Optimized Scatter/Gather Data Operations for Parallel Storage

Latchesar Ionkov
Los Alamos National Laboratory
Los Alamos, NM 87545
lionkov@lanl.gov

Carlos Maltzahn
University of California
Santa Cruz, CA 95064
carlosm@cs.ucsc.edu

Michael Lang
Los Alamos National Laboratory
Los Alamos, NM 87545
mlang@lanl.gov

ABSTRACT

Scientific workflows contain an increasing number of interacting applications, often with big disparity between the formats of data being produced and consumed by different applications. This mismatch can result in performance degradation as data retrieval causes multiple read operations (often to a remote storage system) in order to convert the data. Although some parallel filesystems and middleware libraries attempt to identify access patterns and optimize data retrieval, they frequently fail if the patterns are complex.

The goal of ASGARD is to replace I/O operations issued to a file by the processes with a single operation that passes enough semantic information to the storage system, so it can combine (and eventually optimize) the data movement. ASGARD allows application developers to define their application's abstract dataset as well as the subsets of the data (fragments) that are created and used by the HPC codes. It uses the semantic information to generate and execute transformation rules that convert the data between the the memory layouts of the producer and consumer applications, as well as the layout on nonvolatile storage. The transformation engine implements functionality similar to the scatter/gather support available in some file systems. Since data subsets are defined during the initialization phase, i.e., well in advance from the time they are used to store and retrieve data, the storage system has multiple opportunities to optimize both the data layout and the transformation rules in order to increase the overall I/O performance.

In order to evaluate ASGARD's performance, we added support for ASGARD's transformation rules to Ceph's object store RADOS. We created Ceph data objects that allow custom data striping based on ASGARD's fragment definitions. Our tests with the extended RADOS show up to 5 times performance improvements for writes and 10 times performance improvements for reads over collective MPI I/O.

ACM Reference format:
Latchesar Ionkov, Carlos Maltzahn, and Michael Lang. 2017. Optimized Scatter/Gather Data Operations for Parallel Storage. In *Proceedings of PDSW-DISCS'17: Second Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, Denver, CO, USA, November 12–17, 2017 (PDSW-DISCS'17),* 6 pages.
DOI: 10.1145/3149393.3149397

PDSW-DISCS'17, Denver, CO, USA

DOI: 10.1145/3149393.3149397

1 INTRODUCTION

Advances in computer hardware have led to major increases of the size of numerical simulations run on supercomputers. Future exascale systems will need hundreds of petabytes of storage to satisfy the requirements for scratch space [9]. In order to perform at that scale, future supercomputers will likely employ complex hierarchies of both volatile and nonvolatile memories. Architectures such as burst buffers [14] deliver performance at the cost of complexity.

In the past, most data analysis and visualization was a post-processing step. Due to the scale as, well as the higher cost for data movement, *in situ* and in-transit data processing are more attractive. In many cases, the visualization and analysis applications need only a small subset of the data produced by the application, but since data layout is optimized to increase the simulation's performance, reading the required subset of the data may reduce overall performance of the storage system.

HPC applications store datasets in a single file (N-to-1), one file per process (N-to-N), or a few files (N-to-M). In many cases, as data is saved in files, the semantic metadata doesn't reach the storage system. Many storage systems try to find temporal patterns in the operations so they can predict the future operations and improve I/O performance, but with little knowledge of the overall structure of the data and the application's requirements, these predictions often fail [11].

ASGARD allows developers to define an abstract description of the data produced by the application. They can also provide definitions of the subsets (fragments) of the data used by each process, as well as by other visualization and analysis applications. ASGARD uses a data declaration language that has syntax familiar to most software developers. In addition to a predefined set of primary types, it supports multi-dimensional arrays as well as collections of related fields (i.e., records or structs). The fragments of the dataset are made up of subsets of arrays, records, or combinations of both.

Once the dataset and its fragments are declared, ASGARD provides functionality to query how to convert the content of one fragment to another. ASGARD defines *transformation rules*, which describe the conversion between two fragments. The values required for fragment materialization might not be located in a single fragment, so ASGARD allows the user to get a list of the fragments required to produce a specific fragment.

