Performance Optimization for Relative-Error-Bounded Lossy Compression on Scientific Data

Xiangyu Zou, Tao Lu, Wen Xia, Xuan Wang, Weizhe Zhang, *Senior Member, IEEE*, Haijun Zhang, Sheng Di, *Senior Member, IEEE*, Dingwen Tao, and Franck Cappello, *Fellow, IEEE*

Abstract—Scientific simulations in high-performance computing (HPC) environments generate vast volume of data, which may cause a severe I/O bottleneck at runtime and a huge burden on storage space for postanalysis. Unlike traditional data reduction schemes such as deduplication or lossless compression, not only can error-controlled lossy compression significantly reduce the data size but it also holds the promise to satisfy user demand on error control. Pointwise relative error bounds (i.e., compression errors depends on the data values) are widely used by many scientific applications with lossy compression since error control can adapt to the error bound in the dataset automatically. Pointwise relative-error-bounded compression is complicated and time consuming. In this article, we develop efficient precomputation-based mechanisms based on the SZ lossy compression framework. Our mechanisms can avoid costly logarithmic transformation and identify quantization factor values via a fast table lookup, greatly accelerating the relative-error-bounded compression with excellent compression ratios. In addition, we reduce traversing operations for Huffman decoding, significantly accelerating the decompression process in SZ. Experiments with eight well-known real-world scientific simulation datasets show that our solution can improve the compression and decompression rates (i.e., the speed) by about 40 and 80 p, respectively, in most of cases, making our designed lossy compression strategy the best-in-class solution in most cases.

Index Terms—Lossy compression, high-performance computing, scientific data, compression rate

1 Introduction

Cutting-edge computational research in various domains relies on high-performance computing 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, terabytes or even petabytes of analysis data would be produced easily by scientific simulations per run [1], [2], [3], when trying to capture the time evolution of physics phenomena in a fine spatiotemporal scale. Climate scientists, for instance, need to run large ensembles of high-fidelity 1 km $\times$ 1 km simulations. Estimating even one ensemble member per simulated day may generate 260 TB of data every 16 s across the ensemble [4]. The data volume and data movement rate are imposing unprecedented pressure on storage and interconnects [5], [6], for both writing data to persistent storage and retrieving them for postanalysis. As HPC storage infrastructure is being pushed to the scalability limits in terms of both throughput and capacity [7], the communities are striving to find new approaches to lower the storage cost. Data reduction, among others, is deemed to be a promising candidate by reducing the amount of data moved to storage systems.

Data deduplication and lossless compression have been widely used in general-purpose systems to reduce data redundancy. In particular, deduplication [8] 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. However, the deduplication method rarely works for scientific data, since it typically reduces the scientific dataset by only 20 to 30 percent, as reported in recent studies [9], which is far from being useful in production. On the other hand, lossless compression in HPC is designed to reduce the storage footprint of applications without any data loss (e.g., using the checkpoint/restart mechanism to protect the applications in case of failures). Lossless compression usually suffers from very limited reduction ratios (less than two [10]) on scientific simulation data, since the simulation data are often stored in the form of floating-point numbers (also called *floats* for short in the following text) each with rather random ending mantissa bits. With growing

- *X. Zou, X. Wang, W. Zhang, and H. Zhang are with the Harbin Institute of Technology, Shenzhen, Guangdong 518055, China, and also with Peng Cheng Laboratory, Shenzhen, Guangdong 518055, China. E-mail: xiangyu.zou@hotmail.com, {wangxuan, wzzhang, hjzhang}@hit.edu.cn.*
- *W. Xia is with the Harbin Institute of Technology, Shenzhen, China, Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, Guangdong 518055, China, and also with the Wuhan National Laboratory for Optoelectronics, Wuhan, Hubei 430074, China. E-mail: xiawen@hit.edu.cn.*
- *T. Lu is with Marvell Technology Group, Santa Clara, CA 95054. E-mail: taovcu@gmail.com.*
- *S. Di and F. Cappello are with Argonne National Laboratory, Lemont, IL 60439. E-mail: sdi1@anl.gov, cappello@mcs.anl.gov.*
- *D. Tao is with University of Alabama, Tuscaloosa, AL 35487. E-mail: tao@cs.ua.edu.*

