Lawrence Berkeley National Laboratory

Lawrence Berkeley National Laboratory

Title
In Situ Processing

Permalink
https://escholarship.org/uc/item/3st8x19d

Author
Childs, Hank

Publication Date
2012-11-01

eScholarship.org Powered by the California Digital Library
University of California

In Situ Processing

Hank Childs

Computational Research Division
Lawrence Berkeley National Laboratory,
Berkeley, California, USA, 94720.

Kwan-Liu Ma

University of California, Davis,
Davis, CA, USA, 95616.

Hongfeng Yu

Sandia National Laboratories,
Livermore, CA, USA, 94551.

Brad Whitlock

Lawrence Livermore National Laboratory,
Livermore, California, USA, 94551.

Jeremy Meredith

Oak Ridge National Laboratory,
Oak Ridge, TN, USA, 37831.

Jean Favre

Lawrence Livermore National Laboratory,
Livermore, CA, USA, 94551.

Scott Klasky

Oak Ridge National Laboratory,
Oak Ridge, TN, USA, 37831.

Norbert Podhorszki

Oak Ridge National Laboratory,

Oak Ridge, TN, USA, 37831.

Karsten Schwan

Georgia Institute of Technology,
Atlanta, GA, USA, 30332.

Matthew Wolf

Georgia Institute of Technology,
Atlanta, GA, USA, 30332.

Manish Parashar

Rutgers University,
Piscataway, NJ, USA, 08854.

Fan Zhang

Rutgers University,
Piscataway, NJ, USA, 08854.

October 2012

Acknowledgment

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Some of the research in this work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Legal Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

Abstract

Traditionally, visualization is done via post-processing: a simulation produces data, writes that data to disk, and then, later, a separate visualization program reads the data from disk and operates on it. *In situ* processing refers to a different approach: the data is processed while it is being produced by the simulation, allowing visualization to occur without involving disk storage. As recent supercomputing trends have simulations producing data at a much faster rate than I/O bandwidth, *in situ* processing will likely play a bigger and bigger role in visualizing data sets on the world's largest machines.

The push towards commonplace *in situ* processing has a benefit besides saving on I/O costs. Already, scientists must limit how much data they store for later processing, potentially limiting their discoveries and the value of their simulations. *In situ* processing, however, enables the processing of this unexplored data.

Preface

The material in this technical report is a chapter from the book entitled *High Performance Visualization—Enabling Extreme Scale Scientific Insight* [3], published by Taylor & Francis, and part of the CRC Computational Science series.

Contents

1 Introduction **6**

2 Tailored Co-Processing at High Concurrency **7**

3 Co-Processing With General Visualization Tools Via Adaptors **9**
3.1 Adaptor Design 10
3.2 High Level Implementation Issues 10
3.3 In Practice 11
3.4 Co-Processing Performance 12

4 Concurrent Processing **13**
4.1 Service Oriented Architecture for Data Management in HPC 14
4.2 The ADaptable I/O System, ADIOS 14
4.3 Data Staging for *In Situ* Processing 15
4.4 Exploratory Visualization with VisIt and Paraview Using ADIOS 16

5 *In Situ* Analytics Using Hybrid Staging **16**

6 Data Exploration and *In Situ* Processing **18**
6.1 *In Situ* Visualization by Proxy 19
6.2 *In Situ* Data Triage 20

7 Conclusion **20**

1 Introduction

The terms used for *in situ* processing have not been used consistently throughout the community. *In situ* processing refers to a spectrum of processing techniques. The commonality between these techniques is that they enable visualization and analysis techniques without the significant—and increasing expensive—cost of I/O. On one end of the *in situ* spectrum, referred to as *co-processing,* visualization routines are part of the simulation code, with direct access to the simulation's memory. On the other end of the *in situ* spectrum, referred to as "concurrent processing," the visualization program runs separately on distinct resources, with data transferred from the simulation to the visualization program via the network. Hybrid methods combine these approaches: data is processed and reduced using direct access to the simulation's memory (i.e., co-processing) and then sent to a dedicated visualization resource for further processing (i.e., concurrent processing). Table 1 summarizes these methods.

Technique	Co-processing	Concurrent Processing	Hybrid
Aliases	Tightly coupled Synchronous	Loosely coupled Asynchronous Staging	None
Description	Vis routines have direct access to memory of simulation code	Vis runs on dedicated, concurrent resources and access data via network	Data is reduced via co-processing and sent to a concurrent resource.
Negative Aspects	Memory constraints. Large impact on simulation (crashes, performance).	Data movement costs. Requires separate resources.	Complex. Also shares negatives of other approaches.

Table 1: Summary of *in situ* processing techniques

For approaches that process data via direct access to the simulation's memory (i.e., co-processing and hybrid approaches), another important consideration is whether to use custom, tailored code, written for a specific simulation, or whether to leverage existing, richly featured visualization software. Tailored code is most likely to be well-optimized for performance and memory requirements, for example by using a simulation's data structures without copying them into another form. However, tailored code is often not portable or re-usable, making it tied to a specific simulation code. Utilizing existing, richly featured visualization software is more likely to work well with many simulations. But, typically, the price of this generality may be increased due to the usage of resources, such as memory, network bandwidth, or CPU time.

There are three important *in situ* processing case study scenarios that will be discussed:

- a tailored, co-processing approach at high levels of concurrency, described in Section 2;
- a co-processing approach via an adaptor layer to a richly featured visualization tool, described in Section 3; and
- a concurrent approach in the context of the ADIOS system, described in Section 4.

Hybrid processing is an emerging idea, with little published work; its concepts and recent results are discussed in Section 5. Finally, Section 6 discusses how to use *in situ* processing when the

visualizations and analyses to be performed are not known *a priori*.

2 Tailored Co-Processing at High Concurrency

Tailored co-processing is the most natural approach when implementing an *in situ* solution from scratch and focusing on integration with just one simulation code. The techniques and data structures are implemented with a specific target in mind, resulting in high efficiency in performance and memory overhead, since the simulation's data does not need to be copied. Not surprisingly, many of the largest scale examples of *in situ* processing, to date, have been done using tailored co-processing. Co-processing has been practiced by some researchers in the past with the objective to either monitor or steer simulations [11, 18, 21, 22]. Later results focused on speeding up the simulation by reducing I/O costs [33, 38].

Co-processing has many advantages:

- Accessing the data is very efficient, as the data is already in the primary memory. In contrast, post-processing accesses data via the file system and concurrent processing accesses data via the network.
- Visualization routines can access more data than is typically possible. Simulation codes limit the number of time slices they output for post-processing, since I/O is so expensive. But, with co-processing, the visualization routines can be applied to every time slice, and at a low cost. This prevents discoveries from being missed because the data could never be explored.
- The integrity of the data does need to be compromised. Some simulations produce so much data that the data must be reduced somehow, for example by subsampling; this is not necessary when co-processing.

Co-processing also has some disadvantages:

- Any resources, such as memory or network bandwidth, consumed by visualization routines will reduce what is available to the simulation. Further, existing visualization algorithms are usually not optimized to use the domain decomposition and data structures designed for the simulation code. To be used *in situ*, the algorithms may have to be reformulated so as not to duplicate data and incur excessive interprocessor communication for the visualization calculations.
- The visualization calculations should only take a small fraction of the overall simulation time. If they take longer, the visualization routines can impact the ability of the simulation to advance if they can not finish quickly enough. Although sophisticated visualization methods offer visually compelling results, they are often not acceptable for *in situ* visualization.
- If the visualization routines crash or corrupt memory, then the simulation will be affected—possibly crashing itself.
- The visualization routines must run at the same level of concurrency (in terms of numbers of nodes) as the simulation itself, which may require rethinking some visualization algorithm design and implementation.

