Use of a new I/O stack for extreme-scale systems in scientific applications

M. Scot Breitenfeld[a], Quincey Koziol[b], Neil Fortner[a], Jerome Soumagne[a], Mohamad Chaarawi[a]

[a]The HDF Group, [b]Lawrence Berkeley National Laboratory, United States

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

***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 development of a transactional object storage I/O stack and an extended version of HDF5 capable of managing this next generation I/O stack. This new HDF5 implementation adds support for end-to-end data integrity and security, mapping objects, index building, maintenance and query, and analysis shipping. This research highlights challenges and issues when porting common scientific application codes to use a transactional I/O stack.**

I. Introduction

Scientists, engineers and application developers will 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 [Kogge et al., 2008]. Moreover, other I/O researchers [Isaila et al., 2016] have highlighted significant weaknesses in 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 also lead to the introduction of new or augmented file systems [Mehta et al., 2012]. Furthermore, it is anticipated that failure will be the norm [Kogge et al., 2008] and the I/O system as a whole will have to handle it as transparently as possible, 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 the size and complexity of problems 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, maintenance, 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. I/O APIs are required that support dynamic data models with pointers to express arbitrary relationships between them and to annotate them with expected usage to guide lower layers in the I/O stack.

This paper discusses an effort to port applications using HDF5 to use newly proposed exascale transactional I/O stacks. Section II gives an overview of the envisioned exascale I/O systems and the challenges associated with them. Section III discusses a new transactional storage I/O stack implementation via HDF5, and Section IV discusses the strategies involved in porting scientific applications to the proposed storage I/O stack.

II. Exascale I/O challenges

The likely trend 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 [Lofstead et al., 2013]. As concurrency increases by orders of magnitude, programming styles will be forced to become more asynchronous [Keyes, 2011] and I/O APIs will have to take a lesson from HPC communications libraries, by using non-blocking operations to initiate I/O coupled with event queues to signal completion. I/O subsystems that impose unnecessary serialization on applications, e.g. by providing over-ambitious guarantees on the resolution of conflicting operations, simply will not scale. It will, therefore, become the responsibility of the I/O system to provide, rather than impose, appropriate and scalable mechanisms to resolve such conflicts and the responsibility of the application to use them 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 so that applications can be guaranteed their persistent data models remain consistent in the face of all possible failures. This will also guarantee consistency for redundant object data and filesystem metadata whether such mechanisms are implemented within the filesystem or in middleware. Furthermore, the exascale filesystem namespace should continue to operate at human scale and that the object

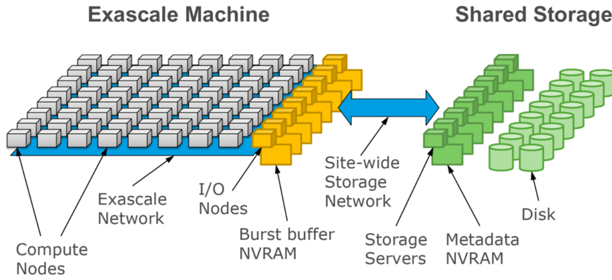

Figure 1. Exascale I/O architecture – NVRAM on I/O Nodes enable burst buffer capability to handle peak I/O load and defensive I/O.

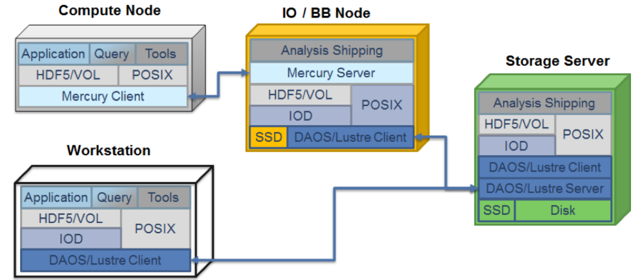

Figure 2. EFF stack configuration.

namespace required to implement scalable I/O should be separate from it. This 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 will provide a common foundation for alternative middleware stacks and high-level I/O models, suitable to different application domains. Additionally, these new APIs, as POSIX does today, need to support alternative filesystem implementations.

A. 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 economically the performance requirements. This dictates a system architecture, Figure 1, that places NVRAM close to the compute cluster in order to exploit its cross-sectional bandwidth for short-term use as a cache or burst buffer (BB) [Nowoczynski et al., 2008], [Bent et al., 2012a], [Bent et al., 2012b]. Disk-based storage bandwidth will be an order of magnitude less. This places fewer demands on the networking needed to connect the storage to the compute, allowing it to use commodity networking hardware so that it can become a site-wide shared storage resource.

