Expediting Scientific Data Analysis with Reorganization of Data

Bin Dong, Surendra Byna, and Kesheng Wu
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA, 94720
Email: {DBin, SByna, KWu}@lbl.gov

***Abstract*—Data producers typically optimize the layout of data files to minimize the write time. In most cases, data analysis tasks read these files in access patterns different from the write patterns causing poor read performance. In this paper, we introduce Scientific Data Services (SDS), a framework for bridging the performance gap between writing and reading scientific data. SDS reorganizes data to match the read patterns of analysis tasks and enables transparent data reads from the reorganized data. We implemented a HDF5 Virtual Object Layer (VOL) plugin to redirect the HDF5 dataset read calls to the reorganized data. To demonstrate the effectiveness of SDS, we applied two parallel data organization techniques: a sort-based organization on a plasma physics data and a transpose-based organization on mass spectrometry imaging data. We also extended the HDF5 data access API to allow selection of data based on their values through a query interface, called SDS Query. We evaluated the execution time in accessing various subsets of data through existing HDF5 Read API and SDS Query. We showed that reading the reorganized data using SDS is up to 55X faster than reading the original data.**

I. Introduction

Large-scale data analysis is key to scientific discoveries today [2]. A recent example is the confirmation of the existence of Higgs boson - the "God particle," where petabytes of high-energy collision data from Large Hadron Collider (LHC) experiment was collected and analyzed [1]. Another example is the Square Kilometre Array (SKA), designed to survey the sky ten thousand times faster than ever before, will produce in excess of one exabyte of raw data per day [11]. These large data sets are stored in files and have to be read from disk to perform analysis tasks. Depending on the amount of data to be analyzed and how the data is organized on file systems, the reading time dominates the total analysis time. Therefore, improving the performance of reading data is critical to data intensive scientific applications [16].

In most scientific analyses, data read performance lags significantly behind write performance because the organization of data files are determined by the data producers who are more concerned about writing as fast as possible. For example, in a plasma physics application, the simulation code VPIC (described in Section II-B) [6] writes out data in the order as it is stored in memory, while a common analysis operation of that data is to find the characteristics of highly energetic particles with their energy above a given threshold. The data records of these high energy particles are scattered around the data file causing poor read performance.

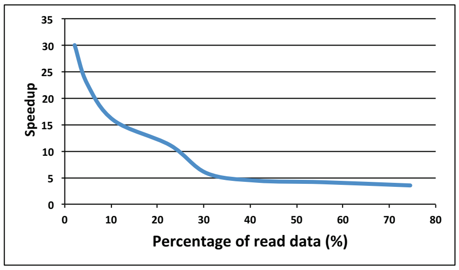

Fig. 1. Speedup in accessing a reorganized data set compared with accessing a raw data set from a single disk. A subset of ≈561 MB data from plasma physics simulation is used in this test. The reorganized data set is sorted according to energy of the particles.

Reorganizing data files for improving read performance is a desriable strategy. Fig. 1 shows a motivating example, where the performance speedup in accessing a VPIC dataset organized on a single disk to match the read pattern. In this case, we sorted the data records according to energy of the particles and read the high energy particles from the disk. The speedup shown is the ratio of the time needed to extract the high energy particle data from the original dataset to the time needed to extract the same data from the sorted dataset. When a small fraction of the data records are read, using the reorganized data is about 30X faster than using the original data. Even when a large fraction of data is accessed, reading from the sorted dataset is faster because of avoiding to sift through the data in memory.

Many research efforts [22], [21], [18], [3], [14] explored optimizing data organization by deriving an optimal file layout strategy for a certain access pattern. However, analysis tasks have to be aware of the reorganized layout to read the data. Moreover, read access patterns to a single file may vary over time and there is no universal data organization that is effective for all read access patterns. Storing data manually in different organizations based on read patterns and changing analysis codes for reading the newly organized data is impractical. An automatic framework that identifies dominant read patterns and reorganizes data automatically is needed. Towards that goal, we are developing a system to enable dynamic reorganization of the data based on established data read patterns. This paper introduces the first step with the Scientific Data Services (SDS) framework. The SDS is a client-

978-1-4799-0898-1/13/$31.00 ©2013 IEEE

server architecture implemented using the new Virtual Object Layer (VOL) of HDF5. Based on the access patterns of two data intensive applications: VPIC and OpenMSI, we designed and implemented parallel sorting-based and parallel transpose-based reorganizations. Using the VOL, analysis tasks read data transparently. The key contributions of this work include:

- Design of the Scientific Data Service (SDS) framework to enable data reorganization and transparent data accesses.
- Design and implementation of a high-level parallel querying API to perform value-based queries for a target data set.
- Implementation a HDF5 VOL plug-in to transparently read data using HDF5 read API.
- Implementation of parallel sorting-based and transpose-based reorganization algorithms.

This rest of the paper is organized as follows: Section II discusses two science use cases where reorganization of data is beneficial for analysis. Section III introduces the SDS architecture and its components. In that section, we also explain sorting-based and transpose-based data reorganizations. Section IV discusses experimental platform. Section V presents the performance results. In Section VI, we review related work and in Section VII, we provide concluding remarks and future directions.

