Interfacing HDF5 with A Scalable Object-centric Storage System on Hierarchical Storage

Jingqing Mu*, Jerome Soumagne*, Suren Byna†, Quincey Koziol†, Houjun Tang†, Richard Warren*
*The HDF Group, Champaign, IL 61820
†Lawrence Berkeley National Laboratory, Berkeley, CA 94720
Email: *{kmu, jsoumagne, richard.warren}@hdfgroup.org, †{htang4, sbyna, koziol}@lbl.gov

***Abstract*—Object storage technologies that take advantage of multi-tier storage on HPC systems are emerging. However, to use these technologies, applications now have to be modified significantly from current I/O libraries. HDF5, a widely used I/O middleware on HPC systems, provides a Virtual Object Layer (VOL) that allows applications to connect to different storage mechanisms transparently without requiring significant code modifications. We recently designed the Proactive Data Containers (PDC) object-centric storage system that provides the capabilities of transparent, asynchronous, and autonomous data movement taking advantage of multiple storage tiers—a decision that has so far been left upon the user on most current systems. To enable PDC's features through HDF5 without modifying application codes, we have developed an HDF5 VOL connector that interfaces with PDC. We present in this paper the connector interface and evaluate its performance on Cori, a Cray XC40 supercomputer located at the National Energy Research Scientific Computing Center (NERSC). Our evaluation demonstrates up to an $8\times$ improvement compared to HDF5 that has the most recent optimizations.**

***Keywords*—asynchronous transfer; data handling; large scale systems; memory management; object-centric models; HDF5;**

I. Introduction

The challenges for scientific applications on upcoming HPC systems when rapidly moving toward exascale, are known from three directions: extreme parallelism, a deepening heterogeneous memory hierarchy, and massively increasing data by volume and complexity. Current data management and I/O technologies present severe limitations in this regard: the POSIX and MPI I/O standards that are the basis for existing I/O libraries and parallel file systems have fundamental restrictions in the areas of scalable metadata operations, semantics-based data movement, performance tuning, asynchronous operations, and scalable consistency of distributed operations—such that simple and efficient methods of data management that can address these challenges are critical for running scientific applications on future HPC systems.

In particular for I/O libraries, one of the challenges to address the diverse performance characteristics of deep storage hierarchy expected in exascale systems is the capability and efficiency of data movement across storage levels. New architectures are considered to contain multiple layers of storage, such as NVRAM on compute nodes, SSD-based burst buffers shared by compute nodes, and disk or SSD-based parallel file system, and tape-based archival storage. Typically, parallel file systems are unaware of multi-level storage hierarchy and need external middleware to manage the hierarchy together. Traditional HPC data management and movement solutions were designed for simpler systems, which are designed to manage each storage layer separately; similarly, scientific data models were designed for a two-tiered storage hierarchy. Another critical deficiency of traditional file systems is metadata management, where files are managed with a small amount of prescriptive metadata leading to the performance bottleneck at metadata servers to locate the files.

Object-based storage systems provide semantics that have the potential to reduce the complexity of storage systems as well as to improve performance. We have developed a user-space object-centric storage and data management system, called Proactive Data Containers (PDC) [1] [2]. PDC provides scalable distributed metadata management [1] and the capabilities of transparent, asynchronous and autonomous data movement to multiple storage tiers [2]. PDC offers a programming interface that applications can use in order to take advantage of the data and metadata management services. A PDC *container* is a container that may reside in a single storage layer (i.e., memory, burst buffer, disk) or span across multiple layers. It contains both the metadata and data objects. The PDC system provides an interface for creating, updating, retrieving, and deleting data objects and for managing metadata on those objects. It is moving us away from existing file-oriented methods, and instead bringing us to exploring novel object-oriented data management methods in an autonomous way.

HDF5 has been widely used in the context of HPC and big data as an I/O middleware capable of supporting extreme scale and complex data structures. HDF5's Virtual Object Layer (VOL) is a storage abstraction layer within the HDF5 library that is designed to target different storage mechanisms while preserving HDF5 objects metadata. The VOL design allows applications to connect to different storage mechanisms transparently without significant code modifications. Several HDF5 VOL connectors have been developed, for instance, PLFS [3], Data Elevator [4], or more recently DAOS [5]—offering HDF5 applications an easy way to use data storage systems transparently with a significant I/O performance increase over POSIX I/O.

Toward making PDC services available to legacy HDF5 applications with minimal source code changes, we implement and present in this paper an HDF5 VOL connector interface

to PDC. This paper and contributions that it presents address the following objectives:

1) Enabling object-oriented storage through HDF5 APIs and library;
2) Enabling implicit and asynchronous data movement through existing HDF5 APIs with minimal code modification;
3) Implementing data movement strategies using TCP and Cray GNI transports;
4) Evaluating and demonstrating an object-centric HPC storage system for scientific use cases using HDF5 APIs.

This paper is organized as follows: we first discuss related work in Section II, and then introduce the object based PDC system in Section III, which enables asynchronous data movement. In Section IV we focus on the HDF5 virtual object layer and provide details of our PDC VOL connector implementation. In section V, we provide experimental results evaluating new methodology in HDF5 utilizing I/O patterns representative of science applications on HPC systems. We conclude our work in Section VI.

II. Related Work

POSIX I/O [6] is known for describing the file access API, data model, and data consistency semantics. Current parallel file systems, such as PVFS [7], [8], Lustre [9], GPFS [10], and NFS [11] were all designed to comply with the POSIX I/O standard. As the original POSIX I/O design was not intended for highly concurrent programming models, which are common in HPC systems [12], that design has now become a performance bottleneck. With an increasing number of memory storage layers and complexity of storage system interactions, the issue of I/O performance is getting imperative and significantly hinders the overall performance of applications [13], [14]. Research efforts have been made to relax the POSIX semantics and alleviate the I/O bottleneck from high-level libraries (e.g., HDF5 [15], netCDF [16], ADIOS [17]), I/O middleware (e.g., MPI-IO [18], TAPIOCA [19]), to I/O forwarding layers [20], all of which provide an array-based data model to organize the data and define data access semantics. However, deep memory and storage hierarchy introduced into modern supercomputer systems further increase POSIX I/O limitations.

