LowFive: In Situ Data Transport for High-Performance Workflows

Tom Peterka, Dmitriy Morozov, Arnur Nigmetov, Orcun Yildiz, Bogdan Nicolae, Philip E. Davis

LowFive: In Situ Data Transport for High-Performance Workflows

Tom Peterka
Argonne National Laboratory
Lemont, IL USA
tpeterka@mcs.anl.gov

Dmitriy Morozov
Lawrence Berkeley National Laboratory
Berkeley, CA USA
dmorozov@lbl.gov

Arnur Nigmetov
Lawrence Berkeley National Laboratory
Berkeley, CA USA
anigmetov@lbl.gov

Orcun Yildiz
Argonne National Laboratory
Lemont, IL USA
oyildiz@anl.gov

Bogdan Nicolae
Argonne National Laboratory
Lemont, IL USA
bnicolae@anl.gov

Philip E. Davis
University of Utah
Salt Lake City, UT USA
philip.davis@sci.utah.edu

***Abstract*—We describe LowFive, a new data transport layer based on the HDF5 data model, for in situ workflows. Executables using LowFive can communicate in situ (using in-memory data and MPI message passing), reading and writing traditional HDF5 files to physical storage, and combining the two modes. Minimal and often no source-code modification is needed for programs that already use HDF5. LowFive maintains deep copies or shallow references of datasets, configurable by the user. More than one task can produce (write) data, and more than one task can consume (read) data, accommodating fan-in and fan-out in the workflow task graph. LowFive supports data redistribution from n producer processes to m consumer processes. We demonstrate the above features in a series of experiments featuring both synthetic benchmarks as well as a representative use case from a scientific workflow, and we also compare with other data transport solutions in the literature.**

***Index Terms*—workflow, data model, data transport, in situ**

I. Introduction

The increasing disparity between supercomputers' floating-point operations rate and I/O bandwidth, along with the need to process data more frequently than the temporal resolution of checkpoints in storage, motivate the use of in situ workflows.

An *in situ workflow* is a collection of programs executing concurrently in an HPC system and communicating through the memory and interconnect of the system instead of physical storage. We call these programs, which are separate executables that can be parallel and multiprocess, *tasks*. A typical example is a simulation coupled with data analysis, although in general multiple tasks may be coupled in an arbitrary directed graph topology. Tasks in an in situ workflow often also checkpoint data to physical storage, but they are not required to use physical storage as a communication mechanism.

Workflows link together multiple disparate tasks and data models. One challenge when using in situ workflows is how to describe the data to a workflow system in a way that is compatible with both the user's view of the data and with the workflow's usage of the data.

To bypass the parallel file system, data meant for storage have to be intercepted and redirected elsewhere. There are different places in the I/O stack where this can be done: from the POSIX I/O layer to inserting new data models into the simulation and analysis codes. All have their advantages and disadvantages.

At one end of the spectrum, the POSIX level would capture the widest range of I/O, at the expense of losing metadata annotation. Burst buffers backed by node-local storage or compute-node memory such as BurstFS [1] and GekkoFS [2] have been proposed by the storage community to bypass the parallel file system. A general-purpose burst buffer file system does not however fully address the in situ data transport problem faced by the workflow community, which is to be able to couple and redistribute complex data structures used by applications.

For example, consider an adaptive mesh refined (AMR) simulation that computes many datasets, spanning a dozen variables at different resolutions, coupled to an analysis task that consumes only a single variable at one resolution. With access to user-level metadata such as dataset names and associated data spaces, only the required dataset would need to be sent from the producer (simulation) to the consumer (analysis); furthermore within each MPI process of producer and consumer, only the subspace at the intersection of the producer and consumer subdomains would be transported. The other datasets not needed by the consumer would never actually have to be written, i.e., sent. The availability of high-level metadata, such as in HDF5, allows a data transport layer to move the minimum amount of data required to satisfy the intersection of producer and consumer access patterns; these metadata are unavailable at the POSIX level of offsets and bytes.

In contrast, a new data model custom tailored for distributed data transport would provide the most flexibility for in situ communication, at the expense of having to make considerable changes to existing codes. This solution has been proposed by the workflow community; e.g., Conduit [3] and Bredala [4] take this approach, and we will cover such methods and compare them in this paper.

Fortunately, there is a convenient middle ground between capturing low-level POSIX bytes and redefining entirely new data models. In a recent release (1.12.0), HDF5 introduced a Virtual Object Layer (VOL) that is called by all HDF5 operations and allows developers to supply plugins intercepting the underlying data. We elected to develop LowFive as an HDF5 VOL plugin, allowing us to take advantage of high-level metadata provided by HDF5 without changing codes' usage of its familiar API.

In this paper, we describe LowFive, a new data transport layer based on the HDF5 [5] data model, for in situ workflows. In selecting HDF5, we were looking for a data model that is widely used by HPC application developers as well as one that is broadly available in other areas of big data, machine learning, and artificial intelligence. Additionally, we sought a data model that is expressive in its metadata descriptions of data types, data spaces, and attributes, so that LowFive could take advantage of that rich metadata to facilitate efficient data transport. We also wanted a generic data model that was not tied to a particular workflow system. Arguably, HDF5 is one the most popular data models meeting our criteria. We developed LowFive as a standalone library not tied to particular workflow system, maximizing its potential applicability and benefit. The following are the key contributions of our research.

- LowFive is a novel HDF5 VOL plugin for distributed direct data transport between in situ workflow tasks.
- We support two data transport modes: tasks can communicate in situ (via MPI message passing), or by reading and writing traditional HDF5 files to physical storage.
- We provide the choice of deep or shallow copies at the granularity of individual datasets. In the former case, LowFive makes a copy of the dataset, and the user is free to modify their local copy without affecting the data transport. In the latter, LowFive creates only a reference to the user's data, implying that the user will not modify the dataset until LowFive no longer needs it (e.g., the consumer task has read the data).
- More than one task can produce (write) data, and more than one task can consume (read) data. In other words, the the workflow task graph can have fan-in and/or fan-out.
- When two or more tasks exchange data (e.g., a producer–consumer relationship), the producer and consumer do not need to have the same number of processes or use the same data decomposition. LowFive enables data redistribution from n producer processes to m consumer processes, supporting full generality of HDF5 data spaces.

We demonstrate the above features in a series of experiments featuring both synthetic benchmarks as well as a representative use case from an actual science application. In the synthetic benchmarks, we also compare with other data transport solutions in the literature.

The remainder of this paper is organized as follows. Section II reviews the literature pertaining to workflow data models and transport layers, as well as other HDF5 VOL plugins. Section III explains the design and implementation of LowFive. Section IV presents experimental results with synthetic benchmarks and a real application. We conclude in Section V with a summary and a look toward the future.