Manuscript received 18 Aug. 2019; revised 3 Jan. 2020; accepted 30 Jan. 2020. Date of publication 10 Feb. 2020; date of current version 5 Mar. 2020. (Corresponding author: Wen Xia.) Recommended for acceptance by W. Yu. Digital Object Identifier no. 10.1109/TPDS.2020.2972548

disparity between compute and I/O, more aggressive data reduction schemes are needed to further reduce data by an order of magnitude or more [4], so the focus has shifted to lossy compression recently.

In this paper, we focus mainly on pointwise relative-error-bounded lossy compression because pointwise relative error bound (or relative error bound for short) is a critical error controlling method broadly adopted by many scientific applications in the lossy compression community. Unlike absolute error control adopting a fixed error bound for each data point, the pointwise relative error bound indicates that the compression error on each data point should be restricted within a constant percentage of its data value. In other words, the smaller the data value, the lower the absolute error bound on the data point. Accordingly, more details can be preserved in the regions with small values under the pointwise relative error bound than with absolute error bound. According to cosmologists, for example, the lower a particle's velocity is, the smaller the compression error it can tolerate. Moreover, many other studies [11], [12] also have focused on pointwise relative error bound in lossy compression.

In particular, ZFP and SZ have been widely recognized as the top two error-controlled lossy compressors with respect to both compression ratio and rate [12], [13], [14], [15]. They have been tested on thousands of computing nodes. For example, the Adaptable IO System (ADIOS) [16] deployed on the Titan (OLCF-3) supercomputer at Oak Ridge National Laboratory has integrated both ZFP and SZ for data compression. Although being the two best-in-class compressors, however, ZFP and SZ still have their own pros and cons because of distinct design principles. With the same error bound setting, SZ usually has higher compression ratios than ZFP does (about 2X or more), while SZ is often 20–30 percent slower than ZFP [12], bringing up a dilemma for users to choose an appropriate compressor. In fact, the compression/decompression rate is as important as the compression ratio, in that many applications require fast compression at runtime because of an extremely fast data production rate such as the X-ray analysis data generated by the Advanced Photon Source and Linac Coherent Light Source [17], [18]. A straightforward question is, can we significantly improve the compression and decompression rates (i.e., the processing speed) based on the SZ lossy compression framework, leading to an optimal lossy compressor for users?

Therefore, the aim of this paper is to significantly improve the performance for both compression and decompression in SZ, while still keeping a high compression ratio and strictly respecting the user-required error bounds. This research is nontrivial because of two reasons: ① Logarithmic transformation, which plays an important role in pointwise relative-error-bounded lossy compression, is a complicated and time-consuming process. How to speed up logarithmic transformation and improve the overall compression rate is challenging. ② To this end, we develop a precomputation-based mechanism and prove its effectiveness in theory. However, our initial model may lead to non-error-bounded cases. How to guarantee the user-required error bound has to be considered carefully.

The key contributions of our work are fourfold:

- We identify the performance bottleneck in SZ. Specifically, our in-depth performance analysis shows that the online logarithmic transformation leads to a low compression and decompression performance for SZ. In absolute terms, it takes about 33 percent of the aggregate compression time and 40 percent of the aggregate decompression time, significantly limiting the compression rate of SZ.
- We propose an efficient precomputation-based mechanism that can significantly increase the compression rate of the pointwise relative-error-bounded compression in SZ. Our solution eliminates the time cost of logarithmic transformation, replacing it with a fast table-lookup method in the point-by-point processing stage. At the same time, we develop an adaptive method to configure parameters in table building, for a better compression rate. With detailed analysis of the complicated mathematical relations among quotient values, quantization factor values, and error bounds, our optimized precomputation-based mechanism can strictly respect specified error bounds to achieve accurate error control.
- We also develop a precomputation-based mechanism for Huffman decoding in SZ by constructing three precomputed tables. We replace the costly repeated tree traversing operations during Huffman decoding with efficient table lookup operations, significantly accelerating the decompression process. We also propose an adaptive method to configure parameters in Huffman decoding automatically.
- We perform a comprehensive evaluation using eight well-known real-world simulation datasets across different scientific domains. Experiments show that our solution does not degrade any compression quality metric tested, compared with original SZ approach, including maximum relative error (MAX E) and mean relative error (MRE). Our precomputation-based mechanism improves the compression rate by about 40 percent and decompression rate by about 80 percent in most of cases. Our codes are available at https://github.com/Borelset/SZ, which has been merged into the SZ package in version 2.1 .

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present the time cost of logarithmic transformation and quantization factor calculation, which motivates us to conduct this research. In Section 4, we present the design and implementation of our precomputation schemes. In Section 5, we evaluate our new schemes with multiple real-world scientific HPC application datasets across different domains, comparing our scheme with the latest SZ and ZFP. In Section 6, we summarize our work.

2 Related Work

Lossless compression fully maintains data fidelity, but it depends on the repetition of symbols in the data sources. Even for slightly variant floating-point values, their binary representations may hardly contain identical symbols (or exactly duplicated chunks). As such, lossless compression suffers from a very low compression ratio [2], [9], [12], [19], [20] on scientific data.

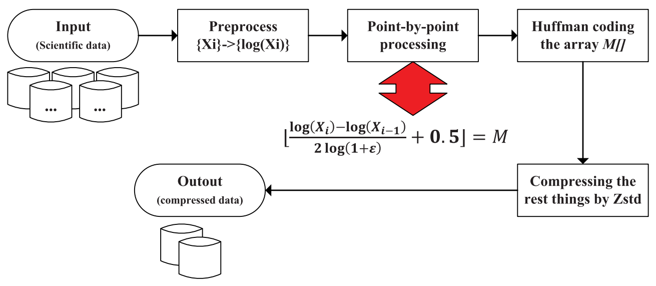
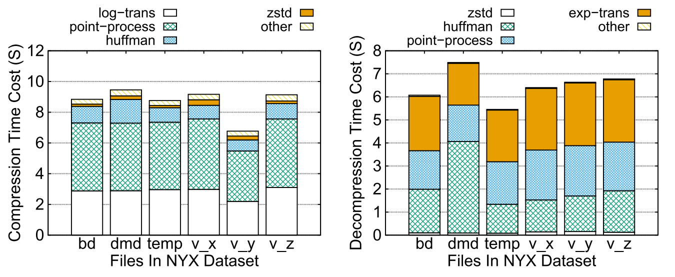


Fig. 1. General workflow of SZ compression with logarithmic transformation.



(a) Compression (b) Decompression

Fig. 2. Breakdown of the compression and decompression time for SZ with logarithmic transformation (using 1E-2 as pointwise relative error bound on NYX dataset).

General acceptance of certain data loss provides an opportunity to dramatically improve the data compression ratio; ZFP [19], ISABELA [21], and SZ [2], [22] are three well-known lossy compressors supporting pointwise relative error bounds.

ZFP [19] is an error-controlled lossy compressor designed for image data. ZFP transforms the floating-point data points to a fixed-point data values block by block. Then, a reversible orthogonal block transformation is applied in each block to mitigate the spatial correlation, and embedded coding [23] is used to encode the coefficients.

Motivated by the reduction potential of spline functions [24], [25], ISABELA [21] uses B-spline-based curve fitting to compress scientific data. Intuitively fitting a monotonic curve can provide a model that is more compressible than fitting random data. Based on this, ISABELA first sorts data to convert highly irregular data to a monotonic curve. Its biggest weakness is slow compression/decompression because of its expensive sorting operation.

