Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System

Brad Whitlock[1] Jean M. Favre[2] Jeremy S. Meredith[3]

[1]Lawrence Livermore National Laboratory, CA, USA
[2]Swiss National Supercomputing Center (CSCS)
[3]Oak Ridge National Laboratory, TN, USA

Abstract

There is a widening gap between compute performance and the ability to store computation results. Complex scientific codes are the most affected since they must save massive files containing meshes and fields for offline analysis. Time and storage costs instead dictate that data analysis and visualization be combined with the simulations themselves, being done in situ *so data are transformed to a manageable size before they are stored. Earlier approaches to* in situ *processing involved combining specific visualization algorithms into the simulation code, limiting flexibility. We introduce a new library which instead allows a fully-featured visualization tool, VisIt, to request data as needed from the simulation and apply visualization algorithms* in situ *with minimal modification to the application code.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing I.3.4 [Computer Graphics]: Graphics Utilities—Application packages

1. Introduction

Large data processing techniques encompass methods such as distributed parallelism, out-of-core processing, multi-resolution, and in situ processing [Chi07]. All but the last of these methods require management of massive I/O to and from disk-based storage. However, recently deployed supercomputers and those planned in the near future provide far more compute capacity than I/O bandwidth and thus suffer from an inherent I/O bottleneck. For example, Table 1 shows approximate I/O rates for selected historical and future supercomputers such as LLNL's planned Sequoia BG/Q computer [ASC03] [FJM*99] [MEGL01] [ASC07] [ORN08] [ASC08] [Law09]. The "Writable FLOPS" shows the ratio of compute performance to the I/O bandwidth, and the "Whole-System Checkpoint" is the ratio of system RAM to I/O bandwidth. The trend shows both that we are less able to save data as it is generated and that we will be able to save data less often. Even accounting for variations in reporting practices for these statistics, it seems clear that the risk of losing important scientific data is growing.

Clever I/O strategies can improve the rate at which data can be stored but, particularly as we approach the exascale era, less data must be stored if application scalability is to be preserved. Applications not able to reduce the amount of data they write will be faced with waiting long periods for their data to be written to disk. Post-processing applications supporting scientific codes with large data are also affected, as they must read all of the data that was written by the scientific code. Thus the penalty of doing I/O to disks is paid twice, first by the simulation, and later by the post-processing code. For example, Peterka [PYRM08] demonstrated Volume Rendering of massive data on BG/P; however, they cite I/O costs of over 90% of the overall runtime, due to the fact that on the BG/P used for the experiments, there is but one I/O node for every 64 compute nodes.

With such challenges at the fore in I/O and with the availability of cycles for computations, it makes sense to reduce the amount of I/O required by instead using extra cycles for activities that were historically done as post-processing steps. This goal can be achieved by combining simulations with in situ processing. Post-processing often consists of data analysis and visualization, both of which significantly reduce the size of the input data to sustainable sizes. For example, visualization can transform petabytes of simulation data into a pictorial representation of just a few megabytes, thus trading very large scale, difficult to manage I/O opera-

tions for very small I/O operations. Also, typical visualization operations take on the order of a few seconds to tens of seconds compared with hundreds or thousands of seconds needed to write full-sized data to disk. In their introduction to the challenges and opportunities for in situ visualization [Ma09], the authors point out that accessing data in situ has many advantages since all pertinent data arrays and geometries are readily available at the full spatio-temporal resolution. Code debugging, run-time monitoring, massive scale calculations aimed at reducing the data's dimensions before I/O to disks are made possible by in situ processing.

Machine	Year	Writable FLOPS	Whole-System Checkpoint
ASCI Red	1997	0.075%	300 sec
ASCI Blue Pac.	1998	0.041%	400 sec
ASCI White	2001	0.026%	480 sec
ASCI Red Storm	2005	0.035%	660 sec
ASCI Purple	2005	0.025%	500 sec
NCCS XT4	2007	0.004%	1400 sec
Roadrunner	2008	0.005%	480 sec
NCCS XT5	2008	0.005%	1250 sec
ASC Sequoia	201x	0.001%	3200 sec

