DAOS for Extreme-scale Systems in Scientific Applications

M. Scot Breitenfeld*, Neil Fortner*, Jordan Henderson*, Jerome Soumagne*,
Mohamad Chaarawi†, Johann Lombardi† and Quincey Koziol‡
*The HDF Group, Champaign, IL 61820
†Intel Corporation, Santa Clara, CA 95054
‡Lawrence Berkeley National Laboratory, Berkeley, CA 94720

***Abstract*—Exascale I/O initiatives will require new and fully integrated I/O models which are capable of providing straightforward functionality, fault tolerance and efficiency. One solution is the Distributed Asynchronous Object Storage (DAOS) technology, which is primarily designed to handle the next generation NVRAM and NVMe technologies envisioned for providing a high bandwidth/IOPS storage tier close to the compute nodes in an HPC system. In conjunction with DAOS, the HDF5 library, an I/O library for scientific applications, will support end-to-end data integrity, fault tolerance, object mapping, index building and querying. This paper details the implementation and performance of the HDF5 library built over DAOS by using three representative scientific application codes.**

***Index Terms*—I/O Software Stack, Storage, Resilience, Exascale, Parallel File System, High Performance Computing, DAOS, HDF5**

1. Introduction

Scientists, engineers and application developers may soon need to address the I/O challenges of computing on future exaflop machines. Economic realities drive the architecture, performance, and reliability of the hardware that will comprise an exascale I/O system [1]. Moreover, I/O researchers [2] have highlighted significant weaknesses in the current I/O stacks that will need to be addressed in order to enable the development of systems that measurably demonstrate all of the properties required of an exascale I/O system. Possible poor filesystem performance at exascale has lead to the introduction of new or augmented file systems [3]. It is also anticipated that failure will be the norm [1] and the I/O system as a whole will have to handle it as transparently as possible, all while providing efficient, sustained, scalable and predictable I/O performance. The enormous quantities of data, and especially of application metadata, envisaged at exascale will become intractable if there can be no assurance of consistency in the face of all possible recoverable failures and if there can be no assurance of error detection in the face of all possible failures.

Furthermore, HPC applications developers and scientists need to be able to think about their simulation models at higher levels of abstraction if they are to be free to work effectively on problems of the size and complexity that become possible at exascale. This, in turn, puts pressure on I/O APIs to become more expressive by describing high-level data objects, their properties and relationships. Additionally, HPC developers and scientists must be able to interact with, explore and debug their simulation models. The I/O APIs should, therefore, support index building and traversal, and be integrated with a high level interpreted language such as Python to permit ad-hoc programmed queries. Currently, high-level HPC I/O libraries support relatively static data models and provide little or no support for efficient ad-hoc querying and analysis.

To provide a high degree of flexibility and portability to the user, the HDF5 library [4] [5] organizes data into a hierarchical tree that is composed of groups, datasets and attributes. Groups and datasets are linked and defined by a link name; attributes are attached to these objects and defined by a name and value. Datasets store the actual data and may be contiguously mapped from an application memory to a file, or stored in more complex patterns to ease further access and analysis of the data. This paper discusses an effort to port applications using HDF5 to use an exascale transactional I/O stack. Section 2 gives an overview of the envisioned exascale I/O systems and the challenges associated with them. Section 3 discusses a new transactional storage I/O stack implementation via HDF5, and Section 4 gives the strategies involved in porting scientific applications to the proposed storage I/O stack and gives numerous benchmarks for each application.

2. Exascale I/O Challenges

One possible approach for application I/O at exascale is to become object oriented. Meaning, rather than reading and writing files, applications will instantiate and persist rich distributed data structures using a transactional mechanism [6]. As concurrency increases by orders of magnitude, programming styles will be forced to become more asynchronous [7] and I/O APIs will have to take a lesson from HPC communications libraries, by using non-blocking operations to initiate I/O. I/O subsystems that impose unnecessary serialization on applications (e.g., by providing over-ambitious guarantees on the resolution of conflicting operations) simply may not scale. It could, therefore, become the responsibility of

the I/O system to provide, rather than impose, appropriate scalable mechanisms to resolve such conflicts. It is then the responsibility of the application to use those mechanisms correctly.

Components and subsystems in the numbers that will be deployed at exascale mean that failures are unavoidable and relatively frequent. Recovery must be designed into the I/O stack from the ground up and applications must be provided with APIs that enable them to recover cleanly and quickly when failures cannot be handled transparently. This mandates a transactional I/O model such that applications can be guaranteed their persistent data models remain consistent in the face of all possible failures. Recovery should also guarantee consistency for redundant object data and filesystem metadata whether such mechanisms are implemented within the filesystem or in middleware. Such behavior can be achieved by confining the object namespace within containers that appear in the filesystem namespace as single files. Higher levels of the I/O stack will see these containers as private scalable object stores, driving the need for a new standard low-level I/O API to replace POSIX for these containers. This provides a common foundation for alternative middleware stacks and high-level I/O models, suitable to different application domains.

2.1. Exascale System I/O Architecture

The economics and performance tradeoffs between disk and solid state persistent storage or NVRAM determine much of the exascale system architecture. NVRAM is required to address performance issues but cannot scale economically to the volumes of data anticipated. Conversely, disks can address the volume of data but not the economical aspects of the performance requirements.

Economics dictates a HPC cluster, Fig. 1, with hundreds of thousands of compute nodes interconnected with a scalable, high-speed, low-latency fabric where all (or a subset) of the nodes, called storage nodes, have direct access to byte-addressable persistent memory and optionally block-based NVMe storage as well. A storage node can export over the network one or more object, each of which corresponds to a fixed-size partition of its directly accessible storage. The goal of the HPC cluster is to have both fault tolerance and concurrency mechanisms. A storage node can host multiple objects within the limits of the available storage capacity.

