Coupling the Uintah Framework and the VisIt Toolkit for Parallel In Situ Data Analysis and Visualization and Computational Steering

Allen Sanderson[1], Alan Humphrey[1], John Schmidt[1], and Robert Sisneros[2]

[1] Scientific Imaging and Computing Institute, University of Utah, Salt Lake City, UT 84112, USA {allen,ahumphrey,jas}@sci.utah.edu
[2] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, Il 61801, USA sisneros@illinois.edu

Abstract. Data analysis and visualization are an essential part of the scientific discovery process. As HPC simulations have grown, I/O has become a bottleneck, which has required scientists to turn to in situ tools for simulation data exploration. Incorporating additional data, such as runtime performance data, into the analysis or I/O phases of a workflow is routinely avoided for fear of excaberting performance issues. The paper presents how the Uintah Framework, a suite of HPC libraries and applications for simulating complex chemical and physical reactions, was coupled with VisIt, an interactive analysis and visualization toolkit, to allow scientists to perform parallel in situ visualization of simulation and runtime performance data. An additional benefit of the coupling made it possible to create a "simulation dashboard" that allowed for in situ computational steering and visual debugging.

Keywords: In Situ Visualization · Runtime Performance Data · Visual Debugging · Computational Steering.

1 Introduction

When discussing techniques for in situ visualization, the focus is typically on the infrastructure for efficiently utilizing the simulation data generated by the application. However, other more ephemeral data is also of interest. We define ephemeral data as that which is generated by the application, optionally written to disk for post hoc analysis, but not otherwise saved or utilized by the application in subsequent time steps. Examples of ephemeral data include in situ simulation analysis (i.e., analysis routines that are directly incorporated into the application) and runtime performance data. For the work presented in this paper, it is this data and the infrastructure required for efficiently utilizing it that are of primary interest.

We also introduce the concept of a simulation dashboard to present parameters and resulting performance data that may be used to control or drive the simulation. The parameters are supplied to the application by the user as part

of the *Uintah Problem Specification*. Measured performance data is rarely written to disk and is therefore an excellent target for incorporation in an in situ framework.

In the context of in situ terminology [8], we are interested in data access, and more specifically how the application makes data available to the in situ infrastructure. Because of the heterogeneity of the data generated and parameters utilized by the application, different approaches were required. In some cases, direct access was possible. In other cases, data access was accomplished via a series of lightweight data structures and wrappers. The lightweight data structures replaced the previous infrastructure within the Uintah framework, and the wrappers were used when it was necessary to work within the current infrastructure.

2 Background

In this paper, we present a hybrid approach that intersects the state-of-the-practice in in situ data analysis and visualization with the use of collected runtime performance data. In this section the foundational background for in situ techniques is outlined, followed by a description of the state-of-the-practice in utilizing diagnostic data in general, of which runtime performance data is a subset.

2.1 In Situ Data Analysis and Visualization

The integration of visualization in HPC workflows relies heavily on supported software. The visualization software staples at HPC centers are those that are open-source, actively developed, and scalable. The difficulties arising from large-scale computing as well as the expected inadequacy of postprocessing due to overwhelming data sizes are well documented [13, 12]. So far, the accepted answer for these challenges, with respect to data analysis and visualization, is in situ processing [4, 19, 8].

The standard software suites for data analysis and visualization, VisIt [9] and ParaView [16], offer in situ libraries [11, 5, 28] in addition to client/server execution models, batch mode processing, and a complete scripting interface. In this work, we instrument the Uintah framework using VisIt's in situ library, *libsim* [28]. Although not quite a middleware approach [10], our implementation does indeed alleviate some of the burden associated with libsim's tightly coupled instrumentation. We direct the interested reader to a state-of-the-practice report [6] for additional, general details regarding in situ techniques.

2.2 Utilization of Diagnostic Data

At the two ends of the HPC community are the large computing hardware and the users who run applications on them. Between are the systems administrators and engineers who maintain that hardware and are focused on maximizing

throughput of users' codes, and is critical given the cost of operation of these large machines. Monitoring system diagnostic data can benefit this effort in a number of ways, and there are many systems in place to do so [20, 1, 3, 2].

The tendency toward heavily visual systems as well as the common visual paradigms bolster motivation for inclusion of system visualizations we use in this work. In particular, we integrate a visualization of the high-speed network fabric favoring a machine layout [27] over purely graphical representations [15, 25]. A defining differentiation of this work is its in situ deployment. The tools and approaches mentioned above rely on mechanisms of collecting, storing, and retrieving stored data external to the application; see OVIS [7] for a representative framework. We sidestep this approach and do it within the application because these external tools can not tap into Uintah's custom debugging streams in the way presented in this paper, which at times is the best and only way to get Uintah-specific diagnostic data.