Table 1: *Historical supercomputer I/O rates*

2. Related Work

Earlier works include SCIRun [JPH*99] which was one of the first general purpose frameworks with a visual programming environment to assist users in assembling data flow networks. SCIRun was designed from the ground up to enable computational steering, by constructing re-usable components for modeling, computation and visualization. It was conceived to run on shared-memory machines, with each module running as an independent thread, and favored the inclusion of newly created simulation components. Another example of a tight-coupling is RVSLIB [SD01], a commercial product of NEC providing a library of subroutine calls to insert into the simulation code. Others have elected to physically separate simulation and visualization reosurces. Ellsworth [EHG*06] copies simulation data to a shared memory segment on a different set of computer nodes to isolate mission-critical computations from other components. Esnard [AEC06] emphasises a data redistribution with parallel data transfer from M compute nodes to N (N « M) visualization nodes, using CORBA to implement their communication protocol. Their simulation source codes are annotated with the EPSN API, describing both the program structure and the data decomposition in order to coordinate the treatment of steering requests in parallel, and ensure the time-coherence of parallel tasks.

More recently, we find other published works in three general types of in-memory coupling for simulations and analysis. First come the strategies which decouple the I/O from the simulation, staging data to a second memory location. The ADIOS library [LZKS09] is a very flexible I/O library enabling multiple transport methods (MPI-I/O, or NETCDF, or HDF5), or asynchronous I/O to high-bandwidth I/O hardware nodes or to remote servers, enabling other codes to interface to the data. Similarly, but applicable only to simulations which already have coded their I/O with the HDF5 API, Soumagne [SBC10] emulates the *MPI-I/O* virtual file driver of HDF5 but redirects the data in parallel to a distributed shared memory buffer over multiple TCP connections. This secondary memory buffer is then available as a data source by an application. They present a parallel ParaView plug-in which acts as the data consumer. Their method, however, suffers from large memory consumption since the data arrays encoded in HDF5 data streams cannot be directly reused by the visualization pipeline, instead requiring a second memory-to-memory conversion into VTK objects.

Other practitioners incorporate domain knowledge and tasks into the simulation codes. For example, Yu [YWG*10] describes a tight coupling of volume and particle rendering in a combustion code, demonstrating the advantage of having access to the fully resolved spatio-temporal data.

Finally, there are attempts to couple general coprocessing libraries with the simulations. Developers at Kitware, Inc [BB10] demonstrated a coprocessing library, whereby a simulation can perform in situ processing for analysis scenarios known a priori. This method relies on a *Data Adaptor* taking the raw data in memory, and formatting it into VTK objects which the ParaView pipeline supports. Coprocessed data may be staged using parallel socket connections to their ParaView server running on a visualization cluster. While this can facilitate user-directed, interactive queries, it is limited to previously coprocessed data.

Our in situ approach best resembles one that combines general coprocessing libraries with the simulation in that it enjoys direct access to data and sharing of compute resources. However, we implement the system within a framework that supports read-on-demand of any quantity exposed by the simulation. This is key because our approach integrates a fully featured visualization and analysis system that can be driven interactively by the user for a myriad of analysis purposes, including scientific analysis, presentation graphics, data exploration, and even code debugging.

3. System Design

Our library targets massively parallel simulations, and uses the paradigm of message passing, based on the MPI library. We address the needs of simulations using a distributed memory parallel programming model, whereby the in situ processing lives in the same memory space as simulations.

3.1. Design Philosophy

We had several goals with the design of our proposed in situ solution:

- *Maximize features and capabilities.* There are numerous use cases for visualization and analysis, particularly in situ. If we focus the feature set on making movies, for example, then we may lack features for interactive debugging.
- *Minimize code modifications to simulations.* We expect this to support numerous application codes, and the least effort it takes to apply to a new simulation, the more likely our library will be adopted by the HPC community.
- *Minimize impact to simulation codes when in use.* Ideally, codes should be able to run the same problems with or without in situ analysis.
- *Zero impact to simulation codes when not active.* Our aim here is that simulation codes should be able to build a single executable, with in situ support built-in, and suffer no detrimental side effects. This allows users to start an in situ session on demand instead of deciding before running a simulation whether or not they will want these capabilities.

3.2. Design Overview

To accomplish these goals, we made several decisions about the architecture of our proposed system. First, we use VisIt [CBB*05] as the data analysis and visualization system to interface with simulation codes. VisIt is an open-source visualization project available on a wide variety of computing platforms. We chose VisIt both for its large feature set and for its proven ability to execute efficiently on massively parallel computer systems [CPA*10].

To interface with VisIt, we created a simulation library ("*Libsim*") that is capable of interfacing with VisIt clients as if it were a parallel VisIt server, providing simulation data to the VisIt processing engine, and advertising computational steering capabilities to an interactive VisIt client.

