SANDIA REPORT

SAND2013-1122
Unlimited Release
Printed March 2013

Data Co-Processing for Extreme Scale Analysis Level II ASC Milestone (4745)

David Rogers, Kenneth Moreland, Ron Oldfield, and Nathan Fabian

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

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

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2013-1122
Unlimited Release
Printed March 2013

Data Co-Processing for Extreme Scale Analysis Level II ASC Milestone (4745)

David Rogers
Scalable Analysis and Visualization
Sandia National Laboratories
P.O. Box 5800 MS 1326
Albuquerque, NM 87185-1326
dhroger@sandia.gov

Kenneth Moreland
Scalable Analysis and Visualization
Sandia National Laboratories
P.O. Box 5800 MS 1326
Albuquerque, NM 87185-1326
kmorel@sandia.gov

Ron Oldfield
Scalable System Software
Sandia National Laboratories
P.O. Box 5800 MS 1319
Albuquerque, NM 87185-1319
raoldfi@sandia.gov

Nathan Fabian
Scalable Analysis and Visualization
Sandia National Laboratories
P.O. Box 5800 MS 1323
Albuquerque, NM 87185-1323
ndfabian@sandia.gov

Abstract

Exascale supercomputing will embody many revolutionary changes in the hardware and software of high-performance computing. A particularly pressing issue is gaining insight into the science behind the exascale computations. Power and I/O speed constraints will fundamentally change current visualization and analysis workflows. A traditional post-processing workflow involves storing simulation results to disk and later retrieving them for visualization and data analysis. However, at exascale, scientists and analysts will need a range of options for moving data to persistent storage, as the current offline or post-processing pipelines will not be able to capture the data necessary for data analysis of these extreme scale simulations. This Milestone explores two alternate workflows, characterized as *in situ* and *in transit*, and compares them. We find each to have its own merits and faults, and we provide information to help pick the best option for a particular use.

Contents

Appendix

Figures

Tables

Executive Summary

Milestone 4745, Data Co-Processing for Extreme Scale Analysis, is successfully completed on time and demonstrated against the letter and spirit of the stated Milestone.

Visualization and data analysis on extreme scale platforms presents critical challenges in the management of data generated by simulations and the interface between simulation and data analysis. Computation speed will continue to outpace storage bandwidth, and power management will become much more of a workflow constraint on advanced architectures, so we anticipate that science on extreme scale machines will require a range of data analysis, filtering, and visualization workflows. With these tools, the analysts will determine the best computation profile for specific problems. In particular, in addition to current practices of post-processing and constrained writes, customers will need *in situ* and *in transit* workflows that allow them flexible options for getting results to persistent storage.

Milestone 4745 provides important study of the behavior of new workflows for visualization and data analysis. In particular, we examine the performance of two proposed workflows: an *in situ* workflow in which visualization and data analysis is coupled directly with a simulation as a library, and an *in transit* workflow in which visualization and data analysis is a separate service connected to the simulation via a network. Each workflow has its own characteristics, and our study details empirical evidence on their respective performances.

As we look ahead to the extreme architectures planned by ASC, it is critical to understand the strengths and weaknesses of these proposed analysis approaches, and to design experiments that will help us understand how to enhance scientific discovery on these new architectures. Milestones such as this one provide important foundations for evaluating our current technical approaches, and for strategic development of analysis capabilities for future architectures.

For this Milestone, we explored the performance characteristics of two proposed technologies developed with significant contributions from Sandia National Laboratories. These technologies leverage existing investments in data analysis, visualization and I/O research, and are part of a long term strategic plan for addressing ASC analysis needs across a spectrum of scales, codes, and science domains. We employ two critical software technologies developed with significant contributions from Sandia National Laboratories. First, we use the Catalyst library to provide *in situ* visualization and data analysis directly to a running simulation. Second, we use the Nessie framework to establish an *in transit* visualization and data analysis service connected to a running simulation.

Our primary motivation for this use case was to use a highly scalable code that provided an example of real physics, so that we could determine performance characteristics when applied against real data. There is significant community development in min-apps available in the community that scale well across large architectures, but whose output is not representative of a real science code. CTH is utilized as a test code for ASC platforms, and has been used in several exploratory in-situ tests in the past. Because of this, the team opted to

use CTH as a driver for this large scale experiment.

Our empirical study comprises over 10 million core hours of running an instrumented simulation and data analysis use case. This use case, involving the fragmentation analysis of an explosion simulated in the CTH shock physics code, is designed in conjunction with a Sandia analysis customer as an exemplar of scientific work.

In addition to demonstrating the scalability of our frameworks, our study also provides insightful comparisons between the *in situ* and *in transit* workflows and the trade-off point between them. We also consider other important parameters such as memory overhead, initialization time, and scheduling.

This SAND report presents the full results of our milestone work and is available to anyone.

Acknowledgements

Our team would like to thank the CTH team and developers who have engineered and delivered this code over the years. As an extremely scalable, reliable science code, CTH provides a powerful basis for performing experiments in large scale data analytics, and we would like to thank the CTH team for their work, which has made this milestone possible.

1 Official Milestone

The following is the ASC milestone our work implements.

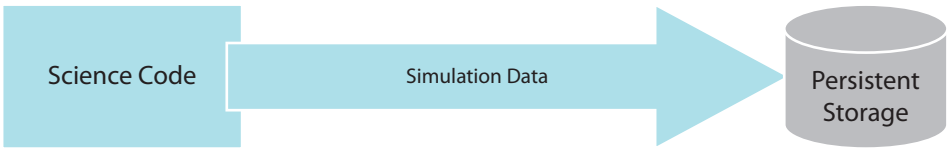
> ASC calculations produce complex datasets that are increasingly difficult to explore and understand using traditional post-processing workflows. To advance understanding of underlying physics, uncertainties, and results of ASC codes, SNL must gather as much relevant data as possible from large simulations. This drives SNL to couple data analysis and visualization capability with a running simulation, so that high fidelity data can be extracted and written to disk. This Milestone evaluates two approaches for providing such a coupling:
>
> 1. In-situ processing provides "tightly-coupled" analysis capabilities through libraries linked directly with the simulation. SNL has collaborated on developing an in-situ capability designed for this purpose.
> 2. In-transit processing provides "loosely-coupled" analysis capabilities by performing the analysis on separate processing resources. SNL provides this capability through a "data services" capability designed for this purpose.
>
> SNL will engineer, test and evaluate customer-driven operations on large-scale data created by a running simulation. The data operations will be performed by instrumented versions of both the in-situ and in-transit solutions, with the resulting performance data published and made available to the ASC community.
>
> A program review will be conducted, and its results documented. A report will be submitted as a record of milestone completion.

Our "customer-driven operations" are encapsulated in the use case given in Section 4, which is designed by Jason Wilke, one of our customers in Mechanical Engineering, to represent a typical shock physics analysis.

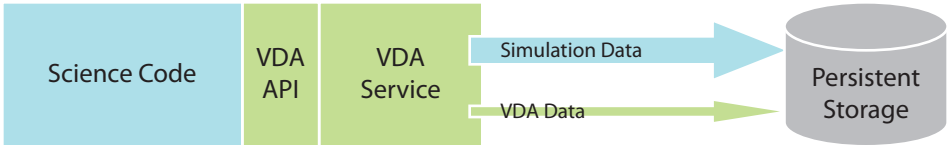
The results of our instrumentation on the coupled simulation and visualization and data analysis are given in Section 5. These results come from experimental runs on the Cielo supercomputer [10] ranging up to over 64 thousand cores. Our measurements record the cost of coupling visualization and data analysis to a running simulation in terms of added execution time and memory overhead, which satisfies the deliverables of the milestone.

The milestone describes "two approaches" for coupling a simulation with visualization and data analysis. These approaches refer to the workflow employed in the full process from simulation to data analysis. These workflows are summarized in Figure 1.

The workflow in Figure 1a is a traditional **offline post-processing** in which the simulation stores all its results to disk for later visualization and data analysis. This is what the milestone means by the "traditional post-processing workflows" with which "complex datasets... are increasingly difficult to explore and understand." Approach 1 of the milestone refers to the **embedded *in situ*** workflow in Figure 1b in which a visualization



(a) Traditional offline post-processing visualization and data analysis.



(b) Embedded *in situ* visualization and data analysis (VDA).

(c) Service-oriented *in transit* visualization and data analysis (VDA).

Figure 1: Traditional and emerging workflow diagrams showing the flow of information from simulation to persistent storage. In all cases data will later be retrieved from storage and further analyzed.

and data analysis library is coupled with the running simulation code. Approach 2 of the milestone refers to the ***in transit*** workflow in Figure 1c in which data is transfered from the simulation job to a separate visualization and data analysis service.

All three workflows are considered in our measurements. The offline post-processing workflow is implemented using the standard CTH I/O to write full meshes to storage and then reading those meshes in a scripted parallel ParaView job. The embedded *in situ* workflow is implemented with the **Catalyst** *in situ* library discussed in Section 2. The *in transit* workflow is implemented with the **Nessie** I/O service described in Section 3.

2 Catalyst

Catalyst is a general purpose, full-featured library that leverages existing implementations of analysis and visualization capabilities. The intent in doing this is threefold. First, by leveraging existing visualization and data analysis libraries we can benefit from the accumulation of over two decades of visualization research and development. Second, by making the library general purpose we can quickly apply our *in situ* visualization and data analysis capabilities to many simulations as opposed to a single simulation. Third, by using our existing code we can integrate the *in situ* tools with our traditional post-processing tools to provide interfaces that users are already familiar and comfortable with and to apply scalable algorithms designed for *in situ* with our post-processing tools.

Catalyst is a C++ library with an API available in C, FORTRAN, and Python. It is built atop the Visualization Toolkit (VTK) [34] and ParaView [6]. This means that Catalyst takes advantage of a large number of algorithms including writers for I/O, rendering algorithms, and processing algorithms such as isosurface extraction, slicing, and flow particle tracking. Catalyst uses ParaView to implement and manage the visualization and data analysis, which is defined using a visualization pipeline [18]. Although it is possible to construct pipelines entirely in C++, a more flexible approach is defining pipelines with Python scripts.

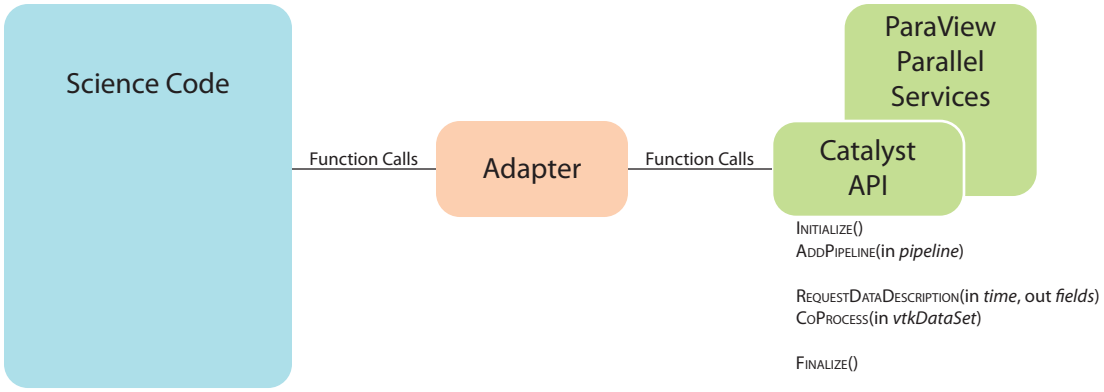


Figure 2: Coupling a simulation with Catalyst.

2.1 Catalyst Architecture

Catalyst boils the ParaView and VTK architecture down into five API calls that manage all the processing required for operating a processing pipeline. Initialize and Finalize are expected calls when dealing with MPI; this is where Catalyst will first access MPI World. RequestDataDescription and CoProcess handle the hand-shake from the simulation code to Catalyst. RequestDataDescription passed current time information to Catalyst, and Catalyst passes back whether or not it should process and which fields it needs. This may allow the simulation code, through the adaptor described below, to efficiently pass only what is necessary at that time. The CoProcess call is where control is passed to Catalyst for processing. This call will hang until Catalyst finishes and returns control to the simulation.

The AddPipeline call is where a majority of the work is done. Although there is usually only one pipeline added to Catalyst, it is possible to add several. It is also common for this pipeline to be written in Python, because this is the native scripting language for ParaView. It is also possible for the pipeline to be coded in C++. While there is overhead cost to using Python code in the pipeline here, it is very low due to its nature as a glue code combining the C++ filters which are written in VTK. The advantage to using Python is that a scripted pipeline can change alongside the simulation input deck without needing to recompile.

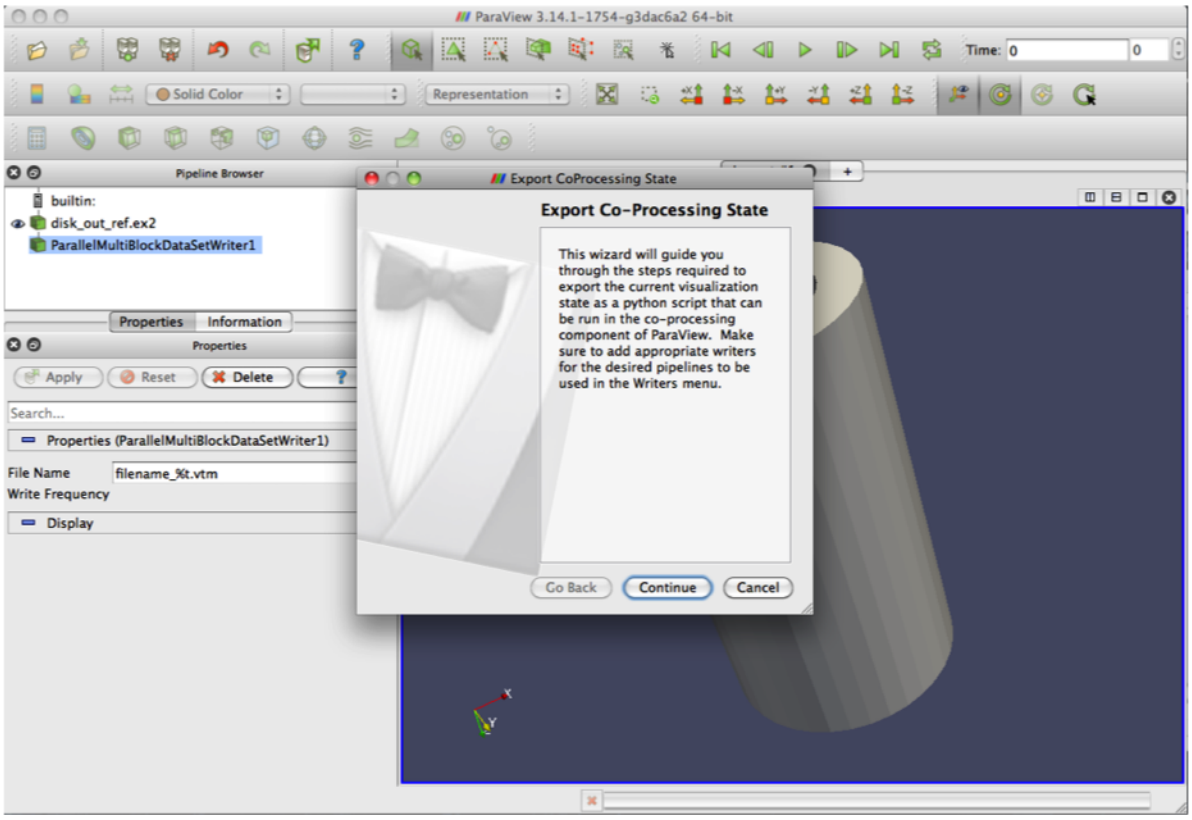


Figure 3: Wizard plugin within ParaView to export interactive traces as Catalyst pipelines for use within a coupled simulation.

In addition to the advantages of writing Python pipeline scripts by hand, there is a well supported plugin for ParaView that will automatically create Catalyst Python scripts automatically from within the GUI, based on what the user does interactively, as shown in Figure 3. This plugin reads in a file and creates the same code object provided by the adaptor (see Section 2.2). The rest of the pipeline operates identically whether the data has been read in from a file or it is coming from an in-memory *in situ* transfer.

The plugin creates images for each view open at the time the script is exported. To write files, one can create objects within the pipeline that act as file write points. In general, these are placed at the end of a long chain of processing to store the resulting processed data, but it is also possible to splice these file writes into in the pipeline at any point, so that intermediate data can be preserved. Each file and image writer can write at an independent frequency, so that the output can be tuned precisely by the analyst. For example, images

(which tend to be very small) can be written out frequently, while large, detailed data files can be written out infrequently.

2.2 Simulation Adaptor

Since Catalyst will extend to a variety of existing simulation codes, our design does not expect its API to easily and efficiently process internal structures in all possible codes directly. Our solution is to rely on adapters — which are small pieces of code written for each new linked simulation — to translate data structures between the simulation's code (for our use case the CTH shock physics code) and Catalyst's VTK-based architecture, as shown in Figure 2. The adapter must also establish a mechanism that allows the simulation to define a visualization pipeline and periodically invoke the data analysis while running the simulation, which in our CTH adapter we control through the CTH input deck.

To conserve memory, our adapter directly interfaces the visualization and data analysis code to the data structures defined by CTH. This interface is challenging because although the blocks of data are represented sequentially in both CTH and VTK, the multidimensional order is different. To address this, our adapter contains an interface wrapper above the standard VTK array. The wrapper reimplements the array's accessor functions to handle the order difference between the two systems. Although there is a minor overhead in additional pointer arithmetic and virtual method calls, it saves us from a deep memory copy.

2.3 References

The Catalyst library and the algorithms we use within CTH are an accumulation of several years work, starting with the development of fragment analysis algorithms with our post-processing tools [13, 19, 21], described in more detail in Section 4. Subsequent work lead to the development of Catalyst [12] and the scaling of algorithms used in conjunction with CTH [11].

3 Nessie

The NEtwork Scalable Service Interface, or Nessie, is a framework for developing parallel client-server data services for large-scale HPC systems [16, 26, 29].