Furthermore, in this case there is an uncommon, but natural coupling of diagnostic data to scientific simulation data that has allowed us to create novel visualizations leveraging this coupling. That is, we view system diagnostic data in both spatial contexts: job layout on the machine as well as compute layout on the simulation mesh. While there are examples of recent work visualizing diagnostic data on the mesh [14], even in situ [29], frameworks serving such data are still reliant on an additional data management solutions, i.e. databases. In such situations the flexible, user-defined data we serve in situ is not as practical.

3 Methods

One of the objectives of this research was to efficiently communicating ephemeral data, data that is typically not visualized or otherwise presented to the user in an in situ framework. To accomplish this objective, lightweight data structures and wrappers were developed and deployed that allowed for direct memory access to the data. The lightweight data structures replaced existing data collection and storage structures in Uintah to reference simulation parameters and global analysis data in a minimally intrusive manner.

3.1 Per-Rank Runtime Performance Data

Per-rank runtime performance may be collected by different parts of the application infrastructure at different stages of the execution path and written to disk for post hoc analysis. Within the Uintah infrastructure runtime performance data is collected by the simulation controller, memory manager, scheduler, load balancer, AMR regridder, and data archiver (I/O). The data collected range from memory usage to MPI wait times. In addition, the Uintah application also collects runtime performance data that is specific to the simulation physics, such as the solver performance.

Gaining access to runtime performance data for in situ communication required that the data be collected centrally and in a generic manner. To facilitate

the centralized collection of data, we leveraged Uintah's Common Component Architecture [17]. The simulation controller is the central component in Uintah and interfaces with all of the other underlying infrastructure components (Scheduler, Load Balancer, I/O, etc.) as well as the application component. As the central component, the simulation controller owned and shared the runtime performance data collected with all of the other components.

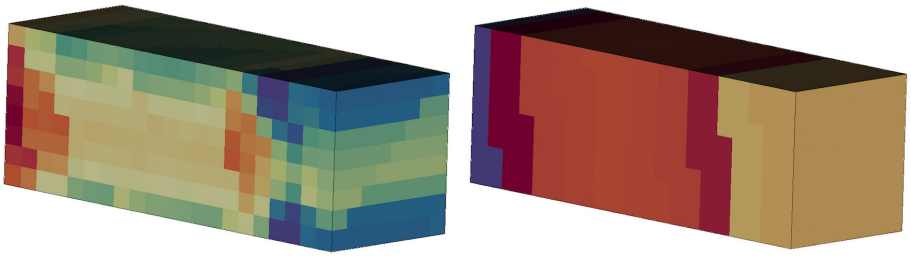
Data Collection To manage the collecting of the per-rank runtime performance data, we created an extended map class. Maps are associative containers that are a combination of a key value and a mapped value. Our extended class maintained the core map abstraction while allowing for generic indexed based access. This allowed the Uintah infrastructure to continue to have the flexibility to utilize a key value, in this case an enumerator specific to the runtime performance measure. When the data needed to be communicated to VisIt, the indexed-based access method was utilized. Iterators were not used as they required the key value type, which was not available to middleware utilized between Uintah and VisIt.

The collecting of runtime performance data was done on a per-rank basis, allowing utilization of our extended map class. However, it was also useful to present global values. To address this need, global MPI reduce operations were built into the extended map class. The Uintah infrastructure also supports multi-threading, which required a reduction over the shared resource on each node. To facilitate this reduction, the extended map class also incorporated a MPI reduce that utilized a split communicator based on the ranks on each node (i.e., all ranks on a node had the same communicator). These reduced values could then be accessed as additional node-level mapped values in our extended map class.

Spatial Granularity Now that the runtime performance data could be collected and reduced at different processor levels, we address the issue of visualizing the data at different spatial granularities. The global values could be shown in a tabular fashion or as a series of strip charts as part of the simulation dashboard (see Section 3.2). Of greater interest was the ability to visualize the runtime performance data.

The Uintah infrastructure utilizes a patch-based structured grid. Each patch contains multiple cells and is assigned to a specific rank. This patch-rank assignment was used to map the runtime performance data to the simulation mesh with all patches on the same rank displaying the same value (Figure 1a). In a similar fashion, the per-node runtime performance data was also mapped to the simulation mesh with all patches on the same node displaying the same value (Figure 1b).

