Understanding and Modeling Lossy Compression Schemes on HPC Scientific Data

Tao Lu[1], Qing Liu[1,2], Xubin He[3], Huizhang Luo[1], Eric Suchyta[2], Jong Choi[2], Norbert Podhorszki[2], Scott Klasky[2], Mathew Wolf[2], Tong Liu[3], and Zhenbo Qiao[1]

[1] Department of Electrical and Computer Engineering, New Jersey Institute of Technology
[2] Computer Science and Mathematics Division, Oak Ridge National Laboratory
[3] Department of Computer and Information Sciences, Temple University

***Abstract*—Scientific simulations generate large amounts of floating-point data, which are often not very compressible using the traditional reduction schemes, such as deduplication or lossless compression. The emergence of lossy floating-point compression holds promise to satisfy the data reduction demand from HPC applications; however, lossy compression has not been widely adopted in science production. We believe a fundamental reason is that there is a lack of understanding of the benefits, pitfalls, and performance of lossy compression on scientific data. In this paper, we conduct a comprehensive study on state-of-the-art lossy compression, including ZFP, SZ, and ISABELA, using real and representative HPC datasets. Our evaluation reveals the complex interplay between compressor design, data features and compression performance. The impact of reduced accuracy on data analytics is also examined through a case study of fusion blob detection, offering domain scientists with the insights of what to expect from fidelity loss. Furthermore, the trial and error approach to understanding compression performance involves substantial compute and storage overhead. To this end, we propose a sampling based estimation method that extrapolates the reduction ratio from data samples, to guide domain scientists to make more informed data reduction decisions.**

I. Introduction

Cutting-edge computational research in various domains relies on high-performance computing (HPC) systems to accelerate the time to insights. Data generated during such simulations enable domain scientists to validate theories and investigate new microscopic phenomena in a scale that was not possible in the past. Because of the fidelity requirements in both spatial and temporal dimensions, analysis output produced by scientific simulations can easily reach terabytes or even petabytes per run [1]–[3], capturing the time evolution of physics phenomena in a fine spatiotemporal scale. The volume and velocity of data movement are imposing unprecedented pressure on storage and interconnects [4], [5], for both writing data to persistent storage and retrieving them for post-simulation analysis. As HPC storage infrastructure is being pushed to the scalability limits in term of both throughput and capacity [6], the communities are striving to find new approaches to curbing the storage cost. Data reduction[1], among others, is deemed to be a promising candidate by reducing the amount of data moved to storage systems.

[1]The term of data reduction and compression are used interchangeably throughout this paper.

Data deduplication and lossless compression have been widely used in general-purpose systems to reduce redundancy in data. In particular, deduplication [7] eliminates redundant data at the file or chunk level, which can result in a high reduction ratio if there are a large number of identical chunks at the granularity of tens of kilobytes. For scientific data, this rarely occurs. It was reported that deduplication typically reduces dataset size by only 20% to 30% [8], which is far from being useful in production. On the other hand, lossless compression in HPC was designed to reduce the storage footprint of applications, primarily for checkpoint/restart. Shannon entropy [9] provides a theoretical upper limit on the data compressibility; simulation data often exhibit high entropy, and as a result, lossless compression usually achieves a modest reduction ratio of less than two [10]. With the growing disparity between compute and I/O, more aggressive data reduction schemes are needed to further reduce data by an order of magnitude or more [11], and very recently the focus has shifted towards lossy compression.

Lossy compression, such as JPEG [12], has long been used in computer graphics and digital images, leveraging the fact that the visual resolution by human eyes is well below machine precision. However, its application in the scientific domain is less well established. Since scientific data are primarily composed of high-dimensional floating-point values, lossy floating-point data compressors have begun to emerge, including ISABELA [13], ZFP [14], and SZ [15]. Although lossy reduction offers the most potential to mitigate the growing storage and I/O cost, there is a lack of understanding of how to effectively use lossy compression from a user perspective, e.g., which compressor should be used for a particular dataset, and what level of reduction ratio should be expected. To this end, the paper aims to perform extensive evaluations of state-of-the-art lossy floating-point compressors, using real and representative HPC datasets across various scientific domains. We focus on addressing the following broad questions:

- Q1: What data features are indicative of compressibility? (Section III-A)
- Q2: How does the error bound influence the compression ratio? Which compressor (or technique) can benefit the most from loosening error bound? (Section III-B)

- Q3: How does the design of compression influence compression throughput? What is the relationship between compression ratio and throughput? (Section III-C)
- Q4: What is the impact of lossy compression on data fidelity and complex scientific data analytics? (Section III-D)
- Q5: How to extract data features and accurately predict the compression ratios of various compressors? (Section IV)

Through answering these questions, we aim at helping HPC end users understand what to expect from lossy compressors. As a completely unbiased third-party evaluation without ad-hoc performance tunings, we hope to shed light on the limitations of existing compressors, and point out some of the new R&D opportunities for compressor developers and the communities to make further optimizations, thus ensuring the broad adoption of reduction in science production. Our experiments for evaluating floating-point data compressors, including scientific datasets and scripts we used for evaluation, are publicly available at https://github.com/taovcu/LossyCompressStudy.

II. Background

Data deduplication and lossless compression fully maintain data fidelity and reduce data by identifying duplicate contents through the hash signature and eliminating redundant data. We utilize two lossless schemes throughout this work, GZIP [16] and FPC [17], for performance comparisons with lossy compression. The *deflate* algorithm implemented in GZIP is based on LZ77 [18], [19]. The core of the algorithm is comparing the symbols in the look-ahead buffer with symbols in the search buffer to determine a match. FPC employs *fcm* [20] and *dfcm* [21] to predict the double-precision values. An XOR operation is conducted between the original value and the prediction. With a high prediction accuracy, the XOR result is expected to contain many leading zero bits, which can be easily compressed. Ultimately, the effectiveness of these methods depends on the repetition of symbols in data. However, for even slightly variant floating-point values, the binary representations contain few identical symbols. Hence, scientific simulations use lossless compression only if necessary, e.g., for checkpoint/restart [22].

The general acceptance of precision loss provides an opportunity to drastically improve the data compression ratio - if two symbols are within the error tolerance, they can be represented using the same code. This paper includes studies using three lossy compressors: ISABELA, SZ, and ZFP, which were shown to be superior in prior work [15]. Each compressor is briefly described as follows.

Motivated by fixed-rate encoding and random access, ZFP [23] follows the classic texture compression for image data. Working in 4^d (where d is the number of dimensions) sized blocks, ZFP first aligns the floating-point data points within a block to a common exponent, which is determined by the largest absolute value. The original data in the block is then converted to mantissas along with the common exponent. Second, the exponent is encoded and stored. The mantissas are then converted to fixed-point signed integers. Third, a reversible orthogonal block transform (e.g., discrete cosine transform) is applied to the signed integers. This transform is carefully designed to mitigate the spatial correlation between data points, with the intent of generating near-zero coefficients that can be compressed efficiently. Finally, embedded coding [24] is used to encode the coefficients, producing a stream of bits that is roughly ordered by their impact significance on error, and the stream can be truncated to satisfy any user-specified error bound.

