Improving I/O Throughput of Scientific Applications using Transparent Parallel Compression

Tekin Bicer
Computer Science and Engineering
Ohio State University
bicer@cse.ohio-state.edu

Jian Yin
Pacific Northwest National Laboratory
Richland, WA
jian.yin@pnnl.gov

Gagan Agrawal
Computer Science and Engineering
Ohio State University
agrawal@cse.ohio-state.edu

***Abstract*—Increasing number of cores in parallel computer systems are allowing scientific simulations to be executed with increasing spatial and temporal granularity. However, this also implies that increasing larger-sized datasets need to be output, stored, managed, and then visualized and/or analyzed using a variety of methods. In examining the possibility of using compression to accelerate all of these steps, we focus on two important questions: *"Can compression help save time when data is output from, or input into, a parallel program?"*, and *"How can a scientist's effort in using compression with a parallel program be minimized?"*. We focus on PnetCDF, and show how *transparent* compression can be supported, thus allowing an existing simulation program to start outputting and storing data in a compressed fashion, and similarly, allow a data analysis application to read compressed data. We address challenges in supporting compression when parallel writes are being performed.**

In our experiments, we first analyze the effects of using compression with microbenchmarks, and then, continue our evaluation using a scientific simulation application, and two data analysis applications. While we obtain up to a factor of 2 improvement in performance for microbenchmarks, the execution time of simulation application is improved up to 22%, and the maximum speedup of data analysis applications is 1.83 (with an average speedup of 1.36).

I. Introduction

Increasing number of cores in parallel computer systems are allowing scientific simulations to be executed with increasing spatial and temporal granularity, providing opportunities to observe underlying phenomenon at scales that have never been feasible in the past. However, such 'big compute' opportunities are creating 'big data' problems. To understand scientific phenomenon underlying these simulations, data from simulations need to be output, stored, managed, and then visualized and/or analyzed using a variety of methods.

Unfortunately, parallel systems are becoming increasingly I/O bound. Thus, large datasets that are generated by parallel applications create several problems:

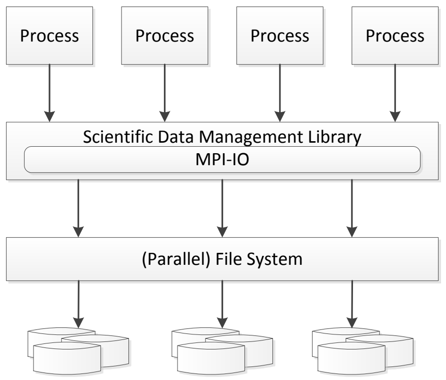
1) The I/O and storage cost of large-scale snapshots or checkpoints of data, which can slowdown the simulation,
2) the difficulty of managing, storing, and moving the massive-scale data after it is generated, and
3) the challenge of reading such large-scale data from an I/O system for analysis, and analyzing data at such scale.

Consider, as just one example, the climate simulations. It is well documented that I/O can limit the scalability of simulations beyond a certain number of cores [21]. It has also been reported that an existing simulation program sustained 2 TB per day of output, and newer models can go to 10-100 PB per day of output. At the same time, parallel systems are expected to become even more I/O bound, and thus, simulation programs will be slowed down even more by the data output steps.

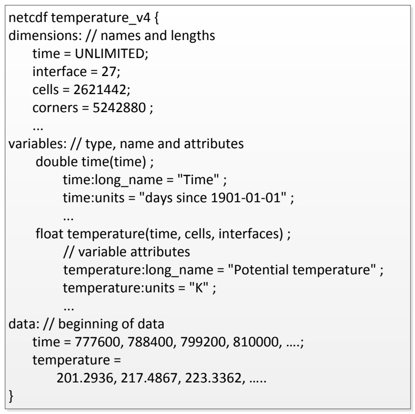
While a range of new techniques, and at different system levels (applications, storage systems, framework designs) will be needed to address the resulting 'big data' problems, one key technology for reducing data volumes is *compression*. Compression has been widely used in computer systems [9], [12], and in recent years, it has been applied on floating point data that is output by scientific simulations [6], [7], [11], [14], [18].

However, we note that most of the work with simulation has been in the context of storing, managing, and moving the datasets, i.e, the second of the three 'big data' issues related to scientific simulations we listed. Clearly, it is also important to decrease the amount of data that needs to be transferred - from clusters where a simulation is run to another cluster or tertiary storage, to a dissemination portal, and/or a system on which the data is analyzed or visualized. However, there is almost no work on enabling transparent parallel compression to speedup a simulation program that outputs large volume of data.

Thus, two important question we need to answer are: *"Can compression help save time when data is output from, or input into, a parallel program?"*, and *" How can a scientist's effort in using compression with a parallel program be minimized?"*. With respect to the latter, we will like to note that parallel simulations are extremely complex programs, and not at all easy to modify. Moreover, these simulations are often executed by scientists that did not implement them, and instead, are simply interested in analyzing their output. Alternatively, a visualization program that works with a large simulation dataset has its own complexity, and modifying such programs is not easy. Thus, compression needs to implemented without any non-trivial modifications to such simulation or analysis

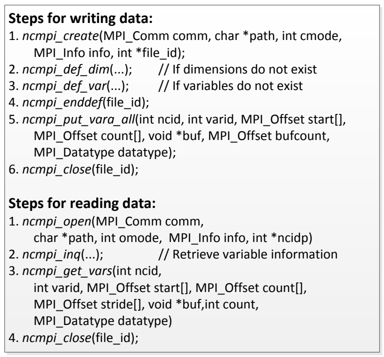


(a) Parallel I/O Libraries Data Flow



```
netcdf temperature_v4 {
dimensions: // names and lengths
      time = UNLIMITED;
      interface = 27;
      cells = 2621442;
      corners = 5242880 ;
      ...
variables: // type, name and attributes
       double time(time) ;
              time:long_name = "Time" ;
              time:units = "days since 1901-01-01" ;
              ...
       float temperature(time, cells, interfaces) ;
              // variable attributes
              temperature:long_name = "Potential temperature" ;
              temperature:units = "K" ;
              ...
data: // beginning of data
      time = 777600, 788400, 799200, 810000, ….;
      temperature =
            201.2936, 217.4867, 223.3362, …..
}
```