Object-based storage has been proposed [21] [22] [23] [24] to overcome the limitations of current parallel file systems, which has long been considered a potential solution for managing rich metadata in scalable environments. It describes an abstract data container that consists of many byte-streams (or objects), each with related attributes. RADOS [25], Amazon S3 [26], and OpenStack Swift [27] have been developed for managing data as objects and storing them in a flat namespace. DAOS [28] is an object-based file system solution currently under development, which provides asynchronous data movement and manages objects in a hierarchical storage with multiple layers, whose scalability is still under evaluation and the features are in development [5]. Furthermore, efforts to implement object-based storage is attempted on individual layer separately, i.e., on disks, in NVRAM, and in memory. SSDUP [29] proposed to redirect data write to burst buffer when it detects random accesses for potential high latency if writing to HDDs. The data in burst buffer is later flushed to HDDs when size is over half of the burst buffer capacity.

Data Elevator [4] provides automatic caching and data movement across multiple levels of storage hierarchies. It uses shared burst buffer as a caching layer before writing data to file system. UniviStor [30] integrates hierarchical and distributed storage devices into a unified view of memory distributed on compute nodes and storage layers in heterogeneous HPC storage and achieves better performance than Data Elevator. Both Data Elevator and UniviStor keep the supporting file format of the data management software that they use and wait for the data to be written to persistent storage. Towards object-centric hierarchical storage system, *Proactive Data Containers* [1] takes advantage of deep storage hierarchies, and provides efficient strategies to support autonomous and asynchronous data management by the PDC service, as well as targeting deep storage hierarchies [2].

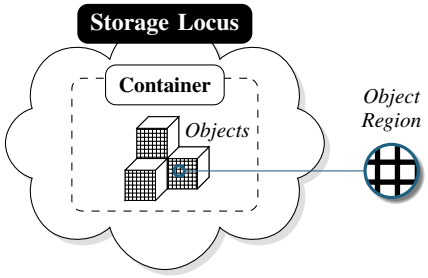
III. PDC System Architecture

In this section, we provide an overview of the *Proactive Data Containers* system and focus on the capabilities and semantics that it provides. For full details of the PDC system implementation, please refer to our previous publications [1], [2].

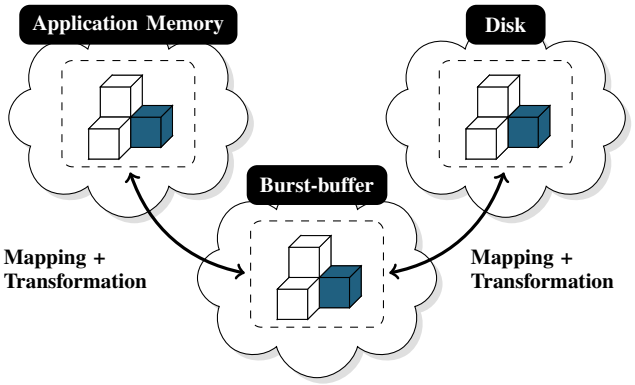
A. PDC Data Model and Semantics

As shown in Figure 1a, PDC organizes data as a set of *objects* within a *Container*. *Object* is a generic term to describe byte streams in an abstract manner. Parts of objects are described using the term *Regions*, where the actual data as well as the metadata associated to it is stored. Region is the basic and fine-grain unit for data movement operations in PDC—these operations are further described in more details. Additionally, all of the previously mentioned entities include *Properties*, regarded as metadata, which contain the descriptive information that is set by the user, or generated by PDC. The properties contain prescriptive metadata such as the data type and dimensions that describe an object and provenance such as user and application information. These objects are managed by PDC services and can be placed at any level of the storage system, and hence, containers, which consist of scientific data, are abstracted within the entire storage stack. This approach spreads the data over different locations within a storage level, which we reference as *storage locus*. As shown in Figure 1b, this representation is also augmented by two types of operations: *object mapping* operations between a memory buffer and an abstract PDC object; and *transformation* operations while data is being moved from one storage locus to the other.

We introduced the concept of *object mapping* in PDC in [2]. As opposed to explicit read and write semantics, object mapping makes data movement operations *implicit* to the user by defining a map operation (i.e., an established memory to



(a) High-level representation of the PDC data model with container, object, and region relationship.



(b) Containers and objects within them can be mapped to one another temporarily or permanently, and have transformations occur during I/O.

Fig. 1: PDC Representation. Abstracted data can reside at any level of the storage hierarchy.

storage relationship with a PDC entity). Similar to POSIX `mmap` semantics, where a file can be mapped to a region of memory, PDC's mapping operations allow PDC object regions to be accessed just like an array in a program. All the user needs to do is to create a mapping between a region within an applications memory and a region within a global PDC object. Once a mapping is established, data movement can occur to keep updates globally visible. However, as opposed to standard POSIX `mmap` operations, concurrent access can and is expected to occur. Therefore, PDC applications are required to use an explicit lock operation on the object before modifying its associated memory and to release that lock when the modification is done. Unlocking a region allows the PDC system to start data movement and to globally propagate the modified data.

Data movement and I/O in PDC is realized *asynchronously*. Once the data has been transferred to a storage locus, further transfers to deeper levels of the storage hierarchy can be realized by PDC without the need for an application to wait for their completion. This capability provides the opportunity for applications to overlap computation with I/O operations and we can also make the safe assumption that data that is written to deeper storage tiers will always fit into that tier, memory representing the lower level and disk representing the higher level. It is worth noting though that application's buffers, which are mapped, can only be re-used and modified once a lock is re-acquired, hence when the transfer to the first level of storage hierarchy has completed. Figure 2 illustrates that mechanism and shows the data flow after a region unlock request has been initiated. If the region is mapped, the data will be first moved from the client's memory to the data server, and once it is safely transferred the region lock can be released. PDC in the meantime though can carry on moving that data to other storage tiers as needed.