In short, it is both challenging and beneficial to design scalable parallel *in situ* visualization algorithms. The resulting visualization should be cost-effective and highlight the best features of interest in the modeled phenomena without constantly acquiring global information about the spatial and temporal domains of the data. A good design must take into account the domain knowledge about the modeled phenomena and the simulation model. As such, a co-processing visualization solution should be developed as a collective effort between the visualization specialists and simulation scientists. Other design considerations for co-processing *in situ* visualization are discussed by Ma [19].

Finally, the recent results of Yu et al. [39] are briefly summarized here,to understand a real-world, tailored co-processing approach. They achieved highly scalable *in situ* visualization of a

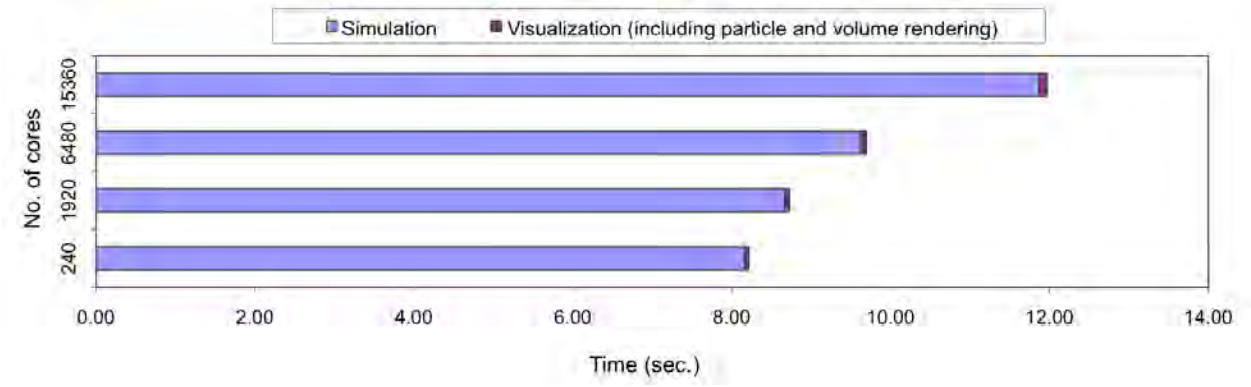

Figure 1: Timing results for both simulation and visualization using different numbers of cores. In this set of tests, the number of grid points on each core is fixed so using more cores provides higher resolution calculations. Visualization is computed every ten timesteps, which is at least 20–30 times more frequent than usual. As shown, the visualization cost looks negligible. The simulation time increases as more cores are used due to the communication required to exchange data after each iteration. The visualization time also grows as more cores are used because of the communication required for image compositing, but it grows at a much slower rate than the simulation time.

turbulent combustion simulation [7] on a Cray XT5 supercomputer, using up to 15,260 cores. Figure 1 shows a set of timing test results. This work was novel outside of its extreme scale, because it introduced an integrated solution for visualizing both volume data and particle data, allowing the scientists to examine complex particle-turbulence interaction, like the type shown in Figure 2. The mixed types of data required considerable coordination for data occurring along the boundaries [39] and highly streamlined routines, like the 2-3 swap compositing algorithm. Finally, their efforts had benefits beyond performance. Before employing *in situ* visualization, the combustion scientists subsampled the data to reduce both data movement and computational requirements for post-processing visualization. This work allowed them to operate on the original full-resolution data.

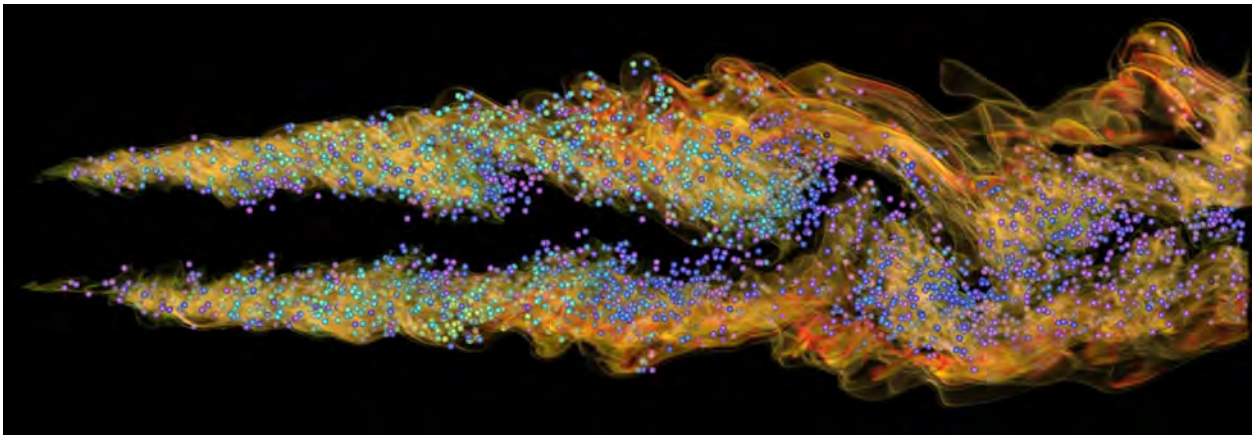

Figure 2: Visualization of the particles and CH_2O field.

3 Co-Processing With General Visualization Tools Via Adaptors

As with tailored code, co-processing with a general visualization system incorporates visualization and analysis routines into the simulation code, allowing direct access to the simulation's data and compute resources. Whereas co-processing with tailored code integrates tightly-coupled visualization routines adapted to the simulation's own data structures, co-processing using general routines from fully featured visualization systems uses an adaptor layer to access simulation data. The adaptor layer is simply a set of routines that the simulation developer provides to expose a simulation's data structures in a manner that is compatible with the visualization system's code.

The complexity of adaptors can vary greatly. If the data structures of the simulation and the visualization system differ, then the adaptor's job is to copy and reorganize data. For example, a simulation may contain particle data, with spatial coordinates and other attributes for each particle. If the simulation's data is organized as an array of structures, and the visualization system requires a structure of arrays, the simulation data must be reorganized to match that of the visualization code, thus consuming more resources. Conversely, adaptors for codes with very similar data structures may share pointers to the simulation's arrays with the visualization code, a "zero-copy" scenario.

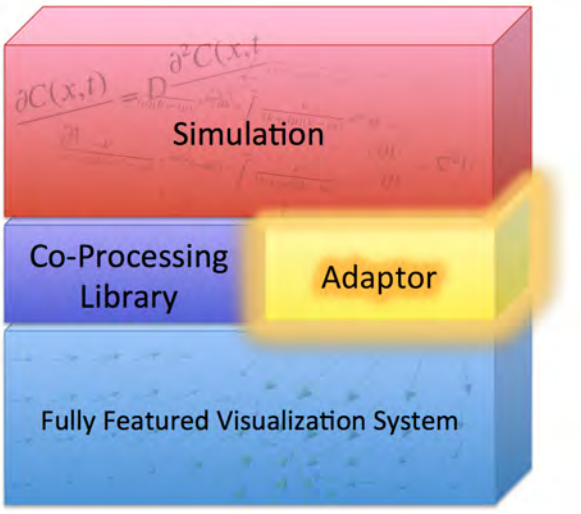

Figure 3: Diagram of the adaptor that connects fully featured visualization systems with arbitrary simulation codes.