II. Related work

Three broad categories of published literature are related to our research. Within a workflow system, users specify data on which to operate using a data model, and we survey the most commonly used data models in workflow systems today. Workflow systems also transport data, specified using the data model, between tasks of the workflow, and we cover some of the popular data transport layers. Because LowFive is implemented as a VOL plugin, we also identify other uses of HDF5 VOL plugins.

A. Workflow Data Models

Workflow systems can repurpose existing data models, e.g., from storage libraries such as HDF5 or from visualization libraries such as VTK, or they can define a new data model for their computing and communication. Re-using existing data models has the potential advantage of minimizing required changes to users' codes when migrating from standalone execution to execution within a workflow system. For example, a simulation code that writes checkpoints in HDF5 potentially requires fewer changes when embedded in a workflow system that uses an HDF5 data model, as compared with a workflow that requires users to describe data in a new model. This is one reason why we selected HDF5 as the data model for LowFive.

The data model used in the Decaf [6] workflow system is called Bredala [4]. Bredala is designed to annotate fields in a data model such that they can be redistributed by a workflow system. In Bredala, data intended to be moved among tasks are first appended to a "container" (essentially serialized into a message) one field at a time, along with annotations indicating how each field is handled during data redistribution. Annotating data members and appending them to a message is done through Bredala API calls in the user's task code.

The Henson [7] workflow system is designed for shared-memory communication among tasks colocated on the same computing node. Data are accessed in user tasks using pointers to shared data. When tasks are containerized, Dhmem [8] allows shared-memory pointer access between Henson workflow tasks in separate containers, with minimal code change and low performance overhead.

A data model is specified in the ADIOS [9] workflow system by means of an external XML file, where structured N-dimensional arrays consist of various primitive types. Attributes relating to global shape of the N-dimensional array as well as the local shape (starting offsets, sizes) of the subarray contained on a local process are included in the XML definition. These metadata are used for data redistribution between a different number of producer and consumer processes. ADIOS contains several different back-ends for physical file storage as well as for in situ communication.

Conduit [3] is another open-source hierarchical data model. Unlike HDF5, it is not tied to a particular file format, nor is it restricted to physical file storage, making it another candidate for in situ workflows. The data model is described hierarchically in YAML or JSON format. Conduit is generic, being able to represent virtually any data. In order to ease application development for typical scientific applications, the Blueprint [10] layer atop Conduit provides data models for commonly used meshes and scalar- and vector-valued multidimensional arrays.

For workflows designed to perform visualization tasks, the VTK data model is commonly used. VisIt Libsim [11], ParaView Catalyst [12], and SENSEI [13] are examples of such systems based on VTK.

B. Workflow Data Transport and Redistribution

In addition to executing tasks that perform computations on data, workflows move data among tasks. The data transport mechanism varies among workflow systems and among locations of communicating tasks; communication via files in storage systems, shared memory, and network communication are the most common mechanisms. In the latter case, data are distributed among separate computing nodes, and can communicate via sockets, low-level network protocols for remote direct memory access (RDMA), or message passing via MPI. In the distributed case, it is also likely for two communicating tasks (e.g., producer and consumer) to have different numbers of processes and different data decompositions, raising the question of how data are redistributed between producer and consumer task while preserving data model semantics.

In situ, data can be moved in a transport layer by either direct messaging between the user tasks or by staging data in an intermediate location. A data staging service is launched on separate compute nodes than the user tasks, whereas direct messaging moves data from one task to another (e.g., using MPI) with no intermediary agent, service, or resources. Data staging is also sometimes called *in transit* or *loosely coupled* in the literature. Direct messaging, the approach taken in LowFive, is conceptually a simpler solution, requiring no additional resources, but it can incur synchronization between tasks; while data staging decouples tasks through the staging area. Gainaru et al. present a thorough analysis of the uses of data staging in a recent paper [14].

Bredala [4] is a direct messaging solution, where various fields in the data model are annotated with flags that allow the semantic integrity of the local subset of the data model to remain intact when data are redistributed (split and merged) between n producer processes and m consumer processes. For example, if the field is a counter, then Bredala understands that it should be adjusted by the difference between consumer and producer data items. The size of semantic items is also specified, so that for example, all three coordinates of a 3-d vector remain colocated during data movement and redistribution. Bredala supports several redistribution policies: round-robin, contiguous, and bounding box intersections. LowFive, in comparison, does not require additional data annotations to inform its redistribution.

DataSpaces [15] is a staging solution that provides a shared space consisting of a set of HPC computing nodes that act as a distributed staging server for client (producer and consumer) tasks. The data abstraction is an N-dimensional array of tuples that can be redistributed, colocating the components of a single tuple. Multiple datasets can be staged, as well as multiple versions (e.g., time steps) of each dataset. Producers and consumers interact with DataSpaces using a put-get API, where global and local shapes are specified, so that DataSpaces can locate and redistribute data. There are two key differences between DataSpaces and LowFive. LowFive has a broader range of supported data types and data spaces (the entire HDF5 data model), and DataSpaces requires a set of staging nodes to be allocated in a separate server application, whereas LowFive transports data directly from producer to consumer processes with no intermediate resource. The latest version of DataSpaces utilizes several Mochi [16] services for remote procedure calls (RPC) and user-level threading. DataSpaces is one of the back-end transport layers used in the ADIOS2 [17] system.

Conduit provides a data transport layer called Relay [18] that includes back-ends for physical file storage using HDF5 and ADIOS I/O libraries, MPI message passing, and web sockets. Depending on the back-end, Conduit Relay can implement direct messaging (e.g., through MPI) or staging (e.g., through ADIOS). To the best of our knowledge, Relay does not provide any in situ redistribution services equivalent to LowFive, Bredala, or DataSpaces. Therefore, we did not conduct performance comparisons with Conduit Relay, whereas we do compare the performance of LowFive with Bredala and DataSpaces.

C. Other HDF5 VOL Plugins

There is a small but growing number of other applications of HDF5 VOL plugins. Although these are used for improving physical storage access, not in situ workflows, it is still informative to know how others are using this feature of HDF5. Data Elevator [19] is a VOL plugin that transparently migrates and accesses data in various levels of the memory–storage hierarchy, from cache to DRAM to burst buffer to parallel file system. Another application [20] converts HDF5 data to the Apache Arrow columnar format for converting between HPC and Big Data storage formats. A third application [21] converts the HDF5 file format to a log-structured layout in order to improve parallel write performance. In all three of these cases, as in LowFive, the user is oblivious to the data transformations occurring in the VOL plugin while making what appear to be ordinary HDF5 I/O calls.