B. User-space Client-Server Model

To execute these operations and manage data, PDC uses a client-server model. Designing a client-server middleware for HPC can be a hard process, both in terms of ease of deployment alongside the user's application and in terms

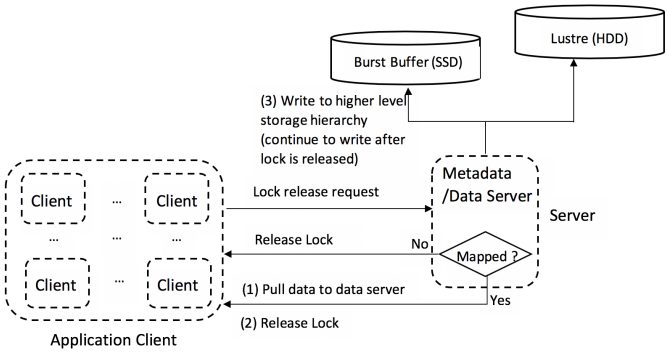


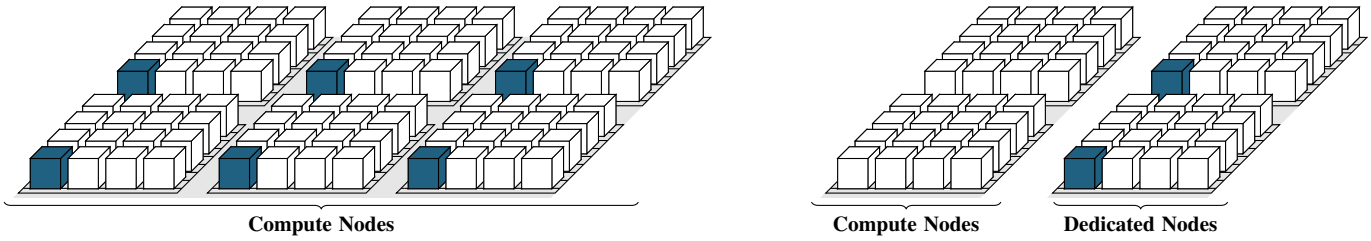
Fig. 2: Data flow through PDC on region unlock requests.

of system resource management. PDC services, though, are designed to run in user space as an additional service process with minimal disruption to the application. In our client-server architecture, PDC servers are responsible for executing both metadata and data management operations.

We have currently implemented two different modes for users to deploy the PDC servers in user-space:

1) **shared mode**, where the server processes run on the compute nodes alongside the client processes and share CPU and memory resources as shown in Figure 3a;
2) **dedicated mode**, where all server processes are placed on dedicated nodes that are separate from the nodes where the client processes are running as shown in figure 3b.

In the first case, the PDC system can take advantage of shared-memory for efficient data movement between node-local clients and servers, while in the second case the PDC system must make use of the native interconnect for high-speed transfers. Users can start any number of PDC servers suitable for the application workload. In shared mode, users are expected to only reserve one core per compute node to run a PDC server while the rest of the cores may be used to run the application processes. In dedicated mode, servers and clients are all allocated to separate nodes, therefore the number of servers used for PDC tasks directly depends on the



(a) Shared server modes, where servers and clients are located on the same node.

(b) Dedicated server modes, where servers are on separate nodes.

Fig. 3: PDC service deployment modes.

user workload and the number of nodes that are available on the system.

IV. Connecting PDC to HDF5

We present in this section the HDF5 VOL connector interface to PDC and its use by applications through the HDF5 API.

A. HDF5 Virtual Object Layer (VOL)

HDF5 is a well-established I/O middleware package, used by a large number of HPC, scientific, and industrial applications. HDF5 provides them with file portability, reliability, and performance when storing their data. By default, the HDF5 library uses its *native* file format when storing data and makes use of MPI-IO to perform parallel I/O, as shown in Figure 4. While this has been a good choice for many years, it also carries on the burden of POSIX I/O semantics and limits that are inherent to the existing native file format, which defines an HDF5 file as a file structure that is contiguously mapped to a file system. For instance, the native file format has a well-known limitation of requiring collective creation of new HDF5 objects, such that the file metadata is ensured to be coherent between processes.

With the emergence of file and storage systems that do not strictly comply with the POSIX I/O standard, new file formats and ways of performing I/O can be defined with additional degrees of freedom. To provide that capability and give developers the ability to store the data in the form of their choice—while preserving the metadata that is attached to the HDF5 objects—the HDF5 library defines a virtual object layer, which will be released in the upcoming 1.12 version of the library. The virtual object layer effectively allows developers to redefine the HDF5 I/O API calls (i.e., related to operations on files, groups, datasets, attributes, etc) by seamlessly re-routing them to the corresponding VOL *connector* backend, which can in turn translate these calls into the operations that it desires to perform.

In the case of PDC and as shown in Figure 4, those operations translate into PDC calls, which in turn interact with the PDC runtime service and PDC storage backends. One of the main advantages of the PDC runtime is that it transparently and automatically provides new capabilities to HDF5 such as asynchronous I/O without requiring any major code change for the application user. One of the difficulties, however, is potentially for the VOL connector developer as the

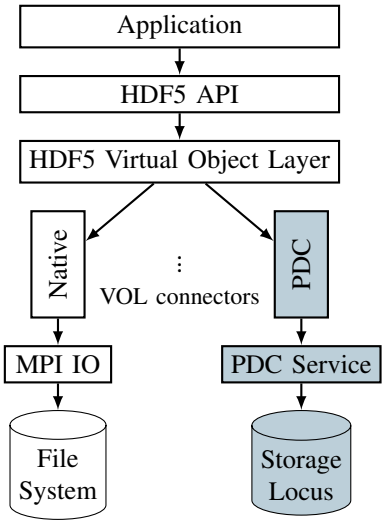


Fig. 4: PDC within Virtual Object Layer. All of the HDF5 I/O related calls are routed to the corresponding VOL connector.

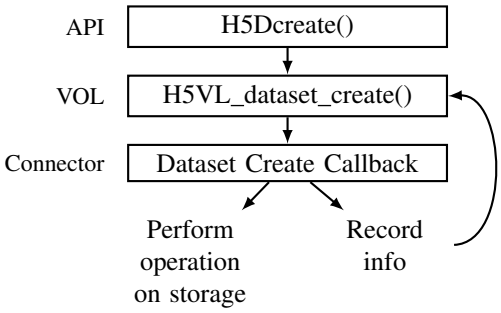


Fig. 5: Dataset create call within the VOL. *Terminal* connectors interface with storage systems while *pass-through* connectors may record info on the fly and re-enter the VOL.

semantics that the underlying layer provides may not always be a direct match with the ones that the HDF5 virtual object layer requires.