Tailored code and general co-processing share many of the same strengths and weaknesses. Both approaches operate in the same address space as the simulation, potentially operating on data directly without conversion. Both approaches cause some extra code to be included in the simulation executable and both may generate additional data, subtracting from the memory available to the simulation. This last property poses problems on many supercomputers, as the trend is to have less and less memory available per core. Memory considerations are exacerbated for co-processing approaches since adaptors may copy data and general visualization tools might have higher memory usage requirements than tailored solutions. The two approaches may also differ in their performance. Tailored code is optimized for the simulation data structures, and so it may not match an optimal layout for already-written general visualization algorithms. But it may be possible with tailored code to optimize these algorithms for the simulation's native data layout.

On the other hand, co-processing with general visualization tools can often be easily enabled

within a simulation through the addition of a few function calls and by adding an adaptor layer. An adaptor code is far less complex to create than the visualization routines themselves, often consisting of only tens of lines of source code. Consequently, adaptor code is cheaper to develop than tailored code and benefits from the use of existing well-exercised routines from fully featured visualization systems. Furthermore, once the data are exposed via the adaptor, they can be visualized in many ways using the general visualization system as opposed to a tailored approach, which might provide highly optimized routines for creating limited types of visualizations.

3.1 Adaptor Design

A successful adaptor layer connecting computational simulations to general purpose visualization tools will have several features inherent in its design:

- *Maximize features and capabilities.* There are numerous use cases for visualization and analysis, particularly *in situ*. Focusing an *in situ* implementation on a specific feature set, for example making movies, will mean that important other use cases will be missing, like interactive debugging. By adapting a general purpose, fully-featured visualization and analysis tool, the *in situ* co-processing tasks gain the capability to use many of these features, as long as the adaptor layer exposes the simulation data properly to the tool.
- *Minimize code modifications to simulations.* The less effort it takes to apply the adaptor to a new simulation, the more easily an *in situ* library can be adopted by a wide range of codes in the HPC community. General purpose tools are already capable of supporting the data model for a wide range of simulation applications, so the barrier for translating data structures should be minimal.
- *Minimize impact to simulation codes when in use.* Ideally, codes should be able to run the same simulation with or without *in situ* analysis. For example, the limited memory situation in distributed systems is exacerbated by co-processing via shared computational nodes, and zero-copy and similar features mitigate the negative impact.
- *Zero impact to simulation codes when not active.* Simulation codes should be able to build a single executable, with *in situ* support built-in, and suffer no detrimental side effects, such as reduced performance or additional dependencies on libraries. Without this feature, users must decide before running a simulation code—or worse, before compiling it—whether or not they wish to pay a penalty for the possible use of *in situ* capabilities, and may choose to forgo the option. Allowing users to start an *in situ* session on demand enables many visualization and analysis tasks, like code debugging, that the user could not have predicted beforehand.

3.2 High Level Implementation Issues

Co-processing is implemented successfully in the general visualization applications ParaView [10] and VisIt [36], two codes built on top of the Visualization Toolkit (VTK) [24]. The codes are similar in many ways beyond their mutual reliance on VTK, including their use of data flow networks to process data. Data flow networks are composed of various interchangeable filters through which data flows from one end to the other with transformations occurring in the filters. In an *in situ* context, the output of the last filter in the data flow network is the product of the *in situ* operation. It may be a reduced data set, statistical results, or a rendered image. For *in situ* use, both applications use adaptor layers to expose simulation data to their visualization routines by ultimately creating VTK data sets that are facades to the real simulation data. ParaView's adaptor layer must be written in C++ and it directly creates VTK objects that will be passed into its data flow networks. VisIt adaptor layers may be written in C, Fortran, and Python and also create VTK data sets that can pass through VisIt's data flow networks.

The co-processing strategies used by ParaView and VisIt are somewhat different where the simulation interacts with the co-processing library. ParaView's co-processing library consists

of a few top-level routines inserted into the simulation's main loop. The simulation data are transformed by the adaptor layer and passed into a co-processing routine, which constructs a data flow network based on an XML specification from ParaView. The results of the pipeline may be written to a disk or staged over the network to concurrent resources, where ParaView may be used for further analysis. VisIt's interaction with the simulation emphasizes a more exploratory style of interaction where, rather than executing fixed data flow networks, the VisIt client dynamically connects to the simulation and directs it to create any number of data flow networks. The end user can be involved with this process and decide which analyses to perform on-the-fly. Since VisIt is able to send plotting commands to the simulation, it integrates differently into the simulation's main loop than ParaView and its adaptor layer is executed only when necessary.

3.3 In Practice

This section explores *Libsim*, the *in situ* library in VisIt, in more detail. *Libsim* wraps all of VisIt's functionality in an interface that is accessible to simulations. In effect, it transforms the simulation into the server in its client-server design. *Libsim* has features consistent with the four design goals listed in Section 3.1: it connects to VisIt, a fully-featured visualization and analysis tool and can expose almost any type of data VisIt understands; it requires only a handful of lines of code to instrument a simulation; it minimizes data copies wherever possible; and it defers loading any heavyweight dependencies until *in situ* analysis tasks begin.

There are two interfaces in *Libsim*: a control interface that drives the VisIt server, and a data interface that hands data to the VisIt server upon request. The control interface is contained in a small, lightweight static library that is linked with the simulation code during compilation, and is literally a front-end to a second, heavier-weight runtime library that is pulled in only when *in situ* analysis begins at runtime. The separation of *Libsim* into two pieces prevents inflation of executable size, even after linking with *Libsim*, while still retaining the ability to access the full VisIt feature set when it is needed.

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 like plot creation and data queries, and letting the client know when the simulation has advanced and that new data is available.

Several aspects of *Libsim* result in benefits for both 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 VisIt being written in C++, no C++ dependencies are introduced into the simulation by linking to the front-end library, which 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, like Python, 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 a list of meshes and fields that the simulation wishes to expose for analysis. The metadata comes from a function in the adaptor layer. Just as in a 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 other functions in the simulation's data adaptor layer are invoked to retrieve only the data needed for the calculation.

VisIt's use of contracts ensures that the minimal set of variables will be exposed by the adaptor layer, reducing any unnecessary data copying from converting variables that are not involved in a calculation. Whenever possible, simulation array duplication is avoided by including simulation arrays directly into the VTK objects, minimizing the impact on the performance and scaling capabilities of simulation codes.

3.4 Co-Processing Performance

An experiment performed by Whitlock, Favre, and Meredith [36] demonstrated that the adaptor approach performs and scales well. Their experiments were conducted on a 216 node visualization cluster with two, six-core 2.8GHz Intel Xeon 5660 processors and 96GB of memory per node. The test system utilizes an InfiniBand QDR high-speed interconnect and a Lustre parallel file system. They ran two types of tests to characterize the performance of this library. The first test determined the cost associated with introducing *Libsim* into the main loop of a prototype simulation. The second test investigated the performance of *in situ* visualization versus I/O in a real simulation code.

Libsim introduces additional overhead in the simulation's main loop: it listens for incoming connections and broadcasts *in situ* analysis requests to distributed tasks. Whitlock et al. wanted to measure the associated time overhead when VisIt was not connected to the simulation, so they added timing code to *Libsim*'s `updateplots` example program. They then ran the example program through 10K main loop iterations. The time spent was small: 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 connection requires loading the VisIt runtime libraries, which imposes a one-time cost of approximately one second.