2.2. Exascale Compute Cluster

Typically, I/O nodes (ION) will run Linux and have direct access to the global shared filesystem. Each ION will serve a different set of compute nodes (CN) to ensure I/O communications between CNs and IONs exit the exascale network as fast as possible. The NVRAM on the IONs will provide a key-value store for use as a pre-staging cache and a hot storage tier to handle peak I/O load and defensive I/O. Write data captured by the hot storage tier, will be repackaged by a layout optimizer according to expected usage into objects sized to match the bandwidth and latency

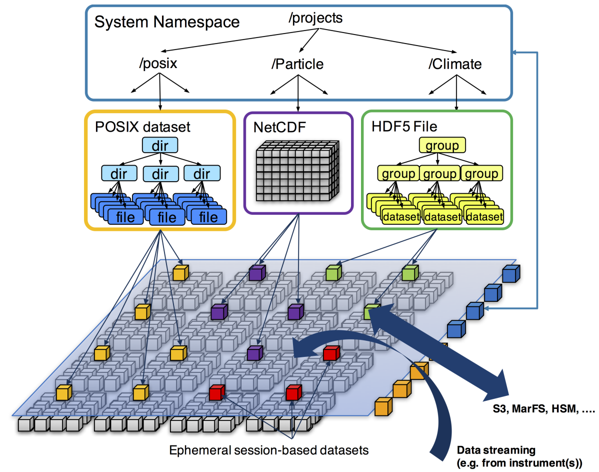

Figure 1. Vision for exascale storage.

properties of the storage tier targeted on the shared global filesystem. These storage objects will then be written in redundant groups using erasure codes or mirroring as appropriate. Object placement will be dynamic and responsive to server load to ensure servers remain evenly balanced and throughput is maximized.

CN or ION failure will be handled transparently by restarting the application from the last accessible checkpoint. In the case of CN failure, or if the NVRAM subsystem used for the hot storage tier is highly available and reliable (i.e., fully redundant and accessible via multiple paths) this will only require rollback to the last checkpoint stored in the hot storage tier. Otherwise the application will have to restart from the last checkpoint saved to the global shared filesystem.

3. A New I/O Software Stack

New HDF5 object storage APIs were developed to add support for end-to-end data integrity, fault tolerance, object mapping, index building and query. The new I/O APIs are implemented in the HDF5 library and provide a layer over the lower level object storage APIs. The top of the stack features a new version of HDF5, which directly interfaces with DAOS, Distributed Asynchronous Object Storage, and provides scalable, transactional object storage containers for encapsulating entire exascale datasets and their metadata, Fig. 2.

The DAOS storage system itself builds on existing techniques [8] [9] and middleware such as Argobots [10] for fast user-level threading and Mercury [11] for low-latency messaging and high-bandwidth data transfers. The essence of the DAOS storage model is a key-array object providing efficient storage for both structured (fixed-size array element addressed by index) and unstructured (variable length data stored in first array index) data, Fig. 3. The *object* key is a 2-level key:

1) *Distribution Key* (dkey) determines the placement. A user groups data under a single dkey to hint

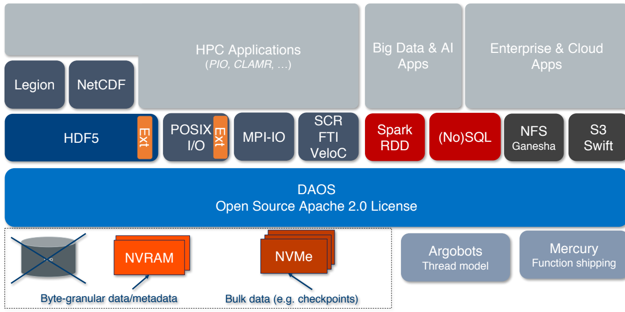

Figure 2. DAOS stack configuration.

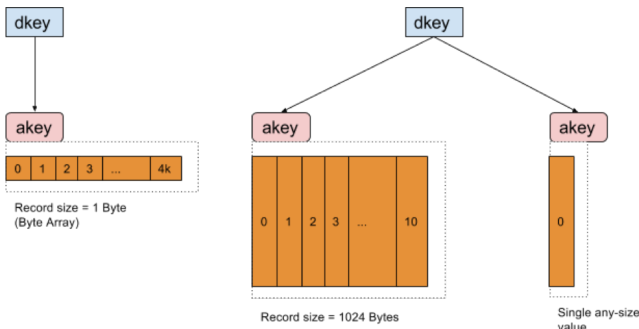

Figure 3. DAOS object model.

locality and colocation of that data on a single target;

2) *Attribute Key* (akey) identifies an array of values.

An array value is an arbitrary blob with an arbitrary size (from 1-byte to many GBs).

Many object schemas (replication/erasure code, static/dynamic striping and others) are provided to achieve high availability and scalability. The schema framework is flexible and easily expandable to allow for new custom schema types in the future. The actual object layout is generated on open. The object class is extracted from the object identifier to determine the schema of the object to be opened. End-to-end integrity is assured by protecting both object data and metadata with checksums during network transfer and storage.

3.1. I/O Transaction Model

The primary goal of the DAOS transaction model is to guarantee data model consistency with highly concurrent workloads. Applications should be able to safely update the dataset in-place and rollback to a known consistent state on failure.

DAOS also introduces the concept of a container which represents an object address space inside a pool. A container is the basic unit of atomicity and versioning. Any time a container is opened, a handle for that container is returned to the user. All object operations are explicitly tagged by the caller with both the container handle and a transaction identifier called an epoch. Operations submitted against the same epoch and container handle are applied atomically to a container on a successful commit.

The DAOS transactional model allows applications to concurrently update a DAOS container through different transactional contexts by utilizing different container handles. All operations submitted with the same epoch (transaction number) and container handle are guaranteed to be atomically committed or aborted. Several applications or processes/threads within an application may independently open and access a container through different handles. DAOS tracks all I/O submitted with both the epoch state and the container handle.

This all or nothing semantic eliminates the possibility of partially integrated updates on a container handle. On a successful commit, an epoch is guaranteed to be immutable, durable and consistent. Unused committed and aborted epochs for a container may be aggregated to reclaim space utilized by overlapping writes and reduce metadata complexity. The user is responsible though for conflicts (for example updating overlapping extents of an array in the same epoch), as DAOS will not have information at aggregation time to resolve such conflicts.

DAOS uses the concept of *transactions*, where one or more processes in the calling program can participate in a transaction, and there may be multiple transactions in progress in a container at any given time. Transactions are numbered, and the calling program is responsible for assigning transaction numbers in the DAOS stack. Updates in the form of additions, deletions and modifications are added to a transaction and not made directly to a container. Once a transaction is committed, the updates in the transaction are applied atomically to the container.