Figure 5 shows how the data is accessed in the file through a VOL connector callback. The VOL intercepts all HDF5 API calls that interact with files and reroutes those calls to the associated VOL callback for the requested API function. For example, a call to `H5Dcreate()` would be implemented within the HDF5 library as Figure 5, where the connector is responsible for the actual I/O operation to the storage system.

Similarly, other operations in the HDF5 library follow the same execution pattern.

In order for an application to use an HDF5 VOL connector, one must first register the connector to HDF5 by calling the `H5VLregister_connector()` function. This effectively registers and initializes the VOL connector, which if successfully initialized will return a unique VOL connector ID. That ID can then be passed to the `H5Pset_vol()` routine, which notifies the file access property list of the VOL connector it needs to use when creating or opening the file (since different VOL connectors could be initialized and used within the same application). For convenience, the library also defines environment variables that allow this information to be set by a user and avoids code modification to the application. Once the VOL connector information is set, the application can carry on with regular HDF5 function use.

B. HDF5 PDC VOL Connector Implementation

Our PDC VOL connector currently only implements a subset of the HDF5 API and in this paper we focus on file and dataset operations. HDF5 files can be easily mapped to PDC containers, while HDF5 datasets are naturally mapped to PDC objects and PDC regions are similar in essence to HDF5 selections.

File create, open and close operations are a direct match to PDC container operations, and therefore no particular implementation challenges were faced. Dataset operations, however, differ from PDC's API as the model chosen for PDC was to make data movement implicit, whereas HDF5 chooses to make data movement explicit by providing explicit read and write semantics. We therefore walk through the details of our implementation in that section.

As presented in the pseudo-code below, dataset create and open operations map to `PDCobj_create()` and `PDCobj_open()` respectively.

```
static void *
H5VL_pdc_dataset_create(void *obj, ..., const
  char *name, hid_t dcpl_id, hid_t dapl_id,
  hid_t dxpl_id, ...)
{
  ...
  /* Create a new object */
  dset->obj.obj_id = PDCobj_create(
    o->file->cont_id, name, obj_prop);
  ...
}
```

```
static void *
H5VL_pdc_dataset_open(void *obj, ...,
  const char *name, hid_t dapl_id,
  hid_t dxpl_id, ...)
{
  ...
  /* Open an existing object */
  dset->obj.obj_id = PDCobj_open(
    name, pdc_id);
  ...
}
```

Dataset write and read operations, however, require some extra handling as PDC does not provide explicit read and write semantics. Therefore, in these connectors callbacks, `PDCbuf_obj_map()` is used to first map the region in memory to the PDC object. A pair of lock and unlock calls are then also used, unlock triggering asynchronous data movement. Finally, the region is unmapped using the `PDCbuf_obj_unmap()` call. Once `H5Dwrite()` returns after the unlock call, the data region has been transferred from the application buffer to the PDC data server, and the second step of data movement to further storage level can be taken care of by PDC, allowing further computation to be overlapped.

Write implementation is illustrated in the pseudo code below:

```
static herr_t
H5VL_pdc_dataset_write(void *_dset,
  hid_t mem_type_id, hid_t mem_space_id,
  hid_t file_space_id, hid_t dxpl_id,
  const void *buf, ...)
{
  ...
  // HDF5 selection to PDC region translation
  ...
  PDCbuf_obj_map((void *)buf, mem_type,
    mem_reg, dset->obj.obj_id, obj_reg);
  ...
  PDCreg_lock(dset->obj.obj_id, obj_reg,
    WRITE, NOBLOCK);
  ...
  PDCreg_unlock(dset->obj.obj_id, obj_reg,
    WRITE);
  ...
  PDCbuf_obj_unmap((void *)buf, mem_reg,
    dset->obj.obj_id, obj_reg);
  ...
}
```

Reads are implemented similarly to writes but because PDC differentiates read locks from write locks (as read locks require less constraints than write locks), we pass the `READ` flag to both lock and unlock calls. This is illustrated in the pseudo-code below:

```
static herr_t
H5VL_pdc_dataset_read(void *_dset,
  hid_t mem_type_id, hid_t mem_space_id,
  hid_t file_space_id, hid_t dxpl_id,
  void *buf, ...)
{
  ...
  // HDF5 selection to PDC region translation
  ...
  PDCbuf_obj_map((void *)buf, mem_type,
    mem_reg, dset->obj.obj_id, obj_reg);
  ...
  PDCreg_lock(dset->obj.obj_id, obj_reg,
    READ, NOBLOCK);
  ...
  PDCreg_unlock(dset->obj.obj_id, obj_reg,
    READ);
  ...
  PDCbuf_obj_unmap((void *)buf, mem_reg,
```

```
    dset->obj.obj_id, obj_reg);
  ...
}
```

The dataset is then closed using `PDCobj_close()` as illustrated below:

```
static herr_t
H5VL_pdc_dataset_close(void *_dset,
  hid_t dxpl_id, ...)
{
  ...
  /* Close the object */
  PDCobj_close(dset->obj.obj_id);
  ...
}
```

C. Application Usage Example with the HDF5 PDC VOL

We provide a detailed example of how an application I/O kernel can be modified to support the HDF5 PDC VOL connector in Figure 6 and mark the two extra function calls that are required in order to make use of the PDC VOL connector. As previously mentioned, HDF5 also provides a way of specifying that information through environment variables, and allowing for no code changes.

For an application to use the HDF5 PDC VOL connector for reading a dataset, it simply needs to follow the write example, using `H5Dopen()` instead of `H5Dcreate()`, and then call `H5Dread()` instead of `H5Dwrite()`. All of the function mapping details from HDF5 to the underlying PDC APIs are hidden by the HDF5 VOL design and are once again transparent to the user. The underlying VOL connector internally initiates map, lock, lock release and unmap operations to that PDC object to trigger data movement and enables asynchronous data movement, transparently allowing for data movement to be overlapped by the following application computation step.

