Transparent In Situ Data Transformations in ADIOS

David A. Boyuka II [1,2,*], Sriram Lakshminarasimhan [3], Xiaocheng Zou [1,2], Zhenhuan Gong [1,2], John Jenkins [1,2], Eric R. Schendel [1,2], Norbert Podhorszki [2], Qing Liu [2], Scott Klasky [2], Nagiza F. Samatova [1,2,*]

[1] North Carolina State University, NC 27695, USA

[2] Oak Ridge National Laboratory, TN 37831, USA

[3] IBM India Research Lab, Bangalore 560045, India

[*] Corresponding authors: daboyuka@ncsu.edu, samatova@csc.ncsu.edu

***Abstract*—Though an abundance of novel "data transformation" technologies have been developed (such as compression, level-of-detail, layout optimization, and indexing), there remains a notable gap in the adoption of such services by scientific applications. In response, we develop an in situ data transformation framework in the ADIOS I/O middleware with a "plugin" interface, thus greatly simplifying both the deployment and use of data transform services in scientific applications. Our approach ensures user-transparency, runtime-configurability, compatibility with existing I/O optimizations, and the potential for exploiting read-optimizing transforms (such as level-of-detail) to achieve I/O reduction.**

We demonstrate use of our framework with the QLG simulation at up to 8,192 cores on the leadership-class Titan supercomputer, showing negligible overhead. We also explore the read performance implications of data transforms with respect to parameters such as chunk size, access pattern, and the "opacity" of different transform methods including compression and level-of-detail.

I. Introduction

In response to the demands of increasing scientific simulation data scale, a variety of "data transformation" techniques have been developed. These transformations change the encoding of scientific data to achieve benefits including: reduced I/O costs and storage footprint via data compression [1]–[5], faster read access times via level-of-detail [6], [7] or storage layout optimization [8], and faster query response times via bitmap [9] or inverted [10] indexing. Also to cope with increasing data scale, many of these data transforms have been applied *in situ* by co-locating with the application on compute cores or *in transit* by utilizing dedicated staging nodes/cores, thus moving away from data *post*-processing to data *co*-processing.

Despite offering numerous potential benefits, actual adoption of in situ data transforms by scientific applications has been inhibited by several obstacles. First and foremost among these is the yet-unresolved key question of *where to place data transformation services within the end-to-end scientific workflow*. This question has been a subject of much debate in the scientific community at large. The consensus, especially among end users, seems to be emerging in favor placement at the *I/O middleware level*. For instance, different scientists working on the exascale GTS fusion plasma and S3D combustion simulation codes have stated that using data compression or indexing through I/O middleware such as ADIOS makes the most sense for them[1].

In response to this critical need, this paper aims to bridge the growing gap between the increasing number of data transformation methods and the limited rate of their transparent integration within scientific workflows. Specifically, the paper makes the following **contributions**:

1) We design a **generic framework** for transparent, in situ data transformation in the ADIOS I/O middleware (see Section III for the rationale behind selecting ADIOS as the platform). This framework is now included in the ADIOS 1.6.0 release.
2) We reduce the *cost of integration* of data transformation services within scientific applications and avoid *data semantics erasure* (i.e., the degeneration of structured data into simple byte arrays when transforms are applied outside of I/O middleware) by ensuring **user transparency** of the proposed framework within ADIOS, thus leveraging existing I/O middleware integration and maintaining the capture of data semantics.
3) We enable *ease of transform experimentation, tuning, and swapping* through **runtime configuration of data transform plugins**. By augmenting the ADIOS XML runtime interface, as opposed to using a hard-coded configuration, transforms can be selected and tuned without modification or recompilation of application code.
4) We support *I/O pipeline flexibility/compatibility* by explicitly **decoupling data transformations and I/O transports** in ADIOS, thus achieving orthogonality between these services. This permits users to continue using existing I/O transports while augmenting their I/O pipeline with data transforms.
5) We address the *missing support for read-optimizing transforms in I/O middleware* by incorporating a **flexible read model** into our transforms framework.
6) We have integrated and evaluated a variety of data

[1] stated by application scientists at an SDAV conference call

transformation methods in the categories of compression, level-of-detail, indexing, and storage layout optimization. While only some appear in our results due to space constraints, the possibility of implementing this range of methods underscores the flexibility of the framework. Code for these plugins, along with the transform framework codebase used in this paper, can be found on GitHub [11] (some plugins require additional libraries [12]).

Using this framework, we also evaluate the benefits and drawbacks of data transformations in general for both writes and reads (see Section V-C). On the write side, we apply multiple data transformations to the existing Quantum Lattice Gas [13] (QLG) simulator on up to 8,192 cores on the leadership-class Titan supercomputer at Oak Ridge National Lab (without modifying the application code), demonstrating the framework's scalability, as well as the potential for in situ compression to reduce I/O time and storage footprint. On the read side, we analyze the performance of different read access patterns over transformed data. We show the impact of "transform opacity" (i.e., the atomicity of transformed data), and how our flexible read model enables some transforms to overcome this effect to improve I/O time, even outperforming the no-transform case.

II. Related Work

Many modern scientific applications use publicly available I/O middleware solutions, such as HDF5 [14], ADIOS [15], and pnetCDF [16] to handle complex parallel I/O tasks in HPC systems. Such libraries provide stable and portable I/O interfaces, and store data in self-describing file formats that form the underpinnings of many scientific data workflows. However, there is limited support for run-time data transformations.

For example, HDF5 offers end-users the ability to incorporate custom filters such as run-time data compression [17]. However, the lack of parallel write support for transforms makes it inapplicable for in situ use at scale, and only hard-coded transform configuration is available (limitations we address in our ADIOS-based approach).

The more recent Damaris [18] middleware uses a dynamic plugin-loading architecture that can support some data transforms (compression has been demonstrated). However, despite its flexibility, the shared whiteboard model employed does not in itself overcome data semantics erasure, as it layers data transforms above the I/O layer, unlike the approach proposed in this paper.

ADIOS [15] features a modular I/O transport layer enabling runtime selection of I/O methods. Attempts to use this layer for data transformations [19]–[21], while yielding useful research results, inherently prevent the user from choosing any other I/O transport. We address this issue through I/O decoupling.

