Fast Error-bounded Lossy HPC Data Compression with SZ

Sheng Di[1], Franck Cappello[12]
[1]Argonne National Laboratory, USA,
[2]University of Illinois at Urbana-Champaign, USA
sdi1@anl.gov, cappello@anl.gov

***Abstract*—Today's HPC applications are producing extremely large amounts of data, thus it is necessary to use an efficient compression before storing them to parallel file systems. In this paper, we optimize the error-bounded HPC data compression, by proposing a novel HPC data compression method that works very effectively on compressing large-scale HPC data sets. The compression method starts by linearizing multi-dimensional snapshot data. The key idea is to fit/predict the successive data points with the bestfit selection of curve fitting models. The data that can be predicted precisely will be replaced by the code of the corresponding curve-fitting model. As for the unpredictable data that cannot be approximated by curve-fitting models, we perform an optimized lossy compression via a binary representation analysis. We evaluate our proposed solution using 13 real-world HPC applications across different scientific domains, and compare it to many other state-of-the-art compression methods (including Gzip, FPC, ISABELA, NUMARCK, ZFP, FPZIP, etc.). Experiments show that the compression ratio of our compressor ranges in 3.3/1 - 436/1, which is higher than the second-best solution ZFP by as little as 2x and as much as an order of magnitude for most cases. The compression time of SZ is comparable to other solutions', while its decompression time is less than the second best one by 50%-90%. On an extreme-scale use case, experiments show that the compression ratio of SZ exceeds that of ZFP by 80%.**

I. Introduction

With increasing scales of scientific simulations, today's HPC applications are producing extremely large amounts of data (in the order of terabytes or even petabytes) during the execution, such that data processing has become a significant bottleneck for extreme-scale HPC applications. As indicated in [1], the CESM climate simulation [2] is able to produce terabytes of data per day for post-processing. As reported in [3], XGC1 simulation can launch a series of simulations with up to 100 PB of data to write out on Titan system. Such a large amounts of data are usually stored in a parallel file system (PFS) such as Lustre [4] for convenience of management. However, the storage bandwidth becomes a serious bottleneck compared to other types of resources (such as high-speed processors, memories, and caches).

In order to save disk space and improve runtime performance as well as post-processing efficiency for exa-scale HPC applications, it is necessary to significantly reduce the size of the data to be dumped during the execution with fairly low computation cost and required compression error bounds. Although lossless compression such as Gzip [5] can guarantee no compression errors, it suffers from low compression ratio when dealing with extremely large amount of dynamic HPC data. To this end, error-bounded lossy compression methods are considered an acceptable trade-off solution, as indicated by recent studies [1], [6] based on production scientific simulations.

It is fairly challenging to design a generic error-bounded lossy compressor with a very high compression ratio for HPC applications. The key reason is that HPC snapshot data have diverse features (such as various dimensions, different scales, and data dynamics in both space and time) with different applications. The data variables used in an application, for example, are usually of high dimensions (such as 5 dimensions) and the data values may change largely in a short period. To this end, existing studies usually ignored the locality of data points in the snapshot. ISABELA [7], for instance, converted the multi-dimensional snapshot data to a sequence of *sorted* data before performing the compression by B-spline interpolation. Due to the loss of the data location information in the sorted data-series, ISABELA has to use an extra index array to record the original index of each point, suffering from low compression ratio especially for a large number of data points.

In this work, we propose a novel HPC data compression scheme (namely Squeeze or SZ for short) with strictly bounded errors and low overheads.

There are three significant novelties/contributions to be addressed in this paper, as summarized below:

- *Fast Data Encoding with Bestfit Curve-fitting Model*. To the best of our knowledge, this is the first attempt to leverage multiple curve-fitting models to encode the data stream. This is inspired by the fact that the adjacent data in the converted data-series are likely adjacent in the original multi-dimensional snapshot, such that the generated data-series may exhibit significant exploitable smoothness. The key step is to check the sequence of data point-by-point, verifying whether they can be predicted within the user-required error bounds by the bestfit curve-fitting models (such as linear curve fitting and quadratic curve fitting). Each predictable data will be replaced by a two-bit code that denotes the corresponding bestfit curve-fitting model. The time complexity of this process is only O(N).
- *Effective Error-bounded Lossy Compression for Un-*

predictable Data. We optimize (also with O(N) time complexity) the error-bounded lossy compression for compressing the unpredictable data, by elaborately analyzing the binary data representation based on desired precision and data value ranges.

- We implement the compressor rigorously and it is available to download from [13] under BSD license, supporting C, Fortran, and Java. We evaluate it by running 13 real-world HPC applications across different scientific domains on the Argonne FUSION cluster [8]. We compare it to many other state-of-the-art compression methods (including Gzip [5], FPC [9], ISABELA [7], NUMARCK [10], ZFP [11], Sasaki et al.'s work [6], and FPZIP [12]). To the best of our knowledge, our evaluation covers the most compression methods and the most types of scientific applications in comparison with other existing compression research. Experiments show that the compression ratio of our error-bounded compressor are in the range 3.3:1-436:1, which is higher than the second best solution twice or even by an order of magnitude[1].

The rest of the paper is organized as follows. In Section II, we formulate the HPC data compression issue. In Section III, we propose a novel HPC data compression approach, based on bestfit curve-fitting models. We describe our optimization strategies on how to compress the unpredictable data in Section III-C. In Section IV, we present the experimental setup and evaluation results with different compression methods. We analyze the related work in Section V, and conclude with a presentation of the future work in Section VI.

II. Problem Formulation

In this paper, we focus on the design and implementation of a data compression method for HPC applications with guaranteed error bounds. In particular, we investigate how to compress every application snapshot to be used for data analytic and visualization. In general, during the execution, the application generates multiple snapshots, each containing multiple variables with specific data types such as multi-dimensional floating-point array, string data and constant values. Since the large majority of disk space for a snapshot is occupied by the floating-point array data, we will mainly focus on how to compress such a portion of data with user-specified compression error bound.

In our work, the compression is controlled using an error bound (denoted by Δ), which is typically specified with user demand. Specifically, the compression errors (defined as the difference between the data points' original values and their corresponding decompressed values) of all data points must be strictly limited in an error bound, i.e., D_i' must be in $[D_i-\Delta,D_i+\Delta]$, where D_i' and D_i refer to a decompressed value and the corresponding original value respectively.

[1]SZ always exhibits the highest compression ratio, except for one case in which the compression ratio of SZ is still very close to the best one.