II. HDF5 VOL and Analysis Use Cases

In this section, we briefly discuss the HDF5 VOL and the two use cases from real scientific applications we used in this study.

A. HDF5 Virtual Object Layer (VOL)

HDF5 is a popular data format library [10] used by many scientific applications around the world. VOL is a new abstraction layer internal to the HDF5 data model, right below the public HDF5 API [7]. The VOL intercepts all HDF5 API calls that could potentially touch the data in a file and forward those calls to a plugin object driver. The VOL provides flexibility to store or read data with the use of custom plugins. For example, a plugin could store data objects in a different file format from the native HDF5 file format, or in a different file layout from the original organization. In this study, we develop a plugin to read data from optimally reorganized data instead of originally stored organization to achieve better performance.

B. Use case: Vector Particle-in-Cell (VPIC) Data Analysis

Our recent VPIC simulation modeled magnetic reconnection, an important mechanism of space weather [6], by tracking two trillion particles on the Hopper supercomputer at National Energy Research Scientific Computing Center (NERSC). Particle properties of interest in this simulation include spatial locations in x,y, and z dimensions, momentum in the three dimensions, and energy. The data size of each particle is $28\ bytes$. The number of data records in a snapshot of the simulation gradually increases from 1 trillion to about 1.5 trillion and the data file size grows from $30TB$ to $42TB$. In this study, we use a subset of these data files, where the energy value of particles is greater than 1.1. The sizes of subset data files are between 2.5 TB and 5 TB per time step.

One of the analysis tasks of this data is to visualize the high energy particles. The domain scientists identify different thresholds for high energy values. Visualizing these particles requires reading energy values and global coordinate datasets in three dimensions. A full scan of the energy data is needed to identify the particles with energy values above a given threshold, i.e. multiple full scans are needed for multiple threshold value searches. The positions of the high energy particles are then read from data files. Because the high energy particles are spread out in the data files, the function to extract the selected x, y, and z values are effectively reading through all the data records is extremely slow.

C. Use case: Mass Spectrometry Imaging Data Analysis

Mass spectrometry imaging (Mass Spec) [17] is a key technology in various biological science applications, such as understanding metabolism. Mass spec analysis visualizes the distribution of molecules at each location within a sample, which is reconstructed as a molecular image showing the localization and abundance of specific molecules (e.g. lipids, proteins, small metabolites, etc.) comprising a sample. The size of an image collected by mass spec devices currently ranges from 10 GB to 50 GB. This size is expected to grow to hundreds of gigabytes as the resolution of mass spec instruments grows. Each dataset contains hundreds of thousands of these images, making the overall size of a dataset multiple terabytes.

An analysis of mass spec dataset requires accessing a subset of data. The access can be denoted as a triple [X, Y, m/z], where X and Y are spatial positions, and m/z is the mass-to-charge ratio. In a three dimensional data (with 100,000 2D images), the analysis application may access a full spectrum of certain coordination (e.g. [4, 4, :]) to compute the average spectrum. Another common access pattern is to access a range of consecutive images (e.g. [:,:,10000:20000]). In the first case, the row-major layout is employed by most I/O libraries, such as HDF5, can deliver a good performance. But, once the file is stored in row-major layout, accessing of consecutive images will perform a large number of small and non-contiguous disk accesses that typically results in poor performance.

In this study, we focus on reorganizing the above mentioned datasets to match the read patterns and to expedite analysis tasks.

III. Scientific Data Services

In this section, we introduce the Scientific Data Services (SDS) framework, explain the main components, and discuss the current implementation of data organizations.

A. Overview of SDS framework

The goals of SDS is to manage reorganization of data that benefits various read access patterns and to improve data reading speeds during analysis. Fig. 2 shows an overview of the SDS framework. The main components of the framework are SDS Server and SDS Client. The SDS server is responsible for reorganizing data, for managing metadata of the reorganized

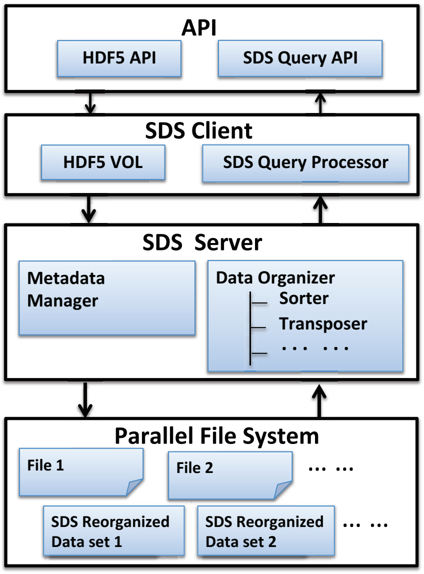

Fig. 2. An Overview of SDS Framework. The main components are SDS Query API, SDS Client, and SDS Server. We currently implement two data organizations using Sorter and Transposer.

and original datasets, and for directing read requests to the optimized reorganized datasets. Current implementation of the SDS Server performs data reorganization on previously written data files and manages metadata related to the location of original and reorganized data files. The API for using SDS is simply HDF5 public API for accessing datasets [10]. SDS also provides a query API, called SDS Query, for performing value based queries on datasets. The SDS client, built as a HDF5 VOL plugin, intercepts HDF5 API calls for reading datasets and then for redirecting them to reorganized datasets that will give better performance. The SDS Client also processes queries coming from the SDS Query API and contacts the server for implementing the queries. The API and the SDS clients are packaged as libraries that applications can dynamically link and run. In this paper, we focus on decribing the implementation of a basic skeleton of the SDS framework and the two data reorganization tools, namely, Sorter and Transposer. The Sorter orders datasets based on a specific variable and the Transposer transposes a 3D dataset to improve the contiguity of read accesses. Current implementation calls the data organizer functions with the use of a driver function that reorganizes data in offline fashion. In future, we will move the driver function into the SDS Server to automate selection of optimal organization based on read patterns and to execute the selected reorganization.