Nessie was originally developed out of necessity for the Lightweight File Systems (LWFS) project [27], a joint effort between researchers at Sandia National Laboratories and the University of New Mexico. The LWFS project followed the basic philosophy of "simplicity enables scalability", the foundation of earlier work on lightweight operating system kernels at Sandia [32]. The LWFS approach was to provide a core set of fundamental capabilities for security, data movement, and storage and afford extensibility through the development of additional services. For example, systems that require data consistency and persistence might create services for transactional semantics and naming to satisfy these requirements. The Nessie framework was designed to be the vehicle to enable the rapid development and deployment of such services.

Although Nessie was originally designed for I/O and system services, it is also useful for development of application-specific data services. For example, we developed services for staging checkpoint data [23, 24, 31], HPC database integration [30], interactive visualization [25], network traffic analysis, and most recently CTH *in transit* analysis [22]. A recent paper describes these services in detail [17].

This section includes a brief description of the Nessie architecture and APIs followed by a more detailed description of the *in transit* service for CTH data analysis using ParaView.

3.1 Nessie Architecture

Because Nessie was originally designed for I/O systems, it includes a number of features that address scalability, efficient data movement, and support for heterogeneous architectures. Features of particular note include 1) using asynchronous methods for most of the interface to prevent client blocking while the service processes a request; 2) using a server-directed approach to efficiently manage network bandwidth between the client and servers; 3) using separate channels for control and data traffic; and 4) using XDR encoding for the control messages (i.e., requests and results) to support heterogeneous systems of compute and service nodes.

A Nessie service consists of one or more processes that execute as a serial or parallel job on the compute nodes or service nodes of an HPC system. We have demonstrated Nessie services on the Cray XT and XE systems at Sandia National Laboratories (SNL) and Oak Ridge National , the Cray XT4/5 systems at Oak Ridge National Laboratory, and a large InfiniBand cluster at SNL. The Nessie RPC layer has direct support of Cray's SeaStar interconnect [7], through the Portals API [8]; Cray's Gemini interconnect [4]; and InfiniBand [5].

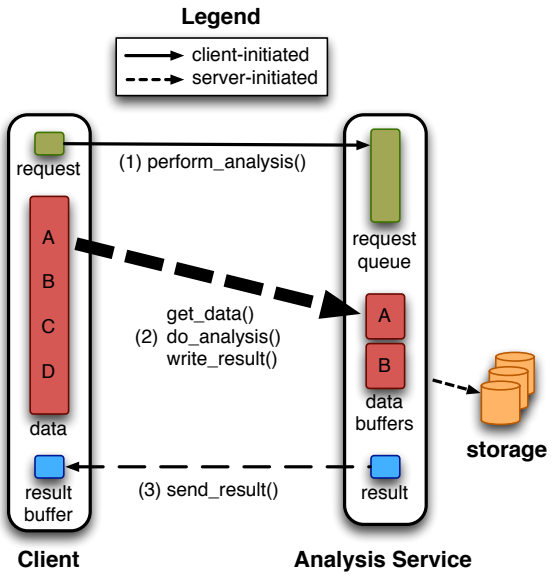


Figure 4: Conceptual protocol for Nessie service doing analysis. The server fetches bulk data through RDMA commands until it has satisfied the request. After completing the data transfers, the server processes the data, writes analysis results to disk, then sends a small "result" back to the client indicating success or failure of the operation.

Nessie API

The Nessie API follows a remote procedure call (RPC) model, where the client (i.e., the scientific application) tells the server(s) to execute a function on its behalf. Nessie relies on client and server stub functions to encode/decode (i.e., marshal) procedure call parameters to/from a machine-independent format. This approach is portable because it allows access to services on heterogeneous systems, but it is not efficient for I/O requests that contain raw buffers that do not need encoding. It also employs a 'push' model for data transport that puts tremendous stress on servers when the requests are large and unexpected, as is the case for most I/O requests.

To address the issue of efficient transport for bulk data, Nessie uses separate communication channels for control and data messages. In this model, a "control" message, also known as a request, is typically small. It identifies the operation to perform, where to get arguments, the structure of the arguments, and perhaps the data itself (if the data is small enough to fit in the fixed-sized request). In contrast, a data message is typically large and consists of "raw" bytes that, in most cases, do not need to be encoded/decoded by the server. For example, Figure 4 shows the transport protocol for an *in transit* service that performs analysis on simulation results data.

The Nessie client uses the RPC-like interface to push control messages to the servers, but the Nessie server uses a different, one-sided API to push or pull data to/from the client. This protocol allows interactions with heterogeneous servers and benefits from allowing the server

to control the transport of bulk data [15, 35]. The server can thus manage large volumes of requests with minimal resource requirements. Furthermore, since servers are expected to be a critical bottleneck in the system, a server directed approach affords the server optimizing request processing for efficient use of underlying network and storage devices – for example, re-ordering requests to a storage device [15].

While it is not strictly necessary on systems that have homogenous clients and servers, we use XDR encoding to provide portable serialization of arguments for the request arguments. This was a design decision made early in the project that allow the client to send arbitrary C-like data structures to the server with minimal development effort. At the time, we were implementing file services for a system where the service nodes were a different architecture (and had different endienness) than the compute nodes. In this case, byte-swaps were necessary for the control structures. Since *rpcgen*, the function that generates the serialization code is pervasive in Unix environments and has been in use for more than a decade, it was the logical choice for argument marshaling.

NNTI API

The Nessie Network Transport Interface (NNTI) provides a portable, lightweight, interface for RDMA operations on HPC platforms. Our current implementation includes support for the Cray Seastar, InfiniBand, Cray Gemini, and IBM DCMF interconnects. The APIs include commands to open and close the interface, connect and disconnect to a peer, register and deregister memory buffers, and finally asynchronously transport (through put, get, and wait commands) bulk data.

Figure 5 illustrates the software stack for applications using Nessie and NTTI. The NNTI library sits below the Nessie RPC library to enable portability across HPC interconnects. In addition to the Nessie library, NNTI is also used by the ADIOS/DataStager [2] to provide the same level of portability and performance. We are also discussing NNTI as the lowest-layer network transport for the Sirocco parallel file system, another ASC project at SNL.

CommSplitter API

The CommSplitter library was designed to overcome a security model limitation in the Gemini interconnect. On current Gemini systems, user-space applications are not allowed to communicate, even if both applications are owned by the same user. We requested this feature and at the time of this writing, Cray is addressing this issue to better support data services in future versions of Gemini. In the mean time, we overcame that limitation by launching our jobs in Multiple Program, Multiple Data (MPMD) mode. MPMD mode enables a set of applications to execute concurrently, sharing a single MPI Communicator. The problem with this approach is that legacy applications were not designed to share a communicator with other applications. In fact, most HPC codes assume they have exclusive use of the `MPI_COMM_WORLD` communicator. When this is not the case, a global barrier,

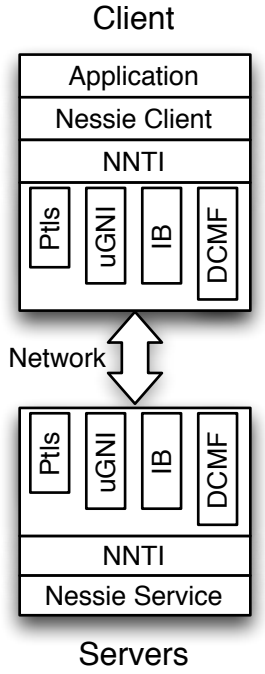


Figure 5: Software stack for applications using the Nessie and NNTI libraries.

such as an `MPI_Barrier` function will hang because the other applications did not call the `MPI_Barrier` function.

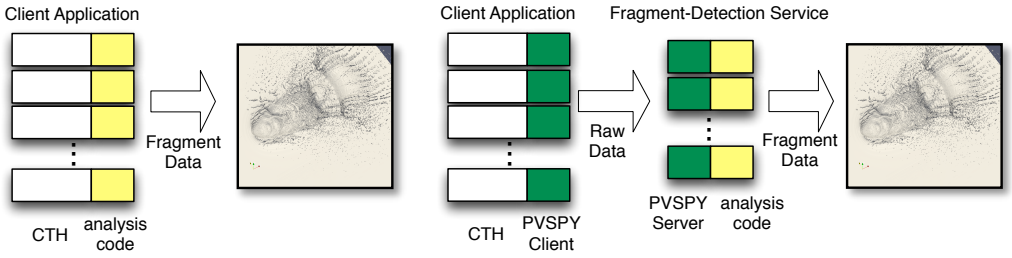
To address this issue, we developed the CommSplitter library to allow applications to run in MPMD mode while still maintaining exclusive access to a virtual `MPI_COMM_WORLD` global communicator.

The CommSplitter library identifies the processes that belong to each application, then "split" the real `MPI_COMM_WORLD` into separate communicators. The library then uses the MPI profiling interface to intercept MPI operations, enforcing the appropriate use of communicators for collective operations.

No changes are required to the application source code to enable this functionality. The user simply links the CommSplitter library to the executable before launching the job. The library has no effect on applications that are not run in MPMD mode.

3.2 CTH *in transit* analysis

In this milestone, we used Nessie to construct an *in transit* CTH data analysis service. The data analysis service exists as its own parallel job that communicates with a parallel CTH job using the Nessie APIs. The *in transit* CTH data analysis library is a drop-in replacement for the PVSPY library [20] used for *in situ* data analysis. This makes comparing *in situ* and



(a) *In situ* data analysis (b) *In transit* data analysis

Figure 6: Comparison of *in situ* (a) and *in transit* (b) fragment detection for the CTH shock physics code.

in transit approaches extremely convenient since it only requires the user to link a different library when compiling CTH. Instead of executing the data analysis on the same compute nodes of the CTH application (as the *in situ* library does), the *in transit* library marshals requests, sends data to the data analysis service, and performs all the data analysis on the separate application. Figure 6 illustrates this process for data analysis that does fragment detection.

For efficiency reasons, the *in transit* PVSPY client implementation does not simply forward all the functions to the service. In many cases, the client maintains metadata to avoid unnecessary data transfers. For example, the PVSPY API includes "setup" functions for initializing data structures, assigning cell and material field names, and setting cell and material fields pointers. Not all of these functions require an immediate interaction with the data service. In fact, the only operations that require bulk data transport are the operations that synchronize the metadata and the data between the client and the server. These operations occur just before a *pvspy_viz* operation that initiates the ParaView coProcessing on the remote service.

The *in situ* version of PVSPY has the notion of a "CTH source" that allows the data analysis code to work directly on the memory of the CTH application without making any copies. Since the *in transit* service does not have access to the physical memory of the CTH application, we created a virtual CTH source on the server that emulates the data structures on the CTH application. That allows the service to use use the same PVSPY library that the client uses in the *in situ* data analysis.

With the exception of the operations to transfer metadata and data to the analysis service, all remote operations are asynchronous, allowing the data analysis on the service to execute in parallel with computation on the CTH application. If one remote visualization operation is not complete by the time CTH is ready to do another visualization operation, CTH has to wait.

3.3 Related Efforts

There are a number of efforts to develop technologies for staging data or providing data services that are related to, and in some cases derived from, Nessie. The two primary competitors of Nessie include the data-staging portions of the ADIOS library from ORNL, and the Glean library from Argonne National Laboratory.

The Adaptable IO System (Adios) is an I/O library that separates the I/O interface from the underlying I/O operations. Using XML configuration files, the user can specify, at runtime, the methods used to perform I/O. For example, MPI-IO, POSIX-IO, netCDF, or their own BP method are all options the use can select. ADIOS also includes methods data transport that allow the staging and processing of data in the same way as Nessie. This staging technology, called DataTap [1] and DataStager [2] derive from early work on Nessie as part of a joint collaboration between GA Tech, SNL, and ORNL. More recent versions of DataStager are also using the NNTI API to provide portable RDMA transport.

The DataSpaces [9] project uses the memory on data-staging nodes as a scratch space for communicating and sharing data among multiple applications. This work is closely aligned with the ADOIS efforts at ORNL. The primary focus is to use asynchronous IO to move data into a staging area and then having a different application retrieve data at a later time.

A recent effort called Glean [36] from Argonne is a start towards both accelerating IO performance and integrating functionality, such as data analysis routines, at the right place transparently. It is very similar to PreDatA, but extends the location of operations to potentially beyond the current machine.

Most of the related efforts focus primarily on the I/O benefits of data-staging, but have not put a tremendous effort into complex analysis. The majority of the analyis codes perform relatively simple statistics and/or visualization. With Nessie, and this effort in particular, we treat the service as a complex parallel application that includes all the synchronization, communication, and scaling issues inherent in HPC parallel applications. These issues require a level of detail and performance tuning that is lacking in other efforts.

A second distinction between our approach and related work is a general philosophy on supporting APIs. While the other approaches, like ADIOS, provide a unified I/O API, our approach is to provide *in transit* implementations of commonly used APIs so the code does not have to change the source code. In this milestone, we implemented an *in transit* version of the pvspy API, the same API used to perform the *in situ* experiments. It was this approach that made comparison between the *in situ* and *in transit* approaches so easy to perform.

3.4 Nessie Availability

The Nessie software is available, open source, as part of the Trilinos I/O Support Package [28]. The package includes the NNTI, Nessie, CommSplitter libraries as well as a collection of CMake macros and other tools for constructing application-specific data services.

4 Experiment Driver

For the purposes of this milestone, we explore the problem of characterizing fragments in an explosion simulation. Simulation is a vital part in understanding shock physics. Although experimentation will always be a necessary tool for scientific inquiry and corroboration, the amount of data we can retrieve with experimentation is limited. Experiments in shock physics usually involve high energy, high velocities, and high variability, all of which hinder detailed, accurate, and repeatable observations during the experiment. When measurements cannot be taken during the experiment, they must be taken after the experiment by observing the remaining material. Much can be learned in the manner, but the transient states during the experiment are lost.

Another limiting factor of experimentation is its high cost and slow turnaround. To create shock physics experiments, physical devices must be fabricated. These devices are then usually destroyed during the experiment. Safety and political issues also often plague shock physics experiments. In some cases, experimentation is simply not feasible. Thus, simulation plays a major role in shock physics analysis.

In this milestone, we use an example simulation of an exploding pipe shown in Figure 7. This similation problem is provided to our group by Jason Wilke. In addition to well representing the kinds of simulation and analysis done at Sandia National Laboratories, this simulation provides interesting results and many different levels of refinement, which allows us to scale the problem from less than 100 cores to well over 10,000 cores.

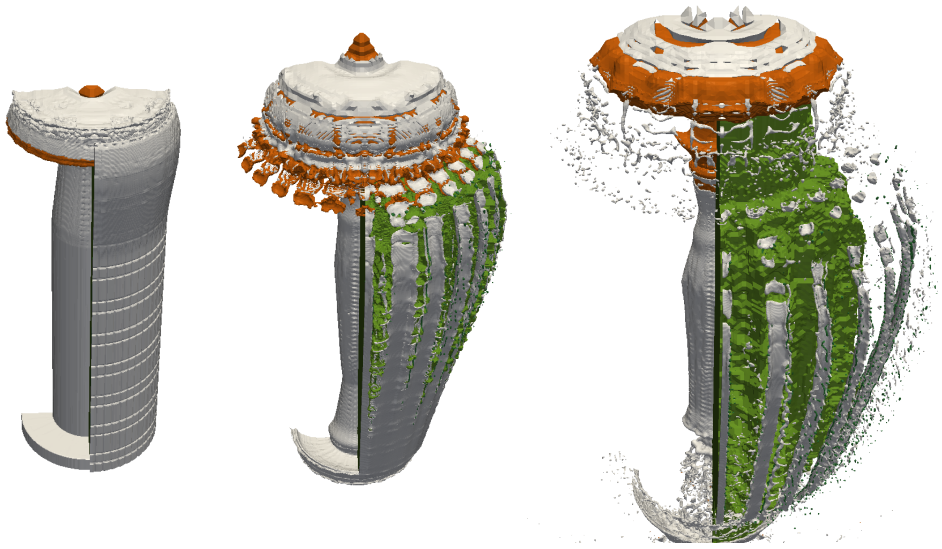


Figure 7: Simulation of an exploding pipe, which presents many prototypical fragment analysis challenges.

One of the most important features in shock physics analysis is material fragments. The physical properties of the fragments, including mass, volume, and shape, as well as their

trajectories, can all be important. In particular, shape can be an important characteristic. Consider the example fragments given in Figure 8. The top fragment is long and sharp, making it more likely to penetrate objects. In comparison, the bottom left fragment is rounded and could have less damage potential. However, the U-shaped fragment in the bottom right may be harmful depending on the scenario, but could be difficult to distinguish from the round fragment in many shape metrics.

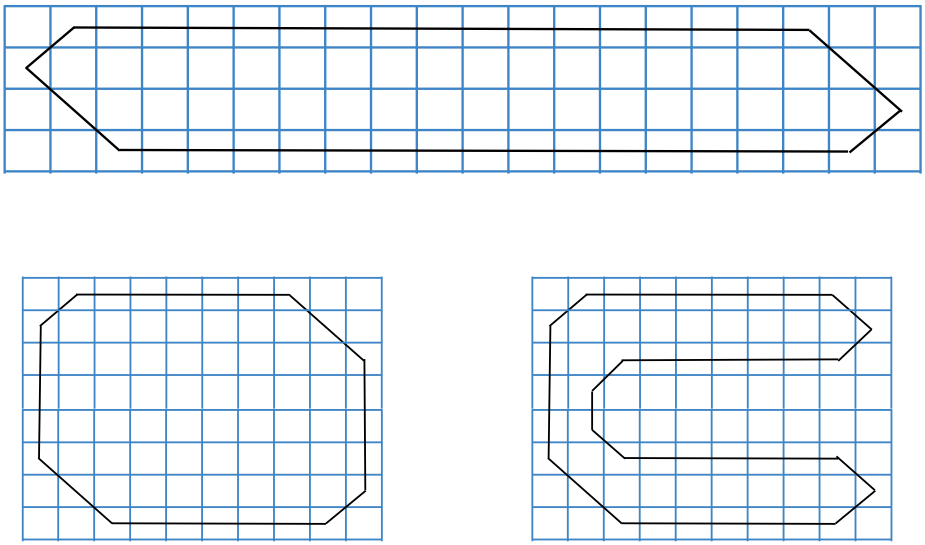


Figure 8: Examples of potential fragments that we would like to characterize.

The simulation code we use is CTH [14,33]. It is an Eulerian shock physics code that uses an adaptive mesh refinement (AMR) data model. These adaptive finite volumes can take up different amounts of space depending on where they are in the model and how closely the simulation is refining the space.