III. Approach

A. In-memory Metadata Hierarchy

LowFive builds in memory a replica of the HDF5 metadata hierarchy. An example of one such hierarchy is shown in Figure 1. Throughout much of the following description and

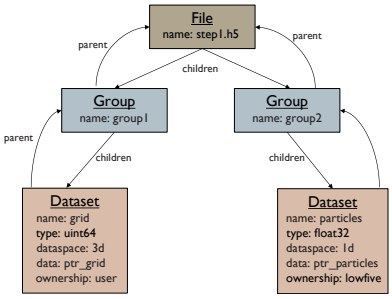

Fig. 1. In-memory metadata hierarchy.

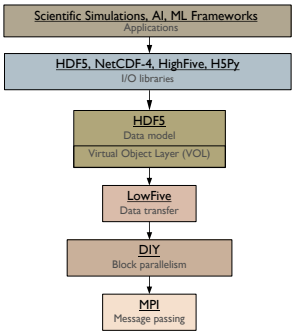

Fig. 2. Software stack.

experimental results, we use this example of one HDF5 file containing two groups (group1 and group2), and each group containing one dataset. Group1 contains a 3D structured grid of scalar values, while group2 contains an unordered list of 3D particles. A physical file may or may not reside on disk; in either case, we build the in-memory metadata hierarchy as shown in the figure.

The example hierarchy in Figure 1 has three levels: file, groups, and datasets, but in general we are able to replicate any complex tree that can be defined in HDF5. For example, tree nodes can contain attributes, use arbitrary data types, and utilize arbitrarily complex data spaces, among other HDF5 constructs. We call our tree a metadata hierarchy, although it may or may not contain deep copies of user data as well as metadata. Whether to copy data or store a shallow reference to them in the tree is configurable by the user at per-dataset granularity.

LowFive is implemented as an HDF5 Virtual Object Layer (VOL) plugin, a plugin system added in HDF5 1.12.0, that catches all HDF5 calls and allows custom code to be executed. Plugging into HDF5 allows LowFive to support not only codes that use the native HDF5 C API, but any library written on top of it. Examples of such libraries are the C++ HighFive library [22], the h5py Python interface [23], the NetCDF-4 bridge to HDF5 [24], and many other custom scientific and AI libraries that build on top of them, such as Scorpio [25], TensorFlow [26], and Keras [27].

The entire software stack is shown in Figure 2. LowFive depends on the DIY block parallel model [28] to perform efficient data redistribution, which in turn uses MPI, as does native HDF5. Although the HDF5 source code is written in C, we implemented LowFive in high-level C++17 at the three levels of abstraction described below. LowFive can be invoked by constructing one of the classes below explicitly in the user task code, or LowFive can be enabled by setting two environment variables. The latter mechanism is useful if changes to the code are to be minimized, including potentially no change to the user code.

a) Base VOL: The lowest level of our plugin is the base layer. Any HDF5 functions that are not redefined in the subsequent layers are caught at this base layer and pass through to native HDF5 file I/O.

b) Metadata VOL: We derive the metadata VOL class from the base VOL class. Here we redefine most of the functions in the base layer with their in-memory metadata counterparts. The implementation is entirely our own: we manage our own tree of HDF5 objects (files, groups, datasets, attributes, etc.) that replicates the user's HDF5 data model. We create new nodes in our tree when the user creates HDF5 objects such as datasets, traverse the tree and locate existing objects when the user opens existing HDF5 objects, and so forth. We manage our own pointers to tree nodes that are the equivalent of HDF5 identifiers. Because the HDF5 native data representation is opaque, we opted to create our own hierarchy mirroring HDF5 metadata rather than attempting to reach into whatever internal representation HDF5 uses. The only exceptions to this are HDF5 data types and data spaces. We use HDF5's internal facilities for their manipulation and serialization, which allows us to support these features in their full generality. (HDF5 data spaces in particular can express complex geometric shapes, so relying on HDF5's internal routines helps ensure correctness.)

c) Distributed Metadata VOL: The highest level in our class organization is the distributed metadata VOL class, which derives from metadata VOL. Here we redefine HDF5 functions that potentially access remote processes, e.g., in order to transfer data over MPI from the processes of a producer task to the processes of a consumer task. This is how we redistribute data as described below and in the experimental results that follow. We implement distributed client-server connections between the processes of a consumer task reading data and a producer task writing data. While we use the terms "writing" and "reading," the distributed metadata VOL class executes these functions by sending and receiving MPI messages.

B. Data Redistribution

The metadata hierarchy serves to connect producers and consumers, avoiding expensive round-trips to storage. Individual tasks are parallelized via MPI processes, and the number of processes need not be the same across tasks connected by LowFive. Because HDF5 does not specify a mechanism to communicate the producer's data decomposition to the consumer—the latter may read the data in a way that does not match the former—and more generally because of the unequal numbers of processes among the cooperating tasks, we must solve the data redistribution problem described below.

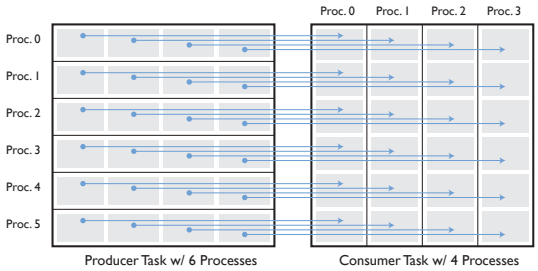

Fig. 3. Example of data redistribution from a producer task with 6 processes to a consumer task with 4 processes.

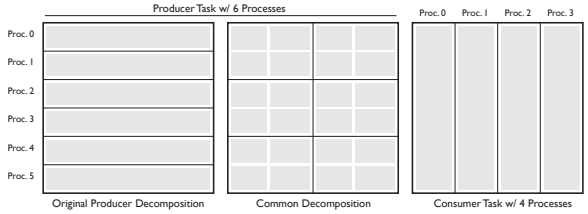

Fig. 4. Data redistribution for the example of Figure 3, showing the common decomposition on which both producer and consumer agree. The common decomposition is a virtual overlay on the producer side. Gray rectangles in the center panel represent the intersections of the producer and consumer decompositions with the common decomposition.

Figure 3 shows an example of a producer with 6 processes sending a distributed dataset to a consumer with 4 processes. The producer decomposed a grid row-wise, while the consumer has a column-wise decomposition. The objective is for all the processes to exchange the correct bits of data. The communication is determined by the intersections of the local data space of each process with the processes of the other task. Importantly, we avoid any unnecessary communication of data not needed by any process, and we also do not aggregate data in a central intermediate location: all communication is direct point-to-point and parallelized among the processes of the tasks. While Figure 3 shows only two tasks, arbitrary fan-in and fan-out of multiple such tasks is also possible.

In the following description of our data redistribution, called `index-serve-query`, assume there are two tasks, producer and consumer, and each task is parallelized over multiple MPI processes. Assume that the producer task generated a dataset, distributed over multiple processes, and that the consumer task requests to access (read) that dataset. Producer and consumer are not required to have equal numbers of processes.