B. SDS VOL Plugin and SDS Query API

As many scientific applications use HDF5 API to store and access data, the SDS framework supports HDF5 API for reading data from HDF5 files. The `H5Dread` function supports reading all or part of a dataset. In addition to supporting the HDF5 read API, we add a querying interface to select data satisfying certain range conditions. Even though scientific data sets are large, the essential information is often contained in a relatively small number of data records. For example, in climate data of petabytes in size (10^{15} bytes), the critical information related to important events, such as hurricanes, might occupy only a few gigabytes (10^9 bytes). Often these events can be determined through a set of conditions on variables such as location, velocity, and intensity. Using the querying interface, users can specify these conditions and retrieve the data records satisfying the conditions. Compared to the common option of reading all data records into memory and then filtering in memory, SDS could perform the selection operations using indexing techniques and significantly reduce the time needed. Our current implementation searches for one variable with a query condition, such as "$\texttt{value1} < \texttt{energy} < \texttt{value2}$".

The SDS Query Interface is implemented as a function called `SDSQuery` that executes using parallel computing resources. An example of applying SDSQuery on a VPIC dataset is shown in Fig 3, where two processes call SDSQuery with same query parameter "$Energy > 1.4$" at the same time. The application provides SDSQuery with the dataset id along with the query condition. After being evaluated by the Query Processor of the SDS Client, SDSQuery returns to each process the offset and the size of data records it is responsible for reading in a sorted dataset. Note that the sorted dataset is a reorganized dataset by the Sorter component. The returned values of the SDSQuery are then used for reading the data.

C. SDS Server and Reorganization

The SDS Server performs data reorganizations and manages metadata for the reorganized datasets. We have implemented a sorting-based organization and a transpose-based organization of data. These two organizations support the two use cases mentioned in Section II.

1) Parallel-Sorting-based Reorganization.: A common data access in scientific applications such as VPIC is to read multiple variables, where one of the variable's values are continuous in an ascending or in a descending order. However, these continuous values may be scattered around the whole dataset. To improve the performance of this read operation, an approach is to sort the data in a data set and then store the sorted data in a new file. Note that this strategy is widely used in relational database management systems, where a data table is typically ordered according to a primary key.

To sort the data, we implemented a parallel sorting algorithm based on the classic Sample Sorting algorithm [13] using MPI. In this algorithm, the whole dataset to be sorted is partitioned into chunks and assigned to MPI processes. Each process sorts its own chunk of data using the quick sort algorithm. Each process then sends some samples to the root process. The root process decides pivots from all samples and broadcasts the pivots to all processes. Based on the pivots, each process then exchanges data using all-to-all communication. In order to avoid the congestion on a single node, only half processes send their data and the other half processes receive the data at any given time. Finally, each process sorts its own data again and writes the sorted data to a HDF5 file. We also store the minimum and maximum values and starting and ending offsets of each chuck to help query processing later.

2) Transpose-based Reorganization.: A multi-dimensional array in HDF5 is typically stored with the row-major ordering. Given a 3D array with dimensions X, Y, and Z, X is the slowest

```
                 Process 0
...
global_query = "Energy > 1.4";
file_id      = H5Fopen("filename.h5", ...);
dset_id      = H5Dopen(file_id, "datasetname", ...);
space_id     = H5Dget_space(dset_id)
SDSQuery(dset_id, global_query, &my_size, &my_offset...);
H5Sselect_hyperslab(space_id,..., my_offset, my_size,...);
buf          = malloc(my_size * sizeof(data type));
H5Dread(dset_id,...,  buf);
...
```

```
                 Process 1
...
global_query = "Energy > 1.4";
file_id      = H5Fopen("filename.h5", ...);
dset_id      = H5Dopen(file_id, "datasetname", ...);
space_id     = H5Dget_space(dset_id)
SDSQuery(dset_id, global_query, &my_size, &my_offset...);
H5Sselect_hyperslab(space_id,..., my_offset, my_size,...);
buf          = malloc(my_size * sizeof(data type));
H5Dread(dset_id,...,  buf);
...
```

Fig. 3. An example of SDSQuery with two processes executing in parallel using the SDS Query API.