There are two types of error bounds, *absolute error bound* and *relative error bound*, which are both widely used in HPC data compression. The absolute error bound (denoted δ) will be set to a constant. Such a bound has been widely used to evaluate the HPC data compression such as [10], [7], [11]. As for the relative error bound, it is a linear function of the global data value range size, i.e., $\Delta=\lambda r$, where $\lambda(\in(0,1))$ and r refer to *error bound ratio* and range size respectively. For example, given a set of data (D_1, D_2, $\cdots$, D_M), the range size r is equal to $\max_{i=1\ldots M}(D_i)-\min_{i=1\ldots M}(D_i)$, so relative error bound can be written as $\lambda(\max_{i=1\ldots M}(D_i)-\min_{i=1\ldots M}(D_i))$. The relative error bound allows to make sure that the compression error for any data point must be no greater than $\lambda\times100$ percentage of the global data value range size. Such a bound was also used to evaluate some data compression methods such as [6]. Both of the above two types of bounds are provided in our compressor for suiting different user demands. If the data range size r changes largely over time, the compression errors have to be limited by taking value range into account, in that the evolved data are to be plotted based on value range.

The *compression ratio* (denoted by ρ) is defined as the ratio of the original total data size to the compressed data size. Suppose the original data size S_o is reduced to S_c after the compression, then the compression ratio can be written as $\rho=S_o/S_c$. With the same compression error bound, higher compression ratio with lower compression/decompression time implies better results.

The objective of this research is to design an efficient data compression approach that can compress run-time HPC snapshots, such that the compression errors are bounded within absolute error bound, relative error bound, or both of them, based on user's demand.

The applications evaluated in this paper belong to seven different scientific domains, including hybrodynamics (HD), magneto hydrodynamics (MHD), gravity study (GRAV), particles simulation (PAR), shock simulation (SH), diffusion simulation (DIFF), and climate simulation (CLI), across from three HPC code packages, including FLASH [14], Nek5000 [20], and CESM [2], as shown in Table I.

III. Bestfit Curve-fitting Model based HPC Data Compression

In this section, we present a novel effective HPC data compression method, by mainly focusing on how to compress multi-dimensional floating-point arrays. Given a d-dimensional floating-point array, the overall compression procedure can be split into three steps: (1) convert the d-dimensional floating-point array to a 1-dimensional array, (2) compress the 1-D array by using a bit-array to record the data points whether they can be predicted by the dynamic bestfit curve-fitting models, (3) compress the unpredictable data by analyzing their binary representations. We describe the three steps in the following text.

Table I
APPLICATIONS USED IN THIS WORK

Domain	Name	Code	Description
HD	Blast2 [15]	Flash	Strong shocks and narrow features
	Sedov [16]	Flash	Hydrodynamical test code involving strong shocks and non-planar symmetry
	BlastBS [17]	Flash	3D version of the MHD spherical blast wave problem
	Eddy [19]	Nek5k	2D solution to Navier-Stokes equations with an additional translational velocity
	Vortex [20]	Nek5k	Inviscid Vortex Propagation: tests the problem in earlier studies of finite volume methods
MHD	BrioWu [18]	Flash	Coplanar magneto-hydrodynamic counterpart of hydrodynamic Sod problem
	GALLEX [21]	Nek5k	Simulation of gallium experiment (a radiochemical neutrino detection experiment)
GRAV	MacLaurin [14]	Flash	MacLaurin spheroid (gravitational potential at the surface/inside a spheroid)
PAR	Orbit [14]	Flash	testing the mapping of particle positions to gridded density fields, mapping of gridded potentials onto particle positions to obtain particle forces, and time integration of particle motion
SH	ShafranovShock [22]	Flash	a problem that provides a good verification for structure of 1D shock waves in a two-temperature plasma with separate ion and electron temperatures
DIFF	ConductionDelta [14]	Flash	Delta-function heat conduction problem: examining the effects of Viscosity
CLI	CICE [23]	CESM	Community sea-ice simulation based on Community Earth System Model
	ATM [2]	CESM	CAM-SE cubed sphere atmosphere simulation with very large data size (1.5TB) produced

A. Converting Multi-dimensional Array to 1-D Array

In the beginning, the multi-dimensional array needs to be converted to a 1-D array (a.k.a., linearization). There are many ways to linearize the multi-dimensional data, such as leveraging space-filling curve methods [28] that can keep the data locality information to a certain extent. The most well-known space-filling curves include Peano curve [31], Moore curve [29], Hilbert curve [30], and Lebesque curve (or Z-order curve) [32].

Based on our experiments, we observe that the computation cost in constructing the simplest space-filling curve is still very significant compared to other portion of time cost in the data compression, especially when the data size to process is very huge. In particular, our experiments show that even though we adopt the simplest Z-order scanning strategy to linearize the multi-dimensional data, the time cost is twice longer than that of the compression without the linearization step because of many costly multiplication operations. As such, we propose to use the intrinsic memory sequence of the data array to construct the 1-D data sequence for compression. The key advantages are two-fold:

- *extremely low conversion cost*: We just need to copy the address of the multi-dimensional array, which can save a lot of time especially when the data size is extremely huge as projected for exascale execution.
- *good locality preservation*: It is worth noting that the data compression could be performed by extremely large number of ranks in parallel, hence the whole data set is actually split into a large number of small tiles to compress, such that the locality of data can be preserved very well.

B. Compressing 1-D Array by Bestfit Curve-Fitting Models

In what follows, we discuss how to compress the 1-D array $\{V_1,V_2,\cdots,V_M\}$. The basic idea is checking each data point in the 1-D array one by one, to see if it can be predicted (within user-required error bounds) based on a few of its preceding values by some curve-fitting model (such as linear-curve or quadratic curve). If yes, we will record the corresponding curve-fitting model for that point in a bit-array. The data that cannot be predicted are called *unpredictable data* and they will be compressed by analyzing the IEEE 754 binary representation before being stored separately (to be discussed in next subsection).

For the data prediction, we adopt three curve-fitting models, *preceding neighbor fitting*, *linear-curve fitting*, and *quadratic-curve fitting*, which are described as follows:

- *Preceding Neighbor Fitting (PNF)*: This is the simplest prediction model, which just uses the preceding value to fit the current value. Suppose the current value is V_i, then its predicted value (denoted by $X_i^{(N)}$) will be estimated as $X_i^{(N)}$=X_{i-1}. Note that the preceding data used in the decompression are not original values, so the PNF prediction here is supposed to be X_{i-1} instead of V_{i-1}. More details will be discussed later.
- *Linear-Curve Fitting (LCF)*: This fitting model assumes that the current value V_i can be estimated by the linear line constructed using its previous two consecutive values. Specifically, the predicted value $X_i^{(L)}$ is derived as $X_i^{(L)}$=X_{i-1}+($X_{i-1}-X_{i-2}$)=$2X_{i-1}-X_{i-2}$.
- *Quadratic-Curve Fitting (QCF)*: Quadratic-curve fitting model assumes that the current value V_i can be predicted precisely by a quadratic curve that is constructed by the previous three consecutive values. Specifically, a quadratic curve (denoted by $f(x) = ax^2 + bx + c$) can be denoted as (0, $X(i-3)$), (1, $X(i-2)$), and (2, $X(i-1)$), respectively. Then, the predicted value at i can be computed by $f(3)$=$9a + 3b + c$, where a, b, and c are computed by the three preceding points (0, $X(i-3)$), (1, $X(i-2)$), and (2, $X(i-1)$). Hence, the predicted value $X_i^{(Q)}$ can be derived as $X_i^{(Q)}$=$f(3)$=$3X_{i-1}-3X_{i-2}+X_{i-3}$.