V. Experimental Evaluation

We evaluate in this section the performance of the PDC VOL connector and also demonstrate the impact of the number of servers on the performance when PDC is deployed in dedicated mode. We compare the performance of writing multiple time steps using the PDC VOL connector with native HDF5. Lastly, we compare the read performance of the PDC VOL connector with that of native HDF5.

A. Experiment setup

To evaluate the performance of the PDC VOL, we ran the experiments with different configurations. We installed PDC on the Cori supercomputer at the National Energy Research Scientific Computing Center (NERSC), which is a Cray XC40 supercomputer with 1630 Intel Xeon Haswell nodes. Each node consists of 32 cores and 128GB of memory. The supporting storage system, Lustre, has 248 object storage targets (OSTs) and is shared by all users.

We ran the experiments using both shared and dedicated deployment modes. With a shared server and client configuration (shared mode), we have one PDC server on each node, which utilizes one core, leaving the remaining 31 cores for user application execution. In dedicated mode, the PDC servers and user's application are on separate nodes. PDC servers in this configuration have only one server per node that provides both metadata server and data server services. In both configuration cases, we have relied on the Mercury [31] RPC library, an HPC-optimized C library for Remote Procedure Calls, as the communication mechanism. In our experiments, we configure Mercury with two communication protocols using the `libfabric` plugin [32] over TCP and Cray GNI. Note that in the latter case, the PDC server is configured to make use of Cray Dynamic RDMA Credentials (DRC) [33] to allow the user's applications and PDC server to share credentials and communicate together through GNI. GNI job runs are therefore currently a little more complex in terms of deployment. To use Cray GNI on Cori, the PDC server/service has to first acquire a credential and wait for the client application to start. The client application then first contacts the server over TCP so that the job can be granted access and use the DRC token. The generated DRC credential is later passed down to Mercury and used by both server and client sides for execution. Once the communication is established, the server and client can proceed and resume their normal execution. Using GNI currently requires the server and client to be in separate sessions but to start at the same time. This is achieved through the srun `pack-group` option. The job script for that run is attached in Appendix Figure 13, Figure 14 and Figure 15.

We used a plasma-physics application's I/O kernel, called VPIC-IO to evaluate the PDC system's performance. VPIC-IO is extracted from VPIC [34], a code developed for simulating several plasma physics phenomenon, including magnetic reconnection in space weather. In VPIC-IO, each MPI process writes a region of 8M (8×2^{20}) particles and each particle has 8 properties. Each region is represented with a 1-D array with a size of 8M on one process. VPIC data structures use 1-D arrays for representing each property and each property is retreated as an object in our design. We also evaluate the read performance by the BD-CATS I/O kernel [35], which is extracted from a parallel clustering algorithm, used for analyzing the data produced by particle simulations. It reads data generated by VPIC or VPIC-IO using the same I/O trace as the BD-CATS implementation of the DBSCAN algorithm. In this kernel, data related to the particles are read among all of the MPI processes in a load-balanced distribution. The original kernels use HDF5 for performing I/O and are highly tuned using MPI-IO and Lustre optimizations [36], [37]. In this paper, we simply re-use those I/O kernels to make them use the PDC VOL connector instead of going through native HDF5 and MPI I/O. The total data size being accounted for goes from 248*GB* for 992 processes from 32 nodes to 3968*GB* for 15872 processes from 512 nodes.

B. H5Dwrite Performance Comparison

We compare the `H5Dwrite()` performance of VPIC-IO in Figure 7 using the following methodologies: HDF5 collective

```
hid_t pdc_vol_id, file_id, fapl_id;
H5VL_pdc_info_t pdc_vol; // to pass VOL connector info
...
/* Register PDC VOL */
pdc_vol_id = H5VLregister_connector(&H5VL_pdc_g, H5P_DEFAULT); // extra step to use PDC VOL

/* Create a new file access property */
fapl_id = H5Pcreate(H5P_FILE_ACCESS);

/* Set the VOL */
H5Pset_vol(fapl_id, pdc_vol_id, &pdc_vol);  // extra step to use PDC VOL

/* Create file */
file_id = H5Fcreate(argv[1], H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);
...
/* Close property */
H5Pclose(fapl_id);

/* Create dataset */
dset_id1 = H5Dcreate(file_id, "x", H5T_NATIVE_FLOAT, filespace, H5P_DEFAULT, H5P_DEFAULT,
    H5P_DEFAULT);

/* Write the data */
H5Dwrite(dset_id1, H5T_NATIVE_FLOAT, memspace, filespace, fapl_id, x);

/* Close dataset */
H5Dclose(dset_id1);

/* Close file */
H5Fclose(file_id);
```

Fig. 6: Application usage example of the PDC VOL connector. There are three lines of code to the original application code to use the PDC VOL.

I/O, HDF5 independent I/O, PDC VOL in shared server mode, PDC VOL in dedicated server mode using TCP protocol and PDC VOL in dedicated server mode using Cray GNI. The `H5Dwrite()` function using the PDC VOL connector involves the times to map the memory buffer to a remote object and to lock and then release the lock on an object without waiting for data to be flushed to disk, while the native `H5Dwrite()` function is issuing MPI I/O calls directly to the file system. For all the evaluations presented in this section, we run the experiments at least ten times and report best numbers. Since there are 8 properties in VPIC-IO, the `H5Dwrite()` function is called 8 times. The time is measured by adding an MPI barrier before the first call and another after the last call to the `H5Dwrite()` function. The total write time for all the 8 properties is collected and used for evaluation in this section. The x-axis shows the number of client processes with the number of PDC servers (in brackets, in these plots as well as in the remaining plots in this section, unless specified otherwise). The native HDF5 I/O performance (collective and independent) was observed on Cori at our time of experiment, which could vary depending on the system software stack installed and system load.

