mark is the sum across all MPI ranks, so it is no surprise that the memory footprint grows with scale for all processing phases.

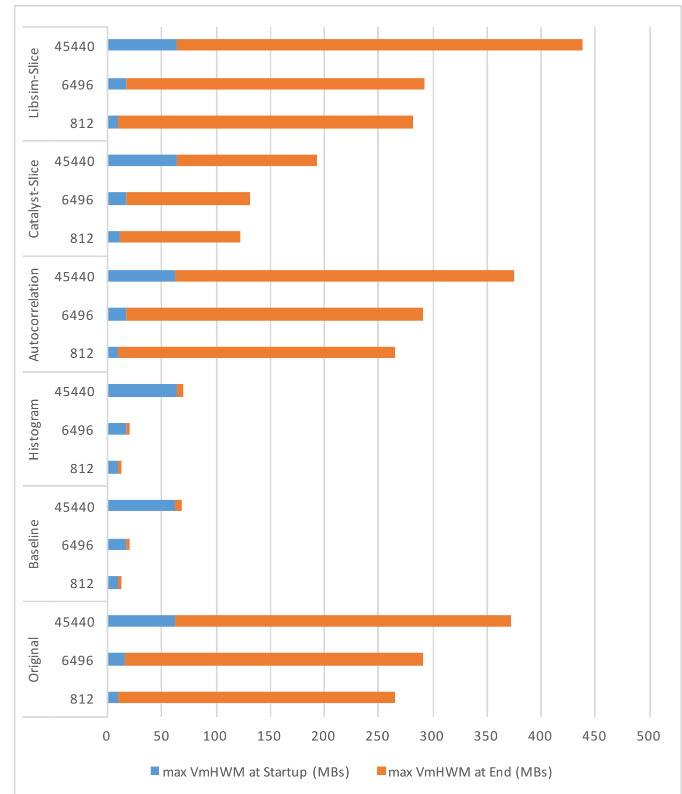

Fig. 7: The memory overhead of the studies runs where the startup executable footprint is represented in blue and the high-water memory mark throughout a run.

4.1.4 ADIOS-FlexPath SENSEI-enabled Test Configurations

The FlexPath transport [19] within ADIOS offers the ability to configure a set of resources for doing *in situ* computations in a number of configurations: on node (traditional *in situ*), across several nodes (in transit), or in hybrid combinations. For this paper, we study the performance of the *Histogram*, *Autocorrelation*, and *Catalyst-slice* configurations using an ADIOS FlexPath endpoint (as in Figure 2). For the purposes of these runs, we have deployed the endpoint onto the same nodes using Cori's slurm scheduler so that each core will have two hyperthreads: one for the simulation, and one for the analysis. This means that we use all of the same cores as the other *in situ* runs, but we

will experience some additional perturbation due to the Linux scheduler utilizing both hyperthreads.

Unlike the other methods discussed so far, the ADIOS FlexPath approach leads to having two different executables that are invoked. The simulation executable can be compiled and linked with the SENSEI infrastructure once, while the endpoint could be modified, recompiled, and redeployed in order to support changes to the intended analysis. Indeed, FlexPath allows for dynamic disconnection and reconnection, so that this could take place while the executable is running [33], although we do not expect this to be a regular concern for SENSEI usage scenarios. As a result, we report two different timing schemes: those for the writer/simulation, and those for the endpoint/analysis.

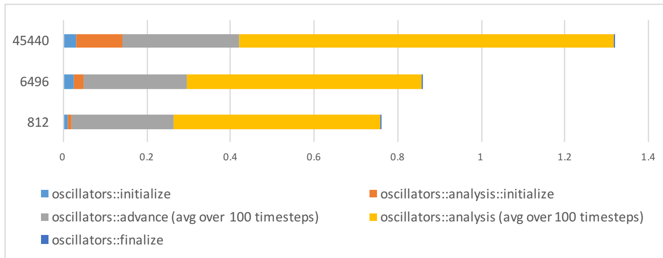

Fig. 8: Time costs (in seconds) of the one-time and per-timestep writer actions when coupled to the histogram computation.

In Figure 8, we see the costs associated with the writer. The timing for `adios::advance` is associated with updating the metadata between the writer and reader, and the `adios::analysis` timing is for transmission of data to the analysis reader as well as any blocking time if the reader is not yet ready. Note that we have not tuned the transport specifically for the hyperthreading shared memory environment; this same transport would also connect processes on separate nodes.

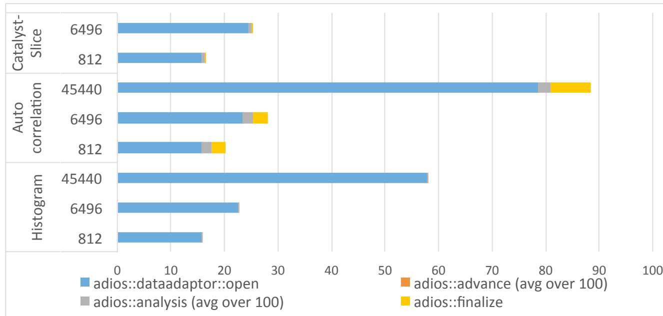

Fig. 9: ADIOS FlexPath *in situ* analysis use case timings (in seconds).

Figure 9 shows the time for the analytic component. The analysis times are in line with the execution times for the Catalyst-slice, autocorrelation and histogram timings seen in §4.1.3. The initialization times for the reader on Cori requires additional tuning; part of that may be due to additional OS jitter from the hyperthread co-allocation, as well as delays caused by shared use of the Cray interconnect. Similar runs on Titan at the Oak Ridge Leadership Class Facility had an order of magnitude lower initialization time, so this is an area for further improvement in the Cori environment.

