Chapter 1

Parallel Visualization Tools

A visualization success, as a scientific endeavor, is measured not by the knowledge of the system itself but rather by how visualization aids in the scientific inquiries of other disciplines. As such, visualization research must consider the tools in which techniques must eventually be deployed.

Visualization tools have a wide variety of considerations for their implementation. Foremost for ultrascale visualization is parallel processing. Ultrascale visualization tools must take advantage of high performance computers to process the copious amount of data generated by the petascale simulations of today. Furthermore, for a tool to be useful it must be available and responsive while accounting for a broad range of operations from I/O to analysis to rendering.

Our work in ultrascale visualization uses ParaView [34] as the deployment platform of choice. ParaView is a general-purpose, open-source, interactive, parallel visualization tool with demonstrated parallel scalability [8, 23, 26, 28, 38]. ParaView is built on top of the Visualization Toolkit (VTK) [33].

1.1 Previous Work

A large proportion of visualization frameworks, libraries, and tools, including ParaView, is based on the metaphor of a visualization pipeline [14,19]. A visualization pipeline embodies a dataflow network in which computation is described as a collection of executable modules that are connected in a directed graph representing how data moves between modules. Many variations of the visualization pipeline are possible including out-of-core streaming [17, 37], extent propagation [3], contracts for modified data requests [10], detachable executives [16], and prioritized streaming [5, 6].

The visualization pipeline is amenable to parallel computing [4]. The most scalable parallel mode for a visualization pipeline is the data parallel execution mode where data is partitioned and the same algorithm is run concurrently on all pieces. Most parallel visualization tools, including ParaView, use this parallel execution mode, which continues to perform well on the leadership class facilities of today [11]. Most serial visualization algorithms can be run in data-parallel mode with little or no modification although there are exceptions such as streamlines [31] and connected components [25].

A large-scale parallel visualization tool also requires an efficient parallel rendering mechanism. Although parallel rendering algorithms can differ significantly, the most scalable instances are of the sort-last class of algorithms [39]. ParaView uses a sort-last rendering library called IceT [22].

1.2 Advanced Pipeline Controls

Leveraging existing visualization tools and libraries is liberating in that new technologies can be deployed faster and that different components can work easily together. However, the same tools can be constraining in that all code must conform to the same execution environment and that any limits in the framework

are difficult to work around. Consequently, some new technologies demand and update to the framework itself. In our case, the framework is the VTK visualization pipeline and the ParaView application.

1.2.1 Time Requests

Until recently, visualization pipelines operated on data at a single snapshot in time. Operating on data that evolved over time entailed an external mechanism executing the pipeline repeatedly over a sequence of time steps. Such behavior arose from data sets being organized as a sequence of time steps and the abundance of visualization algorithms that are time invariant.

Time control can be added to the visualization pipeline by passing time metadata [7]. The process is roughly analogous to passing region or contract information to augment the data flowing in the pipeline [3, 10] except that regions can be expressed in time as well as space. A source declares what time steps or time region is available, and each filter has the ability to augment that time for downstream modules. Likewise, as data is processed each filter may request additional or different time regions. The region request may contain one or more time steps.

These temporal regions enable filters that operate on data that changes over time. For example, a temporal interpolator filter can estimate continuous time by requesting multiple time steps from upstream and interpolating the results for downstream modules.

Some algorithms, such as particle tracing, may need all data over all time. Although such a region may be requested by this mechanism, it is seldom feasible to load this much data at one time. Instead, an algorithm may operate on a small number of time steps at one time, iterate over all time, and accumulate the results. To support this, during execution filters can request a re-execution of the upstream pipeline with different time steps and then continue to compute with the new data.