The SZ compressor has experienced multiple enhancements since the first version 0.1 [22] was released in 2016. SZ 0.1 employed multiple curve-fitting models to compress data streams, with the goal of accurately approximating the original data, which encodes the best-fit curve-fitting type for each data point or marks the data point as unpredictable data if its value is too far away from any curve-fitted value. SZ 1.4 [2] significantly enhanced the compression ratios by improving the prediction accuracy with a multidimensional prediction method plus a linear-scaling quantization method. SZ 2.0 [26] further improved the compression quality for the high-compression cases by leveraging an adaptive method facilitated with two main candidate predictors (Lorenzo and linear regression). The former predictor uses neighboring processed data to predict the next data; the latter processes data as blocks and predicts data according to the characteristics of the block. We [20] proposed an efficient logarithmic transformation to convert a pointwise relative-error-bounded compression problem to an absolute-error-bounded compression problem. It improves the compression quality compared with the prior work [27] that separates all data into several segments, and process each segment of data as the way of absolute error bound compression. However, as we have confirmed in our performance profiling, this will significantly slow the compression and decompression because of its costly logarithmic transformation operations.

Some existing studies are working on combining different lossy compressors to obtain better compression quality. Lu *et al.* [12] conducted a comprehensive evaluation based on SZ and ZFP and proposed a simple sampling method to select the best compressor with higher compression ratio in between. Tao *et al.* [14] proposed an efficient online, low-cost selection algorithm that can predict the compression quality accurately for SZ and ZFP in early processing stages and selects the best-fit compression based on the quality metric Peak Signal-to-Noise Ratio (PSNR) for each data field. Their work, however, relies on the compression performance of SZ and ZFP, so their compression result can never go beyond the best choice from between SZ and ZFP.

3 Motivation

SZ usually leads to higher compression ratios [12] than other compressors with the same error bound setting especially for 1D and 2D datasets, making it one of the best compressors for HPC scientific data. Generally, SZ performs the compression based on the following four steps:

- Applying prediction to the given dataset based on user-set error bound: all the floating-point data values are mapped to an array of quantization factors (integer values), with an accuracy loss restricted within the error bound.
- Constructing a Huffman tree of the quantization factors, and encoding the quantization factors.
- Compressing the unpredictable data points (i.e., the data points whose values cannot be approximated by the first step) by binary-representation analysis.
- Further compressing of the data generated by the above three steps by using lossless compressors such as GZip [28] or Zstandard [29] (usually called Zstd).

This SZ compression framework is particularly effective on absolute-error-bounded compression.

To address the demand of point-wise relative error bound, the SZ team developed a logarithmic transformation [20] which can convert the pointwise relative-error-bounded compression problem to an absolute-error-bounded compression problem, as shown in Fig. 1.

A serious drawback of the logarithmic transformation-based strategy is its high transformation cost, which may significantly lower the compression/decompression rate. As Fig. 2 demonstrates, the logarithmic transformation consumes about 30 percent of the overall compression and decompression time. To avoid such high transformation cost, we propose an efficient precomputation-based mechanism with a fast table-lookup method, dramatically accelerating SZ compression and decompression.

TABLE 1
Key Notations

Term	Explanations
f	float $f = X_i/X_i'$; X_i is an original value; X_i' is its predicted value
M	The quantization code, an integer, which is located in a fixed range L; $M[]$ refers to a set of quantization codes.
L	The range of quantization code M specified by users
V	The range covered by all of $PI(M)$ or $PI'(M)$
ε	The error bound specified by users
PI(M)	An interval where any float x located in, and x could be presented by $(1+\varepsilon)^{2M}$ with a relative error smaller than ε. This is used for Model A in SZ_P′
θ	A parameter used in Model B to control the intersection size
p	A parameter specified by users, to define θ as 2^{-p}
PI′(M)	An interval where any float x located in, and x could be presented by $(1+\varepsilon)^{2(M-\theta)}$ with a relative error smaller than ε. This is used for Model B in SZ_P
$T1$	Store the mapping relation of $f \rightarrow M$ for compression, consisting of several subtables
$T2$	Store the mapping relation of $M \rightarrow f$ for decompression
sub-table	Divide V into several segments by the exponent part of floating-point values in V, each segment corresponds to a sub-table.
grid	Divide each segment into several equal-sized girds, each grid maps to a sub-table entry
Δ	The intersection of two neighboring PI(M). We ensure $\Delta >$ size of the grid, for error control

