2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-8106-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IPDPS53621.2022.00059

Colza: Enabling *Elastic* In Situ Visualization for High-performance Computing Simulations

Matthieu Dorier[†§], Zhe Wang[‡§], Utkarsh Ayachit[§], Shane Snyder[†], Rob Ross[†] and Manish Parashar[‖]
[†]Argonne National Laboratory, 9700 S. Cass Ave., Lemont, Illinois 60439 – Email: {mdorier,ssnyder,rross}@anl.gov
[‡]Rutgers University, New Jersey – Email: jay.wang@rutgers.edu
[§]Kitware, Inc. – Email: utkarsh.ayachit@kitware.com
[‖]University of Utah, Salt Lake City, Utah 84112 – Email: manish.parashar@utah.edu

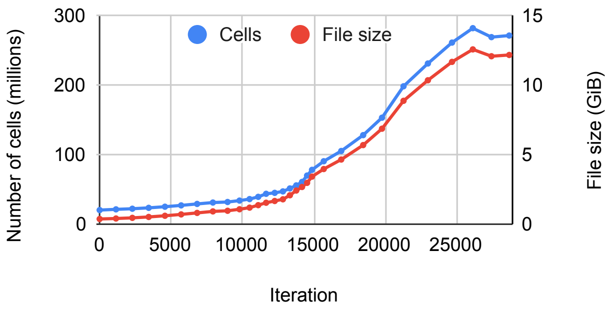
***Abstract*—In situ analysis and visualization have grown increasingly popular for enabling direct access to data from high-performance computing (HPC) simulations. As a simulation progresses and interesting physical phenomena emerge, however, the data produced may become increasingly complex, and users may need to dynamically change the type and scale of in situ analysis tasks being carried out and consequently adapt the amount of resources allocated to such tasks. To date, none of the production in situ analysis frameworks offer such an elasticity feature, and for good reason: the assumption that the number of processes could vary during run time would force developers to rethink software and algorithms at every level of the in situ analysis stack. In this paper we present Colza, a data staging service with elastic in situ visualization capabilities. Colza relies on the widely used ParaView Catalyst in situ visualization framework and enables elasticity by replacing MPI with a custom collective communication library based on the Mochi suite of libraries. To the best of our knowledge, this work is the first to enable elastic in situ visualization capabilities for HPC applications on top of existing production analysis tools.**

I. Introduction

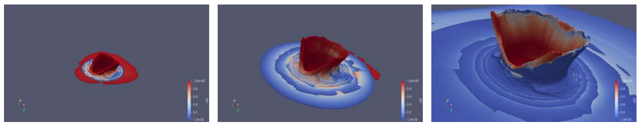
In situ analysis and visualization consist of coupling a high-performance computing (HPC) simulation with an analysis and visualization framework to get live insight into the simulation's evolution. This approach has gained popularity thanks to its adoption by long-standing parallel visualization software such as ParaView [1] and VisIt [2]. Resource allocation for in situ analysis has been the subject of many research works [3], with solutions ranging from periodically interrupting the simulation to use its resources, to dedicating resources to in situ tasks. However, *these solutions assume that the visualization workload remains similar throughout the simulation's run time*, a situation that rarely happens in practice. Indeed, as a simulation progresses and complex physical phenomena appear, the cost of running analysis and visualization algorithms changes, and the initial resource allocation may no longer be optimal.

As an example, Figure 1a shows the number of cells in the unstructured mesh produced by the Deep Water Impact simulation [4] as it progresses, as well as corresponding file sizes in VTK format. Figure 1b illustrates the volume rendering of three iterations taken at the beginning, in the middle, and at the end of the run. This example shows that not only does the data size increase over time, so does the rendering complexity, justifying a modest initial allocation of visualization resources that are augmented over the course of the run.

[§]These authors contributed equally to the work.



(a) Data size (number of cells and corresponding file sizes) of the dataset.



(b) Rendering of the Deep Water Impact dataset using ParaView.

Fig. 1: Analysis of the data produced by the Deep Water Impact simulation, and rendering of three iterations at various stages of the simulation.

With HPC applications moving towards more complex workflows, dynamic resource management for in situ analysis has been identified by the community as a key research direction to enable faster scientific discoveries [5], [6], [7]. Hence the question arises: Can we make in situ analysis *elastic*, that is, capable of accommodating run-time changes in the number of processes executing analysis/visualization tasks, without having to restart the entire workflow?

This question is arguably difficult to answer. On the one hand, existing parallel analysis and visualization frameworks, and the algorithms and libraries they depend on, all assume that the number of processes will remain fixed throughout run time, making their use difficult in an elastic context. The MPI standard, on which they all rely, does not yet provide sufficient functionalities to grow and shrink communicators during run time. On the other hand, designing and implementing a brand new elastic in situ analysis framework from scratch entail an enormous amount of work before this framework can start offering a fraction of the features and performance that existing, nonelastic frameworks already provide.

Therefore, *it is critical to be able to provide elasticity using existing, widely-used in situ visualization technologies.*

In this paper we propose Colza, a modular, *elastic* data-staging service enabling in situ analysis and visualization via ParaView/Catalyst. We built Colza using Mochi [8], a set of building blocks for rapidly developing highly efficient and portable HPC data services. Colza uses dependency-injection to replace MPI in ParaView and its image-compositing library IceT with our own communication layer, called MoNA.

We evaluate Colza with three applications on the Cori supercomputer at NERSC. Our experiments show that (1) typical in situ visualization algorithms perform just as well in Colza when MPI is replaced with our custom communication layer, making it usable in non-elastic contexts, and (2) our solution effectively enables adding and removing processes at run time without needing to restart the simulation nor the staging area.

To the best of our knowledge, Colza is the first data-staging service to provide elastic in situ analysis and visualization capabilities for HPC applications.

The rest of this paper is organized as follows. Section II presents Colza. Section III evaluates it on the Cori supercomputer using three applications, comparing it against an MPI-based version of the same rendering pipelines and against state-of-the-art data staging services. Some aspects of elasticity related to job scheduling and triggers are discussed in Section IV. Section V puts our contribution in perspective with related work. We conclude in Section VI with a summary and a discussion of future work.

II. The Colza elastic framework

The first challenge posed by elastic in situ analysis comes from the fact that existing parallel analysis frameworks rely on MPI, which assumes a static number of processes throughout run time. While MPI provides `MPI_Comm_spawn(_multiple)` to spawn new processes and `MPI_Comm_accept/connect` to establish connections, these functionalities are often not implemented by vendors or have limited support. Proposals to enable communicators to grow and shrink have been around for some time in the context of fault tolerance [9], [10], but no such features have made it into the standard yet. While we are hopeful that MPI will eventually provide new functionalities for dynamic process management, we have to work around those limitations if we are to implement an elastic solution right now. In fact, *successful use cases of elastic computing in HPC might be necessary to motivate the inclusion of those features into the MPI standard.*

A second challenge is to be able to rely on existing, widely-used visualization software, such as ParaView and its in situ library Catalyst. These frameworks not only provide well-tested, well-performing analysis and visualization algorithms, which would be nearly impossible to redesign from scratch just for the sake of enabling elasticity, they are also familiar to users, enabling wider adoption of our solution.