(b) Example NetCDF Header



Steps for writing data:
```
1. ncmpi_create(MPI_Comm comm, char *path, int cmode,
      MPI_Info info, int *file_id);
2. ncmpi_def_dim(...);      // If dimensions do not exist
3. ncmpi_def_var(...);      // If variables do not exist
4. ncmpi_enddef(file_id);
5. ncmpi_put_vara_all(int ncid, int varid, MPI_Offset start[],
      MPI_Offset count[], void *buf, MPI_Offset bufcount,
      MPI_Datatype datatype);
6. ncmpi_close(file_id);
```

Steps for reading data:
```
1. ncmpi_open(MPI_Comm comm,
      char *path, int omode,  MPI_Info info, int *ncidp)
2. ncmpi_inq(...);              // Retrieve variable information
3. ncmpi_get_vars(int ncid,
      int varid, MPI_Offset start[], MPI_Offset count[],
      MPI_Offset stride[], void *buf,int count,
      MPI_Datatype datatype)
4. ncmpi_close(file_id);
```

(c) PnetCDF Write and Read Operations

Fig. 1. PnetCDF Basic Operations and Resulting File Header

programs.

This paper addresses these problems, by focusing on one of a small number of popular scientific data management libraries. We note that formats like HDF5 and PnetCDF (and associated APIs and libraries for parallel I/O) are extremely popular in many scientific communities. Thus, it is clearly desirable that compression be built into such libraries, and transparently be used by applications that either output or read from these datasets. Our current solution has been implemented as part of PnetCDF, though our design ideas are applicable to other formats, such as HDF5, also.

Several challenges arise in integrating compression with a library like PnetCDF. First, applying compression with parallel writes is non-trivial as destination offsets that different processes need to write can change. We describe several approaches for integrating compression with parallel writes, which include a method that performs *sparse* storage of output blocks, a method that uses metadata exchange to allow *dense* storage, and an extension to this scheme based on user provided compression ratios. The next problem we address is of maintaining *transparency*, i.e, allowing an existing application that outputs data in NetCDF to work with minimal changes; and similarly, allowing an existing application that reads NetCDF data to work with compressed data. We show how this can be accomplished. Finally, another challenge is choosing the appropriate compression algorithm, since the cost of compression and decompression must be very small, and compression ratios should be large. We have developed a method that exploits properties of simulation datasets.

We have extensively evaluated our methods and implementation. Using microbenchmarks, we show the benefits of compression in improving read and write times for parallel applications, especially as the number of processes increases and key parameters (like stripe size) are carefully chosen. Next, using one scientific application that outputs its state and two applications that analyze simulation output, we show that execution times of applications can be reduced by using compression.

II. Background: A Scientific Array Storage Format, API, and Library

Recall that one of our key goals is to apply parallel compression while requiring at most trivial modifications to simulation codes that output data, or applications that analyze this data. A large number of scientific simulation store their data in either NetCDF (or PnetCDF) or HDF5 formats. The applications that output the data, and the ones analyzing the data extensively use the corresponding libraries and APIs. Our implementation has been carried out in the context of PnetCDF, though the conceptual issues are identical for HDF5. As a background for presenting the details of our design, we first summarize the main features of PnetCDF.

Parallel NetCDF (PnetCDF) (as well as HDF5) provide high performance parallel I/O operations. The parallel I/O mechanism relies on MPI-IO and mimics its API. Therefore, scientific applications can easily implement I/O operations using these libraries and utilize the features of MPI. For example, users can fine-tune their applications' I/O operations, such as collective calls and data shipping, using MPI-specific features and operations. In addition, hints about the file systems and access patterns can be passed to the MPI-IO layer, and further optimizations at the I/O layer can be enabled. In Figure 1(a), we illustrate how parallel I/O operations are typically performed in these libraries.

One of the key challenges we address is to store a compressed PnetCDF file in a fashion that an application designed to read and analyze an uncompressed PnetCDF files can still work with a very trivial modification (a function call). To be able to explain later how this is accomplished, we show how PnetCDF header information is stored in Figure 1(b). PnetCDF preserves the Unidata's NetCDF file-format. Therefore, stored dataset is *portable*, *self-describing* and *space-efficient*. The description of the data is provided at the very beginning of the created file. Typically this header includes the types, dimensions, attributes and structures of variables. Since the metadata information is carried along with the data, scientists can recall the properties of the dataset with minimum effort. Header information starts with the definition of the dimensions where each dimension name and size are specified. Then, the variables are defined with their types and dimensions. The dimensions, along with the data type, determine the size of a variable in file.

We summarize the steps that involve write and read operations of PnetCDF in Fig. 1(c). These operations are similar to MPI-IO operations, but with subtle differences. Specifically, in steps 2 and 3, user needs to define the dimensions and variables in order to inform library about the properties of data. If dimensions are predefined, user can simply use this information to create the variable. Once the variable is defined, its id can be used to write data to the disk. In order to update and synchronize the header information among processes, *ncmpi_enddef* and *ncmpi_close* functions must be called after the definition and write operations, respectively. The read operations follow a similar set of steps. First, netCDF file needs to be opened with proper read mode. Then, the header information in the file is read by the processes in the system. After header information is available in every process, the variable can be read collectively using its id. If the variable id is unknown, it can be inquired using *ncmpi_inq* function call.

III. Parallel and Transparent Compression for PnetCDF

Compression can decrease the size of generated datasets, and by reducing the volume of data to be written on disks, benefit the execution times. Similarly, for an application that analyzes data, if the dataset is stored in a compressed fashion, we can reduce the volume of data to be read from disks, potentially reducing the execution time of the application. Our goal is to achieve this while maintaining *transparency* to users of applications - i.e. require at most a very trivial change to existing applications that write or read large scientific data. While integrating compression with PnetCDF, we require scientific code to simply perform an additional function call. Therefore, (de)compression functions can be registered to the system and applied to the outputs being produced. Similarly, we expect that the NetCDF file which stores compressed array sections can then be processed by another application that reads NetCDF data.

It turns out, however, that such an integration is non-trivial. There are three specific issues that need to be addressed:

- *Applying Compression in Parallel:* While there are efforts in context of scientific data management libraries for enabling compression, they are limited in terms of functionality. For instance, HDF5 enables compression using a single process, however application of compression with parallel write operations is not supported[1]. This is because compression produces variable-size chunks in the system. These chunks can be sequentially written to a file by a single process, however, with multiple processes - where each process can perform independent I/O - these operations can become a challenge.
- *Transparency:* A scientific data management library, such as PnetCDF, involves significant complexity. We want to support compression with only very trivial changes to an application that outputs NetCDF data. At the same time, we want an existing analysis application to be able to handle a compressed NetCDF file, instead of a regular file, with only very trivial changes.
- *Performance:* Another issue while applying compression during I/O operations is the overhead of compression. Typically, generic compression algorithms cannot perform well with scientific datasets. Therefore, domain specific compression algorithms, which are tailored to dataset properties, are desirable for high performance. However, integrating different compression algorithms for different datasets, or variables, is difficult.