A study by Childs et al. demonstrated that I/O dominates VisIt's execution time on diverse supercomputer architectures and that VisIt's isocontouring 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, of course, 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 became apparent at modest core counts.

The study by Whitlock et al. [36] was designed to measure the impacts and benefits of *in situ* processing via general visualization tools and adaptors. They instrumented GADGET-2 [27], a distributed-memory parallel code for cosmological simulations of structure formation (see Figure 4). They ran GADGET-2 at three levels of concurrency: 32, 256, and 512 cores to measure *in situ* performance versus I/O performance. They 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. They 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. They ran each of these tests using two sets of initial conditions for GADGET-2, generating particle sets with 16M and 100M 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, they found that the timings for collective I/O are relatively consistent. Independent files are the fastest means for 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, they were able to reduce the amount of data written to a single 12MB image file. From this performance data, they observed 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 runtime vs. single-file I/O.

16 Million Particles			
	32 cores	256 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.

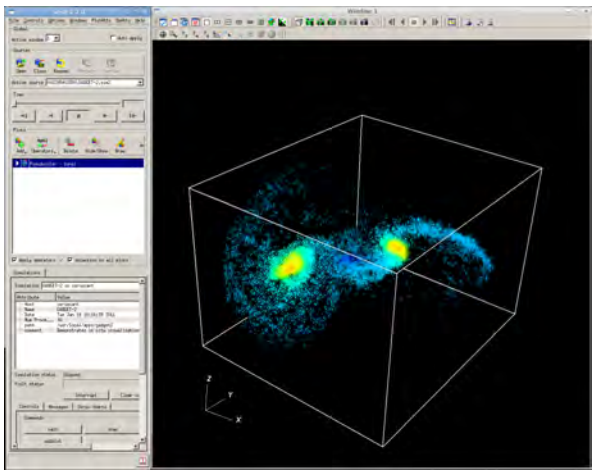

Figure 4: VisIt client connected to GADGET-2 instrumented with *Libsim*. Image source: Whitlock et. al [36].

However, as the number of cores increases, the runtime 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 they could have generated several *in situ* visualizations in the time needed to write a full size snapshot file. This example demonstrates that *in situ* analysis can be an attractive use of extra compute cycles in upcoming exascale computers and a powerful acceleration technique for large scale computations.

4 Concurrent Processing

The concurrent *in situ* approach allows simulation and visualization software to run with minimal interaction, similar in spirit to non-*in situ* usage. The approach resembles a file-based approach, but uses the network for transferring data instead of storage. By keeping the simulation and visualization programs on separate resources, potential errors during processing do not affect the simulation itself. It also allows for scaling the simulation and the visualization code separately. Compared to co-processing, however, there is an extra cost: all data for visualization and analysis has to be transferred to separate processing nodes through the internal network. Fortunately, these costs can be partially ameliorated via smart data filtering [1] and/or compression [16].

In this section, a service-oriented approach to data processing, analysis and visualization is presented, in the context of the ADIOS I/O framework. The framework provides a single I/O API for the simulation to output data (as if writing to files), and the visualization tool to read input data (as if reading from files). It also provides services to transfer data from the output to the reader(s) and a plug-in-based architecture to allow new data processing routines for analysis or visualization steps. ADIOS can organize these steps into a workflow, including a scheduling that takes into account available resources.

4.1 Service Oriented Architecture for Data Management in HPC

The Service Oriented Architecture (SOA) is a methodology for designing software to create interoperable services. It was defined by Thomas Erl of SOA Systems [9]. Some of its requirements must be loosened, however, for performance or complexity reasons when considering data intensive HPC. Specifically, services need not be stateless or opaque to other services. Furthermore, costly service ideas are avoided, such as ontology-based service contracts, per-invocation based service selection, and utilization of an explicit service bus. The following SOA principles have been and are being followed when designing ADIOS for I/O and *in situ* data processing.

- *Abstract and reusable services.* The backbone of a flexible system is its ability to deal with multiple types of input data. For example, a compression service that simply treats data as a bucket of bits, instead of a semantically knowable data structure, will not be sufficient for the needs of extreme scale data management. A self-describing data format, including metadata for introspection, allows services to leverage semantic knowledge about the data to be processed.
- *Service contracts.* The SOA approach requires services to define an interface, or a contract, that defines how the service interacts with external entities. Given the vast range of domains and semantics that need to be supported for an HPC framework, a well understood read/write/query API forms the foundation of the ADIOS infrastructure. Computational scientists are already intimately familiar with the standard data input and output operations. Actions such as opening a file and writing a group of variables are very useful abstractions in providing a familiar yet dynamic and flexible interface to the scientists. In order to address the requirements of data format descriptions there is, however, one additional requirement for the user: the creation of metadata from which the service framework can obtain reasonable semantic information about the data structure.
- *Service location transparency.* While service location transparency is not part of the basic SOA definition, it is an important aspect of attaining the highest level of performance for data intensive computing. In particular, it is paramount that services can be flexibly placed within the data processing pipeline, to maximize the exploitation of available resources.

4.2 The ADaptable I/O System, ADIOS

ADIOS [17] has been developed with the goal of addressing the needs of users of leadership computing facilities (such as those at Oak Ridge and Argonne National Laboratories), as well as users of other large supercomputers. Many past efforts in providing advanced data management infrastructures have burdened scientists with complex APIs and forced them to make extensive changes to their applications. And, while providing adequate functionality, they do not usually perform acceptably at high concurrency. ADIOS has been designed and developed with constant interaction from developers of large-scale simulations from many domains, such as computer science, nuclear physics, combustion, and astrophysics. The primary focus of research has been performance for typical I/O scenarios while maintaining a simple API for a self-describing data format. The result is a framework that provides several methods for performing I/O, allowing users to choose the one best for their needs.

The interface to ADIOS is a simple read/write API. An application can describe the content and organization of an output file in an XML file, which can be created manually by applica-

tion developers. The XML file contains variables with name, type and dimensionality, and also assigned attributes. Alternatively, the program itself can declare the variables by invoking a function. The I/O part of the program code can be generated from the XML automatically, if all of the routines for writing data are in a single location. This is not required, however, as traditional open, write, and close functions can also be used to create the output file. Additionally, the method for writing a group of variables can be chosen in the XML file. This means that if the currently used method is not performing well anymore, then porting the I/O part of the application to a new computer architecture or new I/O system requires only a small change in the XML file. For example, an I/O strategy where all processes write into a single file using MPI-IO, can be replaced with an aggregating strategy, where a small number of processes gather all data and write efficiently into separate files on a parallel file-system. Further, the XML definition of the content allows for an easy way to describe semantic information according to an application domain's predefined storage schema or domain knowledge. Moreover, it allows users to prescribe actions for visualization too, including which tools to use and operations to perform.

The buffering provided by ADIOS works with all of its I/O methods that output in the ADIOS-BP file format, which was designed for efficient utilization of parallel file systems. Buffering allows for using bulk data transfers, which is usually orders of magnitudes faster than writing small chunks of data. The BP format is the fastest format used in ADIOS, in most writing and reading scenarios. It is a self-describing format, where the file content can be discovered and queried from the information stored in the file itself. The format avoids the bottlenecks of reorganizing data from application memory to the file, which are common with other file formats, and also minimizes the contention of file access patterns from many processors to the limited number of file servers.