We found that there was only an average of a 50% runtime penalty associated with the Catalyst-Slice operation compared to doing it inlined in the simulation code. This time counts the additional buffer costs, since the current FlexPath transport does not yet use zero-copy, as well as the penalty associated with co-scheduling the network and cores. A direction for future testing, which has shown promise on other hardware [34], is to subdivide the cores on each node so that, for instance, one core per socket would be for analysis, and the other eleven cores would be for simulation. Additionally, this approach can smoothly transition to in transit deployments, simply by adjusting the launch batch script.

Writes	812	6496	45440
Size	2 GB	16 GB	123 GB
VTK I/O	0.12 s	0.67 s	9.05 s
MPI-IO	0.40 s	3.17 s	22.87 s

Table 1: One time step costs of a write using multi-file VTK I/O versus unoptimized MPI-IO for 812, 6496 and 45440 core runs.

4.1.5 Comparing *In Situ* and *Post Hoc* Configurations

The studies thus far have focused on gaining insight into the "cost of doing *in situ* processing" in various configurations. In this section, we look at the problem a little differently, and do a comparison of a use case done *in situ* with one done in *post hoc* fashion. We conduct tests that quantify the costs of doing *post hoc* processing in a way that parallels the *post hoc* use model. First, a code will write data to persistent storage, which has a cost, and that we report on a per-timestep basis (Fig. 10). Later, an analysis or visualization code will read that data from persistent storage and perform its tasks, which also has a cost, that we report in terms of aggregate cost over complete *post hoc* application runs at varying concurrency (Fig. 11). Finally, we conduct weak-scaling studies of several application configurations run *in situ* (Fig. 12) for the purpose of comparing with the similar-concurrency *post hoc* runs.

Cost of writes. In Fig. 10, we compare a SENSEI-enabled *Baseline* run to a version of *Baseline* instrumented to perform data writes at every timestep. The one-time costs for both `initialize` and `finalize` have a combination of the SENSEI and miniapplication computational overhead. The SENSEI data interface computational and memory overhead is demonstrated to be negligible in §4.1.2.

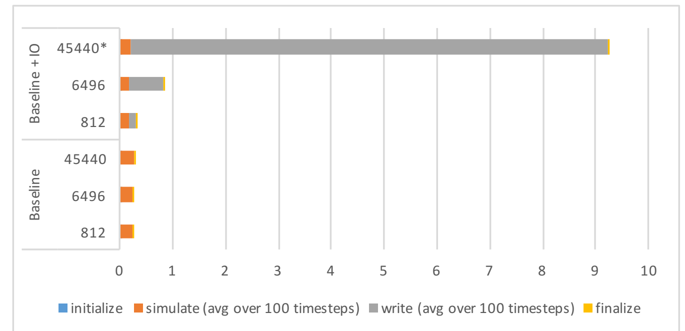

Fig. 10: For a hundred time steps, the initialize (blue), the average per time step simulation time (orange), the average per time step write time (grey), and the finalize (yellow) are depicted for a baseline and a baseline plus I/O runs.

For the 1K run, the write cost has little impact on the overall time to solution. In the 6K run, the writes take about four times as much as the simulation itself. In the 45K run, the writes take about 20× as much time. It is no surprise that I/O takes a substantial time or on-disk storage (2GB per time step for 1K, 16GB per time step for 6K and 123 GB per time step for 45K runs). Figure 10 presents a file-per-core VTK I/O, which should be faster, than a more traditional, but slower, MPI-IO approach (see Table 1).

For the MPI-IO implementation, we relied on a vanilla MPI collective I/O (using `MPI_Type_create_subarray`, `MPI_File_set_view`, `MPI_File_write_all`) to save the multi-dimensional arrays. We set the striping using NERSC's recommended `stripe_large` command. By following the MPI and NERSC recommendations, we get sub-optimal, but realistic performance.

Cost of reads. In a typical *post hoc* use case, all simulation cores will be producing data, although the exact mechanism for doing I/O—collective, non-collective, many-to-few, etc.—will vary from implementation to implementation. In contrast, on the read side, it is typically the case that far fewer cores are involved in *post hoc* analysis. In our studies below of read costs, we use 10% of the cores used in each of the

write configurations above. While this number is somewhat arbitrary, it reflects our experience in *post hoc* use cases.

In Fig. 11, we compare the read times for doing *post hoc* versions of the histogram and autocorrelation computations, along with a ParaView-based slice extraction/rendering. This figure shows significant variability in read times on the NERSC Lustre system at scale. This variability could come from (1) sharing the I/O system resources across the processes of individual run and/or (2) internal and external interference with our runs from multiple simultaneous running programs, which are discussed in detail by Lofstead et al. [35].

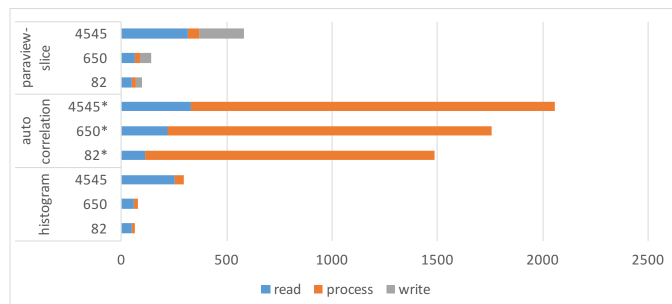

Fig. 11: The histogram, autocorrelation, slice with ParaView post hoc analysis and visualization with time broken out in a stacked bar chart including read (blue), process (orange) and write (grey). Note that since for the *post hoc* scenario, we are using 10% of the cores used to generate the data, we are showing core counts of 82, 650, and 4545. *The autocorrelation runs were done using twice as many nodes (same number of MPI ranks) as the other runs since they need more memory to cache timesteps for the analysis.