ASGARD is based on DRepl [13], which is designed to separate the application's view of the data from the storage layout and support divergent data replication. ASGARD uses the same dataset language and parser. It extends DRepl by providing support for dynamic fragment definitions, and functionality for transforming and gathering data from multiple data fragments. While DRepl provides a POSIX API for accessing the data as files, ASGARD avoids the file interface and focuses on the data layout in application memory and how to transform it optimally to the layout on persistent storage.

In order to decrease network usage, ASGARD splits the transformation rules into two parts. *Remote rules* are executed on the server where the source fragment is located and produce a compact representation of the data for the destination fragment. Once the compact representation is received, ASGARD applies *local rules* to copy the parts of the source fragment to the appropriate locations in the destination fragment. ASGARD, in a sense, provides an extension to the standard scatter/gather operations. The transformation rules allow compact definition of complex patterns. The combination of gather (on the side where the data is stored) and scatter (on the side where data is used) provides a powerful tool for transforming data from the format in which it is produced to a format suited for consumption. The fact that the fragments are defined in advance allows an active storage subsystem to perform optimizations that might be too slow or costly if run at the time the operations are executed.

In order to evaluate ASGARD's benefits, we modified Ceph's [17] RADOS object store system to support ASGARD's transformation rules for storing and retrieving data to objects. We also created a custom object class that allows developers to define an object that represents an ASGARD dataset and how it is partitioned into stripes on storage. Each of the stripes is defined as ASGARD's fragment. When a client needs to read or write a subset of the data, it can get a list of the stripe objects that contain the data as well as a pair of transformation rules that need to be applied to transform the requested subset to each of the stripes. The operations for each of the stripe objects are executed asynchronously to increase the overall performance. Our experiments show that ASGARD can improve the I/O performance by a factor of seven over both collective and non-collective MPI I/O.

2 RELATED WORK

Many projects try to improve the storage system's performance for non-contiguous access. The standard POSIX interface only allows reading and writing of contiguous file regions to/from non-contiguous memory buffers. Research projects like [5, 6] introduce and evaluate more advanced approaches to PVFS [3]. These include lists of I/O operations as well as data descriptions similar to the ones used by MPI I/O. In contrast to MPI, ASGARD is more general and not tied to a communication library allowing adoption in general-purpose storage systems.

MPI's I/O support has many similarities with ASGARD. It allows users to define memory and file data layouts, and also supports collective I/O operations that improve the performance by coalescing I/O operations across multiple MPI ranks before they are sent to the storage system. MPI I/O and ASGARD's transformations partially intersect. The collective MPI I/O operations help with some non-contiguous data access, especially if the overall I/O operations results in contiguous data access [4, 6, 7]. In the general case, the globally non-contiguous data access still doesn't scale well. Additionally MPI I/O performance gains usually require synchronous progress by all ranks, this is less likely at exascale.

Unlike MPI I/O, HDF5[10] stores the data definition within the file together with the data. This allows third-party applications to access it. HDF5 doesn't provide a definition for how the data is accessed by the applications, making it harder for HDF5 and storage systems to optimize data layout and improve I/O performance. Although HDF-5 can use MPI I/O, it usually doesn't perform well due to the internal structure of the HDF-5 files [18].

ADIOS [15] is I/O middleware that separates the data format from the application code by defining it in an XML file that can be modified depending on the cluster configuration in order to optimize performance. ADIOS' data description language is relatively simple, allowing variables from predefined data types, as well as multidimensional arrays containing values of a base type. Choosing XML as its data description language makes it extensible and easily parsed by computers, but difficult for humans. In addition to reading and writing data to buffers, ADIOS supports more complex aggregate operations that are useful for analysis and visualization. It supports different I/O subsystem transports, making it easy to select the best fit for the data patterns and system configuration.

3 DESIGN

In many cases there is not a single "right" data format that is best for storing application data. The goal of this work is to allow developers to easily define the subsets of the data that each process needs, and to provide support for transforming the existing data into those subsets. Such functionality allows the storage subsystem to continually optimize the layout of the data. To support this ASGARD needs semantic knowledge of the application data. ASGARD doesn't try to be an end-to-end solution; instead, it provides functionality that can be integrated into middleware and storage systems. It doesn't cover actual data storage or decisions on where and how data should be replicated in order to optimize I/O performance, but does provide the semantic information so these decisions can be made intelligently.