In order to correctly find fragments, we must first determine what is and is not a fragment. The simulation operates on a finite volume and comprises a set of simulated materials, which each take up a certain fraction of finite cells within that volume. We treat any connected region of cells with material volume fraction above a given threshold as a fragment of that material. Generally speaking, when a simulation begins, each material comprises usually one connected region, which we refer to as the main mass. As the simulation progresses, this region breaks apart and gaps occur between pieces of material, filled either by another material or by the surrounding air. Once there is a gap as wide as at least one cell, we determine that a fragment as formed. The challenge when finding these fragments on a large scale parallel system is that regions that make up the fragments straddle process boundaries, requiring communication between the processes to determine the full shape of a fragment.

Because the number of fragments a shock physics simulation can generate are so numerous, it is seldom realistic for a person to examine every one. It is therefore more beneficial

to first perform computational analytics that provide useful summary statistics and identify particularly interesting fragments. This analysis has the added benefit of reducing the amount of memory required to represent it. Therefore, fragment analysis is a good candidate for *in situ* processing.

A full fragment analysis requires multiple steps.

1. Find block neighbors. This includes determining block neighbors located on different process.
2. Build a conforming mesh over the AMR boundaries. AMR has inconsistent interpolation at interfaces between blocks of different refinements. The conforming mesh resolves the interpolation.
3. Identify the boundaries of fragments. We estimate the boundary as the contour at a threshold between high and low volume fraction.
4. Find the fragment connected components. A connected components analysis brings together all finite elements that belong to a single fragment.
5. Characterize properties of fragments. Given the collected elements for each fragment, find the features such as shape volume, and movement that are of interest.
6. Extract useful information. This could involve, for example, computing histograms of features or extracting a small subset of fragments deemed important.

For the purpose of simplifying the problem and making it tractable for analysis, we abbreviate the problem to include only the identification of fragment boundaries for the results of this milestone. The creation of fragment boundaries is a nontrivial problem in that we must make sure the surface is "watertight" in that the representative mesh surface is conforming and closed. Making a surface from an AMR mesh watertight is challenging because the AMR mesh itself is nonconforming at boundaries between adjacent regions at different levels of refinement.

To generate this watertight fragment surface, we first build a dual mesh of the original AMR mesh. The dual mesh contains a vertex at the center of each cell in the original mesh and an edge through each face of of the original mesh as demonstrated in Figure 9. The advantage of creating a dual mesh is that it is straightforward to build conforming cells across the boundaries of AMR regions with different levels of refinement.

The disadvantage of building these dual meshes in a distributed parallel job is that neighborhood information must be shared between regions that might be located on different processes. Resolving this neighborhood information requires a significant amount of communication. (Such communication would be necessary for any creation of a watertight mesh.) This communication can limit the scalability of the algorithm.

Efficient communication of boundary elements first requires that each process knows the location of the neighbors for each region it holds. If data is loaded with no knowledge of its

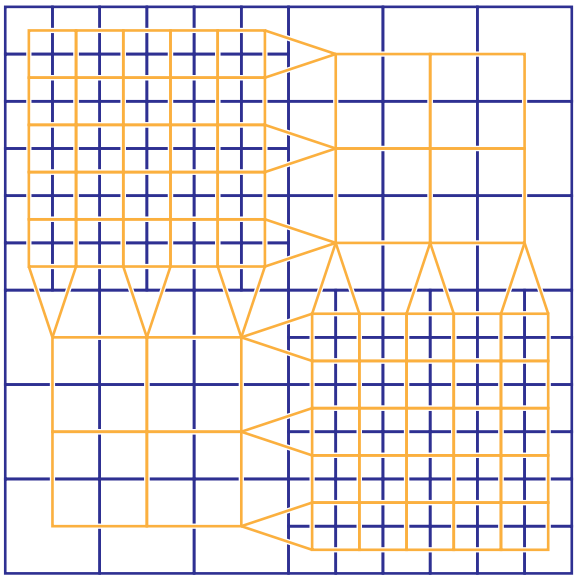


Figure 9: A simple 2D AMR example with 2 different refinement levels (blue lines) and the conforming dual grid we build with it (yellow lines).

decomposition, which is typical in the post-processing of data, then this neighborhood information can be retrieved only through global communication. Our initial **baseline** algorithm starts with this global communication, which we find to severely limit the scalability of the algorithm.

When running the surface creation algorithm as an embedded *in situ* component of CTH, this global communication of finding neighbors is wasteful because CTH already has this information. To take advantage of this neighborhood information, we make a small change to CTH to pass this data decomposition information through its I/O layer to Catalyst. With this data, our **refined** algorithm skips the global communication leaving only the more scalable boundary-data passing. Our analysis shows that the refined algorithm is much more scalable than the baseline algorithm [11]. Unfortunately, we cannot apply the refined algorithm in the *in transit* workflow because this workflow redistributes the data and invalidates this neighborhood information from CTH.

5 Results

This section documents the results of our many experiments designed to characterize the performance of our data analysis workflows. In particular, we are interested in determining the additional overhead our data analysis places on the simulation and the efficiency with which it can be done. These data summarize evaluations run over the course of 10.58 million cpu-hours of execution. The results come from measurements taken from instrumented code as well as the HPCToolkit profiling tool [3].

5.1 Experimental Setup

All experiments were performed on the Cielo supercomputer housed at Los Alamos National Laboratory. Cielo is an 8,944-node Cray XE6 resource for the Advanced Simulation and Computing (ASC) program and is jointly managed by Sandia National Laboratories and Los Alamos National Laboratory under the New Mexico Alliance for Computing at Extreme Scale (ACES) project. Each node contains two AMD Opteron 6136 (Magny-Cours) 8-way processor chips for a total of 16 cores per node. Each core has a peak computational speed of 2.4 GHz, leading to a total theoretical peak of 1.37 Petaflops for the machine. The compute nodes each have 32 GB of memory. The interconnect consists of a proprietary Cray Gemini Network with a 3D Torus topology and has a peak throughput rate of 6 GB/s/link.

This report includes strong scaling and weak scaling results from the following CTH and data analysis experiments. These represent three different workflows, two of which have two different configurations for a total of five experiments.

***In situ*:** A CTH job that directly runs an *in situ* data analysis. Thus, the visualization and data analysis is performed in the same job and memory space as the simulation. Within the *in situ* workflow, we measure two variations of our watertight surface algorithm.

Baseline: As described in Section 4, the baseline version of the algorithm includes a redundant step of global communication to find AMR block neighbors. We include results from this workflow for two reasons: First, we are not able to apply the same optimization to the *in transit* and offline post-processing workflows, so this provides an apples-to-apples comparison of the benefit an *in transit* approach could give using the same algorithm. Second, we anticipate other important analysis algorithms could have similar communication characteristics. For example, a connected components algorithm could require communication between most or all processes to resolve the connectivity of large fragments.

Refined: As described in Section 4, the refined version of the algorithm bypasses the step of global communication by retrieving the AMR block neighbors from the running CTH simulation. We thus expect it to have better scaling performance.

***In transit*:** A CTH job that performs *in transit* data analysis using a separate allocation of compute nodes. Data is transferred from the simulation to a service where visualization and data analysis is performed asynchronously with respect to the simulation. Within the *in transit* workflow, we measure two job scheduling variations.

Extra nodes: Allocate the CTH job with the same number of nodes as we would without the data analysis, and then create a visualization and data analysis service using extra nodes. For example, if the simulation were normally to be run on 256 nodes, then still schedule the simulation on 256 nodes and also schedule the visualization and data analysis service on an additional 16 nodes. This allocation represents a use case where there are additional compute nodes (perhaps with special OS, runtime, or hardware features) that could be used to perform data analysis on behalf of the application. For example, the suggested "burst buffer" architecture for the Trinity system may have special nodes with NVRAM and additional memory that would be appropriate for this type of *in transit* data analysis.

Internal nodes: Divide the nodes normally allocated to the CTH job between the simulation and the visualization and data analysis service. For example, if the simulation were normally to be run on 256 nodes, then schedule the simulation on only 240 nodes and use the remaining 16 nodes to schedule the visualization and data analysis service. We use this workflow to find out if, given an equal number of resources for *in situ* and *in transit*, there might be situations where the preferable approach changes.

Disk-based post-processing: A CTH job that writes Spyplot files instead of doing data analysis. At some point later a batch data analysis job is run on the saved data. This workflow represents the traditional post-processing approach. The number of nodes used matches the number used for data analysis in the Extra nodes version of *in transit*. Note that although CTH is writing out Spyplot files, these files are being read by the ParaView application and running the same algorithm with the same code as the *in transit* workflow and the baseline algorithm for the *in situ* workflow.

All applications complete 500 cycles (i.e., timestep calculations) of the CTH code. The first four applications execute a fragment analysis operation once every 10 cycles. Spyplot is an *in situ* visualization capability written as part of CTH to provide some basic visualization capability such as isosurfaces and cut planes. Because fragment detection is not available as part of Spyplot, we do not consider the *in situ* capability here. Instead, for the Spyplot file application, we output Spyplot data which intended for fragment post-processing by ParaView, and written at a fixed interval in simulated time, calculated so that the application executed 51 I/O operations, equaling the number of fragment analysis operations performed by the *in situ* and *in transit* applications. The number of nodes used is the same as the *in transit* application using extra nodes. There is no way to directly instruct CTH to output Spyplot data every 10 cycles. The resulting data files are then loaded in a separate ParaView data analysis job run at a later time. This data analysis was also performed on Cielo using

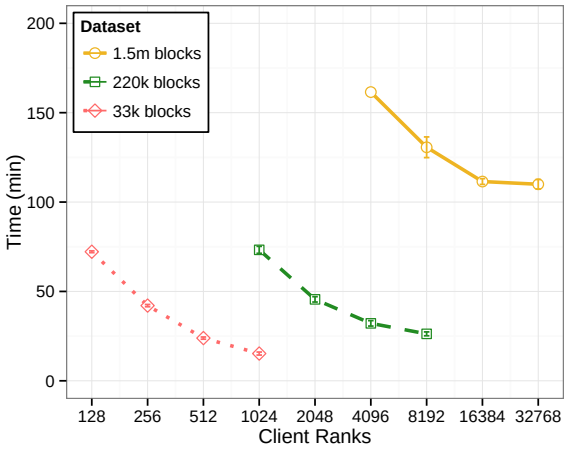
ParaView statically compiled from the same code base as the *in situ* and *in transit* runs, but executed by ParaView's `pvbatch` application. The time to run the simulation, read and write files, and perform the post-processing analysis are all summed together to get a processing time for computation equivalent to that done in the *in situ* and *in transit* workflows.

Table 1: Scaling Overview

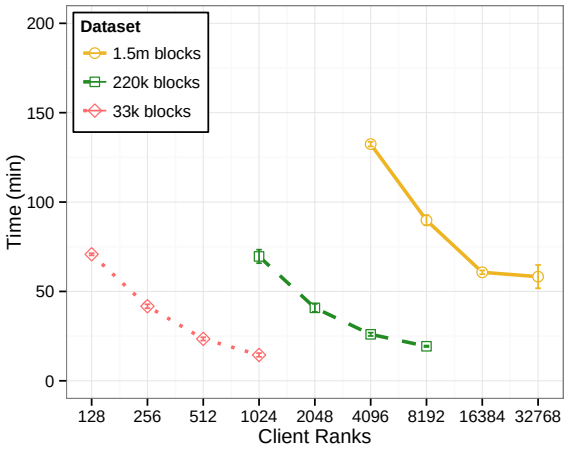
CTH				*In transit* Server			
Most		*In transit*	Internal	Extra Nodes		Internal Nodes	
Cores	Nodes	Cores	Nodes	Cores	Nodes	Cores	Nodes
33K Blocks — 5 levels							
128	8	96	6	16	2	16	2
256	16	224	14	16	2	16	2
512	32	480	30	16	2	16	2
1,024	64	992	62	16	2	16	2
220K Blocks — 6 levels							
1,024	64	768	48	128	16	128	16
2,048	128	1,792	112	128	16	128	16
4,096	256	3,840	240	128	16	128	16
8,192	512	7,936	496	128	16	128	16
1.5M Blocks — 7 levels							
4,096	256	2,496	156	1,024	128	800	100
8,192	512	6,592	412	1,024	128	800	100
16,384	1,024	14,784	924	1,024	128	800	100
32,768	2,048	31,168	1,948	1,024	128	800	100
65,536	4,096	63,936	3,996	1,024	128	800	100

For each application, we ran strong scaling experiments for three different datasets. Each data set comes from the same initial conditions but with a different maximum level of refinement. Thus, measurements of different job sizes with different data set sizes provides a weak scaling overview. Table 1 shows the range of core sizes used for the various experiments. For every application we used the maximum 16 cores-per-node for the CTH client, since CTH is primarily bound by computation and scales very well. For the *in transit* experiments in this section, we used 8 cores for each service node. The exception is in Section 5.5 where we discuss *in transit* performance under different core-per-node counts.

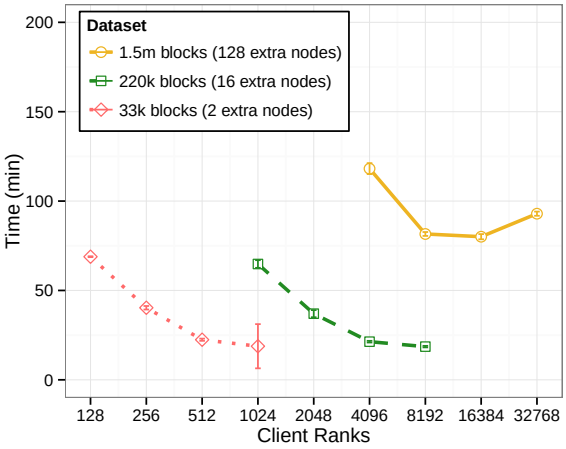
We note that although the *in transit* experiments use 128 visualization nodes when allocated using extra nodes but only 100 nodes when run internally. The rational is that we ran the experiments with extra nodes first, and determined that the visualization is a memory-bound problem with more memory than necessary on the nodes. Thus, we reduced the visualization partition to 100 nodes when using internal nodes, but we were unable to repeat the former experiments to remove this minor discrepancy.



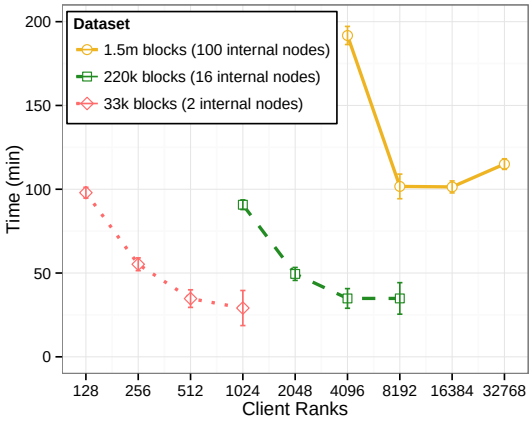
(a) *In situ* baseline.



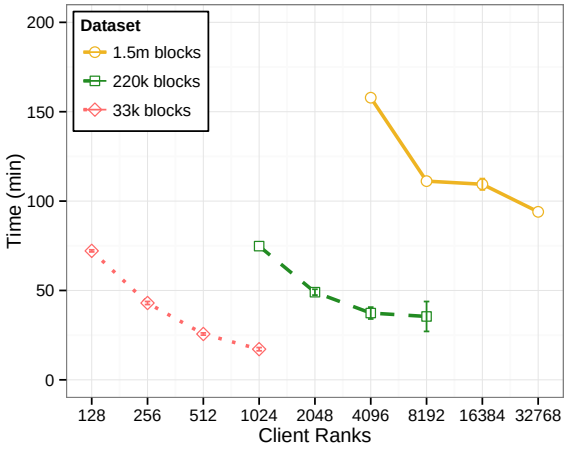
(b) *In situ* refined.



(c) *In transit* with extra nodes.



(d) *In transit* using internal nodes.



(e) Disk-based post-processing.

Figure 10: Total runtime for 500-cycle runs of each workflow.

5.2 Total Execution Time

Our first consideration is the overall runtime of each workflow with each data set and job size.

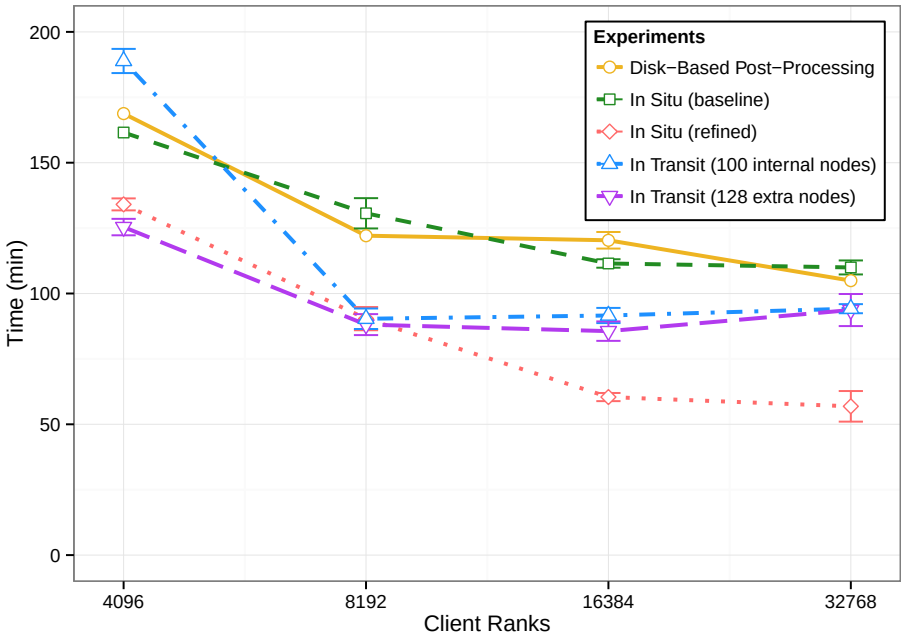
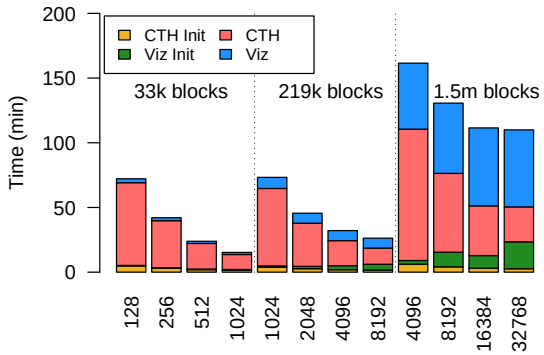


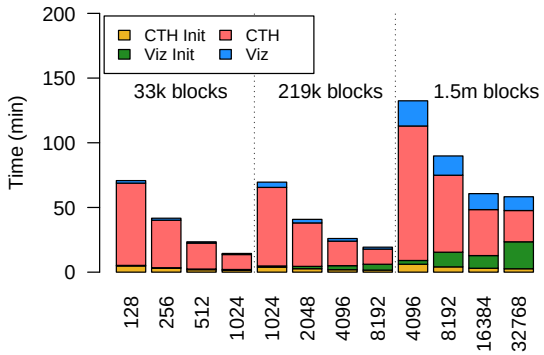
Figure 11: Execution time comparison for the 1.5M block dataset.

