Integrating Online Compression to Accelerate Large-Scale Data Analytics Applications

Tekin Bicer*, Jian Yin†, David Chiu‡, Gagan Agrawal* and Karen Schuchardt†
*Computer Science and Engineering
Ohio State University
E-mail: {bicer, agrawal}@cse.ohio-state.edu
†Pacific Northwest National Laboratories
E-mail: {jian.yin, karen.schuchardt}@pnnl.gov
‡Washington State University
E-mail: david.chiu@wsu.edu

***Abstract*—Compute cycles in high performance systems are increasing at a much faster pace than both storage and wide-area bandwidths. To continue improving the performance of large-scale data analytics applications, compression has therefore become promising approach. In this context, this paper makes the following contributions. First, we develop a new compression methodology, which exploits the similarities between spatial and/or temporal neighbors in a popular climate simulation dataset and enables high compression ratios and low decompression costs. Second, we develop a framework that can be used to incorporate a variety of compression and decompression algorithms. This framework also supports a simple API to allow integration with an existing application or data processing middleware. Once a compression algorithm is implemented, this framework automatically mechanizes multi-threaded retrieval, multi-threaded data decompression, and the use of informed prefetching and caching. By integrating this framework with a data-intensive middleware, we have applied our compression methodology and framework to three applications over two datasets, including the Global Cloud-Resolving Model (GCRM) climate dataset. We obtained an average compression ratio of 51.68%, and up to 53.27% improvement in execution time of data analysis applications by amortizing I/O time by moving compressed data.**

I. Introduction

Science has become increasingly data-driven. Data collected from instruments and simulations is extremely valuable for a variety of scientific endeavors. Both wide-area data dissemination and analysis have become important areas of research over the last few years. These efforts, however, are complicated by the sustained and rapid growth of scientific data sizes. Indeed, increased computational power afforded by today's high-performance machines allows for simulations with ever higher resolutions over both temporal and spatial scales. As a specific example, the Global Cloud-Resolving Model (GCRM) currently produces 1 petabyte of data for a 4 km grid-cell size over a 10 day simulation. Future plans include simulations with a grid-cell size of 1 km, which will increase the data generation by factor of 64. Even in the short term (*i.e.*, by 2015) it will be possible to perform 2 km resolution simulations, where a single time step of one three-dimensional variable will require 256 GB of storage.

At the same time, scientific experiments and instruments are also collecting data with increasing granularity. The Advanced LIGO (Laser Interferometer Gravitational-wave Observatories) Project, funded with a $200 million investment from National Science Foundation, is increasing its sensitivity by a factor of ten, resulting in three orders of magnitude increase in the number of candidates for gravitational wave signals[1].

Unfortunately, wide-area network bandwidth and disk speeds are growing at a much slower rate. This stifles the application scientists' need to download, manage, and process massive datasets. To reduce data storage, retrieval costs, and transfer overheads, compression techniques have proven to be a popular approach among users [17], [2], [9], [23], [6], [35], [30], [22]. Compression has also been recently applied for reading large scientific files in parallel file systems [39]. However, effectively supporting compression for scientific simulation data and integrating compression with data-intensive applications remains a challenge. Specifically, we feel that much additional work is needed along the following directions:

- How can the properties of large-scale scientific datasets, especially the simulation datasets, be exploited to develop more effective compression algorithms?
- How can we develop software that allows easy plug-and-play of compression and decompression algorithms, while allowing the benefits of prefetching, multi-threading, and caching?
- How can software be integrated with a data analysis middleware, to help achieve performance benefits for local and remote data analysis?

To address the above challenges, this paper makes the following contributions. First, we develop a new compression methodology, which exploits the fact that spatial and/or temporal neighbors in simulation data have very similar values. Thus, one can simply store certain base values and the deltas pertaining to these base values, which can be represented more efficiently. In lieu of storing all floating point values, we achieve high compression ratios. Additionally, our compression and decompression scheme exploits hardware supported bitwise operations to keep the costs of coding/decoding very

[1]http://media.caltech.edu/press_releases/13123

low.

Second, we developed a framework that can be used to easily incorporate a variety of compression and decompression algorithms. This framework supports a simple API to integrate compression with an existing application or data processing middleware. Once a compression algorithm is implemented, this framework can allow multi-threaded retrieval, multi-threaded data decompression, and the use of informed prefetching and caching.

Third, we have integrated this framework with a data analysis middleware produced in our earlier work [24], [13]. This framework supports a map-reduce style data processing on large datasets stored locally and/or remotely (which are then retrieved and processed using local resources). This integration allows us to examine the advantages of the compression algorithms and our framework in terms of both data retrieval and wide-area data transfers.

Finally, we evaluated our work using two large datasets (including a GCRM simulation dataset) and three data processing applications. We find that our compression algorithm results in an average compression ratio of 51.68%. Moreover, it outperforms the popular LZO scheme by 38.07% in space and up to 39.76% in performance. For processing on data stored locally, remotely, and both, the performance benefits we obtain range between 40-45% over uncompressed dataset. Our informed prefetching implementation, coupled with multi-threaded decompression, results in up to 27% additional performance improvement.

II. Compression Algorithm for Simulation Data

In this section, we introduce a compression method which can achieve high compression ratio with low decompression overheads. The underlying ideas in this method are applicable to any simulation dataset, although we exemplify the approach using climate simulation data.

To achieve high compression ratios, an algorithm must search for code redundancy and perform complex data transformation for compact data representation. Hence, high CPU and/or memory overheads for both compression and decompression are typical. To achieve better compression overhead and space trade-off, we propose a compression methodology that specifically exploits the domain-specific characteristics of simulation datasets. While many popular compression schemes exploit repeated byte sequences, such long sequences (or *runs*) are infrequent in scientific data, which usually contain highly entropic information represented as floating point values. Our approach is a variation of delta compression with a focus on floating point numbers.

Data sets in the scientific domain often observe spatial and/or temporal properties, which forms a natural dependence in neighboring data points. For example, if we consider a climate dataset, the temperature of a specific location depends heavily on the temperature of the same location in previous time frames, as well as the temperature of neighboring locations. This dependence is typically in the form of a marginal difference in their numerical values. If we consider two cells X and Y that are spatial or temporal neighbors, then we can store X and the difference $\delta = Y - X$, instead of storing both values. Because these δ values are typically distributed in a much smaller range, they can be represented and stored more efficiently.