The central problem data redistribution has to solve is that in the HDF5 data model, the producer and consumer do not know anything about each others' data decomposition: when the communication goes through a file, the producers can write regions of a given dataset however they like, and the consumers are free to read any subset of them. To match the data in situ, the producer and consumer implicitly agree on the *common decomposition*, shown Figure 4, of the given d-dimensional dataset into n blocks (where n is the number of producer processes). The decomposition is found by factoring n into d factors $n_1, \ldots, n_d$ that are as close to each other as possible. The domain is cut up into $n_1 \times \ldots \times n_d$ blocks, and the i-th producer process becomes responsible for the i-th block.

During the `index` procedure, described in Algorithm 1, each producer process sends the bounding boxes of all of its local data spaces (written by the individual HDF5 write operations) to the processes whose blocks they intersect in the common decomposition. Having recorded all such indices, the producers go into the `serve` procedure, described in Algorithm 2, which answers the consumer's queries, described in Algorithm 3. The latter can be of two types: redirects and data queries. To read a given data space, each consumer process first sends a request to the producer processes responsible for the blocks in the common decomposition intersected by the bounding box of the data space. Those producer processes respond with the information about which producers contain data that intersects the bounding box. The consumer then sends the requests for the actual data.

The `index`, `serve`, and `query` functions are written using a custom remote procedure call (RPC) abstraction implemented over MPI. All three functions functions are part of LowFive, hidden from the user code.

Algorithm 1: Index

```
1  Function Index()
2      foreach file do
3          foreach dataset dset do
4              compute common_decomposition given n, the
                 number of producer processes, and the shape of dset
5              foreach data space ds ∈ dset do
6                  bb ← bounding box of ds
7                  prod_int_blocks ← vector of producer processes
                     that intersect bb in common_decomposition
8                  foreach p ∈ prod_int_blocks do
9                      nonblocking send bb to p
10             foreach incoming bounding box bb from process q do
11                 append (bb, q) to the vector boxes[file, dset]
```

Algorithm 2: Serve

```
1  Function Serve()
2      do
3          receive request from consumer c
4          if c requests intersections in (file, dset) with bounding
             box qbb then
5              foreach (bb, p) ∈ boxes[file, dset] do
6                  if bb ∩ qbb ≠ ∅ then
7                      append p to vector
                         producer_procs_with_data
8              send producer_procs_with_data to c
9          else if c requests data in data space ds in (file, dset)
             then
10             foreach data reqion d ∈ (file, dset) do
11                 enqueue the data space of d in buffer b
12                 foreach location v in d ∩ ds do
13                     enqueue d[v] in buffer b
14             send b to c
15     while not done
```

Algorithm 3: Query

Function `Query` ($file$, $dset$, *global (file) data space* fs)
  `// Step 1: Get intersecting boxes from indexed blocks`
  compute $common\_decomposition$ given n, the number of producer processes, and the shape of $dset$
  $bb \leftarrow$ bounding box of fs
  $index\_int\_blocks \leftarrow$ vector of producer processes that intersect bb in $common\_decomposition$
  foreach *block* $p \in index\_int\_blocks$ **do**
    send fs to p
    receive vector $producer\_procs\_with\_data$ from p
  `// Step 2: request and receive data`
  foreach *received producer process* p **do**
    send request for data $(file, dset, fs)$ to p
    receive producer's data space ds and data d from p
    compute the intersection between fs and ds and store data in memory

IV. Experiments and Results

A. HPC Platforms

Two supercomputers are used in our tests. Cori at the National Energy Research Scientific Computing Center (NERSC) and Theta at the Argonne Leadership Computing Facility (ALCF) are both Cray XC40 machines, but with different CPUs. We installed LowFive on both machines, and then ran various experiments at a particular site, determined by machine availability and installation of software dependencies. Each experimental result below specifies which machine was used.

Cori is partitioned into Intel Xeon Phi Knights Landing (KNL) and Intel Xeon Haswell computing nodes. We used both partitions. Cori provides 2,388 Haswell nodes, each having two 16-core CPUs with two hyperthreads, and 128 GB DDR4 RAM. The KNL partition contains 9,688 nodes, each having 68 physical cores with 4 hyperthreads per core, and 96 GB DDR4 RAM. Cori is interconnected by a Cray Aries Dragonfly network, and the machine has a peak aggregate computation rate of approximately 30 PFLOPS.

Theta provides 4,392 Intel Xeon Phi Knights Landing (KNL) nodes. Each node has one KNL 64-core CPU, 16 GB high-bandwidth MCDRAM, 192 GB DDR4 RAM, and 128 GB of SSD storage. Theta is interconnected with a Cray Aries Dragonfly network, and the machine has a peak aggregate computation rate of approximately 12 PFLOPs. We used the high-bandwidth memory in flat mode, with four NUMA domains.

For Cori and Theta, we used the Cray programming environment with the GCC version 11.2 compiler and `-O3` optimization.

B. Synthetic Benchmarks

In the following experiments we couple one producer task with one consumer task. We generate synthetic data consisting of two datasets: a regular grid of 64-bit unsigned integer scalar values and a list of particles, each particle a 3-d vector of 32-bit floating-point values. There are 10^6 regularly structured grid points per producer process and 10^6 particles per producer process. With each grid point occupying 8 bytes and each particle occupying 12 bytes, there are 2×10^7 bytes or 19 MiB of data per producer process.

Total # MPI Procs.	# Producer Procs.	# Consumer Procs.	Total # Grid Points	Total # Particles	Total Data Size (GiB)
4	3	1	3.0e6	3.0e6	0.06
16	12	4	1.2e7	1.2e7	0.22
64	48	16	4.8e7	4.8e7	0.99
256	192	64	1.9e8	1.9e8	3.54
1024	768	256	7.7e8	7.7e8	14.34
4096	3072	1024	3.0e9	3.0e9	55.88
16384	12288	4096	1.2e10	1.2e10	223.51

TABLE I
Number of MPI Processes and Data Sizes for 1 Producer and 1 Consumer Task

The values of the grid points and particles encode their global position in the grid and in the global vector of particles, so that the consumer can validate that data have been correctly redistributed. Average times taken over 3 trials are reported.

