SAND2019-12759C

In-Situ Visualization for the Large Scale Computing Initiative Milestone

Jeffrey A. Mauldin
Sandia National Laboratories
Albuquerque, New Mexico
jamauld@sandia.gov

Thomas J. Otahal
Sandia National Laboratories
Albuquerque, New Mexico
tjotaha@sandia.gov

Anthony M. Agelastos
Sandia National Laboratories
Albuquerque, New Mexico
amagela@sandia.gov

Stefan P. Domino
Sandia National Laboratories
Albuquerque, New Mexico
spdomin@sandia.gov

ABSTRACT

The Sandia National Laboratories (SNL) Large-Scale Computing Initiative (LSCI) milestone required running two parallel simulation codes at scale on the Trinity supercomputer at Los Alamos National Laboratory (LANL) to obtain presentation quality visualization results via in-situ methods. The two simulation codes used were Sandia Parallel Aerosciences Research Code (SPARC) and Nalu, both fluid dynamics codes developed at SNL. The codes were integrated with the ParaView Catalyst in-situ visualization library via the SNL developed Input Output SubSystem (IOSS). The LSCI milestone had a relatively short time-scale for completion of two months. During setup and execution of in-situ visualization for the milestone, there were several challenging issues in the areas of software builds, parallel startup-times, and in the a priori specification of visualizations. This paper will discuss the milestone activities and technical challenges encountered in its completion.

CCS CONCEPTS

• **Computing methodologies → Massively parallel and high-performance simulations**; *Computer graphics*; • **Software and its engineering →** *Massively parallel systems.*

KEYWORDS

In-situ, Scientific Visualization

ACM Reference Format:
Jeffrey A. Mauldin, Thomas J. Otahal, Anthony M. Agelastos, and Stefan P. Domino. 2018. In-Situ Visualization for the Large Scale Computing Initiative Milestone. In *Proceedings of ISAV '19: In-Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV '19).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/1122445.1122456

STATEMENTS AND DISCLAIMERS

1 INTRODUCTION

The generation of scientific analysis and visualizations spans a continuum from highly specific and tightly coupled with the simulation code, in-situ visualization [4], to highly general and loosely coupled through output data files, post-processing visualization. At each point along this continuum, there is a tradeoff among runtime, output frequency, and data size. At the most tightly coupled point, the output might be a single number. This number represents an extremely specific analysis, takes up little space on disk, and might be computed at a very high rate. At the most loosely coupled point, the entire simulation data set might be saved; however, this data set takes up a large amount of disk space, so it can only be performed a few times over the course of the simulation.

The LSCI targeted running key SNL codes at very large scales on the LANL Trinity supercomputer to obtain high-quality visualizations. High-quality visualization is necessary to communicate and emphasize the value of these codes to the Advanced Simulation and Computing (ASC) program. Two SNL codes, SPARC [6] and Nalu [5], were selected for in-situ visualization because prior integration work with ParaView Catalyst [1] at smaller scales had been successful. For LSCI, large scale was defined as using more than half of the Knight's Landing (KNL) CPU architecture nodes on Trinity.

2 DESCRIPTION OF TRINITY

The Trinity supercomputer is a single parallel computing system composed of two processor architectures. There are 9,436 compute nodes that each contain 2 Intel Haswell processors supporting 64 hyper-threads per node. The remaining 9,984 nodes contain Intel Xeon Phi processors KNL and support 272 hyper-threads per node and also have an extra vector instruction unit per core. Intel compilers support additional vector optimization flags to make use of the KNL architecture.

Both Haswell and KNL processors have an extra vector instruction unit per core; for Haswell, it's an AVX2 vector unit, and for KNL it's an AVX512 vector unit that supports wider vector operations. The Intel compilers support additional vector optimization flags to automatically make use of the vector unit in addition to explicit vector instructions within the code.

3 IN-SITU VISUALIZATION USING PARAVIEW CATALYST

Catalyst is a library component of the ParaView scientific visualization application that enables applications to perform in-situ visualization. In order to use Catalyst, application developers create an integration that converts their applications' data structures to and from equivalent data structures in Catalyst. They also insert a call to the Catalyst library when visualization output is desired, normally at some fixed interval of simulation time-steps. In-situ visualization is executed across all MPI ranks on which the simulation is running, and the data for visualization is immediately available in the process memory which is being shared by Catalyst and the simulation. The output from Catalyst is typically a set of parallel rendered images.