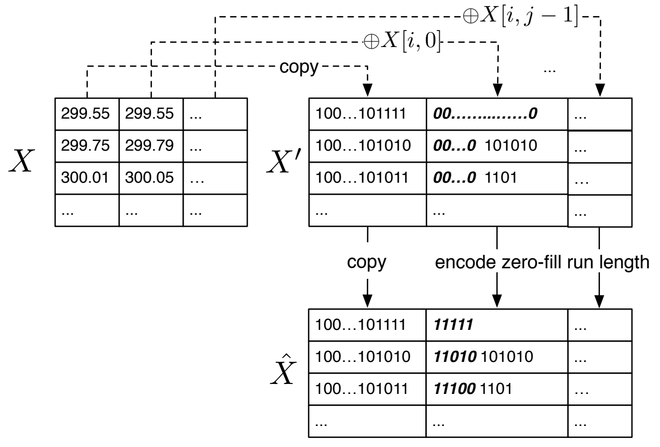

Fig. 1. Climate Compression Example

In Figure 1, we present an example that illustrates this lossless compression scheme over a small dataset. Table X consists of the original climate temperature data. Each cell $X[i, j]$ denotes a temperature reading at latitude i and longitude j, respectively. Next, table X' shows (in binary) the intermediate values, generated as follows,

$$X'[i, j] = \begin{cases} X[i, j], & j = 0 \\ X[i, j] \oplus X[i, j-1], & j > 0 \end{cases}$$

From the binary representation, we can observe that the $\oplus$ (`xor`) operation produces z leading zeros due to the numerical proximity of each cell's neighborhood. This intermediate sequence of bits can now be compressed using a *run-length* to represent z zeroes, followed by a *literal* bit-string representing δ. For instance, a 32-bit intermediate sequence `0x000000EF` can be compressed down to 13 bits which is `11000 1110 1111`. Here, we only use the first 5 bits to represent a *run* of 24 leading zeroes, which is then followed by the 8 bits needed to represent the δ literal, `0xEF`. After this coding is complete, we arrive at the final compressed table $\hat{X}$. It is important to note that the intermediate Table X' is only shown for illustration. Our algorithm directly transforms X to $\hat{X}$ (i.e., there is no additional storage requirement for Table X').

In this example, we considered single precision floating point numbers which are 4 bytes in size. Therefore, the initial 5 bits of the codeword are used for specifying the number of leading zeros, LZ, after applying $\oplus$ operation to adjacent cells. The remaining $32 - LZ$ bits represent the differing bits. This results in a codeword which is instantaneously decodable. Also notice that it is trivial to extend this compression approach to double precision floating point numbers by using 6 bits for the leading zeros, and $64 - LZ$ bits for the differing bits between neighbor cells.

This compression approach also allows us to generate *lossy* codes. For instance, the least significant literal bits representing δ could be ignored. This can be translated into shorter end-

TABLE I
COMPRESSION SYSTEM API: DATA TYPES AND FUNCTIONS

Data Type Declarations

```
typedef struct {                           typedef struct {
   void* user_args;                           file_info_t i_file;
   size_t user_args_size;                     threads_info_t i_threads;
   chunk_args_t* ch_arg;} comp_args_t;        cache_info_t cache_info;} comp_scheduler_args_t;
```

User-Defined Functions for Compression System

```
size_t (*encode_t)(void* input, size_t in_size, void* output, size_t out_size, comp_args_t* comp_args);
```
Required: Encode strategy function is applied to input data chunk, and encoded/compressed data is written to `output`

```
size_t (*decode_t)(void* input, size_t in_size, void* output, size_t out_size, comp_args_t* comp_args);
```
Required: Decode strategy function is applied to `input` compressed data chunk, and decoded/decompressed data is written to `output`

```
int (*prefetch_t)(comp_meta* ch_hist, int n_hist, comp_meta* ch_prefetch, comp_args_t* comp_args);
```
Optional: Prospective data chunks are determined using this function. `chunk_hist` holds the previously accessed data elements.

Provided Functions to Application Layer for I/O Operations

```
size_t comp_read (void* buffer, int count, off_t size, int* eof_check);
int comp_seek (off_t s_offset);
void comp_write(void* buffer, int count, off_t size);
int comp_initialize(comp_scheduler_args_t comp_scheduler_args);
```

to-end application I/O time. In Section IV, we will evaluate over both lossy and lossless codes. Our experiments show that the lossy compression can further improve the performance of data-intensive applications with small error bounds.

III. A Compression System Design for I/O Intensive Application

Besides a climate data specific compression algorithm, another component of our work is a system that can make use of compression transparent to an application or a data processing middleware. This system can incorporate any compression and decompression subsystem in a *plug-and-play* fashion. In this section, we describe the basic components of the compression system, and then show how it can be integrated with a data-intensive middleware [24], [13].

A. System Components

Our compression system, shown in Figure 2, comprises three basic components: Chunk Resource Allocation layer (CRA), Parallel Compression Engine (PCE), and the Parallel I/O Layer (PIOL).

Before the application is executed, the dataset must be compressed and stored. First, dataset is partitioned into *chunks*, and each of these chunks is then compressed independently. The developer can decide the size of the data chunks, which eventually determines the level of parallelism in our compression system. The finer granularity in chunk sizes results in more opportunity for concurrent operations during the execution. However, if the chunk sizes are too small, then the number of I/O requests increases, resulting in overhead.

During the compression of the original dataset, a metadata file is generated per data file. This metadata consists of the `<header,body>` segments. The `header` information includes the location of the compressed file, number of chunks, system buffer size, and user defined data structure into which

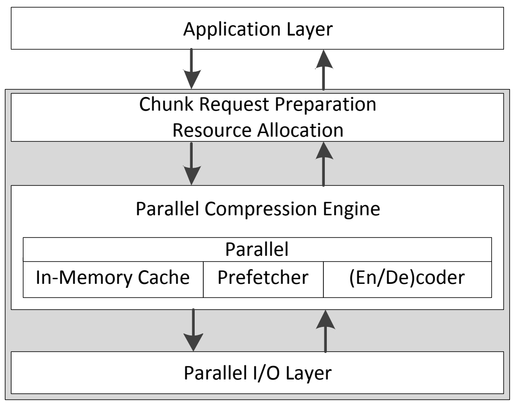

Fig. 2. Compression System Components

the decompressed data is read. The location of the dataset and the storage type enable or disable the parallel I/O functionality of the compression system. The metadata's `body` segment consists of specific information per each compressed data chunk. This includes a chunk's corresponding location in the original dataset and its size. When the application starts, the metadata file(s) are read and the required resources are allocated in the compression system.

In the following, we discuss each component layer in greater detail. We will refer to Figure 3 as we describe the flow of execution in our narrative.