4.3 Data Staging for *In Situ* Processing

I/O bandwidth, relative to the processing power of current supercomputers, is often insufficient and this trend is predicted to only get worse. Already, storing data to files is often a significant bottleneck during simulation, even with the best I/O solutions. In response, ADIOS developers turned their research focus to *in situ* processing for as many data management tasks as possible before writing data to permanent storage. ADIOS had many advantages in transitioning their product for the *in situ* space, including the separation of API from the choice of I/O methods, the buffering-as-service provided to all methods, and the simplicity and completeness of the API to describe data sets.

There are many benefits to using a set of nodes as a staging area, where I/O can be placed before moving it to disk. First, data can be asynchronously and efficiently moved to staging nodes using Remote Data Memory Access (RDMA), as with DataStager [2]. Second, multiple, parallel applications can simultaneously read a multidimensional array, with arbitrary decomposition, as with DataSpaces [8]. And, third, services can apply data processing actions to data in the staging area and as it is being moved from the (parallel) writer to the (parallel) reader, as with PreDatA [41]. In short, combining data processing actions with I/O in a staging area has been shown to be flexible and efficient. Further, the approach minimizes power and I/O issues and is an active focal point for exascale research. Finally, the user experience is simplified: applications do not need to manage how and where each piece of data is stored, as each process observes and accesses all data sets by defining the geometries of the subsets it wants to write or read.

Virtual data spaces can be used for code coupling. In one example, they facilitated multiphysics simulations by combining different plasma fusion models. Although the models were written for exchanging data via the file system, virtual data spaces allowed them to move data asynchronously from memory to memory. The file-based coupling was changed to a faster memory-to-memory coupling by changing the name of the used method in their respective ADIOS configuration XML files. This service-oriented approach to data movement let the application developer teams keep the code of their physics models separate, avoiding the software

development and maintenance issues of a combined code.

The abstractions of data access and transfer, which are performed asynchronously between tasks, enable task pipelining. This means the data can be processed without worrying about the placement of a task and how it will get the data it needs. However, effective description of the data is essential: a self-describing data format and API for reading and writing data are necessary for a task to understand the data it receives. For example, there is a need for semantically meaningful data chunks or portions, such as the data representing entire variables in simulation codes, and there also must be descriptions about those chunks available when processing actions are carried out. The abstractions of data access and transfer, and the effective data description enable a generic plug-in architecture, where both common and domain-specific analysis and visualization tasks can be created as self-contained codes with a well-defined input and output that behave as services and can be publicly released and reused by many applications.

4.4 Exploratory Visualization with VisIt and Paraview Using ADIOS

In this section, a design similar to a client-server design where the server is parallelized and data is read (in parallel) from files is considered, but the data is read from the network, not files. Once a simulation code opts to use the ADIOS API, ADIOS's plug-in architecture allows it to dynamically switch from a file-based I/O to a mode where data is sent directly to a staging server for processing, enabling exploratory visualization using concurrent *in situ*. VisIt and ParaView both have readers for this ADIOS read API, and support both file-based and *in situ*-based ADIOS usage. From the perspective of these tools, there is virtually no difference in obtaining data from the network instead of the disk. The only change needed was to handle timesteps coming separately from the simulation, instead of having access to all timesteps in a file.

An example of concurrent visualization is described here, namely, how output from the Pixie3D plasma fusion code [6] is visualized (see Fig. 5). This code's output is not directly used for visualization because its internal geometries are not useful when visualized. A separate code, Pixplot, is used to read in the output and, on a much smaller number of processes, transforms and writes the data and meshes in Cartesian coordinates. In the past, this post-processing step generated files, which the user of Pixie3D would study with VisIt in a post-processing fashion. The three components—the Pixie3D simulation, the Pixplot transformation and the VisIt visualization tool—represent a typical scenario for concurrent *in situ* visualization. Since each of these codes used ADIOS for writing and reading data for post-processing exploration, it was straightforward to modify them for concurrent visualization. Pixie3D required no change at all. Pixplot required its processing loop to change from opening one file and iterating over all timesteps to loop for an unknown amount of iterations while there is data coming from the simulation, and read a timestep from the staging area, including possibly waiting if the next timestep is not yet available. The same was true for VisIt's ADIOS reader, which had to be modified to deal with only a single timestep at a time.

Additionally, there is a sequential tailored code for Pixie3D data, Pixmon, which generates 2D slices from the 3D output of Pixie3D itself. These small images are stored on a disk, which is accessible by the ESiMon dashboard [30], so the user of the code can monitor the status of the simulation through a set of 2D snapshots, in close to real-time. For this reason, they used the DataSpaces staging service, which runs separately and can serve two readers of Pixie3D data: Pixplot and Pixmon. Pixplot's output is also stored in the distributed, shared space provided by this same staging server. Several timesteps are stored, so that VisIt or, more precisely, the user driving the VisIt client can explore a timestep long after it was generated by the simulation.

5 *In Situ* Analytics Using Hybrid Staging

To achieve better performance and efficiency, high performance computing (HPC) systems are increasingly based on node architectures with large core counts and deep memory hierarchies. For example, Oak Ridge's Titan system uses 16-core AMD Opteron nodes, and the forthcoming

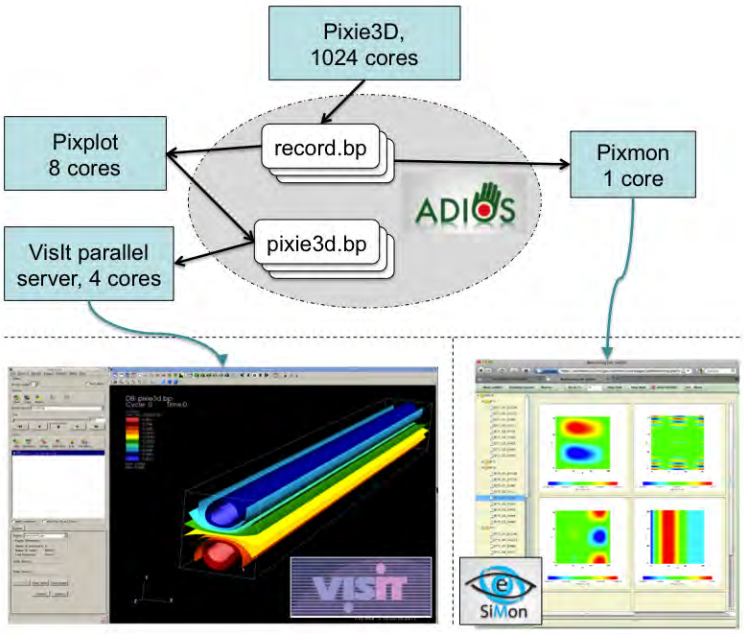

Figure 5: This example has four separate programs—the Pixie3D simulation, Pixplot, Pixmon, and VisIt—strung together as a workflow on the same supercomputer. A fifth program, the staging server (ADIOS), serves the read and write requests of these programs through its API.

petascale supercomputers like Mira at Argonne and Sequoia at Lawrence Livermore will both employ the IBM BlueGene/Q 18-core nodes to exploit more on-chip parallelism. This architectural trend increases the performance gap between on-chip data sharing and off-chip data transfers. As a result, moving large volumes of data using the communication network fabric can significantly impact performance. Further, minimizing the amount of inter-application data exchanges across compute nodes and its consequent network use are critical to achieving high application performance and system efficiency.