Motivated by the reduction potential of spline functions [25], [26], ISABELA [13] uses B-spline based curve-fitting to compress the traditionally incompressible scientific data. Intuitively fitting a monotonic curve can provide a model that is more accurate than fitting random data. Based on this, ISABELA first sorts data to convert highly irregular data to a monotonic curve. Similarly, SZ [15] employs multiple curve-fitting models to encode data streams, with the goal of accurately approximating the original data. SZ compression involves three main steps: array linearization, adaptive curve-fitting, and compressing the unpredictable data. To reduce memory overhead, it uses the intrinsic memory sequence of the original data to linearize a multi-dimensional array to a one-dimensional sequence. The best-fit routine employs three prediction models, based on the adjacent data values in the sequence: constant, linear, and quadratic, which require one, two, and three precursor data points, respectively. And the model that yields the closest approximation is adopted. If none satisfies the pre-defined error bound, SZ marks the data point as unpredictable, which is then encoded by binary-representation analysis. The curve-fitting step transforms the fitted data into integer quantization factors, which are further encoded using Huffman tree. Unlike ISABELA [13], SZ does not sort the original data to avoid the indexing overhead. The encoded data are further compressed using GZIP.

While lossy compression has been identified as a means to potentially reduce scientific data by more than 10x, determining the compressibility of data without compressing the full data, and the impact of information loss on data analytics have not been fully studied. Although trial and error can certainly answer these questions, this incurs overhead in terms of compute and storage, and should be avoided as much as possible. Our proposed evaluation and modeling aim to fill these gaps in data reduction, and allow users to understand the outcome before they perform reduction.

III. Evaluation

We evaluate the compression latency and compression ratio of various compressors on a SUSE Linux Enterprise Server 11 (x86_64) with a 32-core AMD Opteron(tm) 6410 Processor and 256GB DRAM. Our measurements of compression throughput do not include the time spent on disk I/O since the goal is to evaluate the compression algorithms, instead of system performance as a whole. Our evaluations focus on the following metrics: (1) *Error bound*: It limits the accuracy loss during compression. An error bound can be enforced as an absolute or a relative value or both. Assuming the value of a data point is denoted as V, a point-wise absolute error bound

TABLE I: Dataset description.

Dataset	Description
$Dpot$	Electric potential deviation in a plasma physics simulation.
$Astro$	Velocity magnitude in a supernova simulation.
$Fish$	Velocity magnitude in a CFD calculation of cooling air being injected into a mixing tank.
$Sedov$	Pressure of strong shocks in a hydrodynamical simulation.
$Blast2$	Pressure of strong shocks in a gas-dynamical simulation.
$Eddy$	Velocity in a 2D solution to Navier-Stokes equations.
$YF17\_p$	Pressure in a computational fluid dynamics calculation.
$YF17\_t$	Temperature in a computational fluid dynamics calculation.
$Bump$	Flow density of an axisymmetric bump.

of 10^{-4} means that the decompressed value of the data point is in the range of $[V-10^{-4}, V+10^{-4}]$. A point-wise relative error bound of 10^{-4} means that the decompressed value is in the range of $[V \cdot (1-10^{-4}), V \cdot (1+10^{-4})]$. We adopt the relative error bound so that the results from datasets with different magnitude can be easily compared in this work. (2) *Compression ratio*: It is defined as the ratio of the original dataset size to the compressed data size. (3) *Compression and decompression throughput*: They are the rates at which data can be compressed and decompressed. The evaluations are done under four relative error bound configurations: *cfg1*, *cfg2*, *cfg3*, and *cfg4*, corresponding to a point-wise relative error bound of 10^{-6}, 10^{-5}, 10^{-4}, and 10^{-3}, respectively. Error bound is a crucial factor in impacting compression ratio, and we notice some compressors act more conservatively or aggressively than others to exploit the error tolerance. To ensure a fair comparison, we manually tweak the input error bound to ensure the achieved error bound is used for all results.

A. Data features and compression ratio

Previous studies [13], [15], [23] have demonstrated the compression ratios of a broad set of compressors on various datasets. Particularly, Di et al. [15] evaluated the compression ratios of ZFP, ISABELA, and SZ, and discussed some of the designs tradeoffs. However, the compression ratio is not only compressor related, but also dataset dependent. The complex interplay between the features of a dataset and its compressibility has not been fully understood and exploited for lossy compression. As a result, in reality it is challenging, if not impossible, for users to conjecture the compressibility of their data. Our work aims to fill this gap.

We employ nine real scientific datasets, as described in Table I, to evaluate and understand the interplay between compression ratio and data features, using three state-of-the-art lossy compression schemes, ZFP, ISABELA, and SZ, which were shown to the top 3 lossy compressors [15]. In comparison, this paper also evaluates the performance of two lossless compression schemes: FPC and GZIP. Figure 1 demonstrates the key characteristics of datasets. The *CDF* (Cumulative Distribution Function) curve captures the distribution of values within a particular dataset, illustrating both the range and the skewness of a dataset. We further use *byte entropy*, which is expressed as the number of bits per character, to gauge the information density of dataset content, and byte entropy