Chunk Resource Allocation (CRA) Layer:

The CRA is the first component that interacts with the application layer, by initializing the compression system and

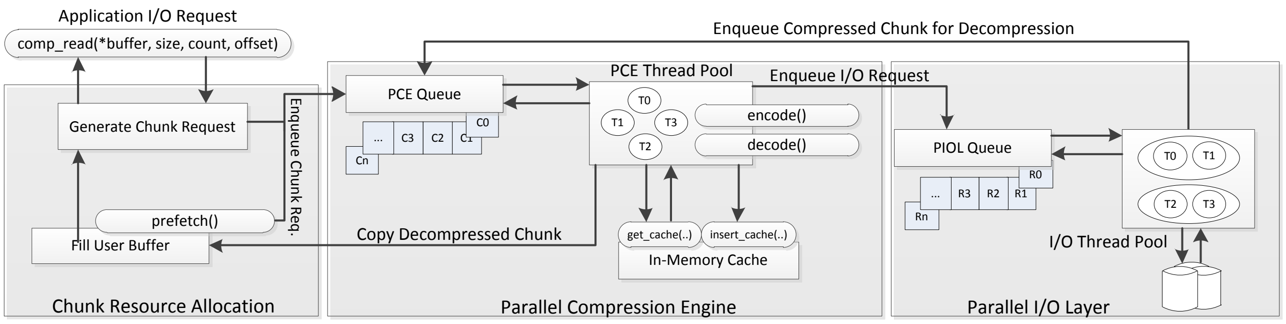

Fig. 3. Execution Flow of Compression System

allocating the necessary resources. A significant aspect of CRA's design is the use of two thread pools in order to provide concurrency for both I/O and computational operations. The threads in these pools are initialized by the CRA after the application layer initializes the compression system. It also allocates the necessary resources for the in-memory cache that we will discuss later.

The CRA exposes a set of functions that can be invoked by the application layer. We present some of these **Provided Functions** in Table I. They are designed to have simple interfaces, similar to any I/O library, so that the compression system can be employed by any application or middleware with a minimal modification to the application layer. Particularly, the application layer can use this API to initiate I/O requests as if it is interacting with the original dataset.

However, the CRA layer transparently converts these to the actual file I/O requests. This involves converting the requested offset addresses into the compressed chunks' offset addresses by using the metadata. Once this conversion is performed, all the chunks corresponding to the user request can concurrently be retrieved and decompressed. Particularly, to use parallelism for data retrieval and compression, the CRA layer determines the corresponding compressed data chunks using the metadata file, and creates the *compressed chunk requests*. Second, these requests are enqueued into a *job queue*, and finally, the CRA layer waits until the user buffer is filled with the original data by the parallel retrieval and decompression threads. Once the user buffer is filled, the CRA returns the call of the application layer.

Parallel Compression Engine (PCE):

The PCE layer is responsible for the main computation of compression system. After the CRA initiates the PCE layer's thread pool, threads begin polling the thread pool's job queue (known as the PCE queue). Once a job is enqueued into the PCE queue, one of the available threads can begin processing it.

At this time, there can be two basic requests which a thread can handle: (1) compressed chunk request, and (2) chunk decompress request. When a PCE thread receives a *compressed chunk request*, it first checks the in-memory cache. If the chunk is readily available, then the content of the data is copied into the user-provided buffer. The copy operation can be performed concurrently, i.e., threads can copy many data chunks to the user buffer at the same time. Since each chunk in the compressed dataset corresponds to different portions of the original data, the copy operations of the threads that work on different chunks are disjoint. If the chunk is not in the cache, then the PCE thread receives a miss, and consequently, I/O requests are created for the missed chunks, and are enqueued to the PIOL queue.

Whenever the parallel I/O (PIOL) layer (described later) retrieves the compressed data chunk, it creates a *chunk decompress request* and enqueues it into the PCE queue for decompression. Our compression system provides an interface for implementing two main functions for the compress and decompress operations, namely, the `decode_t` and `encode_t` functions shown in Table I. If a *chunk decompress request* is read by the PCE thread, it begins applying the user defined `decode_t` to the compressed chunk. Any compression algorithm can be injected into these encode and decode strategy functions. After `decode_t` function is applied to the compressed chunk, a PCE thread copies the decoded data into the user buffer, and sends a signal with the copied data size information to the CRA. The CRA in turn accumulates the size information. Whenever the total size is identical with the user's request, the application layer's request is returned. Meanwhile, the PCE thread inserts the decompressed chunk into the cache and waits for new requests.

Our system exploits a common feature of compression algorithms, which is that a compression algorithm can either be applied to an entire dataset or only a portion. The PCA layer leverages these properties to efficiently and automatically parallelize the compression and decompression of the chunks using customized the compression algorithms. In the compression system, in-memory cache is used for storing the previously accessed data chunks. However, another functionality of it is to increase the efficiency through prefetching, which is described later.

Parallel I/O Layer (PIOL):

If the requested chunk is not already in the cache, it must be retrieved from the compressed dataset. The PIOL layer interacts with the compressed dataset, performing just two basic operations: parallel and sequential I/O. The PIOL component performs parallel I/O whenever possible. Once a compressed chunk request is submitted to the PIOL queue, it checks if the request can be performed concurrently. The I/O threads are grouped so that, while one group of threads perform parallel operations, the other groups can poll the queue and start retrieving the other requested chunks. This approach enables concurrent chunk retrieval and better utilizes the available bandwidth.

Finally, the PIOL layer abstracts the data transfer protocols. For example, a data resource that can be accessed through the web must use the HTTP API, whereas another resource might be accessed simply with the file system I/O interface. In some cases, a single application may even be interacting with data stored in locations with different APIs. The PIOL layer allows a developer to implement I/O operations for different storage resources.

B. Prefetching and the In-Memory Cache

An in-memory cache can help reduce decompression overheads, if certain data chunks are accessed repeatedly. We further improve performance by having the in-memory cache work in conjunction with a prefetching mechanism. The motivation is two-fold: First, for large scientific datasets, the ratio between memory buffer and total dataset size is very small. Second, accesses to large datasets often involve predictable patterns, which can be exploited to mechanize prefetching.

The cache in our system is composed of an ordered set of rows. Each of these rows is a synchronized linked-list. Therefore, any access to consecutive rows can be done concurrently. An insert operation to the cache is efficiently performed through changing the pointers of the link list, and thus no data copy operation is involved. Our cache employs LRU as the default replacement policy.

Our cache does not only hold the data that is already accessed but also the prospective data chunks that are likely to be accessed. The prefetching mechanism allows the compression system to create requests in advance. Although our compression system provides several options for prefetching, the user can also provide a customized prefetching algorithm to the system. The compression system exposes an interface, `prefetch_t`, which provides the history of the previously requested data segments. The user can implement an algorithm to analyze this information and provide hints to the compression system about the prospective data chunks. Then these prospective chunks are converted into chunk requests and enqueued into the PCA layer's queue for concurrent processing and retrieval. The priority of retrieving and decompressing the prospective chunk requests is lower than those of the original chunk request. Therefore, whenever a real chunk request is submitted to any of the system queues, it is pushed in front of the prospective chunk requests and gets processed sooner.

C. Integration with a Data-Intensive Computing Framework

Our compression system can be ported into other applications with small modifications in the application layer. Even though the underlying architecture is complex, it provides a very simple interface, as shown in Table I, to the application developer.