The ability to map the runtime performance data to the simulation mesh at different granularities allowed us to check the mesh layout against the raw performance statistics reported (e.g., load balancing, communication patterns, etc.) and provided insight not usually possible. For instance, the ability to see the patch layout on particular ranks gave insight into the load balancer assignment.



(*a*) (*b*)

Fig. 1. A simple visualization of (a) the per-rank task execution times for 192 ranks and (b) the per-node task execution times for nine nodes mapped onto the simulation mesh (Blue - fast, Red - slow). Of the nine nodes, eight nodes were 20-core nodes, and one node was a 12-core node. Thus the summed time for the 12-core node, in blue is substantially less than the other nodes.

It was also possible to gain insight by visualizing runtime performance data using the physical machine layout. For instance, the runs presented were done on a heterogeneous HPC system comprised of 253 - 12 core nodes, 164 - 20 core nodes, and 48 - 24 core nodes (7468 total cores) connected using an InfiniBand network (For Figures 1 - 4, eight 20 core nodes and one 12 core node were used). To obtain the network layout, the 'ibnetdiscover' command was utilized to discover the InfiniBand subnet, which was then visualized based on the switch connections (Figure 2).

Via a series MPI queries, a mapping between the nodes and cores being utilized by Uintah and the machine layout was created (Figure 2).

Utilizing this mapping and the ability to easily collect rank based runtime performance data using our new map class, performance data could now be visualized using the machine layout. As with the simulation mesh, both the per-rank and per-node runtime performance data could be mapped directly to the machine layout (Figure 3-4). Visualizing the runtime performance data on the machine layout gave insight into how the machine layout affected the runtime performance, for instance, which ranks or tasks were taking the most time for MPI waits or parallel collections.

Per-Task Runtime Performance Data The Uintah runtime infrastructure utilizes dynamic task scheduling for its execution [22]. For a typical CFD simulation, 10 to 50 tasks were assigned uniformly to each patch across the domain. These tasks may be created by the simulation (e.g., solver) or the infrastructure (e.g., I/O) and are able to monitor their runtime performance in the form of execution and wait times. These times can be saved as a series of per-task/per-rank text files for later post hoc analysis, but previously were not otherwise visualized.

To facilitate access to this data for in situ visualization, a map structure utilizing two keys was constructed. The first key mapped the task and the second key mapped the patch. Taking advantage of Uintah's one-to-one mapping

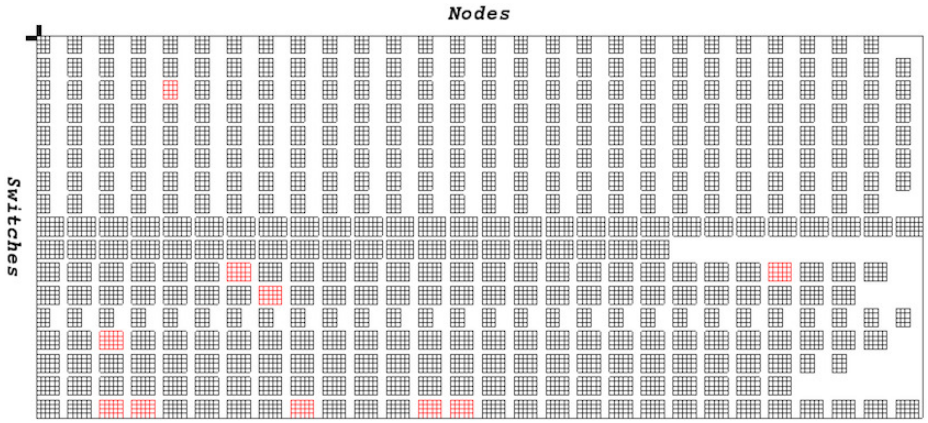


Fig. 2. A visualization of the machine layout based on the switch connections. Each row represents a switch (17 total), each block represents a processor node, and each cell within a block represents a core. Highlighted in red are the nine nodes and 192 ranks used by the simulation.

between patches and tasks visualizing the per-task runtime performance data in situ on the simulation mesh was a straightforward process. Visualizing the per-task runtime performance data on the simulation mesh gave users the ability to understand hot spots and their possible cause, for instance, the time between the end of the execution of one task to the beginning of the execution of the next task for each thread (Figure 5a).