Finally, many standalone *read-optimizing data transforms* have been demonstrated, including level-of-detail encoding [6], [7] and storage layout optimization [8], [22]. However, to our knowledge, no current I/O middleware is capable of offering these transforms, along with their resultant I/O-reduction benefits, to end users as a service.

On the subject of data service placement, besides *in situ* placement, movement of data to staging cores or nodes for processing, *in transit* placement, has also been explored in the literature. For example, DataStager [20] moves data asynchronously from compute nodes to staging nodes, enabling data processing in the staging area. PreDatA [19], an extension of DataStager, characterizes and reorganizes application data to speed up data analytics by offering both in situ and in transit placement options. JITStager [21] provides a more dynamic infrastructure to customize data movement at scale by leveraging both compute and staging nodes.

While in transit placement has proven to be an effective strategy, here we focus on in situ placement of data transforms for two main reasons. First, in situ placement makes more sense as a first step, because achieving in transit placement is then relatively straightforward. For instance, the `DATASPACES` and `DIMES` [23] staging I/O transports send data to staging cores/nodes, where those data may then be written to disk via a second call to ADIOS. By applying "in situ" data transforms at the staging level, we effectively achieve "in transit" data transforms without further effort. In contrast, porting an in transit-only solution back to in situ may be obstructed by implicit dependencies, such as assumed exclusive ownership of staging cores, etc. Second, in transit placement relies on the availability of staging nodes, whereas in situ does not, making the latter more widely applicable at this time. In the future, however, a native in transit data transform framework could be developed in ADIOS as an extension of this work.

III. Background: ADIOS

We choose to base our transform framework in the ADIOS I/O middleware [15] for a number of reasons, besides our hands-on experience with its internals. First, ADIOS has a broad user-base, with more than a dozen large-scale applications using ADIOS in production. Adoption of various data transformation services through our framework by these applications could provide invaluable feedback on the design principles that underlie our framework and suggest ways to improve its functionality. Second, ADIOS already implements a *modular I/O transport framework*, which we require for our goal of decoupling I/O and data transforms. Finally, the "process group" data parallelism model of ADIOS simplifies our parallel implementation.

Next, we review key technical components of ADIOS pertinent to our work.

A. Modular Layered Service Model

ADIOS is divided into three main layers: write and read "transport" layers, and a "common" layer. The transport layers handle input and output from storage, respectively, and are modular, allowing for read and write I/O transport technologies (called "transport methods") to be selected independently at run-time. In this paper, we often group these as simply the "transport layer." In contrast, the common layer acts as the central "hub" of ADIOS, from which the transport layer is invoked. The common layer defines the API for transport methods, as well as the top-level user API.

Here, we develop a new layer, the "**data transformation**" layer, analogous to the transport layer, to support the modular integration of data transformation methods.

B. Process Group Data Model

The fundamental unit of data in ADIOS is the "Process Group," or "PG." A PG contains all data produced by a single writing process during a single write step (e.g., 16 writing processes and 4 write steps yield 64 PGs). This layout is depicted in Figure 1. Each PG contains one or more variables, each defined as a scalar or a multidimensional array of a certain datatype. A variable may be defined across many PGs, with each PG containing a local "block" of that variable, together comprising a logical global array. The PG model achieves high write throughput by eliminating data exchange among processes [24], while also performing well for reads [25].

When reading data, ADIOS abstracts away the division of data into PGs. The transport layer transparently converts read requests in the global coordinate space to a set of PG-local read requests, then stitches the results together before returning them to the user. While this translation is convenient to the user, the fact that it is handled in the transport layer presents a challenge to data transformation, as explained in Section IV-C.

Figure 1. The PG data layout in an ADIOS file. Variables are partitioned by PG's according to which process produced which chunk.

C. Runtime Configuration via XML

To support runtime configuration, ADIOS uses an XML configuration file that is parsed during initialization. It includes the schema for variable types and dimensions, as well as the write transport method to use (along with parameters). We extend this configuration file to enable tagging of transform methods on variables (see Section IV-A).

D. ADIOS Read API

As our flexible read model is based on selectively reading transformed data based on user read requests, it is important to first describe the read semantics of ADIOS. ADIOS supports various read "selections." These currently include "bounding box" (i.e., subvolume) and "point list" selection types, which operate in the global coordinate space, and "writeblock," which reads all of a variable's data from a certain PG. ADIOS also supports two return policies: user buffer mode and chunking mode. For the former, the user supplies a sufficiently-sized buffer for ADIOS to populate with the entire result. For the latter, ADIOS returns pieces of the result ("chunks") over time; this mode is designed for memory-constrained and streaming environments.

Pursuant to user-transparency, we endeavor to support these access methods over transformed data (see Section IV-B). Also, our flexible read model for supporting read-optimized transforms interacts with read requests from this API to decrease I/O time (see Section IV-C).

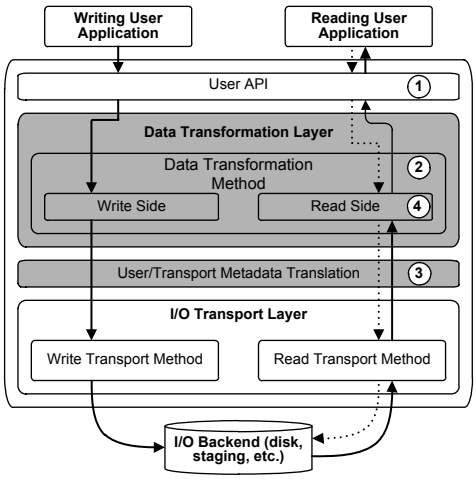


Figure 2. An overview of the transforms framework within ADIOS. The shaded area is newly-added to enable data transformation, while the number markings roughly indicate where each of our four approaches are implemented (1: user transparency, 2: runtime configuration, 3: I/O decoupling, and 4: flexible read model).

IV. Method

Figure 2 presents a high-level overview of our transforms framework within ADIOS. The data transformation layer (the shaded region) provides the framework for incorporating transforms into the ADIOS data flow. Similarly to the I/O transport layer, the data transform layer defines a standardized interface for integrating new data transformation "plugins" into ADIOS. Each plugins is divided into a write and read component, as our flexible read model requires differentiated functionality from the write side (see Section IV-C). Plugins themselves are written in C (as is ADIOS), but their transforms services are also compatible with ADIOS's other language bindings (e.g., FORTRAN and C++).