Figures 10 and 11 show the measured total execution time from each of the different workflows. In cases where we ran the same experiment multiple times, we plot the mean with error bars representing the standard deviation from the mean. The set of plots in Figure 10 show individual timings of each of the workflows for each size dataset; the plot in Figure 11 shows a direct comparison of all the applications for the 1.5M block data set. The results clearly show a "sweet spot" at 8K cores where the *in transit* approach, even though it is using a less scalable algorithm, performs the same as the the refined version of *in situ*. At 16K and 32K, none of the codes running data analysis show significant improvement, the baseline *in situ* and the *in transit* approaches actually take longer. We believe the biggest reason for this is that there is not enough work for the compute nodes. At 32K cores, each core is processing around 46 blocks/node, where the same size problem using 4K nodes requires each core to process 366 blocks. The key to making the *in transit* approach successful is being able to overlap computation and data analysis. If the data analysis portion does not scale particularly well, the compute nodes need sufficient work to hide the analysis cost.

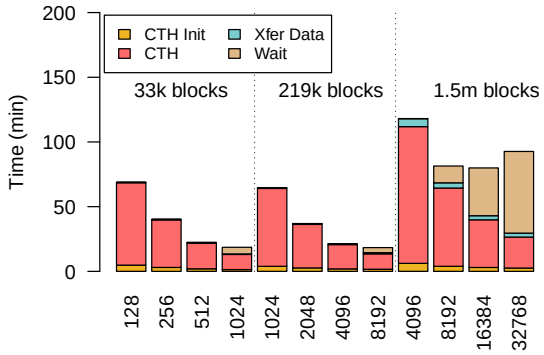
To better understand exactly where the time is being spent, we collect detailed timings of each application using a combination of instrumented timers and profiling tools (HPC-Toolkit). Figure 12 shows the total runtime performance of the five workflows as a stacked bar plots illustrating the portion of runtime associated with select functions. For the *in*



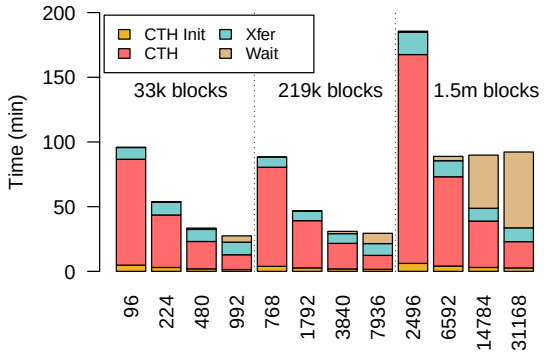
(a) *In situ* baseline.



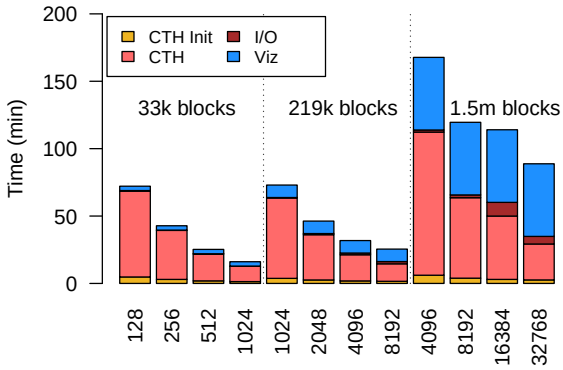
(b) *In situ* refined.



(c) *In transit* with extra nodes.



(d) *In transit* using internal nodes.



(e) Disk-based post-processing.

Figure 12: Illustration showing the contributions of selected operations to the total execution times for each application.

situ workflows, we measure the initialization and computational time of CTH and the analysis/visualization. For *in transit* workflows, we measure the initialization cost of CTH, the cost of transferring data to the service, and the time the client waits for the server operation to complete. The wait only occurs after CTH has finished its operation before the visualization and data analysis is complete. If the visualization and data analysis completes before CTH finishes its operation, then no wait time occurs.

Results from Figure 12a show that there is a clear scaling problem with the data analysis portion (labeled "Viz") of the baseline *in situ* workflow. It almost appears as if the execution time is more dependent on the problem size than the number of cores performing the data analysis. The refined version dramatically improves the performance. This corroborates our previous work [11], but we are now able to take the scaling out to a larger scale to see that the performance appears to be flattening out (although this might be in part due to a small number of blocks per core).

Another important issue these timings reveal is that of the initialization cost of the visualization and data analysis. Although the CTH initialization cost appears to decrease as the core count increases, the initialization cost for data analysis, "Viz Init," increases, accounting for more than 1/3 of the total time for a 500-cycle run. For long runs, the initialization cost will get amortized, but is still large enough to warrant further study.

The *in transit* workflow results in Figures 12c and 12d show that the *in transit* approach effectively hides the data analysis overhead when the clients have sufficient compute resources. For the 1.5M block dataset, Figure 12c shows that *in transit* with an extra 128 nodes successfully hides most of the cost of data analysis at 4K and 8K nodes, but the wait time at larger core counts eliminates any benefit of *in transit* using the baseline algorithm.

The *in transit* workflow, shown in Figure 12d, which carves out a subset of 100 nodes for data analysis, has interesting results as well. Observe that the number of cores used for CTH is much smaller, leading to an increase in the time spent doing CTH computation. Even with this increase in computational cost, there is still benefit. At 4K and 8K, all of the data analysis cost is hidden. At 8K, the total runtime is slightly less than the refined version of *in situ*. This result is a bit of a surprise given that they are both using the exact same number of resources.

An additional test of interest (not performed within the scope of this Milestone) would be to measure performance of the *in transit* workflows using the refined version of the fragment analysis algorithm. If the the *in transit* approach achieved the same performance improvement shown by Figure 12b, the *in transit* approach would be able to hide most of the fragment analysis cost even at large scale. However, using the refined version of the fragment analysis algorithm would be different because the mesh decomposition changes when transferred from client to server. For the neighborhood information to remain consistent, the I/O framework would have to report how data was repartitioned so that the neighborhood information could be mapped to the new decomposition.

Another surprising result is the performance of the spyplot file application. For the size of

datasets we studied, this application performed quite well, showing that the Lustre file system on Cielo is quite strong. The plots include the time spent writing the spyplot files during the experiment and the measured time to perform the data analysis as a post-processing step. One anomaly we notice in Figure 12e is that for the largest data set the I/O time jumps from around 2 minutes with 8192 cores to 10 minutes on 16,384 cores. Looking closer at our log files, we see that we have two experiments contributing to this value. One experiment required about 4.5 minutes to write whereas the other required about 15.5 minutes. We speculate that this second measurement comes from an anomalous condition on Cielo, but we do not have enough data to diagnose further.

5.3 Time-Series Analysis

To illustrate how operation performance changes throughout a single run of the application, we selected one experiment and tracked the performance of each 10-cycle period over a span of a 500 cycle run. Since the "Viz" operation executes every 10 cycles, what is shown is the sum of 10-cycles worth of CTH and the time spent by 1 data analysis operation. We chose to evaluate the experiments that use 8K processors of the 1.5M block dataset — the "sweet spot" mentioned in Section 5.2 — because it is one of the interesting places where the performance of the refined *in situ* application and both *in transit* applications all have similar total runtime.

Since we are running the adaptive-mesh refinement (AMR) variant of CTH, the number of "active" blocks gets recalculated by CTH at various points during a run. Since we expect this to have an impact on CTH execution and *in transit* transfer times, we first plot the number of active blocks for our selected experiment in Figure 13. As simulation progresses,

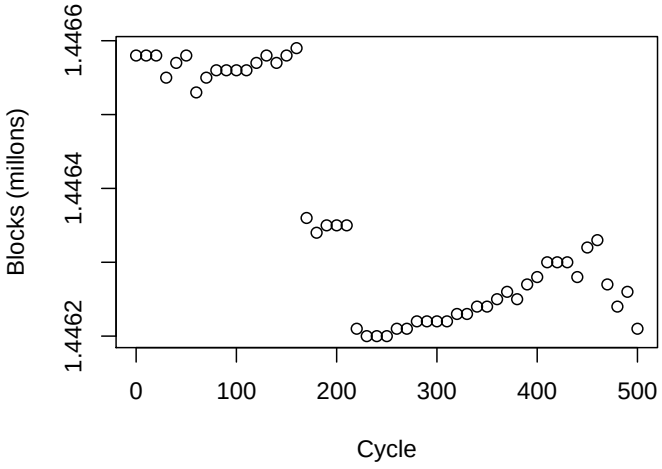
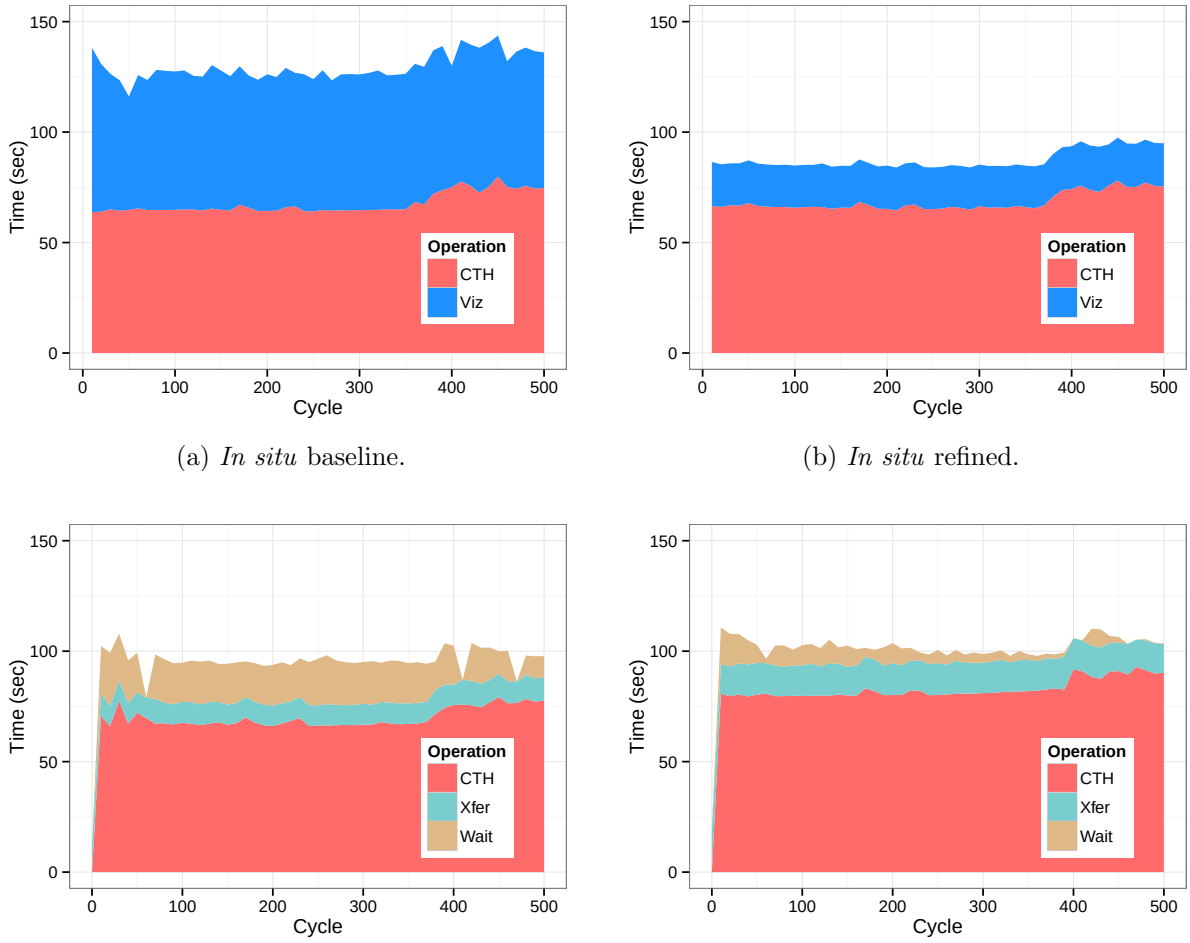


Figure 13: Number of active blocks used in the course of a 500-cycle run of CTH.



(a) *In situ* baseline.

(b) *In situ* refined.

(c) *In transit* with extra nodes.

(d) *In transit* using internal nodes.

Figure 14: Illustration showing the contributions of selected operations for each 10-cycle period of a 500-cycle CTH experiment.

it is common for the number of blocks to increase as finer resolution is needed to capture cumulative physical effects. This simulation has an interesting drop in the number of blocks midway through; however, the drop in the number of blocks is very slight (less than 0.01%), so for practical purposes the number of blocks is constant.

Figure 14 shows the time contribution for a 10-cycle period of each important operation in the *in situ* and *in transit* applications using 8K processors. One observation consistent across the different workflows is that the time spent performing CTH computation tends to increase throughout the life of the application, which is interesting considering that there not a significant increase in either the number of blocks or the visualization processing time.

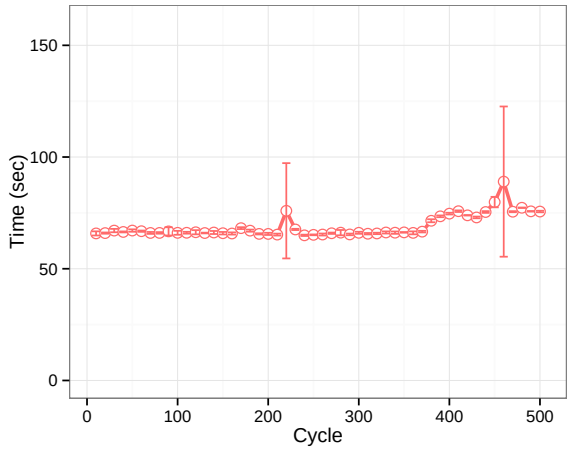
We also observe for the *in transit* applications that as long as there is some wait time, the total run time for each 10-cycle period remains relatively flat throughout the life of the application. We see this in both the *in transit* extra and *in transit* inclusive experiments. The reason for this is fairly obvious. If we assume the "Viz" operation on the service is

essentially constant, then the wait time should be the difference between the Viz time on the service and the CTH time on the client. As the CTH time increases, the wait time decreases by the same amount. Since the transfer time is close to constant (it's based on the number of blocks), the total time at each 10-cycle period remains about the same.

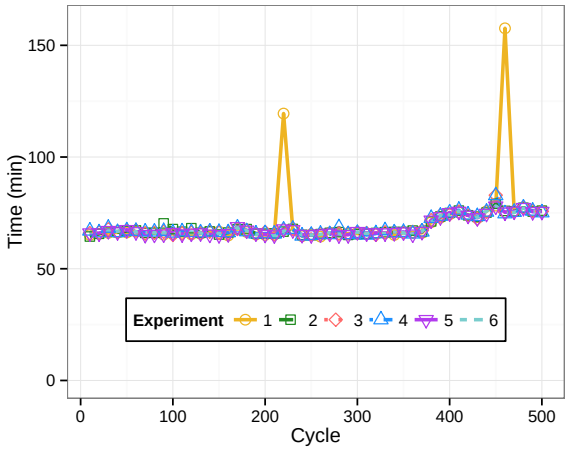
5.4 Runtime Variance

An increasingly important metric for HPC applications is runtime variance. A number of factors could contribute to inconsistent performance across multiple runs. Among them are resource contention for memory, network, or storage systems, operating system noise, and software techniques like garbage collection. Instead of trying to understand the cause of inconsistent behavior in our experiments, we to document the results with the intent of addressing them further in future work.

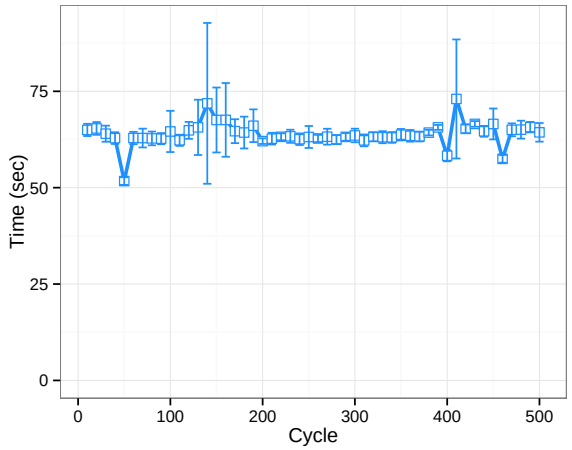
The plots in Figures 15 and 16 show the mean and standard error (the standard deviation from the mean) of the important *in situ* and *in transit* operations for the 8k-core experiments. In each plot, we gathered data from five or more 500-cycle experiments. Most of our measurements for CTH and anlysis operations show a relatively small variance between experiments. There are some outlier measurements in Figures 15a and 16b that we attribute to another event running concurrently on Cielo by happenstance causing contention. Figure 15b shows the CTH experiments overlaid to highlight the outliers in experiment 1. The transfer time of *in transit* workflows (Figures 16a and 16c) have more variance than the rest, which we believe is caused by the Cielo job scheduler not taking into account the communication between the client and server jobs. Further research in improved placement of client and server jobs for *in transit* workflows is ongoing.