In light of these challenges, we developed Colza, an elastic staging area for in situ analysis and visualization. This section details its architecture, interface, and implementation.

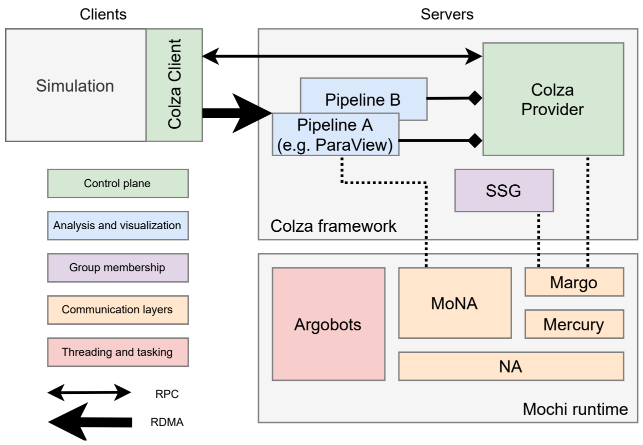


Fig. 2: Overview of the Colza architecture.

A. Architecture and interface overview

Colza is based on Mochi, a set of tools created by Argonne National Laboratory, Carnegie Mellon University, Los Alamos National Laboratory, and The HDF Group, for developing efficient HPC data services. Mochi encourages the development and use of software components managing specific aspects of a service, such as threading, remote procedure calls (RPCs), and group management. It relies on Argobots [11] for threading and on Mercury [12] for RPCs and RDMAs. The Margo library binds Mercury and Argobots together to hide Mercury's network progress loop in an Argobots user-level thread. Margo is used by Colza for control messages across servers and for communications with the simulation.

Figure 2 provides an overview of the Colza architecture. In each staging-area process a Colza *provider* manages *analysis pipelines*, forwarding simulation data to them for processing. The Colza providers not only interact with the client application, they also react to membership changes by registering callbacks to Mochi's scalable service groups (SSG) library. SSG manages group membership and tracks servers as they join and leave, using the SWIM gossip protocol [13]. To enable collective communications across staging area processes in parallel analysis algorithms, we implemented our own custom communication library, called MoNA. The next sections dive deeper into each of these individual components.

B. Colza's core: pipelines and pipeline handles

Central to our framework is the notion of *user-provided pipelines*. A pipeline is a C++ object instantiated on each server that stages incoming data and executes some analysis tasks on it. Colza pipelines are implemented by the user as a C++ class inheriting from the abstract `colza::Backend` class. They are compiled into shared libraries and dynamically loaded *on demand* based on current requirements. They can include any type of processing, depend on additional libraries (e.g. we use Catalyst in this paper), and even leverage on-node accelerators and GPUs. This design allows users to deploy the staging area without any pipeline to begin with, and later decide which pipelines to load and execute based on what they see happening in their simulation. Pipelines that execute parallel operations must have an instance on each process of the staging area.

Pipeline objects provide the following methods, which are exposed as RPCs to the simulation.

- `activate(iteration)`: tells the pipeline that the given iteration of analysis is about to start, and freezes the number of servers, preventing additions and removals by the group membership component until `deactivate` is called .
- `stage(... /* data and metadata */ ...)`: requests the pipeline to stage some data by pulling it from the simulation's memory via RDMA.
- `execute(iteration)`: executes the pipeline's analysis function on the staged data.
- `deactivate(iteration)`: tells the pipeline that the iteration is complete and the staged data can be cleaned up. Processes may now freely join and leave the staging area until the next analysis iteration starts.

Colza provides both a C++ and a Python API. Simulation processes interact with pipelines either via a *pipeline handle*, which references a specific pipeline in a specific server, or most often via a *distributed pipeline handle*, which references a collection of pipeline instances across multiple servers. Both interfaces provide the `activate`, `stage`, `execute`, and `deactivate` functions to their underlying remote pipeline(s), as well as non-blocking versions of these functions. The `stage` function does not send data directly to the target pipeline. Instead, it sends a memory handle along with some metadata (field name, dimensions, type, etc.). The receiving pipeline instance is responsible for pulling data from the simulation's memory using RDMA on the memory handle.

Simulations are expected to issue an `activate` RPC to start a new iteration of in situ analysis, followed by one or more `stage` RPC to send data, an `execute` RPC to process the staged data, and a `deactivate` RPC to indicate that the iteration has completed.

When using a distributed pipeline handle, the `activate`, `execute`, and `deactivate` functions respectively broadcast an `activate`, `execute`, and `deactivate` RPC to all the underlying pipelines, while the `stage` function selects an individual pipeline instance to receive each pieces of data. By default, this selection is based on a *block id* provided as part of the metadata, but users can change this policy.

An "admin" interface to the Colza provider, in the form of a separate library, offers an additional set of RPCs to create and destroy pipelines and to request a server to leave the staging area and shut down. The interface to create a pipeline takes the address of the node in which to deploy it, the name of the pipeline, the path its shared library, and an optional JSON-formatted configuration string that the pipeline may use during initialization. This admin library can be used by the simulation, by the user via external tools, or by any agent that needs to modify the size of the staging area or the type of analysis being carried out. We kept it separate from Colza's client library because of the entirely different nature of its functionalities.

C. MoNA: Communications across pipelines

Distributed analysis rarely involves individual pipelines that can operate without communications. Even a pipeline as simple as computing an average across the data received by multiple staging servers needs a reduction operation. To handle such communications, we developed MoNA,[1] a library built on top of Argobots and NA, Mercury's messaging layer. While Mercury provides RPC on top of NA, MoNA provides collective communications. MoNa was built from the ground up for an environment where communicating processes come and go. After initializing a MoNA progress loop, represented by a `mona_instance_t` handle, a MoNA communicator can be built from a list of addresses using the `mona_comm_create(...)` function. This list of addresses is obtained from the group membership component, SSG, described in Section II-E. From there, MoNA provides collective communication functions similar to MPI, such as `mona_comm_bcast`, `mona_comm_reduce`, and their non-blocking counterparts.

We implemented typical tree-based algorithms for collective operations in MoNA, taking inspiration from the MPICH source code.[2] We do not expect MoNA to perform better than MPICH or OpenMPI, let alone vendor-provided MPI implementations tailored to specific platforms, but MoNA provides at least two advantages over MPI.

First, MoNA can be more easily integrated in the Mochi ecosystem than MPI. Its use of Argobots makes its progress loop capable of yielding to other tasks (such as pipeline execution or control messages) when a user-level thread is blocking on MoNA communications. In MPI, such communications would block the core they execute on, preventing other tasks from executing and therefore wasting resources.[3]

Second, MoNA does not have a notion of *world communicator*, and new communicators can be created as desired when new processes join. This capability makes it much easier to integrate in a system that aims to be elastic.

D. Integration of VTK, IceT, and ParaView

We used the plugable pipeline mechanism described above, along with MoNA, to implement concrete pipelines using ParaView's Catalyst in situ visualization library. Although ParaView and its dependencies rely on MPI for parallelism, the Kitware developers already abstracted communications in VTK, ParaView, and in the IceT image compositing library.