The transforms framework consists of three main components:

- **Runtime-configured plugins** (Section IV-A) allow for different data transforms to be selected/configured by the user, with the plugins themselves defining the actual data transformation.
- **User transparency** and **I/O decoupling** (Section IV-B) ensure that the data transform layer is compatible with ADIOS, both with the User API above and the I/O transport layer beneath.
- **Flexible read model** (Section IV-C) explains how read-optimizing transforms can be empowered to influence I/O and to speed up read times.

A. Runtime-configured Plugins

Traditionally, adding data transform services to a scientific application requires *ad hoc* integration with its tested/stable codebase, a risky and time-consuming proposition. Runtime configuration of data transforms through an I/O middleware can reduce such costs and provide a mechanism for transform experimentation, tuning, and swapping, as changes will not require application code modification/recompilation. It can also avoid "technology lock-in," which leads to the risk of dependence on particular data transform codes that may become unsupported in the future.

```
<var name="temperature" type="double"
    dimensions="NX,NY" transform="zlib:5"/>
```

Figure 3. XML configuration to select data transformation.

The runtime configuration of data transforms plays a simple, yet important, role in the data transform framework: it is the only component that an end-user directly interacts with. The interface is an extension of the existing ADIOS XML configuration for defining data variables (Figure 3). The added attribute, "transform," specifies both the specific data transform to apply (zlib compression, in this case) as well as any parameters that transform accepts (here, the zlib speed vs. quality parameter).

Internally, the user's install of ADIOS is configured with a set of transform plugins, each of which defines the functions necessary for encoding and decoding data passing through the I/O transport layer. The transform name specified in the XML is mapped through a table to the corresponding plugin code, and the parameters after the colon (':') are passed along. This XML is only needed during the write process; when reading, any data transforms that were applied to the data were also recorded alongside it, and so can be detected automatically.

B. User-transparency and I/O Decoupling

One of the key features of the I/O middleware packages used by scientific applications is portable, self-describing file formats, which form the underpinnings of many scientific data workflows. Handling the encoding/decoding of a data transform (e.g., compression) at the application level inherently strips the semantics from the data (datatype, grid dimensions, etc.) by converting it into an opaque byte array, thus distorting the original form of the data, and, as a result, countermanding the self-describing file format and breaking any associated workflow.

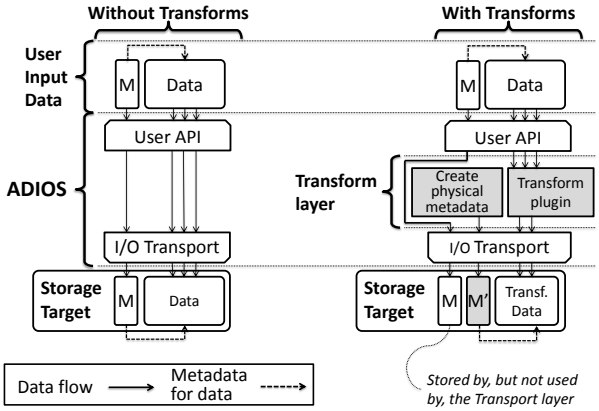
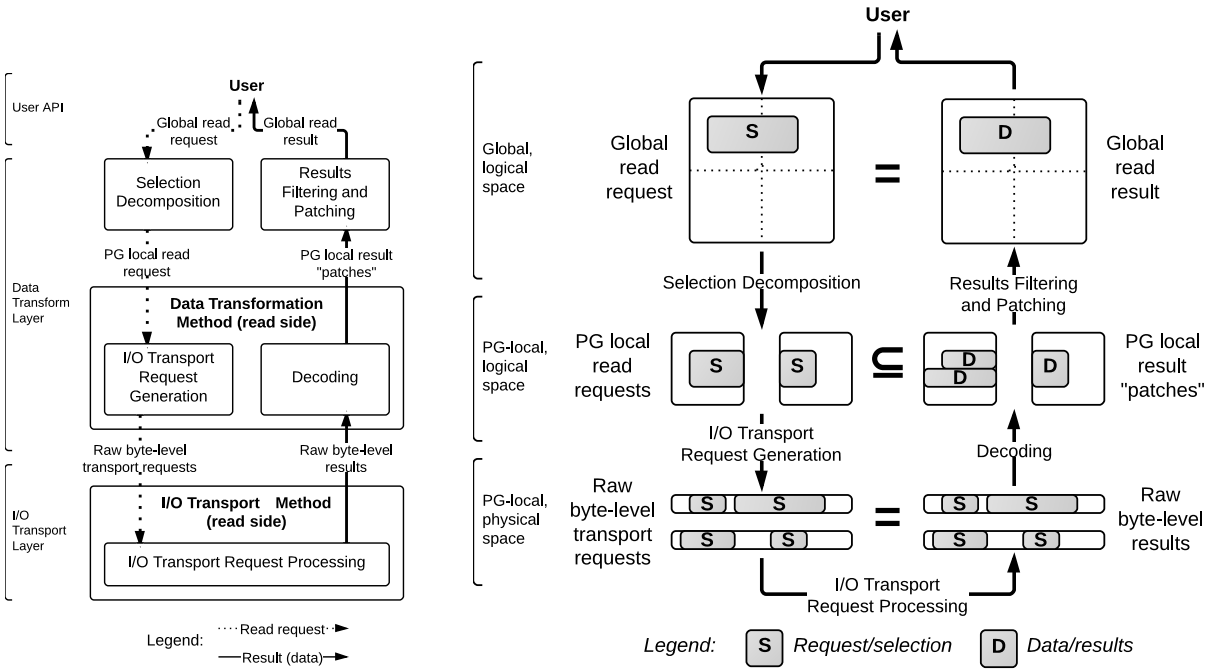


Figure 4. A comparison of writing with and without a data transform. In both cases, the user passes the variable's data along with its corresponding metadata (dimensions and datatype, denoted with an "M" in the figure). Normally, ADIOS uses the user-provided metadata to determine the data layout in storage; however, when a data transform is selected, metadata translation generates a new physical metadata and presents that to the transport layer, instead. The original metadata is stored for later use.

Placing the data transform layer within ADIOS overcomes data semantics erasure, but introduces challenges of its own. By virtue of its location in the ADIOS stack, the data transform layer must now maintain compatibility with both the *User API* above and the *I/O Transport layer* below (as depicted in Figure 2). Additionally, this compatibility must be maintained during both the writes and reads. Here, we cover the write side only, as the read side is more naturally explored in context with the flexible read model (Section IV-C).