There's no surprise here that the read takes a significant amount of time, as much as 5× to 10× that of the miniapplication itself. While changing the number of cores used for reads could change these results, we believe it demonstrates a common *post hoc* use of computational resources.

Comparing *In Situ* and *Post Hoc* configurations. The overall times to solution for the *in situ* configurations, shown in Fig 12, are significantly faster than the *post hoc* configurations. While we did not construct a chart showing the sum of write and read times, it is easy to see the ~9 seconds/write at 45K concurrency, multiplied by 100 timesteps, is significantly longer than any of the configurations shown in Fig. 12.

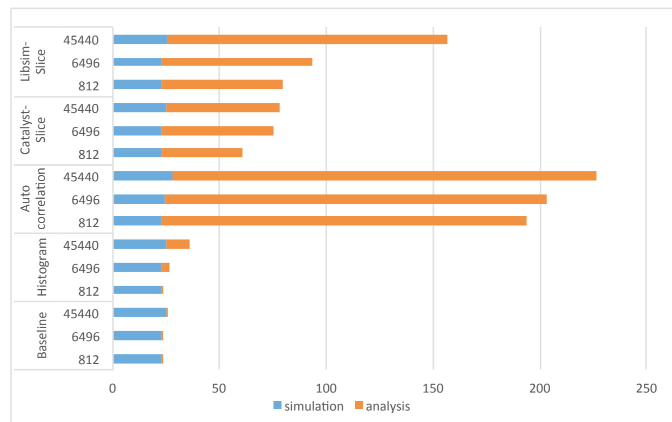

Fig. 12: Weak-scaling of the oscillator simulation with associated analysis time in a time to solution bar graph.

4.2 Science Application Examples

In the subsections that follow, we present results showing coupling three science application codes with the SENSEI *in situ* adaptor, and thence to *in situ* infrastructures. We selected these exemplar applications because they are known to scale on current extreme-scale DOE systems, and also we have a collaborative relationship with the code development teams. These applications cover a spectrum in advanced modeling and simulation including structured- and unstructured-mesh computational approaches.

4.2.1 PHASTA Science Application

Name of code. PHASTA solves the Navier-Stokes [36, 37] equations either directly (DNS) or after turbulence modeling [38] using a stabilized finite element method. It has scaled to over 3M processes [39] on applications ranging from fundamental science benchmarks like channel flow [40] to more complex systems like aircraft [41, 42], cardiovascular systems [43, 44] and multiphase flow [45, 46]. It is an open source code [47] led by K.E. Jansen.

In Situ implementation. PHASTA's[1] core computational routines are written in Fortran 90 and compute over an unstructured grid. The data adaptor uses VTK's zero-copy ability to map the nodal coordinates and field variables while the VTK grid connectivity is a full copy. The grid and fields are constructed as needed but the pointers to the PHASTA grid data structures are passed every time *in situ* is accessed.

The SENSEI infrastructure used Catalyst functionality to generate image outputs from slices, which are 2D slices extracted from a 3D mesh and pseudo-colored by velocity magnitude. To reduce executable size, we used a specific Catalyst Edition that includes rendering and a small subset of the filters available in VTK and ParaView. This Edition executable size was 153 MB after SENSEI, Catalyst and its dependencies (e.g. OSMesa) were statically linked with PHASTA. Without static linking, the executable size was 87 MB.

Platform. Our target platform for running PHASTA was Mira, a BlueGene/Q, at Argonne National Laboratory. PHASTA runs most efficiently on Mira with 4 MPI ranks per core which results in 64 MPI ranks per node. This results in 256 MB of memory per MPI rank. With SENSEI's Catalyst slice output, an image size of 800x200 was able to be generated keeping the same run configuration. When the image output size was increased to 2900x725, the simulation ran out of memory so the number of MPI ranks per core was halved and the number of cores used was doubled. The runs we performed on Mira include:

IS1 1.28 Billion element grid with 262,144 MPI ranks on 4,092 nodes (64 MPI ranks per node) with output image size of 800x200 and 120 time steps.

IS2 1.28 Billion element grid with 262,144 MPI ranks on 8,192 nodes (32 MPI ranks per node) with output size of 2900x725 and 120 time steps.

IS3 6.33 Billion element grid with 1,048,576 MPI ranks on 32,768 nodes (32 MPI ranks per node) with output size of 2900x725 and 30 time steps.

Note that all runs were done outputting images every other time step.

Application results. PHASTA was used to simulate flow over a vertical tail-rudder assembly for a geometry that exactly matches the configuration of an ongoing wind tunnel experiment. As shown in Figure 13, SENSEI provides live, reconfigurable data analytics from an ongoing simulation. This capability enables the science in two important ways. First, it allows scientists and engineers to confirm that the ongoing simulation is properly set up since they can quickly scan the domain to be sure realistic results are being obtained. Second, PHASTA allows many of its input parameters to be reconfigured on the fly. In this way the SENSEI results close the loop on live problem redefinition thus enabling scientists and engineers to interactively explore

[1]PHASTA available at https://github.com/PHASTA/phasta. Our work was based on Git revision "39bc1351f6d" with modifications to use the SENSEI interface.

the fluid flow and see the response in "really useful" time. In this particular application, a very small synthetic jet is placed near the point where the flow would otherwise separate but, using visual feedback from images provided by SENSEI, the frequency and the amplitude of the flow control can be manipulated to interactively determine the combination that, through interactions with the primary flow structures, provide the most improvement to the aerodynamic performance of the aircraft.