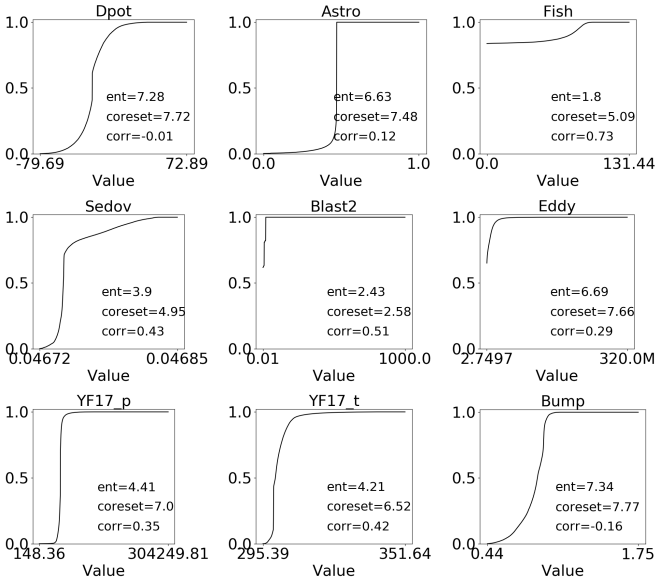


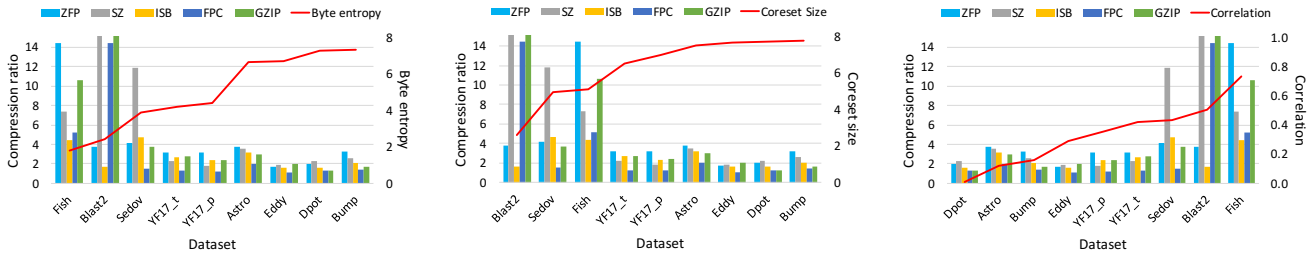
Fig. 1: Data features. The curves show the CDF of data values. *ent*, *coreset*, and *corr* denote the byte entropy, log_2coreset, and serial correlation coefficient, respectively.

ranges in [0, 8]. The higher the information density is, the lower the compressibility is. *Coreset size* is the number of unique symbols (bytes in our case) that compose the majority of a dataset (e.g., 90% of all symbols). A small coreset size means there exists considerable repeated symbols in a dataset, and thus, can be used to estimate the effectiveness of Huffman tree based compression (e.g., GZIP). The range of coreset size is [1, 256], or [0, 8] in logarithm scale. *Serial correlation* measures the extent to which each byte depends upon the previous byte in a dataset. For random data, this value is expected to be close to zero, and for highly correlated data it approaches one. The theoretical value of *serial correlation coefficient* ranges in [-1, 1]. This metric may distinguish random data and data that bear patterns.

Byte entropy, coreset size, and serial correlation coefficient are commonly used to estimate the compressibility of datasets in lossless compression [27]. Therefore, we first investigate whether these metrics continue to be good indicators for lossy compression. Figure 2 shows that in general there is a negative correlation between byte entropy or coreset size and the compression ratio. And there is a positive correlation between the serial correlation coefficient and the compression ratio. These results confirm that data features may be good indicators of compressibility for lossy compression. This motivated us to adopt a sampling based approach to extract data features and estimate compression ratio, without being forced to compress the full data which is costly. Examining data features is particularly beneficial for screening for datasets that are obviously compressible or incompressible.

B. Error bound and compression ratio

Lossy compression has been gaining traction recently, with the potential to achieve an order of magnitude higher performance than lossless compression. Typically as the error



(a) Byte entropy and compression ratio (cfg1). (b) Coreset size and compression ratio (cfg1). (c) Serial correlation and compression ratio (cfg1).

Fig. 2: The relationship between data features and compression ratios for various compressors. The additional evaluation under cfg3 is provided at Data Characteristics and Compression Ratio with cfg3. Note that in these figures we limit the y-axis to 15 to make the short bars discernible. SZ yields a compression ratio of 396.02 with Blast2.

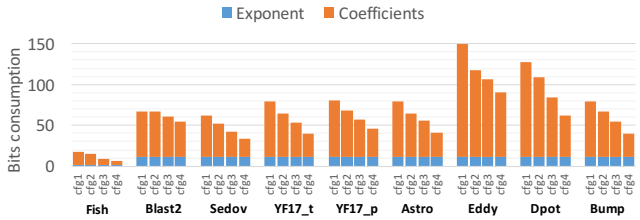


Fig. 3: ZFP compression bit count per block. The original block size is 256 bits, including 4 double-precision floating-point values.

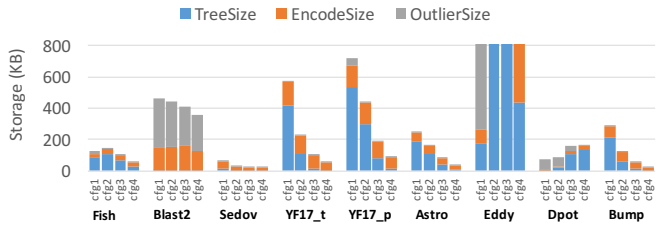
bound is increased, the compression ratio becomes larger. However, if the error bound of the lossy compression is tight, e.g., cfg1, contrary to the common belief, lossy compression may perform worse than lossless compression. For the Eddy dataset, the lossless compressor GZIP achieves a compression ratio of 2.02, while the lossy compressors, ISABELA, ZFP, and SZ, achieve only 1.61, 1.70, and 1.87, respectively. As shown in Figure 2, for 4 out of 9 datasets, GZIP outperforms SZ, and for 2 out of 9 datasets, GZIP outperforms ZFP. For Sedov, YF17_t, YF17_p, and Astro, the compression ratio of GZIP is slightly lower than that of ZFP.

Previous studies show that lossless compression methods rarely achieve more than 1.5x compression ratio [23]. However, our results show that for 7 out of 9 datasets, lossless compressor GZIP can achieve more than 2x compression ratio, with 2 out of 9 datasets achieving more than 4x compression. Although FPC was specifically designed for compressing floating-point data, it is outperformed by the general-purpose GZIP. However, as demonstrated in Figure 6, FPC consistently outperforms GZIP in compression throughputs. In most cases, FPC also achieves better decompression throughputs than GZIP. This confirms the initial design focus of FPC as a high-speed compressor.

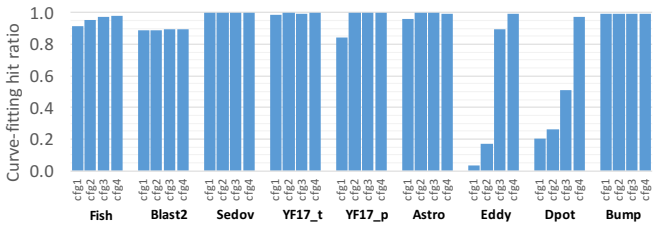
***Finding 1*: When the error bound is tight, lossy compression does not always outperform lossless compression.**

In general, for lossy compression, a larger error bound yields a higher compression ratio. In what follows, we analyze the implementation of ZFP and SZ, and discuss the impact of error bound to the internals of compressors. The output of ZFP consists of two components: the common exponent and the encoding of mantissas. The value of the common exponent of a block is uniquely determined by the values of data points in the block. Therefore, the bits of the common exponent are not affected by the error bound. The mantissas are converted to signed integers and a reversible orthogonal block transformation is performed to generate a set of coefficients, which are further encoded by embedded coding. The error bound determines the number of significant bit planes to be encoded to achieve the target accuracy. A larger error bound always translates a lower data accuracy, resulting in less bit planes to be encoded. As demonstrated in Figure 3, a larger error bound always requires fewer bits to encode the coefficients, leading to a higher compression ratio for ZFP.