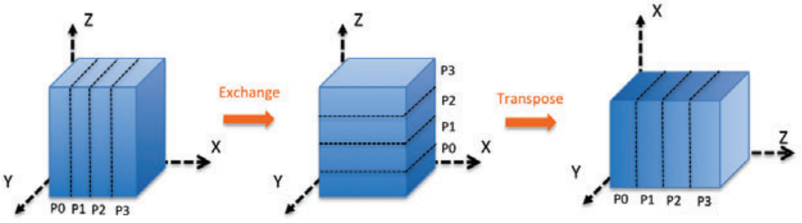

Fig. 4. An example of transposing a 3D array with four processes.

varying dimension and Z is the fastest varying dimension. In this organization, all values with the same X-coordinate (also known as an X plane) are stored contiguously in the data file. Similarly, a line with the same X and Y coordinates (an X-Y line) is contiguous in the file. This layout is optimal for the accessing the X planes and X-Y lines, however, accessing other subsets produce non-contiguous accesses of the data file. For example, accessing a continuous range of image from the mass spec data (i.e., the last dimension of a 3D array) always has poor performance. In order to improve the performance of such noncontiguous access, one method is to transpose the array.

Fig. 4 gives an example of transposing a 3D array by swapping the X and Z dimensions. This example involves four process (P0, P1, P2, and P3). To perform contiguous reads on the original array, we partition the file along the X-axis, as shown in the left part of Fig. 4. Then, all processes use all-to-all communication to exchange their data. After that, the chunks on all processes should be a partition along Z-axis, as shown in the middle of Fig. 4. The transpose algorithm uses the fast memory-to-memory exchanges to allow both input and output operations to be large contiguous I/O operations, and therefore reducing the time needed for the transpose-based reorganization. Finally, each process apply a local matrix transpose between X and Z, which makes the access along Z continuous.

D. SDS Client

The SDS Client contains methods to process the API calls from HDF5 dataset access API and SDS Query API. To process the HDF5 calls, we have developed a HDF5 VOL plugin. We added SDS Query API functions to HDF5 API to perform value-based selection in HDF5 datasets and extended the VOL plugin to handle data queries.

Fig. 5 shows common interactions between an SDS Client and the Server. In the current implementation, the Server performs data reorganizations offline based on user directions. The SDS Client extracts the details of a read function call. The details include the name of the file, the id of the HDF5 dataset, and the query condition or the offsets for the data to be read. It then contacts the SDS Server to obtain the metadata of the dataset. If there exists an optimal reorganized dataset that can give the better performance than accessing the original file, the Server sends the reorganized data location. If an optimal reorganized file is not available, the Server returns a NULL pointer. Upon receiving the metadata, the Client reads the requested data either from the original file or from the reorganized data set.

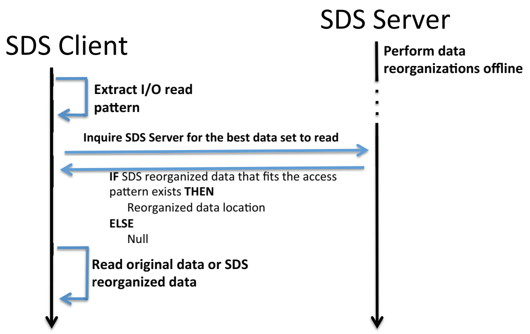
SDS Server
SDS Client
Perform data reorganizations offline
Extract I/O read pattern
Inquire SDS Server for the best data set to read
IF SDS reorganized data that fits the access pattern exists THEN
Reorganized data location
ELSE
Null
Read original data or SDS reorganized data

Fig. 5. Interactions between an SDS Client and the SDS Server

1) SDS Query Processor: The first task of the SDS query processor is to parse the query string received from the SDS Query interface. Then, SDS query will send the extracted information to the SDS server. Based on the response from SDS server, SDS Query Processor decides whether to take more actions about the query. For example, in the case of parallel sorting-based reorganization, SDS stores the minimum and maximum values of each chunk and their corresponding offsets as part of the SDS metadata. When a query requests for a sorted variable, the metadata could be used to locate the actual offsets and the data size of the value satisfying the query. Furthermore, the SDS Query Processor will partition these values among the active processes to provide load-balancing. With the parallel transpose based reorganization, SDS Query

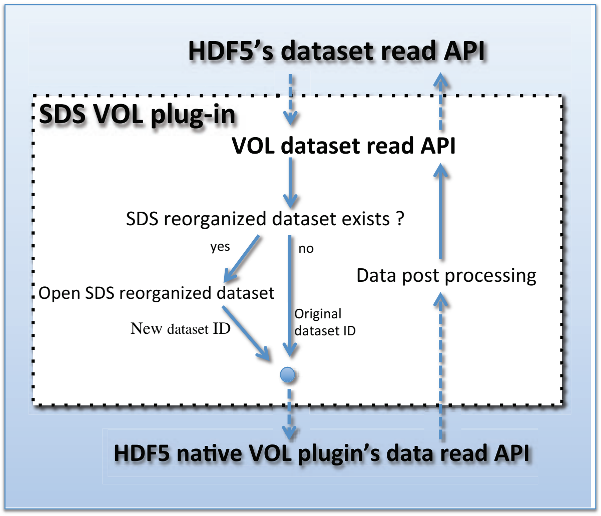

Fig. 6. Internal interactions in SDS VOL Plug-in

Processor will choose the version that requires the least amount of I/O time based on the ordering of the dimensions.