[1]Please see http://www.hdfgroup.org/hdf5-quest.html#p5comp

A. Applying Compression with Parallel Writes

We now discuss different schemes for integrating compression with parallel write and read operations.

Compression with Sparse Storage: In this method, after processes generate data, it is passed to the compression layer. Next, the data is divided into splits, and the compression algorithm is applied on each split. Finally, these splits are written without changing their original offset addresses in the target file. In Figure 2, we show how this method works. Unlike a method we will present later, no additional communication between processes is needed with this strategy. The key idea here is that compressed file is stored in a sparse fashion, i.e, it has unused space in between.

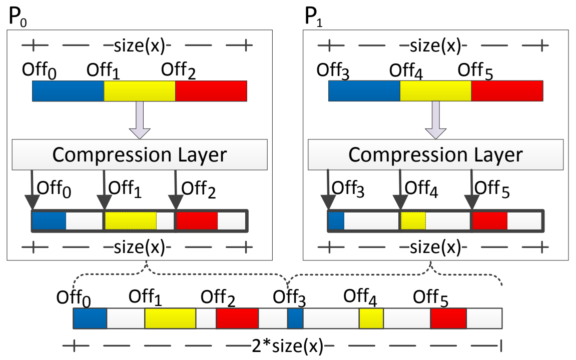


Fig. 2. Write operation using *Sparse Storage*. Each block corresponds to a logical chunk/split. Colors represent the data, whereas white spaces correspond to unused data blocks.

Since application only needs to write and read the *compressed splits*, the I/O performance can still be improved. However, there are several problems with this approach. First, the processes can spend time on seek operations, which can negate many of the benefits of writing or reading less data. Further, the final size of the file is almost the same with the original dataset size because of the empty blocks in the compressed data. Thus, there is no advantage in terms of reducing storage space. However applications can still potentially benefit from the compressed splits, if bandwidth is the main bottleneck in the system.

Compression with Dense Storage: In this method, the compressed splits are stored as contiguous data blocks, thus smaller data file is produced. Figure 3 presents the execution flow of this method. Similar to sparse compression method, data is passed to the compression layer, and compressed splits are generated. Then, processes exchange metadata information of their compressed splits through an additional round of communication, and new destination offset addresses are calculated. For instance, in Figure 3, P_0 broadcasts $Off_0^{'}$ and $Off_2^{'}$ with their corresponding size information. Similarly, P_1 exchanges $Off_3^{'}$ and $Off_5^{'}$ with size information. Next, P_1 recalculates the initial offset address according to $Off_3^{'} = Off_2^{'} + split\_size_{Off_2^{'}}$, and all the other local offset addresses are updated according to the new $Off_3^{'}$ and its size information. Finally, the compressed splits are written to the disk.

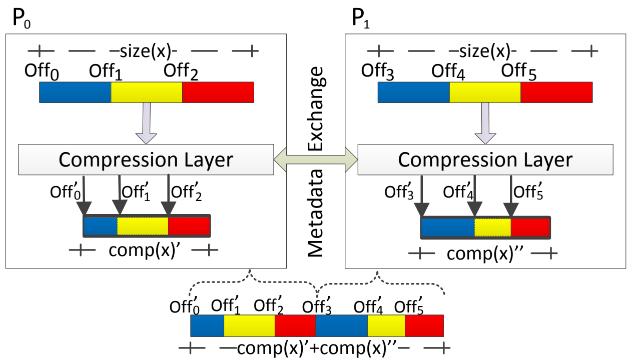


Fig. 3. Write operation using *Dense Storage*. New offset addresses of the compressed data blocks are calculated after compression and metadata exchange.

Metadata exchange phase in dense compression results in extra communication time compared to sparse compression, however the resulting file is smaller. Therefore, file transfers can be done more efficiently. Also the number of required MPI-IO operations is smaller as compared to compression with sparse storage. Specifically, while the compression layer needs to perform I/O operation for each of the compressed split in sparse compression, single collective call is sufficient for the dense method. The continuous write operations can also be optimized by collective MPI-IO operations, which yield higher I/O throughput.

On the other hand, not all applications can perform continuous write operations. To illustrate this, consider the following example. Suppose we have two processes, $P1$ and $P2$, which produce four data segments, S_{11}, S_{12}, S_{21}, and S_{22}, that need to be stored contiguously, in the above order. Assume, however, $P1$ and $P2$ need to perform certain computations, and generate S_{11} and S_{21}, respectively. After writing these segments, they move onto computations and produce S_{12} and S_{22}. In this case, $P2$ cannot know the destination offset address of S_{21}, unless $P1$ generates S_{12} and applies compression on it. While restructuring of the computation and/or storage can address this situation, recall that our goal has been a *transparent* compression.

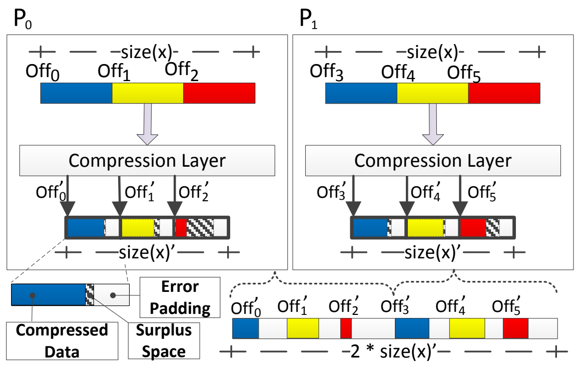


Fig. 4. Write operation using *Hybrid Method*. After compression, the colored blocks represent the compressed data; lined blocks are the surplus of expected compression space; and white spaces represent the space for handling overflowed data.