In addition to an API for dataset definition, ASGARD provides a declarative language that allows one to describe an abstract dataset and its subsets. Using an expressive dataset language makes it easier for scientific application developers to communicate and share the data with other applications. In order to ensure familiarity, we chose syntax similar to the type and data declarations used by languages such as C and C++.

We distinguish two major entities related to data and the way it is used and stored. *Dataset* is an abstract definition that describes the data types and data objects that are of interest of any application, regardless of whether it is the producer or consumer of the data. For scientific applications, datasets are usually big and consist of all data produced by simulations run on thousands of processors. *Fragment* is a subset of the dataset and defines the parts of the data that are of interest to a particular set of applications, an application, or one of the application's ranks.

While designing ASGARD we used the following assumptions:

- Processes create and use a subset of the application data;
- A process' fragment(s) don't change often and don't depend on dataset values;
- Storage systems can optimize performance by using information about the structure of data and the patterns by which data is accessed;
- The data is likely to be stored far from where it is used.

Although the main goal of ASGARD is to improve I/O performance for HPC workloads, we tried to make it general enough so it can also improve other common use cases. This wider applicability can allow ASGARD to be included in a broader range of storage systems.

3.1 DRepl Language

The DRepl language allows declaration of custom data types based on a set of primitive types, as well as composite types such as arrays or structs. Its syntax is loosely based on the syntax for type and variable definitions in the Go language [8], which is similar to C or C++. Details of the DRepl language are described earlier work [13].

3.2 Fragment Transformation

Once an abstract dataset and some fragment descriptions are defined, the user can generate *transformation rules* for converting data from one fragment to another. Generally, only part of the fragment's data will be available in another fragment (i.e., fragments may only partially overlap), and materializing a fragment may require data from multiple fragments.

Transformation rules are generated for a pair of fragments: a source and a destination fragment. They define what data from the source fragment is needed, and how to convert it to the format of the destination fragment.

The layout of a fragment is defined as a list of *blocks*. A block represents a compact region within the fragment. It has an offset from the beginning of the fragment, as well as a size. The blocks can be compound and contain references to other blocks. For example, a block that describes a two-dimensional matrix, contains a reference to a block that defines each element of the matrix. Within a fragment, each block is either a top-level block that defines the fragment's layout, or is referenced by exactly one other block. Additionally, if the data represented by the block is present in other fragments, the block contains a reference to the respective source/destination blocks.

Currently ASGARD defines three types of blocks: SBlock, TBlock, and ABlock. The simplest block, *SBlock*, defines a contiguous region that is always read or replicated as a whole entity. All DRepl primitive types are described as SBlocks. SBlocks don't have references to other blocks. In some cases an optimizer (see Section 4.1) can coalesce multiple adjacent SBlocks into a single, bigger SBlock.

A *TBlock* is a collection of other blocks, generally with different sizes. It corresponds to a `struct` composite type in the DRepl language. A TBlock contains a list of references to other blocks. Their offsets are relative to the offset of the TBlock that points to them. A TBlock can contain holes, or extra padding at the beginning and the end of it, depending on the values of the sub-blocks' offsets. The sub-blocks can't overlap.

Finally, *ABlock* defines a multidimensional array of identical blocks called elements. The definition of an ABlock includes its dimension, number of elements in each dimension, and element order. Currently ASGARD supports two element orders: row-major and column-major, but the design is flexible enough to support the addition of more types like a Hilbert curve [12] or a Z-order [16].