The basic HDF5 sequence of transaction operations on a container for opening and writing is:

1) *start* transaction N;
2) *update* the container;
3) *finish* transaction N.

Transactions can be finished in any order, but they are committed in strict numerical sequence. The application controls when a transaction is committed through its assignment of transaction numbers in “create transaction / start transaction” calls and the order in which transactions are finished, aborted, or explicitly skipped.

The **version** of the container after transaction N has been committed is N. An application reading this version of the container will see the results from all committed transactions up through and including N.

The application can **persist** a container version, N, causing the data (and metadata) for the container contents that are in hot storage to be copied to DAOS and atomically committed to persistent storage.

The application can request a **snapshot** of a container version that has been persisted to DAOS. This makes a permanent entry in the namespace (using a name supplied by the application) that can be used to access that version of the container. The snapshot is independent of further changes to

the original container and behaves like any other container from this point forward. It can be opened for write and updated via the transaction mechanism (without affecting the contents of the original container), it can be read, and it can be deleted.

3.2. HDF5 DAOS Implementation

HDF5 provides a set of user-level object abstractions for organizing, saving, and accessing application data in a storage container, such as groups for creating a hierarchy of objects and datasets for storing multi-dimensional data. The HDF5 binary file format is no longer used in DAOS. Instead, each HDF5 object is now represented as a set of Key-value objects used to store HDF5 metadata, replacing binary trees that index byte streams. A version of the HDF5 library that supports a Virtual Object Layer (VOL) was used for DAOS. For this work, a specialized HDF5 DAOS VOL plug-in interfaces to DAOS replaced the traditional HDF5 storage-to-byte-stream binary format with storage-to-DAOS objects.

Caching and prefetching is handled by DAOS, rather than by the HDF5 library, with the HDF5/DAOS VOL server translating an application's directives for HDF5 objects into directives for DAOS. Whereas HDF5 traditionally provided knobs for controlling cache size and policy, and then tried to "do the right thing" with respect to maintaining cached data, DAOS relies on explicit user directives, with the expectation that written data may be analyzed by another job before being evicted from the hot storage tier. In addition to the changes "beneath" the existing HDF5 API, the DAOS HDF5 version supports features seen as critical to future exascale storage needs: asynchronous operations, end-to-end integrity checking, and data movement operations that enable I/O to a hot storage tier. Additionally, DAOS HDF5 handles DAOS transactions logistics internally (i.e., the schema discussed in Section 3.1), resulting instantly in the capability to improve fault tolerance of data storage and allow near real-time analysis for producer/consumer workloads. Finally, the DAOS HDF5 version has exascale capabilities that are targeted to both current and future users that include both query/view/index APIs to enable and accelerate data analysis and a map object that augments the group and dataset objects.

4. Application I/O Strategies

This section discusses the strategies involved in porting scientific applications to DAOS. Section 4.1 discusses porting the application *CLAMR* to use HDF5 DAOS and gives an overview of the usability and capabilities from the perspective of a typical application code. The second application *Legion,* Section *4.2,* demonstrates using DAOS in a data-centric programming model. The last two applications NetCDF-4 and PIO, Section 4.3, are higher-level I/O libraries built on top of HDF5 and show the utilization of a higher level of abstraction to simplify an application's interaction with HDF5 and DAOS, which in turn should ease the transition to DAOS.

A small Intel DAOS prototype cluster, *Boro*, was used for all development and benchmarks described in Sections 4.1-4.3. Boro consists of Intel Xeon Processor E5-2699 v3 with two CPUs per node. The DRAM (Kingston KVR21R15S4/8) is being used in place of the NVRAM (Fig. 1) and the final storage stage uses Seagate Constellation ES.3 ST1000NM0033 HDD. Boro uses InfiniBand as its network backbone.

Before presenting the applications that were evaluated and ported to utilize the new I/O software stack, it is worth mentioning that both the DAOS library and the HDF5 DAOS backend are still in a prototyping phase. The primary goal of this research was to prove that utilizing the new I/O stack is doable and easy once a middleware library is properly designed on top of DAOS. Tuning for optimal performance in both HDF5 and DAOS was not done as part of this work. Therefore it is unknown how the following issues would effect the performance.

- All the applications, except CLAMR, utilizing DAOS did not use HDF5 chunking layout for dataset storage. Thus, only a single DAOS server with one service thread for I/O is utilized because in this case an HDF5 dataset is mapped to one DAOS object distribution. This undoubtedly will result in a bottleneck once the number of I/Os are increased.
- Communication to the DAOS server uses libfabric with TCP over InfiniBand, whereas communication to the Lustre server is done over InfiniBand verbs.
- Lustre servers use spinning disks for storage, but DAOS used a *tmpfs* file system over DRAM (since NVRAM is not available for this cluster).
- The MPI-I/O driver that HDF5 uses on Lustre will aggregate small I/Os at the client side before submitting I/O to the Lustre servers. No such aggregation is available yet in the HDF5 DAOS plugin nor at the DAOS client, which is something that will be explored in the future.

4.1. CLAMR Application

CLAMR (**C**ell-Based **A**daptive **M**esh **R**efinement) is a testbed application for hybrid algorithm development using MPI and OpenCL GPU code [12]. CLAMR does not use one output file per process due to several issues:

- File systems are limited in their ability to manage hundreds of thousands of files;
- In practice, managing hundreds of thousands of files is cumbersome and error-prone;
- Reading the data back using a different number of processes than the analysis simulation requires redistribution and reshuffling of the data, negating the advantage over more sophisticated collective I/O strategies.

Thus, the I/O strategy for CLAMR is to create one output file per time step and to store each time step in an HDF5 file.

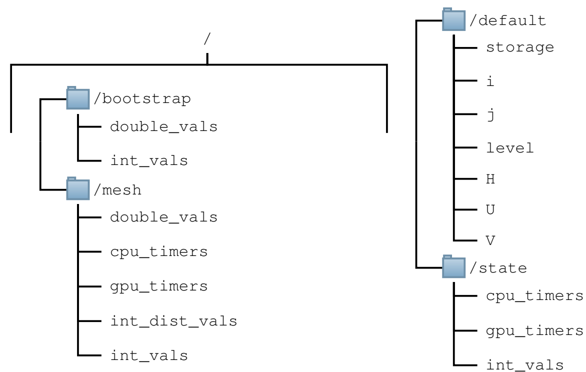

Figure 4. The HDF5 file layout structure for CLAMR.