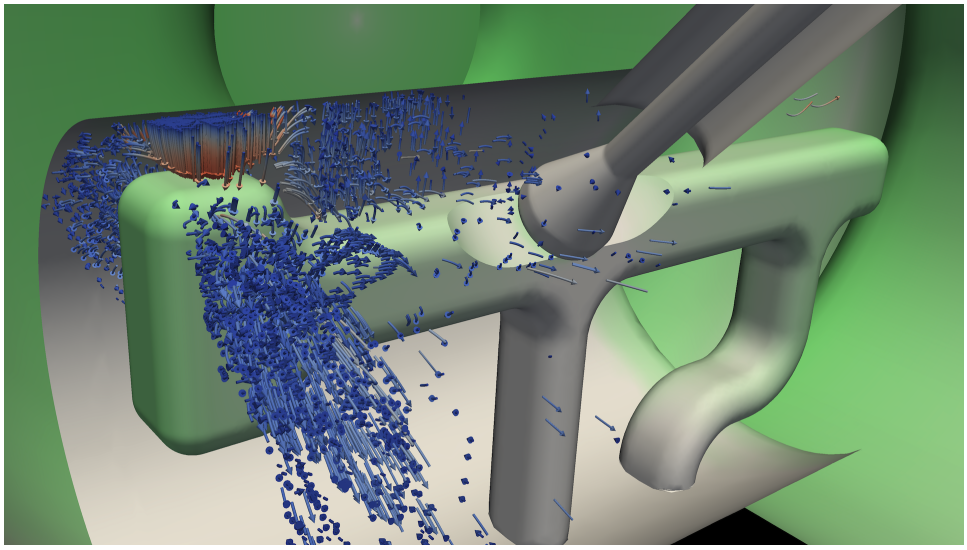


Figure 1.1: Visualization of moving particles in a linear accelerator coupler. The intention of the simulation is to determine whether a damaging phenomenon called multipacting is occurring, which could be evidenced by many particles with the same resonant trajectory such as those in the upper left corner of this image. Data courtesy of Lixin Ge and Greg Schussman of SLAC.

The visualization in Figure 1.1 demonstrates several features made possible with time requests in the visualization pipeline. The data are in two parts: a static mesh of the volume records the periodic changes in electric and magnetic fields within a linear-accelerator coupler cavity and a collection of particles that move through the space. The mesh and particles represent time varying data in two different ways, as a

complex mode representing continuous periodic change and as a sequence of snapshots at discrete time intervals.

Time requests simplify the coordination of the disparate time sequences. ParaView makes a singular time step request to the pipeline, which distributes the requested time to each module that loads the time step with its own mechanism. This time metadata also simplifies the creation of the curved arrows representing particle movement. A filter monitors the time stamps of particles and traces out fixed length paths as they move.

1.2.2 Scripting

Scripting can be an important part of the visualization workflow. Scripting allows users to add custom computations or automate repetitive tasks. A visualization tool can aid script creation by generating scripts to replicate a state or set of actions.

ParaView implements scripting by adding bindings to the Python language. Using Python for ParaView's scripting language provides several advantages. First, it simplifies implementation, debugging, and maintenance by using the ready-made Python interpreter. Second, Python comes with a wide breadth of language constructs and existing packages that users can incorporate into their scripts. Third, Python is a popular language in the scientific community. Using it means that many of ParaView's users will already be familiar and comfortable with the syntax.

There are actually two types of scripts in ParaView. The first type of script is a control, or client-side, script. The control script replicates the actions that can be performed in the ParaView GUI. The control scripting contains bindings to establish and execute a visualization. A control script has no direct access to the data, but it is possible to transfer data from the ParaView server, where the actual data resides, to the client where the script can access it. It is possible within ParaView to trace the actions performed in the GUI to a control script or to capture the state as a script. Such features greatly simplify script creation.

The second type of script is a data-processing, or server-side, script. The data-processing script is inserted into the parallel visualization pipeline and executed in parallel to process data that flows through the pipeline. Data-processing scripts behave almost identically to other pipeline modules generally written in other languages like C++. Implementing the module in a scripting language has the added benefits of enabling rapid prototyping and customized operations.

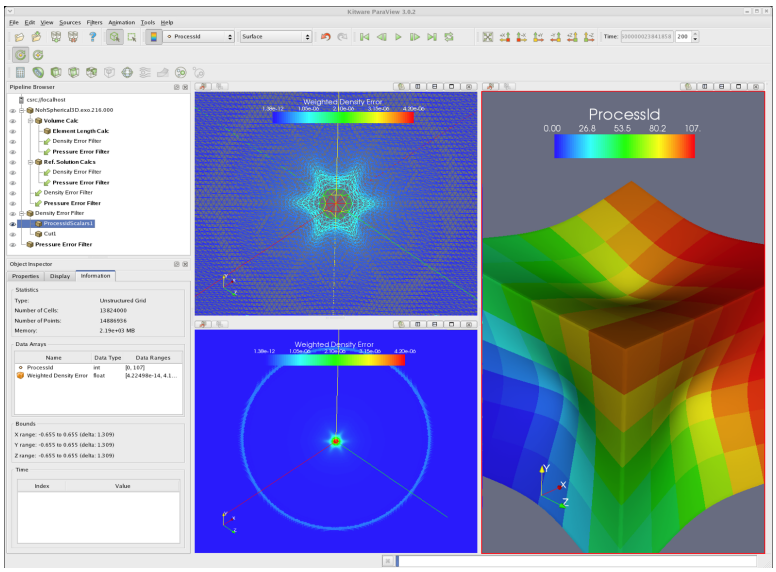


Figure 1.2: An example of verifying ALEGRA simulations using custom scripting operations.