We give an example to further illustrate the above three fitting models, as shown in Figure 1. In the figure, three predicted values for the current data value V_i are denoted by the black cross, blue cross and red cross respectively. They are all predicted by the previous consecutive decompressed value(s), which are either predicted values generated in the compression or the unpredictable values stored separately. Note that it is critical that one should not directly use

original preceding data values $\{V_{i-3},V_{i-2},V_{i-1}\}$ to perform the prediction for the data value V_i, since the preceding data that are to be used in the decompression are not the original preceding data values but the decompressed values with a certain errors. Such a design guarantees the decompressed value X_i to meet user-required error bounds.

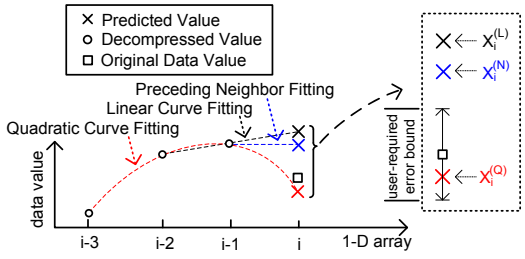


Figure 1. Illustration of Fitting Models

The pseudo-code of compressing the 1-D array by bestfit curve-fitting models is presented in Algorithm 1. In the beginning, the program allocates $2M$ bits of memory for recording the predictability of each value as well as the bestfit fitting prediction model, where M is the number of data points. And then, the algorithm computes the value range size r (i.e., $\max(V_i)-\min(V_i)$), which will be used for comparing the prediction errors to the error bounds. If the compression is performed by each rank in parallel, the $\max(V_i)$ and $\min(V_i)$ can be aggregated by the collective operation *MPI_Allreduce*. In fact, such a collective operation has to be performed anyway if the user demands the relative error bound. Such a collective operation, however, can be performed periodically for reducing the communication overheads in that the data value range usually does not change largely in short periods. Moreover, users are also allowed to specify the compression precision using only absolute error bound for completely avoiding the communication overheads. The algorithm's remaining procedure (line 3-24) checks each value in the 1-D array one by one. Specifically, it will first determine the bestfit curve-fitting model (line 4-7), and check if its prediction error is within the user-required error bounds. As for the example illustrated in Figure 1, the bestfit model is QCF. As mentioned previously, there are two types of error bounds, and users are allowed to set either of them or both of them on demand (to be discussed in more details later). If the prediction error meets the user-required error bounds, the corresponding bestfit curve-fitting model will be recorded via a bit-array υ (line 9-17), otherwise, the current data value will be compressed by a binary-representation analysis (to be discussed in next subsection) and stored approximately in a separate array (denoted by ρ) (line 19-22).

Remarks:

- At most three preceding consecutive values (X_{i-3}, X_{i-2}, X_{i-1}) are required for checking the predictability of the value V_i, hence, the algorithm only needs to keep three extra preceding decompressed values at runtime instead of all of the decompressed values, which means a very low memory overhead. Suppose there are N data points to compress, the total memory overhead is only $\frac{2N+64M}{64N}=\frac{1}{32}+\frac{M}{N}$ of the original memory size, where M refers to the amount of unpredictable data.
- The time complexity of the algorithm is O(N), where N here refers to the amount of floating-point data. Moreover, the major part of the algorithm involves only bitwise operations (such as line 9-16 and line 19-20), so the processing speed is supposed to be very fast.
- As defined in Section II, the user-required precision in our compressor is confined based on two types of error bounds, namely absolute error bound (δ) and relative bound bound (λr), where r is the data value range size. There are four modes/options for users to choose: (1) using only absolute error bound (i.e., each data point's compression error $\epsilon_i=D_i'-D_i\leq\delta$), (2) using only relative error bound ($\epsilon_i=D_i'-D_i\leq\lambda r$), (3) the minimum value of the two bounds (i.e., satisfying $\epsilon_i\leq\delta$ and $\epsilon_i\leq\lambda r$), and (4) the maximum value of the two bounds (i.e., satisfying $\epsilon_i\leq\delta$ or $\epsilon_i\leq\lambda r$).
- The decompression is just a reverse procedure of the above compression algorithm. Specifically, it first parses the bit-array υ to retrieve the predictability and bestfit model information. If the current value is predictable, it will be recovered by the corresponding curve-fitting model, or else, it can be found in a separate data array ρ and it will be recovered by the

Algorithm 1 BESTFIT CURVE-FITTING COMPRESSION

Input: 1-D array data $\{V_1, V_2, \cdots, V_M\}$, user-required error bound information (error bound mode, absolute error bound δ, and/or relative error bound λ)
Output: Compressed byte data $\{\upsilon,\rho\}$, where υ and ρ are used to record the values' predictability and store the unpredictable data respectively.

1: Create a bit-array (denoted υ) with M elements, each occupying 2 bits.
2: Compute the value range size r (= $\max(V_i)-\min(V_i)$).
3: **for** (V_i, i=1,2,$\cdots$,M) **do**
4: $X_i^{(N)} \leftarrow X_{i-1}$ /*Preceding Neighbor Fitting*/
5: $X_i^{(L)} \leftarrow 2X_{i-1}-X_{i-2}$ /*Linear-Curve Fitting*/
6: $X_i^{(Q)} \leftarrow 3X_{i-1}-3X_{i-2}+X_{i-3}$ /*Quadratic-Curve Fitting*/
7: *bestfit_sol* = $\underset{Y=\{(N),(L),(Q)\}}{\arg\min}(|X_i^Y-V_i|)$.
8: **if** ($|X_i^{bestfit\_sol}-V_i|$ meets error bounds) **then**
9: **switch** (*bestfit_sol*)
10: **case** (N)**:**
11: $\upsilon[i] = 01_{(2)}$ /*denote Preceding Neighbor Fitting*/
12: **case** (L)**:**
13: $\upsilon[i] = 10_{(2)}$ /*denote Linear-Curve Fitting*/
14: **case** (Q)**:**
15: $\upsilon[i] = 11_{(2)}$ /*denote Quadratic-Curve Fitting*/
16: **end switch**
17: $X_i \leftarrow X_i^{bestfit\_sol}$ /*record the predicted value for next prediction*/
18: **else**
19: $\upsilon[i] = 00_{(2)}$ /*denote unpredictable data*/
20: Compress V_i by binary-representation analysis.
21: $\rho[j{+}{+}] \leftarrow V_i'$ /*V_i' is the binary decompressed value of V_i*/
22: $X_i \leftarrow V_i'$
23: **end if**
24: **end for**

binary-representation analysis. It is worth noting that the compression algorithm predicts each value by using the preceding *decompressed* values instead of original values, such that the decompression errors can always be guaranteed within user-required bounds.