3.3. VisIt Architecture

VisIt provides a client/server architecture in which the tasks of visualization and data analysis are separated into different component programs. The client programs run on a local computer and leverage hardware acceleration. The server programs run remotely on large supercomputers and are responsible for browsing the file system and for parallel computations. As plots are requested by the client, the VisIt compute server is instructed to read data, assemble data flow networks, execute them, and send the results back to the client for display. VisIt's compute server is optimized for distributed memory parallelism using MPI, and it creates identical data flow networks, consisting partially of VTK [SML96] filters, on all processors. Each data flow network then executes using different pieces of the larger data set. This computation is governed by a central executive that optimizes work flow using contracts. Contracts are a part of the request generated for each operation; they make their way upstream to the source of the data flow network (such as a file format reader plug-in), being modified incrementally by each filter to optimize the size, dimension, extents and ranges of the data the sources should make available. The data sources then initiate the execution of the networks, providing data objects which fit the contracts and are successively transformed by each filter as execution "flows" down the chain of filters, creating the result at the terminal end of the network. See Figure 1 for an example of this.

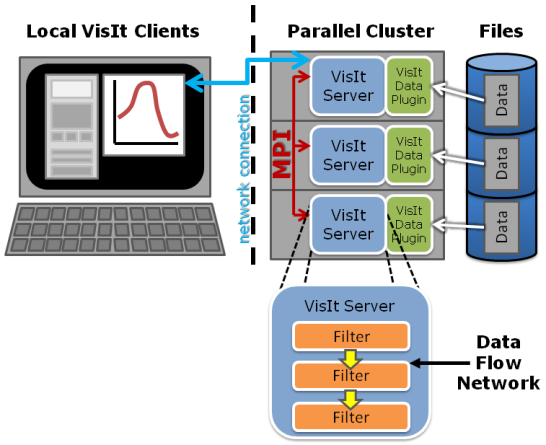

Figure 1: *VisIt architecture diagram for client/server processing of file data using a parallel compute server.*

3.4. In Situ Processing with VisIt

As seen in Figure 2, in a fully running in situ analysis session, the processing diagram looks much the same as a normal parallel VisIt analysis session, except that the VisIt server and the scientific simulation are now the same process, with VisIt's server logic having been dynamically loaded into the simulation. The steps typically proceed as follows:

1. The simulation code launches and starts execution.
2. The simulation regularly checks for connection attempts from VisIt clients.
3. When a VisIt client attempts to connect, the simulation loads the VisIt server library and allows it to complete the connection.
4. The VisIt server asks the simulation for a description of its meshes and data types.
5. Either running or paused, the simulation relies on the VisIt server to handle any VisIt-specific operations, handing it pointers to data when requested.

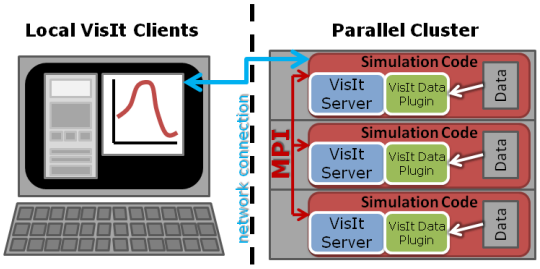

Figure 2: *VisIt architecture diagram for client/server processing of parallel simulation data with in situ library.*

A few minor changes were made to VisIt proper to allow it to function as an in situ solution. Some changes were internal, such as allowing VisIt to connect to an already-running server process instead of launching a new one. Some were graphical, such as new user interface elements to show current simulation connections and advertise control commands supported by the simulation. And some were in the build process, such as linking the VisIt server as both a standalone executable and as a runtime library. Otherwise, VisIt's architecture was already suitable for this style of in situ processing.

3.5. "Libsim": Coupling VisIt and Simulations

There are two interfaces in *Libsim*: one to drive the VisIt server, and one to hand data to the VisIt server upon request.

The control interface is capable of advertising itself to authorized VisIt clients, listening for incoming connections, initiating the connection back to the VisIt client, handling VisIt requests such as plot creation and data queries, and letting the client know when the simulation has advanced and that new data is available.

The data interface uses the extensible plug-in model of VisIt to get data into VisIt's processing pipeline. Our contribution differs from most VisIt database plug-ins which read files from disk. In the *Libsim* library, we instead created a database plug-in which uses data access callback functions to read data from memory – i.e. from running simulations. This plug-in will usually pass data pointers for simulation data as needed to respond to requests from the VisIt server. In some cases, coordinate transformations or data gather operations will be required for legacy codes.