An additional consideration when using Catalyst is how to tell Catalyst what to do when the simulation passes control over to Catalyst. There are three options available. The easiest option is to export a Python script from the ParaView GUI when a surrogate for the larger simulation problem is loaded. The export option does not allow for robust error checking in terms of inputs, nor a mechanism to cleanly report error conditions back to the simulation. The second option is to directly write Python scripts to control Catalyst for a specific purpose. Python scripting offers flexibility, but requires time and more detailed knowledge of the internal Catalyst pipeline API. The third option, which is the least used and documented, is to write a C++ program to control Catalyst. This option trades the flexibility of Python for the better performance of compiled C++.

4 CONTROLLING CATALYST WITH PHACTORI

A data flow language called ParaView Higher level AbstraCTiOn scRipting Interface (Phactori) was created during prior work with Catalyst. Phactori allows users to express their visualization as a data flow that starts from the simulation input to Catalyst. A Phactori data flow can include chains of common data visualization operations such as clip, slice, and contour. The data flow typically includes various camera setups, data color mapping, image sequence output specification, and data product output specification (e.g. output the results of a slice, clip, exterior surface filter, or contour as a 3D dataset). A text file containing Phactori commands is then parsed by a Python implementation of the Phactori language at Catalyst run-time. The parsing and unit-testing of the Phactori implementation ensure that analysts can have confidence Catalyst will produce the desired results each time the simulation passes control to it. In contrast, the export of scripts from ParaView is prone to subtle errors born out of the complexity and completeness of ParaView, such as using data which does not match the simulation output thus getting scripts that do not work, constructing a highly complex workflow that has mysterious problems, creating a programmable filter which crashes on the in-situ callback, doing operations which turn out not to scale well or bloat memory usage, etc. Phactori also passes error information back to the simulation in the event of unexpected behaviors. Phactori makes generating multiple sets of images from different perspectives and variable colorings easier than using the ParaView GUI to export Python scripts. Also, the in-situ visualization development workflow is more tractable using Phactori, allowing for rapid "simulation run/adjust the visualization script" cycles at low node counts.

Recognizing the need for a higher level of abstraction for defining in-situ operations on HPCs is not unique to this project. Cinema (reference) is a tool somewhat similar in nature to Phactori. The general nature of Cinema is to produce a database of data products including images and 3D output. Cinema includes techniques for specifying multiple camera angles, 3D data products, and composable image-like data fields where the mesh cell or point data is stored rather than fixed color values. It is easy to envision Phactori specifying Cinema databases as an output format (either using Phactori or Cinema itself to specify camera angles) or having Cinema employ a Phactori script to produce a particular data product.

5 CATALYST IN NALU

For some codes, in particular Nalu and Sierra (the Sandia Sierra Framework), Catalyst is integrated via a run-time plugin (using dlopen()) to access the program at runtime.

The main advantages to this strategy are (1) If not used, the binary code is not loaded, so the code footprint is zero and (2) the plugin can be managed in a separate build system. A separate build system is sometimes advantageous because it offloads the complex Catalyst build system development and maintenance from the simulation development team, obviates integrating complex Catalyst building into complicated legacy non-cmake build systems, and keeps build and deployment issues from Catalyst from affecting the simulation and vice versa. The main disadvantages are (1) the plugin is usually not built automatically along with the simulation engine and (2) it forces the use of dynamic loading. On some HPCs static binaries are distributed rapidly using specialized communication setups, while shared library loads can require all nodes to independently open and read the shared libraries and this process scales poorly at high node counts.

Over time, this plugin has been moved into the Sandia Engineering Analysis Code Access System (SEACAS) (see https://github.com/gsjaardema/seacas), in the IOSS (input/output system) library. The eventual plan is to make this plugin a part of the standard SEACAS build.

Figure 2 is an example image generated from Nalu via in-situ visualization on knl nodes.

6 CATALYST IN SPARC

Prior to Catalyst integration into SPARC, Catalyst was only integrated into simulation engines which used unstructured grids. SPARC can do both unstructured and structured grids, but goal was to do in-situ visualization with the structured grids, so new work was necessary to map the SPARC structured data into VTK/ParaView structured data formats.

At project start time SEACAS handling of structured grids in parallel was not complete so Catalyst was directly integrated into SPARC, i.e. not via a plugin. Eventually this integration will move to a plugin architecture as well. Since there was not a run-time plugin restriction, a statically linked version of the Catalyst libraries was created (with considerable effort) which allowed examination of differences between the startup times and run times of the statically linked versions of the code and the dynamically linked versions of the code on Trinity. As the structured version of this code is moved into SEACAS it is desirable to maintain the options to build statically, dynamically, and as a run-time plugin.