- Unlike the compression algorithm, the decompression algorithm does not require users to provide the error bound information as the input, because in the compression step, such information can be stored in the compressed stream together with the compressed data.
- One important feature of the above algorithm is that the bit-array υ may exhibit a high consistency on most of the consecutive bits, which can be confirmed by Figure 2. This figure shows bestfit curve-fitting model types ($00_{(2)}$, $01_{(2)}$, $10_{(2)}$, or $11_{(2)}$) determined by the above algorithm for various applications (variable:*density*, time step:100). Such a consistent bit value feature will lead to a very high compression ratio when using Gzip [5] to further compress the bit-arrays thanks to the LZ77 algorithm [24] used by Gzip. Specifically, LZ77 algorithm searches the same repeated sequences of the data and replace them with references to only one single copy existing earlier in the data stream.

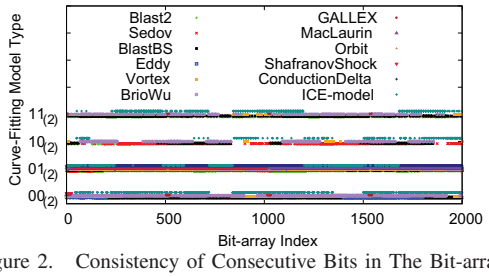


Figure 2. Consistency of Consecutive Bits in The Bit-array ρ

C. Optimization of Lossy Compression for Unpredictable Data by Binary Representation Analysis

In this subsection, we optimize the lossy compression method by analyzing the IEEE 754 binary representation of the unpredictable data, as well as its relationship to the user-required error bounds and the value range size. Specifically, we derive the smallest number of mantissa bits required based on the user-specified error bounds and data value range, such that the insignificant bits can be truncated.

There are three key steps to process.

First, we map all of the unpredictable data to a smaller range by letting all the values minus the median value of the range (denoted by *med*, i.e., *med*=$(\min_i(\rho[i]) + \max_i(\rho[i]))/2$). The newly generated data after the transformation are called *normalized data*, which will be closer to zero than the original ones. Such a step is motivated by the fact that the closer the number is to zero, the less mantissa bits are required to meet the specified precision. We further illustrate it in Figure 3, by comparing the binary representations of two numbers, 100.001 and 0.001. We can see that these two numbers are stored in binary format as $1.10010000...{}_{(2)}\times 2^{6}$ and $1.00001100...{}_{(2)}\times 2^{-10}$ respectively. If the user-required absolute error bound is set to 1.22×10^{-4}(i.e., 2^{-12}), then the leading mantissa bits for the two numbers are supposed to be no less than $6-(-12)$=18 bits and $-10-(-12)$=2 bits respectively. Obviously, the latter requires much less bits to meet the required precision.

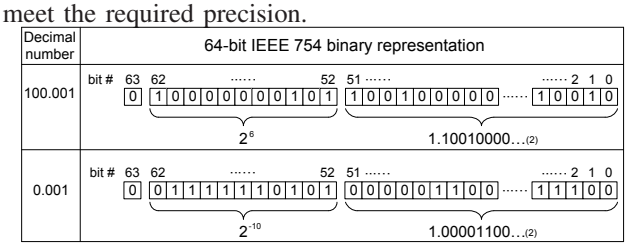


Figure 3. Illustration that a number closer to zero requires less bits

Second, we truncate the value by disregarding the insignificant mantissa part based on the user-required error bounds. For instance, suppose the absolute error bound δ and relative error bound λ provided by the user are both required to be met, then, we can use $\min\{\delta, \lambda r\}$ to serve as an integrated error bound (denoted by Δ), where r is the value range size. Similarly, if the user just requires that either of the two types of error bounds are met, the integrated error bound Δ can be represented as $\max\{\delta, \lambda r\}$. As such, we just need to focus on the integrated error bound Δ in the following analysis. Given any one normalized data value (denoted by v_i, i.e., v_i=$\rho[i]-$*med*), the minimum number of the required mantissa bits (denoted by *RQ_MBits*(v_i)) can be computed by Equation (1):

$$RQ\_MBits = \begin{cases} 0, & Exp(radius) - Exp(\Delta) < 0 \\ MLen, & Exp(radius) - Exp(\Delta) > MLen \\ Exp(radius) - Exp(\Delta), & otherwise \end{cases} \quad (1)$$

where *radius* refers to the radius of the value range (i.e., $\max_i(v_i)$ which is equal to $\frac{\max_i(\rho[i]) - \min_i(\rho[i])}{2}$), *Exp*$(x)$ refers to the value of the exponent binary part of the floating-point number x, and *MLen* is the length of the mantissa (=23 for single-precision data and =52 for double-precision data). There are two points which are worth being noted, as addressed below.

- We use *Exp*(*radius*) instead of *Exp*(v_i) to compute the required mantissa bits. The reason is that although using *Exp*(v_i) can obtain more accurate minimum number of required mantissa bits, the final sequence of the compressed bits for different normalized values (v_i) will likely have different lengths, such that we cannot recover the normalized values based on the mixed sequence of bits. As such, we fix the number of bits required by using *Exp*(*radius*) to compute the required mantissa bits for every normalized data value.
- In our implementation, *byte* is the smallest operation unit (the reason will be described later), so the real number of maintained bits (a multiple of 8) can be computed by the following equation, where $\lceil\cdot\rceil$ is the

ceiling function.

$$REAL\_RQ\_Bits = \left\lceil \frac{1+Exp(v_i)+RQ\_MBits}{8} \right\rceil \cdot 8 \quad (2)$$

At the last step, we perform the leading-zero based floating-point compression method to further reduce the storage size. In particular, we perform the XOR operation for the consecutive normalized values and compress each by using a leading zero count followed by the remaining significant bits. This may significantly reduce the storage size due to the fact that the leading bits of the IEEE 754 binary representations for the similar values are expected to be very similar to each other. In order to limit the number of bits required to store the leading zero count, *byte* serves as the operation unit. As such, we need only 2 bits to record the number of leading-zero bits, which can cover at most 3×8=24 bits. This is fairly enough for vast majority of the data based on our experience. Accordingly, when compressing $\rho[i]$, the first three bytes of its binary representation are compared to that of its preceding value $\rho[i-1]$. The number of leading-zero bytes will be stored using 2 bits if the leading-zero length is no less 1 byte (8 bits), which will thus reduce the storage size.