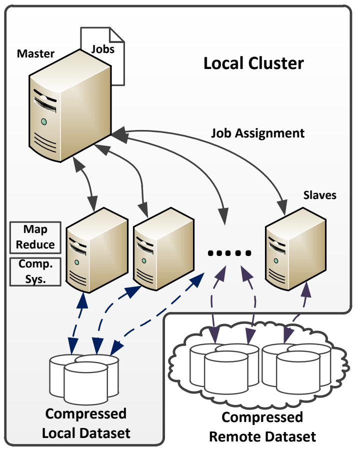

Fig. 4. System Architecture of Data-Intensive Computing Framework

We have ported our compression system into a data-intensive computing framework [24], [13]. This framework provides a Map-Reduce style API for processing data in a local cluster, on the Amazon cloud, and data stored in a distributed fashion across multiple clusters, and/or cloud environments. This framework enables *local* or *co-located* analysis of data similar to a Map-Reduce implementation. In addition, it allows both *remote* and *hybrid* analyses. By *remote* analysis, we refer to a situation where data is stored in a repository (possibly in a cloud setting) that does not have resources for computation. Thus, the data needs to be moved to another cluster for analysis. In *hybrid* mode, data may be stored across a local cluster and a cloud storage system, and/or may be analyzed using resources from both local and cloud settings.

In Figure 4 we show the system architecture of the framework with the compression system. A *local reduction* phase comprises applying an initial processing computation to a data element. After all the data elements in the system are processed, the framework initiates the *global reduction* and computes the final result. The slave nodes are responsible for local reduction phase, while the master node is responsible for assigning the data portions that will be processed by the slave nodes, i.e., job assignment. In the *hybrid* mode, the master node initially assigns the available local data before assigning the remote data.

The compression system is embedded at the points where the framework interacts with the data. All the data movement in the system is performed with compressed data chunks. The decompression operation would be invoked whenever the data is returned to the application layer, i.e., the map and reduce functions. This type of data movement does not only maximize the data transmission throughput from a remote location, but also improves the I/O bandwidth between the local storage resources and the memory.

IV. Experimental Results

This section reports the experimental setup and the results from a detailed evaluation. Particularly, we have the following goals in our experiments:

1) Evaluation of the compression ratio achieved by the lossless and lossy versions of the climate compression algorithm.
2) An analysis of the reduction in execution times of local, remote, and hybrid data processing applications that are built on top of our system.
3) An evaluation of the benefits from prefetching and caching used in our compression system.

A. Experimental Setup

Datasets and Applications: We evaluate our system using three applications. The first application `AT` is I/O-bound, which calculates the average temperature of the different layers in a high-resolution Global Cloud-Resolving Model (GCRM) dataset. The GCRM climate dataset consists of temperature points of the atmosphere with respect to location and time frames. The atmosphere is divided into various layers and each layers' temperature is recorded single-precision floating point numbers. The size of this dataset is 375.41 GB. The second application `MMAT` finds the minimum, maximum, and average temperatures of the different atmospheric layers with respect to different time frames in same climate dataset. The computational intensity is again low for this application, however the volume of intermediate data exchange between the nodes is higher than the `AT` application.

The third application `kmeans` is K-Means clustering, a classic data mining application. It is a CPU-bound application, and far more compute-intensive that the previous applications. In our `kmeans` experiments, we have set $K = 100$, and it is run over the NAS Parallel Benchmark (NPB) numeric message log dataset. The NPB dataset is 237.69 GB in size, consisting double-precision floating point values.

These three applications are implemented using the aforementioned data-intensive processing framework, which enables local, remote, and hybrid data processing. We ported the compression system into the processing framework and replaced the I/O functions with compression system's I/O interface.

Storage and Data Processing Platforms: Both datasets we used are split between local storage and a cloud storage, in order to help evaluate benefits of the compression system using local, remote, and hybrid data analysis scenarios.

The storage system of the local cluster uses the lustre based file system. It has 14 storage nodes with 27 Object Storage Servers (OSS) and 1 Metadata Server (MDS). The nodes are mounted over O2IB, which efficiently utilizes the Quad Data Rate (QDR) InfiniBand network. This storage system provides very high I/O throughput for processing nodes. We used Amazon S3 for remote data storage, with the datasets located at AWS's Northern Virginia cluster. S3 provides relatively lower I/O throughput compared to our local storage system. We stored 270 GB of climate data in the local cluster, while the remaining portion 105 GB is stored on S3. Similar to climate dataset, the NPB dataset is separated into 166 GB stored in the local cluster and 71 GB stored on S3.

The user-defined *chunk sizes* of compressed local and remote datasets are set to 64MB and 512MB, respectively. The larger chunk sizes result in less communication overhead for the remote data retrieval. On the other hand, the high performance local storage infrastructure provides enough I/O throughput with 64MB chunk sizes for parallel decompression and prefetching.

All processing nodes are allocated from the local cluster located at the Ohio State University campus. Each compute node contains Intel Xeon 2.53GHz processor (8 cores) with 12GB memory. These nodes are connected with InfiniBand network. We used 16 compute nodes (total of 128 cores) during our experiments.

Compression Methods: To choose an ideal baseline for detailed comparison with our proposed approach, we applied several well-known compression algorithms on a 2.2 GB climate dataset.

We considered two parameters while selecting a suitable compression algorithm: Decompression speed and compression ratio. In Table II, we present compressed size of the datasets, as well as the compression and decompression times of various algorithms. The most efficient algorithm, both in terms of performance and compression ratio, is our domain-specific climate compression (`CC`) algorithm. The remaining algorithms possess different properties. LempelZiv-Markov (`LZMA`) shows the most compression ratio whereas the Lempel-Ziv-Oberhumer (`LZO`) algorithm is the fastest. Since decompression time is a bottleneck for `LZMA`, `LZO` is selected as a baseline algorithm for comparing against `CC`. `LZO` is a generic compression algorithm which can be applied to any dataset. It provides very fast decompression rates and is being used by popular systems [33], [1].

TABLE II
Initial Compression Algorithm Comparison

	`gzip`	`bzip2`	`LZO`	`LZMA`	`CC`
Comp. Size (MB)	1504	1304	1725	1113	1065
Comp Time (s)	191.44	380.18	367.74	371.88	12.63
Decomp Time (s)	44.97	167.83	17.754	133.91	9.75

Besides an evaluation of our climate dataset compression method, we also wanted to demonstrate the generality of our software framework. For the NPB dataset, we used an efficient double-precision floating point compression (`FPC`) [14].

All the experimental datasets are compressed with parameters that yield to the highest compression ratios. In general, the compression phase of the datasets might take a significant amount of time with these parameters. However, the decompression rates of the selected compression algorithms do not show significant changes. Since we focus on processing large scientific datasets, efficient read operations and high compression ratios are our priorities.

B. Compression Ratios

We first analyze the compression ratios of the algorithms that we have used in our study. Table III shows the size of the original dataset and the compressed datasets.