2) SDS VOL Plugin: We employ the HDF5 VOL to forward the data accesses to the SDS server. As Fig. 6 shows, our SDS VOL plugin still relies on HDF5 native functions to read the data values from the reorganized or the original datasets. The key operation performed by our plugin is to determine if an effective organization of data is available. If it exists, the plugin reads the data from the alternative location. Otherwise, the file access request is passed to the HDF5 native VOL plugin. In some cases, additional post processing may be required before the data is returned to user memory. For example, when working with the transposed data, there may be a need to reshuffle the dimensions of the data in memory so that the dimensions in the output are in the same order as in the original data.

IV. System Configuration

We have conducted all our experiments on the NERSC Cray XE6 supercomputing system, named Hopper [1]. The system has $6,384$ compute nodes, with two 12-core AMD 'MagnyCours' 2.1 GHz processors and at least 32 GB memory per node. All experiments used 16 cores on each node, and at most 16000 cores and 1200 cores are employed to perform the parallel data reorganization and value-based query tests, respectively.

We used a Lustre file system, exported as directory */scratch2*, for storing and reading data file. The file system consists of 13 LSI 7900 disk controllers. Each disk controller is served by two I/O servers, called OSSs (Object Storage Servers), and each OSS hosts six OSTs (Object Storage Target). There is a total of 156 OSTs, which can be considered as a software abstraction of a physical disk. The data files used in the following tests are striped across at most 144 OSTs instead of the default stripe count of 2 OSTs. The striping size for each file is 1MB.

[1] http://www.nersc.gov/systems/hopper-cray-xe6/

V. Results

As explained in Sections II-B and II-C, an analysis of VPIC data involves querying for high energy particles and an analysis of mass spec data involves accessing various subsets of a full dataset. In this section, we describe the data used for our experiments, compare performance of accessing data with SDS with that of the original HDF5 calls, and evaluate scaling behavior of SDS performance for both use cases. We also present the costs of reorganizing data; however, these reorganizations are performed offline and do not affect data analysis performance.

A. Evaluation of SDS Query Processing

1) VPIC Data: We have used five VPIC data files that represent the data from five time steps of our VPIC simulation. The size of the five HDF5 files are 2.5TB, 2.9TB, 3.3TB, 4.0TB, and 5.0TB. Each file contains seven variables: Energy, X, Y, Z, $U_{\|}$, $U_{\perp,1}$, and $U_{\perp,2}$, among which Energy, X, Y, and Z are frequently used together. Analysis of high energy particles requires reading the X, Y, and Z coordinates along with the Energy values. The remaining three variables are not needed for the analysis we tested. To query high energy particles with a given threshold, SDS created partial replica of the original dataset containing the required four variables sorted based on the Energy values.

TABLE I. Size of the data satisfying each query in a 5TB VPIC dataset

Query	Energy $<$ 1.15	1.15 $<$ Energy $<$ 1.4	Energy $>$ 1.4
Size(GB)	1927.00	883.25	2.30

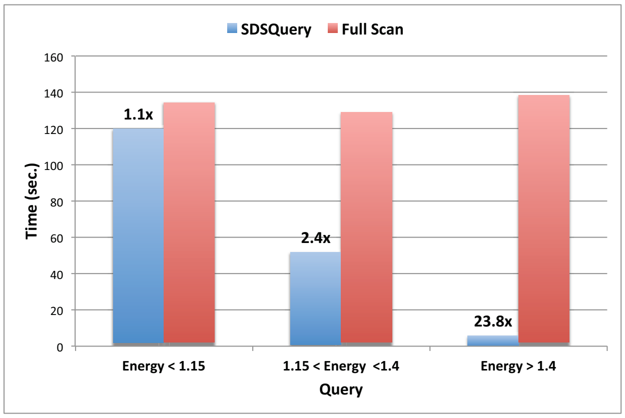

Fig. 7. Performance comparison between full-scan and SDSQuery with a 5TB VPIC data set

2) Performance of SDS Query Processing: We evaluate performance of querying for data that satisfies a given condition on Energy variable of the 5 TB VPIC data file. We evaluate three queries: "Energy <1.15", "Energy $>$ 1.4", and "1.15 $<$ Energy $<$ 1.4". The amount of data accessed in each query is shown in Table I. We compare the time for retrieving data for all four variables from the original data file to the time for executing an equivalent query using SDS that reads a reorganized data file. The former approach reads all data related to the four variables into memory and then searches for the desired data, where the Energy value meets the given threshold. In SDS, the query is expressed using the SDS Query

API and the SDS Server sends the coordinates of data where the threshold is met to the Client, and the Client simply reads contiguous chunks of data from the sorted data file.

Fig. 7 compares the time for reading from the original data, labeled 'Full Scan', and that for executing SDSQuery. Both approaches use 8,000 MPI processes, where each MPI process is mapped to a CPU core. The speedups with accessing the reorganized dataset with SDSQuery are: 1.1X, 2.4X, and 24X, respectively, for the three queries. The full scan approach roughly takes the same amount of time in all three cases. The SDSQuery execution takes significantly small amount of time in reading the smallest fraction of data. Overall, we observe that SDSQuery is beneficial in all cases. The performance benefit is significantly higher in accessing small fractions of the data because SDS can directly read a fraction of data that satisfies query conditions.