Achieving transparency on the write side can be reduced to achieving *physical data independence*, that is, translating between the *logical* and *physical* view of the data. Currently in ADIOS, the user will provide variable data with associated *logical* metadata (e.g., tagging a variable as a 3D array of double-precision values). Then, this same data and metadata are passed to the I/O transport layer, which assumes that they match the physical representation on disk (e.g., data on disk is a contiguous chunk of product-of-dimensions times size-of-datatype bytes).

We break this equivalence to permit a physical representation different from the logical one through *user/transport metadata translation*, as depicted in Figure 4. Two copies of the dimension/datatype metadata are maintained for each transformed variable: a logical and a physical version. The logical metadata is what is received from, and later presented back to, the user. In contrast, the physical metadata is a



(a) Read Process - Component View

(b) Read Process - Dataflow View

Figure 5. The read process for a transformed variable, illustrated through a dual perspective. Figure 5a shows the *component view*, with boxes representing the *actions* taken to complete a read request, whereas Figure 5b shows the *dataflow view*, highlighting the evolution of the *read request itself*, and later the corresponding result data, as it flows through ADIOS. The figures are complementary: the objects in the dataflow view are the transitions (inputs/outputs) between the actions in the component view and, conversely, the actions in the component view correspond to transitions converting between requests/results in the dataflow view.

fabricated 1-dimensional byte array for each transformed variable, with length equal to the post-transform size. The I/O transport layer is only exposed to the physical metadata, effectively "tricking it" into treating the transformed data as a byte array. The logical (original) metadata is *stored* by the transform layer for later use, but is never *interpreted* by the I/O transport layer.

This metadata translation process operates alongside of, and in support of, the *data* translation (i.e., transform). The actual data transform is performed by the specific transform plugin, through the main Write API function, "apply," which takes the original data buffer and produces a new, encoded buffer. More detail on this API is available in the source code on GitHub [11].

Thus, under this strategy, the user always sees the original form of the data, whereas the I/O transport layer receives the transformed data as if the user had written directly as 1D byte arrays, a supported use case in ADIOS. The transform layer plays "middle man," presenting to the user and I/O transport layer their corresponding views of the data, thus achieving both *user transparency* and *I/O decoupling*.

C. Flexible Read Model

We begin with a motivating example for our proposed "flexible read model." Consider the APLOD level-of-detail encoding [7], which optimizes read performance by supporting partial-precision reads with multi-fold I/O reduction *and* full-precision reads with little overhead, both on the same copy of scientific data and with no storage overhead. This encoding (a data transform) can achieve I/O speedup by reading a subset of the data from disk via out-of-core optimization. However, this optimization is nullified if entire chunks of APLOD-encoded data are read from disk by an I/O middleware before passing them to APLOD, as it is too late to achieve I/O reduction after the chunk is read. Instead, this reduction must occur earlier by consulting the specific transform plugin (APLOD, in this case) *during the read scheduling phase* to eliminate unnecessary I/O.

Thus, to fully support read-optimizing transforms, such as APLOD, we propose a protocol for modified read scheduling, herein referred to as the "flexible read model." The read pipeline implementing this model is illustrated in Figure 5 through a dual perspective: the *component view* and the *dataflow view*. Figure 5a shows the sequence of *actions* that take place to answer a user's read request, while Figure 5b depicts the evolution of the *user's request itself*, and later the returned data, as it flows through the pipeline. Each component of this pipeline is described separately in the following subsections.

It should be noted that the read pipeline complexity stems not only from integrating "black-box" transform plugin input into read scheduling, but also from the PG model espoused

by ADIOS. ADIOS's PG model induces a "reader-makes-right" model that involves data patching/filtering procedures to abstract away the division of a file into PGs. Likewise, though the PG model simplifies the transform write side, the read side is complicated by the need to handle this patching/filtering, as well.

It is helpful to consider the different *coordinate spaces* at play throughout this data pipeline. Initially, user selections exist in the *global*, *logical* space, that is, with absolute, global coordinates across all PGs defined in terms of the data's logical form (e.g., 3D array of double-precision values). In contrast, due to *metadata translation* (Section IV-B), the I/O transport layer services requests in the *PG-local*, *physical* space, that is, relative to a given PG and in terms of byte offsets within that PG. Thus, the read process essentially consists of translation of user requests down to the PG-local, physical space, followed by reverse translation of the result data back to the user's global, logical space.

Throughout this process, the transform plugin remains a "black box," so we first discuss the plugin read API we have developed, which defines the "tools" each transform plugin makes available to the framework. After this, we break down each step of the read process in Figure 5a.

```
// For I/O Transport Request Generation
void generate_raw_read_requests(
             global_read_request *g_read,
             pg_read_request *pg_read);

// For Decoding
datablock * raw_read_request_completed(
             global_read_request *g_read,
             pg_read_request *pg_read,
             raw_read_request *completed_raw_read);

datablock * pg_read_request_completed(
             global_read_request *g_read,
             pg_read_request *completed_pg_read);

datablock * read_request_completed(
             global_read_request *completed_g_read);
```

Figure 6. Plugin read API.

1) Plugin Read API Overview: The functions of our plugin read API are listed in Figure 6. The first function, `generate_raw_read_requests`, services the "I/O Transport Request Generation" step in Figure 5a, whereas the other functions support the "Decoding" step.

The `generate_raw_read_requests` function is where the transform plugin inserts its guidance into the read scheduling process: given the portion of the user's selection that overlaps a certain PG (i.e., a PG-local, logical selection), this function must produce a series of PG-local, physical read requests to answer that request. Later on, the other three functions (i.e., `*_request_completed`) are called with the bytes read, and are expected to produce chunks of logical data from them.

The "request" parameter types correspond to the three levels of requests shown in Figure 5b, from top to bottom, and contain information specific to that level:

- `global_read_request` is the original read selection;
- `pg_read_request` describes a PG-local portion of the original read selection; and
- `raw_read_request` is a byte-range read request from storage.

There is a clear hierarchy among these types: each global request begets one or more PG requests, and each PG request begets one or more raw requests. In fact, these types are linked in a tree structure, and each callback receives a "path" in this tree from the root (global request).