TABLE III
COMPRESSED DATASET SIZES

	Orig	CC	CC-2e	CC-4e	FPC	LZO
Climate (GB)	375.41	185.15	161.88	139.15	–	298.97
NPB (GB)	237.69	–	–	–	180.61	237.69

The first row and column specify the type of the compression algorithms and the datasets, respectively. CC refers to the lossless climate compression algorithm and CC-*e specifies the number of dropped least significant bits for lossy climate compression algorithm. The Orig column presents the original dataset size. For the climate dataset, the compression ratio of the lossless CC algorithm is 2.03:1, resulting in 51.68% less space. The lossy climate compression algorithms generate space savings of 56.88% and 62.93%, for CC-2e and CC-4e, respectively. LZO uses 20.4% less space, so the compression ratio of CC over LZO is 1.62:1, highlighting the benefits for our design of a domain-specific scheme for climate/simulation data. For the NPB dataset, the ratio of saved space is 24.01% with FPC. Note that the LZO algorithm is not able to compress the data.

We underline that compression provides at least three benefits in a data analysis scenario. First, it can reduce storage space requirements, and in the case of a cloud environment, this translates to lower storage costs. Second, compression can reduce data transmission costs when data analysis is performed in remote or hybrid settings. Finally, it can reduce data retrieval times. The second and third benefits above can be negated by higher data analysis times, because of the need for decompression after data retrieval. Thus, to understand the true benefits of a compression scheme, we focus on the execution times for a complete data analysis task in the next subsection.

C. System Performance

In our first set of experiments, we compare the execution times of our data-intensive applications with different compression algorithms and environmental settings. Each of the test cases was repeated at least five times and the mean values of the execution times are presented.

In Figure 5, we compare the execution times of the AT application using the climate dataset, considering the three different data storage/processing scenarios: local, remote, and hybrid. For the local mode, only the local data (270 GB) is being processed by the compute nodes. Since the I/O operations are performed locally, the data throughput is higher. The processing nodes retrieve data from a remote (cloud) location in the remote setting where the total stored data size is 105 GB. This type of processing involves more data retrieval and movement time due to the limited wide-area bandwidth. In hybrid setting, both local and remote data are being processed. The hybrid mode simulates the scenarios where one dataset is geographically distributed.

First, we show the overhead of our compression system. The Original-Comp and Original present the elapsed times for processing original (uncompressed) dataset with and without compression system, respectively. The overhead of our system is negligible. For Original-Comp, the decode_t() function copies the retrieved data to the write-buffer without applying any decompression algorithm. Our compression system further avoids copy operations internally. The only parts where the data is being written/copied are the encoding and decoding functions. Since these operations are done in memory, the system's overheads are not noticeable.

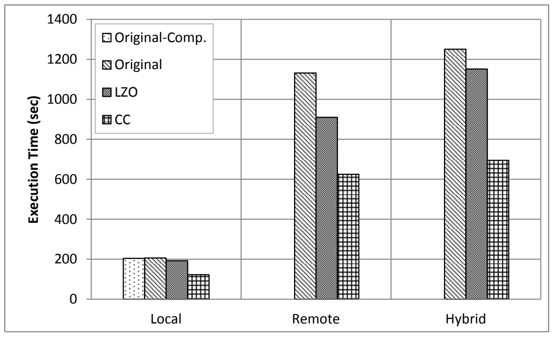

Fig. 5. Execution Times of the AT Application

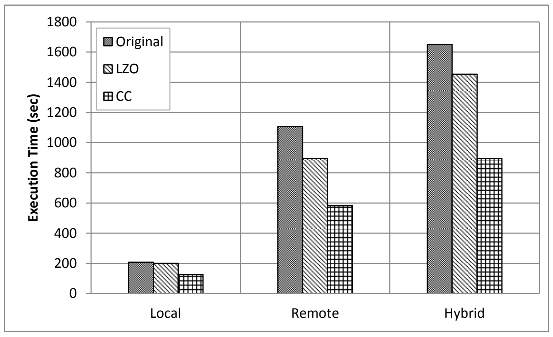

Fig. 6. Execution Times of the MMAT Application

If we focus on different compression algorithms, we see that processing the local dataset with CC algorithm shows a 1.67 speedup over processing the original dataset on the local cluster.

As we noted earlier, there are two factors that impact the performance of the application: the compression ratio of the algorithm and decompression speed. The compression ratio of the algorithm helps with bandwidth usage. The compression ratio of the CC algorithm is more than 2, thus the same amount of information can be read in half time. Therefore, from the overall performance, we can see that decompression method with the CC algorithm introduces only modest overheads.

The LZO compression algorithm shows small improvement in the execution time. The speedup of using LZO is only 1.07 and the saved storage space is 20.4%. If we compare CC and LZO algorithms, we observe that CC outperforms LZO in both execution time and storage space, again showing benefits of developing domain-specific compression algorithms.

For the remote processing configuration, the application performance increases with the compressed dataset and system. Specifically, the speedups of CC and LZO algorithms are 1.81

and 1.24, respectively. The gained benefit from the remote data retrieval times dominates the decompression overhead, and thus decreases the total execution time of the application.

Lastly, we focus on the hybrid configuration. In this setting, the application processes both local and remote data, which is a common configuration when large scientific datasets exceed the limit of local resources. The application still shows significant performance increase with compression system and algorithms. The speedups of using `CC` and `LZO` algorithms are 1.8 and 1.08, respectively.

In Figure 6, we show our results with the `MMAT` application. The speedups of `MMAT` follow the same pattern as with the `AT` application. The `CC` algorithm's speedups are 1.63, 1.90 and 1.85 for local, remote and hybrid settings, respectively. `LZO` compression, on the other hand, shows 1.04, 1,24 and 1.14 speedups for the same settings. Again, our domain-specific compression shows better performance than the generic algorithm.

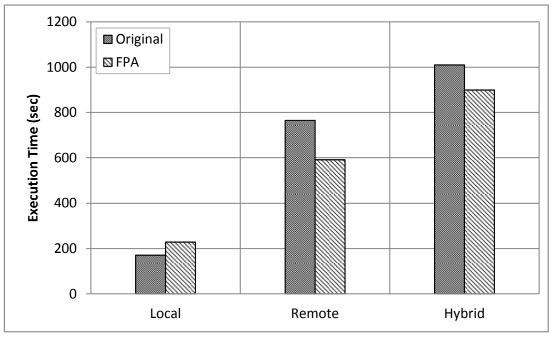

Fig. 7. Execution Times of K-Means Application

To show the generality of our framework, we show the results of `kmeans` application in Figure 7. Similar to previous experiments, the `NPB` dataset is separated into two sets. The size of the stored data portions are 166 GB and 71 GB over local and remote resources, respectively. Recall that `kmeans` is more compute-intensive than the previous climate application. Our first observation is the small overhead with `FPC` in the local setting, which provides relatively high I/O throughput. Therefore, the decompression time results in 33.29% overall slowdown during the execution. If we focus on remote and hybrid settings, we observe that the application can benefit from the compression system and finish the execution in a shorter time period. Similar to `MMAT` and `AT`, even `kmeans` (which is much less I/O-intensive) can benefit from the data retrieval time during the remote data transfer. This gain dominates the decompression time and therefore can decrease the overall execution time. The speedups are 1.12 and 1.30 for remote and hybrid settings, respectively.