The use of data-processing and control scripts is instrumental in many quantitative analysis tasks such as the verification shown in Figure 1.2. A data-processing script is used to compute a known analytic solution to a problem on a mesh. A second data-processing script compares this analytic solution to the Lagrangian-Eulerian finite element results of an ALEGRA simulation and computes an error metric. Because the data-processing scripts are executed in parallel on high performance computers, they can handle arbitrarily large data sets. A control script then automates the process across any number of parameters and result files.

1.3 In Situ

In situ visualization refers to visualization that is run in tandem with the simulation that is generating the results being visualized. There are multiple approaches to in situ visualization. Some directly share memory space whereas others share data through high speed message passing. Nevertheless, all in situ visualization systems share two properties: the simulation and the visualization are run concurrently (or equivocally with appropriate time slices) and the passing of data from simulation to visualization bypasses the costly step of writing to or reading from a file on disk.

The concept of running a visualization while the simulation is running is not new. It is mentioned in the 1987 National Science Foundation Visualization in Scientific Computing workshop report [21], which is often attributed to launching the field of scientific visualization. However, the interest in in situ visualization has grown significantly in recent years.

Recent studies show that the cost of dedicated interactive visualization computers for petascale is prohibitive [9] and that the time spent in writing data to and reading data from disk storage is beginning to dominate the time spent in both the simulation and the visualization [29, 30, 32]. Consequently, in situ visualization is one of the most important research topics in large scale visualization today [2, 15].

A general-purpose in situ visualization tool must be flexible so that it may adapt to multiple simulation codes and execution environments. To that end, ParaView developers and users are pursuing multiple approaches to in situ visualization. These approaches fall into two basic categories. The first is a coprocessing approach where the ParaView parallel visualization services are directly linked into a simulation to provide analysis on request. The second approach is an in transit method where the ParaView visualization services are connected to an I/O transport mechanism for a more loose coupling.

1.3.1 Coprocessing

The ParaView Coprocessing Library [13] is a C++ library with an externally facing API to C, FORTRAN and Python. It is built atop the Visualization Toolkit (VTK) and ParaView. By building the coprocessing library with VTK, it can access a large number of algorithms including writers for I/O, rendering algorithms, and processing algorithms such as isosurface extraction, slicing, and flow particle tracking. The coprocessing library uses ParaView as the control structure for its pipeline. Although it is possible to construct pipelines entirely in C++, the ParaView control structure allows pipelines configured through Python scripts and pipelines connected remotely, either through a separate cluster or directly to an interactive visualization client running on an analyst's desktop machine.

Since the coprocessing library will extend a variety of existing simulation codes, we cannot expect its API to easily and efficiently process internal structures in all possible codes. Our solution is to rely on adaptors, Figure 1.3, which are small pieces of code written for each new linked simulation, to translate data structures between the simulation's code and the coprocessing library's VTK-based architecture. An adaptor is responsible for two categories of input information: simulation data (simulation time, time step, grid, and fields) and temporal data, i.e., when the visualization and coprocessing pipeline should execute. To do this effectively, the coprocessing library requires that the simulation code invoke the adaptor at regular intervals.

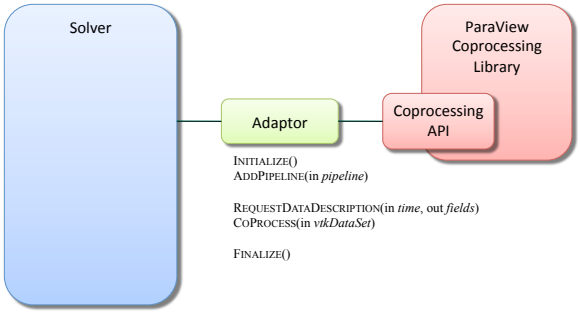


Figure 1.3: The ParaView Coprocessing Library generalizes to many possible simulations by means of adaptors. These are small pieces of code that translate data structures in the simulation's memory into data structures the library can process natively. In many cases, this can be handled via a shallow copy of array pointers, but in other cases it must perform a deep copy of the data.

To maintain efficiency when control is passed, the adaptor queries the coprocessor to determine whether coprocessing should be performed and what information is required to do the processing. If coprocessing is not needed, it will return control immediately to the simulation. If coprocessing is needed, then the coprocessor will specify which fields (e.g., temperature, velocity) are required of the adaptor to complete the coprocessing. Then the adaptor passes this information and execution control to the coprocessing library's pipeline.

1.3.2 In Transit

In transit visualization also known as staged visualization) is a particularly elegant form of in situ visualization that exploits an I/O transport infrastructure. A modern supercomputer's compute rate far exceeds its disk transfer rate. Recent studies show that the latency of the disk storage can be hidden by having a "staging" job running separately but concurrently with the main computation job that is able to buffer data and write it to disk while the main job continues to compute [1, 27]. Rather than dump the results straight to disk, it is feasible to instead (or in addition) perform "in transit" analysis and visualization on these staging nodes as demonstrated in Figure 1.4.