Consider a pipeline that consists of more than one task for processing simulation data, for example, one that includes filtering, pre-processing, analytics, visualization, etc. Using an exclusively concurrent *in situ* solution may result in too much network traffic, while a purely co-processing setup may overtax the resources present on compute nodes. Furthermore, certain analytics processing tasks are inherently difficult to scale, such as those requiring frequent collective communications. But, to effectively utilize the potential of current and emerging HPC systems, coupled data-intensive visualization pipelines must exploit compute node-level data locality and core-level parallelism to the greatest extent possible. Each task should be schedulable for either co-processing or concurrent processing, as per each pipeline's processing characteristics. In other words, there must be flexibility in where each data processing action is performed, including on compute nodes, on staging nodes, or both [20, 42]. Achieving such flexibility is challenging, requiring locality-aware mapping of tasks from separate-yet-coupled applications onto the same cores and efficient support for coordination and data exchange between these coupled applications.

Hybrid *in situ* (illustrated in Figure 6) is a technique that combines dedicated cores for co-processing to reduce data movement with dedicated nodes for concurrent data processing. Its goal is to efficiently enable visualization and analysis of data by reducing data movement by moving computation closer to the data. Successful deployment of such a hybrid *in situ* framework can enable novel data processing scenarios, such as moving data customization and filtering operations to where the simulation data is being produced—transforming data as it moves from source to sink, supporting *in situ* coupling to increase intra-node data sharing, and

dynamically deploying data processing plug-ins (binary and/or source code) for execution within the staging area.

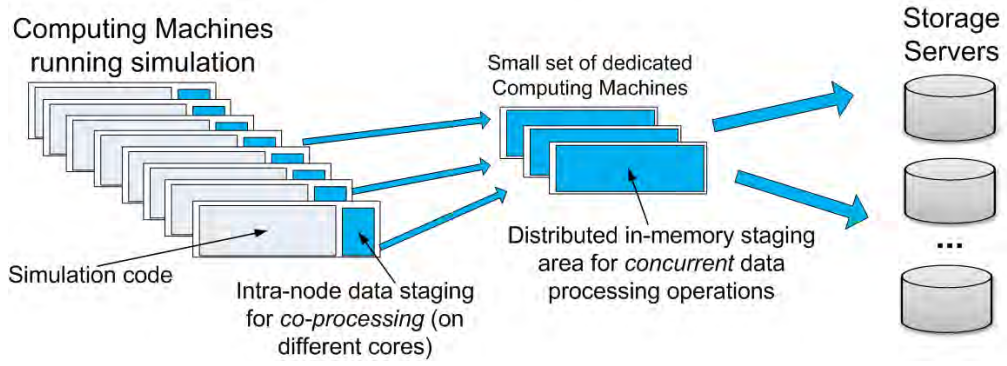

Figure 6: Illustration of hybrid data staging architecture for *in situ* co-processing and concurrent processing.

There have been several research efforts in understanding the efficacy of hybrid *in situ* processing. Early work on commodity networks established that source-based data filtering and compression can substantially reduce the volumes of data being moved and the associated movement overheads [5,37]. Later work [1,4] explored selective co-processing actions that move only the data that is needed from compute to staging nodes. Zheng et al. [42] and Parashar [20] then described the importance of such flexibility and the design of a stream processing system that provided location-flexible operations.

The Co-Located DataSpaces (CoDS) framework [40] uses a workflow-like approach to maximize the on-chip exchange of data for coupled visualization applications by building on DataSpaces [8] and ADIOS (see Section 4). The design of CoDS includes a distributed data sharing and task execution runtime. It employs data-centric task placement to map computations from the coupled application components onto cores so that a large portion of the data exchanges can be performed using the on-node shared memory (see Fig. 7). It also provides a shared space programming abstraction that supplements existing parallel programming models (e.g., message passing) with specialized one-sided asynchronous data access operators, and can be used to express coordination and data exchanges between the coupled applications. The framework builds a scalable, semantically specialized virtual, shared space that is distributed across cores on compute nodes, and provides simple abstractions for coordination, interaction, and data-exchange. The CoDS framework currently supports execution and data-centric task mapping for a workflow that is composed of data parallel application components with regular multidimensional data meshes and domain decompositions.

6 Data Exploration and *In Situ* Processing

In situ visualization is most appropriate when end users know what they want to look for *a priori*. In this case, visualization and analysis may be performed as soon as the data is generated and the end user does not need to be involved. However, the case where the user does *not* know what they want to look for *a priori*—that is, where they want to explore the data—is more problematic, since the amount of time to explore the data is variable and may take hours or days. These time scales make stalling the simulation or keeping a copy of the data in memory unpalatable. Further, some types of exploration may require studying temporal data; keeping multiple time slices in memory almost certainly will exceed memory constraints.

An alternate approach is to transform (and hopefully reduce) the data *in situ* and then explore the data with traditional post-processing. Common approaches involve compression

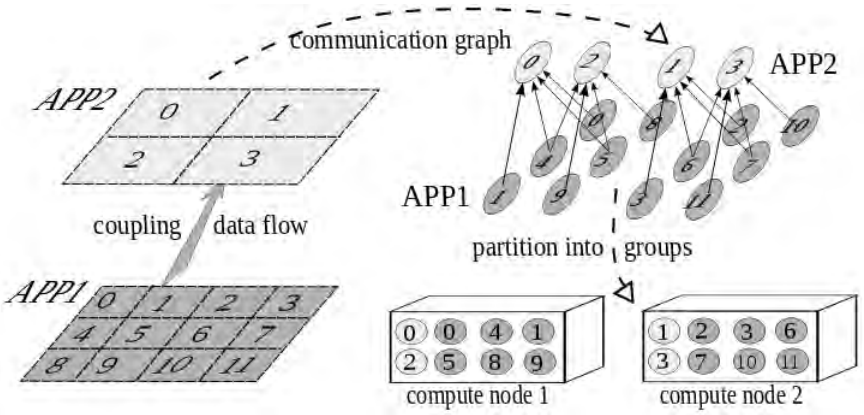

Figure 7: An example of locality-aware, data-centric mapping of two interacting applications onto two multi-core nodes.

Number of cores	240	1920	6480
Simulation time (sec)	8.7204	9.3393	9.5573
I/O time (sec)	9.4563	26.051	52.565
Volume rendering time (sec)	0.3817	0.6155	0.7359
RAF computing time (sec)	1.2775	1.3938	1.3973

Table 3: Timing results for visualization by proxy on a Cray XT5. Each core is assigned a region of $27 \times 40 \times 40$ grid points. Image resolution used is 1024×1024. The number of layers used for RAF is 16. The time for RAF is significantly less than the I/O time, demonstrating the advantage of an *in situ* approach.

(either spatially or temporally), statistical sampling, data subsetting [12,13,23,28,29], or feature identification and tracking [14, 15, 25, 26].

The following subsections describe techniques specifically designed for leveraging *in situ* processing to enable exploration. An algorithm for reducing data while allowing users to modify the characteristics of a volume rendering is described in Section 6.1. An approach where data is compared to a reference simulation and only the most different regions are stored for post-processing is described in Section 6.2.

6.1 *In Situ* Visualization by Proxy

Typically, *in situ* visualization is used to generate an animation that captures features at high temporal resolution. This results in static images rendered based on a fixed set of rendering and visualization parameters and exploration is not possible. This section describes *visualization by proxy*, introduced by Tikhonova et al. in 2010 [31], an approach that enables exploration without requiring the full raw data and a supercomputer. The core idea is that, instead of generating fully rendered results, proxy images are computed that essentially encode isosurface layers by taking into account accumulated attenuation in a view dependent fashion. The resulting proxy images (also called ray attention functions, or RAF, in [31]) enable exploration in the transfer function space. The storage requirement for proxy images corresponds to the number of layers chosen by the user. Ten to fifteen layers were found to be generally sufficient to capture enough details in the data. As a result, the proxy images are typically 10–15 times larger than a single regular image; however, the proxy images are substantially smaller than the size of a video that can offer the same level of exploration.