References to source/destination ABlocks in other fragments require additional information that specifies relation between the positions of elements in the block with elements of the source/destination block. For each dimension i, there are five integer values (a_i,b_i,c_i,d_i,idx_i). The element with index $(x_1,x_2,\ldots,x_n)$ in the original ABlock corresponds to an element $(y_1,y_2,\ldots y_n)$ in the source/destination ABlock, where

$$y_{idx_i} = \frac{a_i x_i + b_i}{c_i x_i + d_i}$$

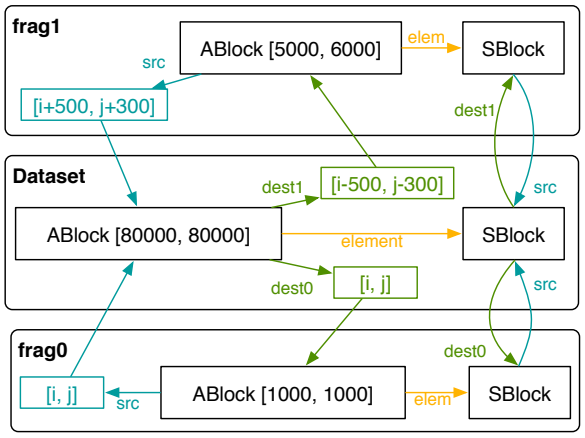

Figure 1: Example of a conversion map of two fragments

If the value of the expression is not a whole number, the element doesn't have a corresponding element in the source/destination block. Once the y_j values are calculated, the offset of the element is calculated using the element order for the source/destination ABlock.

In complex datasets, the elements of an ABlock can be TBlocks, which can in turn contain fields that are ABlocks, with TBlock elements, and so forth. Two fragments of the dataset can contain different subsets of elements and fields for each of the blocks, and ASGARD will recursively apply the correct transformations to convert one fragment to the other.

Figure 1 shows the internal representation defined for the abstract dataset, as well as the two fragments `frag0` and `frag1`.

```
dataset {
  var data[80000, 80000] float64
}
fragment frag0 {
  var ds[i:1000, j:1000] = data[i, j]
}
fragment frag1 {
  var a [i:5000, j:6000] = data[i+500, j+300]
}
```

The dataset has two blocks defined – an ABlock with 2 dimensions [80000,80000]. Each element of the ABlock is a SBlock. Fragment `frag1` contains an ABlock with 2 dimensions [5000,6000]. The ABlock is connected to corresponding ABlock in the dataset, where element $[i,j]$ from it corresponds to element $[i+500,j+300]$ in the dataset ABlock. Similarly, `frag0` defines a smaller ABlock with 2 dimensions: [1000,1000] and its element $[i,j]$ corresponds to the element with the same address in the dataset ABlock.

In distributed environments, where communication is expensive, it is beneficial to split the transformation rules into two parts. The *remote transformation rules* extract the necessary data from the source fragment and convert it into a compact intermediate buffer that can be efficiently transmitted over the network to where the data is needed. The *local transformation rules* use the data from the intermediate buffer and place it in the right format in the destination fragment.

The blocks in the fragments have the appropriate dataset blocks as sources, and accordingly the dataset blocks have the fragments'

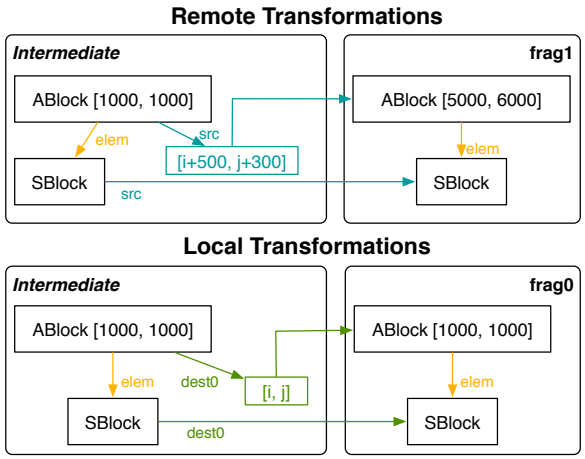

Figure 2: Example of local and remote transformation rules

blocks as destinations. Figure 2 shows the remote and local transformations for materializing fragment `frag0` from fragment `frag1`.

4 IMPLEMENTATION

The ASGARD implementation consists of two main modules: the DRepl language parser, and the transformation engine. The parser is optional, and can be used by applications for easier description of datasets and fragments. For supporting ASGARD's functionality, storage systems only need to include the transformation engine.