VTK provides two abstract classes to handle communications: `vtkMultiProcessController` and `vtkCommunicator`. Both provide an interface for point-to-point and collective communication functions. Their respective child classes `vtkMPIController` and `vtkMPICommunica-`

[1]https://github.com/mochi-hpc/mochi-mona/

[2]https://www.mpich.org/

[3]At the time of writing, MPICH provides the option to compile with Argobots support and would also yield to other Argobots threads when blocking on communication. However, vendor-provided implementations do not provide this capability.

`tor` rely on MPI to provide concrete implementations of this interface. Other VTK classes are agnostic to this implementation. Hence we were able to replace MPI with MoNA by implementing a `vtkMonaController` class and a `vtkMonaCommunicator` class. Making VTK use these classes is done by instantiating a `vtkMonaController` and passing it to `vtkMultiProcessController`'s `SetGlobalController` function prior to setting up the in situ visualization pipeline.

Similarly, VTK's image compositing library IceT, written in C, provides an `IceTCommunicator` structure that lists function pointers for point-to-point and collective communication primitives. The only current implementation of this structure is based on MPI. We provided our own implementation based on MoNA.

Neither VTK nor IceT was modified for us to add MoNA support. ParaView, however, needed to be. ParaView indeed currently creates an `IceTCommunicator` from a `vtkCommunicator` by downcasting the latter to a `vtkMPICommunicator`, extracting its internal `MPI_Comm` handle, and creating an MPI-based `IceTCommunicator` from it. To avoid introducing a dependency on MoNA in ParaView, we added a factory function mechanism in the `vtkIceTContext` class, allowing us to register functions to convert other child classes of `vtkCommunicator` into their corresponding IceT counterpart. We also found that ParaView was initially unable to be reinitialized with a different communicator. Solving this issue required the help of Kitware developers.

These modifications make it possible for users to implement their pipeline the same way they would normally do with Catalyst, or reuse their existing Catalyst pipelines.

E. Group membership and consistency

Colza uses Mochi's Scalable Service Group[4] (SSG) to handle group membership. SSG relies on the SWIM protocol, which provides an eventually consistent *view* (list of members) of the group to all its members using gossip. When a new process joins the staging area, it contacts one of the existing members. This member sends its view to the new process and sends the information about the new member to another randomly selected group member periodically for a given time period. These group members will also forward this information to other randomly selected members. Eventually, all the processes learn about the new member. The same mechanism applies when a process leaves the group. To address the fact that SSG only provides an eventually consistent membership view, we rely on a two-phase commit (2PC) protocol between clients and Colza servers to ensure that all the parties have the same view when `activate` is called.

We chose this approach to minimize development time, since SSG is currently the de-facto group membership component provided by Mochi. We acknowledge however that other strategies could have been (and in the future will be) explored.

Due to space constraints, we leave the in-depth evaluation of the overhead of group management to a future extension of this paper. Our experiments however show that it does not incur any overhead if the group hasn't changed when `activate` is called, and an overhead in the order of a second when the group did change. Furthermore, this overhead depends on SSG's configuration parameters such as how frequently information is exchanged across members.

F. Triggering elasticity

Triggering a change in Colza resources is done differently depending on whether resources are added or removed.

a) Scaling up

Adding a node requires allocating it via the platform's resource manager and starting a Colza daemon on it. The Colza daemon will read connection information from a file generated by existing Colza processes and use SSG to automatically join the existing group. The addition of a node can therefore be triggered by any entity (in particular the user) with the ability to request node allocations from the resource manager. The scientific application itself, or even existing processes of the staging area, could request such addition, provided that a mechanism is available for them to request resources. This could be implemented for example using PMIx [14]. In our experiments in Section III, a job script periodically launches new Colza processes on new nodes.

b) Scaling down

Removing a node is done by sending an RPC to that node, requesting it to leave the staging area. This is done using Colza's admin library. This library can be used by the simulation itself (e.g. if the simulation is able to assess its in-situ requirements), the user via a standalone program, the resource manager, or Colza itself.

Overall, Colza offers various ways of triggering elasticity, which will enable us in the future to study multiple ways of leveraging it, from user-driven interactive in situ analysis, to AI-based monitoring and autotuning methods.

III. Evaluation

In the following, we evaluate our Colza in situ service. More specifically, we first show that adding new nodes via SSG is faster than restarting the full service. We then compare the performance of MoNA against MPI for relevant collective communication operations, in a standalone manner and within the context of in situ visualization tasks. We also show that Colza's performance is on par with two state-of-the-art in situ analysis frameworks: Damaris and DataSpaces. Finally we show that Colza effectively enables elasticity by incrementally adding server nodes while a simulation is running.

A. Platform and applications

We evaluate Colza on NERSC's Cray XC40 Cori supercomputer,[5] using its Haswell partition, which contains 2,388 compute nodes equipped with 32 Intel Xeon E5-2698 v3 1.4 GHz processors and 128 GB DDR4 2133 MHz memory. These nodes are interconnected via a Cray Aries network

[4]https://github.com/mochi-hpc/mochi-ssg/

[5]https://docs.nersc.gov/systems/cori/

arranged in a Dragonfly topology with more than 45 TB/s global peak bisection bandwidth.

Aside from benchmarks, we rely on three applications as the data sources for our experiments.

The Gray–Scott simulation[6] uses reaction-diffusion equations to simulate complex spatial and temporal patterns [15]. It uses a three-dimensional Cartesian partitioning of a regular grid across its processes, generating the same amount of data per process at every iteration.

The Mandelbulb miniapp[7] produces a three-dimensional Mandelbrot fractal [16] and aims at stressing visualization pipelines with complex mesh geometries. It also uses a regular grid, but this grid is partitioned across processes along the z axis, and each process may be in charge of multiple blocks.

Both Gray–Scott and Mandelbulb were originally written as examples for tutorials on the ADIOS I/O system [17] and the ParaView/Catalyst in situ library, respectively.

The Deep Water Impact (DWI) proxy is a Python application that we wrote, based on the *mpi4py* and *meshio* libraries. It reads VTU files produced by the Deep Water Impact simulation [4], and feeds the data into an in situ visualization framework. We use it with a subset of the Deep Water Impact Ensemble Dataset [18] in place of the real simulation for simplicity. The simulation that produced this dataset originally ran on 512 processes, for 30,000 iterations. We use files corresponding to 30 of these iterations, roughly equally spaced throughout the run.[8] At every iteration, the DWI proxy reads the 512 corresponding VTU files, distributing them evenly across available client processes. Each file contains an unstructured mesh along with point and cell data.

The modular implementation of Colza described in Section II-D, along with ParaView Catalyst's ability to run Python scripts in situ, make it easy for us to express visualization tasks as scripts directly exported from ParaView. Figure 3 illustrates the rendered results for Mandelbulb and Gray–Scott. For the Gray–Scott application, we combine multiple levels of isosurfaces with clipping to look at what is happening inside the domain. Figure 3a shows the seed of the simulation at the center, in blue, surrounded by random noise, in red. For Mandelbulb, we use a single level of isosurface. The pipeline used for DWI includes block merging, followed by volume rendering of the unstructured mesh, colored with the velocity field. The rendered result was shown in Figure 1b.

In all our experiments, the application and Colza use distinct nodes. The application runs on a fixed number of nodes, whereas Colza may use a varying number of nodes if elasticity is enabled.