The other datatype listed, `datablock`, represents a chunk of data in the global, logical space to facilitate physical-to-logical data mapping. Each `datablock` contains a buffer, a datatype, and a selection object defining its bounds/shape.

2) Selection Decomposition: The first step in completing a read request is to generate a set of PG-local selections from the user's global selection. While ADIOS's transport layer already does this for non-transformed variables, due to user/transport layer metadata translation, the transport layer no longer sees the logical view of the data, so we replicate this process in the transform layer. Currently, we use ADIOS's algorithm: intersect the user's selection with the bounds of each PG. This performs adequately at current file scales, though in the future a more efficient intersection algorithm (e.g., using space partitioning) could be added.

3) I/O Request Generation: The next step is to translate the PG-local selections into physical (byte-level) reads. As discussed at the top of Section IV-C, the straightforward approach of simply reading the entire PG to be passed through the transform plugin cannot exploit read-optimizing transforms. Thus, we take a *plugin-driven* approach, where the transform plugin itself dictates which portions of the transformed data are required. The `generate_raw_read_requests` function allows the plugin to drive this process by emitting raw read requests based on each PG-local portion of the user's selection.

It's important to note that, even with this flexibility, plugins that do not require flexible read support (e.g., compression) are not complicated. In such cases, any touched PG can simply be scheduled for read in its entirety, requiring only two lines of code, and during the later de-transformation step, the decoded result can be returned as a single datablock with one more line of code.

4) Decoding: Once the raw read requests scheduled by the transform plugin begin to complete, the results are passed back to the plugin via the `*_completed` callback functions in the API (Figure 6). `raw_read_request_completed` is called for each completed raw read, `pg_read_request_completed` is called when all raw reads associated with a PG-local request

are done, and `global_read_request_completed` is called when all raw reads stemming from a user's original read request are finished. Results may be returned from any of these callbacks, giving the plugin flexibility in reconstructing the requested data. For instance, a plugin may wish to return results immediately upon receiving each chunk of raw bytes it requested. Alternatively, it may wish to accumulate raw reads until an entire PG's worth are collected, at which time it may return a single, larger result.

Note that, for simplicity of plugin implementation, plugins may return whatever chunks of data are convenient, even if they contain extraneous data, so long as the sum total fully covers the user's selection. Data unrelated to the user's request are filtered out in the next step.

5) Results Filtering and Patching: As `datablocks` are produced by the transform plugin, their contents must be applied to user's results. As shown by the *PG-local result "patches"* in Figure 5b, there may be many `datablocks` returned, which may include data outside of the original selection. This process is handled by a set of optimized "patching" functions within the framework, which take as input a source *datablock* (from the plugin), a target *datablock* (the user's results buffer), and the intersection region. Each datablock may contain a selection of a different type (bounding box or point list), so an optimized version of the patching function is required for each combination.

Once all `datablocks` have been produced by the transform plugin and properly patched into the results, the user's request is completed (i.e., the user's result buffer is full or all chunks have been returned, depending on result mode).

V. Results

Our primary contribution, adding data transformation services to ADIOS, is demonstrated by successful use of data transforms in ADIOS, which includes the experiments that follow. The explanations accompanying these tests attempt to point out how specific goals, such as runtime configuration and user transparency, are realized in practice.

Additionally, in the course of building and testing the transform framework, we have added several data transform technologies as plugins, including:

- zlib, bzip2, szip, ISOBAR [2] (compression)
- APLOD [7] (level-of-detail)
- ALACRITY [10] (indexing)
- PARLO [8] (layout optimization)

While only some of these appear in our evaluations below for brevity, this range of plugins gives an indication of the flexibility of the data transform framework. Note that the transform framework version used in this paper can be found on GitHub [11], with additional library code needed by some of the transforms found on our website [12].

However, some aspects of the data transform framework do lend themselves to quantitative analysis. Specifically, the measurement of overhead induced by the transform framework itself (writes and reads), the behavior of reads over transformed data, and the effect of read-optimizing transforms, are each examined in the following subsections. First, though, we describe our experimental setup.

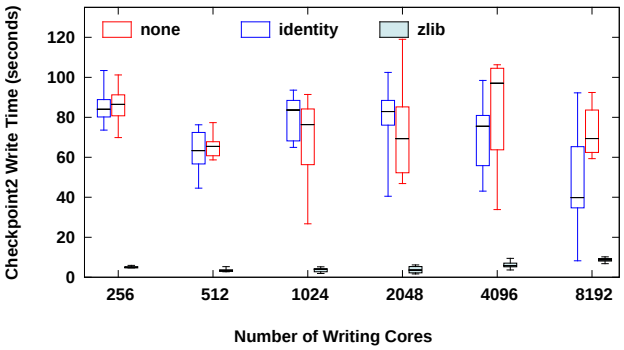


Figure 7. Total I/O time for a QLG simulation checkpoint operation when using different transforms. In this box-and-whiskers plot, the box spans the interquartile range, the line within the box sits at the median, and the "whiskers" extend to the min and max timings.

A. Experimental Setup

The write-related tests are performed on the Titan supercomputer at Oak Ridge National Lab. Titan consists of 18,688 16-core compute nodes (299,008 cores), with 32GB of RAM per node. Additionally, each node houses an nVidia Tesla K20 GPU with 6GB of RAM, although these are not used in our experiments. To provide a real-world scenario for write testing, we run the large-scale Quantum Lattice Gas (QLG) simulator [26], which uses ADIOS for its checkpointing process.

The read-related tests are performed on the Sith development cluster at ORNL. We use data from a large-scale S3D combustion simulation run. The whole dataset is roughly 22TB; however, varying the PG size and the data transforms in our experiments requires maintaining many copies of the data, which is untenable at this scale. Therefore, we extract and use the first 4 timesteps of the "temp" temperature variable, a subset of approximately 50GB. Due to the partitioning of variables and timesteps in ADIOS files, we believe performance on this subset to be indicative of that for cases with more variables and/or timesteps.

All experiments utilized the shared Lustre parallel filesystem, with default Lustre configuration parameters (4 OST stripe count, 1MB stripe size). Also of note, all experiments were performed under the "Widow" Lustre filesystem, prior to the upgrade to "Atlas" in late 2013.