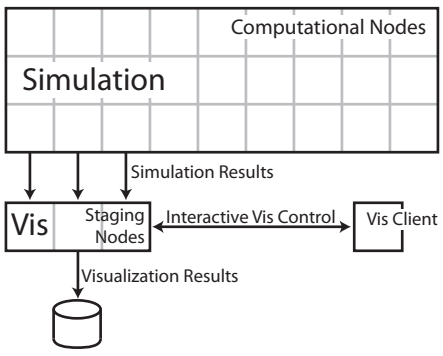


Figure 1.4: In transit visualization leverages an I/O transport layer to intercept data and perform analysis.

In transit visualization uses a separate I/O layer to integrate a simulation with the analysis and visualization. Once such transport mechanism we are using is the Adaptable I/O System framework (ADIOS) [18]. ADIOS is a next-generation I/O framework, which provides innovative solutions to a variety of I/O challenges facing large-scale scientific applications. ADIOS has been designed to separate the I/O API from the actual implementation of the I/O methods.

By decoupling the APIs from the implementation, ADIOS also enables output in a variety for formats ranging from ASCII to parallel HDF5, and also allow the usage of new data staging techniques [12, 36] which can bypass the storage system altogether.

The DataSpaces method in ADIOS [12], provides the abstraction of a virtual semantically specialized shared space that can be associatively and asynchronously accessed using simple, yet powerful and flexible, operators (e.g., put() and get()) with appropriate data selectors. These operators are location and distribution agnostic allowing the in transit visualization to reference data without explicitly dealing with discovery and transfer. Additionally, the DataSpaces staging area exists as a separate application (as demonstrated in Figure 1.4), providing fault isolation for the application. Thus, failures in the coupled codes do not have to propagate to the application.

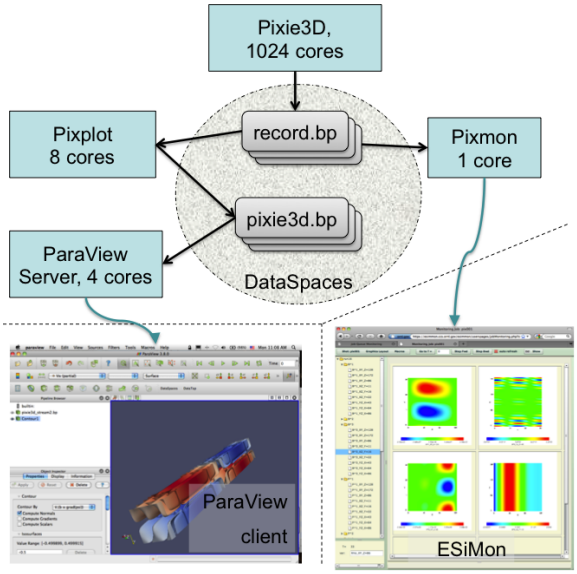


Figure 1.5: Using ADIOS/DataSpaces for in-transit analysis and visualization

An example in transit visualization application is shown in Figure 1.5. It involves five separate applications.

Pixie3D is an MHD code for plasma fusion simulation.

Pixplot is a parallel analysis code for Pixie3D output that creates a larger dataset, which is then studied by the user via visualization.

Pixmon creates 2D slices of the 3D output of Pixie3D and to present them through ESiMon [35] to monitor the run.

ParaView server with an ADIOS reader plugin can read either from a file or from a staging area.

DataSpaces serves these four applications.

The visualization server is controlled by a ParaView client; therefore, the retrieval rate of individual timesteps is varying. In our actual simulation run, Pixie3D generates about 100MB of data every second while Pixplot processed every 30th step and wrote about 500MB every 30 seconds. One compute node for staging is enough to hold 20 timesteps at once and thus 10 minutes of Pixplot run to comfortably analyze the run with ParaView. Since DataSpaces can scale to hundreds of nodes and provides low latency and high bandwidth for data exchange, it can store all timesteps of a run of this nature if needed.

1.3.3 Tool and Script Integration

Simulations are run as batch tasks. It is not possible to ensure that a person is available to interact from start to finish. It is therefore generally necessary to perform at least some of the in situ analysis and visualization as a batch computation as well. Therefore, it needs to be configured before the simulation is run.

In our ParaView-based in situ visualization tools, this configuration is most easily achieved by supplying a script of a similar nature to that described in Section 1.2.2. Although normally created using a text editor, this process can be difficult and error prone. It is much more convenient to visually position a slice plane than it is to enter a point and a normal. For example, in ParaView, the position and the orientation of a plane can be specified using a 3D widget. This widget allows interaction by dragging the plane back and forth and rotating it using the handles of the direction arrow. Based on conversations with collaborators, we believe that most analysts prefer to interact with a graphical user interface than to manually code a coprocessing pipeline.