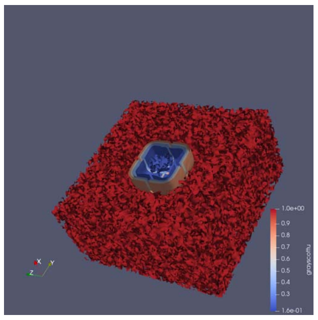
B. Advantages of elasticity in resizing times

The first set of experiments aims to show how much time is required to increase the size of a staging area. Since it requires asking the job manager for resources, which could take anywhere from seconds to hours depending on the machine's activities, we leave job-manager aspects aside and we consider what happens from the instant the job manager gives us extra nodes to work with.[9]

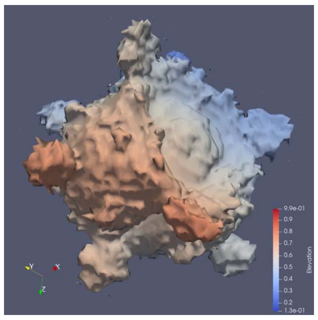
[6]https://github.com/pnorbert/adiosvm/tree/master/Tutorial/gray-scott

[7]https://github.com/mdorier/MandelbulbCatalystExample

[8]For simplicity in this paper we renumbered the iterations 1 to 30 rather than keeping the original iteration numbers. Original iteration numbers are shown in Figure 1a.



(a) Gray–Scott



(b) Mandelbulb

Fig. 3: Rendered results of simulations

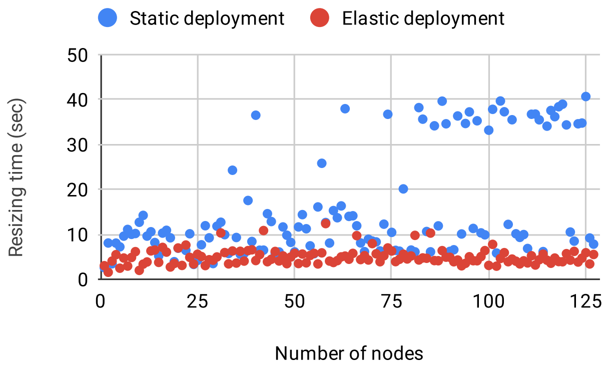


Fig. 4: Resizing times from a staging area of N processes to a staging area of N+1 processes, using either a static or an elastic deployment.

We measure how long it takes for the staging area to go from n to $n + 1$ nodes, using two strategies: (1) in a static deployment we kill the staging area and fully restart it on a new number of nodes, measuring the time between the kill signal and the moment the new staging area is ready to accept client requests; (2) in an elastic deployment we add a new node without shutting down the existing ones and rely on SSG to propagate the membership information, measuring the time between the `srun` command that deployed the new node and the moment the membership information has fully propagated. In both cases we wait 60 seconds between each rescaling to make sure the staging are has reached a stable state.

Figure 4 shows the result of this experiment. The elastic deployment offers a stable resizing time averaging 5 seconds, while the static deployment has a larger and more unpredictable resizing time, ranging from 5 seconds to more than 40 seconds and averaging 16 seconds.

C. Overhead of the MoNA communication layer

We now evaluate whether replacing MPI with our own communication layer causes any performance degradation on the visualization pipeline. We first use benchmarks to compare

[9]In practice we allocate a job with the maximum number of nodes upfront.

Message size	Cray-mpich	OpenMPI	MoNA	NA
8 bytes	1.163	1.527	1.924	2.103
128 bytes	1.215	1.608	1.985	2.122
2 KiB	1.709	2.12	2.714	2.766
16 KiB	5.247	61.451	14.087	-
32 KiB	6.773	59.279	15.305	-
512 KiB	56.371	109.472	72.69	-

TABLE I: Time (in milliseconds) to complete 1000 send/recv operations using Cray-mpich, OpenMPI, MoNA, and NA.

Message size	Cray-mpich	OpenMPI	MoNA
8 B	93.7	204.8	225.1
128 B	90.7	229.9	228.8
2 KiB	92.3	816.3	250.9
16 KiB	79.2	54253.9	304.0
32 KiB	122.8	219104.5	527.9

TABLE II: Time (in milliseconds) to complete 1000 binary-xor reduce operations using Cray-mpich, OpenMPI, and MoNA.

MoNA's performance against MPI. We then use the three applications described in Section III-A with their respective VTK-based pipelines.

1) Communication benchmarks

As a preliminary experiment, we use benchmarks to compare MPI and MoNA's point-to-point and collective performance. For MPI, we use Cray-mpich and OpenMPI, both available as modules on Cori by default. Table I shows the performance of point-to-point communications as a function of the message size. While Cray-mpich generally outperforms all other libraries, MoNA outperforms OpenMPI for large messages (16 KiB and more), probably thanks to its switching to RDMA rather than a rendez-vous protocol. For small messages, we also provide the performance of NA. Despite being built on top of NA, MoNA outperforms it by caching and reusing requests and message buffers, avoiding many small allocations in the process.

Next, we benchmark the performance of these libraries for a *reduce* operation. We chose this operation as an example of collective because it is at the core of the image-compositing algorithm used for parallel rendering. We run the reduce operation on 32 nodes using 16 ranks per node, for a total of 512 processes. Table II shows that once again Cray-mpich outperforms MoNA and OpenMPI. This time however, OpenMPI's performance rapidly degrades to become $1800\times$ slower than Cray-mpich, while MoNA is "only" $4.3\times$ slower.

These results show that while MoNA does not outperform a vendor-optimized MPI implementation like Cray-mpich (we would be surprised if it did), which directly relies on the uGNI library rather than going through several software layers like MoNA and OpenMPI do, it is on par with an open-source MPI implementation like OpenMPI, even outperforming it with naive algorithms. Indeed, MoNA's reduce algorithm is a simple binary-tree-based reduction. We expect that implementing more optimized collectives in MoNA, along with using AVX2 instructions to speed up processing, could further improve its performance. The performance of Cray-mpich shows how much we could gain by having at our disposal a vendor-optimized, elastic MPI implementation, instead of needing to rely on MoNA.

In the following sections, comparisons will only be made against Cray-mpich, since it is the best-performing MPI implementation on Cori.

2) Mandelbulb

We set up the Mandelbulb application to run on up to 512 processes spread across up to 16 nodes (using 32 processes per node). Each client process generates 4 blocks of data, of size $128\times128\times128$ integers (8 MB per blocks). We run Colza such that each Colza node runs 4 Colza processes, and each Colza process serves 4 client processes,[10] leading to a staging area spanning 4 to 128 processes, for a weak-scaling evaluation scenario (the amount of data to process is proportional to the number of Colza servers). We use a Python-based VTK pipeline that we execute for 6 iterations of the application. This pipeline may use either MPI or MoNA as communication layer. We discard the timing of the first iteration in our measurements: this timing is indeed highly variable and significantly larger than that of subsequent iterations because of VTK loading dynamic libraries and initializing a Python interpreter. We compute the average of the next 5 iterations. Figure 5 shows the resulting average pipeline execution time. From these results, we can see that *despite performing worse than MPI in benchmarks, our MoNA-based communication layer does not introduce any significant overhead when used in a real analysis pipeline.*[11] This is due to the fact that our pipeline is computation-intensive rather than communication-intensive. In situ visualization, which represents a subset of in situ analysis, has this advantage that its algorithms are generally embarrassingly parallel, requiring communication only for a final image-compositing step. More communication-intensive algorithms (e.g. Voronoi tessellation) might struggle more with MoNA at present.