"*Libsim*" itself is divided into two pieces. The first is a small, lightweight static library which is linked with the simulation code during compilation, and it is literally a front-end to a second, heavier weight library which is pulled in only when in situ analysis begins at runtime. We call the former the "front-end library" and the latter the "runtime library".

3.6. Architectural Implications

Several aspects of the proposed in situ library result in benefits for interacting and interfacing with simulations. The separation between the front-end library and the runtime library is one such aspect. In particular, modifications to simulation codes to support *Libsim* can be minimal, as the front-end library contains only approximately twenty simple control functions (most of which take no arguments), and only as many data functions as the simulation code wishes to expose. This separation also enables the front-end library to be written in pure C. As such, despite the usage of C++ in VisIt, no C++ dependencies are introduced into the simulation by linking to the front-end library. This is critical since many simulations are written in C and Fortran. Additionally, by providing a C interface, the process of automatically generating bindings to other programming languages is greatly simplified. The separation into front-end and runtime library components also means that the runtime library implementation is free to change as VisIt is upgraded, letting the simulation benefit from these changes automatically without relinking to create a new executable. In addition, by deferring the heavyweight library loading to runtime, there is effectively zero overhead and performance impact on a simulation code linked with *Libsim* when the library is not in use.

Another beneficial aspect is the manner in which data are retrieved from the simulations. First, as soon as an in situ connection is established, the simulation is queried for metadata containing the list of meshes and fields that the simulation wishes to expose for analysis. Just as in normal VisIt operation, this list is transmitted to the client, where users can generate plots and queries. Once the user creates a plot and VisIt starts executing the plot's data flow network, the simulation's data access functions are invoked to retrieve only the data needed for the calculation. By contrast, ParaView's CoProcessing Library [BB10] requires adding all problem-sized fields to a dataset before the minimal set of variables is known, potentially causing unnecessary data copying. Whenever possible in our approach, simulation arrays are used directly in the VTK objects to avoid array duplication, minimizing the impact on the performance and scaling capabilities of simulation codes.

VisIt's contracts are also important for in situ analysis. They provide a uniform method for passing metadata among filters in the data flow network, and they work in several ways to enhance efficiency and scalability. For example, they permit the coordination of data reducing optimizations, such as selecting only data with spatial or variable extents within the range relevant for a given analysis or visualization operation. Contracts also play a role when requesting data: filters append the names of the data arrays they need to a contract as it is sent towards the data source, so that the source only reads and processes the arrays needed for the current operation. All these aspects of contracts are enabled for in situ analysis and thus allow interactive in situ processing to be

done economically, as only the minimal necessary data are assembled for the data flow networks that the user creates on the fly.

4. Instrumenting a Simulation Code

Simulation codes need to be instrumented to utilize our *Libsim* library interface. Additions to the source code are usually minimal, and follow three incremental steps. The first phase initializes the library and alters the simulation's main iterative loop to listen for request for connection from a VisIt client. The second phase involves writing the data access callback functions, the intermediate layer that bridges VisIt simulations to allow data to be shared. The final phase adds functions that let VisIt steer the simulation.

4.1. Adapting the Main Loop

VisIt connects to a simulation by opening a small data file called a *.sim2* file. The *.sim2* file is a small file output by the simulation and it contains information such as the host id and port that VisIt needs to connect to the simulation. A simulation must perform a small amount of initialization at startup, including making library calls that write out the *.sim2* file. All simulations must call *VisItSetupEnvironment* and *VisItInitializeSocketAndDumpSimFile* during their initialization. The former function adds important VisIt-related environment variables that ensure VisIt can locate its shared libraries and plug-ins, while the latter function actually writes out the *.sim2* file that VisIt uses to connect to the simulation. Parallel simulations require a small amount of additional initialization, telling *Libsim* about the simulation's MPI communicator and providing callback functions for broadcasting information.