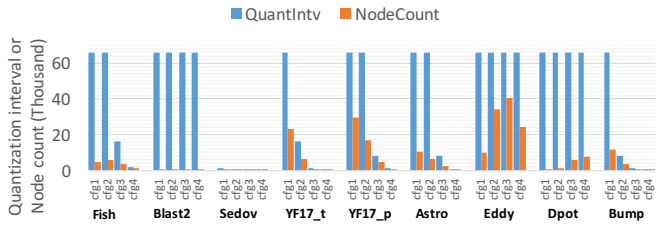
SZ utilizes curve-fitting, scalar quantization, Huffman coding to compress predictable data points, and binary-representation analysis to compress the unpredictable. The compression ratio of SZ on a dataset is multi-factored. First, the curve-fitting hit ratio is a key factor since the curve-fitted data often can be further compressed by one order of magnitude using Huffman coding, while curve-missed data points can only be reduced by about 3x using binary-representation analysis based encoding. Take YF17_t as an example, when the error bound increases from cfg1 to cfg4, the average per curve-missed data point consumes 2.64, 2.20, 1.88, and 1.38 bytes, respectively; meanwhile, per curve-fitted data point consumes 5.91, 2.35, 1.03, and 0.54 bytes, and the corresponding curve-fitting hit ratio is 98.7%, 99.89%, 99.4%, and 99.67%. Therefore, if the curve-fitting hit ratio is low on a dataset, SZ cannot achieve a high compression ratio. Low curve-fitting hit ratio happens to the datasets in which the neighboring values are highly variant. In Figure 1, Eddy and Dpot datasets have high byte entropy and low correlation, both indicating the high variance among neighboring points. In Figure 4(b), Eddy and Dpot datasets have relatively low curve-fitting hit ratios. As a result, as shown in Figure 4(a), the curve-missed outlier points consume large storage space in the compressed data. Note, since Eddy dataset is larger than the others, in Figure 4(a) the Eddy data is limited to 800 KB to make other bars discernible. As a matter of fact, the outlier storage accounts for 84%, 52%, 5.9%, and 0.9% of



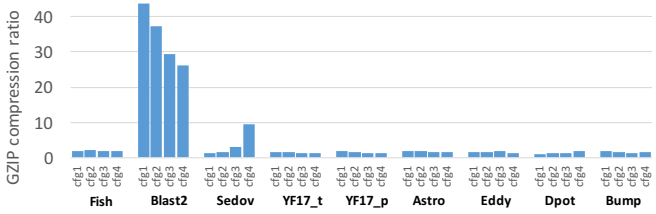
(a) Byte contribution of each encoding component.



(b) SZ curve-fitting hit ratio.



(c) Quantization interval and number of Huffman tree leaf nodes.



(d) GZIP compression ratio on SZ output data.

Fig. 4: SZ compression with various error bounds. We demonstrate the global statistics to illustrate SZ performance.

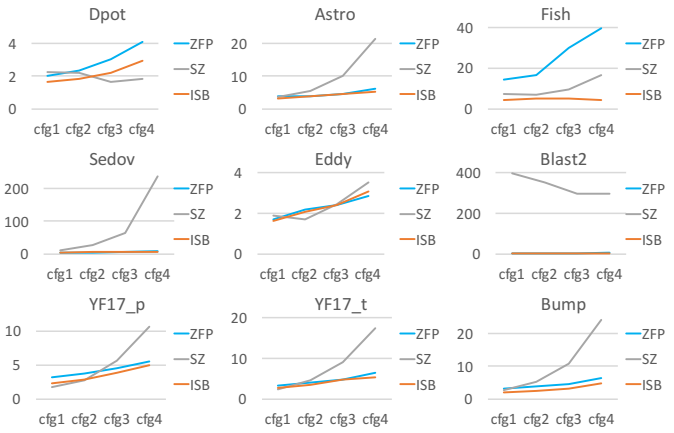


Fig. 5: Compression ratio vs. error bound. Note that x-axis is the error bound, and y-axis is the compression ratio. Due to the space constraint, the y-axis title is removed.

the total compressed Eddy data in cfg1, cfg2, cfg3, and cfg4, respectively.

For SZ compression, a larger error bound mostly yields a higher compression ratio. This is because a larger error bound can narrow the range of quantization factors, resulting in reduced quantization factor slots in the quantization interval. This further requires a smaller number of nodes and lower height of the Huffman tree. Hence the tree size as well as the per quantization factor code length decrease, and fewer bits are used to store the compressed data. Eddy and Dpot datasets are, however, exceptions. When the error bound increases from cfg1 to cfg4, the curve-fitting hit ratio of Eddy and Dpot data increases dramatically. In this case, the total number of quantization factors does not decrease due to the increased number of curve-fitted points. As a result, the compression ratios of these two datasets decrease (Figure 5). We also notice the compression ratio of Blast2 decreases with the error bound, which is mainly caused by the decreased compression ratio of GZIP on SZ compressed data. In Figure 4(d), GZIP achieves a high compression ratio on the SZ-compressed Blast2 data. The reason is that Blast2 data fall into a narrow range, thus, as Figure 4(c) shows, the Huffman tree is very small in size, containing less than 20 nodes. Therefore, on average 25750 curve-fitted data points of Blast2 share the same Huffman coding, causing GZIP to achieve an impressive compression ratio of up to 40.

We observe that as the error bound loosens, the compression ratio of SZ increases faster than that of ISABELA and ZFP. In particular, when the error bound increases from cfg1 to cfg4, the average compression ratio of ISABELA, ZFP and SZ increases by 63%, 86.8%, and 747%, respectively. The reason is, as illustrated in Figure 3, the compression ratio of ZFP is largely determined by the coefficients encoding, which can have a considerable reduction in size when the error bound loosens. However, the size reduction of coefficients is only less than 2x when the error bound increases from cfg1 to cfg4. In contrast, the compression ratio of SZ is mainly determined by the Huffman coding of quantization factors, which can be reduced by more than an order of magnitude when the error bound increases from cfg1 to cfg4. For example, the Huffman tree of YF17_t contains 23166, 6262, 986, and 128 leaf nodes, respectively, and its per curve-fitted points consumes 5.91, 2.35, 1.03, and 0.54 bytes on average. The storage cost per curve-fitted point reduces from 5.91 to 0.54. The reduction is more than one order of magnitude. As shown in Figure 5, the compression ratio of SZ on YF17_t increases from 2.27 to 17.32. In comparison, the compression ratio of ZFP increases from 3.21 to 6.48. When the point-wise relative error bound is loosened to cfg4, SZ can achieve compression ratios of more than 10x for 7 out of 9 datasets, while ZFP can achieve more than 10x for

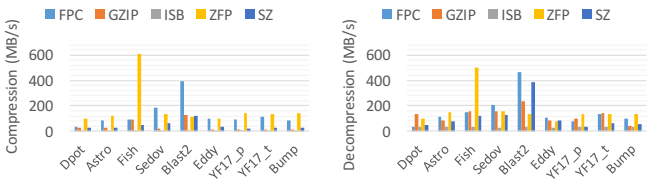


Fig. 6: Compression and decompression throughput (cfg1).

the Fish datasets only.