Fig. 13: Sample zoomed slice of PHASTA simulation through the wing.

Discussion. The main purpose of these runs was to examine how well the SENSEI infrastructure worked at large scale. We started with the *IS1* and *IS2* runs in order to get a baseline for code performance at a reasonably high process count. We then pushed for the larger *IS3* run to ensure that SENSEI will operate efficiently at the problem and compute size that codes like PHASTA are targeting. The timing numbers are shown in Table 2.

The time spent in *in situ* is deemed reasonable for the *IS1* and *IS3* runs with 8.2% and 13% of the run time dedicated to *in situ* processing. Taking 33% of the compute time for the *IS2* run case may be considered excessive enough for some data scientists to avoid using *in situ*. Because of this, we delved further into the timings to analyze what could be improved. The SENSEI one-time costs are a very small fraction of the total compute time and the *in situ* compute time per time step dominates. We found it surprising that there was a significant increase in *in situ* compute time per time step when changing the size of the outputted image (runs *IS1* and *IS2*) while very little difference when the problem and compute size differed (runs *IS2* and *IS3*). Upon further investigation, we determined that the ZLIB compression time in generating the PNG file was the culprit. This is a serial process only computed on rank 0. For an 8 process toy problem, the *in situ* compute time per time step went from 4.03 seconds to 0.518 seconds when skipping the compression portion of generating the PNG file.

Run	*In Situ* One-Time Cost	*In Situ* Compute per Time Step	Total Time	Percent *In Situ* Time
IS1	1.76	1.40	1051	8.2
IS2	1.07	5.24	962	33
IS3	1.93	5.62	653	13

Table 2: PHASTA execution times in seconds.

4.2.2 AVF/Leslie Science Application

Name of code. AVF-LESLIE [48, 49, 50] is a reactive flow multi-physics code 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. AVF-LESLIE is written in FORTRAN90 and has capability to export volumetric datafiles in several formats. It has been used for various applications: flame-vortex interaction, premixed flame turbulence interaction, non-reactive channel/Couette flow, and passive scalar mixing.

***In Situ* implementation.** We instrumented AVF-LESLIE (v10) with a SENSEI adaptor that calculates vorticity magnitude and exposes data array slices (to remove ghost cells). The SENSEI analysis adaptor was provided a VisIt session file to set up the visualization. The visualization consists of 3 isosurfaces and 3 slice planes of vorticity magnitude. Its purpose is to give a visual reference to the evolution of the turbulent flow features from initial mixing through homogeneous turbulence.

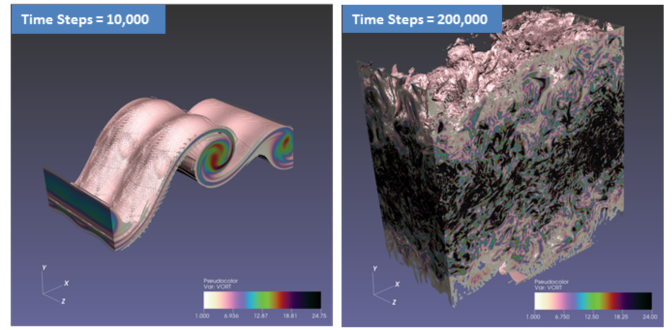

Fig. 14: The Evolution of Temporal Mixing Layer from Initial to Vortex Breakdown

Platform. We conducted benchmarks on Titan at Oak Ridge Leadership Class Compute Facility. The scaling studies were performed on a Cartesian grid size of 1025^3 and physical non-dimensional domain sizes of 4π x 4π x 2π. The study used between 8192 and 131072 cores, using all 16 cores per compute node.

Application results. The benchmarks study the strong-scaling characteristics of AVF-LESLIE, and the memory and computational overhead associated with the *in situ* methods and infrastructure.

Each run represents AVF-LESLIE running for 100 time steps, with SENSEI being called at each time step and Libsim analysis every 5 time steps. The study simulates unsteady dynamics of a temporally evolving planar 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 [51]. Figure 14 presents visualizations of the TML flowfield at 10,000 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.

Before *in situ* processing was implemented, AVF-LESLIE scaled well up to 16K cores, but efficiency degraded at higher core counts. After SENSEI/Libsim *in situ* rendering was added, the per-iteration time for AVF-LESLIE increased due to the time taken for *in situ* processing. The time taken to initialize SENSEI increases with processor count, largely due to one-time Libsim initialization costs (see §4.1.3). The analysis time `"avf_insitu::analyze"`, which includes the time to expose data to SENSEI, read a session file, set up plots, perform data extraction, render geometry, create the composited image, and save the image, quickly exceeded the time spent in the solver `"avf timestep"` due to the complexity of the visualization, as shown in Figure 15. *In Situ* analysis time is highly dependent on the complexity of the analysis, or in this case, the nature of visualizations being produced. Over the 100 time step run, the costs of calling Libsim to produce rendered images added an average of 1-1.5 seconds per time step to the solver runs over the numbers of cores tested.