The transformation engine uses the transformation rules produced by the parser to convert data from a region in memory containing source fragment's data to a region that represents a target fragment that is being materialized.

The transformation library is compact (less than three thousand lines of C code) and can be easily incorporated in most storage systems. It provides support for serializing and deserializing transformation rules for easy transmission over the network.

Transformation rules are applied by iteratively transforming each top-level block from the source fragment to the blocks it references. For complex blocks, like TBlock and ABlock, the transformation is performed recursively for each field/element block they contain.

4.1 Optimizer

As the transformation rules can be complex, ASGARD's performance in the general case can be sub-optimal. There are many common cases when the transformation rules can be optimized further. We implemented some optimizations that cover the most frequently used patterns.

If sequential SBlock fields within a TBlock are copied to sequential fields in a destination TBlock, they can be replaced by a single SBlock with their combined size. If all fields from a TBlock are copied, the whole TBlock is replaced with an SBlock.

ABlock optimizations are currently performed only if both source and destination blocks are of the same element type. They include replacing a dimension of the ABlock with a TBlock if the required elements for the dimension are sequential. For example, if the transformation rules of a 3D floating-point matrix with size (100,100,100) copy the hyper-slab (10…50,10…50,10…50), the original ABlock is replaced by a 2D matrix with size (100,100) containing TBlocks with three fields: a hole of size `float[10]`, a SBlock of size `float[40]`, and another hole of size `float[50]`. Thus applying the transformation rules will be executed for ten thousand (slightly more complicated) elements, instead of one million. As with TBlocks, if all elements from an ABlock are copied, the optimizer replaces it with a SBlock with the same size.

4.2 Ceph Integration

We modified the Ceph distributed storage system to use ASGARD. The modifications are primarily in Ceph's object store device (OSD) layer, RADOS. We utilized Ceph's support for custom data objects in RADOS by creating a new class that defines an ASGARD dataset. For creation of a ASGARD object, the user provides a description of the abstract dataset as well as a list of fragment descriptions for how the dataset is to be partitioned into RADOS objects. The fragments are similar to the standard file striping supported by most parallel file systems. Using ASGARD data descriptions provides much greater flexibility on how the data is partitioned into OSD objects. The dataset object doesn't contain any of the actual data; rather, it only contains the DRepl dataset definition as well as the names and ASGARD definitions of the stripe fragments that contain the data. In order to access data from Ceph's ASGARD objects, the user needs to define a fragment and query the Ceph ASGARD class, which stripe fragments contain the requested data. The result of the query is a list of stripe fragments and transformation rules on how to convert their data to the requested subset. The operation is done once during the initialization stage of the application. Reading and writing data goes directly to the RADOS objects that represent the stripe fragments.

The second part of the Ceph modification is introducing two new operations on RADOS objects. The original RADOS supports accessing sequential regions of an object defined by offset and length. We added two operations *dread* and *dwrite*, that use ASGARD transformation rules to define what data from the object is required and how to convert it to a compact buffer that is passed to/from the client. The transformations are applied on the OSD instances that contain the data, decreasing the number of operations over the network, increasing the overall throughput and decreasing the I/O latency. The client has an opportunity to execute multiple operations asynchronously, potentially to multiple OSD instances, and therefore utilize the full potential of the Ceph RADOS cluster.

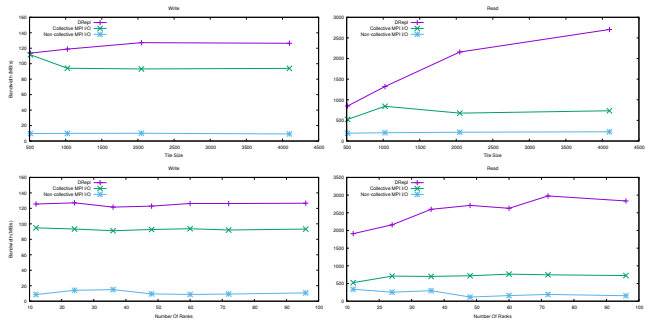

Figure 3: MPI Tile I/O performance

5 RESULTS

In order to evaluate ASGARD's performance, we set up a small Ceph cluster with 4 OSD servers and 1 metadata server, running on 4 nodes. The nodes have an eight-core Intel Xeon processor and 128GB of RAM. We used Ceph's default settings. Each object store instance was using a single rotational SATA disk with a capacity of 4 TB. There were 8 additional nodes that we used during the tests. All nodes were connected with an Infiniband EDR interconnect. We used OpenMPI as our MPI library.