Since HDF5 is a self-describing hierarchal file format, the "metadata" is automatically handled by HDF5. Thus, using HDF5 significantly reduces the internal bookkeeping and file construction required by CLAMR when compared to using POSIX or MPI-IO. The HDF5 implementation organizes the stored variables into datasets and uses groups to hold the datasets themselves, Fig. 4.

Only a call to initialize DAOS was added to CLAMR's HDF5 implementation to utilize DAOS. This DAOS initialization requires a pool id which is passed to CLAMR by setting the environment variable *pid*. The pool id is obtained by starting the DAOS server:

```
$ orterun -np 1 --report-uri ~/uri.txt daos_server -c 1
```

where c is the number of DAOS server threads, and then creating the pool and assigning the pool id to a shell environment variable:

```
$ export pid=$(orterun -np 1 --ompi-server \
    file:~/uri.txt dmg create)
```

The first benchmark tested CLAMR for problem sizes ranging from 2^7 to 2^{10} grid points by powers of 2, with the number of processors varying from 1 to 512 by powers of 2. For the initial run, a stripe count of four and a stripe size of 4 MB was used. The time to write the checkpoint files substantially increase as the number of processors is increased.

To gain a better understanding of what performance factors could be tuned, the Lustre parameters were varied for a fixed problem size of 2^{11} grid points. The Lustre parameters started at a default stripe count of 1 with a stripe size of 1 MB, as well as the case of a stripe count of 4 and stripe size of 1 MB. The results were compared to the first case of a stripe count of 4 and stripe size of 4 MB. The obtained results lead to the conclusion that the Lustre parameters had a slight effect on performance, with the default stripe count of 1 and stripe size of 1 MB resulted in the best throughput.

Finally, different HDF5 chunking layout parameters using the default Lustre settings of a stripe count of 1 and stripe size of 1 MB were investigated. A variety of chunk

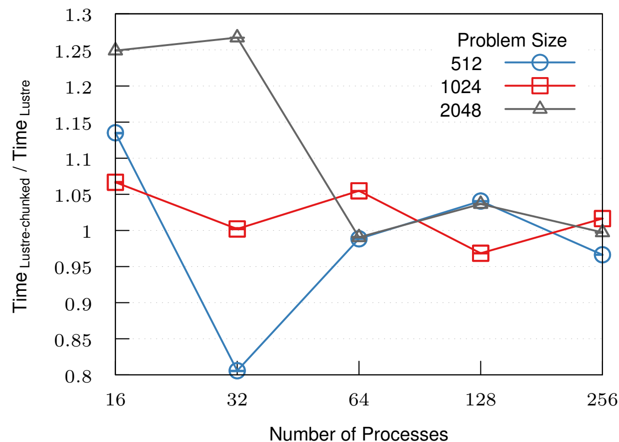

Figure 5. Comparison of CLAMR's Lustre write performance between HDF5 unchunked I/O and HDF5 chunked I/O using the optimal chunk size for the given problem size. Values greater than one represents chunked CLAMR I/O performing worse than unchunked CLAMR I/O.

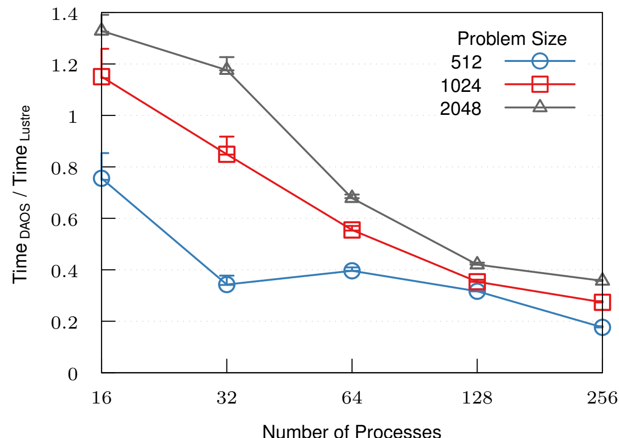

Figure 6. Comparison of unchunked write performance of CLAMR on DAOS/Lustre. Lustre parameters are a stripe count of one and stripe size of 1MB and DAOS used one DAOS server. Values above one represent unchunked CLAMR I/O on DAOS performing worse than unchunked CLAMR I/O .

sizes from 2^8 to 2^{18} by powers of 2 were tested across different problem sizes to obtain the optimal chunk size for the given problem size. The results show an improvement in CLAMR's performance when using the optimal chunk size in comparison to the nonchunked I/O performance, Fig. 5.

These results show that chunking had little positive effect on the performance of CLAMR checkpoint writing, with the result being marginally faster in a few cases and on par with or significantly worse than unchunked I/O performance in the other cases. When the chunk size of 8192 was used with 32 processors on the 512 problem size, a performance increase of 20% is observed, however no chunk size was able to replicate this performance improvement for the other sets of parameters.

CLAMR write performance was then assessed using the

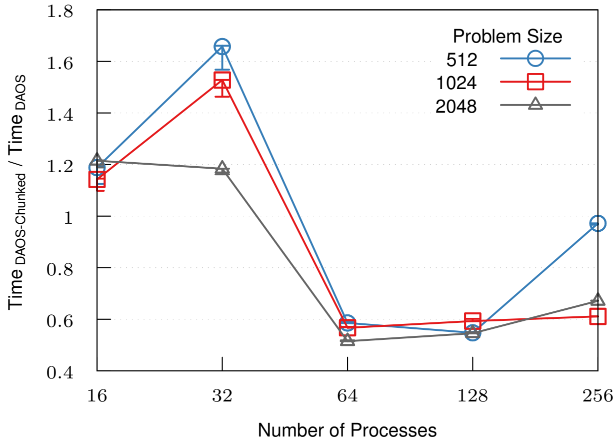

Figure 7. Comparison of chunked and unchunked CLAMR write performance on DAOS. Unchunked CLAMR I/O used one DAOS server; Chunked CLAMR I/O used eight DAOS servers across eight different nodes. Values above one represent chunked CLAMR I/O performing worse than unchunked CLAMR I/O.