B. Exascale compute cluster

The exascale compute cluster is envisioned to have on the order of 100,000 Compute Nodes (CNs). The CNs will run a lightweight runtime/microkernel, supporting HPC communications libraries, and I/O forwarding to staging processes running on dedicated I/O Nodes (IONs). Applications will use a high-level I/O library with an object-oriented API that encrypts or checksums data on its way to storage and decrypts or checks it on retrieval to guarantee end-to-end data integrity and optionally, security.

Typically, IONs will run Linux and have direct access to the global shared filesystem. Each ION will serve a different set of compute nodes 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 BB to handle peak I/O load and defensive I/O. Write data captured by the BB, will be repackaged by a layout optimizer according to expected usage into objects sized to match the bandwidth and latency 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 Burst Buffer 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 Burst Buffer. Otherwise the application will have to restart from the last checkpoint saved to the global shared filesystem.

III. A NEW I/O SOFTWARE STACK

A new object storage I/O API based on HDF5 was developed which adds support for end-to-end data integrity and security, mapping objects, index building, maintenance and query, and analysis shipping. The new I/O API is implemented as a library that can be layered over lower level object storage APIs. Function shipping can be used on the exascale machine to export the API from the IONs to the CNs. The top of the stack features a new version of HDF5, extended to support the new exascale fast forward (EFF) storage semantics for high-level data models, their properties and relationships. In the middle of the stack is IOD, the I/O Dispatcher, which stages data between storage tiers and optimizes placement. Finally at the bottom of the stack is DAOS, Distributed Application Object Storage, which provides scalable, transactional object storage containers for encapsulating entire exascale datasets and their metadata, Figure 2.

A. I/O Transaction model

The new I/O stack uses the concept of transactions, where one or more processes in the application 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 application is responsible for assigning transaction numbers in the EFF storage 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 sequence of transaction operations an application typically performs on a container that is open for writing is:

1) *start* transaction N,
2) add updates for container to transaction N,
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 on the BB 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.

B. HDF5 EFF implementation

HDF5’s programming API 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 the EFF stack, instead, each HDF5 data model object is now represented as a set of IOD objects and IOD Key-value objects are used to store HDF5 metadata, replacing binary trees that index byte streams. A modularized version of the HDF5 library that supports a Virtual Object Layer (VOL) was used with the EFF stack. For this work, a specialized client/server VOL plug-in that interfaces to the IOD API is used, replacing the traditional HDF5 storage-to-byte-stream binary format with storage-to-IOD objects.

Caching and prefetching is handled by the IOD layer, rather than by the HDF5 library, with the HDF5/IOD VOL server translating an application’s directives for HDF5 objects into directives for IOD objects. 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, the EFF stack relies on explicit user directives, with the expectation that written data may be analyzed by another job before being evicted from the burst buffer. In addition to the changes ‘beneath’ the existing HDF5 API, the EFF HDF5 version supports features seen as critical to future exascale storage needs: asynchronous operations, end-to-end integrity checking, data movement operations that enable support for I/O burst buffers, and support for transaction capabilities that improve fault tolerance of data storage and allow near real-time analysis for producer/consumer workloads. Finally, the EFF HDF5 version has exascale capabilities that are targeted to both current and future users: query/view/index APIs to enable and accelerate data analysis, a map object that augments the group and dataset objects, and an analysis shipping capability that sends application-provided Python scripts to execute directly where data is stored.

IV. Application I/O strategies

This section discusses the strategies involved in porting scientific applications to the EFF stack. The first application ported to the EFF stack is *HACC* and gives an overview of the usability and capabilities from the perspective of a typical application code. The other two applications (*netCDF-4* and *PIO*) are higher-level I/O libraries built on HDF5 and demonstrate the use of a higher level of abstraction in order to simplify an application’s interaction with the I/O stack, which in turn should ease the transition to the EFF stack.

A. HACC application

HACC (**H**ardware/Hybrid **A**ccelerated **C**osmology **C**ode) [HACC, 2016] is an N-body cosmology code framework where a typical simulation of the universe demands extreme-scale simulation capabilities. However, a full simulation of HACC requires terabytes of storage and hundreds of thousands of processes, which far exceeds the computational resources available for this research. Consequently, a smaller I/O code, *GenericIO*, was used to mirror the I/O calls in HACC without the need to run an entire simulation. In its current implementation, all the “heavyweight” data is handled using POSIX I/O, and the “lightweight” data is handled using collective MPI-IO.

Critical features for HACC’s I/O include,

- Resiliency for data verification,
  - Checksumming from the application’s memory to the file and vice-versa,
  - Mechanism for retrying I/O operations,
- Sub-filing,
  - Should avoid penalties in the file system associated with locking and contention,
- Self-describing file.