B. Framework Overhead (Write and Read)

In practice, evaluating the overhead of a transform framework itself is non-trivial, as any transform plugin used will necessarily contribute to the total run time. Therefore, we

Table I
STORAGE FOOTPRINT OF CHECKPOINT FILES UNDER DIFFERENT DATA TRANSFORMS. FILE SIZES ARE GIVEN IN GIGABYTES.

Cores	256	512	1024	2048	4096	8192
none	61.04	61.04	61.04	61.05	61.08	61.15
identity	61.04	61.04	61.04	61.05	61.08	61.16
zlib	0.96	0.97	0.94	1.55	1.58	1.59

have developed a special "**identity**" transform plugin, which does not modify data passed through it (thus inducing negligible overhead, yet fully exercising the transform framework by virtue of being treated as a "black box." For fairness, the identity transform leverages our flexible read model to enact the same data sieving policy used by the read transport layer on non-transformed data.

Our first experiment evaluates write performance with a large-scale run of the QLG simulation, which uses ADIOS to write checkpoint files. We vary the number of cores used between 256 and 8,192, and evaluate with transforms "none" (no transform) and "identity", as well as the "zlib" compression plugin to show that realistic transforms work at scale. QLG is configured to write approximately 61GB per checkpoint, a total data size that is kept constant as the core count varies; i.e., strong scaling is used.

Note that *no modifications to the simulation code were required* to apply data transforms in this experiment, and different transforms were selected via a one-line edit in the ADIOS XML file. Also, with respect to I/O decoupling, the results below are based on using the `MPI_LUSTRE` transport method (a collective MPI-IO method with data-alignment algorithms tuned for Lustre), but we have also successfully run this application with the `MPI` and `MPI_AGGREGATE` I/O methods, again without any application code changes.

The checkpoint I/O timings are shown in Figure 7. Each data point is derived from 9 repetitions in order to give a more statistically-significant result. During our tests, the `MPI_LUSTRE` method seemed to experience one-time overhead at the first write operation (with or without transforms enabled), which would have skewed our comparison. Since our interest is in evaluating transform overhead, not I/O transport performance, we simply use the second checkpoint timings instead, which do not exhibit this effect. We use the quartile-based box-and-whiskers plot format due to the I/O variability at this scale. In the plot, each box represents the interquartile range (25th to 75th percentile), with a line at the median, and a "whisker" at the min and max timings.

A few trends are evident in these plots. For lower core counts, write time is quite consistent between "none" and "identity," implying no discernible overhead. Variability increases for larger core counts (as expected, since there is more room for outliers/stragglers), making the comparison less clear, but there is no clear indication of overhead with identity. The "zlib" transform, in contrast, exhibits much lower and more consistent write times than none/identity.

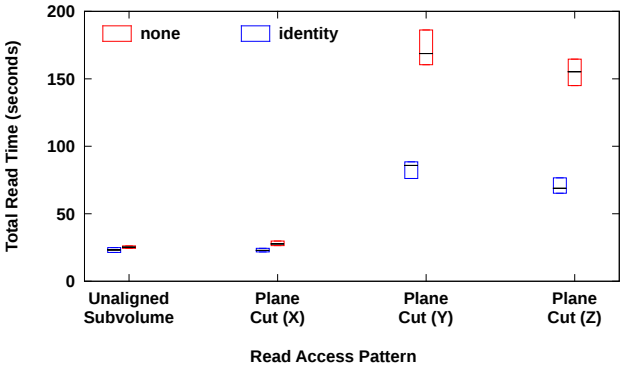


Figure 8. Total read time over S3D data under different transforms. In this box-and-whiskers plot, the box spans the interquartile range and the line within the box sits at the median.

This is due to the highly-compressible nature of the QLG data (as evident in Table V-B), which greatly reduces I/O. While other scientific data are typically less compressible, there is still the potential for I/O reduction to outweigh compression time in some cases.

To test overhead on the read side, we construct another experiment as follows. Starting from S3D combustion data, we generate a no transform ("none") and "identity" transform copy, then compare their performance under different read access patterns. In order to test parallelism on the read side, all tests are performed using 64 reader cores.

The read performance results are given in Figure 8. The test for each datapoint includes 500 randomly-generated read selections (with the same random seed being used for each transform type to ensure fairness). The entire experiment was repeated 30 times, with the interquartile range and median for each configuration plotted.

We anticipate similar performance between "none" and "identity," yet the results show "identity" outperforms "none" in the Y/Z plane cases. Both "identity" and "none" induce the same number of seeks and bytes read, and submit I/O requests in the same order. The experimental results were consistent across multiple reruns on different days, with and without the same Lustre OSTs enforced for both "none" and "identity" files, and with careful attention to eliminating cross-test caching effects (various parameter iteration orders were tested). Total CPU time is roughly the same (at most 1-2% overhead relative to total read time for "identity").

Having ruled out obvious causes, one possible explanation is that, while both "none" and "identity" perform similar data selection computations, the distribution relative to I/O calls differs. While "none" performs these computations *within* the I/O loop, "identity" invokes a different codepath that performs them as pre-/post-processing steps, with a tight, relatively-computeless I/O loop in between. We believe this difference causes "identity" to experience better filesystem caching and responsiveness due to rapidity of I/O requests.

However, users would be advised not to try using "identity" in the hopes of increased read performance.

C. Read Performance of Transformed Data

In any general data transform framework within an I/O middleware, there must enter the concept of "chunking": the global dataspace is partitioned into "chunks" in some fashion, and each chunk is encoded separately by the transform. The only I/O middleware known to the authors with any form of built-in data transform support, HDF5 with its "filters", follows this rule, as do we here.

Based on this general principle, we can make some inferences about read performance over data transformed. For instance, in general, increasing chunk sizes in a file improves the performance of most access patterns by reducing seeks. However, for data transforms that are "opaque" (i.e., where a chunk subset cannot be retrieved without reading/decoding the entire chunk), we expect a counteracting performance penalty as chunk size increases. This implies a tipping point, where the benefit of larger chunks due to larger contiguous access is outweighed by the penalty of reading more and more irrelevant data. In contrast, transforms that are "transparent" (i.e., where a chunk subset may be retrieved by reading/decoding *less* than the entire chunk), this performance penalty is reduced or eliminated, thus propagating the benefit of larger chunks.