If desired, the per-task runtime performance data could also be summed over all tasks for each patch. For instance, the sum of task exection times utilized by Uintah's load balancer's dynamic cost modeler [18], that previously, had not been possible to visualize (Figure 5b). With the implementation of this map, users were able to visualize these sums and see load imbalances.

Initially, the per-task runtime performance data was accessed only for in situ visualization. However, after seeing the utility of visualizing this data, users requested that the data be stored in Uintah's data warehouse [21] so it could be written to disk for post hoc visualization and analysis. As such, a task was created to store the collected data in the data warehouse. Which being a task, it too could monitor its performance. Thus, it was possible to monitor the monitoring.

3.2 The Simulation Dashboard

When performing in situ visualization, it is often desirable to have access to data that is global to the simulation, such as the reduced runtime performance data, all of which may be written to disk for post hoc analysis. However, it is often impractical to analyze a series of text files while performing in situ visualization and therefore a simulation dashboard was created.

As part of VisIt's *libsim* [28], an application can provide a Qt [24] Designer XML file defining a series of widgets that can be used to generate an interactive

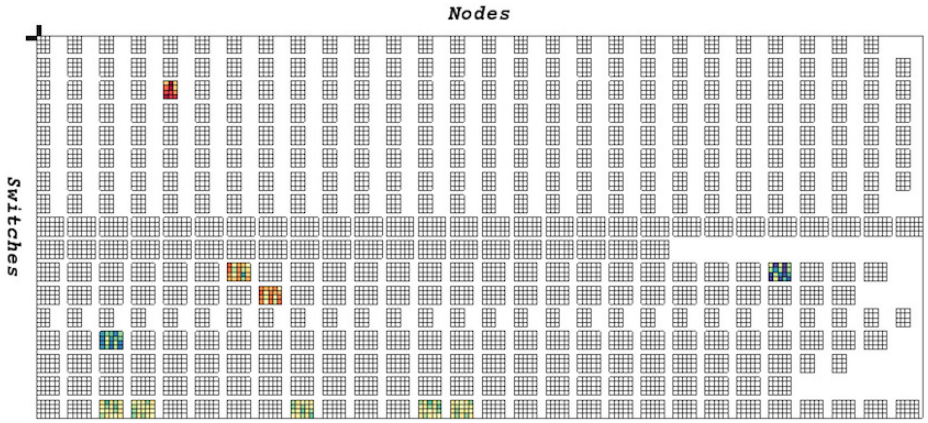


Fig. 3. A visualization of the per-rank task execution time for 192 ranks mapped onto the machine layout (Blue - fast, Red - slow). Compare the figure to the same times mapped onto the simulation mesh in Figure 1a.

user interface on the fly. This user interface forms the basis for the simulation dashboard and is fully customizable by the user. Communication from the application to VisIt was accomplished through an interface function that, in general, requires little more than the widget name and a value. Communication from VisIt to the application was accomplished via callback functions that were registered by the application during the in situ initialization process.

General Simulation Data When running a simulation, a plethora of data can be reported to the user. Some of the data may be read-only or may be read/write. When the data is read/write, a callback function is required to update the values. Examples of read-only data are the simulation time and time increment (Figure 7b(1)). Read-only data also includes wall time intervals (Figure 7b(3)) and AMR grid information (Figure 7b(6)). Examples of read/write data are the simulation time increments in which the user can change most all of the values (Figure 7b(2)).

Access to the data was often directly available from the Uintah component via a function call and required no special handling. Although simple, it required a series of singular calls to VisIt, as opposed to looping through a list of lightweight data structures pointing to the data (as described below).

Uintah also contains in situ methods to calculate both general, (e.g., min/-max) and simulation specific values, (e.g., total momentum) that are stored in Uintah's data warehouse. To retrieve the values for in situ usage, a generic lightweight structure was created to hold the required parameters; the variable name, material (Uintah supports multiple materials), and level (Uintah supports adaptive mesh refinement), the variable label, which contains the variable type, (e.g., scalar or vector) and the variable reduction type, (e.g., min, max, or sum).