To address this, we extend ParaView with a plugin that enables it to write pipelines as coprocessing scripts. An analyst can interactively create a pipeline and then write it out as a Python script, which can be read later by the coprocessing library when the simulation is running. This workflow is demonstrated in Figure 1.6.

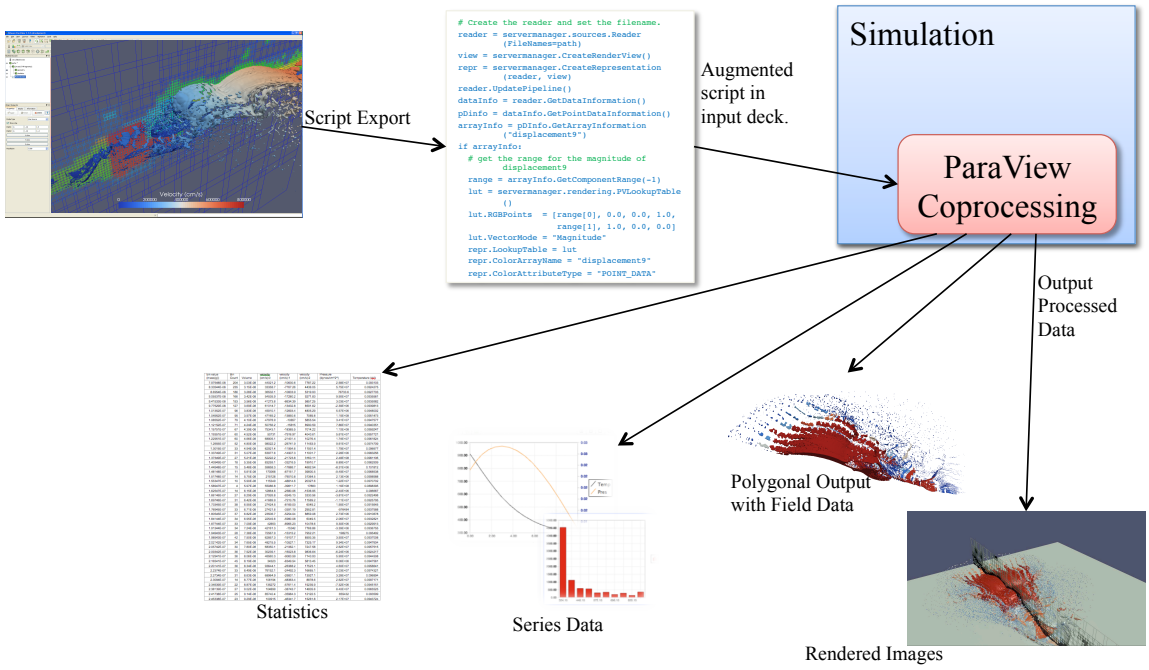


Figure 1.6: Workflow for creating and executing in situ scripts using the ParaView coprocessing library. The ParaView application creates a script from a proxy geometry, which is used in the coprocessing library during simulation to output a variety of data.

1.4 Running at Scale

An important liability for ultrascale visualization tools is the ability to run at scale. Only by taking full advantage of high performance computers can we tackle the visualization problems of ultrascale simulations. Running at scale first requires our tools to be ported to the largest supercomputers. We then must ensure that our framework scales to large numbers of processes and that our rendering system can manage.

1.4.1 Porting to Supercomputers

ParaView is a very portable application. In addition to the standard Windows, Macintosh, Linux set of common desktop platforms, ParaView has been compiled and run on most high-performance computing platforms including Linux clusters, AIX, Blue Gene, and Cray Xt.

Many of these systems have special hardware or operating system on the compute nodes specially designed for large scale jobs. These compute nodes do not contain a full user environment and therefore may not support things like login shells and compilers. Instead, a seperate set of login nodes act as a gateway for compiling and launching jobs.

The compute node may require a different type of executable than the login node. In such a case we are required to cross compile. That is, the compiler creates executables for a "target" platform (the compute node) that is different than the "native" platform (the login node) on which it is compiled.

ParaView uses CMake [20] as its build system, which simplifies the cross compiling process. CMake simplifies the cross compiling process by accepting a toolchain file that specifies the build parameters for the target platform. This will configure the build to use the correct compiler with the correct flags.

Like most large software projects, the ParaView build also requires system introspection when compiled. If the introspection requires running a program on the target platform, CMake cannot perform this automatically. Instead, CMake builds the appropriate test executables and provides an easy mechanism for the person building ParaView to run the executables and capture their results in a configuration script read back into CMake.

The ParaView build also creates some internal programs that are run during the build to automatically create source code. To run these internal platforms on the native platform, the cross-compiling target platform build is pointed to a native build containing the program versions that may run on the native platform.