Tikhonova et al. extended their visualization by proxy technique into a parallel computing environment to support *in situ* visualization [32]. They showed that computing proxy images resembles volume rendering, and can be performed directly on the distributed data out of the simulation without replication. The calculation is as scalable as parallel volume rendering. Table 3 displays some timing results. All simulation data was stored as double-precision floating point values. The proxy images are saved in 32-bit precision. The timing for simulation, I/O, volume rendering, and RAF computing was measured for one timestep. I/O time is how long it took for the simulation to write data to a disk. We can see that volume rendering and RAF computing time accounts only for a small fraction of the total time. The authors noted that, although the fraction for calculating the proxy images is small, real-world scenarios would have the fraction be smaller still, since users would not visualize every timestep in practice.

6.2 *In Situ* Data Triage

Another strategy for using *in situ* techniques to reduce data for subsequent exploration is to conduct a data triage such that the data is stored in a way to effectively supports particular analysis and visualization tasks following the simulation. The basic approach is to reduce and organize the data according to scientists' prior knowledge about certain properties of the simulation and features of interest. The resulting data should be compact and organized specifically to facilitate the offline visualization operations.

A few designs to support co-processing *in situ* data triage have shown opportunities for substantial data reduction and visualization of previously hidden features in the data. Wang et al. [34] computed the importance of each unit block of a flow field, based on how much information and the uniqueness of the information the block contains with respect to other blocks. They showed that the resulting importance field can be used to characterize the flow field, assist in feature extraction process, and guide compression and adaptive rendering. In another study, Wang et al. [35] computed the importance value of each voxel in volume data output from a turbulent combustion simulation according to its distance from the reference features defined by the simulation scientist. First, by factoring the important values into compression, they showed they can achieve impressive savings in storage space. Second, by mapping the importance values to optical properties in the rendering step, the resulting visualization, as shown in Figure 8, allows scientists to see the interaction of small turbulent eddies with the preheated layer of a turbulent flame, a region previously obscured by the multiscale nature of turbulent flow.

The effectiveness of this data triage approach relies on how much prior knowledge about the modeled phenomena and features of interest is available for the design of distance functions for importance measures. *Distance* may have a much broader meaning to consider than distance in spatial domain, temporal domain, and any particular high-dimensional space defined by the scientists, making the *in situ* data triage approach possibly applicable to many simulations.

7 Conclusion

In situ processing refers to a family of techniques that operate on simulation data as it is produced, as opposed to the traditional post-processing model where the simulation stores its data to disk and a visualization program subsequently reads it in. This technique is widely predicted to play a larger and larger role as supercomputers march towards the exascale era, as I/O is increasingly insufficient relative to compute power. Concurrent processing is the *in situ* form where separate resources are allocated for visualization and analysis and data is transferred over the network. Co-processing refers to the form where visualization algorithms have direct access to the memory of the simulation code. It can be done with tailored code that optimizes memory, network bandwidth and CPU time, or through an adaptor layer to a general visualization tool, which brings a wealth of techniques and can work with many simulation codes. Finally, hybrid techniques use elements of concurrent and co-processing to

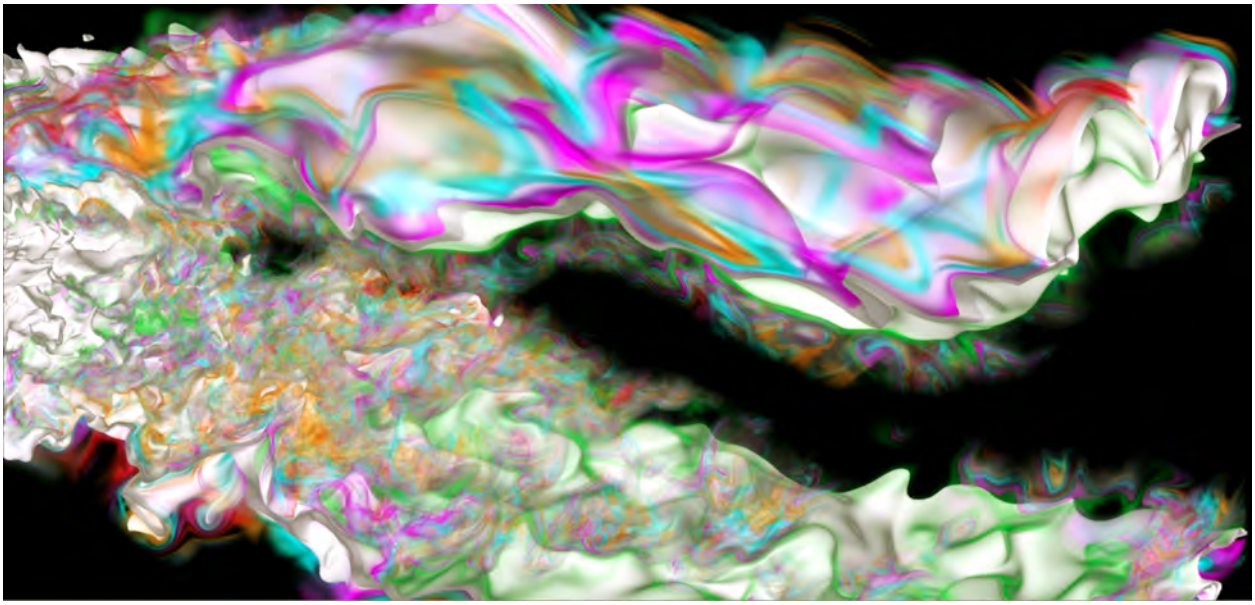

Figure 8: Visualization uncovering the interaction between the small turbulent eddies and the preheated layer of flame. Image source: Wang et al. [35].

provide some of the advantages of concurrent processing while minimizing data movement. All of these techniques have been shown to be successful in real-world settings. Finally, *in situ* can prevent a potential loss of science; it creates an opportunity to process more data than is possible with post-processing.

In situ processing is not a panacea, however, as it is incongruent with explorative use cases, since they rarely have *a priori* knowledge. That said, recent work has looked at using *in situ* processing to extract key information that can later be post-processed. Although these techniques are typically domain-specific or algorithm-specific, they provide hope that exploration can still occur even in configurations where *in situ* processing is required.

References

[1] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan, and Scott Klasky. Just in Time: Adding Value to the IO Pipelines of High Performance Applications with JITStaging. In *Proceedings of Symposium on High Performance Distributed Computing (HPDC)*, pages 27–36, July 2011.

[2] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. DataStager: Scalable Data Staging Services for Petascale Applications. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09, pages 39–48, New York, NY, USA, 2009. ACM.

[3] E. Wes Bethel, Hank Childs, and Charles Hansen, editors. *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, November 2012. `http://www.crcpress.com/product/isbn/9781439875728`.

[4] V. Bhat, M. Parashar, and S. Klasky. Experiments with In-Transit Processing for Data Intensive Grid Workflows. In *8th IEEE International Conference on Grid Computing (Grid 2007)*, pages 193–200, 2007.

[5] Fabián E. Bustamante, Greg Eisenhauer, Patrick Widener, Karsten Schwan, and Calton Pu. Active Streams: An Approach to Adaptive Distributed Systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, page 163, Schoss Elmau, Germany, May 2001.