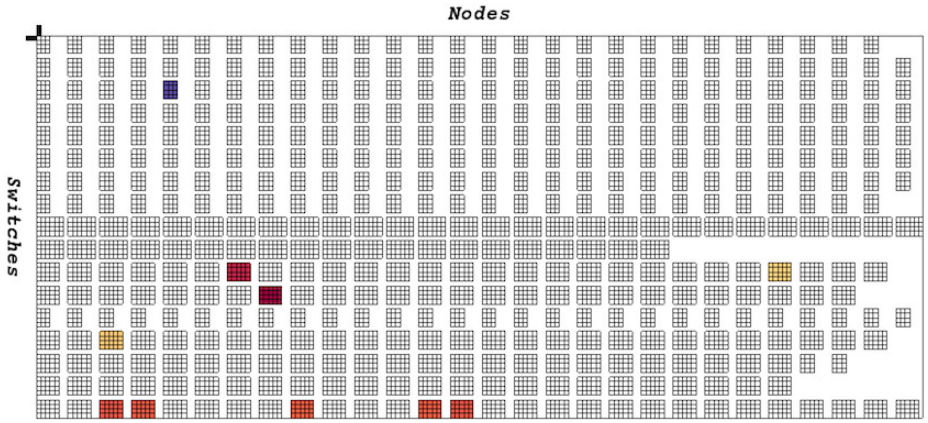
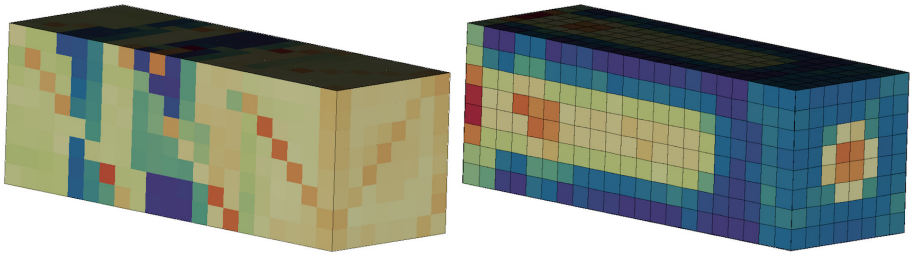


Fig. 4. A visualization of the per-node task execution times mapped onto the machine layout (Blue - fast, Red - slow). Compare the figure to the same times mapped onto the simulation mesh in Figure 1b.



(*a*) (*b*)

Fig. 5. A simple visualization of (a) the per patch task wait times and (b) the per patch load balancer costs for 1344 patches (Blue - fast, Red - slow).

An application developer coding an analysis module utilized the lightweight structure to wrap the values of interest. This structure was then registered with the in situ middleware when the analysis module was initialized. At the end of each time step, the situ interface looped through each registered structure, retrieved the analysis data from the data warehouse, and passed it on to VisIt to display (Figure 7b(5)).

Strip Charts In addition to a tabular display of analysis data (Figure 7b(5)) and runtime performance data (Figure 8a(1-3)), visualizations over time are possible using a series of multivalued strip charts (Figure 6). Although integral to the simulation, the strip charts were not integrated into the simulation dashboard but were instead integrated into VisIt's simulation control window for generiazed use by expanding VisIt's in situ interface to manage strip charts and utilized the QWT [26] library's 2D plotting widget.

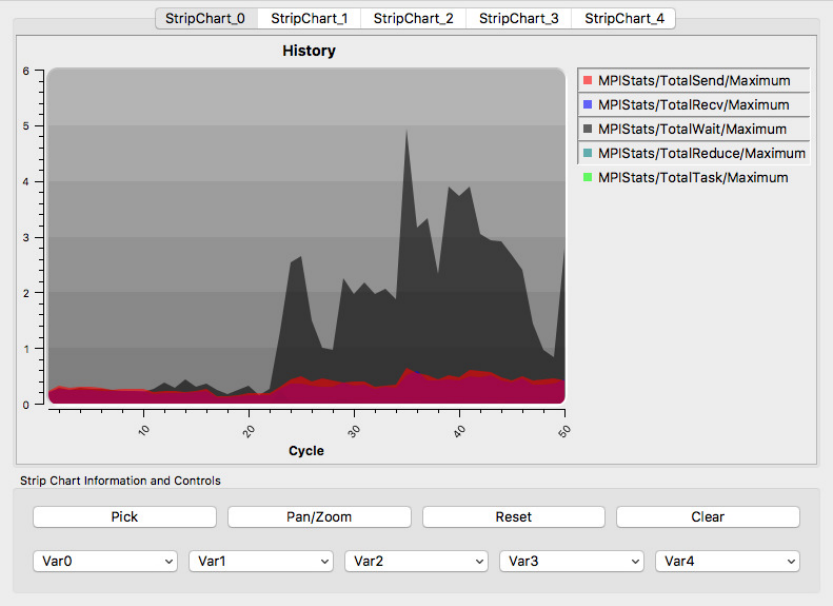


Fig. 6. A strip chart showing MPI runtime performance stats over 50 time steps.