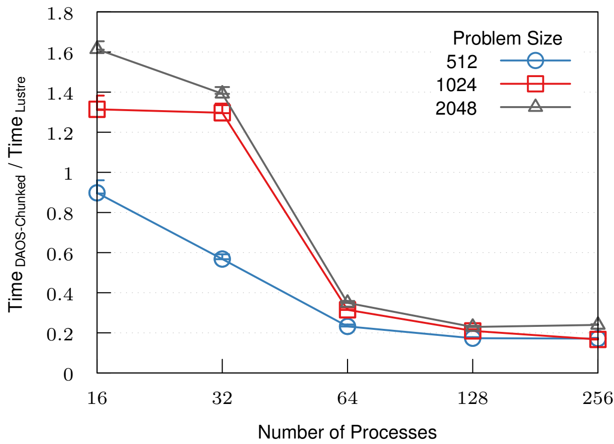

Figure 8. Comparison of chunked CLAMR I/O on DAOS to unchunked CLAMR I/O on Lustre. Values above one represent chunked CLAMR I/O on DAOS performing worse than unchunked CLAMR I/O on Lustre.

DAOS stack both with and without the use of chunking. Using the previous data gathered for CLAMR's performance in writing checkpoint files to a Lustre filesystem, a comparison was drawn, showing that DAOS without chunking performs twice as well on average as compared to Lustre (up to nearly three times as well in specific cases) when the number of processors is sufficiently large, Fig. 6. In order to assess the performance impact that chunking has, 8 DAOS servers were started across 8 different nodes and a variety of chunk sizes ranging from 2^9 to 2^{21} were tested. When chunking is enabled, DAOS performance can exceed the unchunked performance by approximately 25–50%, depending on the number of processors used and the appropriateness of the chunk size chosen for the given problem size, Fig. 7. When the number of processors used is large enough, this in turn leads to a consistent 50–75% increase in performance as compared to Lustre unchunked I/O, Fig. 8.

4.2. Legion Parallel Programming System

Legion [13] is a high-level data-centric and task-based application runtime. Legion's primary data model is based on *Logical Regions* which are the cross product of an *N*-dimensional index space and a multi-variable field space. Logical regions are distinct from the physical regions (memories) that underlie the logical region and provide the physical instantiation of the data. The data model also provides a method of coloring the index space and/or the field space, which can be used to partition an index space or slice the field space of the logical region. The runtime can then use this coloring and a partitioning applied to the logical region to manage a distributed instance of the logical region.

Legion is currently capable of attaching to HDF5 files using the low-level Realm runtime on which Legion is built. Realm maps HDF5 files (or groups) to memory objects within the runtime using a Direct Memory Access (DMA) mechanism to collect updates to individual datasets in an HDF5 file. As part of demonstrating Legion's use of HDF5 on the DAOS storage stack, an example application is created that demonstrates the capabilities of the storage stack. `Tester_io`[1] is a straightforward tour of the HDF5-DAOS features from a simple Legion code and is designed to run the Legion runtime on multiple compute nodes, talking to multiple I/O and storage server nodes.

In the current implementation of HDF5 on top of DAOS, transactions and the event stack are handled internally in the HDF5 library, and consequently, no extra parameters needed to be added to the original HDF5 APIs. Therefore, the number of changes from the original Legion and HDF5 implementation was minimal. One new HDF5 API call, `H5VLdaosm_init`, was introduced into Legion, which initializes the VOL plugin by connecting to the DAOS server pool and registering the driver with the HDF5 library. This function is called only once by all the processes within Legion.

The Legion benchmark relies on the low-level networking layer `gasnet` [14] for network-independent, high-performance communication primitives. The `mpiexec` command in the gasnet wrapper script `gasnetrun_ibv` needed to be updated to include DAOS specific parameters. The command for running `tester_io` on 3 nodes with 131072 elements and 256 shards is:

```
$ GASNET_BACKTRACE=1 GASNET_USE_XRC=0 \
   GASNET_MASTERIP='...' GASNET_SPAWN=-L \
   gasnetrun_ibv -n 3 tester_io -n 131072 -s 256
```

As was the case with CLAMR, the *pool id* is passed to Legion through the environment variable `pid`.

The non-bulk synchronous ability of DAOS via the transaction and epoch model of DAOS was demonstrated bu using `Tester_io`. Fig. 9 shows the typical workload for the

1. `Tester_io` is part of Legion's Github repository and can be found in the source tree at test/hdf_attach_subregion_parallel

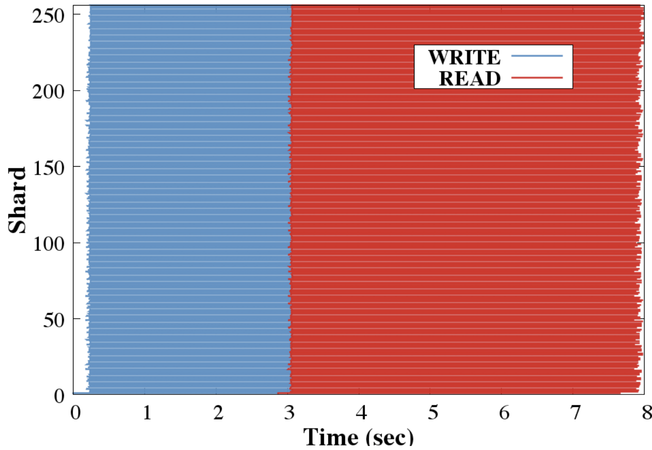

Figure 9. Time-series of the I/O phases (write and read) associated with each shard's global data structure being persisted for HDF5 with DAOS, highlighting Legion's ability to perform independent I/O phases associated with each shard of global data for DAOS.

read/write phase for different shards in `Tester_io`. Additionally, Fig. 9 highlights Legion's capability of scheduling different phases of tasks based on explicit dependencies and, consequently, allows for reading tasks to run concurrently with writing tasks on the same logical region [15].

`Tester_io` was used to study the effects of the number of MPI processes as a function of the number of Legion subregions, where the size of the problem was fixed for the number of elements of 2^{29}. The number of DAOS servers was one and the number of Legion MPI processes was 4, 8 and 15 where each compute node executed only one MPI process, and each node could run 74 tasks. The higher number of process made a large difference when the number of subregions increased to greater than 256 for both reading and writing, Fig. 10 and Fig. 11.