Figure 1 is an example image generated from SPARC via in-situ visualization on KNL nodes.

7 DEVELOPING VISUALIZATIONS USING SMALL SCALE SIMULATIONS

One of the difficulties of in-situ visualization is generating an appropriate visualization setup prior to generating output data. Given the size of the simulations performed as part of this effort, there was only the opportunity to run the simulation at scale a handful of times.

One important technique for mitigating this limitation is using either small-scale versions of the simulation problems or small-scale datasets for generation of the visualization input files. During the final simulations at scale, there was never a case where a visualization script which worked at lower scales failed to work properly at higher scales.

8 RESOURCE IMPACT OF IN-SITU VISUALIZATION ON SIMULATION

An important issue is what resources the in-situ visualization uses relative to the simulation calculations. The following questions are used to address this issue. It is worthwhile to consider both the absolute results and the relative resources used by the simulation calculation and the in-situ visualization calculation.

(1) What is the memory impact of the additional binary code?
(2) What is the memory impact of the run-time operation of the in-situ visualization?
(3) How is the initial simulation startup time affected by the presence of the in-situ visualization code (i.e. how long to get in to "main()", loading all the static code and binary libraries)?
(4) How long does the simulation take to get from initial startup to the first time-step? (mesh decomposition, mesh loading, or restart time)
(5) How long does the simulation code take to execute its first time-step?
(6) How long does the simulation code take to execute subsequent time-steps?
(7) How long does the in-situ visualization take to render at first callback (thus including any additional dynamic library loading)
(8) How long does the in-situ visualization take to render at subsequent callbacks?

For SPARC and Nalu in this particular case study, memory was not an issue, as engineers were dividing the mesh into a large number of nodes relative to the size of the mesh, with only a few thousand or tens of thousands of elements per process, and only a few million or few hundred thousand cells per node. Therefore (1) and (2) are not considered in this paper, though it is recognized that these questions are of considerable interest to Nalu, SPARC, other simulation, and in-situ developers. Using profiling tools or instrumenting the code to obtain interesting fine-grained information such as how much time and how much memory are used by particular operations (e.g. exterior surface, slice, clip, contour, rendering, image composition) has not yet been done but is an interesting near-term objective. Qualitatively it appears that memory usage by Catalyst is not a show-stopping issue but may require mitigation strategies in various circumstances, such as working to reduce build size and avoiding in-situ operations which produce results which are of a similar size to the whole simulation mesh.

Question 3, initial startup time comes into play because time to load binary runtime libraries can become large at scale. In fact, this time was not too noticeable in any situation up to at least 1000 nodes, but it can be very noticeable at 5000 nodes. Prior to this milestone Catalyst was used exclusively as a shared object binary build or as a run-time plugin, both of which require loading of shared objects at run time. This HPC compute platform (Trinity) has special capabilities to distribute the executable as part of an Message Passing Interface (MPI) job faster than dynamic binaries which are loaded on demand by every process in parallel. One option for mitigating this is building everything statically; a difficult task as previously mentioned.

If building a single executable statically is not an option, then reducing the number of libraries and files that need to be loaded will reduce the startup time. Catalyst supports compiling "Editions", which only compile those components necessary for producing a given visualization instead of compiling and linking all of Catalyst [3]. Python supports freezing, which compiles all the python scripts and shared objects necessary for a given python script to run into a single executable [2].

Moving to Catalyst Editions and freezing python greatly reduced the number of libraries from 231 to 35. Linking an entirely static build of the Catalyst libraries also reduced this problem.

One show-stopping issue we overcame was at question 7, the first in-situ callback. It seems obvious in retrospect, but if the in-situ engine is producing serial output of any form to standard output or to one file, including warnings, then the system would work reasonably well on up to about 500 nodes, but grind to a halt at 1000 or 2000 nodes. Work was necessary to guarantee no warnings or other serial output on a few hundred node count, and when all this serial output was eliminated scaling to 5000+ nodes was possible.