1.4.2 Running on Large Number of Processes

As with any high performance computing application, it is important to be able to scale up to a large number of processes. Fortunately, the visualization pipeline in general and the ParaView server specifically scale quite readily as do most of the algorithms implemented as pipeline modules. Figure 1.7 shows evidence of good scaling characteristics of two different algorithms running in the ParaView server.

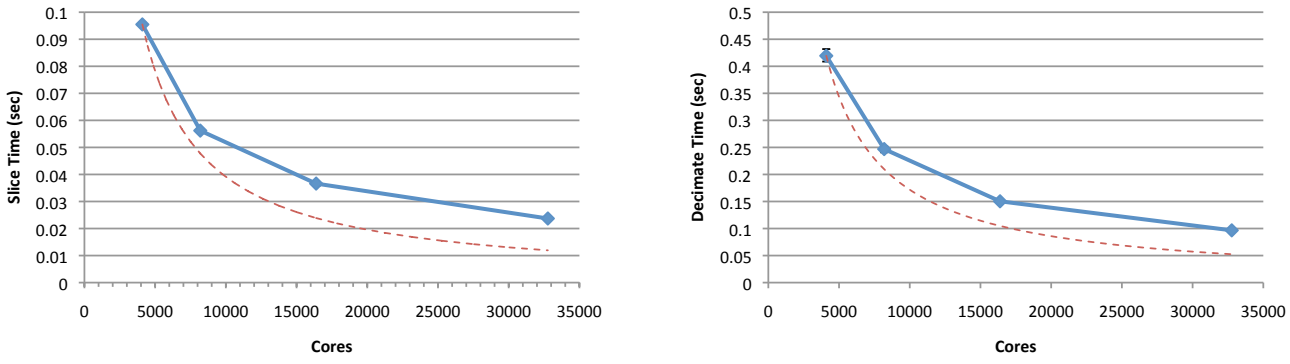


Figure 1.7: Strong scaling of the slice (left) and decimate (right) operations on unstructured PHASTA data. The dashed lines indicate perfect scaling.

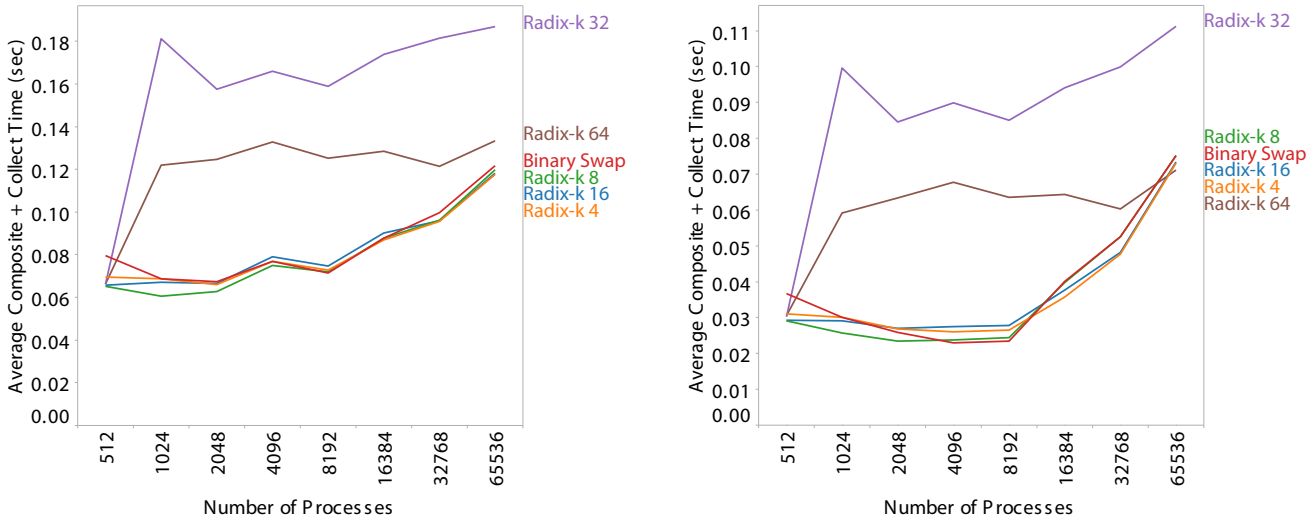
Most of the ParaView parallel framework has scaled quite easily. The problematic portions have come from sections of code that deal with user interactivity and control. For example, ParaView's progress updates misbehaved at large scale. The progress updates show in the GUI roughly what percentage of the currently running algorithm has completed. An accurate count requires collecting progress from all

the processes. Doing so requires little overhead for 100 processes, but is problematic for 10,000 processes. Thus, for large process counts, ParaView now turns off progress reporting.