The numbers of MPI processes and data sizes are shown in Table I. We conduct a weak scaling test, such that the global data size increases proportionally with the number of producer processes. The producer generates two datasets, the grid and the particles, and the consumer reads both of them. Three-fourths of the total processes in the run are allocated to the producer, and the remaining one-fourth are allocated to the consumer. When data are written to physical storage, all processes write collectively to a single HDF5 file in the parallel file system, using MPI-IO. On both Theta and Cori, the Lustre parallel file system is used with default settings.

a) LowFive communicating using a file vs in situ: This experiment, which was run on Theta, compares LowFive data transport when communicating using a file in a parallel file system with LowFive communicating using MPI. 64 MPI processes were allocated to each computing node. Figure 5 shows the results in log-log scale, with the file communication identified as "LowFive File Mode" and the MPI communication labeled as "LowFive Memory Mode." Not surprisingly, writing and reading a file is hundreds of times slower than sending MPI messages. We terminated the file mode at 1,024 MPI processes because of the long run time. A perfect weak scaling curve would be horizontal. The memory mode curve rises slowly, since there is no computation to hide the communication overhead; at the largest scale, communication of 223 GiB of data among 16 K processes takes just over 3 seconds on Theta.

b) LowFive file mode compared with pure HDF5: The next experiment was also run on Theta and measures the overhead introduced by LowFive when communicating in file mode, compared with writing and reading an HDF5 file without LowFive. Figure 6 compares LowFive communicating using a file ("LowFive File Mode") with the producer and consumer writing and reading an HDF5 file directly, without the LowFive layer. This curve is labeled "Pure HDF5." The most overhead is incurred at 64 MPI processes, where the LowFive time is approximately twice as long as the pure HDF5 time. At larger scale, when the file size is larger, the differences

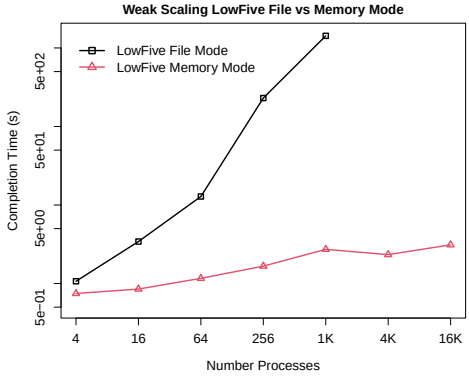

Fig. 5. Time to write/read grid and particles between 1 producer and 1 consumer task, comparing LowFive file and memory modes, in a weak scaling regime, on Theta (KNL).

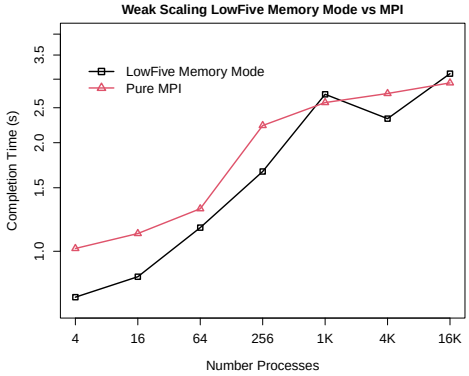

Fig. 7. Time to write/read grid and particles between 1 producer and 1 consumer task, comparing LowFive memory mode, with pure MPI communication, in a weak scaling regime, on Theta (KNL).

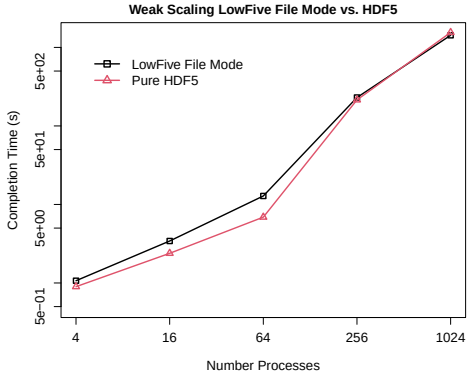

Fig. 6. Time to write/read grid and particles between 1 producer and 1 consumer task, comparing LowFive file mode with pure HDF5 file communication, in a weak scaling regime, on Theta (KNL).

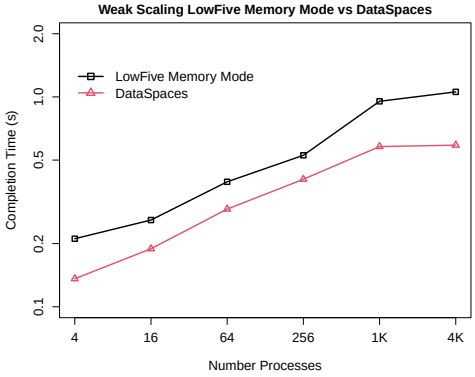

Fig. 8. Time to write/read grid and particles between 1 producer and 1 consumer task, comparing LowFive memory mode, with Dataspaces, in a weak scaling regime, on Cori (Haswell).

between LowFive and pure HDF5 are within the variance of individual runs of the same program.

c) LowFive memory mode compared with pure MPI: This experiment was also run on Theta and measures the overhead introduced by LowFive when communicating in situ compared with a hand-written MPI code that performs the same data redistribution. Figure 7 shows the performance comparison, where the hand-written code is labeled "Pure MPI." In most cases, LowFive performs slightly better (between 10% - 40%), although at scale LowFive is approximately 6% slower. The actual difference at 16 K processes is 0.2 seconds. The reason that LowFive is slightly faster than pure MPI at smaller scales is because LowFive optimizes the serialization of contiguous regions of data better than the hand-written code, which simply iterates over all the data points in the intersection of bounding boxes and serializes them one point at a time.

d) LowFive memory mode compared with DataSpaces: Next we compare with DataSpaces, a popular data transport layer that can be used by ADIOS2. This experiment was run on Cori using the Haswell partition and 32 MPI processes per computing node. Figure 8 compares LowFive in memory mode (communicating over MPI) with the producer and consumer communicating using DataSpaces. DataSpaces is consistently faster than LowFive, although the actual times are on the order of one second or less. The difference at 4K processes is 0.5 s. The DataSpaces server is launched as a separate process that uses dedicated staging nodes to achieve this performance, so the DataSpaces execution requires more total computing resources than LowFive, which sends data directly from producer to consumer. At full scale, we used 4 additional compute nodes for the DataSpaces server. We used the write-local storage version of the DataSpaces API (`dspaces_put_local`) to exchange data in-place, so that the server only maintains indexing metadata.

We hypothesize that the LowFive performance is slowed by synchronization: Because LowFive follows the HDF5 API, the consumer waits for the producer to close the file as a signal that data are ready. Moreover when resolving potential box intersections in the index-serve-query redistribution (§ III-B), indexing the dataset is a collective operation that synchronizes all the MPI processes of the consumer task. In the future, we will investigate how to minimize the synchronization in LowFive.

e) LowFive memory mode compared with Bredala: We also compared performance of LowFive memory mode with Bredala, that data transport layer used in Decaf. This experiment was run on Theta. Figure 9 shows the results. Overall, Bredala did not scale well in this test, with LowFive being

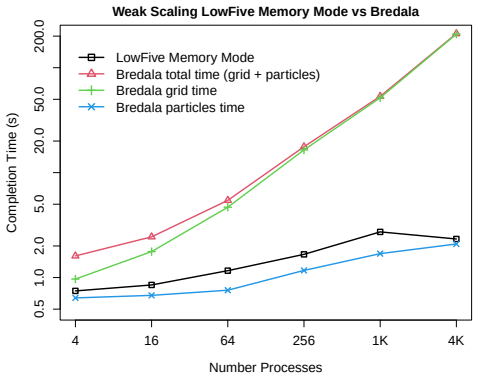

Fig. 9. Time to write/read grid and particles between 1 producer and 1 consumer task, comparing LowFive memory mode, with Bredala, in a weak scaling regime, on Theta (KNL).

much faster. To understand the reasons why, we decomposed the Bredala timing into the time taken to transmit the grid dataset and the time for the particle dataset. We see that the particle dataset performed reasonably, but the grid dataset did not. Bredala uses a different redistribution policy for the particles (a contiguous redistribution) than for the grid (a bounding box redistribution).