In order for VisIt to connect to the simulation, *Libsim* creates a listening socket that can be used to detect inbound VisIt connections. Of course, this means that the simulation must service the socket as well as any input that eventually comes from VisIt. *Libsim* provides the *VisItDetectInput* function for this purpose. *VisItDetectInput* must be called periodically from a simulation's main loop when VisIt connections are permitted. *VisItDetectInput* can be called in a blocking or non-blocking fashion, depending on the needs of the simulation. This allows both timeout and polling-based event handling schemes to be implemented. When no input is available, the simulation is free to return to its calculations. When input is available, *VisItDetectInput* returns various codes which indicate the following conditions: VisIt is trying to connect, VisIt has sent some commands, or, for simulations which accept console input during runs, that a console file descriptor has input to be read (see Figure 3). The simulation is then free to handle the inputs appropriately. It is worth noting that *VisItDetectInput* is called only from the rank 0 process in a parallel application, as only that process has a TCP connection to VisIt. The rank 0 process reads commands from VisIt and all processors participate in a collective MPI broadcast, after which all processors execute the commands in unison.

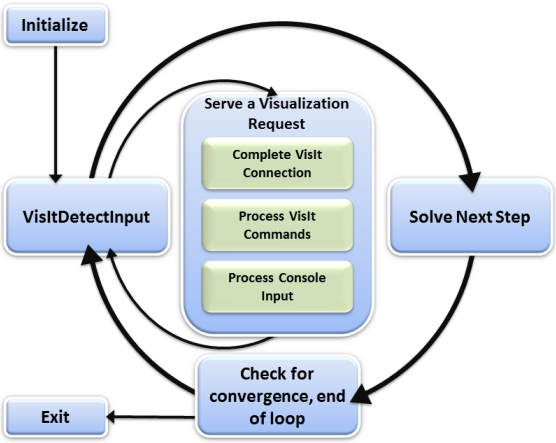

Figure 3: *Simulation control flow after introducing in situ processing.*

4.2. Sharing Data

After instrumenting the simulation's main loop with functions that let VisIt connect and accept commands, the next step is implementing data access callback functions. Data access callback functions are written in C or Fortran and are registered with the simulation runtime library, which will call them on demand when gathering inputs for a data flow network. This model ensures that no resources are wasted exposing simulation arrays that will not be used by the data flow network. Data access callbacks call library functions to allocate opaque handles to data objects such as metadata, mesh, and variable objects. The callback then populates and returns the data object so the simulation runtime library can transform it into a VTK object that can be used inside of VisIt.

The required data layout for data objects matches that of VTK, which means contiguous homogeneous arrays of built-in C types (e.g. int, float, double) are used. If the simulation data layout is incompatible with the VTK data layout then the data access callback can create new temporary storage into which data can be copied and given to VisIt. Such arrays can be marked as being owned by VisIt so VisIt will dispose of the associated memory after completing its calculations. An array whose data layout is compatible with the VTK data objects can be marked as being owned by the simulation so VisIt will treat it as read-only and will make no attempt to free its memory. In that case, the array is used directly in the reference-counted VTK object during in situ processing.

4.3. Metadata

All VisIt database reader plug-ins expose methods for reading both metadata and data. Metadata is lightweight information about the real data such as type, dimensions, extents, and whether the data exists in several pieces, called domains. VisIt uses metadata internally for optimization and to populate variable menus that allow the user to create plots. Retrieval of metadata, domain lists, and plottable data such as meshes, variables, materials, and material species is done using data access callback functions. *Libsim* provides functions for registering the various data access callback functions so they can be invoked. Metadata is central to VisIt's operation so the metadata callback function should be written first. The role of the metadata callback function is to return a populated metadata object, which serves as a container for global state and for various other objects such as mesh and variable metadata objects.

4.4. Meshes

A metadata object will need to include at least one mesh metadata object if real data are to be requested later during analysis. Data flow networks are set up using metadata information, and only when they are executed does VisIt request the real data from the simulation. This contrasts with other in situ schemes which build up an entire data object before the data flow network's inputs are known. Mesh metadata objects contain information about the mesh, including the number of domains which compose it. The number of domains will vary depending on the simulation's scheme for parallelization and load balancing, but it is common to have a 1:1 partition of domains to processors. In that case, the number of domains reported for the mesh would equal the number of processors. However, in VisIt, there is nothing to prevent other more exotic workload distributions. VisIt gains knowledge of how work is distributed among ranks through the use of the domain list callback function. The domain list callback simply returns a domain list object which contains a list of integer identifiers corresponding to the domains that are owned by the calling processor. VisIt's load balancer uses a domain list to restrict work to the processors that have it, meaning that domains will be requested by only their local task.