To understand the trade-off between data retrieval/movement cost, reduction and decompression times, we studied a detailed breakdown of performance. In Figure 8, we present the detailed execution times of local and remote configurations of `MMAT`. `Original` represents the execution time of the original dataset. The `Read` represents the elapsed time to read the data and transfer it to the application layer. The `Reduction` represents the application

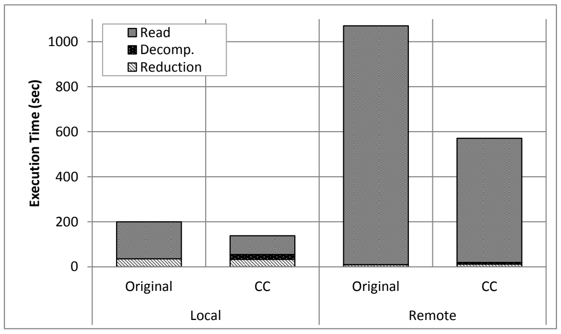

Fig. 8. Breakdown of MMAT Execution Time

layer's time taken to perform the computation. Since this configuration does not work on the compressed dataset, the decompression time, `Decomp`, is zero. If we compare the data read time of `Original` and `CC` settings for the local configuration, we can see that the speedup of using `CC` for read operations is 1.96 over original dataset. However, the decompression time introduces 15.41% overhead in `CC` execution time. Thus, the overall speedup is 1.63.

D. Performance with Lossy Compression

We have repeated the same set of experiments with a lossy `CC` algorithm and compared the execution times and accuracy of the final results with the lossless version. Recall that the `climate` dataset consists of single-precision floating numbers which represent the temperature of different layers in atmosphere. Our lossy `CC` algorithm drops the least significant bits of the floating point numbers. Therefore, it provides better compression ratios with reasonable error rates, without increasing the complexity of compression or decompression.

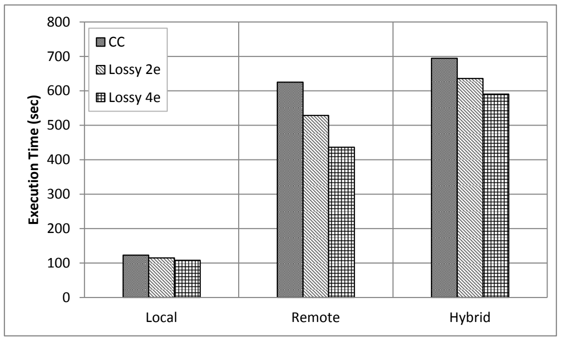

Fig. 9. Execution Times of Climate Application with Lossy Compression

In Figure 9, we present the results with different configurations for AT application. Again, `CC` corresponds to the lossless compression algorithm, whereas `Lossy 2e` and `Lossy 4e` refer to lossy compression, dropping the last 2 and 4 bits, respectively.

Our first, and expected, observation is the reduced execution times with the lossy algorithms. The speedups for `Lossy 2e` over the lossless compression algorithm are 1.07, 1.18 and 1.09 for local, remote, and hybrid settings. The `Lossy 4e` provides 1.13, 1.43, and 1.18 for the same settings. A similar trend can be observed with `MMAT` in Figure 10. The speedups of using lossy compression algorithms range from 1.05 to 1.13 for `Lossy 2e` and 1.08 to 1.26 for `Lossy 4e`. Furthermore, if we compare `Lossy 4e` with the `Original` in Fig. 6, the speedups are 1.76, 2.41 and 2.18.

Clearly, use of a lossy compression method results in some loss of accuracy for the final results. However, for the applications we have considered here, which focus on obtaining a summary or aggregate information from a large dataset, we find that the loss of accuracy is negligible. Specifically, the error bound for our lossy algorithms is $5e^{-5}$.

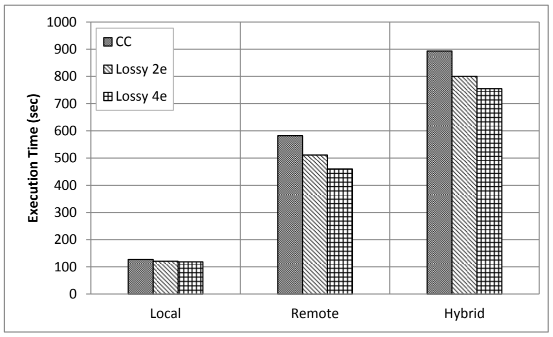

Fig. 10. Execution Times of Climate-Avg Application with Lossy Compression

E. Benefits of Multithreading and Prefetching

In the previous set of experiments, the prefetching and multithreading mechanisms were not being used, and number of I/O and processing threads were set to one. In the following set of experiments, we implement the `prefetch_t()` function so that it returns two prospective data blocks to the compression system after each I/O request from the application. While the application processes the requested data, the compression system fetches the prospective chunks and decompresses them using the PIOL and PCE threads. We chose a different number of PIOL and PCE threads and compared the execution times using the `FPC` and `CC` compression schemes.

For example, the 2P-4IO configuration denotes four PIOL and two PCE threads. We typically use a larger number of PIOL threads than the PCE threads. For each processing thread, there are multiple I/O threads and hence, each chunk can concurrently be fetched and decompressed. Note that each chunk can be decompressed by only one PCE thread, but it can be retrieved by many I/O threads. In this set of experiments, we also assigned one of the CPU cores to specifically operate the compression system. This would reduce the total computing resources available for the application's needs, and thus, we need to examine if the benefits from multi-threaded decompression can overcome the reduction in compute resources.

In Figure 11, we present the execution times of `kmeans` with different numbers of PIOL and PCE threads. Focusing on the local processing scenario, we can observe that the execution times expectedly decrease with increasing number of threads. The prefetching and caching mechanism in our compression system efficiently overlaps the data retrieval and decompression operations with application's computation. If the application requires more computation, the compression system finds more opportunity to fetch and decompress the prefetched data chunks. Since `kmeans` application requires time for its computation, the compression system is able to prepare prefetched chunks for the next request. The speedups for this setting are 1.25 and 1.37 for 2P-4IO and 4P-8IO, respectively.

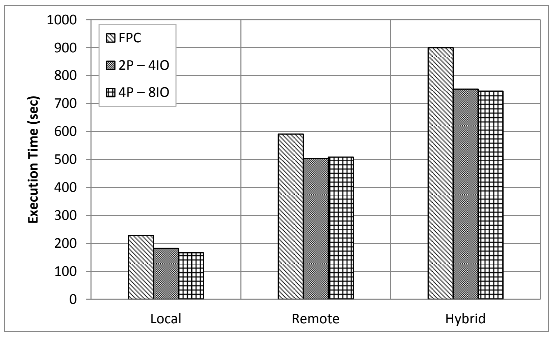

Fig. 11. Execution Times of K-Means Application with Different PIOL and PCE Threads

If we analyze the remote and hybrid settings, we see that the application still benefits from the new configurations. However, if we compare 4P-8IO with 2P-4IO, we observe that the application performance stays the same. The 2P-4IO can utilize all the bandwidth using 4 PIOL threads, thus 4P-8IO is not able to provide any additional benefit. The speedups of these settings range from 1.16 to 1.21.