As for I/O strategies, the HACC team [Habib et al., 2014] found that creating one output file per process resulted in the best write bandwidth compared to other methods because it eliminates locking and synchronization between processors. However, this method is not used 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 complex collective I/O strategies.

The default I/O strategy in HACC is to have each process write data into a distinct region within a single file using a custom, self-describing file format. Each process writes each variable contiguously within its assigned region. On supercomputers having dedicated ION, HACC instead uses a single file per ION. The current implementation of HACC provides the option of using MPI I/O (collective or non-collective) or POSIX (non-collective) I/O routines. Additionally, GenericIO implements cyclic redundancy code (CRC) by adding it to the end of the data array being written [Habib et al., 2014].

1) HACC HDF5 and EFF implementation: Since HDF5 is a self-describing hierarchal file format, much of the "metadata" used in the current implementation of HACC, was automatically handled by HDF5. Thus, using HDF5 greatly reduces the internal bookkeeping and file construction required by HACC when compared to using POSIX or MPI-IO. The use of offsets as pointers to variables (note that these offsets are stored within the HACC file) is eliminated in the HDF5 implementation by instead using datasets to store variables. The HDF5 implementation stores HACC variables as datasets in the file's root group (/).

Associated with each variable's dataset is the cyclic-redundancy-check (CRC). The CRC uses the High-Performance CRC64 Library from Argonne National Laboratory. A CRC is computed for each variable, and each processor computes the CRC for the portion of the array residing on that process. Although the EFF stack automatically performs a checksum from the ION to the disk, this is not the case for HDF5 files by default. However, the CRC can easily be implemented within the HDF5 file by simply computing a CRC for the array (assuming no partial writes are taking place) and writing the CRC as a dataset. The reading program can then read the dataset, compute the CRC for the read data and perform a comparison to the values stored in the CRC dataset. The implied restriction is that the layout of the array among the processors is the same for both the writing and the reading of the arrays. Ensuring a matching CRC for data written and data being read is important when creating raw binary files because the file can be transferred to a machine with a different endianness. Therefore, checks have to be made to ensure the endianness conversions were implemented correctly. In HDF5 however, the library will convert and verify the byte-order automatically, so the use of the CRC may no longer be necessary. The rest of the implementation is fairly straightforward, with only a slight modification of the original HDF APIs, Algorithm 1.

Algorithm 1 HACC I/O stack scheme.

Require: Initialize EFF stack
1: **while** Output variables remain **do**
2: Establish a read context for operations on HDF5 file
3: Create a transaction object
4: Start a transaction
5: Create HDF5 objects associate with variable
6: Write variable
7: Finish transaction
8: Close transaction
9: Release container
10: **end while**
Require: Finalize EFF stack

B. High-level I/O stack libraries

There is a desire to support legacy libraries on the EFF stack that already make use of HDF5. The objective is to have the high-level code manage the transaction requests and to isolate the application code from the I/O stack.

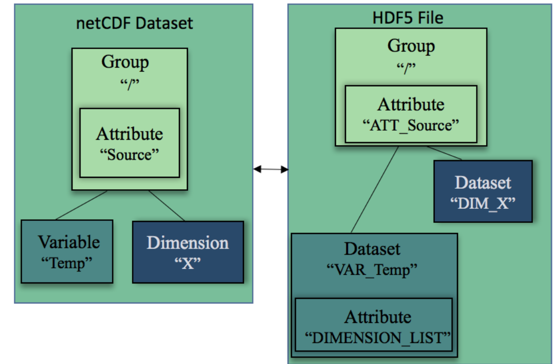

Figure 3. HDF5 schema for NetCDF/EFF.

1) EFF netCDF implementation: NetCDF [NetCDF, 2016] 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 EFF netCDF APIs (denoted by a '*_ff*' suffix) were derived from the original netCDF APIs, maintaining most of the original functionality. However, the possible use of filters with netCDF was disabled since this is currently not supported in the EFF version of HDF5.

The non-EFF 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 IOD transaction model. In addition, it introduces many special cases that must be handled, increasing the difficulty of implementation. Finally, there is currently no EFF implementation of the Dimension Scale APIs and implementing them would be difficult due to the transaction model.

Since backward compatibility was not of a concern, i.e. having netCDF/EFF datasets (containers/files) being accessed independently of the netCDF/EFF 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, Figure 3. 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 treating them as any other variable. Other EFF 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,
- Support for multiple server processes and nodes.

Additionally, the global EFF stack variables are passed as arguments to the NetCDF and from the application and allow access to all the EFF stack variables: read context identifier, version number, event stack identifier and transaction identifier.