3) Scalability of SDSQuery: To evaluate the scalability of querying functionality of SDS and the overhead of query processing, we ran a query with the same condition using different number of CPU cores. Specifically, we use the query "`1.15 < Energy < 1.4`" and the 2.5TB dataset for this experiment. The size of the data that needs to be read for this query is 395 GB (i.e. 30% of the dataset). We scaled the number of cores to run the SDSQuery from 400 to 1200, with 200 increments. Fig. 8 shows that the overhead of SDSQuery and the total time in retrieving the data that satisfies the condition. We can observe that as the number of cores increases, the time to read data decreases. We also observe that the query processing time is a small fraction of the overall time, i.e. the overhead of SDS framework is negligible.

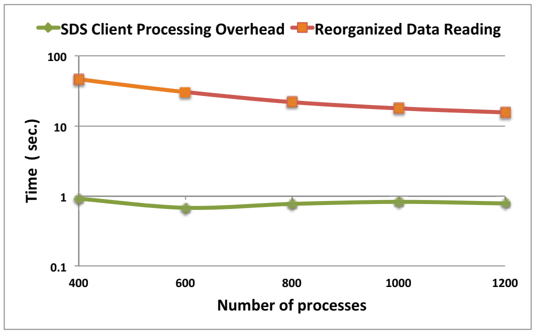

Fig. 8. Scalability of SDS with the "`1.15 < Energy < 1.4`" query on a sorted 2.5TB VPIC file

4) Cost of sorting-based reorganization: For running the queries mentioned in the above results, SDS needs the data to be sorted. While the cost of reorganization is not a significant factor during analysis because SDS performs reorganization of data offline, for evaluation purposes, we tested the cost of sorting-based reorganization explained in Section III-C with five VPIC files. The sizes of these file are 2.5TB, 2.9TB, 3.3TB, 4.0TB, and 5.0TB. The experimental results are shown in Fig. 9. We split the total time to show the time to read data from the original file, perform sorting (Exchange), write the new data set, and other costs including MPI related overhead. We can easily identify that the sorting based reorganization algorithm spends most time in exchanging the data among processes. A major reason for this overhead is that even our reorganization algorithm sorts only one variable (Energy), the VPIC analysis application needs to extract the certain Energy range with their corresponding locations (X, Y, and Z). Hence, in the data sorting process, MPI processes need exchange three extra variables. While we see that the parallel sorting algorithm works fine as the amount of data increases with different data sets, exchange of four variables has the most overhead. As this reorganization is executed offline, this overhead does not affect analysis applications. We are exploring other strategies for reducing the SDS Server overhead in performing sorting.

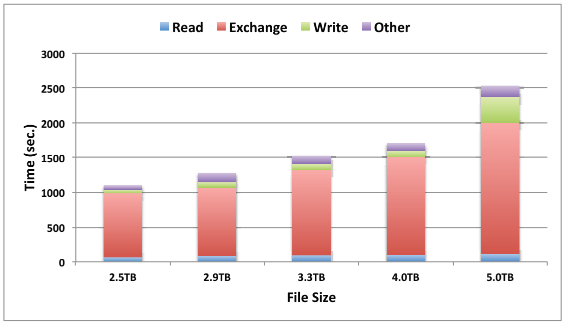

Fig. 9. Execution time of parallel sort-based reorganization on five VPIC datasets

B. Performance of Accessing Array Data

To compare performance of accessing portions of arrays, we used random 3D array data stored in the HDF5 format, resembling images from mass spec experiment explained in Section II-C. We used three datasets to evaluate the performance of SDS. Their sizes are 4GB, 746GB, and 1.8TB. The data required for the mass spec analysis tasks are various images with specified charge mass ratios. This access pattern can be expressed as $[:, :, M : N]$, where the first two colons mean all range of spatial values X and Y separately and M and N means the range of m/z.

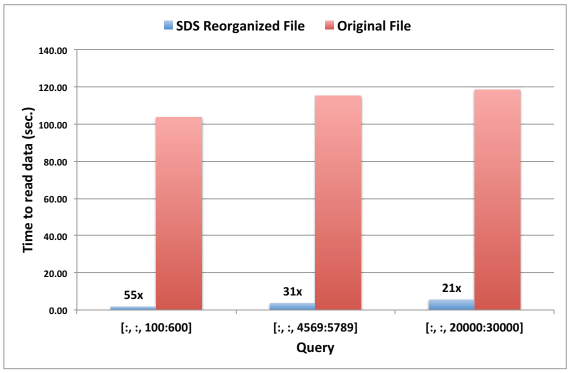

Fig. 10. Read performance of a original mass spec dataset vs. that of a SDS reorganized dataset

Figure 10 compares the time for reading the SDS reorganized file with that for reading the original file with three different image accessing patterns: [:,:,659:789] , [:,:,4569:5789], and [:,:, 20000:30000]. For this comparison, we used the dataset of size 746GB. SDS uses a transpose-based reorganized file of the same size and reshuffles the accessed data in memory before the analysis application uses the data. As the figure shows, the time spent on accessing each image range from the original file is much longer than the time of accessing the same image range from the SDS reorganized file. The main difference in performance is caused by non-contiguous disk accesses in reading data from the original file that takes longer time, whereas SDS reads contiguous chunks of data from SDS reorganized file more efficiently.