The two redistribution policies are illustrated in Figure 10. The contiguous policy at the top of the figure is for a linear 1-dimensional listing of data items, where the redistribution only needs to maintain the same ordering in the global list. Contiguous redistribution allows intersections of producer and consumer processes to be computed easily and data to be moved in contiguous buffers that match the local resident ones. The particles dataset conforms to these requirements. The bounding box redistribution in the bottom of the figure resembles LowFive's redistribution of Figure 3, where intersections in multiple dimensions need to be computed, and data need to be reordered during serialization. In the bounding box redistribution, MPI processes represent n-dimensional subdomains of a global n-dimensional domain, and data are indexed by coordinates in that domain. After redistribution, associated coordinates of a data item must be within the new bounding boxes of each consumer process. This is the case with the grid dataset.

Evidently Bredala does not handle bounding box redistribution efficiently. This result agrees with smaller-scale results in Dreher et al. [4], where the authors showed that the redistribution takes most of the data transport time, and that most of that time is spent computing and communicating the indices of intersecting bounding boxes.

f) Larger data size: Next we compare the three top in situ solutions evaluated so far—LowFive, DataSpaces, and pure MPI—with a data size that is 10 times larger than the previous benchmarks. That is, there are now 10^7 regularly structured grid points and 10^7 particles per producer process, or 190 MiB of data per producer process and 0.55 GiB of data per consumer process (we continue to use three times as many producer processes as consumer processes). The total data size

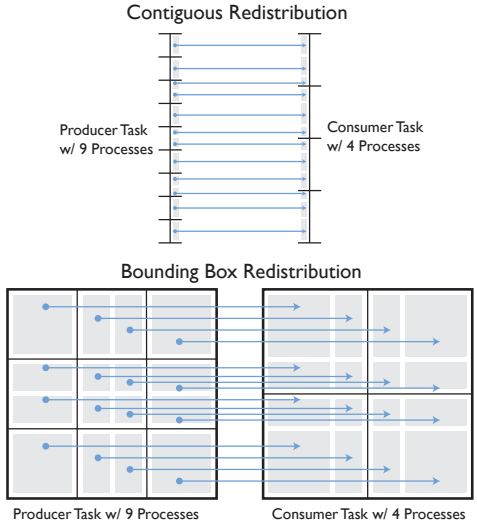

Fig. 10. Two redistribution policies in Bredala: the contiguous policy (top) is used for redistributing a linear list of particles with no spatial requirements, while the bounding box policy (bottom) is used for redistributing regular grid points that need to conform to high-dimensional bounding boxes.

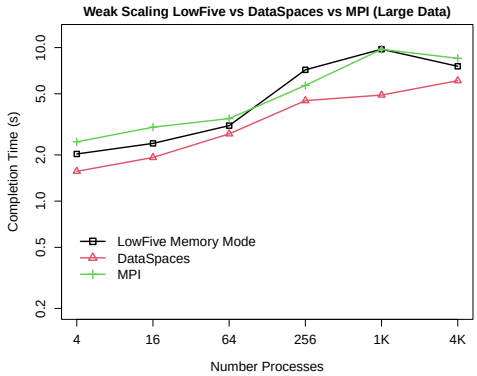

Fig. 11. Time to write/read large size grid and particles between 1 producer and 1 consumer task, comparing LowFive memory mode, DataSpaces, and pure MPI, in a weak scaling regime, on Cori (Haswell).

at the largest scale tested is 0.55 TiB. The objective of this test is to see if the trends we observed remain true if the data are scaled larger. This test is conducted on Cori. Figure 11 confirms that LowFive's performance remains comparable with the hand-written MPI code and with DataSpaces. LowFive remains as fast as MPI and approximately 20% slower than DataSpaces at the largest scale tested.

g) Discussion: The different design choices in LowFive and DataSpaces have implications on resource requirements, usability, and performance. First, DataSpaces is a data staging service launched on separate compute nodes, whereas LowFive moves data directly from producer to consumer with no intermediary agent, service, or additional resources. Second, DataSpaces supports only n-dimensional regular arrays, whereas LowFive supports the full HDF5 hierarchical data model. For example, the data in LowFive can be a collection of multiple smaller regions, or a complete hierarchical graph or mesh structure, which would be tedious to decompose into regular arrays. Third, when a code already uses HDF5,

either directly or through a higher-level library, no change is required to use LowFive and seamlessly switch between storage and in situ data transport. This covers many use cases. In the next section, we show one such example of an actual science workflow that required zero modification to the user codes in order to use LowFive. In contrast, most codes require modification to use DataSpaces.

The other side of this tradeoff is that DataSpaces, by virtue of its additional staging resources and restricted data model, can communicate between 20 to 50% faster than LowFive in our tests. Our results demonstrate that LowFive's advantages—little to no modification to user codes, full hierarchical data model, and no additional services or resources—do have a performance cost compared with DataSpaces. Regarding the performance comparison, we cooperated with the DataSpaces team, who improved their performance during the course of our experiments, to get the most fair comparison possible. We ran DataSpaces in-place; that is, we used `dspaces_put_local` instead of `dspaces_put`, which is why only a small number of additional nodes were needed for indexing metadata, rather than a staging a full data copy. We felt this was the most fair comparison with LowFive, which does not allocate additional memory for indexing and serving data and uses the original data buffers that the user allocated in their code.

C. Scientific Application Use Case

We describe a use case in the study of high-energy physics—cosmology—as further demonstration of the applicability of LowFive to an actual scientific workflow. The scenario, drawn from previously published literature, features a parallel HPC simulation coupled in situ with a smaller-scale parallel analysis task. This case validates our design decision of implementing LowFive as an HDF5 VOL plugin: no changes were made to the HPC simulation nor to the analysis code to generate the following results; the simulation and analysis codes were both used "off-the-shelf." Because we are testing the coupling of unmodified stand-alone codes together in an in situ workflow, we do not compare further here with other technologies such as Bredala or DataSpaces that would have required modifying the codes.