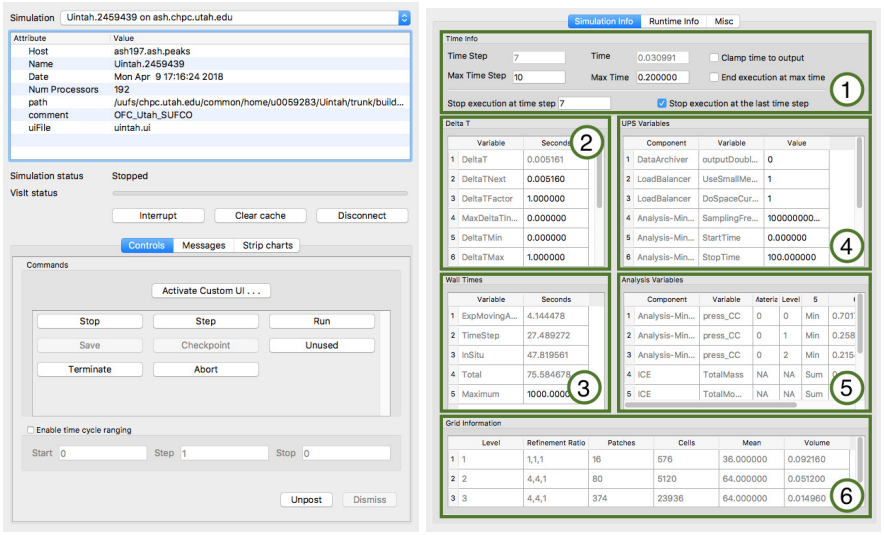
Computational Steering Computational steering allows the user to modify simulation parameters on the fly to change a simulation's outcome [23]. As noted above, data such as the simulation time increment is read/write and thus can be interactively modified by the user. Although computational steering is not typically available in in situ frameworks, it is a natural extension. One of the goals of in situ visualization is to provide real-time feedback to the user, which can then be used to interactively steer the computation.

Uintah simulations are defined via a series of parameters that are part of the *Uintah Problem Specification*. These parameters are used to describe the initial simulation and are typically not changed during the life of the simulation, but may be tuned beforehand via a series of exploration runs.

Changing some of the variables may have side effects, such as requiring new tasks to be performed and forcing the task graph to be recompiled [18]. To facilitate the modification of these parameters in situ, a lightweight structure was created to hold the parameter name, type description, memory reference, and a flag indicating whether the task graph would need to be recompiled.

Similar to an analysis component, an application developer coding a simulation component utilized the lightweight structure to wrap the parameter. This structure was then registered with the in situ middleware when the component was initialized. At the beginning of each time step, the situ interface looped through each registered structure, retrieved the simulation parameter, and passed it to Uintah, where it was utilized for the subsequent time step (Figure 7b(4)).

Visual Debugging As part of VisIt's Simulation window, it is possible for the user to issue commands to the simulation to stop, run, step, save, checkpoint, abort, etc., the simulation (Figure 7a). These commands are defined by the simulation and give the user some level of control. However, when debugging, the user may want finer grained control. For instance, for a simulation that is known to crash at particular time step, the user may want to stop the simulation just before that time step and begin debugging (Figure 7b(1)). This ability is especially important because the overall state of the application would be maintained as long as possible, thus not perturbing the debug condition.



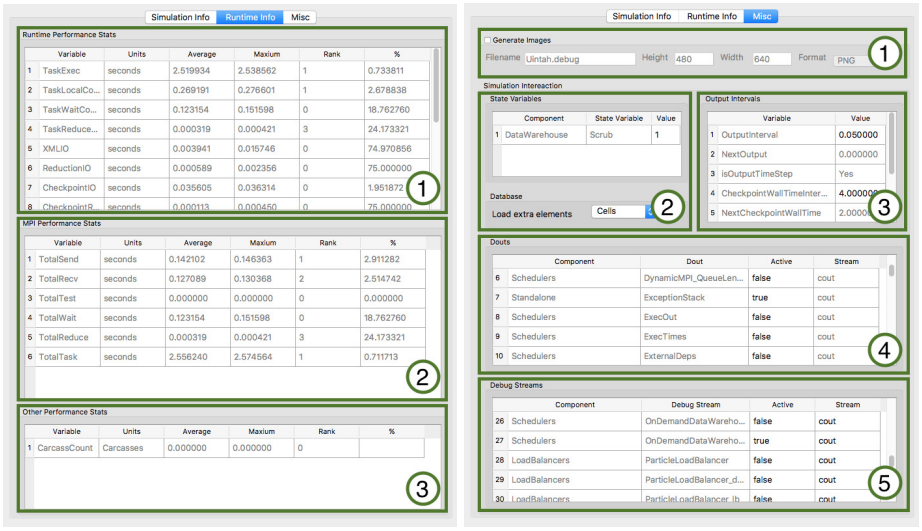
(a) (b)