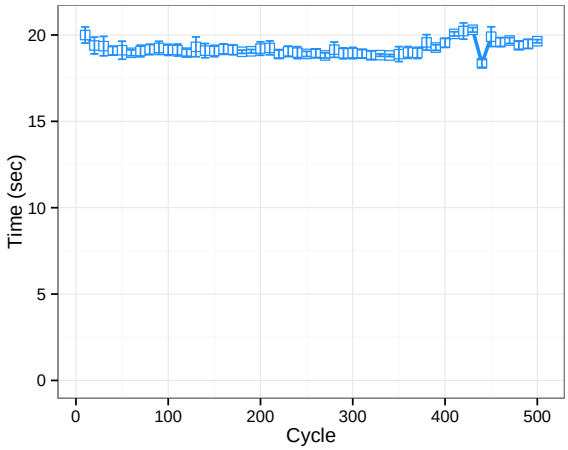
(a) CTH mean and stddev



(b) CTH overlay

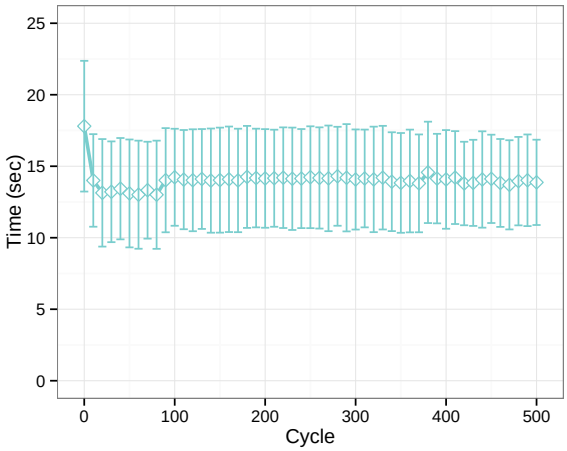


(c) *In situ* (baseline) data analysis

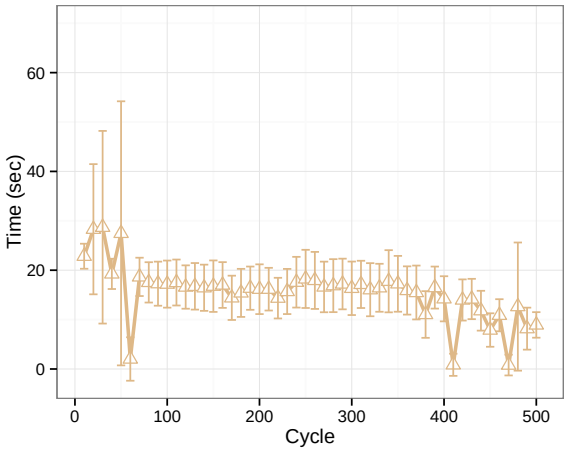


(d) *In situ* (refined) data analysis

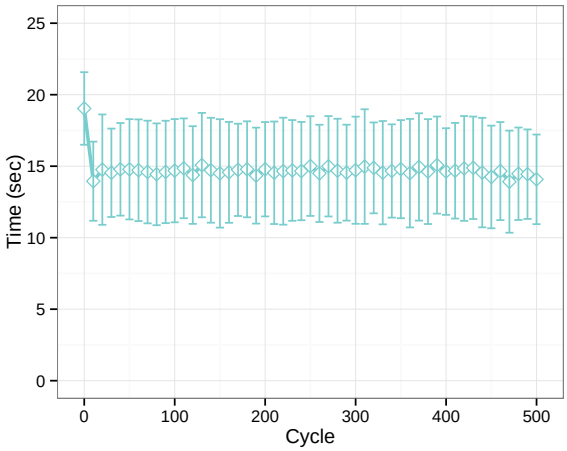
Figure 15: Plots (a), (c), and (d) show the mean time and standard error for CTH and the *in situ* data analysis operations. Plot (b) shows an overlay of the CTH experiments to show outliers for experiment 1.



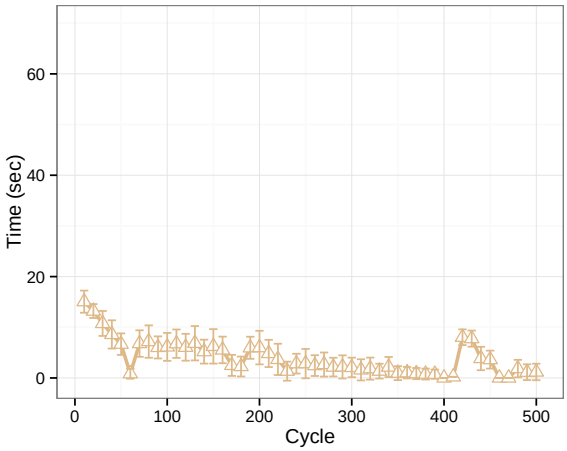
(a) *In transit* (extra nodes) transfer time.



(b) *In transit* (extra nodes) wait time.



(c) *In transit* (inclusive) transfer time.



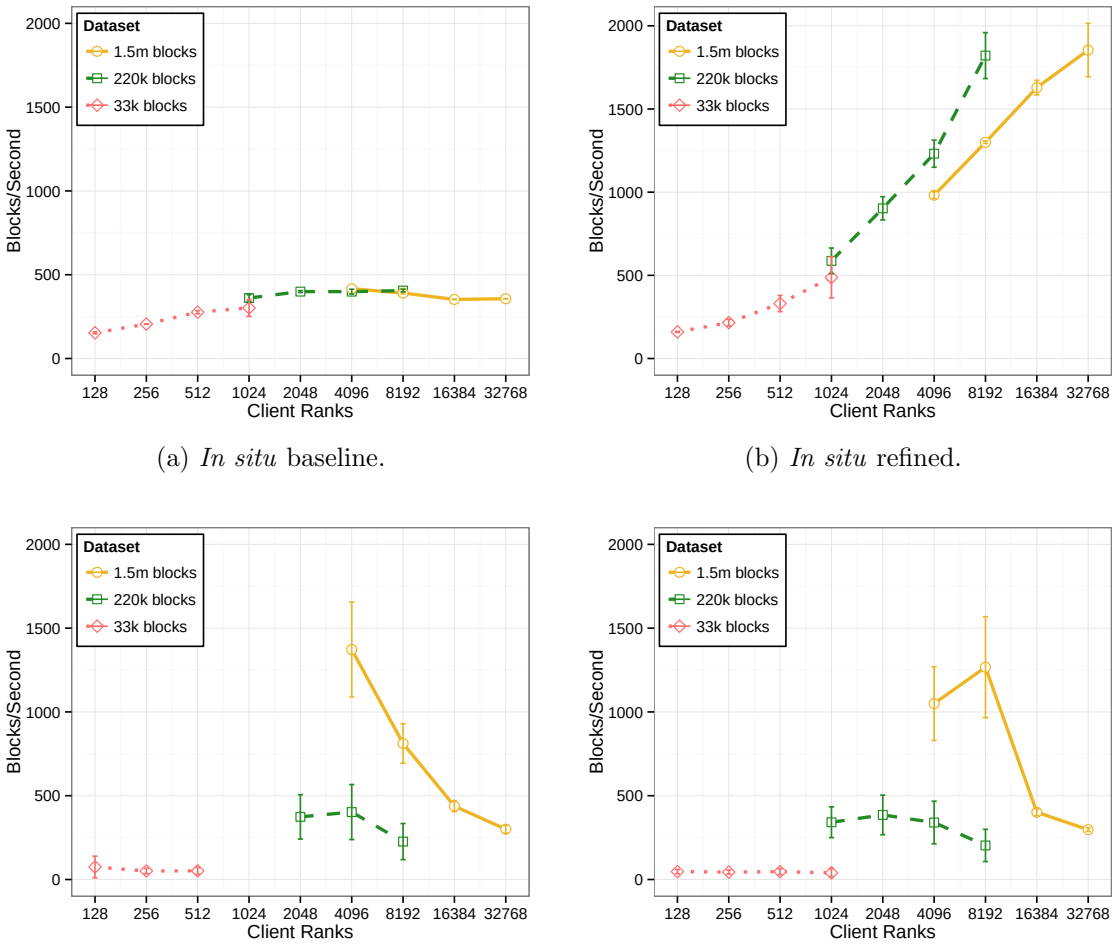
(d) *In transit* (inclusive) wait time.

Figure 16: Plots of the mean time and standard error for the *in transit* transfer and wait operations.

5.5 Scaling Analysis

Block Processing Rate

One way to get a good understanding of the scalability of an operation is to look at the throughput or processing rate. In this case, we look at the rate at which the data analysis operation processes blocks as we increase the number of client cores.



(a) *In situ* baseline.

(b) *In situ* refined.

(c) *In transit* with extra nodes.

(d) *In transit* with internal nodes.

Figure 17: Processing rate of data analysis portions of the four different applications.

Figure 17 shows the processing rates of the data analysis portion of the *in transit* applications and the "effective" processing rate of the data analysis portion (transfer time plus wait time) of the *in transit* applications. Note that the effective *in transit* rate does not include any processing time overlapped with the simulation execution, and thus could be much larger than the actual processing rate. As we expect, the baseline application scales fine for the small data set, but starts to really drop off for the medium and large data. We see a dramatic improvement in the *in situ* refined application as it scales consistently well all the way to 32K cores. The *in transit* applications are also not that surprising. Since we use a

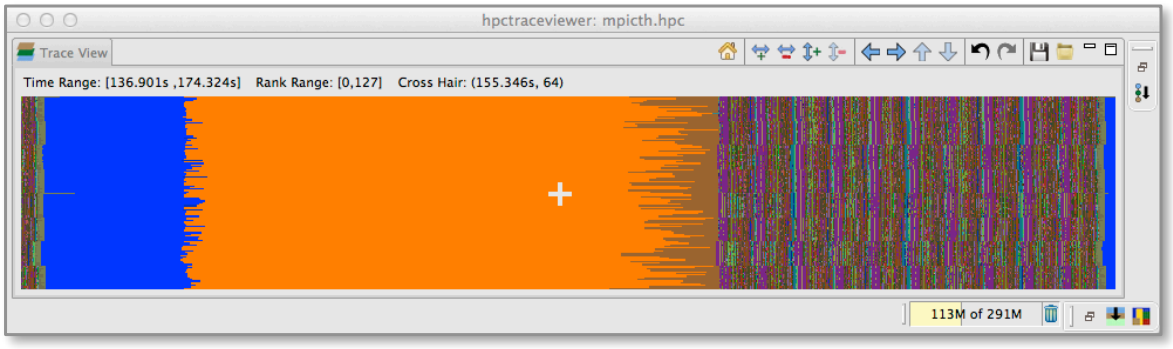
fixed number of nodes for a dataset, we expect the effective processing rate to be essentially flat. The dropoff for the medium and large datasets is due to the excessive waiting, identified in Figures 12c and 12d.

Node scaling for *in transit* experiments

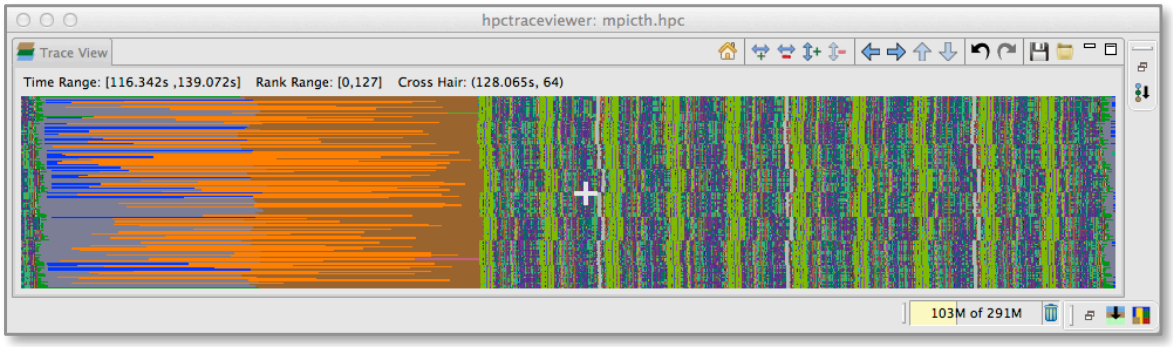
In this section, we evaluate performance of the *in transit* applications when using different numbers of cores/node for the *in transit* service. To better understand the impact of changing the core count for the services, consider the three HPCToolkit-generated performance traces in Figure 18. The traces show a 10-cycle window of execution for a 128-core job using 1 server node. For this small experiment, we used a dataset of only 5k blocks.

The tracing results show a dramatic difference in network performance and wait time between all three of the experiments. We believe the relatively poor network performance in the 2-core experiment is caused by contention. Because only 2 cores can process the bulk-data requests at a time, the clients are either waiting for network transfers to complete, or the are waiting for the server to finish copying the data to a server buffer, causing the request to sit in the server's pending queue. The larger wait time on the 2-core experiment tells us that for this size problem, there is a computational benefit to increasing the number of cores performing the data analysis.

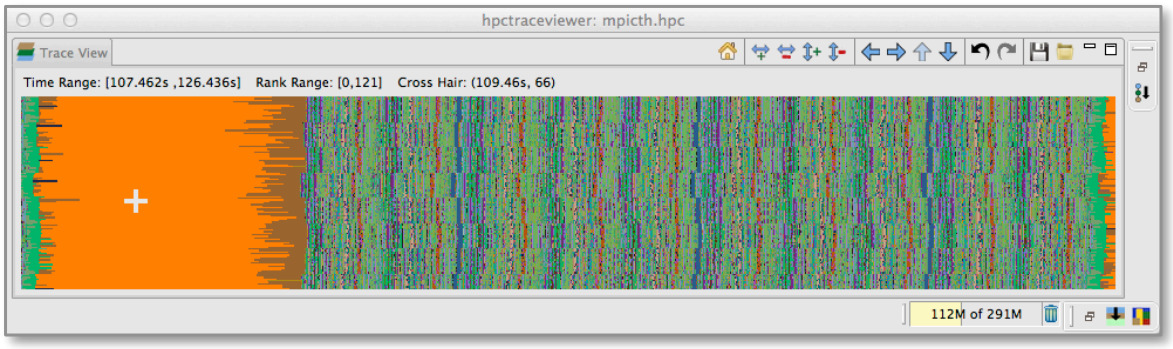
The plots in Figure 19 show differences in the total, transfer, and wait times for the 2, 4, and 8 cores/node runs of the *in transit* (extra) application for the three different data sets. While the smaller datasets show clear benefit of using more cores/node for data analysis, the inverse seems to be true for the large datasets. Unfortunately, we did not run enough experiments of this type to make definitive claims. A further investigation of the impact of core scaling for the large dataset is required.



(a) *In transit* with 2 cores/server.

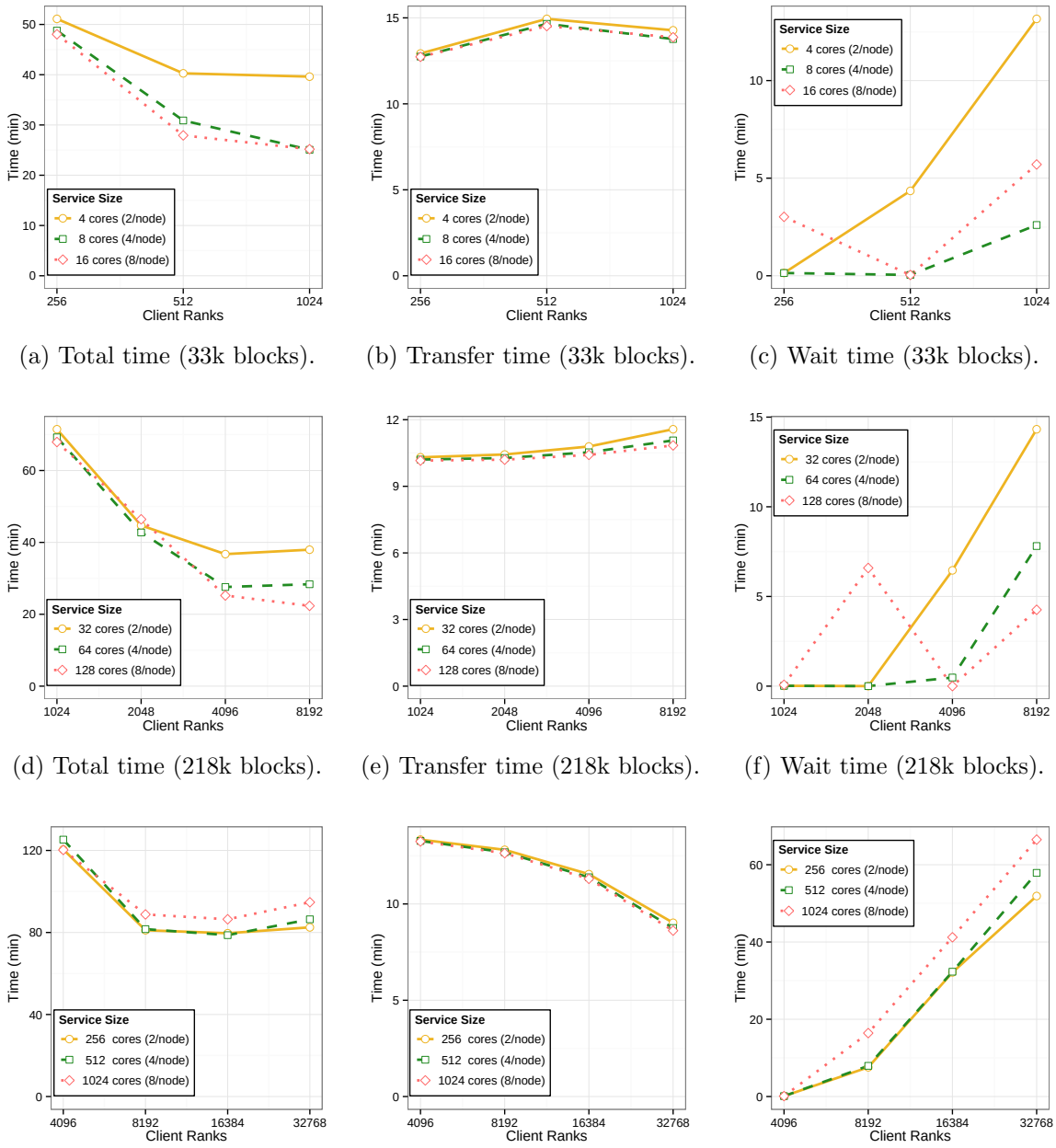


(b) *In transit* with 4 cores/server.