Table 1 essentially is the effort to answer questions 3-8 for SPARC on this HPC platform. We were executing 50 SPARC timesteps for every Catalyst callback. Notable observations are: 1) In-situ visualization results are produced at all scales in all cases. It is useable (but not necessarily preferable) in all build configurations. 2) Static builds make a large difference in binary load times. For the worst shared object case, these load times could take a noticeable percentage of the overall run time allocation. 3) Mesh decomposition time dominates over shared library load time, even in the case with

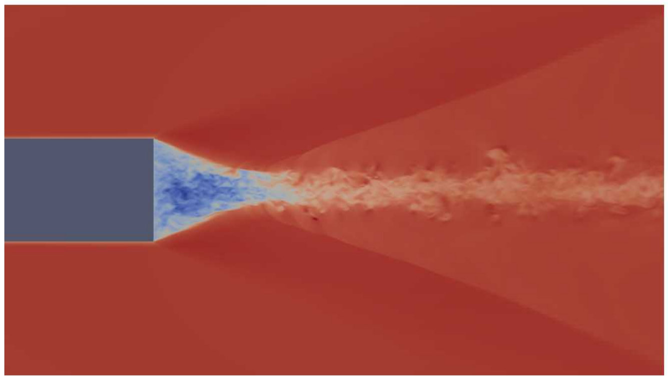

Figure 1: In-situ generated image from a SPARC 3D simulation of a cylinder in an airflow. Slice, colored by X velocity.

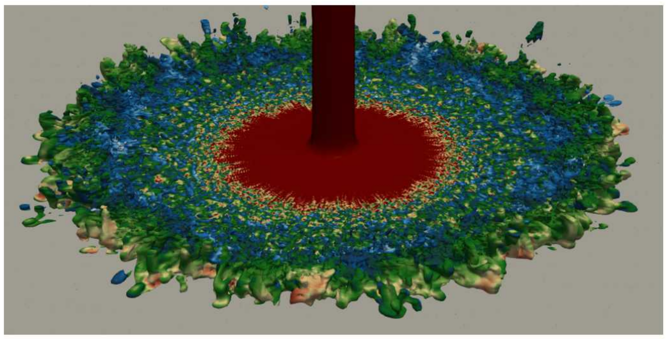

Figure 2: In-situ generated Nalu simulation image of a hot air jet impinging on a surface. Isosurface at nonzero velocity, colored by temperature.

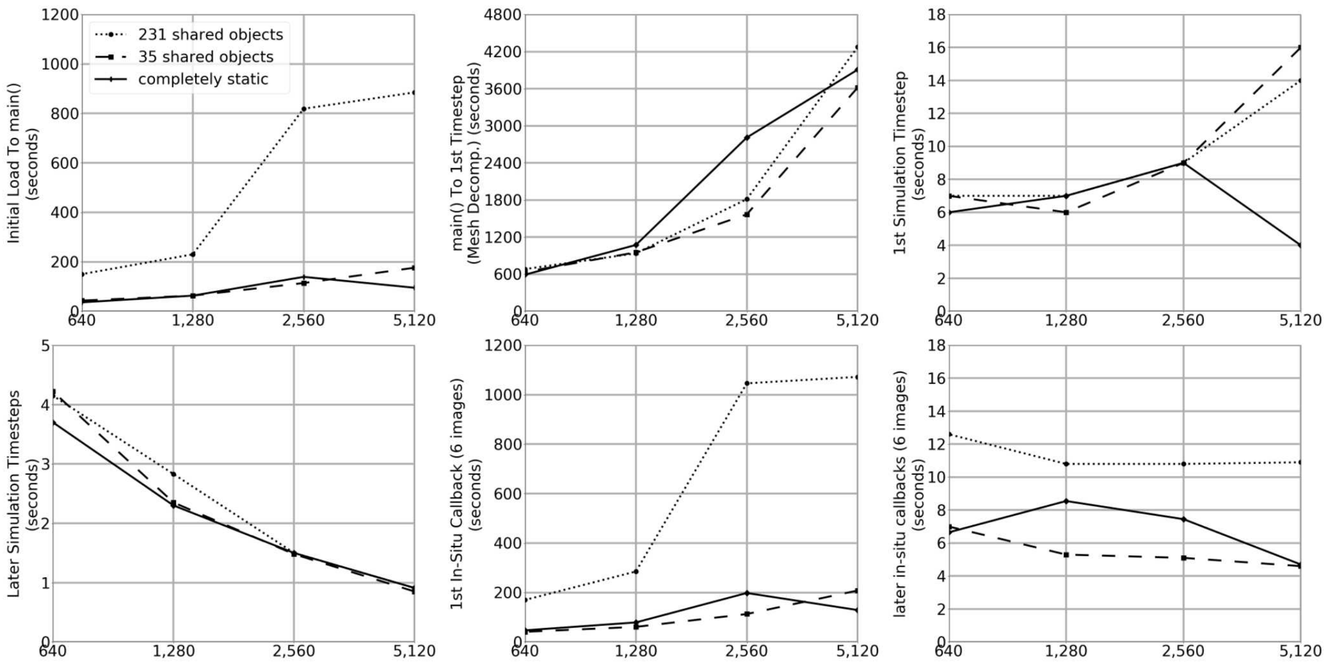

Figure 3: Strong Scaling Of Startup And Run Times (X Axis is number of ATS-1/Trinity Knight's Landing Nodes)