The compression with dense storage can be made more broadly applicable through a *hybrid method*. The idea here is that if the application developer can provide the expected *compression* and *error ratios* to the system, then compressed split sizes and offset addresses can be calculated in advance. For instance, assume that compression layer of each process works on 100MB of data split, and expected compression and error ratios are 1.35 and 0.05, respectively. Then the computed split size becomes ∼76.9MB, in which ∼2.9MB of space is reserved to address possible inaccuracy in user estimate of the compression ratio. Since all original splits share the same size information, destination offset addresses can be derived. Specifically, split sizes can be calculated with $compressed\_split\_size = original\_split\_size \times (1/(comp\_ratio - err\_ratio))$. Similarly, the destination offset addresses of the compressed splits can be computed using $Off_i^{'} = Off_i \times (1/(comp\_ratio - err\_ratio))$, where $i > 0$. In Figure 4, we present the data execution flow for improved hybrid method.

Here, prior knowledge about the compression ratio eliminates the metadata exchange requirement. If the expected compression ratio does not meet the real compression ratio, the padding for the errors can handle the overflowed data. In figure, $surplus\ space + compressed\ data$ corresponds to the expected split size after compression, and $error\ padding$ refers to the additional space for managing the overflowed data.

This method can eliminate the metadata exchange overhead, and also make the approach more broadly applicable as compared to the original dense method. However, it requires a reasonably correct estimation of compression ratio. This can be difficult to accomplish if the application developer has limited information about the data. In this case, sampling can be used to estimate compression ratios.

TABLE I
API for Supporting Compression in PnetCDF

Data Type Declarations

```
 typedef struct {
   off_t orig_offset; // Original offset address of data block
   size_t orig_size; // Original data block size
   off_t comp_offset; // Dest. offset address of compressed data block
   size_t comp_size; // Compressed data block size
} comp_metadata_info_t;
// Metadata information of the compressed data blocks
```

```
 typedef struct {
   void* user_args;
   size_t user_args_size;
   chunk_args_t* ch_arg;
} comp_args_t;
// User defined arguments for
compression functions
```

Required User-Defined Functions for Compression

`size_t (*comp_f)(void* input, size_t in_size, void* output, size_t out_size, comp_args_t* comp_args);`
Description: User implemented compression function is applied to input data chunk, and computed data is written to `output`

`size_t (*decomp_f)(void* input, size_t in_size, void* output, size_t out_size, comp_args_t* comp_args);`
Description: User implemented decompression function is applied to `input` compressed data chunk, and computed data is written to `output`

Introduced PnetCDF Functions

`int ncmpi_comp_reg(int var_id, size_t (*comp_f)(...), size_t (*decomp_f)(...), comp_args_t* u_args,..);`
Description: Registers user defined (de)compression functions and variable id to PnetCDF. Also sets the system specific parameters, such as split size.

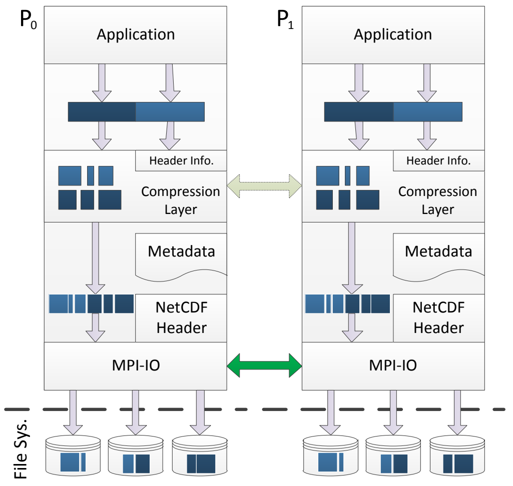


Fig. 5. Write Operation using Compression System

B. System Design for Transparent Compression

We now describe how compression, including the possibility of dense or hybrid storage, is integrated while keeping the process transparent to the application. In Figure 5, we show the modified version of PnetCDF which accomplishes this. First, developer informs compression system about target variable and the parameters that are related with the compression operations. Through calling `ncmpi_comp_reg` function, variable id, pointers to (de)compression algorithms and user specific data structures are registered to the compression system. Also system parameters such as buffer and split (chunk) sizes are defined in this stage. Once these information are provided, compression system can retrieve the variable information using netCDF header file and apply user defined functions.

The important data structures and function definitions are provided in Table I. After the registration of compression parameters and relating the variable, data can be written to the file using compression system. Initially, application prepares data and passes it to the PnetCDF library, exactly as would happen in the original application. The compression layer, then, divides the original data into *split_size* chunks, and applies user provided compression algorithm. The size of the splits determine the granularity of data accesses. Specifically, the smaller split sizes enable fine grain data access to the elements in compressed chunks. However, finer granularity might also decrease the compression ratio. The larger split sizes, on the other hand, result in coarse grain data access, and might provide higher compression ratios.

After splits are compressed, the metadata information of each split is generated, which consists of *orig_offset, orig_size* and *comp_size* information. During the compression operation, generated splits are written into an internal buffer. Therefore, extra copy operation is avoided. Once the compression is performed on data, compression layer exchanges the metadata information of splits with a collective call, and calculates the destination offset address of each split. Then, the offset information of the compressed chunks, *comp_offset*, are set. This collective call is initiated from within the modified PnetCDF library, i.e., there is no change in the application code. Lastly, the final layout of the data is updated on the netCDF header, and a consistent global view of the data layout is achieved.

Figure 5 illustrates the process of writing data splits to a parallel file system using MPI-IO collective calls. Optimization of write operations can significantly affect the performance. Specifically, if the alignment of the compressed data chunks on file are not done properly, then write operations can create contention on the disk and result in overhead.

Reading Compressed Scientific Data: The read operation of the compressed data is the same, irrespective of whether sparse or dense storage is used. Before application starts accessing the stored data, the compression parameters are registered in the system. Also, the location of the metadata file is provided to the compression layer and read into the memory. Once the application layer requests data from the PnetCDF, the compressed splits that correspond to the original offset addresses are detected in the metadata file. Then, these compressed chunks are retrieved from the file, and user defined decompression algorithms are applied within the modified PnetCDF library. The requested elements are directly decompressed into the user provided buffer. The compression layer applies optimizations while requesting the compressed splits. For example, if the requested data is continuous, then compression layer combines the compressed split requests into a single collective call. Therefore, the number of I/O operation on file system is decreased. Furthermore, compressing the dataset using many splits increases the depth of parallel decompression.

C. Compression Method and Performance

Compression has been used in different areas of computing systems, including file systems, big data processing middlewares and database management systems. However, these systems typically focus on generic data compression. Unfortunately, most of the scientific datasets consist of high precision floating point numbers, which are known to be difficult to compress by generic compression algorithms. Although there have been works on scientific data compression [7], [18], we focus on domain specific approaches.