4.5. Variables

After mesh-related implementation has been completed, the rest of the metadata object can be filled out so VisIt will know which variables can be used. The simulation's variable callback function is called by VisIt any time field data (scalars, vectors, tensors, arrays, and labels) is needed by the data flow network. The variable callback must create a variable data object and initialize it with simulation arrays (see Figure 4). This mode simply passes simulation data arrays back to VisIt for use in VTK objects. If the simulation data layout is incompatible with VTK's data layout then simulation data must be copied into temporary arrays that can be used to initialize VTK objects.

```
visit_handle
GetVariable(int domain, const char *name,
  void *cbdata)
{
  visit_handle h = VISIT_INVALID_HANDLE;
  SimData_t *sim = (SimData_t *)cbdata;
  if(strcmp(name, "pressure") == 0)
  {
    VisIt_VariableData_alloc(&h);
    VisIt_VariableData_setDataD(h,
      VISIT_OWNER_SIM, 1, sim->nx*sim->ny,
      sim->pressure);
  }
  return h;
}
```

Figure 4: *Variable callback function written in C.*

4.6. Adding Control Functions

Simulations sometimes have command line interfaces which let their users control them using textual console commands as they execute. *Libsim* contributes some basic capabilities that let the user advertise simple commands that appear in the VisIt graphical user interface's Simulation window. These command buttons provide basic simulation steering interactively within VisIt. We have outfitted many of our simulation examples with command buttons that let the user run, halt, and single step through simulations (see Figure 5), though the nature of the buttons can be selected entirely by the simulation writer. Commands are implemented by exposing command objects in the simulation metadata and then implementing a command callback function in the simulation. Future work will add custom simulation user interfaces to VisIt.

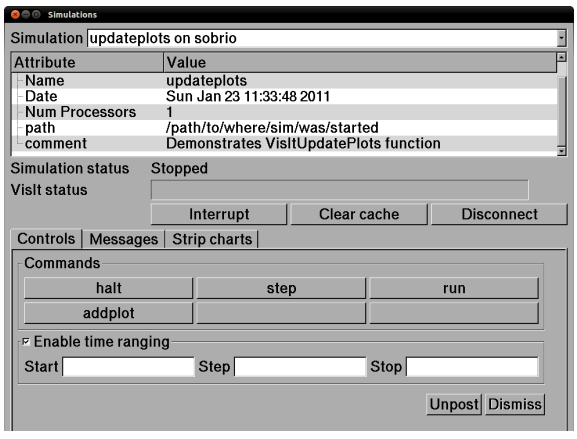

Figure 5: *Command buttons in VisIt's Simulation window.*

5. Results

We conducted our experiments on a 216 node visualization cluster with two, 6 core 2.8GHz Intel Xeon 5660 processors and 96Gb of memory per node. Our test system utilizes an InfiniBand QDR high-speed interconnect and a Lustre parallel file system. We ran two types of tests to characterize the performance of our library. The first test determined the cost associated with introducing *Libsim* into the main loop of a toy simulation. The second test investigated the performance of in situ visualization versus I/O in a real simulation code.

Adding *Libsim* to a typical simulation main loop means calling *VisItDetectInput* and *MPI_Bcast*. We added timing code to *Libsim*'s updateplots example program to measure the associated overhead without VisIt connected to the simulation. We then ran the example program through 10K main loop iterations. The timing results were 2μs and 8μs, respectively for 512 core runs. The measurements remain consistent once VisIt connects and is not requesting data. Connecting to VisIt does increase the amount of memory used by the simulation as this initiates loading the VisIt runtime libraries, which imposes a one time cost of approximately 1 second.

It has been demonstrated that I/O dominates VisIt's execution time on diverse supercomputer architectures [CPA*10]. That study indicated that VisIt's iso-contouring performance at 8K up to 64K cores was on average over an order of magnitude faster than the I/O operations needed to obtain the data from disk. In situ visualization uses simulation arrays directly, and usually allows VisIt's pipeline to execute in a fraction of the time required for I/O. Although tests using thousands of cores would better represent large supercomputers, in situ's performance advantage over I/O even becomes apparent at modest core counts.

For our in situ experiment, we instrumented GADGET-2 [Spr05], a distributed-memory parallel code for cosmological simulations of structure formation (see Figure 6). We ran GADGET-2 at 3 levels of concurrency: 32, 256, and 512 cores to measure in situ performance versus I/O performance. We isolated the time required for GADGET-2 to write its data to disk in two modes: one using collective I/O to a single file, and again in a mode where each task writes its own disk file. We separately recorded the timings of just the visualization/analysis processing in the VisIt pipeline, rendering a Pseudocolor plot of a scalar variable and saving a 2048 square pixel image to disk. We ran each of these tests using 2 sets of initial conditions for GADGET-2, generating particle sets with 16 million and 100 million particles.