Discussion. Since the Libsim visualization was complex, it was executed one out of every 5 times in which SENSEI was invoked by the solver. This means that 4/5 times, the SENSEI analysis time was low and the 1/5 times that Libsim analysis was invoked, the time was high. To see the actual costs of calling Libsim to produce the visualizations, AVF-LESLIE reported the time spent in SENSEI analysis for each time step. Figure 16 shows for the 65K run that the cost of generating the images via Libsim is in the range of 7-8 seconds while the normal SENSEI overhead for the data adaptor is less than 0.5 seconds, showing that SENSEI overhead is low and analysis overhead can be arbitrarily high, depending on the requested operations.

It is important to contrast the *in situ* overhead to the traditional *post hoc* workflow. At 1025^3 and 65K core, AVF requires approximately 24

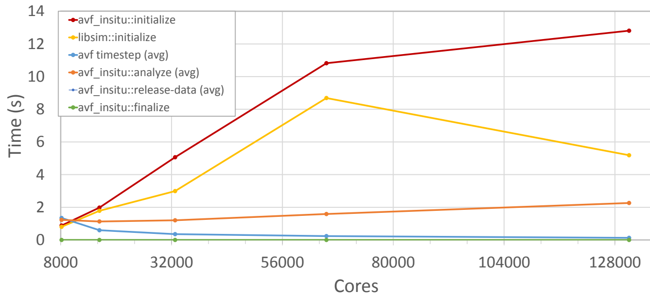

Fig. 15: AVF-LESLIE Performance with SENSEI/Libsim *In Situ* Processing 1025^3 Dataset (Render Session for Isosurface and Coordinate Cuts).

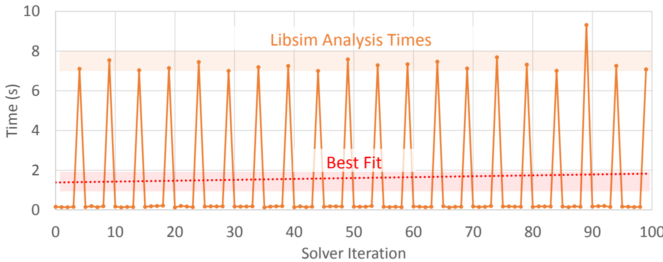

Fig. 16: Per Iteration cost for calling SENSEI at 65K on 1025^3 dataset when Libsim analysis pipeline is invoked every 5 time steps.

seconds to save a time step of volume data. Therefore, based upon the current overhead numbers, one can afford 3-4 times greater temporal resolution for visualization and analysis in comparison to writing out volume data for *post hoc* processing. This capability is very important for transient data and resolving turbulent flow characteristics. Future work is focused on reducing this *in situ* overhead further and applying it towards extracting further knowledge about the TML flowfield.

4.2.3 Nyx – Computational Cosmology

Name of code. Nyx [52] is BoxLib-based code—co-developed by the Center for Computational Sciences and Engineering (CCSE) and the Center for Computational Cosmology (C^3) at the Lawrence Berkeley National Laboratory—for large-scale cosmological simulations exploring structure formation in the universe[2].

Problem focus. We quantify the impact of integrating lightweight *in situ* analysis—computing histograms using VTK and slices using ParaView/Catalyst—on simulation time and memory requirements for a realistic application scenario. In contrast to the miniapplication study (Section §3.3), Nyx simulation time steps require significantly more compute time to complete, reducing the impact of analysis on the simulation.

***In Situ* implementation.** We consider only simulations that do not use adaptive mesh refinement and represent the domain as a single level, comprising axis-aligned rectilinear boxes. We avoid data replication by directly passing a pointer to the BoxLib data to VTK and blanking out ghost cells in the Nyx simulation by associating a `vtkGhostLevels` attribute—a byte array of flags marking ghost cells—with the mesh.

Platform. We evaluate the impact of the SENSEI framework on the resource utilization of the Nyx code by running 40 timesteps, the typical number of steps comprising a convergence study [53], on the Cori system at NERSC. We consider simulations with grid sizes of 1024^3, 2048^3 and 4096^3 initialized from files with 1024^3, 2048^3 and 4096^3 particles, respectively. To ensure load balancing, we run the simulations on 512 cores (16 nodes), 4096 cores (128 nodes) and 32768 cores (1024) nodes for the 1024^3, 2048^3 and 4096^3 simulations respectively, using one MPI rank per core. We compare simulations that use SENSEI to compute histograms or a slice to running a baseline Nyx executable compiled without SENSEI support. Due to limited compute time, we omitted histogram computation for the 4096^3 simulation.

[2]Nyx is available to CCSE collaborators at gamera.lbl.gov:/usr/local/gitroot/Nyx.gita. We used the Nyx "LyA" simulation based on Git revision "0b2a7e613c1c7f56108fd3b869e27fcfe0700e2b." Nyx is based on BoxLib, which is available at https://github.com/BoxLib-Codes/BoxLib. Our experiments used Git revision "d3f5141be3a8768bbf1d540f4172bdbd716970b1" of BoxLib.

Application results.

Executable size: The SENSEI framework increases the size of the static Nyx executable significantly from 68MB to 109MB, also resulting in longer link times. This size increase has a negligible effect on simulation run time and memory utilization.

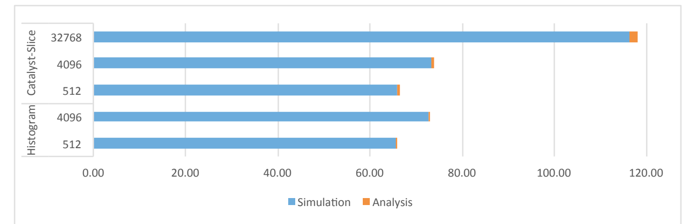

Fig. 17: Scaling results for Nyx instrumented with SENSEI *in situ* analysis. Times are averaged over all time steps.