***Finding 2*: SZ benefits the most from the increasing error bound due to the design of quantization encoding, with a few exceptions where SZ has a lower compression ratio with a higher error bound.**

C. Compression throughput

As shown in Figure 6, for lossless compression, FPC achieves a higher compression throughput than GZIP, and for lossy compression, ZFP outperforms SZ. ISABELA is significantly slower than all other compressors due to the data ordering involved.

FPC sequentially predicts each value and XORs the original value with the prediction, resulting in leading zero bits that are compressible. The FPC compression only needs to go through the data stream once, i.e,, single-pass [10]. Similarly, ZFP examines the data stream once, including encoding the data blocks, calculating the common exponent and mantissas, and conducting the orthogonal block transform on mantissas followed by the embedded coding on the coefficients.

In contrast, SZ requires at least three passes necessitated by the construction of a global Huffman tree. First, the curve-fitting is conducted to determine the predictable and unpredictable data points. For those predictable data points, quantization factors are calculated. Second, quantization factors and their occurrences are used to build a global Huffman tree with the quantization factors as symbols. Finally, quantization factors are encoded with Huffman coding and the unpredictable data points are encoded by binary-representation analysis. In addition to the curve-fitting, ISABELA sorts data in exchange for higher smoothness of data, and this design choice adds additional overhead to compression.

***Finding 3*: Compressors that do not require global data structures run faster than those that do. However, the latter may bring a higher reduction ratio.**

From Figure 2 and 6 jointly, we observe that compression throughput has a positive correlation with compression ratio. Both FPC and GZIP achieve the best compression ratio and throughput on the Blast2 and Fish dataset, and this is also the case for SZ and ZFP. In other words, compressibility is also a good indicator of compression throughput. While throughput is hardware dependent, a quantitative estimation of compression ratio can illustrate the relative throughput performance. This serves as one of the motivations that drive us to model the compression ratio in Section IV.

***Finding 4*: Compression throughput is correlated to compression ratio, implying that more compressible data can be compressed faster.**

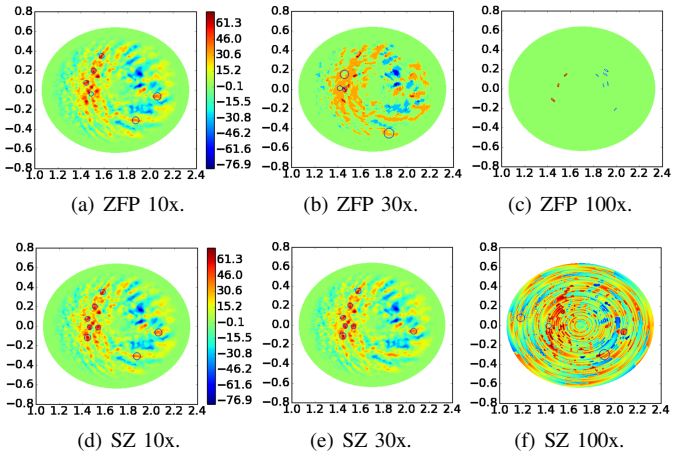
In Figure 6, the decompression of GZIP, ISABELA, and SZ is much faster than the compression, and in what follows, we explain the rationale behind this observation. For GZIP, the compression routine needs to scan the search buffer to match the first symbol in the look-ahead buffer to encode data in tokens. In contrast, the decoding iteratively parses tokens to reconstruct the original data. Without expensive operations such as scanning, the GZIP decompression is much faster than the compression. Similarly, for SZ, the decompression is considerably faster. The reason is that the compression involves the Huffman tree construction and the time complexity of this procedure is non-trivial. In comparison, SZ decompression only involves Huffman tree traversal, which is much cheaper in complexity. The decompression of ISABELA is orders of magnitude faster than its compression, which involves expensive data sorting. FPC and ZFP have similar compression and decompression performances, because they do not involve complicated structures and processing in both encoding and decoding.

***Finding 5*: Decompression is commonly faster than compression. This suggests that data compression is especially useful in the cases where data are compressed once and decompressed frequently, which is typical of HPC workload.**

D. Impact of lossy compression on data analytics

In this section we focus on discussing the impact of lossy compression on data analytics. We demonstrate blob detection on Dpot, which is used by fusion scientists to examine the electrostatic potential in a fusion device and study the trajectory of high energy particles. Herein, we use the blob detection function implemented in OpenCV, an open source computer vision library, to identify areas with high electric potentials in a 2D plane of Dpot data. It uses simple thresholding, grouping, and merging techniques to locate blobs. The blob detection parameters *minThreshold, maxThreshold, minArea, minConvexity* are empirically set as *10, 200, 200, and 0.5*, respectively.

Figure 7 illustrates the blob detection results with varying compression ratios using ZFP and SZ. We evaluated three compression ratios: 10, 30, and 100. The circled areas are identified as high energy blobs. The error bound is tuned to achieve a target compression ratio. As the error bound increases, the features of the blobs captured will change, in terms of both the numbers of blobs and their positions. In Figure 7 most blobs detected in the original data can still be detected with 20x reduction. However, when the compression ratio reaches 30x for ZFP and 100x for SZ, the majority of



(a) ZFP 10x. (b) ZFP 30x. (c) ZFP 100x.

(d) SZ 10x. (e) SZ 30x. (f) SZ 100x.

Fig. 7: Blob detection on the compressed Dpot. We are not able to demonstrate the ISABELA results because it cannot achieve a10x compression ratio. The Y-axis and X-axis are spatial coordinate R and Z, respectively of a fusion device.

blobs are lost, which means the resulting error is too high to be acceptable for blob detection.

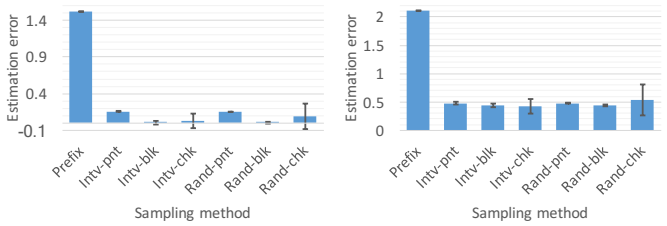
***Finding 6*: Lossy compression may seriously distort data, thus having disastrous impact on data analytics. Determining a proper error bound is key to performing meaningful lossy compression in science production.**

IV. Compression Ratio Estimation

HPC resources are precious and for applications running at scale on HPC systems, it is important to make judicious decisions to use resources efficiently. This is once again the case for data compression. The trial and error approach to finding the best reduction strategy may potentially waste compute cycles and storage resources. Therefore, in this section we develop sampling based analytical models to estimate the reduction performance, with the goal of enabling users to understand the performance without being forced to compress the full data. As such, users can make more informed decisions on questions such as, "will this compressor provide the desired reduction ratio so that my data can fit into the storage?" to avoid wasting compute resources on incompressible data. We present schemes that can accurately estimate the compression ratio of ZFP and SZ, two lossy compressors that achieved the best performance in 12 out of 13 datasets in prior work [15].