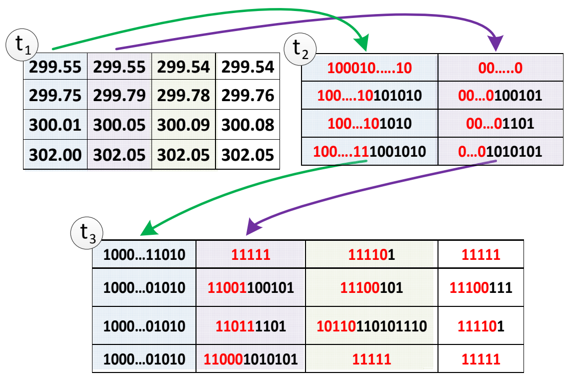


Fig. 6. Example: Compressing Scientific Datasets

Our compression approach is based on differential compression [6]. Basically, our method exploits the relationship between adjacent cells, and stores the difference. Consider the example in Figure 6. Assume that we have a multidimensional array which consists of temperature values of neighboring locations. Since the temperature value of a coordinate is expected to be similar to its neighboring locations, the differences are not significant. This property can be exploited for efficient compression. Specifically, we can apply *xor* among adjacent cells and store the differences along with the number of zeros. We can perform this operation on t_1, using $t_1[i][j] \oplus t_1[i][j-1]$, and t_2 can be generated. Note that, the generated values consist of two parts, a sequence of leading zeros and a difference part. Next, the number of leading zeros are counted and represented in bits. If, for example, an array consists of single precision floating point numbers (32 bits), then the number of leading zeros can be represented with 5 bits. Therefore, we can store the number of leading zeros to the first five bits, and append the remaining difference part. The final compressed array is shown in t_3. Notice that, the array t_2 is virtual, i.e. t_3 can directly be derived from t_1.

IV. Experimental Results

In this section, we evaluate the performance of our parallel compression system using a set of microbenchmarks, one parallel simulation application that outputs a NetCDF file, and two analysis applications that read (and analyze) NetCDF files. Specifically, we conducted the following experiments. First, we focused on characterizing the benefits of compression for a parallel program that writes or reads large-scale datasets. In the process, we also studied the impact of different parameters, such as stripe sizes in parallel file system, on performance. Yet another goal was to compare the performance against compression implemented in HDF5 library (where compression can only be applied sequentially). Next, our goal was to understand how parallel compression can benefit a scientific simulation program that snapshots its state periodically. Finally, we focus on applications that read and analyze scientific datasets, and examine the benefits of using compression.

A. Experimental Setup

We performed our experiments in our local cluster at the Ohio State University. The cluster consists of 160 nodes where each node has 12 GB memory and 8 cores (Intel Xeon E5630, 2.53GHz). The nodes are connected with Mellanox ConnectX-2 InfiniBand QDR cards (40Gbps). The compute nodes are attached to a parallel file system, Lustre, which provides high I/O throughput. Our file system consists of 4 storage nodes and each node has 2 physical disks. The total number of object storage target (OSTs) is 8, and there is one metadata server in the system.

As a scientific simulation application, we used miniMD, which is one of the compute and communication intensive mini-applications in Mantevo project[2]. MiniMD is a lightweight version of LAMMPS molecular dynamics application. It simulates the interaction of atoms, and computes parameters such as location and temperature. In our experiments, we set the problem size of miniMD to $128 \times 128 \times 128$, and specified the number of timesteps as 500. For each timestep, the coordinate information of the atoms is stored, which is $\sim$97 MB in size, in a NetCDF file. The idea is that such a file will allow a scientist to visualize movement of particles later. The total size of the generated data after 500 iterations is $\sim$49 GB.

We evaluated the read performance of our system using different data analysis applications. The first application, `AT`, calculates the average temperature of atmospheric layers in climate dataset. The second application, `MATT`, computes the minimum, maximum and average temperatures in climate dataset with respect to different time frames. The inter-communication overhead of this application is higher than the AT, and it requires larger memory to operate. Both data analysis applications perform their computation on Global Cloud-Resolving Model (GCRM) climate dataset. The climate dataset consists of temperature values with respect to time and location. It divides atmosphere in cells and layers. The temperature values are stored as single precision floating point numbers where each of them represents a Kelvin degree. We have generated two climate datasets, which are 34 GB and 136 GB, respectively. They are stored in NetCDF 64-bit format for large data storage support.

In all our experiments, each version is run at least five times, and the average of the results are presented. All applications and the PnetCDF library are compiled with highest optimization parameters.

B. Microbenchmarks

In this section, we analyze the effects of file system parameters and evaluate the benefits of parallel compression using microbenchmarks.

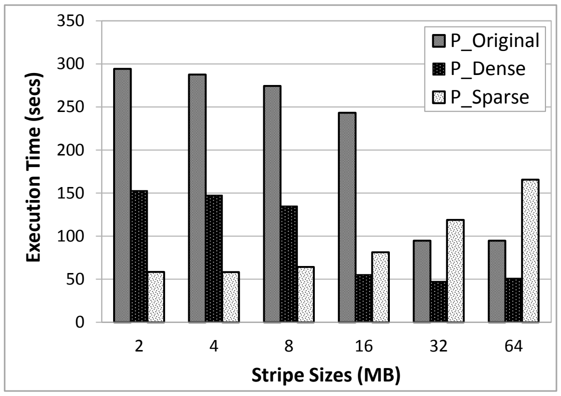


Fig. 7. Comparing Write Times with Varying Stripe Sizes; Versions: Original (Uncompressed), Compression with Sparse Storage, and Compression with Dense Storage; Experimental Setup: 64 processes and 34 GB total output; Chunk size is 32 MB, and stripe count is 8.

In Figure 7, we show the performance of our compression system while varying stripe sizes. Stripe sizes determine the distribution of data chunks to OSTs, and therefore, it directly affects the parallelization of I/O operations and contention on the physical disks. The experiment involves varying stripe sizes with all three versions, which are P_Original, P_Dense, and P_Sparse. P_Original shows the write times of uncompressed dataset to a PnetCDF file. Similarly P_Dense and P_Sparse versions represent the write times of dense and sparse compression schemes, respectively. The microbenchmark uses 64 processes that output a total of 34 GB dataset (uncompressed). Processes iteratively write data to the disks, in which each process writes 270 MB data for each iteration. The *split_size* is set to 32 MB, hence the data is divided into 9 chunks in each iteration. The average compression ratio of our algorithm is 1.9:1. Therefore, the versions with dense and sparse storage write a total of 17.7 GB data, where each process stores 158.8 MB data for each iteration. Also note that the average size of each compressed chunk is 16.8 MB, whereas the original is 32 MB.