[6] L. Chacón, D. A. Knoll, and J. M. Finn. An Implicit, Nonlinear Reduced Resistive MHD Solver. *Journal of Computational Physics*, 178:15–36, May 2002.

[7] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K.-L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale Direct Numerical Simulations of Turbulent Combustion Using S3D. *Computational Science and Discovery*, 2(015001), 2009.

[8] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows. In *Proceedings of 19th International Symposium on High Performance and Distributed Computing (HPDC'10)*, June 2010.

[9] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[10] N. Fabian, K. Moreland, D. Thompson, A.C. Bauer, P. Marion, B. Geveci, M. Rasquin, and K.E. Jansen. The ParaView Coprocessing Library: A Scalable, General Purpose In Situ Visualization Library. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV), 2011*, pages 89 –96, Oct. 2011.

[11] A. Globus. A Software Model for Visualization of Time Dependent 3-D Computational Fluid Dynamics Results. Technical Report RNR 92-031, NASA Ames Research Center, 1992.

[12] Luke Gosink, John C. Anderson, E. Wes Bethel, and Kenneth I. Joy. Variable Interactions in Query-Driven Visualization. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of Visualization 2007)*, 13(6):1400–1407, October 2007.

[13] Luke J. Gosink, Christoph Garth, John C. Anderson, E. Wes Bethel, and Kenneth I. Joy. An Application of Multivariate Statistical Analysis for Query-Driven Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):264–275, 2011.

[14] G. Ji and H.-W. Shen. Efficient Isosurface Tracking Using Precomputed Correspondence Table. In *Proceedings of Symposium on Visualization (VisSym) '04*, pages 283–292. Eurographics Association, 2004.

[15] G. Ji, H.-W. Shen, and R. Wegner. Volume Tracking Using Higher Dimensional Isocontouring. In *Proceedings of IEEE Visualization '03*, pages 209–216. IEEE Computer Society, 2003.

[16] Sriram Lakshminarasimhan, Neil Shah, Stéphane Ethier, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F. Samatova. Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data. In *Euro-Par (1)*, pages 366–379, 2011.

[17] J. Lofstead, Fang Zheng, S. Klasky, and K. Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, May 2009.

[18] Kwan-Liu Ma. Runtime Volume Visualization of Parallel CFD. In *Proceedings of International Parallel Computational Fluid Dynamics Conference (ParCFD)*, pages 307–314, 1995.

[19] Kwan-Liu Ma. In Situ Visualization at Extreme Scale: Challenges and Opportunities. *IEEE Computer Graphics and Applications*, 29(6):14–19, November/December 2009.

[20] Manish Parashar. Addressing the Petascale Data Challenge Using In-Situ Analytics. In *Proceedings of the 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, PDAC '11, pages 35–36, New York, NY, USA, 2011. ACM.

[21] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of ACM/IEEE Supercomputing Conference (SC)*, 1995.

[22] J. Rowlan, G. Lent, N. Gokhale, and S. Bradshaw. A Distributed, Parallel, Interactive Volume Rendering Package. In *Proceedings of IEEE Visualization Conference*, pages 21–30, October 1994.

[23] Oliver Rübel, Prabhat, Kesheng Wu, Hank Childs, Jeremy Meredith, Cameron G. R. Geddes, Estelle Cormier-Michel, Sean Ahern, Gunther H. Weber, Peter Messmer, Hans Hagen, Bernd Hamann, and E. Wes Bethel. High Performance Multivariate Visual Data Exploration for Extemely Large Data. In *Supercomputing 2008 (SC08)*, Austin, Texas, USA, November 2008.

[24] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In *VIS '96: Proceedings of the 7th Conference on Visualization '96*, pages 93–100. IEEE Computer Society Press, 1996.

[25] D. Silver and X. Wang. Tracking and Visualizing Turbulent 3D Features. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 3(2):129–141, 1997.

[26] D. Silver and X. Wang. Tracking Scalar Features in Unstructured Datasets. In *Proceedings of IEEE Visualization '98*, pages 79–86. IEEE Computer Society Press, 1998.

[27] V. Springel. The Cosmological Simulation Code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, December 2005.

[28] Kurt Stockinger, E. Wes Bethel, Scott Campbell, Eli Dart, and Kesheng Wu. Detecting Distributed Scans Using High-Performance Query-Driven Visualization. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, October 2006.

[29] Kurt Stockinger, John Shalf, Kesheng Wu, and E. Wes Bethel. Query-Driven Visualization of Large Data Sets. In *Proceedings of IEEE Visualization 2005*, pages 167–174. IEEE Computer Society Press, October 2005.

[30] R. Tchoua, S. Klasky, N. Podhorszki, B. Grimm, A. Khan, E. Santos, C. Silva, P. Mouallem, and M. Vouk. Collaborative Monitoring and Analysis for Simulation Scientists. In *International Symposium on Collaborative Technologies and Systems (CTS)*, pages 235–244, May 2010.

[31] A. Tikhonova, C. Correa, and Kwan-Liu Ma. Visualization by Proxy: A Novel Framework for Deferred Interaction with Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1551–1559, 2010.

[32] A. Tikhonova, Hongfeng Yu, C. Correa, and Kwan-Liu Ma. A Preview and Exploratory Technique for Large Scale Scientific Simulations. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 111–120, 2011.

[33] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R. O'Hallaron. From Mesh Generation to Scientific Visualization: An End-to-End Approach to Parallel Supercomputing. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC)*, 2006.

[34] Chaoli Wang, Hongfeng Yu, and Kwan-Liu Ma. Importance-Driven Time-Varying Data Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1547–1554, 2008.

[35] Chaoli Wang, Hongfeng Yu, and Kwan-Liu Ma. Application-Driven Compression for Visualizing Large-Scale Time-Varying Data. *IEEE Computer Graphics and Applications*, 30(1):59–69, January/February 2010.

[36] Brad Whitlock, Jean Favre, and Jeremy Meredith. Parallel In Situ Coupling of Simulation with a Fully Featured Visualization System. In *Proceedings of 11th Eurographics Symposium on Parallel Graphics and Visualization (EGPGV'11)*, pages 101–109, April 2011.

[37] Y. Wiseman and K. Schwan. Efficient End to End Data Exchange Using Configurable Compression. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, ICDCS '04, pages 228–235, March 2004.

[38] Hongfeng Yu, Tiankai Tu, Jacobo Bielak, Omar Ghattas, Julio C. Lpez, Kwan liu Ma, David R. Ohallaron, Leonardo Ramirez-guzman, Nathan Stone, Ricardo Taborda-rios, and John Urbanic. Remote Runtime Steering of Integrated Terascale Simulation and Visualization. In *HPC Analytics Challenge, ACM/IEEE Supercomputing 2006 Conference (SC06)*, 2006.

[39] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In Situ Visualization of Large-Scale Combustion Simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, May/June 2010.

[40] Fan Zhang, Ciprian Docan, Manish Parashar, Scott Klasky, Nobert Podhorszki, and Hasan Abbasi. Enabling In-situ Execution of Coupled Scientific Workflow on Multi-core Platform. In *Proceedings 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, May 2012.

[41] Fang Zheng, H. Abbasi, C. Docan, J. Lofstead, Qing Liu, S. Klasky, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA—Preparatory Data Analytics on Peta-scale Machines. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, Apr. 2010.

[42] Fang Zheng, Hasan Abbasi, Jianting Cao, Jai Dayal, Karsten Schwan, Matthew Wolf, Scott Klasky, and Norbert Podhorszki. In-situ I/O Processing: a Case for Location Flexibility. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, pages 37–42, 2011.