3) Gray–Scott

We ran the Gray–Scott application with 512 processes on 16 nodes and a full domain size of 2 GB for each iteration. Every process generates a fixed portion of the whole data evenly. The staging area is deployed on 4 to 128 processes (1 to 16 nodes) to provide a strong-scaling use-case (fixed total amount of data for a varying number of processes). Figure 6 shows the average resulting pipeline execution time for 5 iterations. For this application as well, we notice little difference in execution time between MoNA and MPI.

4) Deep Water Impact

We run the DWI proxy on 2 client nodes, with 16 client processes per node. Since each iteration in the dataset consists of 512 VTU files, each process reads 16 files per iteration and sends the data to Colza via its Python interface. Contrary to Mandelbulb and Gray–Scott, which produce the same amount

[10]Because we cannot add processes dynamically when using MPI, in both MoNA and MPI scenarios we redeploy the entire application and staging area at the appropriate scale between each experiment.

[11]Other experiments, not presented here due to space constraints, led to the same conclusion at different scales, with different block sizes, and with a different number of blocks per server.

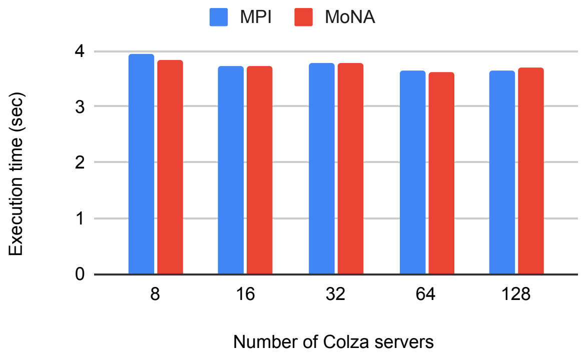


Fig. 5: Execution time of the pipeline for the Mandelbulb application using the MPI and MoNA communication layers at various scales, with a block size of 8 MB and a number of blocks proportional to the staging area size (weak scaling, hence the somewhat equal execution time at all scales).

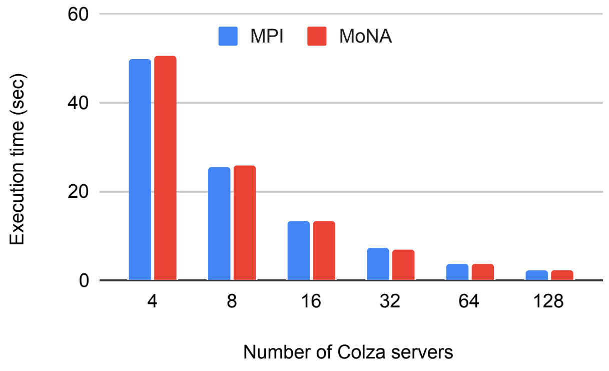


Fig. 6: Execution time of the pipeline for the Gray–Scott application using either MPI or MoNA as communication layer at various scales, with a fixed total data size of 2 GB (strong scaling). Note: weak scaling results lead to the same conclusion.

of data at every iteration, the rendering payload in DWI increases at every iteration. Hence in Figure 7 we show the rendering time as a function of the iteration number, with both MPI and MoNA, and with 8, 16, 32, and 64 Colza processes (1, 2, 4, and 8 nodes respectively).

The results show that the performance of MoNA is again on par with and MPI-based rendering pipeline, even outperforming it for some iterations at a scale of 8 and 16 processes.[12]

D. Comparison with Damaris and DataSpaces

To show that Colza can rival state-of-the-art in situ analysis frameworks, we ran the Mandelbulb miniapp with Damaris [19] and DataSpaces [20]. Just like Colza, Damaris' ability to be extended through plugins made it easy to reuse the same rendering pipeline in both cases. DataSpaces was recently refactored to make use of Margo (via its C++ interface Thallium), making it close to Colza in terms of its dependencies. We use 32 nodes in total: 64 client processes across 16 nodes, and 64 analysis servers for Colza/Damaris/DataSpaces

[12]This result is reproducible and while we haven't been able to identify why this happens, we suspect that MoNA's use of shared memory when processes are on the same node gives it an advantage at small scales.

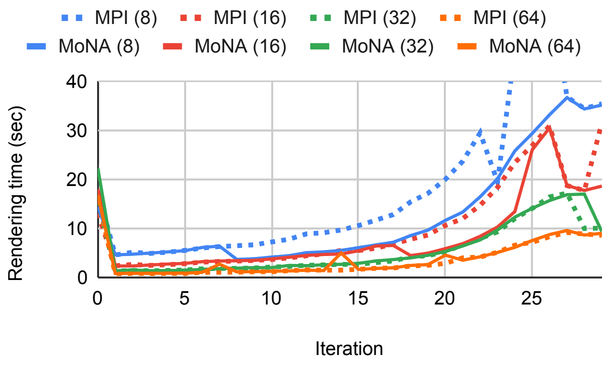


Fig. 7: Execution time of the pipeline for the Deep Water Impact proxy application using either MPI or MoNA as communication layer at various scales. The graph was truncated for readability reasons. At iteration 25 and 26 with 8 processes, the MPI-based pipeline's rendering time is around 60 seconds.

across the remaining 16 nodes. We use a block size of $64 \times 64 \times 64$ integers (1 MB per blocks) and 32 blocks per client process.

Figure 8 shows that Colza outperforms Damaris, both with MPI and MoNA. In Damaris' defence, this difference in run time could be attributed to the fact that the analysis plugin in Damaris is triggered independently by all the clients, hence if a client calls `damaris_signal` earlier than other clients, its corresponding server will enter the plugin earlier and will have to wait for other Damaris servers to become ready as well. In Colza, the `execute` call is triggered by a single client process, then coordinated across servers.

DataSpaces, which uses the same MPI-based pipeline as Colza+MPI, outperforms Colza when Colza uses MoNA, but does not when it uses MPI.

This experiment shows that Colza rivals with Damaris and DataSpaces in a static scenario, while also bringing elasticity as an option. In fact, Colza overcomes several other limitations linked to Damaris' reliance on MPI.

- Damaris splits `MPI_COMM_WORLD` to dedicate some ranks to data processing. This requires changes to the application so that it no longer relies on this communicator. Using Colza, the application's use of MPI remains unchanged.
- Damaris must be deployed at the same time as the application, while Colza, can be deployed and shut down when needed, potentially in a separate job.
- Damaris imposes that the number of dedicated processes divides the number of client processes. Colza allows any number of server processes for any number of clients.
- Deploying Damaris and its clients as a single application means that they must use the same launcher parameters (processes per node, binding to cores, threading, etc.).

DataSpaces does not have the above drawbacks and could more easily take advantage of our work, in particular thanks to its use of Mochi and Mochi's emphasize on software composability.