Overall, the experiment shows that with their respective best choices for stripe sizes, dense and sparse storage can reduce execution times by nearly 50% and 35%, respectively. However, the behavior of different versions under different stripe sizes is intriguing. Our first observation is the sharp decrease in execution time after 16 and 32 MB stripe sizes for P_Original and P_Dense versions, respectively. The main reason of this is the better utilization of the disks and minimized contention. Specifically, the number of OSTs and disks determine the available parallelism on the file system; and the stripe size specifies corresponding offset addresses of the OSTs. For example, if the stripe size is 32 MB and there are 8 OSTs in the system, then system can properly align 256 MB of continuous data to these OSTs and parallelize the write operation. Since each process writes 270 MB data for each iteration in P_Original, 32 MB stripe size gives best performance. Similarly, P_Dense shows the best performances starting from 16 MB, as it is writing less data in all.

Now, let us examine the performance of compression with P_Sparse. With smaller stripe sizes, P_Sparse version improves the performance, but it should be noted that we are not obtaining the best uncompressed I/O performance. Processes in sparse compression perform write operations after compressing each split; and with small stripe sizes, the distribution of the splits are more balanced. At larger stripe sizes, the number of write operations to each OST increases, and furthermore, non-contiguous write operations reduce overall performance.

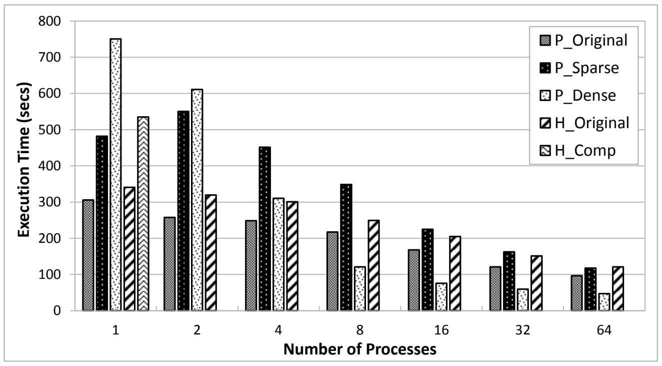


Fig. 8. Comparing Write Times with Varying Number of Processes; Versions: Original (Uncompressed), Compressed Dense and Sparse Storage using PnetCDF, and HDF5 Original(Uncompressed) and Sequential Compression; Stripe size is 32 MB.

In Figure 8, we present the write times of the same dataset with varying number of processes. The stripe size is set to 32 MB for all versions. In this set of experiments, we also perform a comparison against HDF5 compression. Although, HDF5 supports sequential compression with single process, parallel compression is not available. Also, the compression method we have implemented using our framework was not available with HDF5. Therefore, we implemented our compression algorithm in HDF5 library in order to facilitate a comparable experiment. In all, five different versions have been used in these experiments. P_Original and H_Original versions involve uncompressed writes with PnetCDF and HDF5 libraries, respectively. `P_Sparse` and `P_Dense` versions represent executions in which the data is compressed using sparse and dense storage methods, respectively, both with PnetCDF. Sequential compression with HDF5 is presented as `H_Comp`, where we can only present result for a single process.

Our first observation from Figure 8 is the overhead of compression with small number of processes. While the number of processes increases this overhead is divided among processes and application starts benefiting from the compression. Note that parallel applications with write operations tend to become more I/O bound as the number of processes increases. Since the contention on file system and network increases with increasing number of processes, I/O operations start introducing overhead in the system. Compression, on the other hand, is a compute-intensive operation which is embarrassingly parallel. Thus, compression times decrease linearly with the number of processes, whereas the I/O times do not, and with increasing number of processes, the time spent on compression is easily outweighed by I/O times. Compression in PnetCDF followed by dense storage is again almost always better than sparse storage. The improvement with `P_Dense` is faster than `P_Original` and `H_Orig` versions by an average of 1.79 and 2.71, respectively, with 4 or more processes.

`H_Comp` version follows a similar trend with `P_Dense` where the small number of processes introduces overhead in the system. While sequential compression can reduce storage and data transfer costs, parallel compression is a desirable feature for improving the performance of I/O operations.

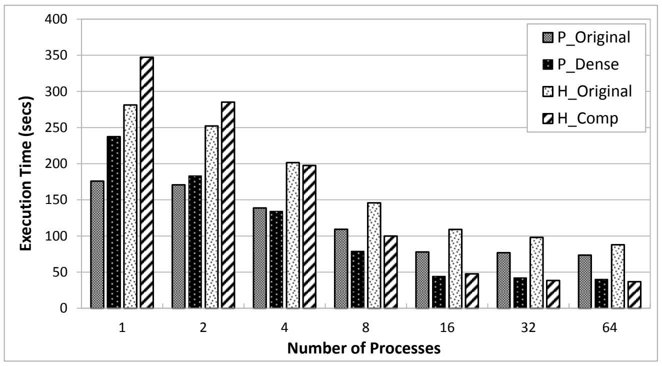


Fig. 9. Comparing Read Times with Varying Number of Processes; Versions: Original (Uncompressed), Compressed Dense Storage using PnetCDF, and HDF5 Original(Uncompressed) and Compressed; Stripe size is 32 MB.

Figure 9 shows the read performance of modified PnetCDF and HDF5 using compressed and original data. We used the same setup with previous experiments and varied the number of processes. Since we have already demonstrated that compression with dense storage is better than sparse storage, we restrict ourselves to only the dense version, `P_Dense`. In this set of experiment, we also present the performance results of compressed read operations in HDF5 with varying number of processes.

Similar to write operations, compression introduces some overhead with small number of processes. However, after 4 processes, application starts benefiting from compression, i.e, the cost of decompressing the file is smaller than the reduction in read times achieved. Specifically, the speedups of P_Dense over $P\_Original$ are between 1.38 and 1.78 after 4 processes. Although the ability to compress files during parallel writes is not available with HDF5, parallel reads can still be performed. The parallel read performance of HDF5 using compressed data is shown with `H_Comp`, where this version improves the read times by 31.4-60.7% after 4 processes considering the `H_Original`.