A. Methodology

The key idea to estimate compression ratio is to extract the salient properties of data through sampling, based upon which we extrapolate the performance of full data. By and large, *prefix* [28], *interval* [29], and *random* sampling [27] are the common methods to sub-sample data. In particular, prefix sampling selects the front part of data until a pre-defined sampling ratio is satisfied. Interval sampling selects data points at a regular interval, starting from a random offset and then selecting every *k-th* (k is ratio of the full dataset size to the sample size) element. Random sampling selects each element with an equal probability. Interval or random sampling may also choose various sampling granularities: point, block, and chunk, meaning each sample is a single data point, a few (e.g. 4), or a large number (e.g. 32) of sequential data points, respectively. For the convenience of discussion, CR is used to denote the compression ratio, and the estimation error is defined as $EstimationError = \frac{CR_{Sample} - CR_{Full}}{CR_{Full}}$, where CR_{Sample} and CR_{Full} denote the compression ratio of the sample and full dataset, respectively. Note that if $EstimationError < 0$, it indicates that the compression ratio is under-estimated.



(a) ZFP. (b) SZ.

Fig. 8: Estimation accuracy of compression ratio. The sampling ratio is 1%, and the relative error bound is cfg3. The x-axis represents the sampling method, and the y-axis is the average $|EstimationError|$ of nine datasets. *Intv* and *Rand* denote the interval and random sampling, and *pnt*, *blk*, and *chk* denote the sampling granularity of point, block, and chunk, respectively. The error bars are the standard deviation of estimation errors in three runs.

B. Sampling Based Compression Ratio Estimation

In order to extrapolate the performance of the full dataset accurately, the compressor must have bounded locality, first proposed in prior work [27], i.e., the compression of a given data point is either completely independent, or is influenced only by a limited set of neighboring points. A sampling method must preserve the locality of data points. If a data point is sampled, those data points which have influence also need to be sampled. As shown in Figure 8, for ZFP compression, random block based sampling achieves an average estimation accuracy of at least 99% with the standard deviation of about 1%. Interval block based sampling can also achieve an estimation accuracy of 99% with the standard deviation of 2%. Because of the skewness of data distributions, prefix sampling cannot achieve an accurate estimation since it is biased towards the front part of data points, and the average estimation error is up to 150%. We notice that ZFP compresses data in a block of four points in one dimension, and blocks are compressed independently of each other. Therefore, block based sampling for ZFP preserves data locality, resulting in a high accuracy and a low deviation. In comparison, point based sampling breaks the bounded locality, and chunk based sampling, despite that it preserves the locality, the coarse granularity causes a high estimation deviation. As a result, neither point nor chunk based sampling is as accurate as the block based sampling.

***Statement 1**: For ZFP compression, the compression ratio of*

TABLE II: A list of symbols for SZ compression

Symbols	Description
$PointCount$	Number of points in a dataset
$QuantIntv$	Quantization interval
$NodeCount$	Number of Huffman tree nodes
$HitRatio$	Curve-fitting hit ratio
$TreeSize$	Size of Huffman tree
$EncodeSize$	Total size of Huffman coding
$OutlierCount$	Number of curve-missed points
$OutlierSize$	Total size of curve-missed points
$TotalSize$	Size of compressed data
CR	Compression ratio

sample data is an unbiased estimation of the compression ratio of full data, using block based random sampling method.

Proof. The compression ratio of sample data is said to be an unbiased estimation of the full dataset, when the compression ratio of sample data is expected to equal that of the full dataset. As mentioned, ZFP compresses floating-point data in blocks. Assuming the full and sample dataset contain f and s blocks, $BlockSize$ is the original size of a block, and $CR_{Full\_block_i}$ and $CR_{Sample\_block_j}$ indicate the compression ratio of the i^{th} and j^{th} blocks in the full and sample datasets, respectively. After reduction, the size of block i is $RS_{Full\_block_i}$. For the full dataset, the reciprocal of compression ratio is:

$$\frac{1}{CR_{Full}} = \frac{\sum_{i=1}^{f} RS_{Full\_block_i}}{f * BlockSize} = \frac{1}{f}\sum_{i=1}^{f}\frac{1}{CR_{Full\_block_i}}$$

Similarly, for the sample data,

$$\frac{1}{CR_{Sample}} = \frac{1}{s}\sum_{j=1}^{s}\frac{1}{CR_{Sample\_block_j}}$$

As a result of random sampling, for any $block_j$ $(1 \le j \le s)$ in the sample, the probability that this block is $block_i$ $(1 \le i \le f)$ in the full dataset is $\frac{1}{f}$. For any j $(1 \le j \le s)$, the expected reciprocal of compression ratio of $block_j$ in the sample can be calculated as:

$$\mathrm{E}[\frac{1}{CR_{Sample\_block_j}}] = E[\frac{1}{f}\sum_{i=1}^{f}\frac{1}{CR_{Full\_block_i}}]$$

The overall expected reciprocal of compression ratio of the sample can be calculated as:

$$\mathrm{E}[\frac{1}{CR_{Sample}}] = \frac{1}{s}E[\sum_{j=1}^{s}\frac{1}{CR_{Sample\_block_j}}]$$

$$= \frac{1}{s}\sum_{j=1}^{s}E[\frac{1}{f}\sum_{i=1}^{f}\frac{1}{CR_{Full\_block_i}}]$$

$$= \mathrm{E}[\frac{1}{f}\sum_{i=1}^{f}\frac{1}{CR_{Full\_block_i}}] = E[\frac{1}{CR_{Full}}]$$

Therefore, the expected value of CR_{Sample} equals that of CR_{Full}, and the compression ratio of the sample dataset, using random block based sampling, provides an unbiased estimation for the full dataset. □

***Statement 2**: For SZ compression, no sampling method can guarantee an unbiased estimation of the full dataset.*

Proof. SZ adopts the Huffman tree to encode the quantization factors of the curve-fitted points, with the goal of further reducing the storage footprint. The compression ratio of sample data point j $(3 \le j \le s)$ is influenced by point $j-2$ and $j-1$, and the associated Huffman tree constructed based on the sample data. Similarly, the compression of full data point i $(3 \le i \le f)$ is influenced by point $i-2$ and $i-1$, and a different Huffman tree constructed based on the full data. For any given point j in the sample, even if the neighboring points are accordingly sampled to maintain the bounded locality, the two Huffman trees are likely different. Therefore, there does not exist a one-to-one or linear relation between $CR_{Sample\_point_j}$ and CR_{Full}. By Jensen's inequality [30], for a non-linear function f and a mean-unbiased estimator U of a parameter p, the composite estimator $f(U)$ is not a mean-unbiased estimator of $f(p)$. That is, mean-unbiasedness is not preserved under non-linear transformations. □