The performance of Legion using DAOS and Lustre is presented in Fig. 12 for 15 MPI processes and where the number of elements is 2^{30}. The slowdown in DAOS and the big performance hit when compared with Lustre in this case was expected, since the Legion tester code generates a large number of I/O calls, and the HDF5 implementation in Legion did not use chunking layout for the Datasets. This results in all the I/O calls being serialized at the DAOS server to one service thread, whereas the Lustre server utilizes many threads to handle the incoming I/Os. Furthermore, as mentioned earlier, the network interface used by DAOS on this cluster was libfabric over TCP which is slower compared to InfiniBand verbs utilized by Lustre. We also noticed that the *tmpfs* file system was returning out of space errors for larger problem sizes, even when space was available, after a certain number of memory allocations triggered by the large number of I/O operations from the client. All those issues will be addressed in future development of both DAOS and the HDF5 DAOS backend.

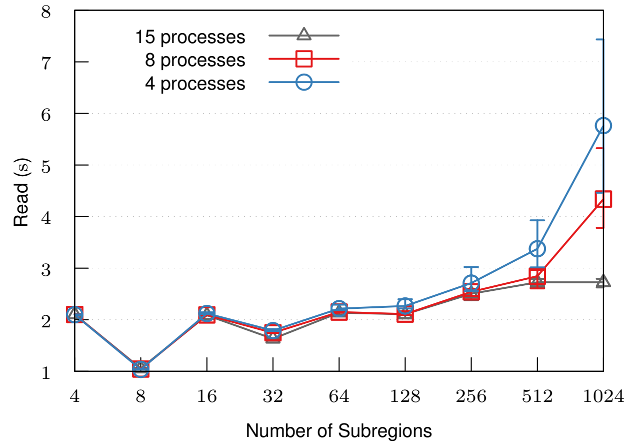

Figure 10. DAOS read performances as the number of MPI processes is increased from 4 to 15 processes for a different number of subregions.

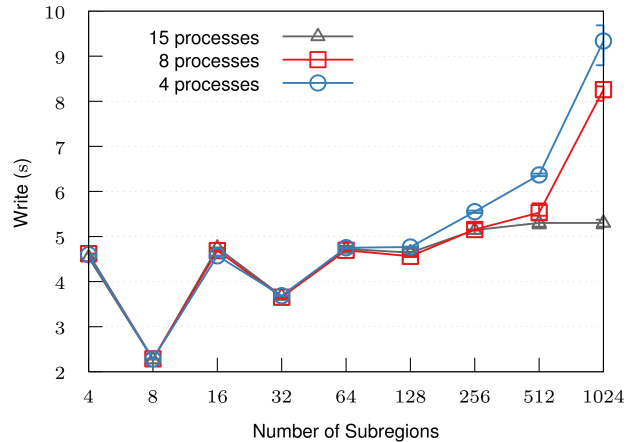

Figure 11. DAOS write performances as the number of MPI processes is increased from 4 to 15 processes for a different number of subregions.

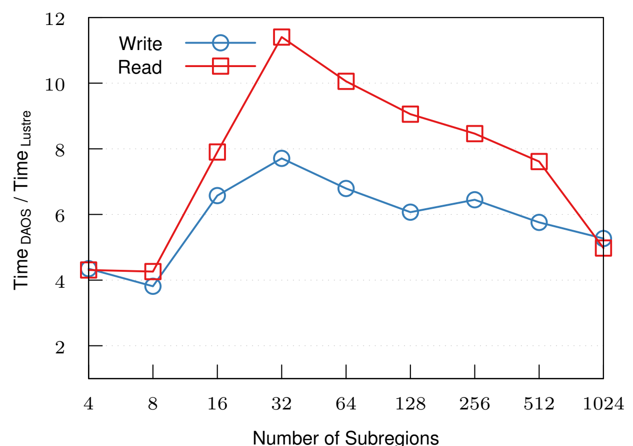

Figure 12. DAOS read and write performance in comparison to Lustre. DAOS uses a single service thread while Lustre uses multiple threads on multiple servers.

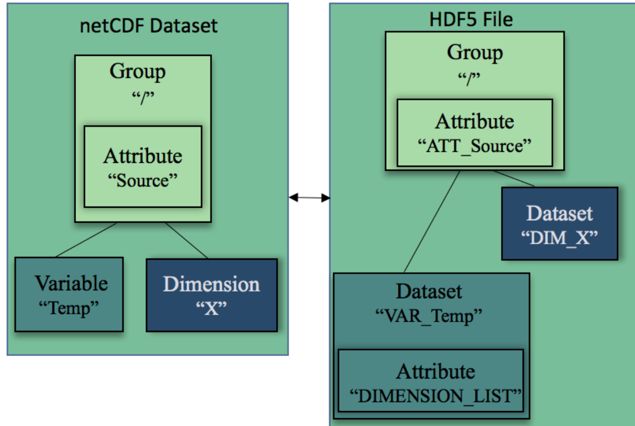

Figure 13. HDF5 schema for NetCDF/DAOS.

4.3. High-level I/O Stack Libraries

There is a desire to support legacy libraries on the DAOS stack that already make use of HDF5. The objective is to have mid-level code completely manage DAOS, isolating the application code from the I/O stack.

4.3.1. DAOS NetCDF Implementation. NetCDF [16] is a set of software libraries used to facilitate the creation, access, and sharing of array-oriented scientific data in self-describing, machine-independent data formats. A new set of DAOS NetCDF APIs were derived from the original NetCDF APIs, maintaining most of the original functionality.

The non-DAOS version of NetCDF added dimensions to variables by the use of HDF5 *Dimension Scale* APIs. Storing dimensions with coordinate variables (variables used as a dimension scale for a dimension of the same name) is intuitive and is self-describing for applications that access the file directly through HDF5. However, this approach introduces several dependencies between datasets and attributes that do not fit well with the DAOS transaction model. In addition, it introduces many special cases that must be handled, increasing the difficulty of implementation. Finally, there is currently no DAOS implementation of the Dimension Scale APIs and implementing them would be difficult due to the transaction model.