(c) *In transit* with 8 cores/server.

Figure 18: HPCToolkit-generated traces showing a 10-cycle window of execution for a 128-core job using 2, 4, and 8 cores for the server nodes.



(a) Total time (33k blocks). (b) Transfer time (33k blocks). (c) Wait time (33k blocks).

(d) Total time (218k blocks). (e) Transfer time (218k blocks). (f) Wait time (218k blocks).

(g) Total time (1.5m blocks). (h) Transfer time (1.5m blocks). (i) Wait time (1.5m blocks).

Figure 19: *In transit* performance for 2, 4, and 8 cores per server node.

5.6 Memory

The following memory plots were taken by examining free memory on the nodes using /proc/meminfo. Measuring free memory is conservative because it also accounts for caches and buffers, so although you may run out of free memory in a system, the execution will not immediately fail due to memory freed from those other sources. However, these are an indication of the worst case possible. On Cielo there are 16 cores per node and in each case the nodes were loaded with 16 MPI ranks executing the statically linked executable.

In order to understand CTH memory usage results, it is important to note that CTH preallocates memory based on a value for "max number of blocks" provided through the input deck. Because of this, CTH memory usage is highly impacted by the user specification. In this case we ran the results with what we believe are reasonable values for "max number of blocks" given the size of the problem. Figure 20 shows the corresponding max blocks for each run.

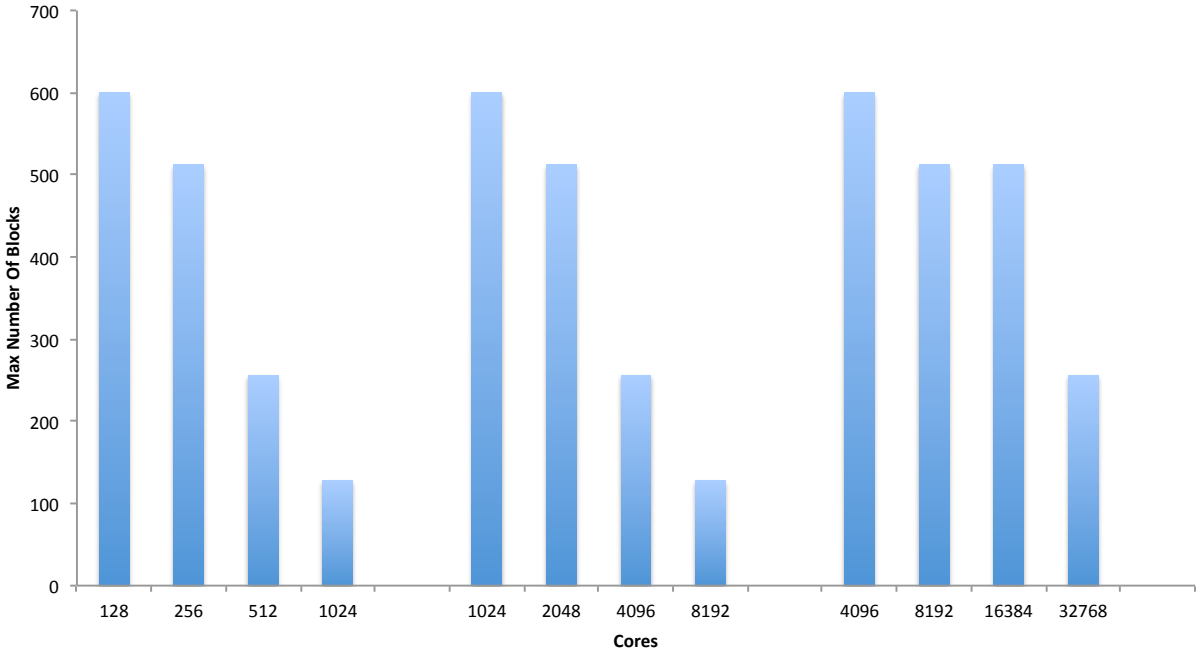


Figure 20: A plot of the "max number of blocks" parameter supplied to CTH for each run.

Figure 21 provides an overview of all memory measurements taken for the system. The memory taken by CTH is quite flat, as expected, for each run. However, even though the size of the mesh changes throughout the simulation, the memory overhead for *in situ* and *in transit* runs changes only moderately. Thus, for the rest of the results analysis, we summarize all measurements as simply the maximum value, which is a reasonable representation of all values.

Figure 22 gives a summary of the extra memory used when using Catalyst for *in situ* data analysis during our experiments. Likewise, Figure 23 gives the same summary for the *in*

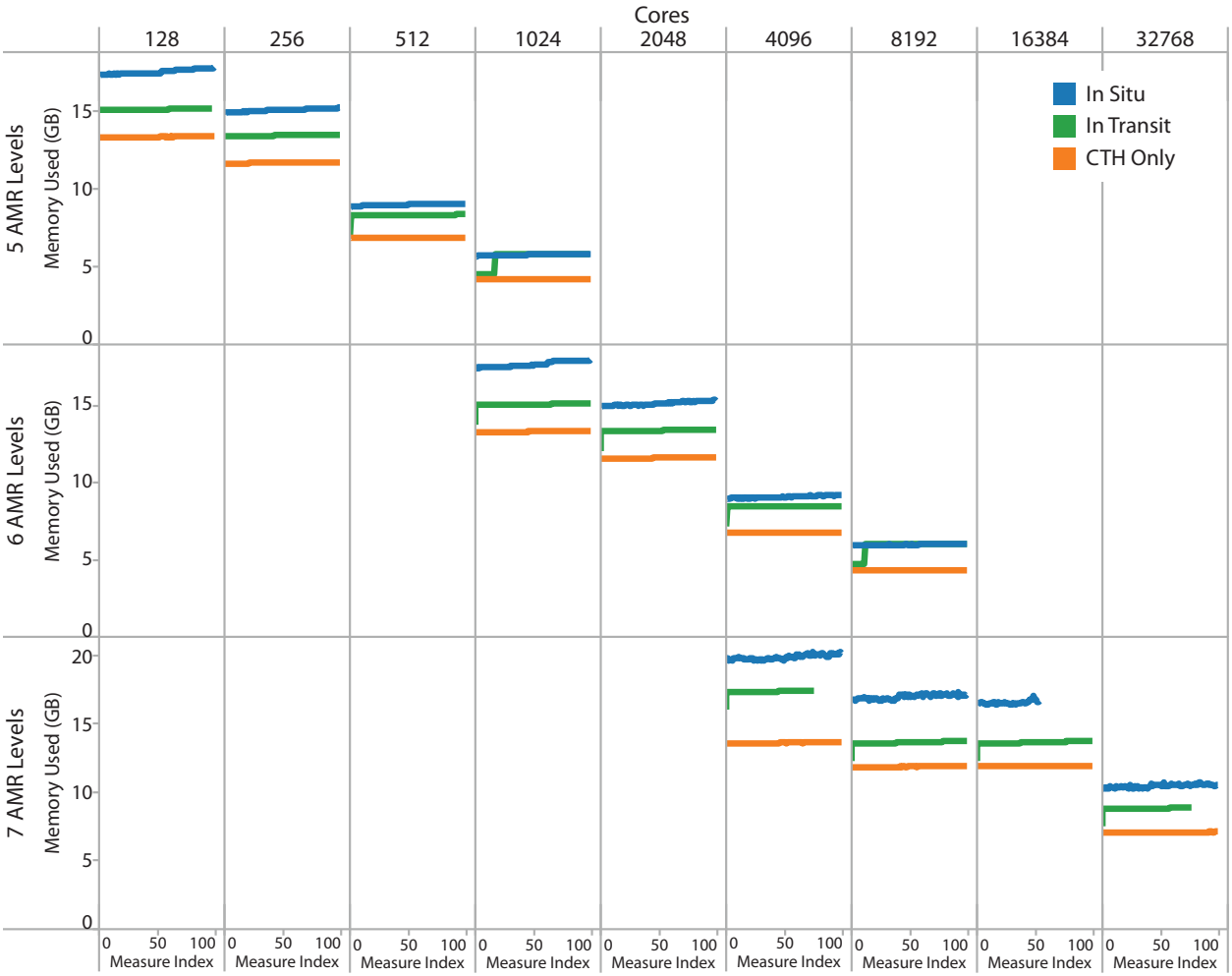


Figure 21: Matrix of all measurements taken for memory usage comparisons. Each measurement is the total memory in use in a node (so for *in situ* and *in transit* memory includes both CTH simulation and overhead). Each measurement is plotted as the maximum memory use in all nodes at the time the measurement was taken.

transit memory overhead for the nodes within the simulation (memory usage on the separate data analysis job is not given). In all cases the added overhead is less than 50% than the memory used by CTH itself, and in most cases the additional overhead is significantly smaller than that.

Figure 24 compares the amount of memory per node added when using *in situ* versus *in transit*. As expected, the *in transit* approach requires a smaller memory overhead than *in situ* within the nodes of the simulation. Thus, *in transit* could be a better option when the simulation requires as much memory per process as possible, but the *in transit* approach also requires separate nodes to be reserved for the data analysis, which also may cause the total amount of available simulation memory to be reduced if nodes must be taken away from the simulation.

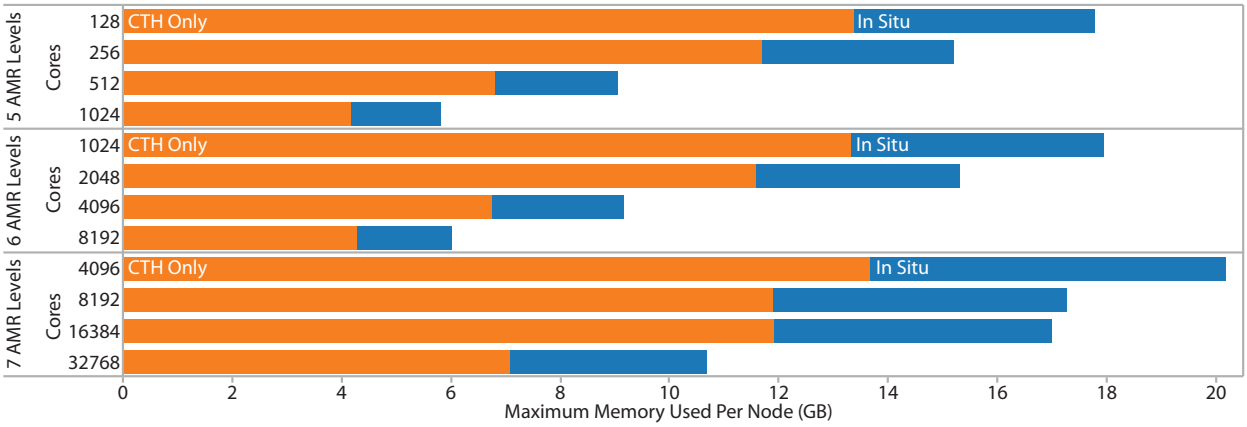


Figure 22: Plot of average per node memory usage of the *in situ* run on Cielo.

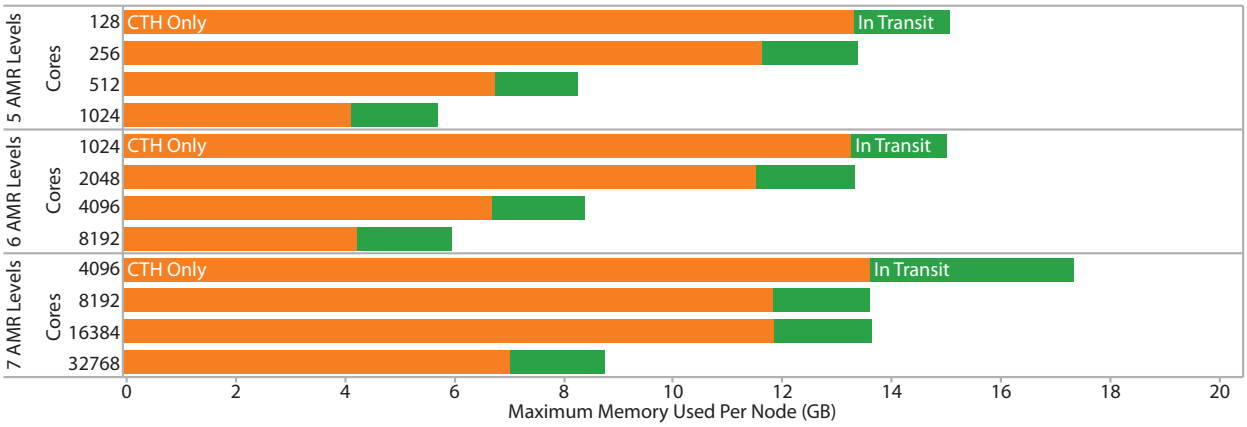


Figure 23: Plot of average per node memory usage of the *in transit* run on Cielo.

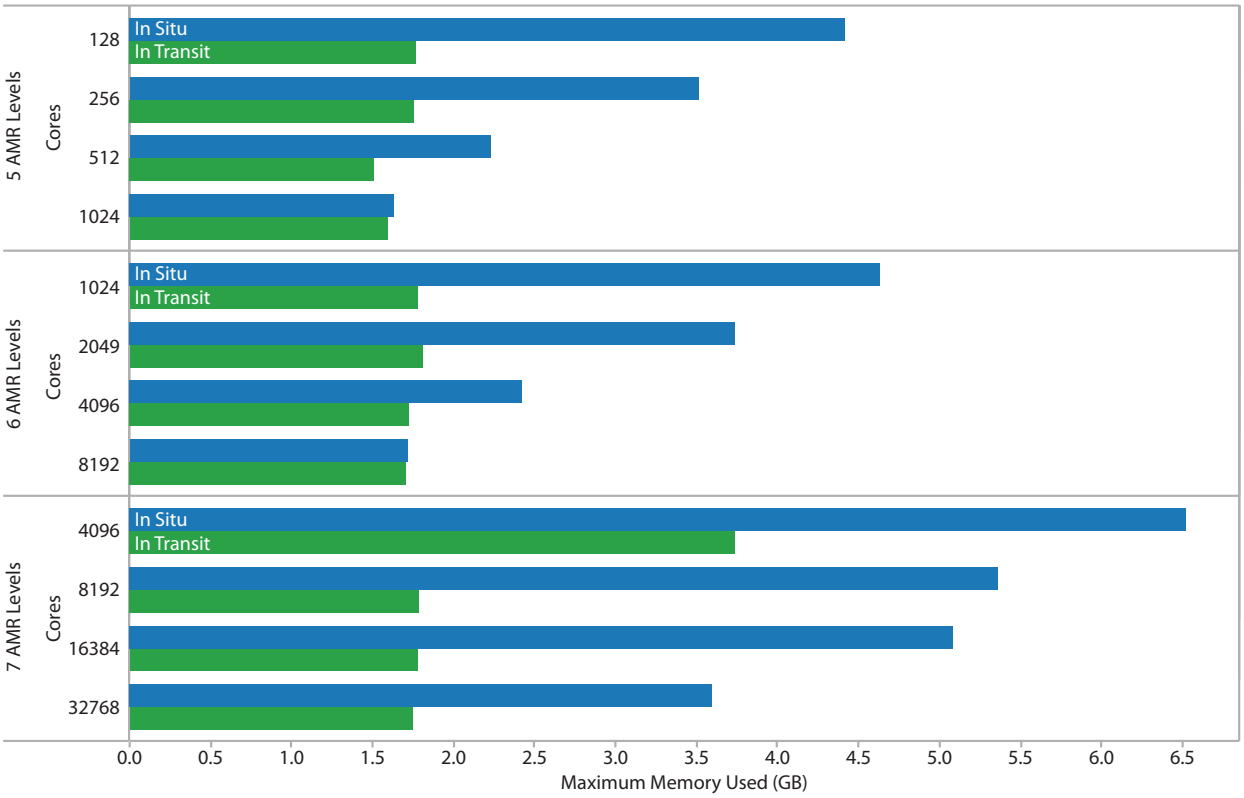


Figure 24: Plot of the average overhead per node of both *in situ* and *in transit*

6 Conclusions

This document summarizes a significant scaling study resulting from over 9 million core-hours of execution and analyzes the comparative performance of multiple workflows for performing visualization and data analysis on simulation results. Most of these workflows benefit from running in tandem with the simulation to analyze its transient data before it is written to storage. Based on this analysis, we make the following conclusions.

***In transit* can provide a performance improvement over *in situ* in some circumstances, but the window is narrower than we anticipated.** *In transit* data analysis has an added overhead above embedded *in situ* data analysis involving transferring data between parallel jobs. Given a data analysis algorithm with perfect linear scalability, we suspect *in transit* workflows will always have an added cost, and our results support this. With a data analysis algorithm that does not scale perfectly, possibly due to communication overhead, it is theoretically possible for *in transit* to be faster by reducing the size of the data analysis job. This is one of the motivations for choosing a data analysis task that requires significant communication. In our results, we do find instances where *in transit* is faster, but by a smaller margin and for fewer configurations than we initially anticipated. So although *in transit* has several other positive features, we do not anticipate performance to be the main motivations for using it.

The efficiency of *in transit* relies on balancing the time spent in simulation and data analysis. The significant overhead cost, apart from data transfer, in the *in transit* workflow is the idle time spent in the simulation waiting for the visualization and data analysis service to become ready or the idle time spent in the visualization and data analysis service waiting for the simulation to send more data. This idle waiting time is minimized when the simulation and data analysis spend the same amount of wall clock time between transfers. Although not demonstrated in this work, it is possible to "auto-balance" the work between simulation and data analysis by, at every iteration of the simulation, transfer data to the data analysis if and only if the data analysis service is ready to accept more work. The disadvantage of such an approach is that the idle process time could be replaced with unnecessary extra data analysis or less data analysis than necessary. However, we suspect that controlling the amount of visualization and data analysis performed through job allocation sizes fits well with users' rules of thumb about resource allocation.

Memory overhead will be an important trade-off space. The baseline amount of memory added to the CTH job to perform *in situ* processing is roughly 100MB per core. Considering that our embedded *in situ* library is a fully featured visualization toolkit containing over 2 million lines of code and algorithms developed over almost 2 decades, this overhead is not unreasonable. Nevertheless, this footprint can be problematic for simulations already tight on memory. Because of this, efforts are already underway to improve our memory footprint by making finer modules and being more selective on the available

algorithms. This, of course, requires a compromise between the size of the library and the algorithms that are dynamically available. We also note that our algorithm has the potential to generate sizable meshes of its own. Thus, it may be fruitful to pursue and support incremental algorithms where possible.