The performance of the PDC VOL connector in shared server mode is 1.7X to 4.9X faster compared to independent native HDF5 method, with an average of 3.3X, and is 2.9X to

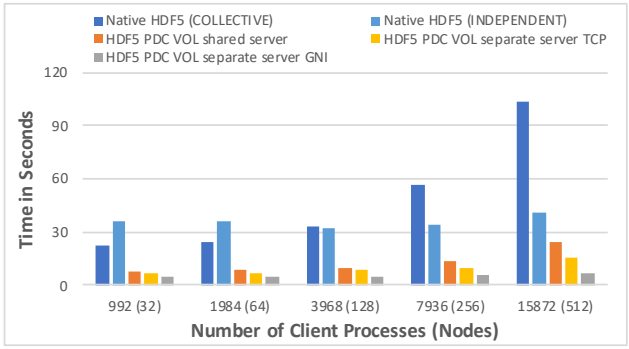


Fig. 7: `H5Dwrite()` performance using different methodologies for one time step. Total data size goes from 248*GB* for 992 processes to 3968*GB* for 15872 processes. The number of PDC servers for each configuration is equal to the number of compute nodes, indicated in parenthesis.

4.2X faster compared to collective native HDF5 method, with an average of 3.5X. In dedicated server mode, with additional nodes utilized as servers, the PDC VOL connector achieves 3.1X to 6.7X faster performance compared to collective HDF5 I/O, with an average of 4.7X, and achieves 3.8X to 4.8X faster performance compared to independent native HDF5, with an average of 4.4X. To further improve the performance, we

evaluate the performance with Cray GNI. It allows 4.6X to 15.6X speedup compared to collective native HDF5, with an average of 8.3X, and allows 6.1X to 7.6X speedup compared to independent native HDF5, with an average of 6.6X. With native HDF5, collective or independent, the data is written directly to the lustre file system. With all the other three PDC VOL asynchronous methods, the data is moved to PDC servers when `H5Dwrite()` returns, allowing for further data movement to the file system to be overlapped with following computation. Figure 7 shows the actual wait time in the `H5Dwrite()` call for the user, and not the total time for data movement down to the file system.

C. Varying Servers in Dedicated Mode

In the previous experiments in dedicated mode, we had a configuration of one server per node and the number of server processes was the same as the number of client nodes. In this section we evaluate the impact of the number of servers on the performance of `H5Dwrite()`. The experiments are run with 992 client processes on 32 nodes and involve varying number of additional nodes, from the number of 4 to 32, for 32 servers sitting. The total size of data to be written is 248*GB*. Figure 8 shows one time step of `H5Dwrite()` time, increasing when less servers are available. More servers would naturally provide more bandwidth and achieve the best performance depending on the system resources availability, though fewer servers are still able to provide a reasonable performance.

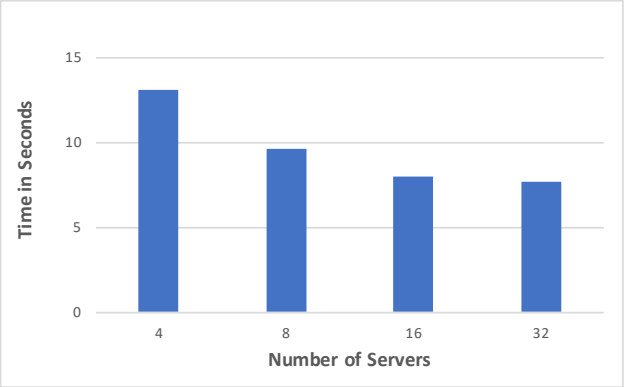


Fig. 8: `H5Dwrite()` performance using dedicated server mode with varying the number of servers. Total data size is 248*GB* for 992 processes.

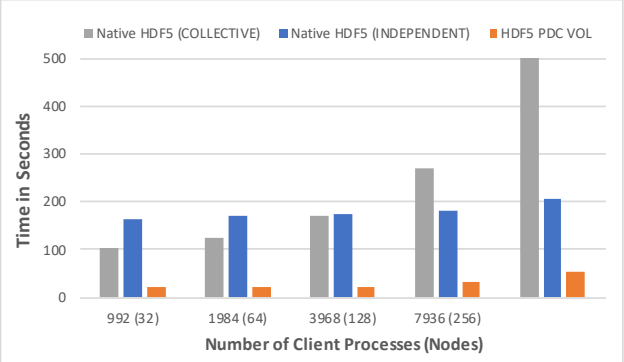


Fig. 9: Total time including `H5Dwrite()` for 5 timesteps and `H5Dclose()`.

D. Multiple Time Steps of H5Dwrite Performance Comparison

We mentioned in section V-B that data movement is still happening after the `H5Dwrite()` call returns, unless the application user chooses to wait for the data to be flushed to disk. We experimented with 5 successive time steps of I/O using `H5Dwrite()` calls for each dataset and show the results in Figure 9. In this case, all data will be flushed between time steps. We use the PDC VOL connector in shared server mode for this experiment and observed 4.8X to 9.5X speedup compared to native HDF5 collective I/O and 3.8X to 8.3X speedup compared to native HDF5 independent I/O, with an average speedup of 7.3X and 6.6X separately.

E. Total Execution Time for VPIC-IO

To reflect the total time consumed by the whole VPIC-IO application, we measured the time from the first HDF5 file create by `H5Fcreate()` until HDF5 file close function `H5Fclose()`. In this experiment we only covered one time step of `H5Dwrite()` for each property within VPIC-IO. The more time steps the application executes, the more performance benefits it gains by utilizing the PDC VOL connector. Figure 10 shows the real total execution time of VPIC-IO covering just one time step of `H5Dwrite()`. We can see that the time for the PDC VOL in shared server mode is 1.5X to 2.9X faster compared to native HDF5 collective I/O, and 1.4X to 2.2X faster than HDF5 independent I/O. For the best case, using a PDC separate server and GNI, it achieves 1.4X to 5.0X and 1.8X to 2.2X performance speedup compared to HDF5 collective and independent I/O respectively. This time reflects when the user chooses to wait for data to be flushed to disk and then exit from the application. If the user chooses not to wait for the data but lets the server working on it, the performance is shown in Figure 11. The performance when letting PDC taking care of the data achieves 2.1X to 3.8X and 1.8X to 3.2X faster performance compared to HDF5 collective and independent I/O methods respectively.