Since backward compatibility with NetCDF's file format was not of a concern (i.e., having NetCDF/DAOS datasets/containers/files being accessed independently of the NetCDF/DAOS API) the existing NetCDF4 schema for HDF5 was abandoned in order to simplify the implementation. All variables and dimensions were implemented as HDF5 datasets, all groups as HDF5 groups, and all attributes as HDF5 attributes. As a convention, all dimensions have the string DIM_ prepended to the name in HDF5, all variables have the string VAR_ prepended, and all attributes have the string ATT_ prepended. Variables have an HDF5 attribute DIMENSION_LIST, invisible to the NetCDF API, that stores references to the dimensions for the variable, Fig. 13. Dimensions are implemented as a scalar dataset of type H5T_STD_U64LE, where the value indicates the dimension length or all 1s (i.e. `(uint64_t)(int64_t)-1`) to indicate an unlimited dimension. This implementation avoids all name conflicts without having to add any special cases to the code, and also allows the removal of code paths for handling coordinate variables as a special case, instead of treating them like any other variable. All the NetCDF APIs that use this new schema were appended with a "`_ff`" in their names. Other DAOS additions to NetCDF included:

- Support for unlimited dimensions, but only for collective access and only for the slowest changing dimension;
- "Links" from variables to their dimensions, allowing the variables to be queried about their dimensions.

4.3.2. DAOS *Parallel I/O* (PIO) Implementation. A common application library which uses NetCDF is the software associated with the **A**ccelerated **C**limate **M**odeling for **E**nergy (ACME) [17]program. ACME uses the package *Parallel I/O* (PIO) [18] to perform I/O which, in turn, uses as its backend the NetCDF file format. Since the global DAOS stack variables are isolated from both PIO and NetCDF, PIO APIs simply use the DAOS "`_ff`" NetCDF APIs mentioned in Section 4.3.1. The transaction number is automatically initialized and incremented as needed within HDF5. Similar to the NetCDF convention, all new DAOS PIO C APIs are indicated by appending a "`_ff`" to the function names.

PIO expects as input from the application the partitioned data arrays for each process. Additionally, PIO has the option for requesting a subset of the CN that will perform the I/O. Hence, PIO aggregates the I/O from each process to only a subset of processes for I/O. The I/O processes then use NetCDF APIs to carry out the I/O. PIO implements two methods for aggregating the I/O from all the processes to the subset of I/O processes. In the box method, each compute task will transfer data to one or more of the I/O processes. For the subset method, each I/O process is associated with a unique subset of compute processes for which each compute process transfers data to only one I/O process [19]. In general, the subset method reduces the overall communication cost when compared to the box method.

Additionally, since PIO has the capability of using a subset of processes for I/O, `H5VLdaosm_init` (i.e., a DAOS HDF5 API used to start the DAOS stack) uses the MPI sub-communicator group so that only those processes involved in I/O will initialize the DAOS stack. This initialization of the DAOS stack happens automatically when the I/O MPI sub-communicator is created in PIO and it is finalized when this same sub-communicator is freed in PIO.

PIO's testing program `pioperformance.F90` uses two input files; the first file contains namelist settings for the testing parameters and the second file contains the decomposition information from a PIO program (e.g. CESM, ACME). The test program reads namelist and then generates test data consisting of integers, 4-byte reals and 8-byte reals. It then writes the data using DAOS NetCDF via PIO APIs

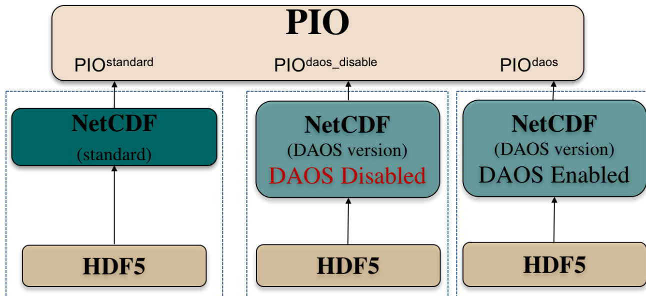

Figure 14. Three combinations of PIO, NetCDF and HDF5.

and then reads the data back using DAOS PIO, checks for correctness, and outputs the data rate in reading and writing the data.

There are two versions of NetCDF: (1) the "standard" version is the unmodified v4.4.1 of NetCDF available from Unidata and (2) the "DAOS" version, which is a modified v4.4.1 of NetCDF, as previously discussed. Three combinations of PIO, NetCDF and HDF5 are investigated, Fig. 14:

1) $\text{PIO}^{\text{standard}}$—The PIO configuration uses standard HDF5 with the standard version of NetCDF;
2) $\text{PIO}^{\text{daos_disable}}$—The PIO configuration uses standard HDF5 with the DAOS version of NetCDF, where the DAOS capabilities in NetCDF are disabled;
3) PIO^{daos}—The PIO configuration uses the DAOS version of HDF5, and the DAOS capabilities in NetCDF are enabled.

Thus, for both case 1 and 2, a POSIX file is getting written, via HDF5, to a Lustre backend.

All tests were made on Boro to compare PIO with Lustre and DAOS, and to verify the DAOS PIO implementation. An investigation of the Lustre parameters resulted in a stripe count of 1 with a stripe size of 2MB being used for all the Lustre benchmark results. The symbol in the figures is obtained by running the benchmarks ten times and averaging the I/O times over those ten runs. The vertical line segment represents the minimum and maximum of I/O over the ten runs.

The benchmark used 1024 processes and the decomposition file, `piodecomp1024tasks03dims05.dat`. The modified DAOS version of NetCDF outperforms the standard NetCDF version for both reading (Fig. 15), and writing (Fig. 16). Moreover, the DOAS implementation is on average faster than the Lustre implementation for reading and matches Lustre complete times for writing.

5. Conclusions

The increase from petascale to exascale for storage and I/O requires a new approach to the architecture because it is not possible to just scale prior systems. As shown, DAOS is primarily designed for these next generation systems which

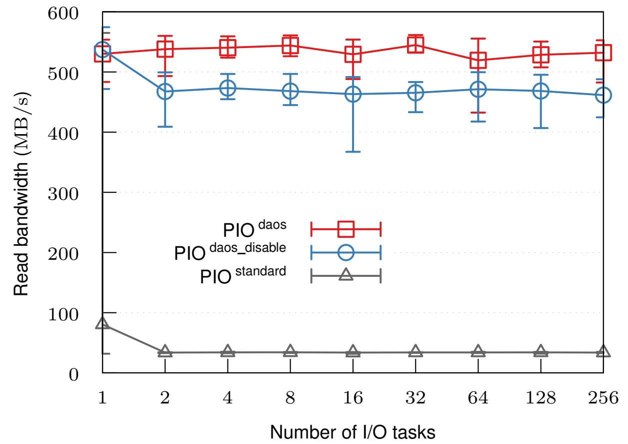

Figure 15. Read performance comparison between Lustre and DAOS for 1024 processes.