2) EFF Parallel I/O implementation: A typical application library which uses netCDF is the software associated with the **A**ccelerated **C**limate **M**odeling for **E**nergy (ACME) [ACME, 2016]program. ACME uses the package Parallel I/O (PIO) [PIO, 2016] to perform I/O which, in turn, uses as its backend the netCDF file format. Since the global EFF stack variables are passed as arguments to netCDF, the EFF stack parameters are controlled from within PIO. This allows for a PIO API to call multiple netCDF APIs and to use the same or different transactions depending on which set of netCDF calls are made within the PIO API. The transaction number is initialized in the application code and is then automatically incremented as needed within the EFF PIO APIs. Hence, transaction management is handled in the PIO library, but it is still accessible to the application.

Exposing the event stack identifier allows the application to use asynchronous I/O. To do this, the application creates an event stack and passes it to an operation that can be asynchronous. At the point in the future when the application needs that operation to be finished, it can use HDF5 EFF APIs *H5ESwait* or *H5ESwait_all* to block until it is completed. Even if no event stack is passed to some metadata operations, netCDF will use an internal event stack to issue asynchronous operations that can run concurrently, improving performance. In this case, netCDF will wait on all operations before returning. Similar to the netCDF convention, all new EFF PIO C APIs are indicated by appending a "*_ff*" to the C function names. The Fortran EFF PIO APIs were implemented by overloading the current Fortran PIO APIs, hence if the optional parameters are not present then PIO well automatically default to the non-EFF netCDF APIs.

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 [Edwards et al., 2016]. 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, *EFF_init* (an EFF HDF5 API used to start the EFF stack) uses the MPI sub-communicator group so that only those processes involved in I/O will initialize the EFF stack. This initialization of the EFF 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.

V. 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 simply scale past systems. This work highlights challenges and issues that must be addressed when porting current application codes to a transaction model for I/O. The transaction model for exascale I/O is a novel technique, and POSIX developers may find programming in an environment in which data becomes readable only after commit quite unnatural. However, some relaxation, for example, to provide visibility of one's own uncommitted updates, may lead to an easier transition for application and middleware developers.

Acknowledgements

Funding for this project was provided through Intel subcontract #CW1998599 from Intel's contract #B613306 with Lawrence Livermore National Security.

References

[ACME, 2016] ACME (2016). http://climatemodeling.science.energy.gov/projects/accelerate climate-modeling-energy.

[Bent et al., 2012a] Bent, J., Faibish, S., Ahrens, J., Grider, G., Patchett, J., Tzelnic, P., and Woodring, J. (2012a). Jitter-free co-processing on a prototype exascale storage stack. *IEEE Symposium on Mass Storage Systems and Technologies*, (August 2016).

[Bent et al., 2012b] Bent, J., Grider, G., Kettering, B., Manzanares, A., McClelland, M., Torres, A., and Torrez, A. (2012b). Storage challenges at Los Alamos National Lab. *IEEE Symposium on Mass Storage Systems and Technologies*.

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

[Habib et al., 2014] Habib, S., Pope, A., Finkel, H., Frontiere, N., Heitmann, K., Daniel, D., Fasel, P., Morozov, V., Zagaris, G., Peterka, T., Vishwanath, V., Lukic, Z., Sehrish, S., and Liao, W.-k. (2014). HACC: Simulating Sky Surveys on State-of-the-Art Supercomputing Architectures. 4.

[HACC, 2016] HACC (2016). Retrieved from http://trac.alcf.anl.gov/projects/genericio.

[Isaila et al., 2016] Isaila, F., Garcia, J., Carretero, J., Ross, R., and Kimpe, D. (2016). Making the case for reforming the I/O software stack of extreme-scale systems. *Advances in Engineering Software*, 16(10):1–6.

[Keyes, 2011] Keyes, D. E. (2011). Exaflop/s: The why and the how. *Comptes Rendus Mécanique*, 339(2-3):70–77.

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

[Lofstead et al., 2013] Lofstead, J., Dayal, J., Jimenez, I., and Maltzahn, C. (2013). Efficient transactions for parallel data movement. *Proceedings of the 8th Parallel Data Storage Workshop on - PDSW '13*, pages 1–6.

[Mehta et al., 2012] Mehta, K., Bent, J., Torres, A., Grider, G., and Gabriel, E. (2012). 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*, pages 746–752.

[NetCDF, 2016] NetCDF (2016). Retrieved from www.unidata.ucar.edu/software/netcdf/.

[Nowoczynski et al., 2008] Nowoczynski, P., Stone, N., Yanovich, J., and Sommerfield, J. (2008). Zest : Checkpoint storage system for large supercomputers. *Proceedings of the 2008 3rd Petascale Data Storage Workshop, PDSW 2008*.

[PIO, 2016] PIO (2016). Retrieved from www.ncar.github.io/ParallelIO.