4 A Precomputation-Based Transformation Scheme for Lossy Data Compression

In this section, we present the theories and practices of precomputation-based transformation scheme. First, we prove the feasibility of replacing logarithmic transformation and floating-point quantization procedures in previous SZ design [20] (denoted as **SZ_T**) with an efficient precomputation-based table lookup procedure (denoted as **SZ_P**). Second, we discuss how to construct the tables for the precomputation-based transformation and the adaptive methods for some parameters. We also provide a detailed algorithm description of our approach **SZ_P**. Table 1 lists and describes some key notations to help understand the design of **SZ_P** in this section.

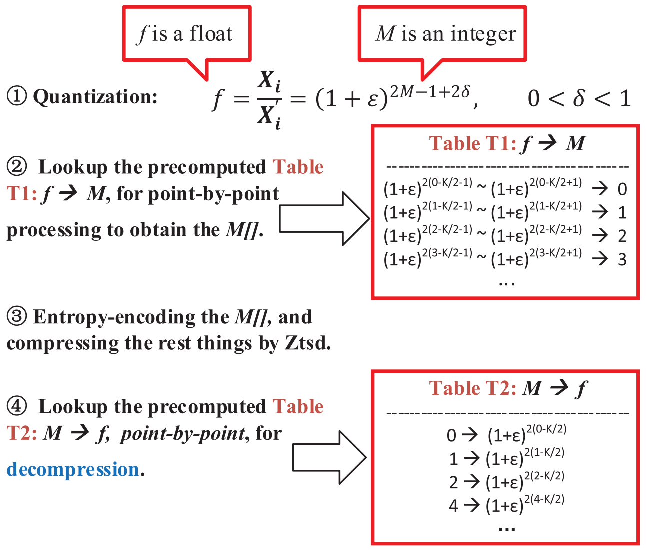
4.1 Theoretical Derivation of Precomputation-Based Transformation Scheme for SZ

When SZ compresses floats with an absolute error bound, each data point will be predicted to calculate the quantization code, converting a floating-point number lossy compression problem to an integer number lossless compression problem as follows.

$$\left\lfloor \frac{X_i - X_i'}{2\varepsilon} + 0.5 \right\rfloor = M. \tag{1}$$

In the above equation, ε is a user-predefined relative error bound; M is a derived integer called the quantization code or factor, which can be further encoded by using Huffman coding to reduce storage space; and X_i' is the multidimension-based predicted value of X_i [2].

For relative-error-bounded compression, as introduced in the preceding section and shown in Fig. 1, the latest SZ version (SZ_T) converts the relative-error-bounded compression



f is a float
M is an integer
① Quantization: $f = \frac{X_i}{X_i'} = (1+\varepsilon)^{2M-1+2\delta}, \quad 0 < \delta < 1$
② Lookup the precomputed Table T1: $f \rightarrow M$, for point-by-point processing to obtain the $M[]$.
Table T1: $f \rightarrow M$
$(1+\varepsilon)^{2(0-K/2-1)} \sim (1+\varepsilon)^{2(0-K/2+1)} \rightarrow 0$
$(1+\varepsilon)^{2(1-K/2-1)} \sim (1+\varepsilon)^{2(1-K/2+1)} \rightarrow 1$
$(1+\varepsilon)^{2(2-K/2-1)} \sim (1+\varepsilon)^{2(2-K/2+1)} \rightarrow 2$
$(1+\varepsilon)^{2(3-K/2-1)} \sim (1+\varepsilon)^{2(3-K/2+1)} \rightarrow 3$
...
③ Entropy-encoding the $M[]$, and compressing the rest things by Ztsd.
④ Lookup the precomputed Table T2: $M \rightarrow f$, point-by-point, for decompression.
Table T2: $M \rightarrow f$
$0 \rightarrow (1+\varepsilon)^{2(0-K/2)}$
$1 \rightarrow (1+\varepsilon)^{2(1-K/2)}$
$2 \rightarrow (1+\varepsilon)^{2(2-K/2)}$
$4 \rightarrow (1+\varepsilon)^{2(4-K/2)}$
...