IV. Performance Evaluation

We first describe the experimental setup used in the evaluation and then present the evaluation results by comparing our solution to 7 other state-of-the-art compression methods.

A. Experimental Setup

In our experiments, we carefully compare our approach to as many state-of-the-art compression methods as possible, including lossless compressors such as Gzip and FPC, and lossy compressors such as ZFP, ISABELA, NUMARCK, and Sasaki et al.'s approach (namely SSEM in our evaluation based on the authors' last names). Gzip, FPC, ZFP and ISABELA are all off-the-shelf compressors that can be downloaded for free. We implemented NUMARCK rigorously and confirmed the correctness of the implementation by comparing the compression results to that of the published paper [10]. We also improve NUMARCK to get a higher compression ratio than its original version. Specifically, the original NUMARCK has to store the data values exactly when the data values are 0 or all of data points in a snapshot have the same values, while our improved NUMARCK will store only one data value and the number of data points instead. We also optimized the number of clusters split in the Kmeans clustering for NUMARCK. In addition, NUMARCK requires to periodically save the original non-compressed snapshots. The longer periodic intervals, the higher compression ratios yet the larger compression errors. In our evaluation, the interval is set to 5 because this setting leads to relatively small compression errors. We implemented SSEM strictly based on the published paper [6], and also further devised an error-bounded version for SSEM, namely SSEM(eb) in the following evaluation. The difference between SSEM(eb) and SSEM is that the former compares the decompressed data value to the original value during the compression and store original data if the deviation is beyond the error bound. All experiments are conducted on the Argonne FUSION cluster server, which has 16 cores and 64 GB memory.

We carefully evaluate the compression ratios and compression error for all of the 8 compression methods, by using the real-world HPC applications. For the first 12 applications (listed in Table I), each is run through 1000 iterations (time steps), so there are 1000 snapshots generated after each run. Every snapshot for FLASH application has 10 variables with 82k-655k data points in total, which is comparative to the data size used in the evaluation of other related research such as ISABELA [7] and NUMARCK [10]. The last application ATM (as shown in Table I) is a very good case to evaluate the compressor's ability in dealing with extremely large data set, because its data size is about 1.5TB in total (63 snapshots each of which is about 24GB in data size).

We use different compressors to compress and decompress each snapshot for each application, and compare the compression results across different compression methods. Since SSEM splits all data into high-frequency part and low-frequency part half-by-half along each dimension, it cannot be used to compress the data array with odd dimension sizes. In particular, SSEM cannot deal with Eddy's application data because its array is 128×32×5×5 in shape. In our evaluation, we adopt the absolute error bound mode and set the error bound to 10^{-6}, which is accurate enough for most of research as indicated by the scientific research teams at Argonne. In practice, we suggest to set the error bounds by combining the absolute bound and relative bound, in order to fit the value ranges.

The compression effect is evaluated by using compression ratio and compression error. The compression ratio is defined as the ratio of the original data size to the compressed data size, and the compression error is defined as the deviation of the original data value and the decompressed data value. There are several metrics regarding compression error, such as maximum compression error and mean compression error. Due to the space limit, we mainly present the distribution (CDF) of the maximum compression errors, because we find in our experiments that the maximum compression error distribution is always consistent with other types of error metrics such as mean compression error.

B. Experimental Results

In what follows, we first show the compression results for the first 12 real-world applications, and then evaluate the compression ability with extremely large data set by using the last production application ATM.

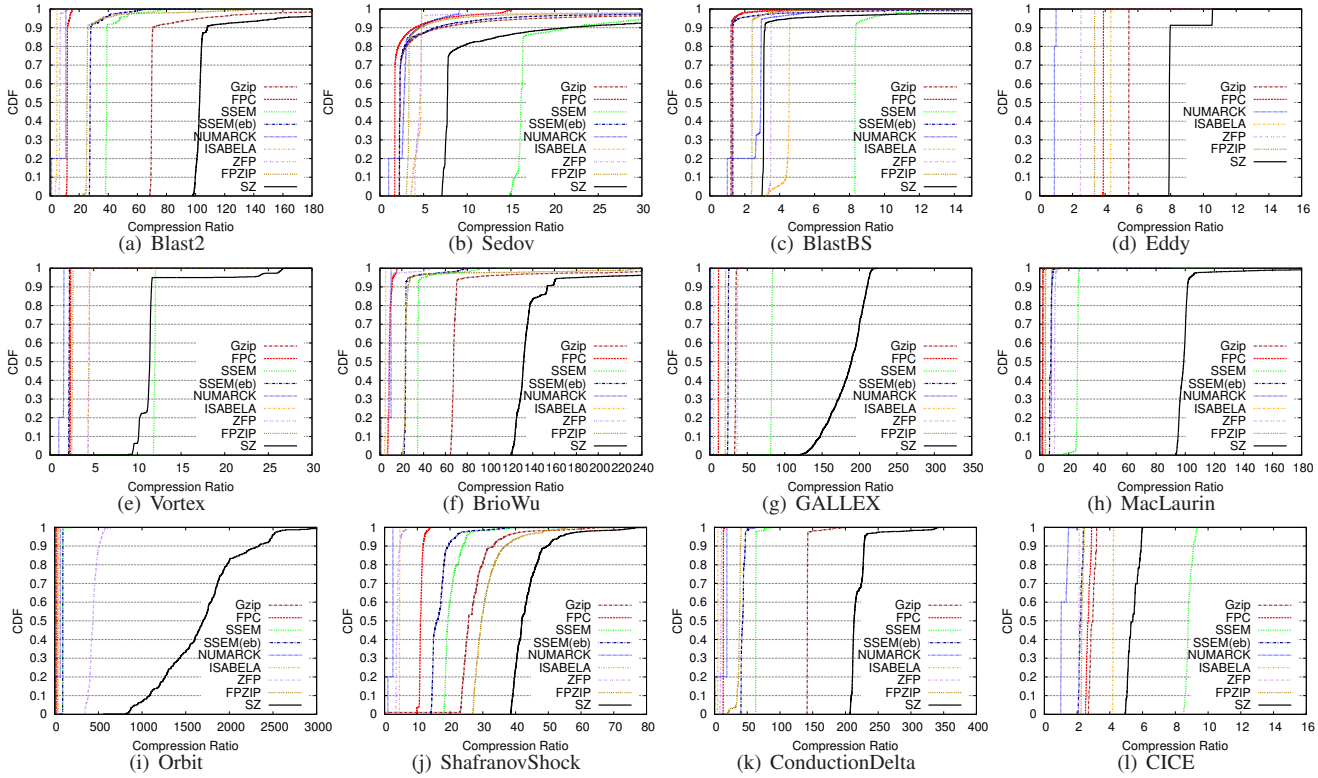
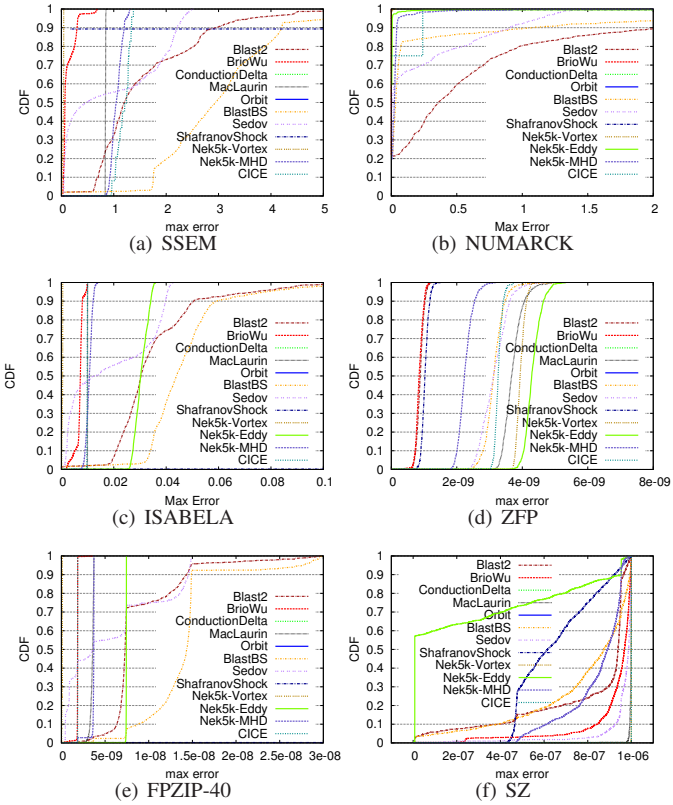