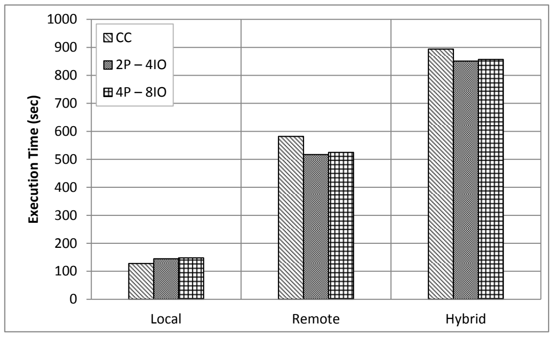

Fig. 12. Execution Times of MMAT Application with Different PIOL and PCE Threads

Next, we consider the `MMAT` application, with results shown in Figure 12. As we mentioned before, the prefetching and caching mechanism of the compression system exploits the computation time of the application. The `MMAT` application is I/O bound, and therefore, for the local processing configuration, the application finishes its computation and requests more data chunks before it is fetched. Even though the system takes advantage of the small computation time of the application, the context switching of the threads introduced an overhead. The slowdowns of the application are below 14% for this setting. Similar to `kmeans`, the compression system maximizes the utilization of the available bandwidth using PIOL threads for remote data retrieval. Therefore, it is able to benefit from parallel I/O requests. The speedups of hybrid and remote configurations range between 1.05 and 1.11 for `MMAT`.

V. Related Work

Compression [28], [20] has been an attractive topic for systems that deal with large datasets for a number of years. Our proposed compression method is a variation of delta encoding in which the relationships between adjacent data points are exploited. There are many types of delta compression methods which have been used in different areas, including signal processing. Our compression method is specifically tailored for GCRM dataset and takes advantage of its properties.

Another compression method which has similarities with our approach is discrete quantization. This method is also efficient in both space saving and (de)compression throughput, which make it suitable for real time data analysis, including stream processing [8]. It uses mean value in order to compute the deltas of a group of data points. Then each data point is mapped to a table cell, in which each cell is specified with an n-bit value. The difference of original values and delta are stored in these cells. The table size determines if the compression method is lossy or not. This compression method needs to store mean value, and the n-bit location information for each of the data point. Considering the highly entropic content of the scientific datasets, the table size might be large which can easily be a bottleneck.

In filesystems, various compression systems are employed to maximize available storage. For instance, Windows NTFS provides transparent data compression using Lempev-Ziv algorithm [17]. Similarly, FuseCompress [2], a mountable filesystem, and e2compr [9], an extension for ext2, can be used to add compression feature to Linux. Typically, the compression algorithms provided by filesystems need to be generic to store various data types. In contrast, we focus on efficient storage of scientific datasets which are difficult to compress with generic compression algorithms.

Compression has also extensively been used in database systems in order to improve the storage requirements and the query execution times[23], [6], [35], [30], [22]. ISABELA-QA [26] is a parallel query processing engine which exploits *knowledge priors*. It enables efficient processing for both spatial-region and variable-centric queries using compressed scientific datasets and indices. Their system uses ISABELA [27] for compressing the data.

Outside of databases, data-intensive middleware systems have also explored the use of compression. Nicoloe, *et al.* have developed BlobSeer, a distributed data management service that provides efficient reading, writing, and appending functionalities to its users [32]. A recent work by the same authors focuses on a transparent compression system that is built in BlobSeer [31]. Their system provides optimizations, such as overlapping decompression with I/O operations and selecting the compression algorithm. While our system also takes advantage of overlapping application computation and (de)compression operations, we further use informed prefetching and pipelined decompression to increase concurrency.

Welton, *et al.* developed a set of compression services that transparently perform compression operations [39]. Their services run on data nodes that handle the I/O requests. Apache has also initiated several projects for large-scale data analysis and storage [4], [5]. These systems use generic compression algorithms, such as LZO[1], [33], in order to optimize I/O. Similarly, Google's data storage and analysis systems, MapReduce[18] and BigTable[15], take advantage of compression algorithms; an example being Snappy[3]. Another data analysis system, DryadLINQ[41], uses compression for its intermediate data exchange, thus improving the I/O throughput. These data-intensive frameworks provide generic compression algorithms which do not perform well with scientific datasets.

Schendel, `et al.` proposed a compression methodology, ISOBAR, for detecting highly entropic contents in hard-to-compress data, such as scientific datasets [37]. They consider analyzing double precision floating point numbers byte-by-byte, and then apply the best compressor. In [38], authors introduce a hybrid framework which overlaps I/O and compression operations. While we also focus on compressing scientific datasets, we assume that the user has certain domain specific insights about the data and can exploit its properties. Therefore, a compression algorithm can directly apply a bit-level compression and bypass byte-level analysis. Furthermore, our system uses informed prefetching to further increase I/O performance and amortize the compression overhead.

Compression has also been used for enhancing the performance and storage of cache and memory [34], [21], [19], [36], [40]. A recent work of Makatos, *et al.* introduced FlaZ, a transparent compression system, which leverages the performance of SSD-based caches [29]. FlaZ operates on the kernel level and resides in I/O path of user application and SSD. It provides two generic compression algorithms, zlib and LZO, and supports I/O concurrency. If the storage medium can handle parallel I/O requests, e.g. SSDs, concurrent operations can significantly increase the performance.

In comparison to these efforts, our focus is on compression of scientific datasets and developing a system that provides an easy-to-use API for integrating new compression algorithms, while supporting optimizations such as prefetching and multi-threaded decompression. Furthermore, our system can be easily integrated with various data-intensive applications and middleware systems.

There have been studies on the effects of using compression for energy efficiency. Although the energy efficiency of compression algorithms has been researched in the context of sensor networks and wireless devices [11], [10], research on the energy consumption of modern data centers is more recent. In [16], authors propose a decision algorithm that allows users identify if compression is beneficial or not, in terms of energy consumption. The authors implement their system in MapReduce and show that compression can increase the energy efficiency up to 60%. We have not studied energy benefits from our system, though similar benefits can be expected.

Other efforts have focused on management and dissemination of scientific simulation datasets, such as the earth's climate system simulation data [12]. Transferring and accessing climate datasets have been an active research topic [25], [7].

VI. Conclusion

This paper presented the challenge of making compression more effective, practical, and usable for applications that analyze large-scale scientific datasets. First, we introduced a compression method which can efficiently compress and decompress highly entropic scientific datasets through exploiting domain specific properties. Second, we developed a framework in which different compression algorithms can easily be integrated. This framework automatically parallelizes I/O and (de)compression operations, and overlaps these with computation. Lastly, we presented how our compression framework can easily be integrated with data-intensive middleware or applications.