Simulation wall-clock times and time overhead per simulation time step: Running the Nyx simulations took approximately 45 minutes for the 1024^3 simulation, one hour for 2048^3 simulation and two hours fifteen minutes for the 4096^3 simulations. Wall-clock time differences between runs of the same simulation—due to differences in I/O and communication network utilization—were generally larger than the time added by the analysis. In fact, some runs without *in situ* analysis took longer to complete than some run with analysis enabled. Figure 17 shows that the *in situ* analysis time is negligible compared to solution time, both for the histogram and the slice at all concurrency levels. Since the simulation runs 40 timesteps and both histogram and slice require less than a second per time step, the total run time difference due to *in situ* analysis is less than a minute, less than a typical time difference between multiple runs of the same simulation.

Memory overhead: We collected system memory statistics (VmPeak, VmSize, VmHWM, VmRSS) using BoxLib's built in profiling for all runs. Most statistics varied significantly over multiple runs of the same executable, making interpretation difficult. The peak resident set size (VmHWM) is the most stable measure across multiple runs. Based on these measurements, the memory overhead for the histogram is minimal, mainly due to the overhead of 2MB for the ghost zone array per MPI rank. The slice increases memory usage by 200–300MB, which is small compared to simulation size and memory available per node and consistent with our expectations based on the miniapplication study. Further experiments are necessary to quantify the exact memory overhead.

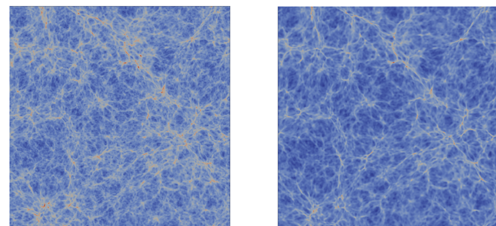

Fig. 18: Time steps 200 and 300 of the 1024^3 Nyx Lyman α forest simulation. Simulations often only save every 100th time step. The difference between these time steps is considerable, hampering feature tracking.

Temporal resolution: Simulations typically only produce a plot file for analysis every 100th time step to avoid I/O. Figure 18 shows that the simulation changes significantly over a 100 time steps, making it difficult to track features. Producing images for every time step makes it possible to observe gradual changes in the simulation and easily track features.

Posthoc analysis: Writing plot files takes approximately 17 seconds for 1024^3, 80 seconds for 2048^3 and 312 seconds for 4096^3, though these plot files contain eight variables, more than we use in the analysis. Even if some analysis needs to be performed post hoc, each plot file that does not need to be written to disk saves significant time, making it possible for *in situ* analysis to amortize itself.

Discussion. In several use cases, *in situ* analysis can produce valuable results with barely noticeable impact on the simulation. Going forward, we need to add more analysis operations to the SENSEI framework. Typically Nyx simulations use 1–2 MPI ranks per compute node and use OpenMP within a node. For effective use in simulations, *in situ* analysis must support hybrid MPI+OpenMP (or other thread-based) execution models.

4.3 Discussion and Lesson Learned

The miniapplication, as a low-cost proxy for a simulation, proved to be a highly useful approach for studying the runtime/memory impact of *in situ* methods and infrastructures at scale. With the miniapplication, we were able to quickly identify and repair many bottlenecks that would otherwise not be apparent when run at smaller scale. It enabled us to quickly explore different design patterns for *in situ* methods, and to engage in an agile and rapid design/test cycle for the generic data interface. The miniapplication approach has a much lower cost-of-entry for these activities compared to working with a production simulation code.

5 Conclusion and Future Work

The primary findings suggest that, even in the presence of some variation from one *in situ* infrastructure or method to another, that the runtime and memory overhead for *in situ* methods and infrastructure is quite low, especially when compared to the performance of modern computational solver codes. We show that using *in situ* for analysis offers significant cost advantages when compared to the *post hoc* approach when using a traditional I/O path. The performance studies reveal room for improvement in all *in situ* infrastructures. For example, weak-scaling characteristics for a BSP design and execution pattern where there is a final reduction (e.g., Fig. 12) show room for improvement as concurrency increases. The studies, both miniapplication and science applications, focus on a specific type of design and execution pattern. Future work will examine additional design and execution patterns, as well as other types of *in situ* use cases and science applications having additional computational approaches, such as unstructured meshes and particle-based codes.

The idea of a generic data interface that would enable portability is something that is of significance. Our results show that a particular approach, the SENSEI generic *in situ* interface, (§3.2) is highly flexible, has low overhead, and is broadly applicable to a potentially large number of simulation codes. It provides portability, so that a simulation using the interface can make use of, without code changes, four different *in situ* infrastructures. Conversely, an *in situ* method that makes use of this interface can, without code modifications, be used in multiple *in situ* infrastructures.

As platforms continue to evolve, it will be increasingly important that *in situ* methods and infrastructures are able to adapt to new platforms as well as new simulation codes that run on those platforms. For example, concepts like run-time configuration of shared cores within a node is of great interest, as simulations will often adapt themselves for optimal performance, perhaps by choosing core counts that allow them to meet certain memory footprint requirements, or by taking advantage of architectural features, like the burst buffers on Cori, to achieve accelerated staging operations or as a workaround to node-level memory limitations. *In Situ* methods and infrastructures need to adapt to work harmoniously in these settings. Ongoing benchmarking will aid in better understanding how changes in architecture and system components might affect the balance of *in situ* vs. *post hoc* performance. We found Cori to be a highly useful platform. We complemented those results with large-scale science application runs on Mira and Titan; the *in situ* elements of those runs performed as predicted by the the miniapplication results on Cori.