Figure 4. CDF of Compression Ratios (note that SSEM, NUMARCK and ISABELA do not respect specified error bound as shown in Figure 5)

Figure 4 presents the cumulative distribution function (CDF) of the compression errors for all of the 8 compression methods working on the 12 HPC applications. Each distribution curve is plotted based on the separate compression of the 1000 snapshots for each application. It is clearly observed that the compression ratio of SZ is significantly higher than that of other compressors. In absolute terms, the overall average compression ratios are in the range [3.3,436] for the 12 applications, which is higher than the second best solution twice or even by an order of magnitude in general. The compression ratios of ZFP and Gzip are in [2.1,10.2] and [1.8,160] respectively. There are only three exceptions (Sedov, BlastBS, and CICE) in which some other compressors look better than SZ with respect to the compression ratio. For instance, SSEM looks better than SZ in Sedov and CICE, but note that SSEM is actually **not error-bounded** compressor. From among all **error-bounded** compressors, the only exception where SZ's compression ratio is lower appears in BlastBS, where ZFP's compression ratio is slightly higher than SZ's.

In what follows, we analyze the key reasons why our solution leads to much higher compression ratios than others. We mainly discuss the lossy compressors here, and the discussion about lossless compressors can be found in Section V. SSEM transforms (by using Haar wavelets) all data to a new data set such that most of data are close to zero, and then approximates the transformed data by vector quantization. Its compression ratio highly depends on the Haar wavelet transform: at least $\frac{1}{2^D} \times 100\%$ of data cannot be close to zero (where D is the number of dimensions) after the Haar transform, such that the compression ratios will be degraded accordingly. Since SSEM is not an error-bounded compressor, we extend SSEM to be an error-bounded version (namely SSEM(eb) in the figure) by storing exactly the data whose decompressed values are against the error bounds. The compression ratio thus degrades significantly (as shown in the figure). NUMARCK computes the relative differences for each data point between adjacent time steps and performs the vector quantization over them. The key reason for its low compression ratio is that it has to periodically save the original snapshots to avoid the large compression errors, significantly degrading the overall compression ratios. ISABELA sorts all of the data for generating a sequence of relatively smooth data, but sorting data shuffles the data locations such that it has to store the data indices additionally, which will limit the compression ratio significantly. The limitation of ZFP is that its current release supports only up to 3 dimensions, while most of applications here (such as FLASH and Nek5000) are using 4+ dimensions. That is, we have to merge some dimensions (such as treating the 5D array $\{128,70,4,4,4\}$ as 3D array $\{128,70,64\}$) before using the ZFP. On the other hand, similar to JPEG compression principle, ZFP converts the original data values (actually the ones after the exponent



(a) SSEM (b) NUMARCK (c) ISABELA (d) ZFP (e) FPZIP-40 (f) SZ

Figure 5. Distribution (CDF) of Maximum Compression Errors

alignment) to a series of data whose values are very close to zero, by making use of the similarity of the neighboring data points. In comparison to their strategy, we approximate each data point based on not only its neighboring data values but also the curve-fitting predictions with three preceding values in the data stream, such that the data values will have more opportunities to be approximated by curve fitting. The reason SZ leads to relatively low compression errors on the BlastBS simulation is that its data are not very smooth such that most of them cannot be predicted precisely by curve-fitting. How to improve SZ's compression ratio for BlastBS will be studied in our future work.

In Figure 5, we present the maximum error distribution of different compressors in compressing the 1000 snapshots[1]. It is clearly observed that SZ can always guarantee the compression errors strictly within user-specified bound 10^{-6}. ZFP can also guarantee the compression error bounds, but it over-preserves the compression errors to be around 10^{-9} in most of cases, compared to the user-specified error bound.

We present the compression time and decompression time in Table II and Table III respectively. Due to the space limit, we present the cost for four typical compressors, including *ISABELA*, *ZFP*, *SZ w/o Gzip*, and *SZ with Gzip*. The compression/decompression times of other compressors usually range in between *SZ w/o Gzip* and *ISABELA*.

[1]Some curves cannot be seen in Figure 5(b) due to too large values.

Through Table II, we can see that ISABELA's compression time is significantly more than others', because of its costly sorting operation. ZFP's compression time ranges in [0.7,90.2] seconds, while that of the SZ with Gzip ranges in [0.95,84.3] seconds. It is worth noting that without Gzip, the compression time of SZ is in [0.56,39.8], which is only 50%-80% of ZFP's compression time in most cases. The key reason why SZ runs much faster than others is that its time complexity is $O(N)$ and the whole compression procedure is composed of fast operations such as bitwise operations. Moreover, we observe that the decompression time is also much less than that of other compressors. The decompression time of SZ is only 10%-50% of others' in most of cases. The decompression time of SZ is observed much less than its compression time, because unlike the compression, the decompression just computes the data using the corresponding best-fitting model and recover the unpredictable data using bitwise operations.