1) Scalability of Accessing Reorganized Array Data: We evaluate the performance of reading reorganized mass spec dataset by using different number of cores varying from 50 to 125 CPU cores. We used the 746GB file in this experiment and tested accessing a fixed image with coordinates of [:, :, 20000:30000].

Fig. 11 shows that in all cases, the overhead of SDS VOL implementation is a fraction of a millisecond, which is negligible. Also, with increasing the number client processes, the time used to read the target data from the file decreases gradually. This proves that the overhead of SDS VOL does not have any impact on analysis performance and the benefits with SDS reorganization is immense.

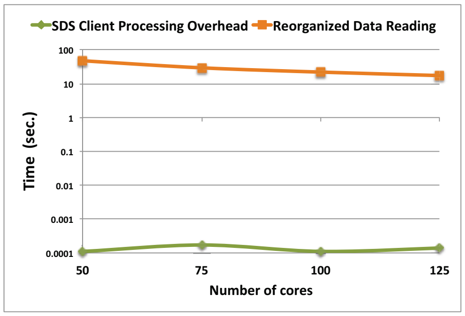

Fig. 11. Scalability of accessing array ranges with SDS. Access pattern of this test is [:, :, 20000:30000] from an mass spec data file with a size of 746 GB

2) Overhead of transpose-based reorganization: We show the overhead of transpose-based reorganization although this overhead does not affect analysis performance. The overhead of transposing a 3D array includes time for reading data, for exchanging data, for transposing subset of data, and for writing results into SDS reorganized file. We used various files to perform transpose, where the sizes of files range from 4 GB to 1.8 TB. Fig. 12 shows that the time to perform transpose-based reorganization of a 1.8T array takes less than 500 seconds. The time needed for exchanging data in transposing operation is again a dominant portion of the total execution time along with that to read data to memory. As SDS Server performs these reorganizations offline, the analysis applications benefit from the SDS framework.

VI. Related Work

There are several ongoing efforts to reorganize file layout to improve I/O performance. We classify them into two categories: system-level file reorganization [25], [12] and file-level reorganization [21], [22]. The system-level file reorganization reallocates multiple files among available I/O servers. The target of the system-level reallocation includes reducing variance of I/O servers [12], improving load balance [25], [26] and enhancing energy efficiency [24]. The file-level reorganization can be further divided into two categories: static and dynamic.

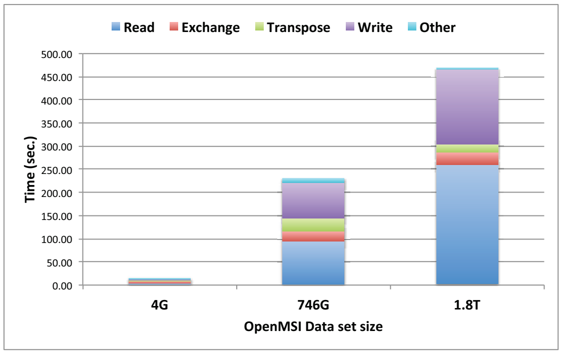

Fig. 12. Time spent for transpose-based reorganization with three mass spec data sets.

The example of static file-level reorganizations are EDO [22] and 2-D layout [21] and multidimensional chunks [18]. PLFS [3] improves checkpointing by writing data to separate files while presenting to user a coherent view of the checkpoint as a single file. Generally, these efforts focus on deriving the optimal layout strategy for specific access patterns, i.e., they work well for applications with fixed access patterns. However, different scientific applications typically have different access characteristics, and these characteristics tend to vary over time [9], [20]. Therefore, it is necessary to reorganize the files dynamically. Moreover, the reorganization and the subsequent data read operations require knowledge of the file layout. With SDS framework, we target performing data reorganization and data reads from replicated datasets that give improved performance transparently. This paper describes HDF5 VOL-based data read operations.

Querying data with conditions is a common data access method in database management systems. However, usage of databases in managing large scientific data is scarce because scientific datasets do not easily fit into the relational data model. Scientific datasets typically fit into array data model. A recent attempt for using array model in database management systems is SciDB [4]. SciDB requires the scientific data to be loaded into the database and then use query languages, called Array Query Language (AQL) and Array Functional Language (AFL), to access data. Instead of developing a new database system, there are also efforts to modify file systems with high-level semantics [5], [15], [19]. While these efforts are likely to be accepted by a few scientific communities, we believe that the array data model needs to be supported as a first class citizen instead of being supported through layers of metadata. We have recently developed FastQuery [8], [6], an array-based querying library based on FastBit [23], that indexes and queries data in parallel and works with array data in scientific data formats such as HDF5 and NetCDF. SDS is the next step towards incorporating decision making on reorganizing, indexing, and querying data transparently.

VII. Conclusions and Future Work

The majority of the scientific data files are organized by the data producers who are more concerned about writing as fast as possible. Their organizations are typically not well-suited for analysis operations. The paper presents the first step of Scientific Data Services (SDS), a framework aiming to combine the merits of database management systems and parallel file systems without pushing the burden involved in either systems on to users. The current implementation of SDS reorganizes files based on its varying access pattern for expediting the read operations. The SDS is based on a client-server architecture currently using the new Virtual Object Layer (VOL) of HDF5. Based on the access patterns of two data-intensive applications: VPIC and Mass Spec Imaging, we designed and implemented parallel sorting-based and parallel transpose-based reorganizations under the SDS framework. Furthermore, we designed and implemented a value based query interface for SDS. Experimental results demonstrate that reading from reorganized files can increase the performance of accessing VPIC and mass spec data up to 23X and 55X, respectively.