Fig. 7. (a) VisIt's simulation control window, which allows the user to issue commands to the simulation. (b) The simulation dashboard with general information; (1) simulation times, (2) time stepping, (3) wall times, (4) Uintah problem specification parameters, (5) analysis variables, and (6) AMR information. Values shown in light gray are read-only, whereas those in black may be modified by the user.

Debugging may require changing the output frequency, turning on debug streams, (e.g., text-based output), or exposing intermediate data values. To facilitate access to the debug streams, they were rewritten to self-register when constructed. Once registered, the debug streams can then be accessed by the in situ interface via a centralized list (Figure 8b(4-5)).

To expose state variables, the lightweight structure developed for exposing the *Uintah Problem Specification* parameters was utilized (Figure 8b(2)). One such state variable controlled the scrubbing of intermediate variables from Uintah's data warehouse. To reduce memory, intermediate variables not needed for checkpointing are deleted at the completion of a time step and not normaly vi-

sualized. However, visualizing these variables can be an important part of code verification.



(*a*) (*b*)

Fig. 8. The simulation dashboard with (a) runtime performance data; (1) infrastructure, (2) MPI, and (3) application specific, and (b) other information; (1) image generation, (2) state variables, (3) output and checkpointing frequency, and (4) and (5) debug streams. Values shown in light gray are read-only, whereas those in black may be modified by the user.

4 Results

The main result of coupling the Uintah Framework with the VisIt toolkit is allowing for Uintah developers and users to visualize and explore empheral data. For instance, Uintah infrastructure developers using per-rank runtime performance data were for the first time able to visualize the load balancing on the simulation mesh and machine layout insteading of resorting to text output. A Uintah application developer, using the per-task runtime performance data, was able to validate that a particular computational task had a uniform execution regardless of the boundary and mesh configuration. A Uintah user, who previously did a series of parameter runs, instead was able to use the in situ computataional steering tools to change input parameters such as the time stepping on the fly, thus reducing the time spent on parameter exploration runs.

Another result of the coupling was that as data was exposed in different scenarios, Uintah developers and users requested new diagnostics. For instance, our map class was initially written for Uintah's runtime infrastructure but then

was also used to separately collect MPI-specific performance measures. Later, another map was utilized by the application components to collect simulation specific runtime performance measures. The generic nature of the map class made adding and exposing data for in situ visualization a very simple process, taking a matter of minutes.

Similarly, when the machine layout view was developed, Uintah developers requested new diagnostics that only made sense with that layout, such as the number of patches and tasks per rank. This request was soon followed by others for the ability to interactively query physical node properties such as the on-board memory and temperature. At the same time, the lightweight wrappers made adding and exposing parameters for computational steering a very simple process. Users could add the needed code in a matter of minutes, re-run their simulation, and interactively explore while asking "what if questions."

From a more practical point of view, code maintainability became easier when it was necessary to modify the Uintah infrastructure, whether for new data collections or for managing debug streams. At the same time, it became clear that visualization developers needed to have a more integral role in the development of the Uintah Framework.

5 Conclusion

In this paper, we have presented work to couple the Uintah Framework with the VisIt toolkit to allow scientists to perform parallel in situ analysis and visualization of runtime performance data and other ephemeral data. Unique to the system is the ability to view the data in different spatial contexts: job layout on the machine and the compute layout on the simulation mesh. We have also introduced the concept of a "simulation dashboard" to allow for in situ computational steering and visual debugging. The coupling relied on incorporating lightweight data structures and wrappers in a minimally intrusive manner into Uintah to access the data and communicate it to VisIt's in situ library.

In several cases, the Uintah developers found the infrastructure changes to be beneficial beyond the need for in situ communication. The developers were able to collect data more easily and visualize it in ways not previously possible thus gaining new insight.

From this experience, two lessons were learned, the coupling of a simulation to an in-situ framework will require changes to its infrastructure, and is a chance to re-evaluate previous design decisions. When infrastructure changes are not possible, other mechanisms such as lightweight wrappers are key.