The goal of our experiments is to demonstrate the benefits of offloading complex data transformations to the storage system. We compared ASGARD to MPI I/O, as both provide similar functionality and are relatively easy to use. The first test we ran was the MPI Tile I/O benchmark from the Parallel I/O Benchmarking Consortium [2]. We modified the code and added support for Ceph ASGARD objects. We ran the code on total 12 nodes, including the 4 that run Ceph. Tile I/O partitions a 2D array of data into multiple MPI ranks. Each rank is responsible for reading or writing a tile of the whole array. We varied the size of the tiles from 512x512 to 4096x4096 elements while running 8 ranks per node (96 total ranks). In order to evaluate the bottlenecks of the setup, we also ran the tests with the same tile size (4096x4096), but varying the number of ranks per node, from 1 to 8 (total 96 ranks).

Figure 3 shows the read and write performance while varying the tiles and ranks. ASGARD outperforms both read and write for the non-collective and collective MPI I/O operations. We used Ceph's monitoring infrastructure to inspect the number of operations and read/write bandwidth going to the OSD instances. ASGARD's implementation used the fewest number of operations per second. While both ASGARD and collective MPI I/O had similarly high data size per operation, non-collective MPI I/O used a high number of operations with much lower data size per operation. During write operations, in addition to writing data to the OSDs, non-collective MPI I/O tests were also reading a lot of data.

We also evaluated I/O performance using the HPIO benchmark [1]. It is highly customizable and allows I/O workloads that support contiguous/noncontiguous data layout both in memory and in files. For these tests we run the benchmark only on the 8 nodes that were not running Ceph, with 8 ranks per node (total 64).

For the first evaluation we tried to mimic a use case where the dataset consists of a one-dimensional array of structs, containing three 64-bit values. The application is only concerned with one of

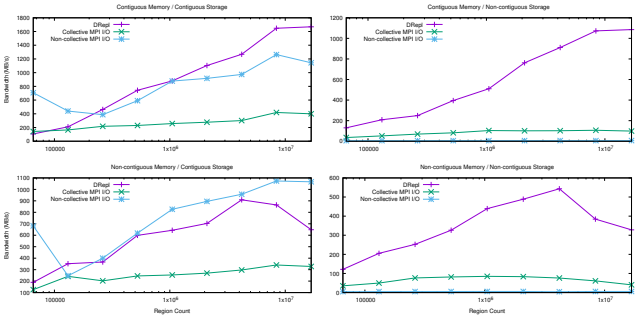

Figure 4: HPIO read performance

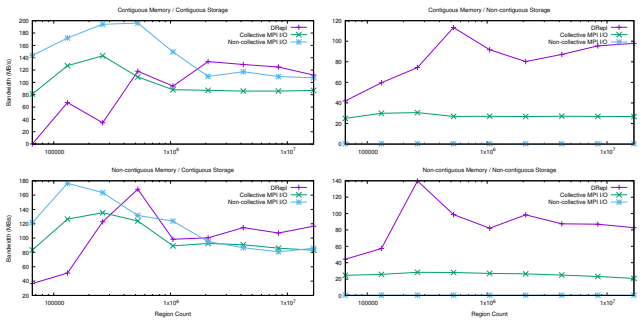

Figure 5: HPIO write performance

these values. When the memory or the file layouts are contiguous (C), only the value of interest is stored. When they are non-contiguous (N), all data is stored, but only the value of interest is accessed. Figure 4 shows the results for reading, while varying the size of the array. For contiguous reads from storage, non-collective MPI I/O access performs well, as Ceph file system is well-optimized for reading the contiguous file layout and most of the data is prefetched to the local cache. ASGARD's performance is comparable, but can probably be improved with further optimization. When data is read from the file in non-contiguous patterns, both collective and non-collective MPI I/O degrade significantly, by a factor of 100 for the non-collective case. ASGARD clearly outperforms both MPI I/O modes and is up to ten times better than collective MPI I/O. Figure 5 shows similar results for write performance. ASGARD performs noticeably worse for very small writes, because the overhead for sending the transformation rules, as well as encoding and decoding them, becomes the main bottleneck. For non-contiguous file patterns, ASGARD performs up to five times better than collective MPI I/O.