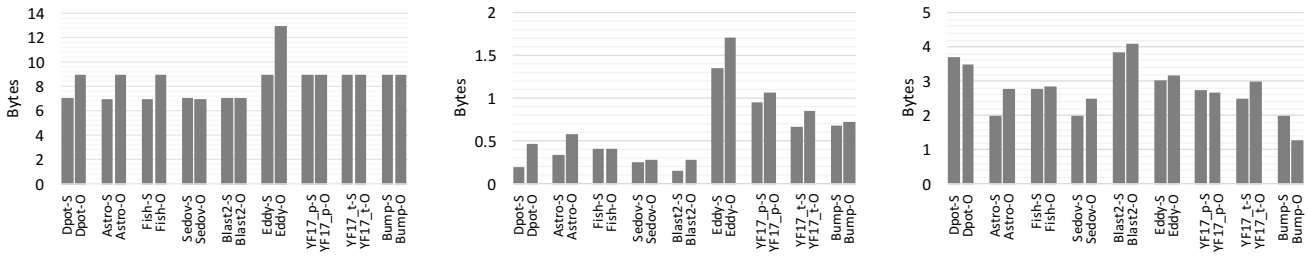
As confirmed in Figure 8(b), using the compression ratio of samples to estimate that of the full dataset is inaccurate for SZ. The average estimation error is at least 42%. Therefore, more advanced models are needed to accurately estimate the SZ compression ratio.

***Finding 7*: For compression schemes which have bounded locality, sampling based approach can provide an unbiased estimation of the full data performance. Without bounded locality, the compression ratio of sample data may deviate significantly from that of the full data.**

C. Advanced Models for Estimating SZ Compression Ratio

In SZ compression, a curve-fitted data point is mapped to a quantization factor, observing a pre-defined error bound, and a curve-missed point will be encoded by binary-representation analysis. Quantization factors may occur with different probabilities, thus, can be encoded using the Huffman tree for storage efficiency. Therefore, the compressed data of SZ consists of three parts: the Huffman tree, the encoding of curve-fitted points, and the encoding of curve-missed points. The binary-representation analysis based encoding for curve-missed points is fairly straightforward to estimate, since our tests show the encoding has approximately resulted in equal size for sample and full data (Figure 9(a)), our model assumes the encoding size is proportional to the number of data points. However, as Figure 9 shows, the storage cost per Huffman tree node, and the code length per curve-fitted data point varies considerably in the sample and full data. Moreover, we notice that the majority of datasets result in a high curve-fitting ratio (Figure 4), and in this case Huffman tree and the encoding will account for most of space after compression.

To predict the storage cost of the Huffman tree and the corresponding Huffman coding, the key is to accurately estimate the number of tree nodes, which determines the tree size and height, which directly impact the average code length. We identify the opportunity to predict the node count of Huffman tree through observing the distribution of quantization factors. Figure 10 demonstrates the distribution in compressing the sample and full data, respectively. It is evident that the distribution of sample and full data are highly similar, both approximately following Gaussian distribution. Given this important observation, this paper employs the Gaussian



(a) Per Huffman tree node cost. (b) Huffman code length per curve-fitted point. (c) Per curve-missed point storage cost.

Fig. 9: A quantitative analysis on the storage cost of SZ encoding. Suffix "-S" and "-O" denote the sample and original dataset, respectively. The sampling ratio in this case is 1%.

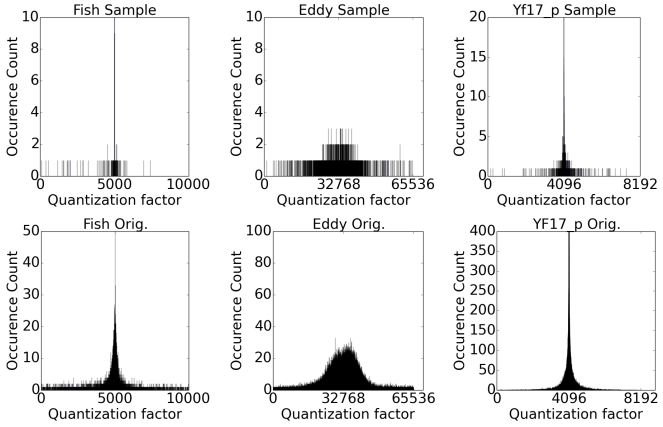


Fig. 10: Distributions of quantization factors produced by the quantization encoder in SZ, with SZ determined quantization intervals.

distribution to model the quantization factors. In summary, the steps of estimating SZ compression ratio of the full data are as follows:

1) Compress the sample data, and collect the compression metrics, as listed in Table II.
2) Estimate the size of curve-missed data points, assuming the sample and full dataset have similar storage cost per curve-missed point (confirmed in Figure 9).
3) Build a Gaussian model based on the quantization factors obtained from compressing the sample.
4) Estimate the node count of the Huffman tree of the full dataset based on the Gaussian model.
5) Estimate the Huffman tree size and encoding size, based on the estimated node count, assuming the height of a Huffman tree is proportional to the logarithm of node count.

The size of compressed data consists of the size of outliers (curve-missed points), Huffman tree, and Huffman encoding of the curve-fitted data points. Namely,

$$TotalSize_{Full} = OutlierSize_{Full} + TreeSize_{Full} + EncodeSize_{Full}$$

We assume per outlier size of the sample and full data are the same, thus

$$OutlierSize_{Full} = \frac{OutlierSize_{Sample}}{OutlierCount_{Sample}} \times OutlierCount_{Full}$$

Similarly the per tree node size of sample and the full dataset are the same, thus

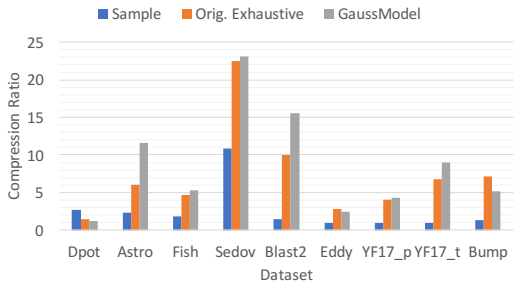


Fig. 11: Estimation of SZ compression ratio using Gaussian distribution to model the quantization factor.

TABLE III: SZ compression ratio estimation.