E. Elasticity in practice with Colza

So far we have considered the group membership and the

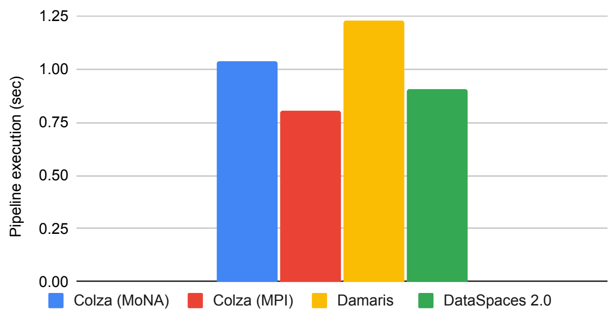


Fig. 8: Pipeline execution time for the Mandelbulb application when using it in Colza (with either an MPI or MoNA communication layer), as a Damaris plugin (set in "dedicated nodes" mode), or in DataSpaces.

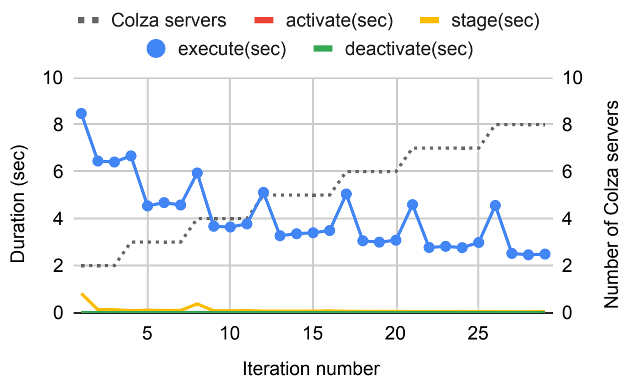


Fig. 9: Exercising elasticity with the Mandelbulb application and Colza resized from 2 to 8 nodes.

communications aspects of Colza separately. In this section we demonstrate Colza' ability to change its number of processes dynamically. We do so with the Mandelbulb and DWI applications.

1) Mandelbulb

We execute the Mandelbulb application on 16 nodes, using 16 processes per node. Each process generates a single $128 \times 128 \times 64$-element block, for a total of 256 blocks (1 GB of data in total). Colza is initially deployed on 2 nodes, using 1 process per node. Every 60 seconds, we add a new Colza node, up to a total of 8 nodes. We measure the duration of `activate`, `stage`, `execute`, and `deactivate` calls for each iteration, and monitor the number of Colza nodes in use as the application progresses.

The results, shown in Figure 9, illustrate the decrease in pipeline execution time as the staging area is resized to larger scales. When a new node is added, we observe an increase of execution time for that iteration, caused by the need for the new process to initialize VTK. The `activate`, `stage`, and `deactivate` calls represent a negligible portion of run time, respectively lasting an average of 4 milliseconds, 100 milliseconds, and 0.6 milliseconds.

Note that in a real application, only `activate`, `stage`, and `deactivate` calls would represent an overhead for the application. Since the purpose of a staging area is to perform analysis in the background, while the application continues

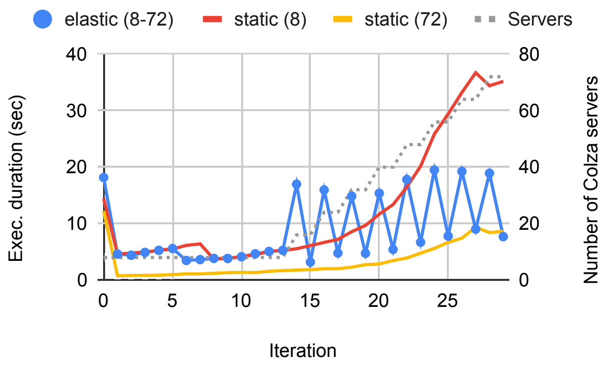


Fig. 10: Execution time of the rendering pipeline for the Deep Water Impact proxy application with Colza being resized from 1 to 9 nodes (8 to 72 processes) every other iteration after iteration 13, and compared with a static deployment of 8 or 72 processes.

running, the non-blocking version of `execute` would be used in practice.

2) Deep Water Impact

We run the DWI proxy application in the same manner as in Section III-C4. We initially use a Colza staging area spanning a single node with 8 processes. Starting from iteration 13, we add 8 new Colza processes (1 extra node) at every other iteration, up to using 72 processes towards the end of the run. Figure 10 shows the performance of the rendering pipeline as the application progresses. We compare this rendering time against cases in which we maintain 8 and 72 Colza processes throughout the run. Once again we find an overhead every time new processes are added, as these processes need to initialize their copy of the pipeline. However we also see that resizing Colza enables keeping the rendering time bounded – to 20 seconds if we include the resizing overhead (in practice, resizing would however not be done so frequently or with such small increments), and to 10 seconds without this overhead– while the static pipeline's rendering time keeps increasing. *This experiment demonstrates the practical benefit of elasticity for applications that have irregular data sizes and complexity.*

IV. Discussion

In this section we discuss relevant aspects of our work that are outside the scope of the evaluation done in this paper and present opportunities for future research.

A. Integration with job schedulers

In this paper, we allocated the maximum number of nodes needed upfront for each experiment because the job scheduler could not resize our job. However job schedulers are starting to provide resizing capabilities: SLURM allows us to reduce the size of a running job using `scontrol update` with the `NumNodes` option,[13] but doesn't let us increase it. LFS provides the `bresize` command[14] to increase and decrease job sizes.

[13]https://slurm.schedmd.com/scontrol.html

[14]https://www.ibm.com/docs/en/spectrum-lsf/10.1.0?topic=reference-bresize

We could envision enhancing job schedulers with information that would help them make allocation decisions. For instance, adding more resources to an existing job could have priority over scheduling new jobs. The fact that an application is elastic could also be communicated to the job scheduler, giving it the opportunity to scale the job up and down depending on resources available.

B. Why triggering elasticity

Some reasons for triggering a change in resources were listed by Dorier et al. [7].

In the context of applications that exhibit changes in complexity over time (such as Deep Water Impact), elasticity could be used to try overlapping as best as possible the duration of an iteration in the simulation side and in the analysis side. In Figure 10 for example, enabling elasticity allows the analysis time to remain under 10 seconds (with the exception of iterations that have to initialize new servers). Such a trigger would be application-driven.

Other triggers include the user manually changing the amount of resources to setup new analysis pipelines, or as mentioned above, the job scheduler adding or reclaiming nodes to optimize the platform's overall resource usage.

If the simulation was itself elastic, elasticity could take the form of an exchange of resources between the simulation and the analysis code, without involving the job scheduler.

V. Related Work

As part of a more general self-adapting in situ framework, Jin et al. [21] presented an evaluation of the advantage of dynamically changing the number of cores used by in-transit processing. However their experiment used a fixed-size MPI application that split `MPI_COMM_WORLD` into a simulation group, an in-situ group, and an in-transit group of processes. They worked around the limitation of MPI for their evaluation by relying on fewer cores when necessary, but these cores remained allocated and part of the application, and they could not add more cores later on beyond the ones that were originally present within `MPI_COMM_WORLD`.[15] Their findings however supports the potential benefits of elasticity for in situ analysis.

Duan et al. [22] presented CoREC, a process recovery solution that cooperates with a data resiliency scheme. This solution aims to recover failed staging servers so as to maintain the performance of the data staging framework over the lifetime of the workflow. In CoREC, however, the staging area serves only as in-memory data storage bridging simulations and analysis applications. Analysis tasks do not run within the staging area, making it easier for the staging area to be resized.