Initialization time matters. Our scaling efforts to date focus on the scalability of the algorithms invoked during the run of a simulation. The initialization cost, a one-time penalty, has yet to be seriously considered. However, based on our HPCToolkit measurements, initialization becomes a significant cost at high process counts.

Disk-based I/O is not dead... yet. Our initial assumption was that it would not be feasible to output full results at a fine enough temporal resolution from CTH to disk storage to perform our high fidelity data analysis. However, our control workflow shows that although the overall time to write data to disk and then read back again incurs a large cost, it is still realistic to do so. Thus, users may still choose to incur the extra overhead to use a traditional offline post-processing visualization and data analysis workflow. This is an important consideration in providing flexibitly for our end users.

Better job scheduling is important. One of the more complicated parts of running an *in transit* workflow is scheduling the simulation job and service job to run in tandem. Frankly, the capabilities of the scheduler are inadequate for our needs. We cannot start and stop jobs independently and make reconnections dynamically. Another experiment we would like to do but is challenging to schedule is to allow simulation and service to share nodes. Since each node has 16 cores, perhaps we could get better transfer performance by allocating one core per node for service and the rest for simulation. A similar scheduling scheme will be important to take advantage of burst buffers in future architectures.

References

[1] Hasan Abbasi, Jay Lofstead, Fang Zheng, Karsten Schwan, Matthew Wolf, and Scott Klasky. Extending I/O through high performance data services, September 2009. DOI 10.1109/CLUSTR.2009.5289167.

[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 IEEE International Symposium on High Performance Distributed Computing*, pages 39–48, Garching, Germany, 2009. ACM Press.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[4] R. Alverson, D. Roweth, and L. Kaplan. The Gemini system interconnect. In *Proceedings of the 18th Annual Symposium on High Performance Interconnects (HOTI)*, pages 83–87, Mountain View, CA, August 2010. IEEE Computer Society Press.

[5] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2, October 2004.

[6] Utkarsh Ayachit et al. *The ParaView Guide: A Parallel Visualization Application*. Kitware Inc., 4th edition, 2012. ISBN 978-1-930934-24-5.

[7] Ron Brightwell, Kevin Pedretti, Keith Underwood, and Trammell Hudson. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[8] Ron Brightwell, Rolf Riesen, Bill Lawry, and Arther B. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *Proceedings of the International Parallel and Distributed Processing Symposium*, page 268, Fort Lauderdale, FL, April 2002. IEEE Computer Society Press.

[9] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: an interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th IEEE International Symposium on High Performance Distributed Computing*, pages 25–36, Chicago, IL, June 2010.

[10] Doug Doerfler, Manuel Vigil, Sudip Dosanjh, and John Morrison. The cielo petascale capability supercomputer. In *Cray User Group*, 2011.

[11] Nathan Fabian. *In situ* fragment detection at scale. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 105–108, October 2012. DOI 10.1109/LDAV.2012.6378983.

[12] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C. Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E. Jansen. The ParaView coprocessing

library: A scalable, general purpose in situ visualization library. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 89–96, October 2011. DOI 10.1109/LDAV.2011.6092322.

[13] Lisa Ice, Nathan Fabian, Kenneth D. Moreland, Janine C. Bennett, David C. Thompson, David B. Karelitz, and W. Alan Scott. Scalable analysis tools for sensitivity analysis and UQ (3160) results. Technical Report 2009-6032, Sandia National Laboratories, September 2009.

[14] E. S. Hertel Jr., R. L. Bell, M. G. Elrick, A. V. Farnsworth, G. I. Kerley, J. M. McGlaun, S. V. Petney, S. A. Silling, P. A. Taylor, and L. Yarrington. CTH: A software family for multi-dimensional shock physics analysis. In R. Brun and L.D. Dumitrescu, editors, *Proceedings of the 19th International Symposium on Shock Physics*, volume 1, pages 377–382, Marseille, France, July 1993.

[15] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, New York, NY, 2001.

[16] Jay Lofstead, Ron Oldfield, Todd Kordenbrock, and Charles Reiss. Extending scalability of collective I/O through Nessie and staging. In *Proceedings of the 6th Parallel Data Storage Workshop*, Seattle, WA, November 2011.

[17] Jay Lofstead, Ron A. Oldfield, and Todd H. Kordenbrock. Experiences applying data staging technology in unconventional ways. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Delft, The Netherlands, May 2013. IEEE/ACM.

[18] Kenneth Moreland. A survey of visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 19(3):367–378, March 2013. DOI 10.1109/TVCG.2012.133.

[19] Kenneth Moreland, Nathan Fabian, Pat Marion, and Berk Geveci. Visualization on supercomputing platform level II ASC milestone (3537-1B) results from Sandia. Technical Report SAND 2010-6118, Sandia National Laboratories, September 2010.

[20] Kenneth Moreland, Nathan Fabian, Pat Marion, and Berk Geveci. Visualization on supercomputing platform level II ASC milestone (3537-1b) results from Sandia. Technical Report SAND2010-6118, Sandia National Laboratories, September 2010.

[21] Kenneth Moreland, C. Charles Law, Lisa Ice, and David Karelitz. Analysis of fragmentation in shock physics simulation. In *Proceedings of the 2008 Workshop on Ultrascale Visualization*, pages 40–46, November 2008.

[22] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Joudain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld,

Michael E. Papka, and Scott Klasky. Examples of in transit visualization. In *Proceedings of the PDAC 2011 : 2nd International Workshop on Petascale Data Analytics: Challenges and Opportunities*, Seattle, WA, November 2011.

[23] Ron A. Oldfield. Lightweight storage and overlay networks for fault tolerance. Technical Report SAND2010-0040, Sandia National Laboratories, Albuquerque, NM, January 2010.

[24] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Rolf Riesen, Maria Ruiz Varela, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, San Diego, CA, September 2007.

[25] Ron A. Oldfield, Brett W. Bader, and Peter Chew. Supporting multilingual document clustering on the Cray XT3. In *SIAM Conference on Parallel Processing and Scientific Computing*, February 2010.

[26] Ron A. Oldfield, Todd Kordenbrock, and Jay Lofstead. Developing integrated data services for Cray systems with a Gemini interconnect. In *Cray User Group Meeting*, April 2012.

[27] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[28] Ron A. Oldfield, Gregory D. Sjaardema, Gerald F. Lofstead II, and Todd Kordenbrock. Trilinos I/O Support (Trios). *Scientific Programming*, August 2012.

[29] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, September 2006.

[30] Ron A. Oldfield, Andrew Wilson, George Davidson, and Craig Ulmer. Access to external resources using service-node proxies. In *Proceedings of the Cray User Group Meeting*, Atlanta, GA, May 2009.

[31] Charles Reiss, Gerald Lofstead, and Ron Oldfield. Implementation and evaluation of a staging proxy for checkpoint I/O. Technical report, Sandia National Laboratories, Albuquerque, NM, August 2008.

[32] Rolf Riesen, Ron Brightwell, Patrick Bridges, Trammell Hudson, Arthur Maccabe, Patrick Widener, and Kurt Ferreira. Designing and implementing lightweight kernels for capability computing. *Concurrency and Computation: Practice and Experience*, 21(6):793–817, August 2008.

[33] Robert G. Schmitt, David A. Crawford, Raymond L Bell, and Eugene S. Hertel. Adaptive mesh refinement and multi-phase flow in the CTH. In *Workshop on Numerical methods for multi-material fluid flows*, Paris, France, September 2002.

[34] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object Oriented Approach to 3D Graphics*. Kitware Inc., fourth edition, 2004. ISBN 1-930934-19-X.

[35] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings ofSupercomputing '95*, page 57, San Diego, CA, December 1995. IEEE Computer Society Press.

[36] Venkatram Vishwanath, Mark Hereld, Vitali Morozov, and Michael E. Papka. Topology-aware data movement and staging for I/O acceleration on Blue Gene/P supercomputing systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 19:1–19:11, New York, NY, USA, 2011. ACM.

A Signed Letter from Committee

Date: March 5th, 2013
Subject: Achievement of ASC Level II Milestone 1

To Whom It May Concern:

On Tuesday, March 5th, 2013, a formal review of ASC Level II Milestone 4547, Data Co-Processing for Extreme Scale Analysis was held. We, the committee, found that the Milestone was completed on time, and demonstrated against the letter and spirit stated in the Milestone.

Committee members were Becky Springmeyer (LLNL), Berk Gevci (Kitware, Inc), Ron Brightwell (1423), Mike Glass (1545), Kim Mish (1542), Dino Pavlakos (9326), Kendall Pierson (1542), Jason Wilke (6634)

Sincerely,

Jim Ahrens, Milestone Committee Chair (LANL)

B Executive Summary Slides

Milestone 4547

Data Co-Processing for Extreme Scale Analysis

SAND# 2013-1427 P

Executive Summary of Milestone Report

March 5, 2013

David Rogers, Ron Oldfield, Kenneth Moreland and Nathan Fabian

Sandia National Laboratories

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

Summary

- Milestone 4745 "Data Co-Processing for Extreme Scale Analysis" was successfully completed on time, and demonstrated against the letter and spirit of stated Milestone.
- The Milestone Team completed over 10.5 million cpu hours of Cielo tests on both *in situ* and *in transit* analysis capabilities on a problem provided by a Sandia analyst.
- The results of these experiments have been detailed in a SAND report, which is published as an unclassified unlimited release document, available to the entire mod/sim community

The path to Exascale

Milestone 4745 is an important step in capability development, customer engagement, and scalability development on the path to exascale. It represents significant work on the development of both *Catalyst*, an open source *in situ* analysis capability, and *Nessie*, an open source data services capability.

This Milestone is part of an integrated R&D roadmap aimed at characterizing, understanding, and promoting solutions for complex analysis problems on advanced architectures.

It is an important foundation step in developing cross-cutting capabilities.

Milestone 4745

SC calculations produce complex datasets that are increasingly difficult to explore and understand using traditional post-processing workflows. To advance understanding of underlying physics, uncertainties, and results of ASC codes, SNL must gather as much relevant data as possible from large simulations. This drives SNL to couple data analysis and visualization capability with a running simulation, so that high fidelity data can be extracted and written to disk. This Milestone evaluates two approaches for providing such a coupling:

- In-situ processing provides ``tightly-coupled'' analysis capabilities through libraries linked directly with the simulation. SNL has collaborated on developing an in-situ capability designed for this purpose.
- In-transit processing provides ``loosely-coupled'' analysis capabilities by performing the analysis on separate processing resources. SNL provides this capability through a ``data services'' capability designed for this purpose.

SNL will engineer, test and evaluate customer-driven operations on large-scale data created by a running simulation. The data operations will be performed by instrumented versions of both the in-situ and in-transit solutions, with the resulting performance data published and made available to the ASC community.

A program review will be conducted, and its results documented. A report will be submitted as a record of milestone completion.

Motivation

SC calculations produce complex datasets that are increasingly difficult to explore and understand using traditional post-processing workflows. To advance understanding of underlying physics, uncertainties, and results of ASC codes, SNL must gather as much relevant data as possible from large simulations. This drives SNL to couple data analysis and visualization capability with a running simulation, so that high fidelity data can be extracted and written to disk.

- *Note: ASC program will benefit from a detailed understanding of the relationship between analyst tasks, analysis operations, and disk I/O performance.*

In situ and *in transit* workflows

- *In situ* processing provides ``tightly-coupled'' analysis capabilities through libraries linked directly with the simulation. SNL has collaborated on developing an *in situ* capability designed for this purpose.

Diagram of *in situ* workflow, accomplished in this Milestone through the use of *Catalyst*, an open source, VTK-based analysis library.

- *In transit* processing provides ``loosely-coupled'' analysis capabilities by performing the analysis on separate processing resources. SNL provides this capability through a ``data services'' capability designed for this purpose.

Diagram of *in transit* workflow, in which the science code communicates with data services nodes to perform analysis operations. This is accomplished in this Milestone through the use of *Nessie*, an open source data services library.

Milestone 4745, completion criteria

SC calculations produce complex datasets that are increasingly difficult to explore and understand using traditional post-processing workflows. To advance understanding of underlying physics, uncertainties, and results of ASC codes, SNL must gather as much relevant data as possible from large simulations. This drives SNL to couple data analysis and visualization capability with a running simulation, so that high fidelity data can be extracted and written to disk. This Milestone evaluates two approaches for providing such a coupling:

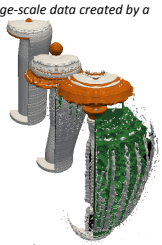
- In-situ processing provides ``tightly-coupled'' analysis capabilities through libraries linked directly with the simulation. SNL has collaborated on developing an in-situ capability designed for this purpose.
- In-transit processing provides ``loosely-coupled'' analysis capabilities by performing the analysis on separate processing resources. SNL provides this capability through a ``data services'' capability designed for this purpose.

SNL will engineer, test and evaluate customer-driven operations on large-scale data created by a running simulation. The data operations will be performed by instrumented versions of both the in-situ and in-transit solutions, with the resulting performance data published and made available to the ASC community.

A program review will be conducted, and its results documented. A report will be submitted as a record of milestone completion.

Experiment Driver

Milestone focused on *"customer-driven operations on large-scale data created by a running simulation"*



Customer driver use case: characterize fragments in an explosion simulation, an analysis step critical for understanding shock physics

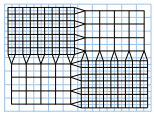
- Partner: Jason Wilke
- Critical steps
  - Find fragments (multiple operations required)
  - Characterize fragments (mass, velocity, etc.)
  - Extract useful information

Milestone experiments focused on identifying the fragments. This operation is a significantly complex part the analysis, so it serves as a useful way to characterize the operations in the driver use case.

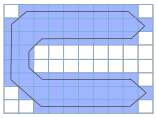
Full range of data experiments run at 32k cores on Cielo. Partial experiments done at 64k cores on Cielo. This report presents results from the 32k runs.

Fragment detection

- Operations required for fragment detection (requires a watertight surface)
  1. Find block neighbors
  2. Build a conforming mesh over AMR boundaries
  3. Identify boundaries of fragments



Step 2



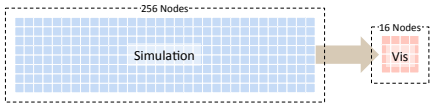
Step 3

Implemented Workflows

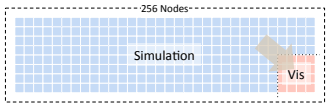
- ***In situ***: A CTH job that directly runs *in situ* data analysis
  - **Baseline**: Basic algorithm with somewhat redundant step of global communication to find AMR block neighbors
  - **Refined**: Improved algorithm that gets AMR block neighbors from CTH
- ***In transit***: CTH transfers data to separate server job
  - **Extra nodes**: CTH job size same as other runs, extra nodes are used to allocate the VDA service
  - **Internal nodes**: CTH job given fewer nodes that are assigned to VDA service so that together both jobs use the same nodes as other runs
- **Post-processing**: Write Spyplot files from CTH, then post process analysis by reading back in and batch processing in ParaView.

In Transit Allocations

"Extra Nodes" allocated for VDA services



"Internal Nodes" included in job allocation



Experiment Configurations

- All experiments performed on Cielo supercomputer at LANL, jointly managed by Los Alamos National Laboratory and Sandia National Laboratories
  - 8,944 node Cray XE6
  - Node: 2 AMD Opteron 6136 (Magny-Cours) 8-way processor chips
    - Total of 16 cores/node
    - 2.4 GHz peak computation speed per core
  - Peak of 1.37 Petaflops
  - 32 GB memory/node

Experiment, cont'd

- All applications complete 500 cycles (i.e., timestep calculations) of the CTH code.
- The first four applications execute an analysis operation once every 10 cycles
- Spyplot file application outputs spyplot data at a fixed interval in simulated time, calculated so that the application executed the same number of analysis operations performed by the *in situ* and *in transit* applications
  - Total number of analysis operations is the same
- Data captured was from instrumented code and HPCToolkit

Experiment, cont'd

- For each application, we ran strong scaling experiments for three different datasets.
  - Each data set comes from the same initial conditions but with a different maximum level of refinement
  - Measurements of different job sizes with different data set sizes provides a weak scaling overview.

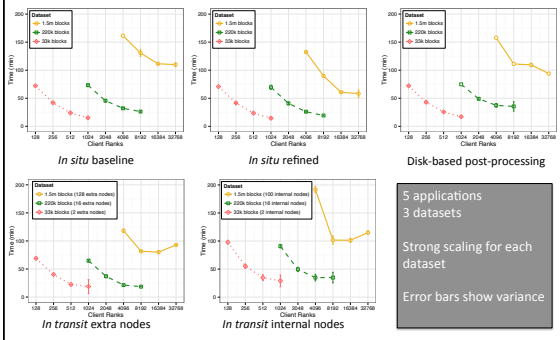
CTH				*In transit* Server			
Most		*In transit*	Internal	Extra Nodes		Internal Nodes	
Cores	Nodes	Cores	Nodes	Cores	Nodes	Cores	Nodes
33K Blocks — 5 levels							
128	8	96	6	16	2	16	2
256	16	224	14	16	2	16	2
512	32	480	30	16	2	16	2
1,024	64	992	62	16	2	16	2
220K Blocks — 6 levels							
1,024	64	768	48	128	16	128	16
2,048	128	1,792	112	128	16	128	16
4,096	256	3,840	240	128	16	128	16
8,192	512	7,936	496	128	16	128	16
1.5M Blocks — 7 levels							
4,096	256	2,496	156	1,024	128	800	100
8,192	512	6,592	412	1,024	128	800	100
16,384	1,024	14,784	924	1,024	128	800	100
32,768	2,048	31,168	1,948	1,024	128	800	100
65,536	4,096	63,936	3,996	1,024	128	800	100

Table shows the range of core sizes used for the various experiments. For every application we used the maximum 16 cores-per-node for the CTH client, since CTH is primarily bound by computation and scales very well.