Our ongoing efforts include developing an on-line access pattern detection method to enable SDS to make decisions for reorganizing files and for accessing the relevant reorganized files automatically and transparently. We are also working on providing a general interface to support different reorganization methods thus efficient data layout optimizations can be plugged in to SDS by the high performance computing community.

Acknowledgment

The authors gratefully acknowledge the helpful discussions with Drs. Homa Karimabadi, William Daughton, Vadim Roytershteyn, Oliver Rübel, and Benjamin Bowen on data access patterns.

This work is supported in part by the Director, Office of Laboratory Policy and Infrastructure Management of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and used resources of The National Energy Research Scientific Computing Center (NERSC).

References

[1] The ATLAS experiment: Big data and the hunt for the god particle. Technical report, ORIONNetwork, 2012.

[2] G. Aloisioa, S. Fiorea, I. Foster, and D. Williams. Scientific big data analytics challenges at large scale. In *Proceedings of Big Data and Extreme-scale Computing (BDEC)*, 2013.

[3] J. Bent, G. Gibson, G. Grider, et al. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 21:1–21:12, 2009.

[4] P. G. Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 963–968. ACM, 2010.

[5] J. B. Buck, N. Watkins, C. Maltzahn, and S. A. Brandt. Abstract storage: moving file format-specific abstractions intopetabyte-scale storage systems. In *Proceedings of the second international workshop on Data-aware distributed computing*, DADC '09, pages 31–40, 2009.

[6] S. Byna, J. Chou, O. Rübel, Prabhat, H. Karimabadi, et al. Parallel i/o, analysis, and visualization of a trillion particle simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 59:1–59:12, 2012.

[7] M. Chaarawi and Q. Koziol. HDF5 Virtual Object Layer. Technical report, Available: https://confluence.hdfgroup.uiuc.edu/display-/VOL/Virtual+Object+Layer, 2011.

[8] J. Chou, K. Wu, and Prabhat. Fastquery: A parallel indexing system for scientific data. In *CLUSTER*, pages 455–464. IEEE, 2011.

[9] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, 1995.

[10] 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 2011 Workshop on Array Databases*, AD '11, pages 36–47, 2011.

[11] M. Francis. Future telescope array drives development of exabyte processing. Technical report, Available at: http://arstechnica.com/-science/2012/04/future-telescope-array-drives-development-of-exabyte-processing/, 2012.

[12] L.-W. Lee, P. Scheuermann, and R. Vingralek. File assignment in parallel i/o systems with minimal variance of service time. *IEEE Trans. Comput.*, 49(2):127–140, Feb. 2000.

[13] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the versatility of parallel sorting by regular sampling. *Parallel Comput.*, 19(10):1079–1103, Oct. 1993.

[14] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six degrees of scientific data: reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 49–60, 2011.

[15] J. L. Naps, M. F. Mokbel, and D. H.-C. Du. Pantheon: Exascale file system search for scientific computing. In *SSDBM*, volume 6809 of *Lecture Notes in Computer Science*, pages 461–469. Springer, 2011.

[16] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf. ...and eat it too: high read performance in write-optimized hpc i/o middleware file formats. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 21–25, 2009.

[17] W. Reindl, B. P. Bowen, M. A. Balamotis, J. E. Green, and T. R. Northen. Multivariate analysis of a 3d mass spectral image for examining tissue heterogeneity. *Integr. Biol.*, 3:460–467, 2011.

[18] S. Sarawagi and M. Stonebraker. Efficient organization of large multidimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 328–336, 1994.

[19] M. Seltzer and N. Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th conference on Hot topics in operating systems*, HotOS'09, pages 1–1, 2009.

[20] E. Smirni and D. A. Reed. Lessons from characterizating the input/output behavior of parallel scientific applications. *Perform. Eval.*, 33(1):27–44, June 1998.

[21] X.-H. Sun, Y. Chen, and Y. Yin. Data layout optimization for petascale file systems. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, PDSW '09, pages 11–15, 2009.

[22] Y. Tian, S. Klasky, H. Abbasi, J. Lofstead, R. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu. Edo: Improving read performance for scientific applications through elastic data organization. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 93–102, 2011.

[23] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. G. R. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Laurent, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W.-M. Zhang. FastBit: Interactively Searching Massive Data. *Journal of Physics Conference Series, Proceedings of SciDAC 2009*, 180:012053, June 2009. LBNL-2164E.

[24] T. Xie. Sea: A striping-based energy-aware strategy for data placement in raid-structured storage systems. *IEEE Trans. Comput.*, 57(6):748–761, June 2008.

[25] T. Xie and Y. Sun. A file assignment strategy independent of workload characteristic assumptions. *Trans. Storage*, 5(3):10:1–10:24, Nov. 2009.

[26] Y. Zhu, Y. Yu, W. Y. Wang, S. S. Tan, and T. C. Low. A balanced allocation strategy for file assignment in parallel i/o systems. In *Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage*, NAS '10, pages 257–266, 2010.