Table II
TOTAL COMPRESSION TIME ON 1000 SNAPSHOTS (IN SECONDS)

Application	DataSize	ISA.	ZFP	SZ w/o Gzip	SZ with Gzip
Blast2	787MB	129	8	6.4	9.4
Sedov	660MB	115	9.3	6.3	11.9
BlastBS	984MB	73.2	17	11.9	23.1
Eddy	820MB	143	17.4	8	14.2
Vortex	580MB	108	8.6	5.5	8.7
BrioWu	1.1GB	132	9.6	8.7	9.8
GALLEX	270MB	31	1	1.9	2.5
MacLaurin	6.3GB	1285	55	22.8	28.5
Orbit	152MB	19	0.7	0.56	0.95
Shaf.Shock	246MB	38.3	4.9	1.7	2.9
Cond.Delta	787MB	84	6.2	3.8	6.2
CICE	3.7GB	790	90.2	39.8	84.3

Table III
TOTAL DECOMPRESSION TIME ON 1000 SNAPSHOTS (IN SECONDS)

Application	DataSize	ISA.	ZFP	SZ w/o Gzip	SZ with Gzip
Blast2	787MB	35	9.3	1.3	2.5
Sedov	660MB	29.3	9.3	1.3	3
BlastBS	984MB	44	5.8	1.1	2.1
Eddy	820MB	35.6	17.5	1.2	2.8
Vortex	580MB	24.9	9.6	0.9	2
BrioWu	1.1GB	45.8	10.7	1.9	3.2
GALLEX	270MB	11.2	1.1	0.24	0.75
MacLaurin	6.3GB	277	52.5	3.8	5.6
Orbit	152MB	6.6	0.7	0.1	0.44
Shaf.Shock	246MB	12.7	5.6	0.26	1.27
Cond.Delta	787MB	34.3	7.1	0.92	2
CICE	3.7GB	162	86.4	7.6	14.7

Lastly, we present in Table IV the compression results for SZ and ZFP by using the application ATM with extremely large data size (totally 1.5TB). We mainly evaluate SZ and ZFP because ZFP is the most competitive one based on the above-shown evaluation results. The compression errors are not presented because the two solutions both lead to satisfactory compression errors based on user demands (either δ=10^{-4} or δ=10^{-6}), similar to Figure 5 (d) and (f). Through Table IV, we observe that SZ's compression ratio is 4.02-5.4, significantly higher than that of ZFP by about 80%. For comparison, the compression ratio of using Gzip to compress the ATM's snapshot data is only 1.33, which is much worse than ZFP and SZ. SZ's compression

is slower than that of ZFP by about 40%. Note that the compression time of SZ includes the time cost of Gzip. If we exclude the Gzip time, the compression times of SZ for δ=10^{-4} and δ=10^{-6} are only 22396 seconds and 25464 seconds respectively, which are less than that of ZFP by 18% and 20% respectively. It is also worth noting that scientific simulation data are produced only once but could be used many times later on for the post-analysis. That is, the decompression performance is more significant than compression performance for users. As shown in Table IV, SZ's decompression is faster than that of ZFP by about 4 times, clearly indicating that SZ outperforms ZFP.

Table IV
EVALUATION USING ATM WITH FAIRLY LARGE DATA SIZE (1.5TB)

	error bound δ=10^{-4}			error bound δ=10^{-6}		
	CR	**Cmpr time**	**Decmpr time**	**CR**	**Cmpr time**	**Decmpr time**
ZFP	3	27166 sec	30395 sec	2.3	31627 sec	36254 sec
SZ	5.4	43980 sec	6598 sec	4.02	51951 sec	7788 sec

V. RELATED WORK

Existing HPC data compression strategies can be split into two categories, lossless compression [5], [33], [12] and lossy compression [7], [10], [11], [6], which we will discuss respectively as below.

Typical lossless compressors include Gzip [5], LZ77 [24], Huffman encoding [33], FPC [9] and Fpzip [12]. Gzip [5] is a generic compression tool that can compress any type of data stream, such as video stream and graph file. It integrates the LZ77 [24] algorithm and Huffman encoding [33] to perform the compression. LZ77 algorithm makes use of a sliding window to search the same repeated sequences of the data and replace them with references to only one single copy existing earlier in the data stream. Huffman encoding [33] is an Entropy-based lossless compression scheme which assigns each symbol in the data stream a unique prefix-free code. FPC [9] is a lossless floating-point data array compressor, by analyzing the IEEE 754 binary representations and leveraging finite context model. Fpzip [12] was proposed to compress the HPC data compression by particularly focusing on the floating-point data compression. Fpzip can obtain higher compression ratio because of its more elaborate analysis on the HPC floating data, such as predictive coding of floating-point data. The common issue of such lossless compression methods is the relatively low compression ratio, which will significantly limit the performance of the runtime data processing or post-processing especially for exascale scientific simulation.

To improve the compression ratio, many lossy data compressors have been proposed in the recent years. Various compressors adopt different strategies, such as Bspline interpolation and vector quantization. A typical example using Bspline interpolation is ISABELA [7], which converted the multi-dimensional floating-point arrays in snapshots to *sorted* data-series before performing the data compression by B-spline interpolation. Due to the loss of the data location information in the sorted data-series, ISABELA has to use an extra index array to record the original index/location for each point in the data-series, significantly suffering from low compression ratio especially for the snapshot with extremely large number of data points. Vector quantization is a very common lossy compression scheme and a typical example is NUMARCK which has three steps to compress a snapshot: (1) periodically compute relative differences of values between adjacent time-steps for each data point, (2) approximate the differences by using vector quantization, and (3) replace the original data by quantized data. The key limitation of NUMARCK is its limited compression ratio.

Due to the huge challenge of HPC data compression, many recent lossy compressors combine different strategies. For example, four steps are performed in SSEM [6]: (1) split the data into low-frequency set and high-frequency set by wavelet transform, (2) perform vector quantization based on the distribution of high-frequency data set, (3) encode and record the data by bitmap, and (4) compress the output by Gzip. There are two drawbacks in the compression method: (1) it cannot guarantee the compression error bound and (2) it cannot work with the array with odd dimension sizes (such as 5X5). ZFP [11] is a lossy data compressor involving fixed-point integer conversion, block transform, bit-plane encoding, etc. Our experiments show that ZFP often compares favorably with other solutions except SZ.

We present in the previous section the evaluation results by comparing our proposed lossy compressor to all of the above solutions by using 13 applications across from 7 different scientific domains. Our solution leads to the overall highest compression ratio with satisfied compression errors, and lowest decompression times. It is also worth noting that our data compressor has a fairly high usability because it is suitable for any shape of the data array.

VI. CONCLUSION AND FUTURE WORK

In this paper, we present a novel error-bounded HPC floating-point data compressor with a state-of-the-art design. The key idea is to check each data point to see if it can be approximated by some bestfit curve fitting model and replace it by using a two-bit code indicating the model type if the approximation is within user-specified error bound. We evaluate our solution by using 13 applications across 7 different scientific fields and compare it to 7 other state-of-the-art compressors. Our compressor supports C, Fortran, and Java, and it is available to download under BSD license. The key findings are summarized below:

- SZ's compression ratio ranges in [3.3,436], which is higher than the second best solution twice or even by an order of magnitude in most of cases.
- The compression errors under SZ are always strictly limited in the user-specified error bound.
- The compression time of SZ is comparative to those of other solutions, while its decompression time is less

than the second best one by 50%-90% in most cases.