Read read access patterns also play a role here. Depending on the underlying linearization of the data, some access patterns (such as plane cuts perpendicular to the slowest-varying dimension) yield far more contiguous access than others (such as plane cuts on another dimension). For non-transformed and transparently-transformed data underpinned by such a linearization, non-contiguous access patterns will degrade performance at larger chunk sizes, because the number of seeks between non-contiguous segments of pertinent data will increase. This effect does not occur for opaque transforms, which already read/decode each touched chunk in its entirety, regardless of access pattern. Thus, the performance gap between transparent and opaque transforms should shrink for less-contiguous access patterns.

Based on these conjectures, we devise an experiment to vary access patterns and vary chunk sizes (i.e., in ADIOS, PG sizes) for different transforms. We perform this test with three access patterns: subvolume, plane X (contiguous), plane Y (non-contiguous); four PG subdivisions of the global space: 2^3, 4^3, 8^3, and 16^3 chunks; and six transforms: none, identity, zlib compression, ISOBAR compression, and APLOD level-of-detail at two precision levels (partial and full precision). Each experiment run consists of 50 reads of a given access pattern, and we perform 10 repetitions of each experiment configuration and average the results.

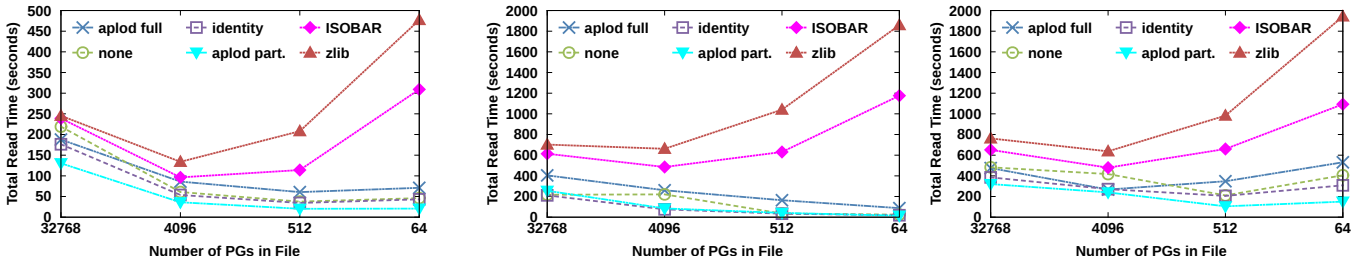
Among these six transforms, four are considered to be "transparent" (none , identity, and both APLOD modes) and two "opaque" (zlib and ISOBAR, both being whole-PG compression techniques). The results are shown in Figures 9a, 9b, and 9c. The trends inferred above can be observed in action, as well as a few other insights:

1) None and identity perform very similarly, implying negligible overhead by the transform framework.
2) The transparent transforms (none, identity, and APLOD) show improved performance with larger chunk size on mostly-contiguous access patterns (subvolume and plane X), but less so for non-contiguous access patterns (plane Y).
3) For the opaque transforms (zlib, isobar), on the other hand, varying chunk size induces the tipping point speculated above. Performance is also better on the subvolume pattern, which has a higher data-requested to data-in-touched-PGs ratio, reducing the amount of unnecessary data read.
4) Between plane X and Y access patterns, which read roughly the same amount of data, the opaque transforms perform similarly, whereas the transparent transforms perform far better on plane X, which is contiguous relative to the data linearization.
5) ISOBAR outperforms zlib due to superior compression ratio and decompression throughput on the hard-to-compress S3D data.
6) Partial-precision APLOD reads *outperform* none/identity, since the reduced precision enables fewer bytes to be read from storage, whereas full-precision APLOD reads perform slightly slower than none/identity, as they are not completely transparent, and the APLOD data reconstitution process requires some computation.

VI. Discussion and Future Work

In this paper, we address the need for integrated *data transforms* in I/O middleware. Through our experience with this project, however, we see I/O middleware as a potential platform for an even greater array of *data services* in the future. We perceive at least three fundamental types of data services: *data transforms*, *I/O transports*, and *data derivatives*. Two of these are now possible in ADIOS: *data transforms* are presented in this paper, and *I/O transports* (services for efficiently moving, storing, and retrieving data) are already supported by ADIOS's modular I/O layer. However, the third service class, *data derivatives* (services for deriving new objects, features, and information from existing data), are not currently available in I/O middleware as transparent, in situ, I/O-decoupled services in a fashion similar to data transforms.

We believe developing an I/O middleware framework for in situ data derivatives to be another area of great potential. Data derivative services include such active areas of research as in situ analytics [19], [20], feature extraction [27], visualization [28], [29] and non-clustered indexing [30]. Seamless data derivative services in I/O middleware, analogous to the



(a) **Subvolume** access pattern (b) **X plane cut** access pattern (c) **Y plane cut** access pattern

Figure 9. Read performance over S3D data under different access patterns and transform methods. "aplod full" and "aplod part" refer to APLOD in full-precision and partial-precision read modes, respectively.

data transforms described in this paper, could similarly aid the widespread adoption of these innovative technologies.

Also based on our experience here, it has become apparent that some complex data services span beyond this basic trichotomy of data service classes. For instance, the DIRAQ indexing workflow [31] incorporates both indexing/compression (a data transformation) and in-network index merging and I/O aggregation (an I/O transport). Similarly, ISOBAR hybrid [32] overlaps compression (data transform) with I/O transport of incompressible data. In both cases, the data transform and I/O transport components cannot be distilled as sequential phases, but are intrinsically linked.

Data services like these offer a range of performance benefits to applications, yet their cross-cutting nature makes fitting them into a single framework difficult. Thus, another direction for future work would be to explore a generalized data service model that supports having a single technology span the entire data pipeline in a scientific workflow. In the context of such an approach, our data transform layer would be a valuable building block by providing a data model, plugin API, and modular pipeline stage for data transforms.

VII. Conclusion

To bridge the gap between innovative data transformation methods and real-world adoption, we present a framework for in situ data transformation in scientific applications through the ADIOS I/O middleware. The framework supports data transformation services in a user-transparent and runtime-configurable manner, has decoupled I/O transport and data transformation services, and realizes read-optimizing transforms through a flexible read model.

We evaluate the data transform framework with the large-scale QLG scientific simulation at up to 8,192 cores, as well as through parallel read tests, with no significant overhead. We also explore the general performance implications of data transformations with respect to chunk size, access pattern, and transform "read opacity".