Nyx [29] is a cosmological simulation code. The underlying PDE solver is AMReX [30], a framework for massively-parallel adaptive mesh refinement computations. Nyx also relies on AMReX for I/O. AMReX provides two I/O options: HDF5 (all the simulation data are written into a single file) and plotfiles, a binary format specifically designed by AMReX developers to be optimized for large-scale simulations. Here the data are split into separate files among groups of simulation processes. At certain time steps, the cosmologists need to identify regions of high density, called *halos*. This analysis is performed by Reeber [31]–[33].

In our experiments, we ran Nyx and Reeber on the Cori KNL partition. We ran simulations for different grid sizes: 256^3, 512^3, 1024^3, and 2048^3. In all experiments we used 256 computing nodes for Nyx and 64 nodes for Reeber, with 16 MPI processes per node (in total, 4096 process for Nyx and 1024 for Reeber). Nyx also used OpenMP parallelism, with 16 threads per process. Striping of the file system plays an important role in performance evaluation. We found that the medium striping (recommended by NERSC) gives the best results for the Cori scratch file system, and we used this setting. Since we are interested in the I/O performance, we ran only the first two time steps of the simulation, to produce two snapshots to be analyzed by Reeber. We compared the following 3 scenarios.

- Baseline HDF5. Nyx saves data to disk in HDF5 format, and after Nyx finishes, Reeber reads the data from the files. This is the worst scenario for large-scale problems, because all the data are saved to a single HDF5 file, and that is the reason why we do not have the timings for 2048^3: the I/O did not finish in 1.5 hours.
- Plotfiles. Here we use the native AMReX format, but the data still go to disk.
- LowFive. Here we connect Nyx and Reeber in situ with LowFive. The Python script, which uses Henson [7] to orchestrate this experiment, first creates the DistMetadataVol plugin, to ensure that the data exchange is performed in situ, and then calls Nyx and Reeber. We note that thanks to Henson and LowFive, no changes were required neither to Nyx, nor to Reeber; we only had to relink the codes as shared objects.

To work around the inefficiencies in HDF5, the AMReX writer uses a separate procedure to repack the data into a layout more amenable to disk I/O. Unfortunately, this undermines LowFive's zero-copy ability, which assumes that the data passed to the write operation remain in the same location in memory until the file is closed. As a result, we disable zero-copy in LowFive, and up to three copies of the same data (one native, one repacked, and one in LowFive) can exist in memory simultaneously.

In Table II, we report the time it took to write the two snapshots and to read them. We intentionally omitted the reporting of the plotfiles read time, which was unexpectedly long. When we questioned the cosmologists about this, we learned that code for reading plotfiles was not optimized and not an accurate reflection of true performance. We therefore also excluded the plotfiles read time from our calculation of the speed-factor in the last column of the table, to avoid inflating the improvement of LowFive over plotfiles. The speed-up we report is a lower bound, assuming the plotfile reading time is zero. We calculate the speed-up factor in the last two columns (how much time we gain by switching to LowFive from HDF5 and plotfiles, respectively). If we compare standard HDF5 with LowFive, the advantage of using LowFive is immense. Even for the native AMReX plotfile format, LowFive is faster by an order of magnitude.

V. Conclusions

A. Recap

We presented LowFive, an in situ data transport layer for HPC workflows. While the primary communication mecha-

Data Size	LowFive Write Time	LowFive Read Time	HDF5 Write Time	HDF5 Read Time	Plotfiles Write Time	LowFive vs HDF5	LowFive vs Plotfiles
256^3	2.87	0.106	5.46	0.37	4.42	1.9	1.54
512^3	2.00	0.287	104.20	0.69	18.10	52.01	9.03
1024^3	2.87	0.628	920.44	3.02	35.00	320.00	12.17
2048^3	7.69	3.205	x	x	154.52	x	20.09

TABLE II
RESULTS OF NYX–REEBER USE CASE. TIMINGS ARE IN SECONDS, ON CORI (KNL).

nism is MPI, we also provide options to communicate using files in a parallel file system. We built our solution on the HDF5 VOL technology. HDF5 is one of the most common data models, with many HPC, big data, and AI applications either using HDF5 directly or able to convert to/from HDF5. Writing LowFive as a VOL plugin, we benefit from HDF5's rich metadata describing the data model while affording users the familiarity of HDF5 and minimizing code modifications to their applications. In particular, we made no changes whatsoever to the Nyx and Reeber applications in order to use LowFive. By creating our own version of the HDF5 hierarchy in memory and communicating over MPI, we were able to sidestep many of the performance issues plaguing HDF5 physical file I/O, and HPC parallel I/O in general.

Several conclusions are evident from our performance evaluation. We saw that communicating over MPI is much faster than through physical files, as expected. When communicating through physical files, the overhead incurred by LowFive over using pure HDF5 was negligible. Similarly, when communicating through MPI, LowFive's overhead was negligible compared with a hand-written MPI code.

When comparing with DataSpaces, DataSpaces outperformed LowFive by 50% in the small-scale test and 20% in the larger-scale test. Both libraries scaled similarly; i.e., their performance curves were roughly parallel. DataSpaces required additional resources compared with LowFive, and the DataSpaces API is limited to regular structured n-dimensional arrays. LowFive, by virtue of HDF5's rich data model, supports many more complex data types and spaces, both globally and locally, and can redistribute them. The comparison with Bredala showed that LowFive outperformed Bredala overall, both in terms of time and scalability.

In a science use case drawn from the literature, LowFive continued to perform well. In the cosmology study, LowFive outperformed both the HDF5 and plotfiles I/O formats used by the cosmologists, by significant margins. No changes were required to the simulation and analysis codes in order to use LowFive.

B. Lessons Learned

One lesson learned is that although a plugin architecture such as VOL is a convenient vehicle for injecting custom functionality, maintaining compatibility with an existing standard requires in-depth knowledge of the underlying library. Developing LowFive required a significant amount of effort in reverse-engineering the HDF5 source code. Nonetheless, we believe that it is valuable to re-purpose popular software in this way instead of reinventing the wheel, and the HPC community should design software (like HDF5) from the outset with carefully-planned abstraction layers (e.g., the VOL) meant to be used by other groups for other purposes, at a high level, with better documentation and support for its use.

It is important to understand and balance the trade-offs in using such a combined design. On one hand, a VOL plugin like LowFive makes interchanging physical file I/O with in situ communication seamless, affording high-performance zero-copy parallel communication transparently. However, simulation codes can still negate some of these advantages. A salient example is the implementation of HDF5 I/O in Nyx, where the AMReX writer repacked the data, forcing LowFive to make a deep copy. Some other simulations gather all data to a single MPI process before writing output serially, undermining LowFive's ability to exploit parallel point-to-point communication between many processes simultaneously. A promising approach to reconciling different approaches for I/O and in situ communication is to hide custom workarounds for file I/O inside another VOL plugin, so that the top-level user API remains parallel HDF5 I/O. This is already happening in some cases such as the other VOL plugins cited in § II-C.