F. H5Dread Performance Comparison

BD-CATS-IO is a read I/O kernel, which reads data produced by VPIC-IO in a load balanced way. In Figure 12, we show the performance of reading a single time step of data that was written in previous VPIC experiments, calling instead the `H5Dread()` function. The PDC VOL connector in shared server mode achieves 1.3X to 1.7X better performance compared to native HDF5 collective I/O and 1.4X to 2.8X better performance compared to native HDF5 independent I/O. The PDC VOL connector in dedicated server mode is 1.8X to 3.4X faster compared to native HDF5 collective I/O, and is 2.7X to 4.6X faster compared to native HDF5 independent I/O. With Cray GNI, the PDC VOL in dedicated server

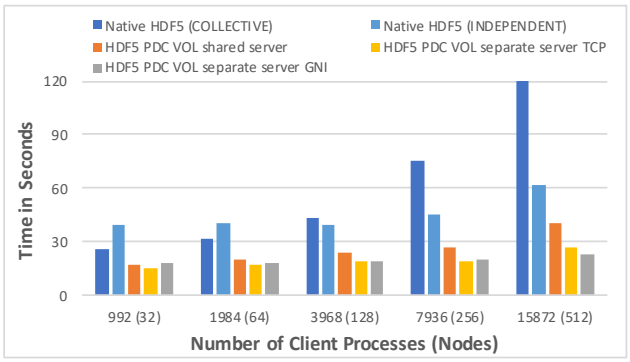


Fig. 10: Total elapsed time for the execution of VPIC-IO.

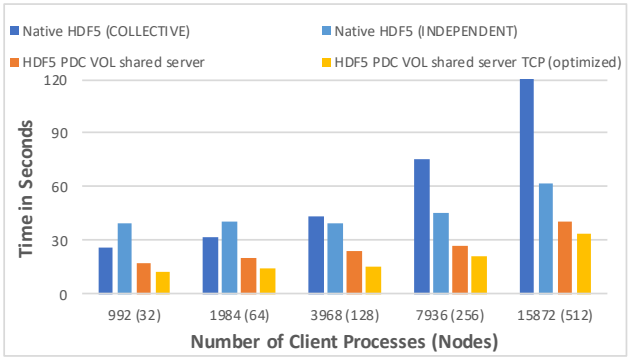


Fig. 11: Total elapsed time for the execution of VPIC-IO if the user chooses not to wait for data to be flushed to disk.

mode executed 2.1X to 5.3X faster compared to native HDF5 collective I/O, and 4.1X to 5.8X faster compared to HDF5 independent I/O. The read performance speedup is not as much as the write performance due to the fact that `H5Dread()` requires the data to be fetched from the PDC backend file system to PDC data server and then tranferred back to the application.

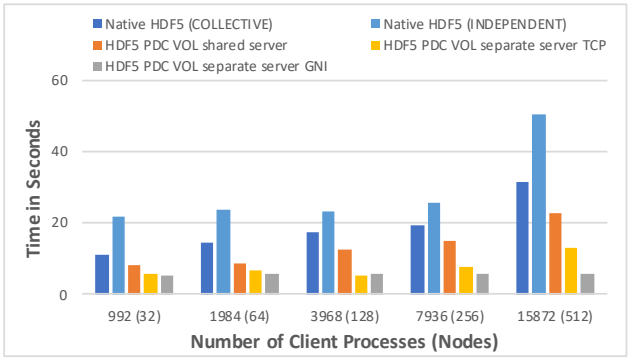


Fig. 12: `H5Dread()` performance using different methodologies for one timestep. Total data size goes from 248*GB* for 992 processes to 3968*GB* for 15872 processes. The number of servers for each setup is the number in parenthesis.

VI. Conclusion and Future Work

In this paper we presented how to take advantage of PDC through HDF5 by developing an HDF5 VOL connector, which enables implicit and asynchronous data movement to different storage tiers with minimal to zero code modification, and is able to be deployed in different scenarios using native network fabric transports such as Cray GNI on modern supercomputers. We also evaluated and demonstrated that interface, which showed a significant performance gain over native HDF5, as file system accesses are no longer issued directly by the application, but are instead handled by the PDC service.

When mapping HDF5 to PDC, one apparent limitation of HDF5 that transpired is its current inability to provide to the application a way of directly exposing the user's memory, as PDC is able to do through map operations. We will in future work study how that type of semantic could be brought into HDF5, allowing users to establish a direct mapping of the application memory to the storage, which similarly to `mmap` operations can allow more efficient transfers and paging to take place and be handled by the PDC system, rather than having users to explicitly direct of the amount data that needs to be written at a given time. This naturally requires applications to adapt their code in order to reflect this type of change, and would hence be more intrusive than the current solution presented in the paper.

Acknowledgment

This work is supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 and DE-SC0016454. This research used resources of the National Energy Research Scientific Computing Center (NERSC).

Appendix

```
srun --pack-group=0 -n 1 drc_server.sh &
sleep 5
srun --pack-group=1 -n 1 drc_client.sh
export PDC_DRC_KEY=`cat $SCRATCH/drc.txt`
....
srun --pack-group=0 -n 32 pdc_server.exe &
sleep 5
srun --pack-group=1 -n 992 h5_pdc_write cc
```

Fig. 13: A sample Slurm job script used to launch a test application with PDC VOL connector using Cray GNI.

```
procIdx=${SLURM_PROCID}
if [ $procIdx -eq 0 ]; then
/mercury/build/bin/hg_test_drc_auth -c ofi -p
    tcp -H ipogif0 -L -a
fi
```

Fig. 14: A sample job script (drc_server.sh), used to obtain dynamic RDMA credentials.

```
procIdx=${SLURM_PROCID}
if [ $procIdx -eq 0 ]; then
/mercury/build/bin/hg_test_drc_auth -c ofi -p
    tcp -H ipogif0 -a
fi
```

Fig. 15: A sample job script (drc_client.sh), which is used to obtain DRC.

In Figures 13, 14, and 15, we provide sample Slurm job scripts for running the PDC VOL using Cray GNI on Cori supercomputer at NERSC. The first script (Figure 13) is used to launch a test application with the PDC VOL using Cray GNI. It has four `srun` commands: Dynamic RDMA creditials (DRC) server, DRC client, PDC server, and the HDF5 application. In Figure 14, we show the script (`drc_server.sh`) to obtain DRC and in Figure 15, we show the script for running the Mercury client to obtain DRC.

REFERENCES

[1] H. Tang, S. Byna, F. Tessier, T. Wang, B. Dong, J. Mu, Q. Koziol, J. Soumagne, V. Vishwanath, J. Liu, and R. Warren, "Toward Scalable and Asynchronous Object-centric Data Management for HPC," in *CCGRID*, 2018.

[2] J. Mu, J. Soumagne, H. Tang, S. Byna, Q. Koziol, and R. Warren, "A Transparent Server-Managed Object Storage System for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2018, pp. 477–481.

[3] K. Mehta, J. Bent, A. Torres, G. Grider, and E. Gabriel, "A plugin for HDF5 using PLFS for improved I/O performance and semantic analysis," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 11 2012, pp. 746–752.

[4] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data Elevator: Low-Contention Data Movement in Hierarchical Storage System," in *HiPC*, 2016, pp. 152–161.

[5] M. S. Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol, "DAOS for Extreme-scale Systems in Scientific Applications," *CoRR*, vol. abs/1712.00423, 2017. [Online]. Available: http://arxiv.org/abs/1712.00423

[6] S. R. Walli, "The POSIX Family of Standards," *StandardView*, vol. 3, no. 1, pp. 11–17, Mar. 1995.

[7] P. Carns, W. Ligon III, R. Ross, and R. Thakur, "PVFS: A Parallel Virtual File System for Linux Clusters," *Linux J.*, 2000.

[8] M. Moore *et al.*, "OrangeFS: Advancing PVFS," *FAST poster session*, 2011.

[9] P. J. Braam, "The Lustre storage architecture," *White Paper*, 2004.

[10] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters." in *FAST*, vol. 2, 2002, pp. 231–244.

[11] S. Microsystems, "NFS: Network File System Protocol Specification," in *RFC*. RFC Editor, 1989.

[12] K. J. Barker *et al.*, "Entering the Petaflop Era: The Architecture and Performance of Roadrunner," in *Supercomputing*, 2008.

[13] M. Jung, W. Choi, J. Shalf, and M. T. Kandemir, "Triple-A: A Non-SSD Based Autonomic All-flash Array for High Performance Storage Systems," *SIGPLAN Not.*, vol. 49, pp. 441–454, 2014.

[14] F. Schürmann and et.al., "Rebasing I/O for Scientific Computing: Leveraging Storage Class Memory in an IBM BlueGene/Q Supercomputer," in *ISC*, 2014, pp. 331–347.

[15] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the HDF5 technology suite and its applications," in *EDBT/ICDT*, 2011, pp. 36–47.

[16] J. Li, W. keng 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 *Supercomputing, 2003 ACM/IEEE Conference*, Nov 2003, pp. 39–39.

[17] Q. Liu, J. Logan, Y. Tian *et al.*, "Hello ADIOS: the challenges and lessons of developing leadership class I/O frameworks," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[18] R. Thakur, W. Gropp, and E. Lusk, "On Implementing MPI-IO Portably and with High Performance," in *ICPADS*, 1999, pp. 23–32.

[19] F. Tessier, V. Vishwanath, and E. Jeannot, "TAPIOCA: An I/O Library for Optimized Topology-Aware Data Aggregation on Large-Scale Supercomputers," in *CLUSTER*, 2017, pp. 70–80.

[20] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. Ross, "Improving I/O Forwarding Throughput with Data Compression," in *CLUSTER*, 2011, pp. 438–445.

[21] A. L. Brown and R. Morrison, "A Generic Persistent Object Store," *Software Engineering Journal*, vol. 7, no. 2, pp. 161–168, March 1992.

[22] J. E. B. Moss, "Design of the mneme persistent object store," *ACM Trans. Inf. Syst.*, vol. 8, no. 2, pp. 103–139, Apr. 1990. [Online]. Available: http://doi.acm.org/10.1145/96105.96109

[23] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison, "Persistent object management system," *Software: Practice and Experience*, vol. 14, no. 1, pp. 49–71, 1984. [Online]. Available: http://dx.doi.org/10.1002/spe.4380140106

[24] G. A. Gibson *et al.*, "A Cost-effective, High-bandwidth Storage Architecture," *SIGPLAN Not.*, vol. 33, pp. 92–103, 1998.

[25] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters," in *PDSW*, 2007, pp. 35–44.

[26] Amazon. Amazon Web Services. Http://s3.amazonaws.com.

[27] J. Arnold, *OpenStack Swift: Using, administering, and developing for swift object storage*. O'Reilly Media, Inc., 2014.

[28] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and Friends: A Proposal for an Exascale Storage System," in *Supercomputing*, 2016, pp. 50:1–50:12.

[29] X. Shi, M. Li, W. Liu, H. Jin, C. Yu, and Y. Chen, "SSDUP: a traffic-aware ssd burst buffer for HPC systems," in *ICS*, 2017.

[30] T. Wang, S. Byna, B. Dong, and H. Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[31] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross, "Mercury: Enabling Remote Procedure Call for High-Performance Computing," in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.

[32] Libfabric. [Online]. Available: https://ofiwg.github.io/libfabric/

[33] J. Shimek and J. Swaro, "Dynamic RDMA Credentials," in *Cray User Group (CUG) Meeting*, 2016. [Online]. Available: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap108s2-file1.pdf

[34] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulationa)," *Physics of Plasmas*, vol. 15, no. 5, 2008.

[35] M. M. A. Patwary *et al.*, "BD-CATS: Big Data Clustering at Trillion Particle Scale," in *Supercomputing*, 2015.

[36] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi *et al.*, "Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation," in *Supercomputing*, 2012, pp. 59:1–59:12.

[37] B. Behzad, S. Byna, S. M. Wild, M. Prabhat, and M. Snir, "Improving parallel I/O autotuning with performance modeling," in *HPDC*, 2014, pp. 253–256.