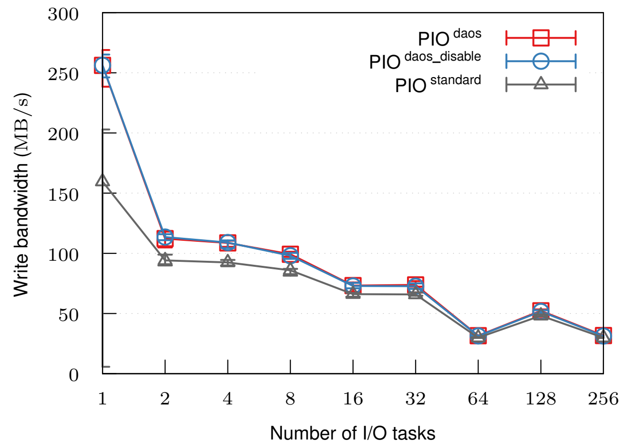

Figure 16. Nearly identical DAOS and Lustre write performance for 1024 processes.

use NVRAM and NVMe storage technology as a means to provide a high bandwidth storage tier very close to the compute node.

This research highlights the challenges and issues when porting existing applications codes to a transaction model for I/O. The transition to using DAOS for application and middleware developers is made easier by using HDF5, which hides the DAOS schema within HDF5. Although this paper focuses on HDF5 and DAOS, other I/O libraries such as MPI-I/O and POSIX I/O will also be supported on top of DAOS in the future.

The performance of DAOS in comparison to Lustre from a diverse set of applications was highlighted. Although the comparison was not justified due to the significant differences in the hardware and software stack, in similar workloads DAOS completed either faster or equivalent to Lustre. Other cases highlight the importance of leveraging multiple DAOS server processes and threads to avoid the bottleneck at the server side when performing large number of I/Os from the client without any aggregation, and pro-

vided ground for improving both the DAOS implementation and the HDF5 plugin for DAOS, as both are still in the prototyping phase.

Acknowledgments

Funding for this project was provided through Intel subcontract #CW1998599 from Intel's contract #B613306 with Lawrence Livermore National Security. Access to the Boro cluster was provided by Intel.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, M. Richards, and A. Snavely, "ExaScale Computing Study : Technology Challenges in Achieving Exascale Systems," *(P. Kogge, Editor and Study Lead)*, vol. TR-2008-13, pp. 1–278, 2008.

[2] F. Isaila, J. Garcia, J. Carretero, R. Ross, and D. Kimpe, "Making the case for reforming the I/O software stack of extreme-scale systems," *Advances in Engineering Software*, vol. 16, no. 10, pp. 1–6, jul 2016. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0965997816301740

[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," *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, pp. 746–752, 2012.

[4] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An Overview of the HDF5 Technology Suite and its Applications," in *Proceedings of the EDBT/ICDT Workshop on Array Databases*, ser. AD '11. New York, NY, USA: ACM, 2011, pp. 36–47.

[5] M. Folk, R. McGrath, and N. Yeager, "HDF: an update and future directions," in *International Geoscience And Remote Sensing Symposium (IGARSS)*, vol. 1, 1999, pp. 273–275 vol.1.

[6] J. Lofstead, J. Dayal, I. Jimenez, and C. Maltzahn, "Efficient transactions for parallel data movement," *Proceedings of the 8th Parallel Data Storage Workshop on - PDSW '13*, pp. 1–6, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2538542.2538567

[7] D. E. Keyes, "Exaflop/s: The why and the how," *Comptes Rendus Mécanique*, vol. 339, no. 2-3, pp. 70–77, feb 2011.

[8] P. Carns, K. Harms, J. Jenkins, M. Mubarak, R. Ross, and C. Carothers, "Impact of data placement on resilience in large-scale object storage systems," in *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, May 2016, pp. 1–12.

[9] P. Carns, K. Harms, J. Jenkins, M. Mubarak, R. B. Ross, and C. Carothers, "Consistent Hashing Distance Metrics for Large-Scale Object Storage," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, poster. [Online]. Available: http://sc15.supercomputing.org/sites/all/themes/SC15images/tech_poster/tech_poster_pages/post117.html

[10] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, A. Castello, D. Genet, T. Herault, P. Jindal, L. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, and P. H. Beckman, "Argobots: A Lightweight Threading/Tasking Framework," 2016, Report ANL/MCS-P5515-0116.

[11] 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.

[12] D. Trujillo, R. Robey, N. Davis, and D. Nicholaeff, "Cell-based Adaptive Mesh Refinement on the GPU with Applications to Exascale Supercomputing," in *APS Four Corners Section Meeting Abstracts*, Oct. 2011. [Online]. Available: http://meetings.aps.org/link/BAPS.2011.4CF.F1.36

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012, pp. 1–11.

[14] D. Bonachea, "GASNet Specification, v1.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-02-1207, Oct 2002. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5764.html

[15] N. Watkins, Z. Jia, G. Shipman, C. Maltzahn, A. Aiken, and P. McCormick, "Automatic and transparent I/O optimization with storage integrated application runtime support," *Proceedings of the 10th Parallel Data Storage Workshop on - PDSW '15*, pp. 49–54, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?id=2834976.2834983

[16] NetCDF, "Retrieved from www.unidata.ucar.edu/software/netcdf/," 2016.

[17] ACME, "Accelerated Climate Modeling for Energy (ACME) Project Strategy and Initial Implementation Plan," 2014. [Online]. Available: https://climatemodeling.science.energy.gov/sites/default/files/publications/acme-project-strategy-plan.pdf

[18] PIO, "Retrieved from www.ncar.github.io/ParallelIO," 2016.

[19] J. Edwards, J. M. Dennis, and M. Vertenstein, "Parallel I/O library (PIO)," *http://ncar.github.io/ParallelIO/*, 2016.