Acknowledgements

This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, through the grant "Scalable Analysis Methods and *In Situ* Infrastructure for Extreme Scale Knowledge Discovery," program manager Dr. Lucy Nowell. This research used resources of the Argonne Leadership Computing Facility (ALCF), the Oak Ridge Leadership Computing Facility (OLCF), and the National Energy Research Scientific Computing Center (NERSC).

References

[1] R. W. Hamming, *Numerical Methods for Scientists and Engineers (1st. ed.)*. McGraw-Hill, 1962.

[2] S. A. (ed.), "Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization," Houston, TX, USA, Feb. 2011.

[3] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, and E. W. Bethel, "*In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms, a State-of-the-art (STAR) Report," *Computer Graphics Forum, Proceedings of Eurovis 2016*, vol. 35, no. 3, Jun. 2016.

[4] H. Childs and et al., "The In Situ Terminology Project," https://ix.cs.uoregon.edu/ hank/insituterminology/index.cgi?n=Phase1D.Phase1DSurveyInput, Feb. 2016.

[5] E. E. Zajac, "Computer-made perspective movies as a scientific and communication tool," *Communications of the ACM*, vol. 7, no. 3, pp. 169–170, Mar. 1964.

[6] "NCAR Graphics," http://ngwww.ucar.edu/, last accessed Feb. 2016.

[7] R. Heiland and M. P. Baker, "A survey of co-processing systems," Technical Report, NCSA University of Illinois, Tech. Rep., August 1998. [Online]. Available: http://sda.iu.edu/docs/CoprocSurvey.pdf

[8] G. A. Geist, J. A. Kohl, and P. M. Papadopoulos, "CUMULVS: Providing Fault-Tolerance, Visualization, and Steering of Parallel Applications," *International Journal of High Performance Computing Applications*, vol. 11, no. 3, pp. 224–236, 1997.

[9] R. Haimes, "pV3: A Distributed System for Large-scale Unsteady Visualization," in *AIAA Paper 91-0794*, 1994.

[10] C. Upson, T. A. Faulhaber, Jr., D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam, "The Application Visualization System: a computational environment for scientific visualization," *j-IEEE-CGA*, vol. 9, no. 4, pp. 30–42, Jul. 1989.

[11] J. D. Mulder, J. J. van Wijk, and R. van Liere, "A survey of computational steering environments," *Future Gener. Comput. Syst.*, vol. 15, no. 1, pp. 119–129, Feb. 1999. [Online]. Available: http://dx.doi.org/10.1016/S0167-739X(98)00047-8

[12] T. Goodale, G. Allen, G. Lanfermann, J. Mass, T. Radke, E. Seidel, and J. Shalf, "The Cactus Framework and Toolkit: Design and Applications," in *Vector and Parallel Processing - VECPAR '2002, 5th International Conference*. Springer, 2003.

[13] "SCIRun: A Scientific Computing Problem Solving Environment," 2015, Scientific Computing and Imaging Institute (SCI), Download from http://www.scirun.org.

[14] A. C. Bauer, B. Geveci, and W. Schroeder, *The ParaView Catalyst User's Guide v2.0*. Kitware, Inc., 2015.

[15] N. Fabian, K. Moreland, J. Mauldin, B. Boeckel, U. Ayachit, and B. Geveci, "Instruction memory overhead of in situ visualization and analysis libraries on hpc machines," Ultrascale Visualization Workshop at SC'14, November 2014.

[16] B. Whitlock, J. M. Favre, and J. 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. Eurographics Association, 2011, pp. 101–109.

[17] H. Childs, K.-L. Ma, H. Yu, B. Whitlock, J. Meredith, J. Favre, S. Klasky, N. Podhorszki, K. Schwan, M. Wolf, M. Parashar, and F. Zhang, "In Situ Processing," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, ser. Chapman & Hall, CRC Computational Science, E. W. Bethel, H. Childs, and C. Hansen, Eds. Boca Raton, FL, USA: CRC Press/Francis–Taylor Group, Nov. 2012, pp. 307–329, http://www.crcpress.com/product/isbn/9781439875728, LBNL-6466E.

[18] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield *et al.*, "Hello adios: the challenges and lessons of developing leadership class i/o frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[19] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/-subscribe system for large-scale science analytics," in *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. IEEE, 2014, pp. 246–255.

[20] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka, "Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 19:1–19:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063409

[21] A. Globus, "A software model for visualization of large unsteady 3-d cfd results," in *33rd Aerospace Sciences Meeting and Exhibit*, ser. AIAA, 1995. [Online]. Available: http://arc.aiaa.org/doi/abs/10.2514/6.1995-115

[22] Y. Ye, R. Miller, and K.-L. Ma, "In situ pathtube visualization with explorable images," in *Proceedings of the 13th Eurographics Symposium on Parallel Graphics and Visualization*, ser. EGPGV '13. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2013, pp. 9–16. [Online]. Available: http://dx.doi.org/10.2312/EGPGV/EGPGV13/009-016

[23] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. 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*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 424–434. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.40

[24] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV '13)*, Oct. 2013, pp. 67–75.