Kress et al. [23] presented models to evaluate the cost efficiency of in-transit visualization. Their works highlight the difficulties of configuring in situ and in transit analysis to achieve the best performance, and motivates the need for more dynamic approaches that would allow finding optimal configurations at run time.

With regard to redesigning the core algorithms used by in situ analysis and visualization, Wang et al. [24] proposed an in situ framework based on the MapReduce paradigm [25]. This solution, inspired by cloud computing, would be much better suited to elasticity than the algorithms currently used in ParaView. It falls in the category of works that attempt at redesigning algorithms, rather than relying on existing, well-established tools.

In this work we used ParaView, which abstracts its communications and relies on libraries that have abstracted their communications sufficiently that we could replace MPI. Fortunately, abstraction at this level is common practice. The DIY library [26] (which is also used internally by ParaView) has a `communicator` class wrapping MPI. The Damaris [27] middleware has an abstract RPC layer (the `Reactor`) with implementations using MPI 2, MPI 3, or shared memory. VisIt/LibSim [2] is much more tightly coupled to MPI. Although its `avtParallelContext` class internally provides some level of abstraction, the only API function allowing controls over the communication layer is `VisItSetMPICommunicator`, which allows changing the communicator. This shows that, with some work, other production analysis and visualization packages could evolve beyond MPI.

ADIOS2 [28] is a well-established interface to bridge simulations with analysis code. Its SST engine [29], [30] enables connecting data producers with consumers, with ADIOS taking care of data redistribution via RDMA, enabling streaming-like coupling of HPC applications. Looking at the ADIOS2 code, we find that while the SST engine depends on a `Comm` communicator class, this class is abstract, with a concrete implementation relying on MPI. Hence by injecting MoNA into ADIOS2, the work presented in this paper could be adapted to work within the ADIOS2 interface as well.

While MPI has been evaluated in the past for use in HPC data services [31], [32], we have yet to see data services that rely on it in production. Service developers nowadays tend to move beyond MPI [33] and rely either on lower-level networking libraries such as Libfabric [34] or UCX [35] or on RPC frameworks such as Mercury and other libraries such as those provided by Mochi. The Mochi components are also used in a number of HPC data services including Intel's DAOS [36], UnifyFS [37], and GekkoFS [38].[16] Our choice of relying on Mochi for our Colza data-staging service falls within this trend.

Mochi was also used by Cheriere et al. [39] to implement an elastic data storage service for HPC. They focus on the cost of migrating data when upscaling and downscaling, and on load rebalancing. Resizing our staging area does not require data migration. However their work could be combined with ours, thanks to Mochi's component-based development methodology, to ultimately design an elastic service providing both storage and in situ analysis capabilities.

[15]This technical detail is not described in their paper but was confirmed by one of their authors.

[16]https://wordpress.cels.anl.gov/mochi/projects-using-mochi/

With respect to current limitations of MPI, we note that the MPI 4.0 draft standard [40] provides promising enhancements that could help us in the future. MPI sessions [41], in particular, would let us create an MPI context within a non-MPI application, which would allow us to run unmodified MPI-based ParaView/Catalyst pipelines by creating an MPI session for each `execute` phase. Our experiments in Section III show how Colza would be improved by relying on MPI rather than MoNA.

In the fault-tolerance space, User Level Failure Mitigation (ULFM) [42], [43] presents lightweight MPI extensions to detect communicator failure, along with solutions to recovery from the failure. Gamell et al. [44], [45] presented a framework based on ULFM to show how communicators can shrink to remove failed processes, and the new process can be merged into the new communication group. Similarly, Rizzi et al. [46] present a solution to use ULFM to detect process crashes for partial differential equation calculation. We hope that in the future, such extensions to MPI will be available more broadly.

An alternative to replacing MPI would be to rely on AMPI [47], which decouples MPI ranks from the processes they run on by assigning them to user-level threads. In such a design, the number of MPI ranks used by ParaView/Catalyst would remain fixed, while their mapping to actual resources would change over time.

Fox et al. [48] presented E-HPC, a batch scheduler that enables elasticity in HPC workflows. Elasticity is managed by checkpointing the application, shutting it down, and transparently restarting it on a different number of processes, in contrast to our work, which aims not to shut down the application. Similarly Raveendran et al. [49] discuss how to adapt MPI-based applications for a cloud environment such as AWS. They also adopt checkpointing and data redistribution mechanisms to enable rescaling.

VI. Conclusion

Our work developing and evaluating Colza, an elastic in situ analysis framework, showed the feasibility of implementing elasticity using production analysis tools. We were successful in enabling elasticity in ParaView by injecting a custom communication layer to replace MPI, and via tricks to enable changing ParaView's underlying communicator when new processes join. We showed that Colza achieves performance on par with existing in situ frameworks that rely on MPI, despite some overhead from its MoNA communication layer when compared with a vendor-provided, well-optimized MPI implementation. Our results, in particular with the Deep Water Impact dataset, shows the potential benefit of elasticity in a real-life scenario, where the data complexity increases over time. We hope that this work will encourage other researchers to explore elasticity within other analysis frameworks, and will motivate the MPI community in providing concrete solutions in the MPI standard.

In the future, we plan to (1) make our framework capable of handling process crashes, effectively enabling fault tolerance with *unexpected/unplanned* resizing; (2) enable automatic resizing as a response to performance constraints or optimization targets; and (3) enable state-full pipelines, for which shutting down a process requires data migration.

Acknowledgment

We thank Kenneth Moreland and Andy Bauer for helping us understand ParaView, VTK, and IceT through Kitware's Discourse. We thank John Patchett for providing the Deep Water Impact dataset. Finally we thank Gail Pieper for proofreading and editing this paper. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

References

[1] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "ParaView Catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ACM, 2015, pp. 25–29.

[2] T. Kuhlen, R. Pajarola, and K. Zhou, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the 11th Eurographics Conference on Parallel Graphics and Visualization (EGPGV)*, 2011.

[3] H. Childs *et al.*, "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020. [Online]. Available: https://doi.org/10.1177/1094342020935991

[4] R. Imahorn, I. B. Rojo, and T. Günther, "Visualization and analysis of deep water asteroid impacts," in *2018 IEEE Scientific Visualization Conference (SciVis)*, 2018, pp. 85–96.

[5] T. Peterka, D. Bard, J. Bennett, E. Bethel, R. Oldfield, L. Pouchard, C. Sweeney, and M. Wolf, "ASCR workshop on in situ data management: Enabling scientific discovery from diverse data sources," USDOE Office of Science (SC)(United States), Tech. Rep., 2019.

[6] T. Peterka, D. Bard, J. C. Bennett, E. W. Bethel, R. A. Oldfield, L. Pouchard, C. Sweeney, and M. Wolf, "Priority research directions for in situ data management: Enabling scientific discovery from diverse data sources," *The International Journal of High Performance Computing Applications*, vol. 34, no. 4, pp. 409–427, 2020.