C. Impact on a Scientific Simulation Application

Most of the scientific simulations are long running applications, which generate large volumes of data. There might be several reasons to store this data. The first, and increasingly popular, reason is the need for visualizing or post-analysis of the progress of simulations in order to understand the underlying phenomenon. Another reason is tolerating failures. Since these scientific simulations are prone to failures due to their long running nature, periodically checkpointing the state information of application can significantly reduce the re-computation time.

We used *miniMD*, which iteratively calculates the location of atoms according to force information, as a representative scientific application. It outputs the location of all particles after each time-step. We integrated modified version of PnetCDF into miniMD simulation application, and observed the execution times with and without compression. The original size of written data is ∼47 GB, whereas the compressed size is ∼35 GB.

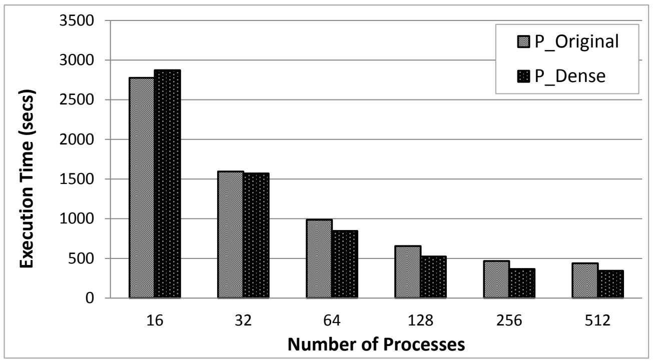


Fig. 10. MiniMD Application Execution Time - With and Without Compression and Different Number of Processes

The results are presented in Figure 10. With 16 processes, our compression system introduces 3% overhead, which is because of the compression cost. However, with increasing number of processes, compression benefits the application execution time. Specifically, with 32 or more processes, the total execution time of the application decreases between 2% and 22%, where the best gain, 22%, is seen with 512 processes. The computation and communication times in miniMD application dominate the write time. In order to present the advantage of using compression, we show the elapsed time of write operations in Figure 11. Speedups for write operations using compression range from 1.9 to 2.35 with 32 or more processes. Overall, increasing number of cores in parallel machines and systems can create contention on file system and network, however compression can benefit such simulations and minimize the overhead. This observation is more obvious when applications are I/O bounded.

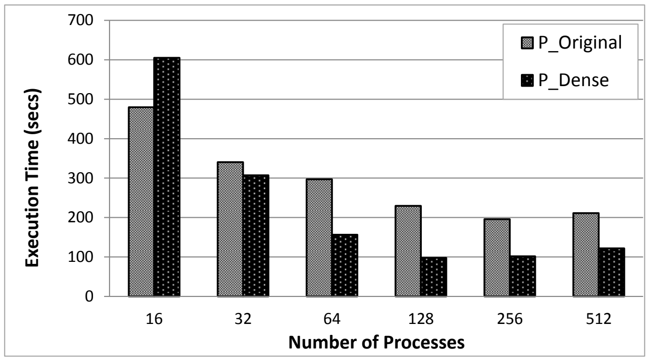


Fig. 11. MiniMD Application Write Time - With and Without Compression and Different Number of Processes

D. Impact on Scientific Data Analysis Applications

We had previously shown that compression can improve performance for a microbenchmark that performs parallel read operations. In this section, we investigate the performance of data analysis applications with and without compression. These applications first read the data using the modified PnetCDF or HDF5 library, and then perform the analysis. The original size of the dataset is 136 GB. This dataset is compressed and stored using 32 MB chunks and stripe sizes, and the resulting dataset size is close to 71 GB.

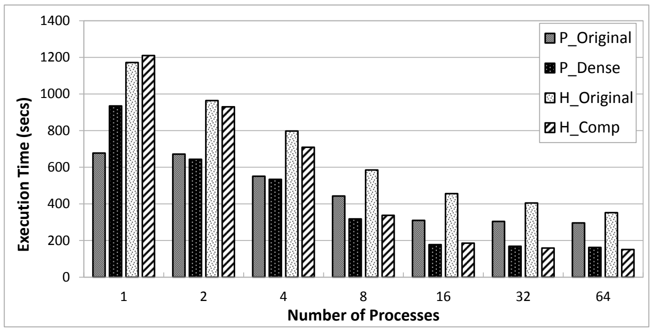


Fig. 12. AT Application Execution Time with Different Number of Processes

We present the *AT* application performance using compression system in Figure 12. Similar to previous set of experiments, both `H_Comp` and `P_Dense` versions significantly benefit the application execution while the number of processes increases. The compression system introduces some overhead with small number of processes in `P_Dense` version. However after 8 processes, compression system improves the performance of I/O operations. The speedups for `P_Dense` against `P_Original` are between 1.74 and 1.83 after 8 processes, where the highest speedup is observed with 64 processes. `H_Comp` version also shows a similar trend with `P_Dense`, i.e. increasing number of processes shows better performance than the small number of processes.

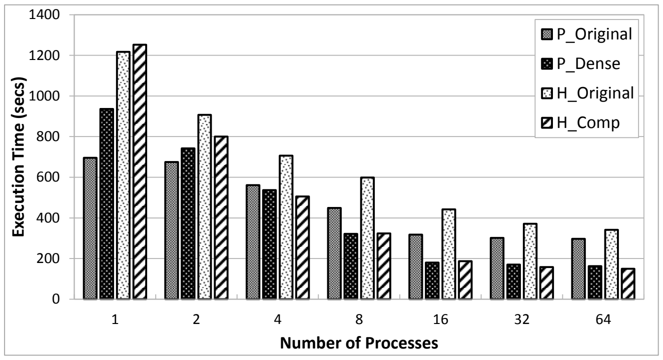


Fig. 13. MATT Application Execution Time with Different Number of Processes

We repeat the same set of experiments with *MATT* application and present results in Figure 13. Again, read operations on compressed dataset present better execution times with higher number of processes. Since the *MATT* application involves more communication and computation, its execution times are higher than the *AT* application. For this set of experiment, the reduction in execution times are between 29% and 45% for P_Dense version, and range from 46% to 58% for H_Comp version after 4 processes.