In the larger test case where 100M particles were saved, GADGET-2 would generate a 2.8Gb snapshot file. Since the amount of data was constant for each test run, we find that the timings for collective I/O are relatively consistent. Independent files are the fastest means of writing files with smaller core counts, though performance in this mode is inversely proportional to the core count, as the I/O subsystem can absorb only a limited number of simultaneous requests before its performance starts to degrade. By substituting a visualization operation, in this case a Pseudocolor plot, for writing a full 2.8Gb GADGET-2 snapshot file, we were able to reduce the amount of data we write to a single 12Mb image file. From our data, shown in Table 2, we observe that for the selected visualization operation, in situ processing is competitive with single-file I/O and exceeds collective I/O performance. For smaller core counts with large sets of particles, the work performed per core is higher, resulting in a longer run time versus single-file I/O. However, as the number of cores increases, the run time of the visualization processing alone is far lower than that of either single-file I/O or collective I/O. The margin of performance is large enough that we could have generated several in situ visualizations in the time needed to write a full size snapshot file. This makes in situ analysis an attractive use of extra compute cycles in

16 Million Particles			
	32 cores	256 cores	512 cores
I/O 1 file	2.76s	4.72s	-
I/O N files	0.74s	0.31s	-
VisIt pipeline	0.77s	0.34s	-
100 Million Particles			
	32 cores	256 cores	512 cores
I/O 1 file	24.45s	26.7s	25.27s
I/O N files	0.69s	1.43s	2.29s
VisIt pipeline	1.70s	0.46s	0.64s

Table 2: *Performance of visualization/analysis vs. I/O for 16M and 100M particles at different levels of concurrency.*

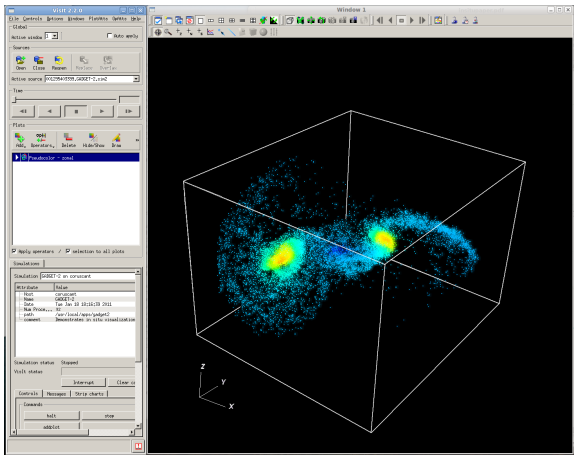

Figure 6: *VisIt client connected to GADGET-2 instrumented with Libsim.*

upcoming exascale computers and a powerful acceleration technique for large scale computations.

6. Conclusions

We have demonstrated why in situ processing of simulation data will become more important than ever as compute capacity on new supercomputers further overwhelms their I/O subsystems. We have implemented *Libsim*, an easy to use library that enables in situ computations within simulation codes by leveraging the analysis and visualization capabilities of VisIt. *Libsim* adheres to design principles which are likely to find acceptance within simulation code design teams. Namely, we minimize impact to simulation performance, minimize amount of new code that must be written, and we provide access to a fully featured, parallel visualization and analysis tool that excels at scale. Finally, we have demonstrated how to instrument GADGET-2, a well-known open source simulation, using our library and that adopting in situ techniques results in speedups relative to traditional I/O-based post-processing. *Libsim* is fully integrated with the open-source distribution of VisIt, and our instrumentation of the GADGET-2 code has been re-distributed to its original author to serve as an example for the community.

7. Future Work

We have created a useful library for instrumenting simulation codes for in situ data analysis and visualization, and there are many ways it can be enhanced. To handle the inherent challenges at the highest levels of concurrency, we expect to enhance our library to provide means of more automatic in situ analysis so it can be less user-driven. In addition, we plan to further limit resources consumed by the VisIt runtime libraries in order to lessen the impact that in situ analysis has on the simulation.

The in situ visualization community will also be faced with many challenges, as highlighted by Ma [Ma09]. There are general challenges such as the need to reformulate some classic visualization algorithms to better take advantage of the domain decomposition imposed by the simulations, the need to more carefully schedule inter-processor communications at high concurrency, and the need to accommodate data extraction needs which can never be fully formulated before the scientists actually "see" their data. We believe that a fully featured library such as VisIt is best positioned to accommodate the high degree of flexibility required for interactive and exploratory in situ visualization and analysis.

8. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contracts DE-AC05-00OR22725 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Our thanks go also to Volker Springel, who made his GADGET-2 simulation code open source, and contributed additional code and guidance to generate initial conditions for larger benchmarks.

References

[AEC06] AURELIEN ESNARD N. R., COULAUD O.: A steering environment for online parallel visualization of legacy parallel simulations. In *Proceedings of the 10th International Symposium on Distributed Simulations and Real-Time Applications (DS-RT 2006)* (2006).

[ASC03] ASC Red web site. http://www.sandia.gov/ASCI/Red/, 2003.

[ASC07] Introducing Red Storm. http://www.sandia.gov/ASC/redstorm.html, 2007.

[ASC08] Roadrunner – science at the petascale. http://www.lanl.gov/asc/docs/rr_factsheet.pdf, 2008.

[BB10] BAUER A., BOWMAN M.: CoProcessing. http://www.paraview.org/Wiki/CoProcessing, 2010.

[CBB*05] CHILDS H., BRUGGER E. S., BONNELL K. S., MEREDITH J. S., MILLER M., WHITLOCK B. J., MAX N.: A contract-based system for large data visualization. In *Proceedings of IEEE Visualization 2005* (2005), pp. 190–198.

[Chi07] CHILDS H.: Architectural challenges and solutions for petascale postprocessing. *Journal of Physics: Conference Series 78 012012 78* (2007).

[CPA*10] CHILDS H., PUGMIRE D., AHERN S., WHITLOCK B., HOWISON M., PRABHAT, WEBER G. H., BETHEL E. W.: Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications 30* (2010), 22–31.

[EHG*06] ELLSWORTH D., HENZE C., GREEN B., MORAN P., SANDSTROM T.: Concurrent visualization in a production supercomputer environment. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization 2006)* (2006), 997–1004.

[FJM*99] FRANKE H., JANN J., MOREIRA J., PATTNAIK P., JETTE M.: An evaluation of parallel job scheduling for ASCI Blue-Pacific. *SC Conference* (1999), 45.

[JPH*99] JOHNSON C., PARKER S., HANSEN C., KINDLMANN G., LIVNAT Y.: Interactive simulation and visualization. *IEEE Computer 32*, 12 (1999), 59–65.

[Law09] LAWRENCE LIVERMORE NATIONAL SECURITY, LLC: ASC Sequoia request for proposals. https://asc.llnl.gov/sequoia/rfp/, 2009.

[LZKS09] LOFSTEAD J., ZHENG F., KLASKY S., SCHWAN K.: Adaptable, metadata rich IO methods for portable high performance IO. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy* (2009).

[Ma09] MA K.-L.: In situ visualization at extreme scale: Challenges and opportunities. *IEEE Computer Graphics and Applications 29*, 6 (November/December 2009), 14–19.

[MEGL01] MAY J., EAST D., GRIFFING B., LOUIS S.: Large scale computing at LLNL. *5th Workshop on Distributed Supercomputing* (2001).

[ORN08] Anatomy of Jaguar. http://www.olcf.ornl.gov/computing-resources/jaguar/, 2008.

[PYRM08] PETERKA T., YU H., ROSS R., MA K.-L.: Parallel volume rendering on the IBM Blue Gene/P. In *Proceedings of Eurographics Parallel Graphics and Visualization Symposium 2008* (Crete, Greece, 2008).

[SBC10] SOUMAGNE J., BIDDISCOMBE J., CLARKE J.: An HDF5 MPI virtual file driver for parallel in-situ post-processing. In *Proceedings of the 17th European MPI conference* (2010), pp. 62–71.

[SD01] SHUN DOI TOSHIFUMI TAKEI H. M.: Experiences in large-scale volume data visualization with rvslib. *Computer Graphics 35*, 2 (2001), 10–13.

[SML96] SCHROEDER W. J., MARTIN K. M., LORENSEN W. E.: The design and implementation of an object-oriented toolkit for 3D graphics and visualization. *Proceedings of the 7th conference on Visualization* (1996), pages 93–ff.

[Spr05] SPRINGEL V.: The cosmological simulation code GADGET-2. *mnras 364* (Dec. 2005), 1105–1134.

[YWG*10] YU H., WANG C., GROUT R. W., CHEN J. H., MA K.-L.: In-situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications 30*, 3 (May/June 2010), 45–57.