	Sample 1	Sample 2	Orig.	Estm 1	Estm 2
PointCnt	960	9696	97104	97104	97104
QuantIntv	20000	20000	20000	20000	20000
NodeCnt	430	1865	4723	2758	3951
HitRatio	0.9979	0.9983	0.9991	0.9979	0.9983
TreeSize	7732	33562	85006	49593	71101
EncodeSize	900	10133	104375	118936	111596
OutlierCnt	2	16	78	202	160
OutlierSize	6	44	233	607	441
TotalSize	7688	44407	195762	169136	183138
CR	1.00	1.75	3.97	4.59	4.24
EstmError	74.8%	56%	0	15.7%	6.9%

$$TreeSize_{Full} = \frac{TreeSize_{Sample}}{NodeCount_{Sample}} \times NodeCount_{Full}$$

Since the average length of Huffman coding is proportional to the tree height, which is the logarithm of tree node count, $EncodeSize_{Full} = \frac{EncodeSize_{Sample}}{log_2 NodeCount_{Sample}} \times log_2 NodeCount_{Full}$

Table III demonstrates the effectiveness of the above model in estimating the compression ratio of the Astro dataset. *Sample 1* and *Sample 2* correspond to chunk based sampling with the sampling ratio of 1% and 10%, respectively. *Estm 1* and *Estm 2* show the estimated compression ratio for *Sample 1* and *Sample 2*, respectively, along with all detailed compression metrics. *Orig.* shows the results of directly compressing the original dataset. Overall these results show that simply using the sampling based approach does not work as well for SZ, resulting in high estimation errors of more than 56%. In comparison, the proposed estimation model, termed as GaussModel, can lower the estimation error to 6.9%. Figure 11 further demonstrates the performance of our approach on other datasets. *Orig. Exhaustive* denotes the direct compression of a full dataset. In general, the

estimation error of GaussModel is much lower than the naive sampling based approach. Specifically, GaussModel has an average estimation error of 29%, while the naive sample based estimation has an average error of 73%. We comment that using samples to extrapolate performance inevitably introduces errors, due to the huge information loss in sampling. However, the results show our model combining with sampling can dramatically reduce the estimation error.

V. Conclusion and Future Work

In this paper, we conduct comprehensive evaluations of lossy compression schemes on HPC scientific datasets. We expose how lossy compression schemes work, what data features are indicators of compressibility, the relationship between the compression ratio and the error bound, how the compressibility influences the compression throughput, as well as the impact of lossy compression on data fidelity. We also present how to properly sample datasets for the purpose of making data compression ratio estimation. We prove that random block-based sampling ensures an unbiased compression ratio estimation for ZFP, and the estimation accuracy can be up to 99%. The proposed *GaussModel* dramatically increases the SZ estimation accuracy. This work can help HPC users understand the outcome of lossy compression, which is crucial for the broad adoption of lossy compression to HPC production environments. The new understanding of the relationship between data features and compressibility, as well as accurate compression ratio estimation models are essential to enable the online "to compress or not" decision and the compressor selection. We plan to implement and integrate a decision algorithm into ADIOS (Adaptable IO system) [31], so that scientists can simply describe their requirements and transparently use the *best* compression scheme.

Acknowledgment

The work is partially sponsored by U.S. National Science Foundation grants CCF-1718297, CCF-1717660, CNS-1702474, and DOE SIRIUS project.

References

[1] T. Lu and et al., "Canopus: A paradigm shift towards elastic extreme-scale data analytics on hpc storage," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 58–69.

[2] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *IEEE International Parallel and Distributed Processing Symposium*, 2017.

[3] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 912–922.

[4] D. Ghoshal and L. Ramakrishnan, "Madats: Managing data on tiered storage for scientific workflows," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017, pp. 41–52.

[5] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu, "Arrayudf: User-defined scientific data analysis on arrays," in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2017.

[6] ASCAC Subcommittee, "Top ten exascale research challenges," 2014. [Online]. Available: https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf

[7] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, 2016.

[8] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 1–11.

[9] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, July 1948.

[10] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2009.

[11] I. Foster and et al., "Computing just what you need: Online data analysis and reduction at extreme scales," in *European Conference on Parallel Processing (Euro-Par'17)*, Santiago de Compostela, Spain, 2017.

[12] G. K. Wallace, "The jpeg still picture compression standard," *IEEE transactions on consumer electronics*, vol. 38, no. 1, pp. xviii–xxxiv, 1992.

[13] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. Samatova, "Compressing the incompressible with isabela: In-situ reduction of spatio-temporal data," *Euro-Par 2011 Parallel Processing*, pp. 366–379, 2011.

[14] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[15] S. Di and F. Cappello, "Fast error-bounded lossy hpc data compression with sz," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 730–739.

[16] J.-l. Gailly, "gzip: The data compression program," 2016. [Online]. Available: https://www.gnu.org/software/gzip/manual/gzip.pdf

[17] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, Jan 2009.

[18] A. Lempel and J. Ziv, "On the complexity of finite sequences," *IEEE Transactions on information theory*, vol. 22, no. 1, pp. 75–81, 1976.

[19] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[20] Y. Sazeides and J. E. Smith, "The predictability of data values," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1997, pp. 248–258.

[21] B. Goeman, H. Vandierendonck, and K. De Bosschere, "Differential fcm: Increasing value prediction accuracy by improving table usage efficiency," in *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. IEEE, 2001, pp. 207–216.

[22] N. Sasaki and et al., "Exploration of lossy compression for application-level checkpoint/restart," in *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*. IEEE, 2015, pp. 914–922.

[23] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[24] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Transactions on signal processing*, vol. 41, no. 12, pp. 3445–3462, 1993.

[25] S. Wold, "Spline functions in data analysis," *Technometrics*, vol. 16, no. 1, pp. 1–11, 1974.

[26] X. He and P. Shi, "Monotone b-spline smoothing," *Journal of the American statistical Association*, vol. 93, no. 442, pp. 643–650, 1998.

[27] D. Harnik, R. I. Kat, O. Margalit, D. Sotnikov, and A. Traeger, "To zip or not to zip: effective resource usage for real-time compression." in *FAST*, 2013, pp. 229–242.

[28] J. Zhou, Y. Li, V. K. Adhikari, and Z.-L. Zhang, "Counting youtube videos via random prefix sampling," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11. New York, NY, USA: ACM, 2011, pp. 371–380.

[29] O. Rana and et al., "Paddmas: parallel and distributed data mining application suite," in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000, pp. 387–392.

[30] J. L. W. V. Jensen, "Sur les fonctions convexes et les inégalités entre les valeurs moyennes," *Acta mathematica*, vol. 30, no. 1, pp. 175–193, 1906.

[31] Q. Liu and et al., "Hello adios: The challenges and lessons of developing leadership class i/o frameworks," *Concurr. Comput. : Pract. Exper.*, vol. 26, no. 7, pp. 1453–1473, May 2014.