We have evaluated our work using two large datasets (including a climate dataset) and three data processing applications. We find that our compression method results in an average compression ratio of 51.68%. Moreover, it outperforms the popularly known algorithm LZO by 38.07% in space saving, and up to 39.76% in performance. For local, remote, and hybrid data processing, the performance benefits we obtain range between 40-53% over uncompressed dataset. Our informed prefetching implementation, coupled with multi-threaded decompression, results in up to 27% additional performance improvement over the version without prefetching or multi-threading.

Acknowledgment

The authors would like to thank Robert L. Jacob from Argonne National Laboratory for his support and assistance. This work was supported in part by the Office of Biological and Environmental Research, Office of Science, U.S. Department of Energy, under contract DE-ACO2-O6CH11357.

References

[1] Enabling LZO Compression in HBase. http://wiki.apache.org/hadoop/UsingLzoCompression/, 2012. [Online; accessed September-2012].

[2] FuseCompress. http://code.google.com/p/fusecompress/, 2012. [Online; accessed September-2012].

[3] Snappy. http://code.google.com/p/snappy/, 2012. [Online; accessed September-2012].

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

[5] The Apache HBase. http://hbase.apache.org/, 2012. [Online; accessed September-2012].

[6] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[7] W. E. Allcock, I. T. Foster, V. Nefedova, A. L. Chervenak, E. Deelman, C. Kesselman, J. Lee, A. Sim, A. Shoshani, B. Drach, and D. N. Williams. High-performance remote access to climate simulation data: a challenge problem for data grid technologies. In *SC*, page 46, 2001.

[8] F. Altiparmak, D. Chiu, and H. Ferhatosmanoglu. Incremental quantization for aging data streams. In *Proceedings of the Seventh IEEE International Conference on Data Mining Workshops*, ICDMW '07, pages 527–532, Washington, DC, USA, 2007. IEEE Computer Society.

[9] L. Ayers. E2compr: Transparent file compression for linux., 1997.

[10] S. J. Baek, G. de Veciana, and X. Su. Minimizing energy consumption in large-scale sensor networks through distributed data compression and hierarchical aggregation. *Selected Areas in Communications, IEEE Journal on*, 22(6):1130 – 1140, aug. 2004.

[11] K. C. Barr and K. Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, Aug. 2006.

[12] D. E. Bernholdt, S. Bharathi, D. Brown, K. Chanchio, M. Chen, A. L. Chervenak, L. Cinquini, B. Drach, I. T. Foster, P. Fox, J. Garcia, C. Kesselman, R. S. Markel, D. Middleton, V. Nefedova, L. Pouchard, A. Shoshani, A. Sim, G. Strand, and D. N. Williams. The earth system grid: Supporting the next generation of climate modeling research. *CoRR*, abs/0712.2262, 2007.

[13] T. Bicer, D. Chiu, and G. Agrawal. A Framework for Data-Intensive Computing with Cloud Bursting. In *Proceedings of Conference on Cluster Computing*, Sept. 2011.

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

[15] 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.

[16] Y. Chen, A. Ganapathi, and R. H. Katz. To compress or not to compress - compute vs. io tradeoffs for mapreduce energy efficiency. In P. Barford, J. Padhye, and S. Sahu, editors, *Green Networking*, pages 23–28. ACM, 2010.

[17] M. Corp. File Compression and Decompression in NTFS. http://msdn.microsoft.com/en-us/library/Aa364219/, 2012. [Online; accessed September-2012].

[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[19] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *In Proceedings of 1993 Winter USENIX Conference*, pages 519–529, 1993.

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

[21] M. Ekman and P. Stenstrom. A robust main-memory compression scheme. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 74 – 85, june 2005.

[22] G. Graefe and L. Shapiro. Data compression and database performance. In *Applied Computing, 1991., [Proceedings of the 1991] Symposium on*, pages 22 –27, apr 1991.

[23] T. H. Group. Szip Compression in HDF5. http://http://www.hdfgroup.org/doc_resource/SZIP/, 2012. [Online; accessed September-2012].

[24] W. Jiang, V. T. Ravi, and G. Agrawal. A Map-Reduce System with an Alternate API for Multi-core Environments. In *CCGRID*, pages 84–93, 2010.

[25] R. Kettimuthu, A. Sim, D. Gunter, B. Allcock, P.-T. Bremer, J. Bresnahan, A. Cherry, L. Childers, E. Dart, I. Foster, K. Harms, J. Hick, J. Lee, M. Link, J. Long, K. Miller, V. Natarajan, V. Pascucci, K. Raffenetti, D. Ressman, D. Williams, L. Wilson, and L. Winkler. Lessons learned from moving earth system grid data sets over a 20 gbps wide-area network. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010)*, Jun 2010.

[26] S. Lakshminarasimhan, J. Jenkins, I. Arkatkar, Z. Gong, H. Kolla, S.-H. Ku, S. Ethier, J. Chen, C. S. Chang, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. Isabela-qa: query-driven analytics with isabela-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 31:1–31:11, New York, NY, USA, 2011. ACM.

[27] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova. Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data. In E. Jeannot, R. Namyst, and J. Roman, editors, *Euro-Par (1)*, volume 6852 of *Lecture Notes in Computer Science*, pages 366–379. Springer, 2011.

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

[29] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas. Using transparent compression to improve ssd-based i/o caches. In C. Morin and G. Muller, editors, *EuroSys*, pages 1–14. ACM, 2010.

[30] W. K. Ng and C. V. Ravishankar. Block-oriented compression techniques for large statistical databases. *IEEE Trans. on Knowl. and Data Eng.*, 9(2):314–328, Mar. 1997.

[31] 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.

[32] B. Nicolae, G. Antoniu, and L. Bougé. Blobseer: how to enable efficient versioning for large object storage under heavy access concurrency. In *Proceedings of the 2009 EDBT/ICDT Workshops*, EDBT/ICDT '09, pages 18–25, New York, NY, USA, 2009. ACM.

[33] M. Oberhumer. LZO, A real-time data compression library. http://oberhumer.com/opensource/lzo/, 2012. [Online; accessed September-2012].

[34] O. Ozturk, M. T. Kandemir, and M. J. Irwin. Using data compression for increasing memory system utilization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 28(6):901–914, 2009.

[35] M. Pöss and D. Potapov. Data compression in oracle. In *VLDB*, pages 937–947, 2003.

[36] L. Rizzo. A very fast algorithm for ram compression. *SIGOPS Oper. Syst. Rev.*, 31(2):36–45, Apr. 1997.

[37] 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.

[38] 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.

[39] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. B. Ross. Improving i/o forwarding throughput with data compression. In *CLUSTER*, pages 438–445, 2011.

[40] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association.

[41] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In R. Draves and R. van Renesse, editors, *OSDI*, pages 1–14. USENIX Association, 2008.