1.4.3 Rendering

Rendering, of course, is an integral part of visualization, and an ultrascale visualization tool must contain a parallel rendering algorithm capable of performing well at scale. ParaView uses IceT [22] for its parallel rendering.

IceT is a parallel rendering library designed for large-scale applications. IceT uses a sort-last style of parallel rendering. In this class of parallel rendering algorithms, every process renders a local partition of data to a local image buffer, and then the image buffers are combined in a parallel composite.



(a) Transparent Geometry

(b) Opaque Geometry

Figure 1.8: Performance of binary swap and several versions of radix-k on Intrepid up to 65,536 cores (from Moreland et al [24]).

IceT contains several technologies that enables it to perform parallel rendering at scale [24]. These technologies include the radix-k compositing algorithm, described in Chapter **[cross reference to Tom's chapter here, where I assume he talks of radix-k]**, as well as several optimization an load balancing improvements.

Acknowledgments

[This statement and these logos *must* appear *somewhere* in the document. It does not have to be here. Presumably there will be a section of the report where the Office of Science and SciDAC funding are acknowledged. It would be a good place to also put these (as well as acknowledgments from all other participating institutions).]

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Bibliography

[1] Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., and Zheng, F. DataStager: Scalable data staging services for petascale applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC '09)* (2009). DOI=10.1145/1551609.1551618.

[2] Ahern, S., Shoshani, A., Ma, K.-L., et al. Scientific discovery at the exascale. Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization, February 2011.

[3] Ahrens, J., Brislawn, K., Martin, K., Geveci, B., Law, C. C., and Papka, M. Large-scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications 21*, 4 (July/August 2001).

[4] Ahrens, J., Law, C., Schroeder, W., Martin, K., and Papka, M. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Tech. Rep. #LAUR-00-1620, Los Alamos National Laboratory, 2000.

[5] Ahrens, J. P., Desai, N., McCormic, P. S., Martin, K., and Woodring, J. A modular, extensible visualization system architecture for culled, prioritized data streaming. In *Visualization and Data Analysis 2007* (2007), no. 64950I.

[6] Ahrens, J. P., Woodring, J., DeMarle, D. E., Patchett, J., and Maltrud, M. Interactive remote large-scale data visualization via prioritized multi-resolution streaming. In *Proceedings of the 2009 Ultrascale Visualization Workshop* (November 2009). doi=10.1145/1838544.1838545.

[7] Biddiscombe, J., Geveci, B., Martin, K., Moreland, K., and Thompson, D. Time dependent processing in a parallel pipeline architecture. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (November/December 2007), 1376–1383. doi=10.1109/TVCG.2007.70600.

[8] Cedilnik, A., Geveci, B., Moreland, K., Ahrens, J., and Favre, J. Remote large data visualization in the paraview framework. In *Eurographics Parallel Graphics and Visualization 2006* (May 2006), pp. 163–170.

[9] Childs, H. Architectural challenges and solutions for petascale postprocessing. *Journal of Physics: Conference Series 78*, 012012 (2007). doi=10.1088/1742-6596/78/1/012012.

[10] Childs, H., Brugger, E., Bonnell, K., Meredith, J., Miller, M., Whitlock, B., and Max, N. A contract based system for large data visualization. In *IEEE Visualization 2005* (2005), pp. 191–198.

[11] Childs, H., Pugmire, D., Ahern, S., Whitlock, B., Howison, M., Prabhat, Weber, G. H., and Bethel, E. W. Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications* (May/June 2010), 22–31.

[12] Docan, C., Parashar, M., and Klasky, S. DataSpaces: An interaction and coordination framework for coupled simulation workflows. In *19th ACM International Symposium on High Performance and Distributed Computing (HPDC'10)* (Chicago, IL, June 2010).

[13] Fabian, N., Moreland, K., Thompson, D., Bauer, A. C., Marion, P., Geveci, B., Rasquin, M., and Jansen, K. E. The ParaView coprocessing library: A scalable, general purpose *In Situ* visualization library. In *IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV 2011)* (October 2011).

[14] Haeberli, P. E. ConMan: A visual programming language for interactive graphics. *ACM SIGGRAPH Computer Graphics 22*, 4 (August 1988), 103–111.

[15] Johnson, C., Ross, R., et al. Visualization and knowledge discovery. Report from the DOE/ASCR Workshop on Visual Analysis and Data Exploration at Extreme Scale, October 2007.

[16] Kitware, I. *The VTK User's Guide*, 11th ed. Kitware, Inc., 2010. ISBN 978-1-930934-23-8.

[17] Law, C. C., Martin, K. M., Schroeder, W. J., and Temkin, J. A multi-threaded streaming pipeline architecture for large structured data sets. In *Proceedings of IEEE Visualization 1999* (October 1999), pp. 225–232.