In the future work, we plan to further improve the compression ratio for some application cases (such as BlastBS) in which ZFP exhibits a slightly higher compression ratio. We also plan to enable our compressor to support common HPC data formats such as netCDF and HDF5.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Contract DE-AC02-06CH11357; and by ANR RESCUE, INRIA-Illinois-ANL-BSC Joint Laboratory on Extreme Scale Computing, and Center for Exascale Simulation of Advanced Reactors (CESAR) at Argonne.

REFERENCES

[1] A.H. Baker, H. Xu, J.M. Dennis, M.N. Levy, D. Nychka, and S.A. Mickelson, "A Methodology for Evaluating the Impact of Data Compression on Climate Simulation Data, " in *The ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC14)*, pp. 203-214, 2014.

[2] Community Earth Simulation Model (CESM). Available at https://www2.cesm.ucar.edu/.

[3] Storage Systems and Input/Output for Extreme Scale Science (Report of The DOE Workshops on Storage Systems and Input/Output). Available at http://science.energy.gov/∼/media/ascr/pdf/programdocuments /docs/ssio-report-2015.pdf.

[4] Lustre File System. Available at lustre.org.

[5] Gzip compression. Available at http://www.gzip.org.

[6] N. Sasaki , K. Sato, T. Endo, and S. Matsuoka, "Exploration of Lossy Compression for Application-level Checkpoint/Restart, " in *2015 IEEE 29th International Parallel and Distributed Processing Symposium*, pp. 914–922, 2015.

[7] S. Lakshminarasimhan, N. Shah, S. Ethier, S. Klasky, R. Latham, R. Ross, and N.F. Samatova, "Compressing the Incompressible with ISABELA: In-situ Reduction of Spatio-temporal Data, " in *17th International Conference on Parallel and Distributed Computing (Euro-Par)*, pp. 366–379, 2011.

[8] Fusion Cluster. [online]. Available at : http://www.lcrc.anl.gov/

[9] M. Burtscher and P. Ratanaworabhan, "High Throughput Compression of Double-Precision Floating-Point Data, " in *Data Compression Conference (DCC'07)*, pp.293–302, 2007.

[10] Z. Chen, S.W. Son, , W. Hendrix, A. Agrawal, W. Liao, and A. Choudhary, "NUMARCK: machine learning algorithm for resiliency and checkpointing, " in *IEEE/ACM Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC14)*, pp. 733–744, 2014.

[11] P. Lindstrom, "Fixed-Rate Compressed Floating-Point Arrays, " in *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, 2014.

[12] p. Lindstrom and M. Isenburg, "Fast and Efficient Compression of Floating-Point Data, " in *EEE Transactions on Visualizatoin and Computer Graphics*, 12(5):1245–1250, 2006.

[13] SZ compression library. [online]. Available at: https://collab.mcs.anl.gov/display/ESR/SZ.

[14] ASCF Center. FLASH User's Guide (Version 4.2). [online] Available at http://flash.uchicago.edu/site/flashcode/user_support/ flash2_users_guide/docs/FLASH2.5/flash2_ug.pdf

[15] p. Colella and P.R. Woodward, "The Piecewise Parabolic Method (PPM) for Gas-Dynamical Simulations," in *Journal of Computational Physics (JCP)*, 54:174–201, 1984.

[16] L.I. Sedov, "Similarity and Dimensional Methods in Mechanics (10th Edition)," New York: Academic, 1959.

[17] A.L. Zachary, A. Malagoli, and P. Colella, "A Higher-Order Godunov Method for Multidimensional Ideal Magnetohydrodynamics," in *SIAM Journal of Scientific Computing*, 15(2):263–284, 1994.

[18] M. Brio and C.C. Wu, "An Upwind Differencing Scheme for the Equations of Ideal Magnetohydrodynamics," in *Journal of Computational Physics*, 75:400–422, 1988.

[19] O. Walsh, "Eddy Solutions of the Navier-Stokes Equations," in *Proceedings of The Navier-Stokes Equations II - Theory and Numerical Methods*, 306–309, Oberwolfach 1991.

[20] P. Fisher, "Nek5000 User Guide," [online] Available at http://www.mcs.anl.gov/fischer/nek5000/examples.pdf.

[21] A. Obabko. Simulation of Gallium Experiment. [online] Available at: http://www.cmso.info/ cmsopdf/princeton5oct05/talks/Obabko-05.ppt

[22] V.D. Shafranov. The structure of shock waves in a plasma. in *Sov. Phys. JETP*, 5:1183, 1957.

[23] D. Bailey, M. Holland, E. Hunke, B. Lipscomb, B. Briegleb, C. Bits, and J. Schramm. Community Ice CodE (CICE) User's Guide (Version 4.0). [online]. Available: http://www.cesm.ucar.edu/models/ccsm4.0/cice/ice_usrdoc.pdf.

[24] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression, ", in *IEEE Transactions on Information Theory*, 23(3): 337-343, 1977.

[25] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, and N. Maruyama, S. Matsuoka. FTI: High Performance Fault Tolerance Interface for Hybrid Systems. in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11)*, pages 32:1–32:32, 2011.

[26] S. Di, M.-Slim Bouguerra, L.A. Bautista-Gomez, and F. Cappello. Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications. in *Proceedings of 28th International Parallel and Distributed Processing Symposium (IPDSP'14)*, pages 1181-1190, 2014.

[27] S. Di, L.A. Bautista-Gomez, and F. Cappello. Optimization of a Multilevel Checkpoint Model with Uncertain Execution Scales. in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'14)*, pages 907-918, 2014.

[28] K. Buchin, "Organizing Point Sets: Space-Filling Curves, Delaunay Tessellations of Random Point Sets, and Flow Complexes," Ph.D Dissertation, Department of Mathematics and computer science, Freien Universitat Berlin.

[29] E. H. Moore, "On certain crinkly curves, " Trans. Amer. Math. Soc., 1: 72-90, 1900.

[30] D. Hilbert: Über die stetige Abbildung einer Linie auf ein Flächenstück. Mathematische Annalen 38 (1891), 459-460.

[31] G. Peano, "Sur une courbe qui remplit toute une aire plane, " Math. Ann., 36: 157–160, 1890.

[32] H. Lebesgue, "Leçns sur l'intégration et la recherche des fonctions primitives professées au Collège de France, " Gauthier-Villars, 1904.

[33] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes, " in *Proceedings of IRE*, 40(9): 1098–1101, 1952.