[25] T. Fogal, F. Proch, A. Schiewe, O. Hasemann, A. Kempf, and J. Krüger, "Freeprocessing: Transparent in situ visualization via data interception," in *Eurographics Symposium on Parallel Graphics and Visualization: EG PGV:[proceedings] sponsored by Eurographics Association in cooperation with ACM SIGGRAPH. Eurographics Symposium on Parallel Graphics and Visualization*, vol. 2014. NIH Public Access, 2014, p. 49.

[26] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, "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*, ser. ISAV2015. New York, NY, USA: ACM, 2015, pp. 30–35. [Online]. Available: http://doi.acm.org/10.1145/2828612.2828625

[27] "Conduit," April 2016. [Online]. Available: http://software.llnl.gov/conduit/

[28] "VTK," June 2010. [Online]. Available: http://www.vtk.org/

[29] "ParaView," June 2010. [Online]. Available: http://www.paraview.org/

[30] "VisIt," June 2015. [Online]. Available: http://visit.llnl.gov

[31] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[32] M. Howison, E. W. Bethel, and H. Childs, "Hybrid Parallelism for Volume Rendering on Large, Multi, and Many-core Systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 17–29, Jan. 2012, lBNL-4370E.

[33] J. Dayal, J. Lofstead, G. Eisenhauer, K. Schwan, M. Wolf, H. Abbasi, and S. Klasky, "Soda: Science-driven orchestration of data analytics," in *e-Science (e-Science), 2015 IEEE 11th International Conference on*. IEEE, 2015, pp. 475–484.

[34] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, p. 78.

[35] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the io performance of petascale storage systems," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–12.

[36] C. H. Whiting, K. E. Jansen, and S. Dey, "Hierarchical basis in stabilized finite element methods for compressible flows," *Comp. Meth. Appl. Mech. Engng.*, vol. 192, no. 47-48, pp. 5167–5185, 2003.

[37] C. H. Whiting and K. E. Jansen, "A stabilized finite element method for the incompressible Navier-Stokes equations using a hierarchical basis," *International Journal of Numerical Methods in Fluids*, vol. 35, pp. 93–116, 2001.

[38] K. E. Jansen, "A stabilized finite element method for computing turbulence," *Comp. Meth. Appl. Mech. Engng.*, vol. 174, pp. 299–317, 1999.

[39] M. Rasquin, C. Smith, K. Chitale, S. Seol, B. Matthews, J. Martin, O. Sahni, R. Loy, M. Shephard, and K. Jansen, "Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wing design," *Computing in Science and Engineering*, vol. 16, no. 6, pp. 13–21, 2014.

[40] A. E. Tejada-Martínez and K. E. Jansen, "A dynamic Smagorinsky model with dynamic determination of the filter width ratio," *Physics of Fluids*, vol. 16, pp. 2514–2528, 2004.

[41] O. Sahni, J. Wood, K. Jansen, and M. Amitay, "Three-dimensional interactions between a finite-span synthetic jet and a crossflow," *Journal of Fluid Mechanics*, vol. 671, pp. 254–287, 2011.

[42] J. Vaccaro, Y. Elimelech, Y. Chen, O. Sahni, K. Jansen, M., and Amitay, "Experimental and numerical investigation on steady blowing flow control within a compact inlet duct," *International Journal of Heat and Fluid Flow*, vol. 54, pp. 143–152, 2015).

[43] I. Vignon-Clementel, A. Figueroa, K. Jansen, and C. Taylor, "Outflow boundary conditios for three-dimensional finite element modeling of blood flow and pressure in arteries," *Comp. Meth. Appl. Mech. Engng.*, vol. 195, pp. 3776–3796, 2006.

[44] H. J. Kim, C. A. Figueroa, T. J. R. Hughes, K. E. Jansen, and C. A. Taylor, "Augmented lagrangian method for constraining the shape of velocity profiles at outlet boundaries for three-dimensional finite element simulations of blood flow," *Comput. Methods Appl. Mech. Engrg.*, vol. 198, no. 45-46, pp. 3551–3566, 2009.

[45] F. Behafarid, K. Jansen, and M. Podowski, "A study on large bubble motion and liquid film in vertical pipes and inclined narrow channels," *International Journal of Mulitphase Flow*, vol. 75, pp. 288–299, 2015.

[46] J. Rodriguez, O. Sahni, R. L. Jr., and K. Jansen, "A parallel adaptive mesh method for the numerical simulation of multiphase flows," *Computers and Fluids*, vol. 87, pp. 115–131, 2013.

[47] K. Jansen, "https://github.com/phasta."

[48] C. D. Spradling, "Spec cpu2006 benchmark tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[49] T. M. Smith and S. Menon, "The structure of premixed flames in a spatially evolving turbulent flow," *Combustion science and technology*, vol. 119, no. 1-6, pp. 77–106, 1996.

[50] C. Stone and S. Menon, "Open-loop control of combustion instabilities in a model gas turbine combustor*," *Journal of Turbulence*, vol. 4, no. 20, 2003.

[51] R. W. Metcalfe, S. A. Orszag, M. E. Brachet, S. Menon, and J. J. Riley, "Secondary instability of a temporally growing mixing layer," *Journal of Fluid Mechanics*, vol. 184, pp. 207–243, 1987.

[52] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. V. Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[53] Z. Lukić, C. W. Stark, P. Nugent, M. White, A. A. Meiksin, and A. Almgren, "The lyman α forest in optically thin hydrodynamical simulations," *Monthly Notices of the Royal Astronomical Society*, vol. 446, no. 4, pp. 3697–3724, 2015.