CTH				*In transit* Server			
Most		*In transit*	Internal	Extra Nodes		Internal Nodes	
Cores	Nodes	Cores	Nodes	Cores	Nodes	Cores	Nodes
33K Blocks — 5 levels							
128	8	96	6	16	2	16	2
256	16	224	14	16	2	16	2
512	32	480	30	16	2	16	2
1,024	64	992	62	16	2	16	2
220K Blocks — 6 levels							
1,024	64	768	48	128	16	128	16
2,048	128	1,792	112	128	16	128	16
4,096	256	3,840	240	128	16	128	16
8,192	512	7,936	496	128	16	128	16
1.5M Blocks — 7 levels							
4,096	256	2,496	156	1,024	128	800	100
8,192	512	6,592	412	1,024	128	800	100
16,384	1,024	14,784	924	1,024	128	800	100
32,768	2,048	31,168	1,948	1,024	128	800	100
65,536	4,096	63,936	3,996	1,024	128	800	100

Results

Total Runtime for All Experiments



In situ baseline | *In situ* refined | Disk-based post-processing

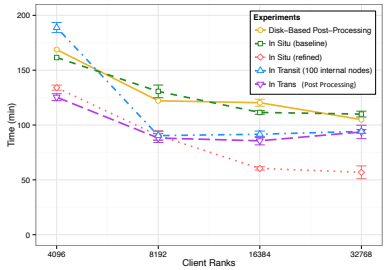
In transit extra nodes | *In transit* internal nodes

5 applications
3 datasets

Strong scaling for each dataset

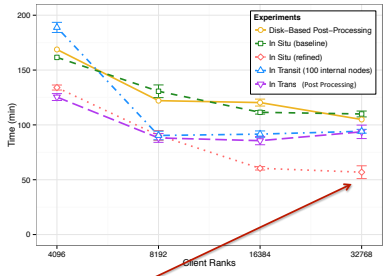
Error bars show variance

Pipeline Summary Timing (1.5m blocks)



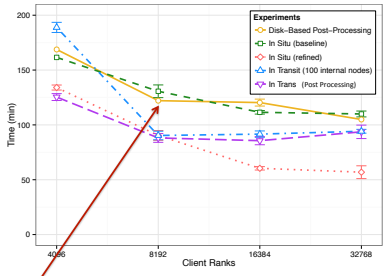
Acceptable scaling performance, with the exception of the baseline algorithm.

Pipeline Summary Timing (1.5m blocks)



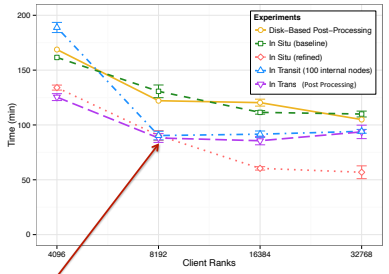
No significant improvement at 32K cores. Probably insufficient work for analysis (only 45 blocks per process).

Pipeline Summary Timing (1.5m blocks)



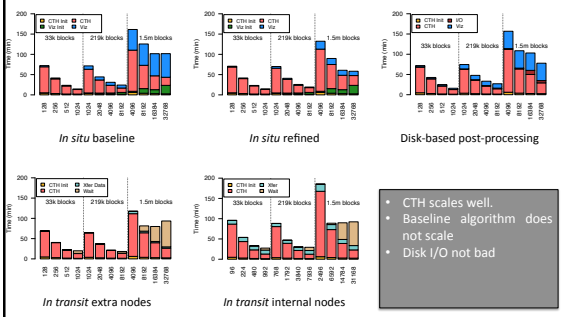
Writing files surprisingly fast. Although slower than most alternatives, still a viable option.

Pipeline Summary Timing (1.5m blocks)



"Sweet spot" at 8K cores: *in transit* with unrefined algorithm equal to *in situ* with refined algorithm.

Timing Per Task

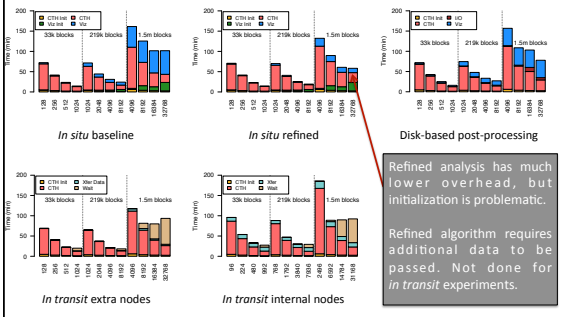


In situ baseline | *In situ* refined | Disk-based post-processing

In transit extra nodes | *In transit* internal nodes

- CTH scales well.
- Baseline algorithm does not scale
- Disk I/O not bad

Timing Per Task



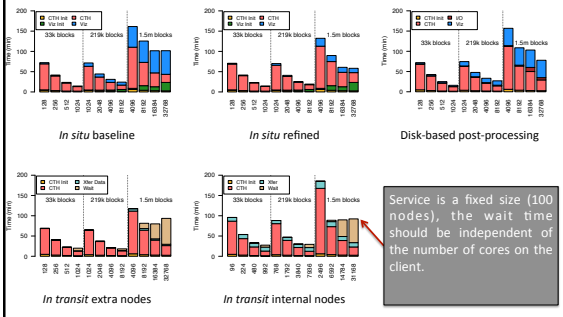
In situ baseline | *In situ* refined | Disk-based post-processing

In transit extra nodes | *In transit* internal nodes

Refined analysis has much lower overhead, but initialization is problematic.

Refined algorithm requires additional data to be passed. Not done for *in transit* experiments.

Timing Per Task

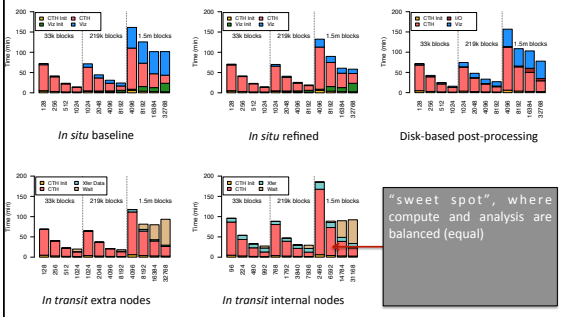


In situ baseline | *In situ* refined | Disk-based post-processing

In transit extra nodes | *In transit* internal nodes

Service is a fixed size (100 nodes), the wait time should be independent of the number of cores on the client.

Timing Per Task



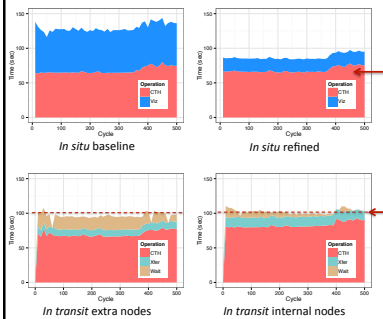
In situ baseline | *In situ* refined | Disk-based post-processing

In transit extra nodes | *In transit* internal nodes

"sweet spot", where compute and analysis are balanced (equal)

Time-Series Analysis (8k cores)

10-cycle increments



In situ baseline | *In situ* refined

In transit extra nodes | *In transit* internal nodes

Although number of blocks changes very little, CTH runtime gets longer as simulation progresses.

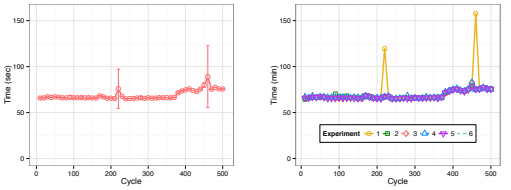
Vis time is roughly constant.

In transit will "win" when xfer+wait is less than viz.

In transit can flatten the runtime as long as extra simulation time consumes only wait time.

Time-Series Analysis: Variance

10-cycle increments

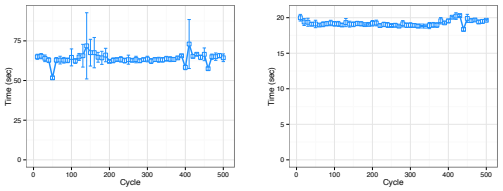


CTH (mean+std) | CTH (overlaid)

Anomalies that cause large variance are clearly identified in overlay plot.
Not sure the true cause of the outliers.

Time-Series Analysis: Variance

10-cycle increments

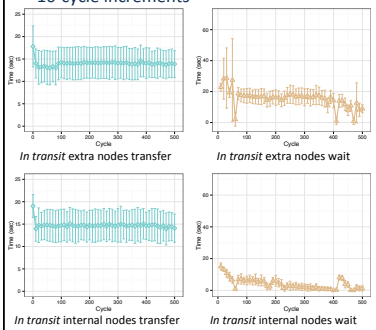


In situ baseline | *In situ* refined

Refined algorithm has much less communication, resulting in less variance.
Scales are different in the two plots.

Time-Series Analysis: Variance

10-cycle increments



In transit extra nodes transfer | *In transit* extra nodes wait

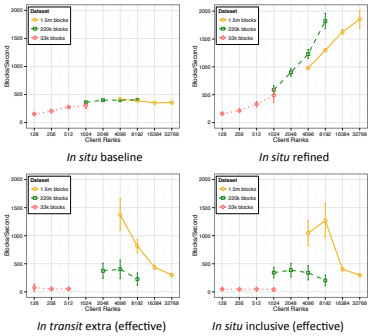
In transit internal nodes transfer | *In transit* internal nodes wait

In transit transfer times have noticeable variance.

Possibly caused by placement issues.

Scheduler does not optimize allocations for transfers between jobs.

Block Processing Rate (Viz)



In situ baseline | *In situ* refined

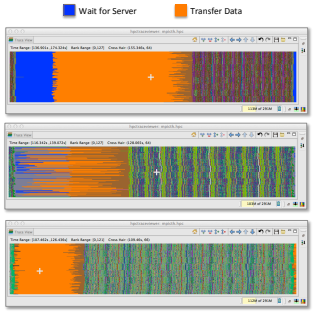
In transit extra (effective) | *In situ* inclusive (effective)

Refined algorithm is scalable, baseline algorithm is not. (Verifies previous work.)

In-transit plots show "effective" processing rate. Since viz time is flat, we expect the effective rate to decrease.

At 4k and 8k, effective processing rate of in transit outperforms in situ.

In Transit Node Scaling

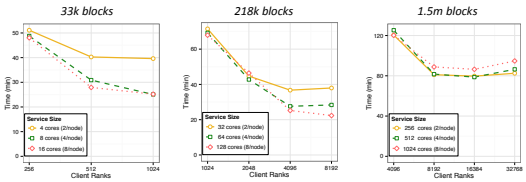


2 server cores: 64:1
- 10 cycles in 37 secs
- Client idle waiting for servers (also affects xfers)

4 server cores: 32:1
- 10 cycles in 23 secs

8 server cores: 16:1
- 10 cycles in 19 secs
- Less than 1% time waiting

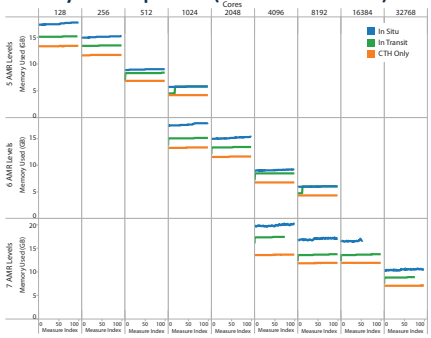
In Transit Node Scaling



For small datasets, there is clear benefit to using 4 and 8 cores/node (agreement previous slide)

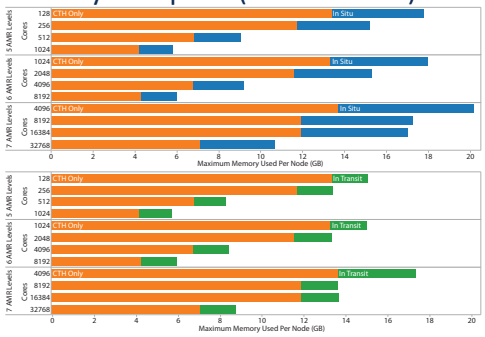
For 1.5m blocks datasets (at large scale), the opposite appears to be true. Needs further study.

Memory Footprint (on code side)



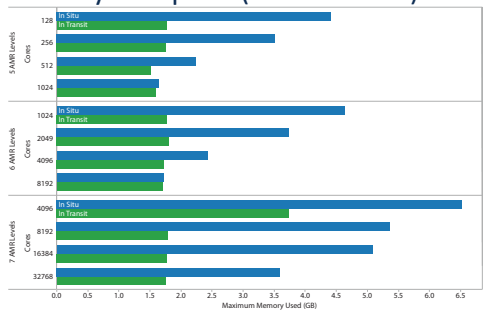
All memory measurements. Holds relatively steady for all workflows

Memory Footprint (on code side)



Memory overhead generally falls between 25% and 50%

Memory Footprint (on code side)



In transit generally has less memory overhead, but requires extra nodes allocated

Conclusions

Conclusions

***In transit* can provide a performance improvement over *in situ* in some circumstances, but the window is narrower than we initially expected it would be.**

In transit data analysis has an added overhead above embedded *in situ* data analysis involving transferring data between parallel jobs. Given a data analysis algorithm with perfect linear scalability, we suspect *in transit* workflows will always have an added cost, and our results support this. With a data analysis algorithm that does not scale perfectly, possibly due to communication overhead, it is theoretically possible for *in transit* to be faster by reducing the size of the data analysis job. This is one of the motivations for choosing a data analysis task that requires significant communication. In our results, we do find instances where *in transit* is faster, but by a smaller margin and for fewer configurations than we initially anticipated. So although *in transit* has several other positive features, we do not anticipate performance to be the main motivations for using it.

Conclusions

The efficiency of *in transit* relies on balancing the time spent in simulation and data analysis.

The significant overhead cost, apart from data transfer, in the *in transit* workflow is the idle time spent in the simulation waiting for the visualization and data analysis service to become ready or the idle time spent in the visualization and data analysis service waiting for the simulation to send more data. This idle waiting time is minimized when the simulation and data analysis spend the same amount of wall clock time between transfers. Although not demonstrated in this work, it is possible to "auto-balance" the work between simulation and data analysis by, at every iteration of the simulation, transfer data to the data analysis if and only if the data analysis service is ready to accept more work. The disadvantage of such an approach is that the idle process time could be replaced with unnecessary extra data analysis or less data analysis than necessary. However, we suspect that controlling the amount of visualization and data analysis performed through job allocation sizes fits well with users' rules of thumb about resource allocation.

Conclusions

Memory overhead will be an important trade-off space.

The baseline amount of memory added to the CTH job to perform *in situ* processing is roughly 100MB per core. Considering that our embedded *in situ* library is a fully featured visualization toolkit containing over 2 million lines of code and algorithms developed over almost 2 decades, this overhead is not unreasonable. Nevertheless, this footprint can be problematic for simulations already tight on memory. Because of this, efforts are already underway to improve our memory footprint by making finer modules and being more selective on the available algorithms. This, of course, requires a compromise between the size of the library and the algorithms that are dynamically available. We also note that our algorithm has the potential to generate sizable meshes of its own. Thus, it may be fruitful to pursue and support incremental algorithms where possible.

Conclusions

Initialization time matters

Our scaling efforts to date focus on the scalability of the algorithms invoked during the run of a simulation. The initialization cost, a one-time penalty, has yet to be seriously considered. However, based on our HPCToolkit measurements, initialization becomes a significant cost at high process counts.

Disk-based I/O is not dead . . . yet.

Our initial assumption was that it would not be feasible to output full results at a fine enough temporal resolution from CTH to disk storage to perform our high fidelity data analysis. However, our control workflow shows that although the overall time to write data to disk and then read back again incurs a large cost, it is still realistic to do so. Thus, users may still choose to incur the extra overhead to use a traditional offline post-processing visualization and data analysis workflow.

Conclusions

Better job scheduling is important

One of the more complicated parts of running an *in transit* workflow is scheduling the simulation job and service job to run in tandem. Frankly, the capabilities of the scheduler are inadequate for our needs. We cannot start and stop jobs independently and make reconnections dynamically. Another experiment we would like to do but is challenging to schedule is to allow simulation and service to share nodes. Since each node has 16 cores, perhaps we could get better transfer performance by allocating one core per node for service and the rest for simulation. A similar scheduling scheme will be important to take advantage of burst buffers in future architectures.

Future Work

- Algorithm comparison. Three similar algorithms with three different scaling behaviors
  - Contour algorithm (perfectly scalable)
  - Refined water tight contours (reasonably scalable)
  - Baseline water tight contours (not scalable)
- No-wait analysis (*in transit*)
  - Perform analysis if and only if the service is ready
- Investigate initialization cost of *in situ* vis
- Zero copy transfers (*in transit*)
- Additional apps at Cielo scale
- Improved OS and runtime support
  - Scheduling, placement, node sharing, specialized runtimes, …

Summary

- Milestone 4745 "Data Co-Processing for Extreme Scale Analysis" was successfully completed on time, and demonstrated against the letter and spirit of stated Milestone.
- The Milestone Team completed over 9 million node hours of Cielo tests on both *in situ* and *in transit* analysis capabilities on a problem provided by a Sandia analyst.
- The results of these experiments have been detailed in a SAND report, which is published as an unclassified unlimited release document, available to the entire mod/sim community

DISTRIBUTION:

1 James Ahrens
Los Alamos National Laboratory
P.O. Box 1663
Los Alamos, NM 87545

1 Berk Geveci
Kitware, Inc.
28 Corporate Drive
Clifton Park, NY 12065

1 Lucy Nowell
U.S. Department of Energy
SC-21
19901 Germantown Road
Germantown, MD 20874-1290

1 Becky Springmeyer
Lawrence Livermore National Laboratory MS 555
P.O Box 808
7000 East Ave.
Livermore, CA 94551

1	MS 1319	Ronald Brightwell, 01423
1	MS 0380	Micheal Glass, 01545
1	MS 1319	Suzanne Kelly, 01423
1	MS 0380	Kyran Mish, 01542
1	MS 0823	Constantine Pavlakos, 09326
1	MS 0380	Kendall Pierson, 01542
1	MS 0783	Jason Wilke, 06634
2	MS 1326	Nathan Fabian, 01461
2	MS 1326	Kenneth Moreland, 01461
2	MS 1319	Ron Oldfield, 01423
2	MS 1326	David Rogers, 01461
2	MS 0822	Jeff Mauldin, 09326
2	MS 0822	Warren Hunt, 09326
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories