FINAL TECHNICAL REPORT

DOE AWARD #:	DE-SC0016454
SPONSORING PROGRAM OFFICE:	Department of Energy, Office of Science
RECIPIENT:	The HDF Group 410 E. University Ave., Ste. 200 Champaign, IL 61820
PROJECT TITLE:	Proactive Data Containers for Scientific Storage
PROJECT PI (HDF Group):	Jerome Soumagne
TEAM MEMBERS:	The HDF Group Richard Warren, Jingqing Mu, Neelam Bagha ANL Venkat Vishwanath, Francois Tessier LBNL Suren Byna, Quincey Koziol, Houjun Tang, Teng Wang, Bin Dong, Jialin Liu
PERIOD COVERED (The HDF Group):	May 15, 2016 – December 31, 2019

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under Award Number DE-SC0016454.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Final Report for:
Proactive Data Containers for Scientific Storage

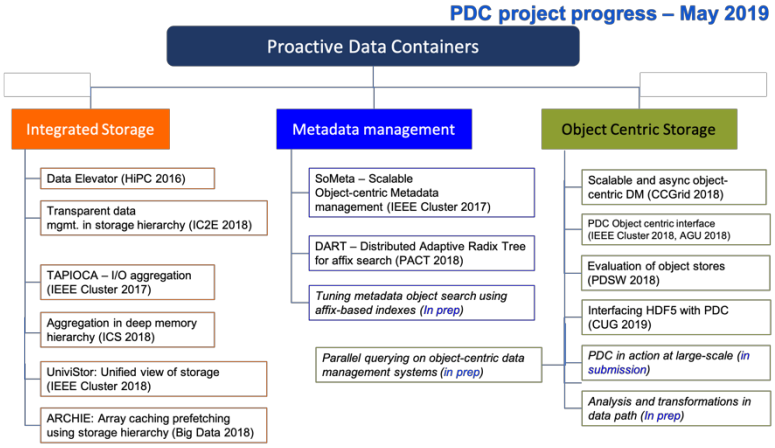
Suren Byna, Quincey Koziol, Houjun Tang, Teng Wang, Bin Dong, Jialin Liu (LBNL)
Jerome Soumagne, Jingqing Mu, Richard Warren (The HDF Group)
Venkat Vishwanath, François Tessier (ANL)

1. Executive Summary

Emerging HPC systems are expected to be deployed with an unprecedented level of complexity, due to a deep system memory/storage hierarchy and heterogeneity of the storage hardware. This hierarchy is expected to range from CPU cache through several levels of volatile memory to nonvolatile memory, traditional hard disks, and tape. Simple and efficient methods of data management and movement through this hierarchy is critical for scientific applications using exascale systems. Existing storage system and I/O (SSIO) technologies face severe challenges in dealing with these requirements. POSIX and MPI I/O standards that are the basis for existing I/O libraries and parallel file systems present fundamental challenges in the areas of scalable metadata operations, semantics-based data movement performance tuning, asynchronous operation, and support for scalable consistency of distributed operations.

Moving toward new paradigms for SSIO in the extreme-scale era, we have proposed to investigate novel object-based data abstractions and storage mechanisms that take advantage of the deep storage hierarchy and enable proactive automated performance tuning. In order to achieve these overarching goals, we initiated an effort to develop a fundamental new data abstraction, called Proactive Data Containers (PDC). A PDC is a container within a locus of storage (memory, NVRAM, disk, etc.) that stores science data in an object-centric manner. Managing data as objects enables powerful optimization opportunities for data movement and transformations. The R&D focus of this project are: 1) formulation of object-oriented PDCs and their mapping in different levels of the exascale storage hierarchy; 2) efficient strategies for moving data in deep storage hierarchies using PDCs; 3) techniques for transforming and reorganizing data based on application requirements; and 4) novel analysis paradigms for enabling data transformations and user-defined analysis on data in PDCs.

Toward achieving these overarching goals, we designed an object-centric application programing interface (API) for HPC, scalable metadata management for object-centric storage systems, and data movement optimizations such as Data Elevator for moving data between two levels of storage devices and TAPIOCA for efficient aggregation of data on compute nodes. We then implemented several components of the PDC system. They include metadata management, data placement services, remote procedure calls, data aggregation, etc. We have put them together into



the overall PDC framework. In the past year, we have worked on optimizing the PDC framework, API, unifying multiple heterogeneous storage layers, caching and prefetching, and interfacing the HDF5 with PDC. The following is a list of brief descriptions of the components developed in the past year.

1. Implemented an object-centric data management API for the PDC system that allows applications to map and unmap memory data objects to storage data objects. When a memory object is unmapped PDC asynchronously moves the data to and from the storage devices transparently and achieves up to 65% performance improvement compared to HDF5. PDC approach can overlap the I/O latency with computation cost between subsequent time steps.
2. In order to make the PDC object storage model available to a large number of applications without requiring existing applications to change their I/O interface, we have developed a method for HDF5 applications. The HDF5 library provides the Virtual Object Layer (VOL) feature that allows a connector to intercept the HDF5 API and redirect it to the PDC API. We have tested the PDC VOL connector with a plasma physics application and a Big Data clustering application's I/O, which shows up to 4X performance improvement compared to a highly optimized HDF5 implementation.
3. In the presence of multi-layer and distributed storage devices, providing a unified view of storage eases data placement and movement, which are currently performed manually by users. We developed a strategy for providing a unified namespace, called UniviStor, that provides a unified view of heterogeneous storage devices located on individual compute nodes, shared burst buffer, and parallel file systems.
4. We developed a caching and prefetching strategy for array data that is stored in HDF5. The Array Caching in storage Hierarchy (ARCHIE) observes the pattern of chunks of data being accessed and prefetches them into hierarchical storage layers. We developed this technique as part of the Data Elevator VOL connector that transparently intercepts HDF5 read calls and redirects to the cached data chunks, achieving up to 6X performance advantage compared to an approach with no caching. We will bring the prefetching strategy into the PDC framework in future.
5. HPC file systems have been moving towards relaxing POSIX constraints, using object-based storage concepts. To understand the status of existing object-based storage systems, such as Ceph RADOS, Intel DAOS, and OpenStack Swift, we have evaluated I/O performance using VPIC (plasma physics) and BD-CATS (big data clustering) applications. Among all object stores we evaluated, Intel DAOS has demonstrated outstanding performance by leveraging next generation NVMe/SCM technologies
6. We have used the PDC API for reading and writing cosmology and plasma physics simulations data, and for querying an astronomy observation datasets at a very large scale. In exercising the PDC API in this process, we have implemented metadata and data caching optimizations, that improved the PDC framework performance by multi-fold, depending on the available cache. We will enhance this method further based on the prediction methods we developed in ARCHIE.

2. High-level Architecture of the PDC System

Current I/O standards, such as POSIX-IO and MPI-IO, present fundamental challenges in the areas of semantic-based data movement optimizations, asynchronous operations, scalable metadata operations, and scalable support for consistent distributed operations. Object-centric mechanisms have been proposed but the existing approaches have not yet been able to express data structures that can transcend through all the layers of the hierarchy.

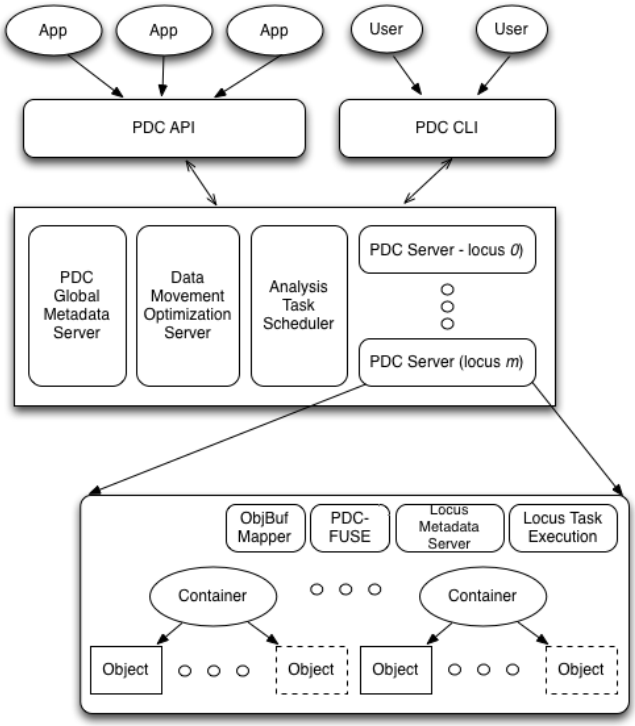


Figure 2. High-level architecture of the PDC system. Applications use the PDC API to create and manipulate (modify, move, access, delete) containers and objects. PDC servers manage metadata, move data container and objects, schedule analysis tasks, etc.

We have designed Proactive Data Containers (PDCs) towards enabling efficient and scalable scientific data management for the upcoming exascale storage systems. PDC starts with moving away from the existing file-oriented approaches, and explores novel with object-centric data management approaches. A PDC is a container of objects, where the objects are managed by PDC services and are placed in any level of the storage hierarchy (i.e., NVRAM, disk, tape, etc.). We show that the containers consisting of scientific data in an object-centric manner in Figure 2, where each container is shown within a storage layer for simplicity. It provides efficient data movement operations in critical areas of the exascale data management software stack. Data transformations according to future use of the data and analysis may take place 'pro-actively' while the data is in the containers. In this paper, we focus mainly on the data I/O services, without the emphasis on proactive analysis.

Our implementation of PDC adopts a client-server approach to monitor and manage I/O operations on objects. With this approach, PDC servers are able to asynchronously handle the I/O operations in the background, while the client application can proceed with their computation without waiting for I/O, exploiting compute and storage resource strengths of each location. Additionally, PDC allows applications to create containers and place objects at the various memory/storage locations along with defining the mappings and transformation. **In the next sections, we will provide the details of PDC API (§ 3), HDF5 interface to PDC (§ 4), Unified view of storage (§ 5), array caching using storage hierarchy (§ 6), an early evaluation of existing object-based storage technologies (§ 7), and evaluation of PDC at large-scale (§ 8), and then discuss briefly the ongoing research activities (§ 9).**

3. Object-centric Data Management Interface

Motivation

Scientific applications on upcoming HPC systems are facing challenges from three directions: extreme parallelism, a deepening heterogeneous memory hierarchy, and data that is massively increasing in volume and complexity. In particular, one of the challenges to address the diverse performance characteristics of deep memory hierarchies expected in exascale systems is the capability and efficiency of data movement across storage layers. Existing HPC data management and movement solutions, which were designed for simpler systems are no longer able to handle that level of complexity; similarly, scientific data models, which have been designed for 2-tiered storage hierarchies, need to be revised to embrace a more gradated storage hierarchy.

Solution

Moving toward an end-to-end, object-centric data abstraction and storage mechanism that takes advantage of deep storage hierarchies and enables proactive automated performance tuning, we have been investigating *Proactive Data Containers* (PDC) . A PDC is a container that may reside in a single storage location (i.e., memory, NVRAM, disk, etc.) or span across multiple levels, and stores data in an object-oriented manner. The PDC system provides an interface for creating, updating, retrieving, and deleting data objects and for managing metadata on those objects.

The main abstractions of PDC are its data constructs and operations performed on the data constructs. As shown in Figure 3, the data constructs include *Containers*, *Objects*, and *Regions*, all of which can have different *Properties*. An *Object* is a generic term to describe byte streams in an abstract manner. Objects in PDC are globally visible, independently of their storage hierarchy. A *Container* is a collection of *Objects* that share similar user defined attributes. PDC also uses the concept of *Spatial Regions*, which partition the problem domain into smaller sub-regions. Each region contains the actual data and associated metadata, and is the basic unit for data movement operation in PDC. All of this previously mentioned entities include *properties*, regarded as metadata.

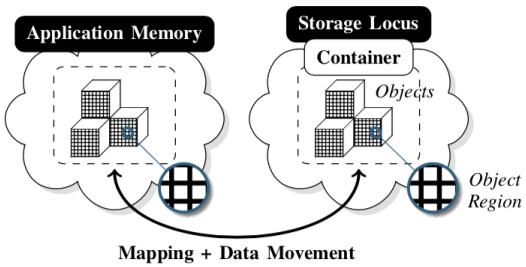


Figure 3. A mapping operation established between an application's memory region and a PDC object region.

Common I/O libraries have been providing so far *explicit* data movement operations through read and write functions. Here we introduce the concept of *object mapping* to make those calls *implicit* to the user by defining a *map* operation. Figure 4 shows four major steps to enable data movement by PDC without making any explicit data copy or transfer call. As the first step the client initiates a call to create an object by calling PDCobj_create(), the metadata server receives the RPC call and processes it. Then we introduce the important step (2) mapping mechanism and step (3) locking mechanism as following:

Object Mapping Mechanism: As highlighted in Figure 3, the object mapping primitive allows a user to define a mapping between a region within an application's memory and a region within a global PDC object. Mapping operations are defined on a per-region basis and can be thought of as a publish and subscribe mechanism, in the sense that once a mapping is established and a region is published, data movement can occur to keep updates globally visible. When defining a mapping, the application provides property information about the mapped region, which is essential for the PDC system to keep track of the mappings that are established and prevent potential overlaps. More complex mappings can be built upon this primitive such as object to object mapping, though we focus on the application memory to PDC object mapping exclusively.

Consistency and Locking Mechanism: To keep data consistency between the application's memory and the PDC object, we propose locking semantics for PDC objects (at the region granularity so that multiple regions of an object can be concurrently updated) and distinguish *read locks* from *write locks*. Assuming the mapping from memory to object has already been established, when a user has the intent to modify the application's memory region, the object region *write* lock must be acquired before any memory write access can occur (or that access would be considered as undefined). After the write lock is acquired, further changes to the object data region is not allowed globally until the lock is released. Once the lock is released per user's request, it effectively notifies the PDC system that it is now safe to move data between storage locations, and data movement will occur asynchronously if the memory region has been modified. Similarly in the case of read locks, the application expresses the intent of accessing the data and effectively prevents the PDC system from making any implicit update to the memory mapped buffers while they are being read.

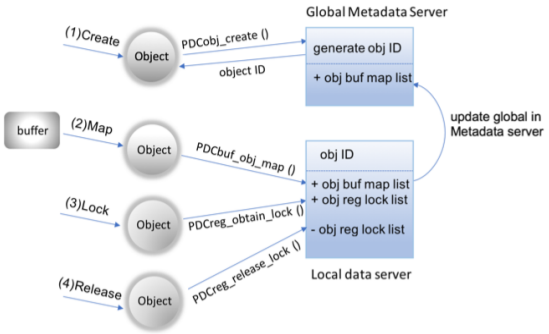


Figure 4. Four major steps to enable data movement by PDC without making any explicit data copy or transfer call.

Results

We use 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 memory. The shared file system Lustre is HDD-based. There is also an SSD-based "Burst Buffer", located between compute nodes and storage systems on Cori.

With the *shared* mode, we have one PDC server on each node, which occupies one core leaving the remaining 31 cores for user application execution. In the *dedicated* mode, PDC servers and user's application are on separate nodes. We configure Mercury with two communication protocols using the libfabric plugin over TCP and over Cray GNI. In the latter case, the PDC server was configured to use Cray DRC to allow the user's application and PDC server to share credentials and communicate together.

We used VPIC-IO to evaluate the PDC system's performance, which is extracted from VPIC [34], a code developed for simulating several plasma physics phenomenon. In VPIC- IO, each MPI process writes a region of 8M (8×2^{20}) particles and each particle has 8 properties. Each reported time includes client and server communication time, data and metadata server communication time, metadata maintain time and data movement time, if any of these is involved. We only report the time by multithreading server execution.

The overhead of PDCbuf_obj_map() and PDCreg_obtain_lock() are very low. Once a client initiates a lock release operation by calling PDCreg_release_lock(), the data server takes over the task and determines if data transfer is needed by going through the object region lock list. If so, the data server initiates a one-sided data transfer using remote memory access (RMA) by Mercury, which makes use of cross memory attach (CMA) for zero copy transfers when using shared-memory in shared mode, or makes use of native RDMA exposed by the system (e.g., uGNI over libfabric) in dedicated mode. The data server responds to the application lock request once it receives data from the client, and in the background asynchronously writes to lower storage tiers (i.e., burst buffer, file system). With two copies of results, data in data server is able to be quickly shared to other analysis or post processing tasks, such as in situ processing, while the copy in permanent storage allows the data to be recovered even if the application sees a fault for any reason.

With the extra threads dealing with asynchronous transfer, lock release time is around 0.5s for the shared mode (Figures 5) and 0.6s for the dedicated mode (not presented) by hiding the actual data transfer time to lower storage, whereas the time is ≈2s to 2.5s without asynchronous I/O.

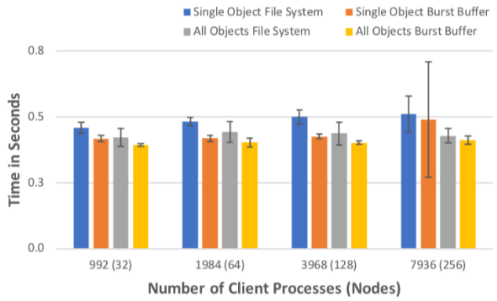


Figure 5. Time to release the lock for one object with share server.

Object Lock Release with GNI: To enable faster communication and data transfer, we compare the lock release time using Cray GNI instead of TCP used in previous experiments, since libfabric with GNI is targeting multithreaded applications requiring concurrent access to Aries high speed network (Cori network), with minimal contention between threads. In Figure 6 we show that the performance of lock release for dedicated mode using GNI is ≈3s compared to 4.5s using TCP protocol to release all objects.

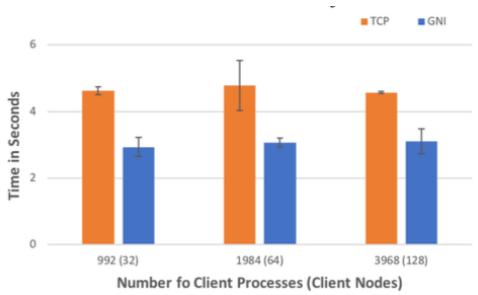


Figure 6. Time to release the lock for all objects with dedicated server.

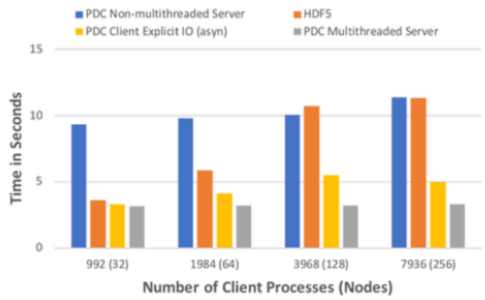


Figure 7. Time to move data from application memory to burst buffer by the user's point of view.

PDC Server Controlled Data Movement Performance Comparison: We compare the I/O performance of the new PDC mechanisms against HDF5 (with multiple optimizations [35]) for VPIC-IO and against our previous work of explicit I/O in PDC [1], by moving data from memory to a lower storage tier through Lustre (not presented) and burst buffer (Figure 7) from the view of application users. Each process writes 256*MB* of data. We use PDC servers in shared mode since that was the configuration in our previous work. When using HDF5, data is written to the storage directly as a single file. From the users' perspective, the performance of the new PDC server-managed method is improved by 65% and 49% on average compared to HDF5 writing directly to file system and burst buffer respectively, with asynchronous data movement. It also outperforms our previous client-explicit asynchronous I/O by 10% and 25% on average with respect to writing to file system and burst buffer.

Next steps

The next steps include enabling data transformation and analysis framework in PDC as well as making communication between servers topology aware.

4. HDF5 interface for PDC

Motivation

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 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. 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 8. 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.

Solution

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.

We recently designed the Proactive Data Containers 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.

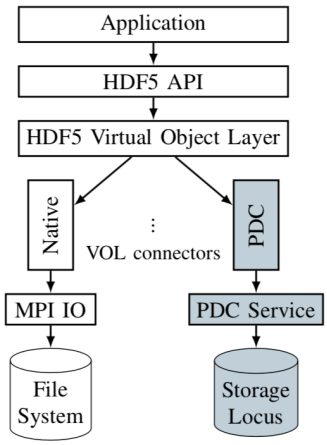


Figure 8. PDC within Virtual Object Layer. All of the HDF5 I/O related calls are routed to the corresponding VOL connector.

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

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.

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 call 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);
  ...
}
```

Results

We use the experimental setup as the previous setup. We compare the H5Dwrite() performance of VPIC-IO in Figure 9 using the following methodologies: HDF5 collective 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.

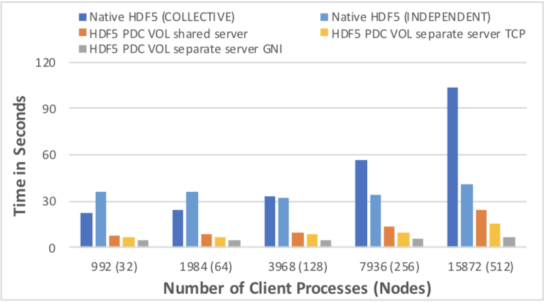


Figure 9. H5Dwrite() performance using different methodologies for one time step. Total data size goes from 248*GB* for 992 processes to 3968 *GB* for 15872 processes. The number of PDC servers for each configuration is equal to the number of compute nodes, indicated in parenthesis.

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 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 9 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.

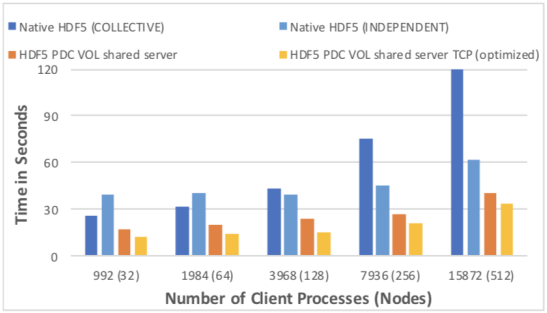


Figure 10.Total elapsed time for the execution of VPIC-IO if the user chooses not to wait for data to be flushed to disk.

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(). If the user chooses not to wait for data and let PDC server to take care of it (noted as optimized in Figure 10), 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.

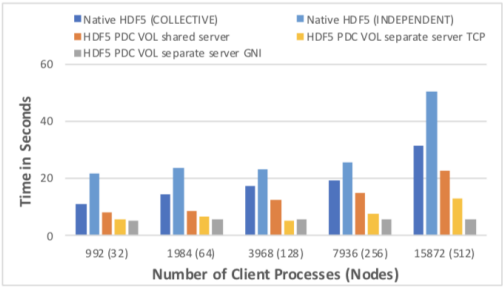


Figure 11. H5Dread() performance using different methodologies for one timestep. Total data size goes from 248*GB* for 992 processes to 3968 *GB* for 15872 processes. The number of servers for each setup is the number in parenthesis.

BD-CATS-IO is a read I/O kernel, which reads data produced by VPIC-IO in a load balanced way. In Figure 11, 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.

Next steps

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 is similar to `mmap` operations that 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.

5. UniviStor

Motivation

To improve the I/O performance on HPC systems that are traditionally limited by slow disk-based devices, the storage subsystem is being expanded in both hierarchical and distributed manner. Hardware such as NVRAM-based burst buffer is added on each compute node, or on I/O nodes accessible by all compute nodes, or on both. In addition, faster DRAM and Storage-Class Memory (SCM) are being added on compute nodes. Efficient use of the deep memory and storage hierarchy is a complex task for application developers and HPC system users without a unifying storage sub-system view. Existing storage management solutions are generally designed to manage one or a few layers, and require applications or users to explicitly move data across layers using each layer's individual solutions.

Solution

Providing a unified storage view integrating memory and storage layers is challenging. First, a unified address space is needed for data in all layers. Second, each layer has distinct performance characteristics, an integrated system has to be optimized for each layer. For instance, DRAM or SCM has high performance that is sensitive to context switches and task placement among cores. In contrast, shared storage layers, such as shared burst buffer and parallel file systems (PFS) are vulnerable to the I/O contention caused by multiple processes concurrently accessing the same file.

To ease the burden of rewriting existing codes, an integrated storage solution should be compatible with standard parallel I/O libraries, such as MPI-IO, HDF5 and netCDF. While existing software generally supports standard I/O on the shared burst buffer, there is still a lack of counterpart support on the node-local DRAM/burst buffers. Furthermore, a core benefit of the DRAM and burst buffer layer is the acceleration of scientific workflows with in-situ/in-transit analysis, where analysis programs can immediately read data that are close to computing nodes. However, the aforementioned parallel I/O libraries do not support this feature, forcing analysis programs to read only after a simulation completes or requiring an application-specific workflow management implementation. Although ADIOS supports in-situ/in-transit analysis using DataSpaces/FlexPath, applications based on other I/O libraries have to be rewritten with ADIOS' I/O interface.

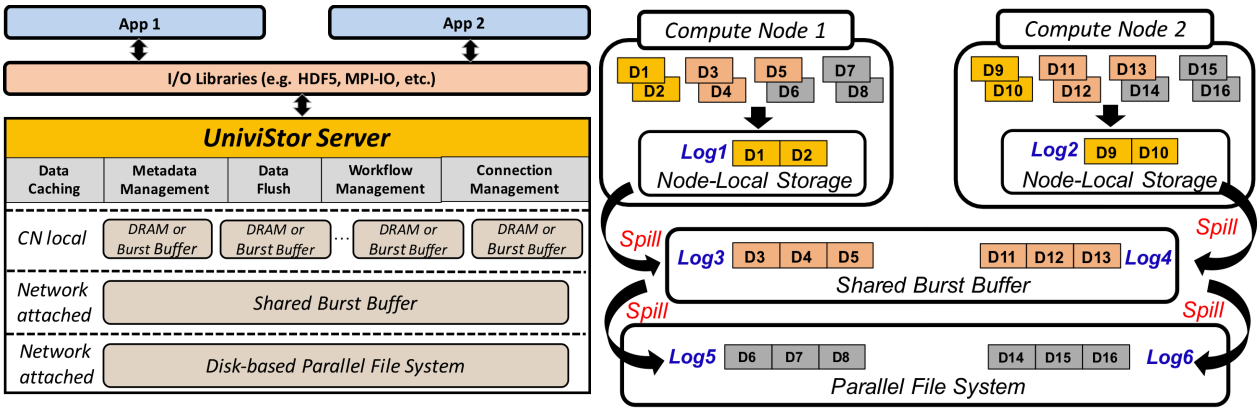


Figure 12. High-level view of UniviStor's storage hierarchy (left) and the distributed data placement, where UniviStor spills data to lower-level storage layers as the storage provided by the application using UniviStor fills up (right-side). In this figure, when the node-local storage is full, data is spilled to shared burst buffer and when that is also full data is spilled to parallel file system.

Towards providing a unified view of various storage layers, we have designed and implemented ***UniviStor***, a system that exposes the distributed and hierarchical storage spaces to applications as a single mount point. A high-level architecture of UniviStore with node-local storage, shared burst buffer, and a parallel file system is shown in Figure 12. UniviStor adopts the design philosophy of several state-of-the-art data service systems, such as Data Elevator and BurstFS that decouple address management from data management, and implement the address management as a distributed key-value service. More importantly, it extends this philosophy with a unified address management for different layers and location-aware data service. Furthermore, we have developed an interference-aware resource scheduling procedure that accelerates writing and reading data using the DRAM/SSDs distributed on compute nodes. We have introduced adaptive data striping for load balanced data movement to the disk-based PFS. UniviStor provides this unified service via the MPI-IO interface, which is used by high-level I/O libraries such as HDF5. We have also added parallel I/O support for in-situ/in-transit analysis on DRAM/burst buffers with a lightweight workflow management.

R&D of UniviStor has the following contributions:

- Design and implementation of UniviStor to integrate node-local and shared storage devices into a unified storage space using a distributed metadata service to manage the address space. UniviStor provides compatibility with existing I/O APIs and hides the complexity of managing different storage layers.
- Performance optimization strategies to support distributed and hierarchical placement of data using log-structured writes and interference-aware data movement scheduling for fast node-local caching.
- An analytical performance model to guide adaptive data striping on a parallel file system.
- A lightweight mechanism to orchestrate applications with data access dependencies when data are distributed across a multi-layer storage subsystem.

Results / Current Status

We have evaluated UniviStor using both benchmarks and application I/O workloads, and compared our system with the state-of-the-art solutions, including Data Elevator and Lustre, on

a production HPC system. Our experiments demonstrate that UniviStor integrates both node-local and hierarchical storage layers efficiently. UniviStor outperforms Data Elevator and Lustre by up to ~17X and ~46X, respectively (shown in Figure 13). We have also evaluated the performance of UniviStor in using various combinations of storage hierarchy to support data generation and analysis workflow.

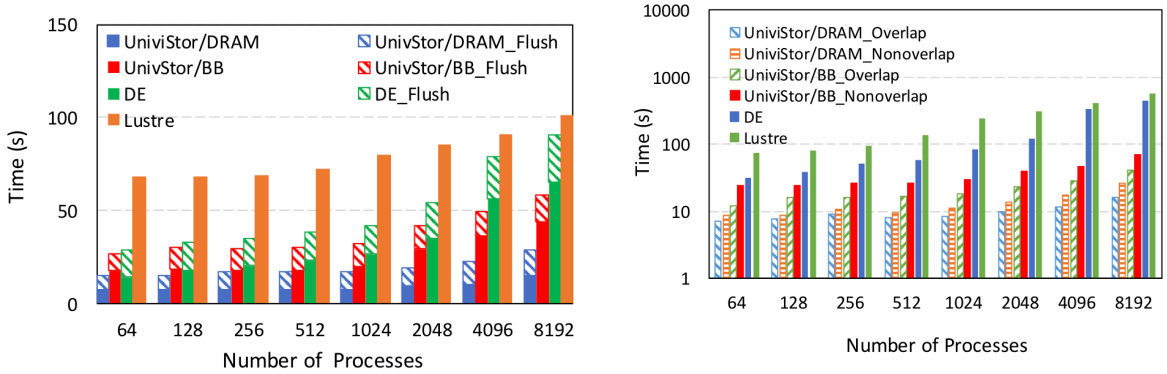


Figure 13. Performance of UniviStor in writing data from a plasma physics simulation kernel (left) and in reading data by a big data clustering algorithm (right). Both these use cases use HDF5 as the I/O API. In the case of writes, UniviStor is 46X faster than Lustre and 17X faster than Data Elevator.

Next Steps

We are moving the technology and prototype achieved in this work into the PDC runtime system. Another avenue of continuation of this work is moving the integration into programming models, where memory address space and storage namespace are unified into a single continuum, where applications view all the objects as

6. ARCHIE - Array caching

Motivation

While recent advances in HPC storage systems are adding multiple levels of storage hierarchy, traditional file systems are not equipped with supporting hierarchy of storage layers. To handle the bursty nature of generating and writing data, exascale computing designs are including node-local, non-volatile storage, and shared burst buffers. These layers act as faster caching layers to store the data temporarily. However, traditional file systems are designed for managing a single layer of storage devices, such as arrays of disks or solid-state drives (SSD). While there are solutions for moving the data between fast non-volatile storage and slow disk-based storage, solutions to accelerate reading data by prefetching into the intermediate storage layers are still required. While a few systems support prefetching the entire files, they often require user involvement. For example, Cray DataWarp provides tools for users to manually or programatically move data to a faster shared burst buffer. Similarly, software such as DDN IME and DAOS also need storage infrastructure changes or user involvement in data movement.

Moreover, solutions using array semantics and prefetching data at smaller granularity than the entire file are unavailable. Scientific data analysis frameworks such as TensorFlow and ArrayUDF work on large multi-dimensional arrays, e.g., 2D images. Incorporating array semantics into hierarchical storage

optimizations can help to further reduce data reading cost in data analysis tasks. Meanwhile, these data analysis frameworks are mostly designed to operate on a single storage layer. Providing a transparent way to use hierarchical storage for these systems is a critical task.

Solution

With the goal of providing a transparent and efficient data prefetching solution using array semantics in a hierarchical storage subsystem, we propose an ARray Caching in HIErarchical storage system (ARCHIE). This system is parallel by design and runs concurrently with data analysis applications to prefetch array data sets. ARCHIE provides an "array semantics"-aware prefetching function to move array data from slow storage devices into faster ones that are closer to analysis applications before the application issues the data read calls. This prefetching system reduces I/O overhead in most array-based data analysis applications.

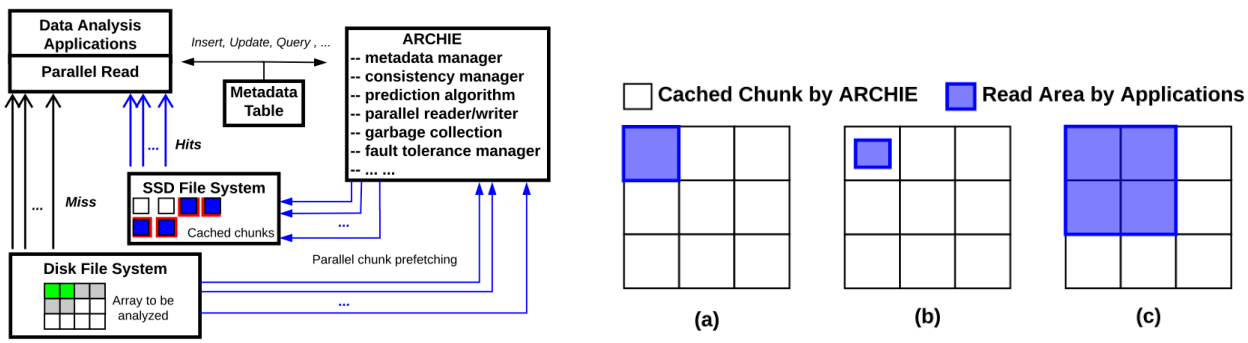


Figure 14. An overview of ARCHIE in caching data in a hierarchical storage system (left) Three base cases of querying cached array chunks in ARCHIE (right). ARCHIE supports caching and prefetching: a) The read region matches the boundary of a cached chunk. b) The read region is a portion of a single cached chunk. c) The read region is formed by multiple cached chunks. Other cases can be a combination of these base cases in querying cached chunks in ARCHIE.

The technical contributions of this effort are:

- Development of a new array cache management system for hierarchical storage to move large array data efficiently across heterogeneous devices.
- Design of a parallel prefetching method for large arrays to improve data read performance in analysis tasks. Our method also augments prefetched data with ghost zones, a typical requirement in array-based data analysis.
- Development of a fault-tolerance mechanism for the caching system to recover from errors automatically that may crash an application or the ARCHIE service.
- Implementation of ARCHIE using the HDF5 Virtual Object Layer (VOL) to provide caching and prefetching capabilities without requiring source code modifications.

Results / Current Status

We demonstrated the effectiveness of ARCHIE with three scientific data analysis codes, including convolutional neural network (Figure 15), gradient computing (Figure 16), and vorticity analysis (Figure 17). We ran our experiments on the Cori HPC system at NERSC that is equipped with a hierarchical storage system and thousands of computing nodes. The results show that ARCHIE is up to 6X faster than DataWarp, state-of-the-art storage system for managing burst buffers.

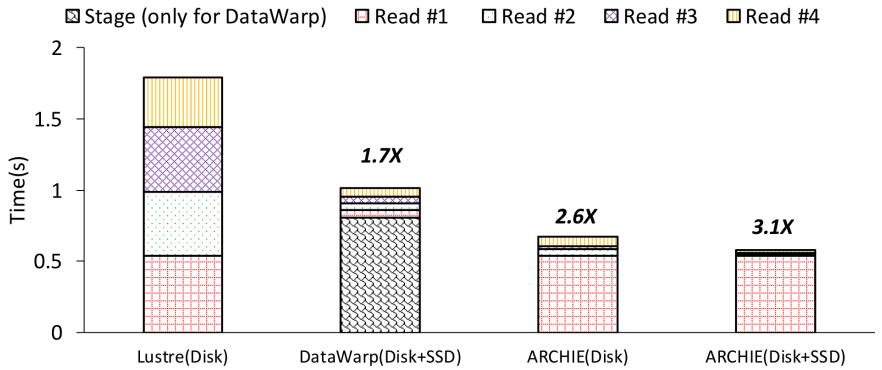


Figure 15. Evaluation of applying convolutional neural network (CNN) on CAM5 climate dataset

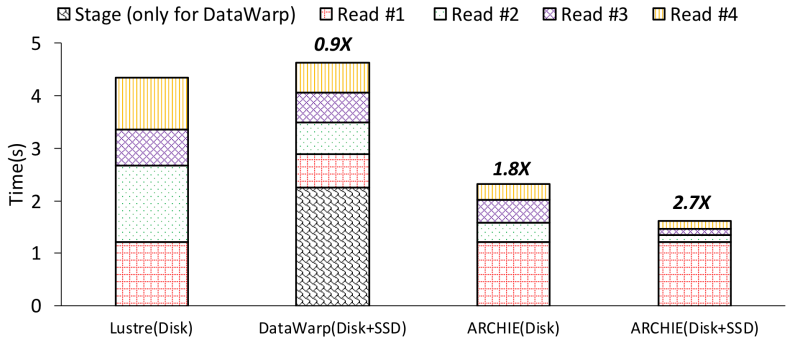


Figure 16. Evaluation of gradient calculation of a plasma physics dataset

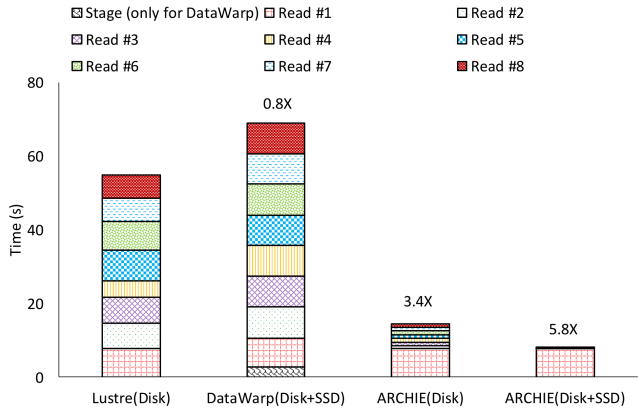


Figure 17. Evaluation of vorticity computation for a combustion (S3D) dataset

Next Steps

Using the prediction of future accesses and the optimization techniques developed in ARCHIE, we will design a data access pattern analysis and prediction service in the PDC runtime system.

7. Object-store evaluation

Motivation

POSIX-based parallel file systems provide strong consistency semantics, which many modern HPC applications do not need and do not want. Object store technologies avoid POSIX consistency and are designed to be extremely scalable, for use in cloud computing and similar commercial environments.

Solution

We evaluated three object store systems: Intel DAOS (in development), Ceph RADOS (jewel version), and Openstack Swift. For our experiments, we used the HDF5 library with the HDF5 Virtual Object Layer (VOL) feature. The VOL feature allows to redirect the applications' HDF5 calls to the underlying object storage systems' APIs with minimum application code change. It will be released by The HDF Group in the upcoming HDF5 1.12.0 release in Summer 2019. To enable applications to use above object stores, we developed an HDF5 VOL connector for Openstack swift, and we re-used the existing HDF5 VOL connector for Intel DAOS (developed by The HDF Group under the Fast-Forward project), and an early prototype of RADOS VOL (currently being developed by The HDF Group under the Mochi project). We compared these object stores using the aforementioned HDF5 VOL connectors with three HPC applications: VPIC, H5Boss, and BDCATS.

Results / Current Status

Through the evaluation, we demonstrated that object stores have better scalability than POSIX file systems like Lustre, in both large contiguous write and random small write. We observed important API differences between RADOS and Swift, e.g., RADOS has better support for partial read/write, which enables us to implement a better parallel I/O solution. Among all object stores we evaluated, Intel DAOS has demonstrated outstanding performance by leveraging next generation NVMe/SCM technologies, although our results should be followed up when the tmpfs-emulated SCM can be replaced with actual SCM. In Figure 18, we show an initial comparison of DAOS and Lustre, in writing 1K and 10K HDF5 datasets to the respective storage systems.

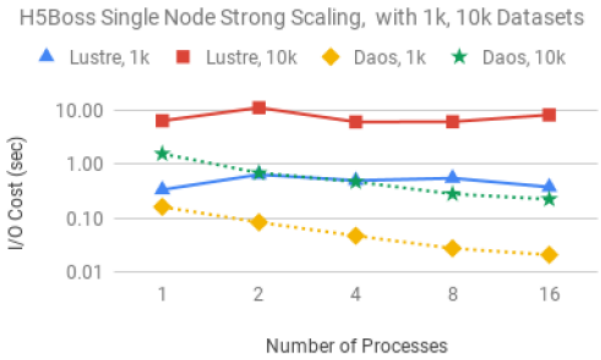


Figure 18. H5Boss Single Node Strong Scaling

Next Steps

We will expand our evaluation to larger scales and to other object storage solutions.

8. PDC Science Drivers

Motivation

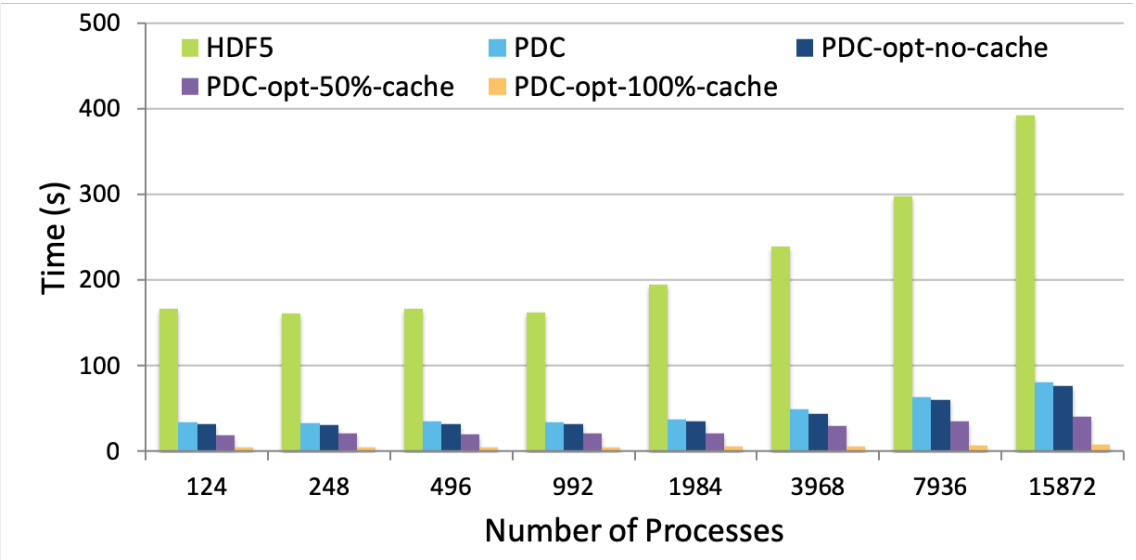
Efficient management of scientific data on high-performance computing (HPC) systems has been a challenge as it often requires knowledge of various hardware and software components of the systems as well as tedious manual effort in optimizing parallel I/O for each application. This situation is exacerbated by the fact that storage systems on upcoming exascale supercomputers are equipped with an unprecedented level of complexity due to a deep system memory and storage hierarchy. Simple and effective data management methods are critical for numerous scientific applications that are storing and analyzing massive amounts of data on HPC systems. Object-centric data management systems (ODMS) are gaining popularity on HPC systems because of their relaxation of POSIX-IO semantics and management of objects using a flat namespace. However, tuning an ODMS to achieve its full potential still remains a challenge.

Solution

We designed and developed (1) a data aggregationmethod that makes runtime decisions on scheduling I/O requests, (2) a prefetching and caching strategy that takes advantage of the knowledge of an ODMS and usesmulti-level storage hierarchy efficiently, and (3) an efficient metadata aggregation and asynchronous updating approach to achieve high performance with a relaxed data consistency requirement. We have implemented these new techniques in the Proactive Data Containers (PDC) system.

Results / Current Status

We have evaluated our optimization strategies first using a set of benchmarks and then applying them to a dark energy simulation, a space geometry survey, and a large-scale cosmology simulation. These evaluations are performed on a Cray XC40 system located at NERSC. We show that using our proposed techniques, an object-centric storage system achieves a multi-fold speedup compared to the existing parallel file system using POSIX-IO-based I/O libraries.



Read time for BD-CATS-IO with different configurations and number of processes.

9. Ongoing Research Activity and Next Steps

9.1. In data path analysis

While in-locus analysis in PDC, i.e., analyzing data while data objects are being moved between different storage loci, remains a work in progress, much of the infrastructure to enable data transforms and user defined analysis on PDC objects has been completed in this reporting period. This work differs from other approaches, such as SENSEI, in that PDC analysis is tied to data movement and/or data updates on PDC objects as opposed to providing APIs between applications such as visualization and performance monitoring. Our work leverages some of the ideas of ArrayUDF (https://bitbucket.org/arrayudf) in that PDC analysis is focused on providing an API that allows user defined operations to be run as data becomes available or when data objects are modified (e.g., after write locked objects become unlocked). Additionally, there are internal hooks to potentially invoke user defined functions written in FORTRAN, python, or Julia. Because PDC objects record the native data storage order (ROW major or COLUMN major) of each object, data movement can thus invoke data transformations automatically if necessary. Similarly, because PDC records the datatype associated with an object, transport between objects of different type signatures can be optimized, e.g. doubles can be moved as floats when the target does require the double precision. This functionality can be enabled by defining PDC properties on objects which are captured and then utilized by the PDC runtime as part of the analysis registration or data movement APIs.

In Figure 18 below, we show an example of a stencil application which mirrors those described in the ArrayUDF project. While PDC is obviously somewhat more complex at this initial stage of development to write than the C++ stencil approach, the PDC data iterator should be more *cache friendly* and thus have better overall performance. One obvious difference in this example is that PDC approach requires the calls back into the PDC library to access the data of interest. On the one hand, data iterators may complicate writing our analysis functions. On the positive side, the approach should minimize data copying and also provide a reasonable compromise for generating high performance algorithms such as we show below.

```
/* ArrayUDF neon/CoRTAD (Coral Reef Temperature Anomaly Database)
 * Running average in which the output point is averaged against the 3 previous points,
 * e.g. c(0,0,0)+c(-1,0,0)+c(-2,0,0)+c(-3,0,9) */
size_t cortad_avg_func(float *stencil[4], float *out, size_t elements)
{  size_t k;
   for(k=0; k < elements; k++)
      out[k] = (stencil[0][k] + stencil[1][k] + stencil[2][k] + stencil[3][k]) / 4.0;
   return 0;
}

size_t neon_stencil( pdcid_t iterIn, pdcid_t iterOut)
{
   …
    number_of_slices = PDCobj_data_getSliceCount(iterOut);
    if ((blockLengthIn = PDCobj_data_getNextBlock(iterIn, (void **)&dataIn, dimsIn)) == 0)
      printf("neon_stencil: Empty Input!\n");
    else {
        stencil[0] = stencil[1] = stencil[2] = stencil[3] = dataIn;
        for (k=0; k< number_of_slices; k++) {
          if ((blockLengthOut = PDCobj_data_getNextBlock(iterOut, (void **)&dataOut, dimsOut)) > 0) {
             if (cortad_avg_func(stencil,dataOut,blockLengthIn) == 0) {
                stencil[0] = stencil[1];
                stencil[1] = stencil[2];
                stencil[2] = stencil[3];
             }
             blockLengthIn = PDCobj_data_getNextBlock(iterIn, (void **)&dataIn, NULL);
             stencil[3] = dataIn;
          }
        }
    }
   return result;
```

```
}

main()
{
   …
   // create regions
    r3 = PDCregion_create(3, offsets3d, my3D_dims);
    r4 = PDCregion_create(3, offsets3d, my3D_dims);
    input3d_iter = PDCobj_data_iter_create(obj3, r3);
    output3d_iter = PDCobj_data_iter_create(obj4, r4);

    PDCobj_analysis_register("neon_stencil:arrayudf_example", input3d_iter, output3d_iter);
    ret = PDCbuf_obj_map(my3DTestArray, PDC_FLOAT, r3, obj3, r4);
    ret = PDCreg_obtain_lock(obj3, r3, WRITE, NOBLOCK);
    ret = PDCreg_release_lock(obj3, r3, WRITE);
   …
}
```

Figure 18. The PDC version of the CoRTAD analysis application

PDC transforms follow a similar approach to the analysis function registration. By default, the PDC runtime will look for a specified transform in a predefined shared library (in this case: libpdctransforms.so), though as we've seen with the analysis registration function, the transform registration may specify an alternate source for function resolution, e.g.

```
PDCbuf_map_transform_register("pdc_transform_compress", &x[0], reg_x, obj_xx, reg_xx, 0, INCR_STATE,
DATA_OUT);
```

In this example, the transform specification may include a ":[shared-library|executable]" as in the previous CoRTAD analysis example. The general form of the transform registration is to:

- Identify the function entrypoint and an optional shared object file which contains the entrypoint.
- With the PDCbuf_map version, we provide a pointer to the actual source for the transform data &x[0] above;
- The input region. Which provides the shape and datatype information of the input data array.
- The mapped target object (obj_xx)
- The target region. Which provides the shape and datatype information of the target data array.
- A "state" identifier. Unmodified == 0
- A state modifier, i.e. the action to take upon completion of the transform;
- A Data specification, e.g. DATA_OUT == the source of a mapping operation. If Data is currently on the client and the target is the SERVER, then the specified transform occurs on the CLIENT. DATA_IN, is the converse, i.e. given the same starting conditions, the DATA_IN operation will occur on the SERVER.

Lastly, we note that user defined functions in PDC are invoked via the dynamic shared object library APIs (*dlopen* and *dlsym* are used to resolve function pointers) which allow the loading and running of functions defined in shared libraries and/or user executables. This mechanism also enables the export of user defined functions from the client executable to the PDC servers and thus enables execution of client code directly on the PDC data servers.

9.2. Parallel querying on PDC

The process of scientific discovery involves extracting useful information from a potentially large amount of data through a series of queries, for example, physicists often need to locate and visualize a relatively small amount of high energy particles, and a query such as ``$Energy >

2.0$" would come up naturally. To answer such a query, a large amount of efforts have been put in by researchers. Traditional database management systems (DBMS) such as Berkeley DB, PostgreSQL, manage the user's data as well as additional semantic information, and allow users to obtain query results with an SQL interface. In scientific database community, SciDB is proposed to store and query array-structured data.

However, these database systems require converting data to their format, and the import process is often time-consuming and requires extensive user involvement. Additionally, the results produced by these systems are often text-based and needs to be converted to another format for further analysis and/or visualization. With the increasing complexity of the data analysis pipeline, introducing such a system to a scientific workflow would lead to a significant amount of effort on maintaining the database and sharing the data to other components of the workflow.

The PDC framework offers data stored as objects along with a significant metadata that can be used for tagging data. We are developing a querying mechanism on the PDC metadata and data objects. We will test this with various scientific benchmarks and compare performance with traditional approach of sifting through the entire data manually, with science-oriented databases such as SciDB, and any existing parallel data search methods on scientific data formats directly.

10. Publications using the PDC project funding

10.1. 2018 - 2019

1. Richard Warren, Jerome Soumagne, Houjun Tang, Suren Byna, Quincey Koziol, and Jingqing Mu, “Analysis in the Data Path of an Object-centric Data Management System”, in preparation
2. Houjun Tang, Suren Byna, Bin Dong, and Quincey Koziol, “Parallel Query Evaluation for Object-centric Data Management Systems”, in preparation
3. Houjun Tang, Suren Byna, Jialin Liu, Quincey Koziol, Bin Dong, and Teng Wang, “Tuning Object-centric Data Management Systems for Large Scale Scientific Applications”, in submission
4. Jingqing Mu, Jerome Soumagne, Suren Byna, Quincey Koziol, Houjun Tang, and Richard Warren, "Interfacing HDF5 with A Scalable Object-centric Storage System on Hierarchical Storage", Accepted for publication - Cray User Group (CUG) 2019
5. Babak Behzad, Suren Byna, Prabhat, and Marc Snir, "Optimizing I/O Performance of HPC Applications with Autotuning", ACM Transactions on Parallel Computing (TOPC), Volume 5 Issue 4, March 2019, Article No. 15, doi: 10.1145/3309205
6. Haoyuan Xing, Sofoklis Floratos, Spyros Blanas, Suren Byna, Prabhat, Kesheng Wu, Paul Brown, "ArrayBridge: Interweaving declarative array processing in SciDB with imperative HDF5-based programs", 34th IEEE International Conference on Data Engineering (ICDE) 2018
7. Bin Dong, Teng Wang, Houjun Tang, Quincey Koziol, Kesheng Wu, and Suren Byna "ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage", IEEE International Conference on Big Data (IEEE BigData) 2018
8. Suren Byna, Quincey Koziol, Venkatram Vishwanath, Jerome Soumagne, Houjun Tang, Kimmy Mu, Richard Warren, François Tessier, Bin Dong, Teng Wang, and Jialin Liu, "Proactive Data Containers (PDC): An object-centric data store for large-scale computing systems", AGU Fall Meeting 2018
9. Jialin Liu, Quincey Koziol, Gregory Butler, Neil Fortner, Mohamad Chaarawi, Houjun Tang, Suren Byna, Glenn Lockwood, Ravi Cheema, Kristy Kallback-Rose, Damian Hazen, and Prabhat,

"Evaluation of HPC Application I/O on Object Storage Systems", 3rd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS), 2018 (Held in conjunction with SC18)

10. François Tessier, Paul Gressier, Venkatram Vishwanath, “Optimizing Data Aggregation by Leveraging the Deep Memory Hierarchy on Large-Scale Systems”, ICS 2018: The 32nd ACM International Conference on Supercomputing
11. Fahim Chowdhury, Jialin Liu, Quincey Koziol, Thorsten Kurth, Steven Farrell, Suren Byna, Prabhat, Weikuan Yu, "Initial Characterization of I/O in Large-Scale Deep Learning Applications", Work in Progress paper, 3rd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS), 2018 (Held in conjunction with SC18)
12. Teng Wang, Suren Byna, Bin Dong, and Houjun Tang, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC", IEEE Cluster 2018.
13. Kimmy Mu, Jerome Soumagne, Houjun Tang, Suren Byna, Quincey Koziol, and Richard Warren, “A Server-managed Transparent Object Storage Abstraction for HPC”, IEEE Cluster 2018.

10.2. PDC Publications reported in previous reports

14. Houjun Tang, Suren Byna, Francois Tessier, Teng Wang, Bin Dong, Jingqing Mu, Quincey Koziol, Jerome Soumagne, Venkatram Vishwanath, Jialin Liu, and Richard Warren, “Toward Scalable and Asynchronous Object-centric Data Management for HPC”, 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid) 2018
15. Bharti Wadhwa, Suren Byna, and Ali R. Butt, “Toward Transparent Data Management in Multi-layer Storage Hierarchy for HPC Systems”, IEEE International Conference on Cloud Engineering 2018 (IC2E 2018)
16. Houjun Tang, Suren Byna, Bin Dong, Jialin Liu, and Quincey Koziol, “SoMeta: Scalable Object-centric Metadata Management for High Performance Computing”, The IEEE Cluster Conference 2017
17. François Tessier, Venkat Vishwanath, and Emmanuel Jeannot, “TAPIOCA: An I/O library for optimized topology-aware data aggregation on large-scale supercomputers”, The IEEE Cluster Conference 2017
18. Jialin Liu, Quincey Koziol, Houjun Tang, François Tessier, Wahid Bhimji, Brandon Cook, Brian Austin, Suren Byna, Bhupender Thakur, Glenn Lockwood, Jack Deslippe, Prabhat, “Understanding the I/O Performance Gap Between Cori KNL and Haswell”, Cray User Group Conference 2017 (CUG 2017)
19. Bin Dong, Suren Byna, Kesheng Wu, Prabhat, Hans Johansen, Jeffrey N. Johnson, and Noel Keen, “Data Elevator: Low-contention Data Movement in Hierarchical Storage System”, The 23rd annual IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), 2016.
20. Francois Tessier, Preeti Malakar, Venkatram Vishwanath, Emmanuel Jeannot and Florin Isaila, “Topology-Aware Data Aggregation for Intensive I/O on Large-Scale Supercomputers”, 1st Workshop on Optimization of Communication in HPC runtime systems (IEEE COM-HPC16), Held in conjunction with ACM/IEEE SuperComputing'16 Conference, 2016.

11. Presentations and Outreach

- Suren Byna, “Efficient Scientific Data Management on Supercomputers - HDF5 and Proactive Data Containers”, Invited talk at the University of Western Florida, Apr 2019
- Suren Byna, "Proactive Data Containers (PDC): An object-centric data store for large-scale computing systems", AGU Fall Meeting 2018, Dec 2018

- Bin Dong, "ARCHIE: Data Analysis Acceleration with Array Caching in Hierarchical Storage", IEEE BigData 2018, Dec 2018
- Jialin Liu, "Evaluation of HPC Application I/O on Object Storage Systems", PDSW 2018, Nov 2018
- François Tessier, “Optimizing Data Aggregation by Leveraging the Deep Memory Hierarchy on Large-Scale Systems”, ICS 2018, July 2018
- Kimmy Mu, "UniviStor: Integrated Hierarchical and Distributed Storage for HPC", IEEE Cluster 2018, Sep 2018
- Kimmy Mu, “A Server-managed Transparent Object Storage Abstraction for HPC”, IEEE Cluster 2018, Sep 2018
- Suren Byna, “Efficient Scientific Data Management on Supercomputers”, Summer intern seminar at LBNL, July 2018

12. Professional Service

- Bin Dong, Program Committee member, HPCC 2019
- Bin Dong, Program Committee member, INFOCOM 2019
- Bin Dong, Program Committee member, HPCCI 2019
- Bin Dong, Program Committee member, CIT 2018
- Bin Dong, Program Committee member, HPCC 2018
- Bin Dong, Publications chair of SSDBM 2017
- Quincey Koziol, Program Committee member, SC Asia 2019
- Houjun Tang, Program Committee member, ICIIT 2018
- Houjun Tang, Reviewer, IEEE Cluster 2018
- Houjun Tang, Reviewer, Big Data 2018
- Houjun Tang, Reviewer, Distributed and Parallel Databases Journal
- Houjun Tang, Reviewer, Journal of Parallel and Distributed Computing
- Houjun Tang, Reviewer, International Journal of Grid and High Performance Computing
- Suren Byna, Technical Paper Program Committee, SC18
- Suren Byna, Reviewer, DOE SBIR program
- Suren Byna, Program Committee member, IPDPS 2019
- Suren Byna, Program Committee member, CCGrid 2019
- Suren Byna, Reviewer, Journal of Future Generation Computer Systems
- Suren Byna, Program Committee member, NAS 2018
- Suren Byna, Track Chair - Data Management and Processing Techniques, ICIIT 2018
- Suren Byna, Program Committee member, IEEE Cluster 2018