C. Future Work

In the future, we plan to investigate how to reduce synchronization in LowFive, so that we can consume data as soon as it is available, and overlap reading and writing of data to the extent possible. We are working on profiling our communication at finer grain in order to see where the remaining bottlenecks are. Scheduling the communication pattern to alleviate congestion is something we have not investigated in LowFive. All producer processes act as servers, and all consumer processes act as clients, in our fully parallel point-to-point messaging interchange. Depending on the communication workload, various collective algorithms may be more efficient than point-to-point; scheduling communication for a variety of workloads is a future research topic. LowFive currently covers approximately 80% of the HDF5 API, and we are working on adding the remaining HDF5 functions to LowFive. We are also actively building a higher-level workflow system that uses LowFive as its transport layer.

ACKNOWLEDGMENTS

The authors wish to thank Quincey Koziol for his extensive help with HDF5 Virtual Object Layer; Houjun Tang, Jean Sexton, and Zarija Lukić for their help with Nyx and its use of HDF5, and Phil Carns for his help installing and configuring Mochi for DataSpaces. This work is supported by Advanced

Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357 and DE-AC02-05CH11231, program manager Margaret Lentz. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract no. DE-AC02-05CH11231. This work is also based upon work by the RAPIDS2 Institute supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the Advanced Computing (SciDAC) program under Award Number DE-SC0023130.

REFERENCES

[1] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An Ephemeral Burst-Buffer File System for Scientific Applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 807–818.

[2] M.-A. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS—A Temporary Burst Buffer File System for HPC Applications," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 72–91, 2020.

[3] C. Harrison, "Conduit," https://llnl-conduit.readthedocs.io/en/latest/, 2021.

[4] M. Dreher and T. Peterka, "Bredala: Semantic Data Redistribution for In Situ Applications," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 279–288.

[5] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and its Applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 2011, pp. 36–47.

[6] O. Yildiz, M. Dreher, and T. Peterka, "Decaf: Decoupled dataflows for in situ workflows," in *In Situ Visualization for Computational Science*. Springer, 2022, pp. 137–158.

[7] D. Morozov and Z. Lukic, "Master of Puppets: Cooperative Multitasking for In Situ Processing," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 285–288.

[8] T. Hobson, O. Yildiz, B. Nicolae, J. Huang, and T. Peterka, "Shared-Memory Communication for Containerized Workflows," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2021, pp. 123–132.

[9] D. Pugmire, N. Podhorszki, S. Klasky, M. Wolf, J. Kress, M. Kim, N. Thompson, J. Logan, R. Wang, K. Mehta *et al.*, "The Adaptable IO System (ADIOS)," in *In Situ Visualization for Computational Science*. Springer, 2022, pp. 233–254.

[10] C. Harrison, "Blueprint," https://llnl-conduit.readthedocs.io/en/latest/blueprint.html, 2021.

[11] B. Whitlock, J. Favre, and J. Meredith, "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)*, vol. 10. Eurographics Association Aire-la-Ville, Switzerland, 2011, pp. 101–109.

[12] 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*, 2015, pp. 25–29.

[13] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, "The SENSEI Generic In Situ Interface," in *2016 second workshop on in situ infrastructures for enabling extreme-scale Analysis and visualization (ISAV)*. IEEE, 2016, pp. 40–44.

[14] A. Gainaru, L. Wan, R. Wang, E. Suchyta, J. Chen, N. Podhorszki, J. Kress, D. Pugmire, and S. Klasky, "Understanding the Impact of Data Staging for Coupled Scientific Workflows," *IEEE Transactions on Parallel and Distributed Systems*, 2022.

[15] 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.

[16] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, "Mochi: Composing Data Services for High-Performance Computing Environments," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020.

[17] 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.

[18] C. Harrison, "Blueprint," https://llnl-conduit.readthedocs.io/en/latest/relay.html, 2021.

[19] B. Dong, S. Byna, K. Wu, H. Johansen, J. N. Johnson, N. Keen *et al.*, "Data Elevator: Low-contention Data Movement in Hierarchical Storage System," in *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. IEEE, 2016, pp. 152–161.

[20] J. Ye, A. Kougkas, and X.-H. Sun, "HDF5 VOL Connector to Apache Arrow," https://sc21.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost151.html, Nov 2021.

[21] W. keng Liao and K. yuan Hou, "Log-Layout Based VOL - an HDF5 VOL Plugin for Storing Datasets in a Log-Based Layout," https://github.com/DataLib-ECP/vol-log-based, 2022.

[22] BlueBrain, "HighFive - HDF5 header-only C++ Library," https://github.com/BlueBrain/HighFive, 2022.

[23] A. Collette, *Python and HDF5: Unlocking Scientific Data*. " O'Reilly Media, Inc.", 2013.

[24] R. K. Rew, B. Ucar, and E. Hartnett, "Merging NetCDF and HDF5," in *20th Int. Conf. on Interactive Information and Processing Systems*, 2004.

[25] J. Krishna, "Scorpio – parallel i/o library," https://e3sm.org/scorpio-parallel-io-library/, 2020.

[26] O. G. Yalçın, "A Guide to TensorFlow 2.0 and Deep Learning Pipeline," in *Applied Neural Networks with TensorFlow 2*. Springer, 2021, pp. 95–120.

[27] A. Gulli and S. Pal, *Deep Learning with Keras*. Packt Publishing Ltd, 2017.

[28] D. Morozov and T. Peterka, "Block-Parallel Data Analysis with DIY2," in *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*. IEEE, 2016, pp. 29–36.

[29] A. S. Almgren, J. B. Bell, M. J. Lijewski, Z. Lukić, and E. Van Andel, "Nyx: A massively parallel amr code for computational cosmology," *The Astrophysical Journal*, vol. 765, no. 1, p. 39, 2013.

[30] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves *et al.*, "Amrex: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, pp. 1370–1370, 2019.

[31] B. Friesen, A. Almgren, Z. Lukić, G. Weber, D. Morozov, V. Beckner, and M. Day, "In situ and in-transit analysis of cosmological simulations," 2016. [Online]. Available: http://dx.doi.org/10.1186/s40668-016-0017-2

[32] D. Smirnov and D. Morozov, "Triplet merge trees," in *Topological Methods in Data Analysis and Visualization*. Springer, 2017, pp. 19–36.

[33] A. Nigmetov and D. Morozov, "Local-global merge tree computation with local exchanges," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: ACM, 2019, pp. 60:1–60:13. [Online]. Available: http://doi.acm.org/10.1145/3295500.3356188