Going forward, as more ephemeral data is collected and communicated to the in situ framework scalability will need to be assured. In addition, the collection and communication of emphermal data must continue to have a minimal impact on the simulation physics. For instance, currently there is litte impact on memory resources when compared to those being used by the simulation physics. However, that could certainly change as additional diagnostics are incorporated.

6 Acknowledgment

This material was based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. The authors wish to thank the Uintah and VisIt development groups.

7 Disclaimer

References

1. Bright-computing cluster manager. http://www.brightcomputing.com/Bright-Cluster-Manager.php.
2. Cacti. http://www.cacti.net/.
3. Nagios. http://www.nagios.com/.
4. S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, , et al. Scientific discovery at the exascale. In *Report from the DOE ASCR 2011 Workshop on Exascale Data Management*, 2011.
5. U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin. 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*, pages 25–29. ACM, 2015.
6. A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock, et al. In situ methods, infrastructures, and applications on high performance computing platforms. In *Computer Graphics Forum*, volume 35, pages 577–597. Wiley Online Library, 2016.
7. J. Brandt, V. De Sapio, A. Gentile, J. Mayo, P. Pebay, D. Roe, D. Thompson, and M. Wong. The ovis analysis architecture. *Sandia Report SAND2010-5107, Sandia National Laboratories*, 2010.
8. H. Childs. The in situ terminology project. http://ix.cs.uoregon.edu/~hank/insituterminology. Accessed: 2018-04-06.
9. H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. CRC Press, Oct 2012.
10. M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro. Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework. In *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*, pages 67–75. IEEE, 2013.
11. N. Fabian, K. Moreland, D. Thompson, A. C. Bauer, P. Marion, B. Gevecik, M. Rasquin, and K. E. Jansen. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

12. A. Geist and R. Lucas. Major Computer Science Challenges At Exascale. *International Journal of High Performance Computing Applications*, 23(4):427–436, 2009.
13. A. Hoisie and V. Getov. Extreme-Scale Computing - Where 'Just More of the Same' Does Not Work. *Computer*, 42(11):24–26, Nov. 2009.
14. K. A. Huck, K. Potter, D. W. Jacobsen, H. Childs, and A. D. Malony. Linking performance data into scientific visualization tools. In *2014 First Workshop on Visual Performance Analysis*, pages 50–57, Nov 2014.
15. K. E. Isaacs, A. G. Landge, T. Gamblin, P.-T. Bremer, V. Pascucci, and B. Hamann. Exploring performance data with boxfish. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1380–1381. IEEE, 2012.
16. KitWare. ParaView, http://www.paraview.org/.
17. G. Kumfert, D. E. Bernholdt, T. G. W. Epperly, J. A. Kohl, L. C. McInnes, S. Parker, and J. Ray. How the common component architecture advances computational science. *Journal of Physics: Conference Series*, 46(1):479, 2006.
18. J. Luitjens and M. Berzins. Improving the performance of uintah: A large-scale adaptive meshing computational framework. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, pages 1 – 10, 05 2010.
19. K.-L. Ma. In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, 2009.
20. M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
21. Q. Meng and M. Berzins. Scalable large-scale fluid-structure interaction solvers in the uintah framework via hybrid task-based parallelism algorithms. *Concurrency and Computation: Practice and Experience*, 26:1388–1407, 2014.
22. Q. Meng, J. Luitjens, and M. Berzins. Dynamic task scheduling for the uintah framework. *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–10, 2010.
23. J. D. Mulder, J. J. van Wijk, and R. van Liere. A survey of computational steering environments. *Future Gener. Comput. Syst.*, 15(1):119–129, Feb. 1999.
24. H. Nord and E. Chambe-Eng. The qt company. https://www.qt.io. Accessed: 2018-04-06.
25. O. Padron and D. Semeraro. Torusvis: A topology data visualization tool. In *Proceedings of the Cray User Group meeting*, 2014.
26. U. Rathmann and J. Wilgen. Qwt user's guide. http://qwt.sourceforge.net. Accessed: 2018-04-06.
27. R. Sisneros and K. Chadalavada. Toward understanding congestion protection events on blue waters via visual analytics. In *Proceedings of the Cray User Group meeting*, 2014.
28. 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*, EGPGV '11, pages 101–109, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association.
29. C. Wood, M. Larsen, A. Gimenez, C. Harrison, T. Gamblin, and A. Malony. Projecting performance data over simulation geometry using sosflow and alpine. In *2017 Forth Workshop on Visual Performance Analysis*, pages 1–8, Nov 2017.