6 CONCLUSIONS

ASGARD provides a simple language for describing complex scientific datasets and subsets. It creates a compact representation of the transformations required for data conversion between fragments. ASGARD is designed for distributed systems and intelligent storage elements. It optimizes the size of the data sent over the network by offloading some of the conversion to the storage system. The results demonstrate that complex scatter/gather transformation rules allow superior performance without synchronous progress mandated by collective MPI I/O.

The availability of rich semantic information about the dataset as well as the separation between the fragments' declarations and the data conversions provide many opportunities for performance optimizations. Results showed that ASGARD reduced the number of operations going to the the OSD per data transferred.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, through the Storage Systems and Input/Output (SSIO) for Extreme Scale Science Program under Award Number ERKJ312, Program Manager Lucy Nowell. This paper has assigned LANL publication number: LA-UR-17-21363.

REFERENCES

[1] HPIO Benchmark. http://users.eecs.northwestern.edu/~aching/research_webpage/hpio.html

[2] Parallel I/O Benchmarking Consortium. MPI Tile I/O Benchmark. http://www.mcs.anl.gov/research/projects/pio-benchmark

[3] Philip H Carns, Walter B Ligon III, Robert B Ross, and Rajeev Thakur. 2000. PVFS: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux Showcase & Conference-Volume 4*. USENIX Association, 28–28.

[4] Avery Ching, Alok Choudhary, Kenin Coloma, Wei-keng Liao, Robert Ross, and William Gropp. 2003. Noncontiguous i/o accesses through mpi-io. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*. IEEE, 104–111.

[5] Avery Ching, Alok Choudhary, Wei-keng Liao, Rob Ross, and William Gropp. 2002. Noncontiguous i/o through pvfs. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE, 405–414.

[6] Avery Ching, Alok Choudhary, Wei-keng Liao, Lee Ward, and Neil Pundit. 2006. Evaluating I/O characteristics and methods for storing structured scientific data. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 10–pp.

[7] Kenin Coloma, Avery Ching, Alok Choudhary, Wei-keng Liao, Rob Ross, Rajeev Thakur, and Lee Ward. 2006. A new flexible MPI collective I/O implementation. In *Cluster Computing, 2006 IEEE International Conference on*. IEEE, 1–10.

[8] Alan A.A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language* (1st ed.). Addison-Wesley Professional.

[9] Gary Grider. 2011. Exa-Scale FSIO Can we get there? Can we afford to?. In *7th IEEE International Workshop on Storage Network Architecture and Parallel I/Os*.

[10] The HDF Group. Hierarchical data format version 5. http://www.hdfgroup.org/HDF5

[11] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. 2013. I/O acceleration with pattern detection. In *Proceedings of the 22nd international symposium on High-Performance Parallel and Distributed Computing*. ACM, 25–36.

[12] David Hilbert. 1891. On the continuous mapping of a line on a surface part. *Mathematical Annals* 38, 3 (1891), 459–460.

[13] Latchesar Ionkov, Michael Lang, and Carlos Maltzahn. 2013. Drepl: optimizing access to application data for analysis and visualization. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–11.

[14] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. 2012. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 1–11.

[15] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and Integration for Scientific Codes Through the Adaptable IO System (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments (CLADE '08)*. ACM, New York, NY, USA, 15–24. DOI : https://doi.org/10.1145/1383529.1383533

[16] Guy M Morton. 1966. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York.

[17] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[18] Weikuan Yu, Jeffrey S Vetter, and H Sarp Oral. 2008. Performance characterization and optimization of parallel I/O on the Cray XT. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 1–11.