V. Related Work

Compression has been used widely in order to ease the management of large datasets [12], [9]. Application of compression algorithms in scientific datasets and data management libraries has received significant attention in recent years. There are several projects which focus on improving the I/O throughput of scientific applications, and compression has been attempted in the context of many of these. PnetCDF [13], [4] and Parallel HDF5 [3] are both widely used scientific data management libraries. Although there have been research on improving data analysis, I/O throughput and data transfer performance of scientific data management libraries [22], [20], utilization of compression in these libraries is limited. Currently, to the best of our knowledge, PnetCDF does not provide any compression feature. HDF5, on the other hand, lets users apply generic compression algorithms, such as *ZLIB* and *SZIP*, but only with single process, i.e. there is no support for parallel compression. The main reason for this limitation is that variable-length chunks that are produced after compression, which has been addressed in our approach.

ADIOS [15] is another scientific data management library which lets developers decide on the I/O interface using an XML configuration file. ISOBAR compression method [18] has been integrated with ADIOS [19], where efficiency being maintained by interleaving compression with I/O. Although, there are similarities between our compression method and ISOBAR, our methods for parallelizing compression in scientific data management libraries is not limited with a particular compression approach, and moreover, we allow transparent compression for the large class of applications where PnetCDF is used.

Data-intensive middleware/storage systems also extensively use compression to improve I/O throughput. Some of these efforts include compression in Hadoop[1], BlobSeer[16], HBase[5], and BigTable[8]. These systems do not provide scientific data formats. Our work focuses on supporting compression inside a widely used scientific data management library which can provide such formats.

Supporting efficient fault tolerance for scientific applications is another area where compression has been applied. Islam *et. al.* developed MCREngine [10], which is a checkpointing system library that uses compression to improve I/O throughput. Their system exploits the information provided by the scientific data management libraries in order to detect the similar variables and the best compression algorithm. Our main contribution is in supporting transparent parallel compression which requires minimum effort from the developer, and its application is not limited to checkpointing for fault-tolerance.

Generic compression algorithms typically cannot provide good compression ratios with scientific datasets which are highly entropic. Therefore, efficient domain-specific compression algorithms are desired. In our previous work [6], we proposed a compression methodology for scientific datasets, which has also been implemented in our current work. There are also a number of other efforts that have focused on efficient compression of this type of data [17], [14], [7], [18], [11].

VI. Conclusion

In this paper, we have investigated the use of compression for programs that output or analyze large-scale scientific data. First, we introduced two storage methods, *sparse* and *dense*, which enable scientific applications to perform parallel compression during write operations. Then, we integrated our approaches to a widely used scientific data management library, PnetCDF, and provide transparent compression while exploiting and maintaining the properties of NetCDF. The modified PnetCDF library exposes an API in which developers can easily plug different compression algorithms and relate them with data variables.

We have evaluated our methods using microbenchmarks, a scientific simulation application, and two scientific data analysis applications. Our experiments show that compression can significantly improve the I/O throughput of applications. For simulation application, which performs periodic write operations, our dense storage method improved the execution time up to 22%, while decreasing the storage requirement by 25.5%. The average speedup of data analysis applications is 1.36, where the maximum speedup is 1.83.

Acknowledgments: This work was partially supported by NSF award ACI-1339757.

References

[1] The Apache Hadoop Project. http://hadoop.apache.org/, 2012. [Online; accessed September-2012].

[2] Mantevo Project. http://mantevo.org, 2013. [Online; accessed October-2013].

[3] Parallel HDF5. http://www.hdfgroup.org/HDF5/PHDF5, 2013. [Online; accessed October-2013].

[4] Parallel netCDF. http://trac.mcs.anl.gov/projects/parallel-netcdf, 2013. [Online; accessed October-2013].

[5] The Apache HBase. http://hbase.apache.org/, 2013. [Online; accessed October-2013].

[6] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt. Integrating online compression to accelerate large-scale data analytics applications. In *IPDPS*, pages 1205–1216. IEEE Computer Society, 2013.

[7] M. Burtscher and P. Ratanaworabhan. Fpc: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Computers*, 58(1):18–31, 2009.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[9] F. Douglis. On the role of compression in distributed systems. *SIGOPS Oper. Syst. Rev.*, 27(2):88–93, Apr. 1993.

[10] T. Islam, K. Mohror, S. Bagchi, A. Moody, B. de Supinski, and R. Eigenmann. Mcrengine: A scalable checkpointing system using data-aware aggregation and compression. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, 2012.

[11] J. Iverson, C. Kamath, and G. Karypis. Fast and effective lossy compression algorithms for scientific datasets. In C. Kaklamanis, T. S. Papatheodorou, and P. G. Spirakis, editors, *Euro-Par*, volume 7484 of *Lecture Notes in Computer Science*, pages 843–856. Springer, 2012.

[12] D. A. Lelewer and D. S. Hirschberg. Data compression. *ACM Comput. Surv.*, 19(3):261–296, 1987.

[13] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[14] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1245–1250, 2006.

[15] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. 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*, pages 15–24. ACM, 2008.

[16] B. Nicolae. High throughput data-compression for cloud storage. In A. Hameurlain, F. Morvan, and A. Tjoa, editors, *Data Management in Grid and Peer-to-Peer Systems*, volume 6265 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-15108-81.

[17] A. Padyana, C. Sudheer, P. K. Baruah, and A. Srinivasan. High throughput compression of floating point numbers on graphical processing units. In *Proceedings of the 2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC)*, pages 313–318. IEEE, 2012.

[18] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Isobar preconditioner for effective and high-throughput lossless data compression. *Data Engineering, International Conference on*, 0:138–149, 2012.

[19] E. R. Schendel, S. V. Pendse, J. Jenkins, D. A. Boyuka, II, Z. Gong, S. Lakshminarasimhan, Q. Liu, H. Kolla, J. Chen, S. Klasky, R. Ross, and N. F. Samatova. Isobar hybrid compression-i/o interleaving for large-scale parallel i/o optimization. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 61–72, New York, NY, USA, 2012. ACM.

[20] Y. Su, Y. Wang, G. Agrawal, and R. Kettimuthu. Sdquery dsi: integrating data management support with a wide area data transfer protocol. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13, Denver, CO, USA - November 17 - 21, 2013*, page 47. ACM, 2013.

[21] P. L. Vidale, H. Weller, and B. N. Lawrence. Weather and Climate Modelling: ready for exascale? http://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/whitepapers/BDEC%20workshop-PL_Vidale.pdf, 2013. [Online; accessed October-2013].

[22] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over hdf5. In *CCGRID*, pages 335–342. IEEE Computer Society, 2013.