Acknowledgment

The authors would like to thank Dr. Min Soe from Rogers State University for providing access to their Quantum Lattice Gas (QLG) simulation code, and for assisting with its configuration for use in performance testing our system.

We would like to acknowledge the use of resources at the Leadership Computing Facilities at Oak Ridge National Laboratory, OLCF. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E.. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research and the U.S. National Science Foundation (Expeditions in Computing and EAGER programs).

References

[1] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. Samatova, "Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data," in *Euro-Par Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 6852, 2011, pp. 366–379.

[2] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova, "ISOBAR preconditioner for effective and high-throughput lossless data compression," in *Proc. International Conference on Data Engineering (ICDE)*, 2012, pp. 138–149.

[3] M. Adams and R. Ward, "Wavelet transforms in the JPEG-2000 standard," in *Proc. Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, vol. 1, 2001, pp. 160–163 vol.1.

[4] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245 –1250, 2006.

[5] M. Burtscher and P. Ratanaworabhan, "FPC: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2009.

[6] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2001, p. 2.

[7] J. Jenkins, E. Schendel, S. Lakshminarasimhan, T. Rogers, D. A. Boyuka, S. Ethier, R. Ross, S. Klasky, and N. F. Samatova, "Byte-precision level of detail processing for variable precision analytics," in *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[8] Z. Gong, D. Boyuka II, X. Zou, Q. Liu, N. Podhorszki, S. Klasky, X. Ma, and N. Samatova, "PARLO: PArallel Runtime Layout Optimization for scientific data explorations with heterogeneous access patterns," in *Proc. Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[9] K. Wu, "FastBit: An efficient indexing technology for accelerating data-intensive science," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 556, 2005.

[10] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, N. Shah, E. R. Schendel, S. Ethier, C.-S. Chang, J. H. Chen, H. Kolla, S. Klasky, R. B. Ross, and N. F. Samatova, "Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying," in *Database and Expert Systems Applications (DEXA) (2)*, 2012, pp. 16–30.

[11] "Transforms framework on github," https://github.com/ornladios/ADIOS.git, branch `ncsu-transforms-ccgrid`.

[12] http://www.freescience.org/cs.

[13] G. Vahala, M. Soe, B. Zhang, J. Yepez, L. Vahala, J. Carter, and S. Ziegeler, "Unitary qubit lattice simulations of multiscale phenomena in quantum turbulence," in *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 1–11.

[14] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *Proc. Workshop on Array Databases (AD)*, 2011.

[15] 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 *Proc. Workshop on Challenges of Large Applications in Distributed Environments (CLADE)*, 2008, pp. 15–24.

[16] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A high-performance scientific I/O interface," in *Proc. High Performance Computing, Networking, Storage and Analysis (SC)*, 2003, p. 39.

[17] P.-S. Yeh, W. Xia-serafino, L. Miles, B. Kobler, and D. Menasce, "Implementation of CCSDS lossless data compression in HDF," in *Proc. Earth Science Technology Conference*, 2002.

[18] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *Proc. Cluster Computing (CLUSTER)*, 2012, pp. 155–163.

[19] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA–preparatory data analytics on peta-scale machines," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[20] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: scalable data staging services for petascale applications," *Cluster Computing*, vol. 13, pp. 277–290, 2010.

[21] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky, "Just in time: adding value to the IO pipelines of high performance applications with JITStaging," in *Proc. High-performance Parallel and Distributed Computing (HPDC)*, 2011, pp. 27–36.

[22] Z. Gong, T. Rogers, J. Jenkins, H. Kolla, S. Ethier, J. Chen, R. Ross, S. Klasky, and N. F. Samatova, "MLOC: Multi-level layout optimization framework for compressed scientific data exploration with heterogeneous access patterns," in *Proc. International Conference on Parallel Processing (ICPP)*, 2012.

[23] M. Franklin, A. Halevy, and D. Maier, "From databases to dataspaces: a new abstraction for information management," *SIGMOD Record*, vol. 34, no. 4, pp. 27–33, 2005.

[24] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu, "Six degrees of scientific data: Reading patterns for extreme scale science IO," in *Proc. High Performance and Distributed Computing (HPDC)*, 2011, pp. 49–60.

[25] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, K. Schwan, and M. Wolf, "...and eat it too: High read performance in write-optimized HPC I/O middleware file formats," in *Proc. Petascale Data Storage Workshop (PDSW)*, 2009.

[26] G.Vahala, J. Yepez, M. Soe, B. Zhang, and S. Ziegeler, "Poincare recurrence and spectral cascades in 3D quantum turbulence," *Phys. Rev.*, 2011.

[27] F. Zhang, S. Lasluisa, T. Jin, I. Rodero, H. Bui, and M. Parashar, "In-situ feature-based objects tracking for large-scale scientific simulations," in *Proc. 2012 SC Companion: High Performance Computing, Networking, Storage and Analysis (SCC)*, 2012, pp. 736–740.

[28] H. Yu, C. Wang, R. Grout, J. Chen, and K.-L. Ma, "In situ visualization for large-scale combustion simulations," *Computer Graphics and Applications, IEEE*, vol. 30, no. 3, pp. 45–57, 2010.

[29] K.-L. Ma, "In situ visualization at extreme scale: challenges and opportunities," *Computer Graphics and Applications, IEEE*, vol. 29, no. 6, pp. 14–19, 2009.

[30] J. Kim, H. Abbasi, L. Chacon, C. Docan, S. Klasky, Q. Liu, N. Podhorszki, A. Shoshani, and K. Wu, "Parallel in situ indexing for data-intensive computing," in *Proc. Large Data Analysis and Visualization (LDAV)*, 2011, pp. 65–72.

[31] S. Lakshminarasimhan, D. A. Boyuka II, S. V. Pendse, X. Zou, J. Jenkins, V. Vishwanath, M. E. Papka, and N. F. Samatova, "Scalable in situ scientific data encoding for analytical query processing," in *Proc. High-performance Parallel and Distributed Computing (HPDC)*, 2013.

[32] E. R. Schendel, S. V. Pendse, J. Jenkins, D. A. Boyuka, Z. Gong, S. Lakshminarasimhan, Q. Liu, S. Klasky, R. Ross, and N. F. Samatova, "ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization," in *Proc. High-performance Parallel and Distributed Computing (HPDC)*, 2012.