[7] M. Dorier, O. Yildiz, T. Peterka, and R. Ross, "The challenges of elastic in situ analysis and visualization," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV '19. New York, NY, USA: ACM, 2019, workshop, pp. 23–28. [Online]. Available: http://doi.acm.org/10.1145/3364228.3364234

[8] R. B. Ross *et al.*, "Mochi: Composing data services for high-performance computing environments," *Journal of Computer Science and Technology*, vol. 35, pp. 121–144, 2020. [Online]. Available: https://link.springer.com/article/10.1007/s11390-020-9802-0

[9] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2000, pp. 346–353.

[10] W. Bland, H. Lu, S. Seo, and P. Balaji, "Lessons learned implementing user-level failure mitigation in MPICH," in *2015 15th IEEE/ACM international symposium on cluster, cloud and grid computing*. IEEE, 2015, pp. 1123–1126.

[11] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[12] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2013, pp. 1–8.

[13] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable weakly-consistent infection-style process group membership protocol," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 2002, pp. 303–312.

[14] R. H. Castain, J. Hursey, A. Bouteiller, and D. Solt, "Pmix: process management for exascale environments," *Parallel Computing*, vol. 79, pp. 9–29, 2018.

[15] A. Doelman, T. J. Kaper, and P. A. Zegeling, "Pattern formation in the one-dimensional Gray–Scott model," *Nonlinearity*, vol. 10, no. 2, p. 523, 1997.

[16] J. Aron, "The Mandelbulb: first "true" 3d image of famous fractal," *New Scientist*, vol. 204, no. 3736, pp. 54–55, 2009.

[17] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, "Flexible IO and integration for scientific codes through the adaptable IO system (adios)," in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[18] J. Patchett and G. Gisler, "Deep water impact ensemble data set," Tech. Rep., 2017, lA-UR-17-21595. [Online]. Available: http://datascience.dsscale.org/wp-content/uploads/2017/08/DeepWaterImpactEnsembleDataSet_Revision1.pdf

[19] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 2013, pp. 67–75.

[20] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2012.

[21] T. Jin, F. Zhang, Q. Sun, M. Romanus, H. Bui, and M. Parashar, "Towards autonomic data management for staging-based coupled scientific workflows," *Journal of Parallel and Distributed Computing*, vol. 146, pp. 35–51, 2020.

[22] S. Duan, P. Subedi, P. Davis, K. Teranishi, H. Kolla, M. Gamell, and M. Parashar, "CoREC: Scalable and resilient in-memory data staging for in-situ workflows," *ACM Transactions on Parallel Computing (TOPC)*, vol. 7, no. 2, pp. 1–29, 2020.

[23] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire, "Opportunities for cost savings with in-transit visualization," in *High Performance Computing*, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds. Cham: Springer International Publishing, 2020, pp. 146–165.

[24] Y. Wang, G. Agrawal, T. Bicer, and W. Jiang, "Smart: A MapReduce-like framework for in-situ scientific analytics," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[25] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[26] D. Morozov and T. Peterka, "Block-parallel data analysis with DIY2," in *Proceedings of the 2016 IEEE Large Data Analysis and Visualization Symposium LDAV'16*, Baltimore, MD, 2016.

[27] M. Dorier, G. Antoniu, F. Cappello, M. Snir, R. Sisneros, O. Yildiz, S. Ibrahim, T. Peterka, and L. Orf, "Damaris: Addressing performance variability in data management for post-petascale simulations," *ACM Transactions on Parallel Computing (TOPC)*, vol. 3, no. 3, p. 15, 2016.

[28] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020.

[29] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "Adios 2: The adaptable input output system. a framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302560

[30] J. Logan, M. Ainsworth, C. Atkins, J. Chen, J. Y. Choi, J. Gu, J. M. Kress, G. Eisenhauer, B. Geveci, W. Godoy, M. B. Kim, T. Kurc, Q. Liu, K. V. Mehta, G. Ostrouchov, N. Podhorszki, D. Pugmire, E. D. Suchyta, N. Thompson, O. Tugluk, L. Wan, R. Wang, B. Whitney, M. D. Wolf, K. Wu, and S. A. Klasky, "Extending the publish/subscribe abstraction for high-performance i/o and data management at extreme scale," *Bulletin of the IEEE Technical Committee on Data Engineering*, vol. 43, no. 1, 3 2020. [Online]. Available: https://www.osti.gov/biblio/1606642

[31] R. Latham, R. Ross, and R. Thakur, "Can MPI be used for persistent parallel services?" in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2006, pp. 275–284.

[32] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using MPI in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 43–48.

[33] F. Liu, C. Barthels, S. Blanas, H. Kimura, and G. Swart, "Beyond MPI: New communication interfaces for database systems and data-intensive applications," *ACM SIGMOD Record*, vol. 49, no. 4, pp. 12–17, 2021.

[34] O. W. Group *et al.*, "Libfabric," https://www.openfabrics.org/.

[35] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: an open source framework for HPC network APIs and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[36] M. Hennecke, "DAOS: A scale-out high performance storage stack for storage class memory," *Supercomputing Frontiers*, p. 40, 2020.

[37] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror, D. Ivanov, T. Wang, C. P. Steffen, and U. N. N. S. Administration, "UnifyFS: A distributed burst buffer file system – 0.1.0," 10 2017. [Online]. Available: https://www.osti.gov/biblio/1408515

[38] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS – a temporary distributed file system for HPC applications," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 319–324.

[39] N. Cheriere, M. Dorier, G. Antoniu, S. M. Wild, S. Leyffer, and R. Ross, "Pufferscale: Rescaling HPC data services for high energy physics applications," in *Proceedings of the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (Ccgrid)*, ser. CCgrid '20. IEEE/ACM, 2020, conference. [Online]. Available: https://ieeexplore.ieee.org/document/9139614

[40] "MPI 4.0rc 2020 draft specification," https://www.mpi-forum.org/docs/, accessed: 2021-03-24.

[41] N. Hjelm, H. Pritchard, S. K. Gutiérrez, D. J. Holmes, R. Castain, and A. Skjellum, "MPI Sessions: Evaluation of an Implementation in Open MPI," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, 2019, pp. 1–11.

[42] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra, "Post-failure recovery of mpi communication capability: Design and rationale," *The International Journal of High Performance Computing Applications*, vol. 27, no. 3, pp. 244–254, 2013.

[43] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in mpi," in *European MPI Users' Group Meeting*. Springer, 2012, pp. 193–203.

[44] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 895–906.

[45] R. Van Der Wijngaart, M. Gamell, K. Teranishi, E. Valenzuela, M. A. Heroux, and M. Parashaar, "Fenix a portable flexible fault tolerance programming framework for mpi applications." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2016.

[46] F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, B. Debusschere, O. LeMaitre, and O. Knio, "Ulfm-mpi implementation of a resilient task-based partial differential equations preconditioner," in *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, 2016, pp. 19–26.

[47] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive MPI," in *International workshop on languages and compilers for parallel computing*. Springer, 2003, pp. 306–322.

[48] W. Fox, D. Ghoshal, A. Souza, G. P. Rodrigo, and L. Ramakrishnan, "E-hpc: a library for elastic resource management in hpc environments," in *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science*, 2017, pp. 1–11.

[49] A. Raveendran, T. Bicer, and G. Agrawal, "A framework for elastic execution of existing mpi programs," in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. IEEE, 2011, pp. 940–947.