231 shared libraries. 4) SPARC execution time per-timestep was about the same in all cases 5) Besides faster startup, the rendering times were twice as fast in static builds. This was a surprise, as the expectation prior to the timings was that load time costs might be high for shared libraries but the additional overhead for runtime operations like function calls would be minimal. It is unknown why this slowdown occurs. One theory is that there is a penalty for function calls which cross a library boundary and there are 231 libraries.

Runs at 640, 1280, 2560, and 5120 nodes with the same mesh provide data for a strong scaling study with the results shown in Figure 3. (No comprehensive weak scaling study was conducted due to issues with getting simulation runs through the HPC queue into execution in a timely fashion.) The "Later Simulation Timesteps" plot shows the advantageous scaling speedup effects during the simulation execution. The "Later In-Situ Callbacks" plot shows that the time taken by the in-situ visualization is essentially flat as the node count increases, suggesting that simulation engineers will want to make in-situ callbacks less frequently at higher scales to ensure that the in-situ callbacks do not utilize a high percentage of the overall run time. The startup/first-time plots show how increasing node counts most adversely affect the 231 shared object build of SPARC with an interesting jump in time from 1280 nodes to 2560 nodes. Note that the mesh decomposition time dominates startup time at all node counts for all builds. According to SPARC developers, newer SPARC builds may reduce this time significantly.

Table 1: Timings at 5120 Nodes (Seconds)

timed item	**231 shared objects**	**35 shared objects**	**completely static**
initial load (to main)	885	176	95
main to first SPARC timestep (mesh decomposition)	4275	3619	3908
first SPARC timestep	14	16	4
later SPARC timestep	0.91	0.85	0.91
first in-situ callback (6 images)	1072	208	129
later in-situ callback (6 images)	10.9	4.6	4.7

9 CONCLUSION

The work during the LSCI milestone has demonstrated that Catalyst is capable of producing high-quality visualizations in-situ when coupled to two simulation codes running at scale, with acceptable impact on the simulation performance.

Special attention was given to how the software was compiled, and the load-time consequences of having many dynamic libraries linked into the simulation executables. As the simulation size scaled-up, it became necessary to use frozen Python to mitigate load-time issues caused by large numbers of Python modules.

Using smaller scale simulation problem meshes and the Phactori data flow language was essential in developing useful visualizations. Iterations of simulation executions were very limited at full scale.

ACKNOWLEDGMENTS

The authors would like to express their thanks to Micah Howard and Sam Browne for their help with SPARC. We extend our gratitude to Kurt Ober and Ron Olfield for excellence in project initiation and management, and give thanks Dave Karelitz for many helpful suggestions and technical assistance.

REFERENCES

[1] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. 2015. ParaView Catalyst: Enabling In Situ Data Analysis and Visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV2015).* ACM, New York, NY, USA, 25–29. https://doi.org/10.1145/2828612.2828624
[2] Utkarsh Ayachit and Pat Marion. 2013. Beat the heat by freezing Python in ParaView. https://blog.kitware.com/beat-the-heat-by-freezing-python-in-paraview
[3] Andy Bauer. 2014. ParaView Catalyst Editions: What Are They? https://blog.kitware.com/paraview-catalyst-editions-what-are-they
[4] Andrew Bauer, H Abbasi, J Ahrens, H Childs, Berk Geveci, S Klasky, Kenneth Moreland, P O'Leary, Veena Vishwanath, Brad Whitlock, and E. Wes Bethel. 2016. In Situ Methods, Infrastructures, and Applications on High Performance Computing Platforms. *Computer Graphics Forum* 35 (06 2016), 577–597. https://doi.org/10.1111/cgf.12930
[5] S. Domino. 2015. *Sierra Low Mach Module: Nalu Theory Manual 1.0.* Technical Report SAND2015-3107W, Sandia National Laboratories Unclassified Unlimited Release (UUR). Sandia National Laboratories. https://github.com/spdomin/NaluDoc
[6] Micah Howard and Srini Arunajatesan. 2016. SPARC v. 8/17/2016, Version 00.