[18] Lofstead, J., Zheng, F., Klasky, S., and Schwan, K. Adaptable, metadata rich IO methods for portable high performance IO. In *IEEE International Symposium on Parallel & Distributed Processing, IPDPS'09* (May 2009). DOI=10.1109/IPDPS.2009.5161052.

[19] Lucas, B., Abram, G. D., Collins, N. S., Epstein, D. A., Gresh, D. L., and McAuliffe, K. P. An architecture for a scientific visualization system. In *IEEE Visualization* (1992), pp. 107–114.

[20] Martin, K., and Hoffman, B. *Mastering CMake*, 5th ed. Kitware, Inc., 2010. ISBN 978-1-930934-22-1.

[21] McCormick, B. H., DeFanti, T. A., and Brown, M. D., Eds. *Visualization in Scientific Computing (special issue of Computer Graphics)*, vol. 21. ACM, 1987.

[22] Moreland, K. IceT users' guide and reference version 2.1. Tech. Rep. SAND 2011-5011, Sandia National Laboratories, August 2011.

[23] Moreland, K., Fabian, N., Marion, P., and Geveci, B. Visualization on supercomputing platform level ii asc milestone (3537-1b) results from sandia. Tech. Rep. SAND 2010-6118, Sandia National Laboratories, September 2009.

[24] Moreland, K., Kendall, W., Peterka, T., and Huang, J. An image compositing solution at scale. In *SC Conference on High Performance Computing, Networking, Storage and Analysis* (November 2011).

[25] Moreland, K., Law, C. C., Ice, L., and Karelitz, D. Analysis of fragmentation in shock physics simulation. In *Proceedings of the 2008 Workshop on Ultrascale Visualization* (November 2008), pp. 40–46.

[26] Moreland, K., Rogers, D., Greenfield, J., Geveci, B., Marion, P., Neundorf, A., and Eschenberg, K. Large scale visualization on the Cray XT3 using paraview. In *Cray User Group* (2008).

[27] Nisar, A., keng Liao, W., and Choudhary, A. Scaling parallel I/O performance through I/O delegate and caching system. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing* (November 2008).

[28] Patchett, J., Ahrens, J., Ahern, S., and Pugmire, D. Parallel visualization and analysis with ParaView on a Cray Xt4. In *Cray User Group* (2009).

[29] Peterka, T., Yu, H., Ross, R., and Ma, K.-L. Parallel volume rendering on the ibm blue gene/p. In *Proceedings of Eurographics Parallel Graphics and Visualization Symposium 2008* (2008).

[30] Peterka, T., Yu, H., Ross, R., Ma, K.-L., and Latham, R. End-to-end study of parallel volume rendering on the ibm blue gene/p. In *Proceedings of ICPP '09* (September 2009), pp. 566–573. doi=10.1109/ICPP.2009.27.

[31] Pugmire, D., Childs, H., Garth, C., Ahern, S., and Weber, G. Scalable computation of streamlines on very large datasets. In *Proceedings of ACM/IEEE Conference on Supercomputing* (November 2009).

[32] Ross, R. B., Peterka, T., Shen, H.-W., Hong, Y., Ma, K.-L., Yu, H., and Moreland, K. Visualization and parallel I/O at extreme scale. *Journal of Physics: Conference Series 125*, 012099 (2008). doi=10.1088/1742-6596/125/1/012099.

[33] Schroeder, W., Martin, K., and Lorensen, B. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*, fourth ed. Kitware Inc., 2004. ISBN 1-930934-19-X.

[34] Squillacote, A. H. *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 2007. ISBN 1-930934-21-1.

[35] Tchoua, R., Klasky, S., Podhorszki, N., Grimm, B., Khan, A., Santos, E., Silva, C., Mouallem, P., and Vouk, M. Collaborative monitoring and analysis for simulation scientists. In *2010 International Symposium on Collaborative Technologies and Systems, (CTS 2010)* (May 2010), pp. 235–244.

[36] Tian, Y., Klasky, S., Abbasi, H., Lofstead, J., Grout, R., Podhorszki, N., Liu, Q., Wang, Y., and Yu, W. Edo: Improving read performance for scientific applications through elastic data organization. In *IEEE Cluster 2011* (Austin, TX, 2011).

[37] Vitter, J. S. External memory algorithms and data structures: Dealing with massive data. *AVM Computing Surveys 33*, 2 (June 2001).

[38] White, D. Red storm capability visualization level II ASC milestone #1313 final report. Tech. Rep. SAND 2005-5989P, Sandia National Laboratories, 2005.

[39] Wylie, B., Pavlakos, C., Lewis, V., and Moreland, K. Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications 21*, 4 (July/August 2001), 62–70.