Fig. 3. Our approach using two precomputed tables to directly transform the error-controlled quantization codes f[] to integers M[].

to an absolute-error-bounded compression by a logarithmic transformation as follows.

$$X_i \times (1-\epsilon) \; < \; X_i \; < \; X_i \times (1+\epsilon) \tag{2}$$

$$\Longrightarrow \quad log(X_i) + log(1-\epsilon) \; < \; log(X_i) \; < \; log(X_i) + log(1+\epsilon).$$

Before logarithmic transformations, X_i are all converted to their absolute value because logarithmic transformation can not process negative values. Their signs are saved in an array, and we will use X_i to represent the converted data points in the remaining of the paper.

Due to $log(X_i) + log(1-\epsilon) < log(X_i) - log(1+\epsilon)$, we can infer that if Inequality (3) holds, then the Inequality (2) will also hold. Therefore, a relative-error-bounded compression problem as Equation (2) could be converted into an absolute-error-bounded compression problem as below.

$$log(X_i) - log(1+\epsilon) \; < \; log(X_i) \; < \; log(X_i) + log(1+\epsilon). \tag{3}$$

Thus, we can use an equation to show how data are processed in SZ with a relative error bound as follows.

$$\left\lfloor \frac{\log X_i - \log X_i'}{2\log(1+\varepsilon)} + 0.5 \right\rfloor = M. \tag{4}$$

Then, we can deduce Equation (5) and subsequently Equation (6) as follows.

$$\frac{\log X_i - \log X_i' + \log(1+\varepsilon)}{2\log(1+\varepsilon)} = M + \delta, \quad 0 \le \delta < 1 \tag{5}$$

$$\Longrightarrow \quad f = \frac{X_i}{X_i'} = (1+\varepsilon)^{2M-1+2\delta}, \quad 0 \le \delta < 1. \tag{6}$$

Our approach is inspired from Equation (6). Fig. 3 demonstrates how the precomputation-based table lookup in SZ_P can avoid the time-consuming logarithmic transformation and

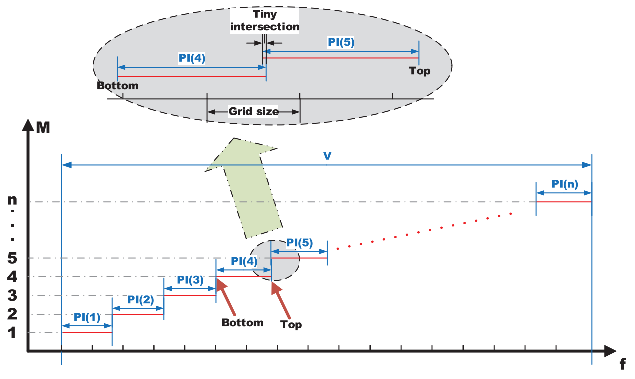


Fig. 4. A general description about model A of SZ_P.

quantization factor calculation in SZ_T. The main idea is to construct these two mappings to bypass the logarithmic transformation. However, how to build these two mappings and prove the effectiveness of this method is challenging. We provide a detailed theoretical analysis of our SZ_P method as follows.

Equation (6) indicates that for any floating-point f, we can get an integer M. M should belong to a fixed and predefined range, namely, the interval capacity $\boldsymbol{L}$. If he value of f gets M out of L, f would be considered unpredictable, and it would be handled by SZ's separate compression (by truncating insignificant bits in binary representation). Based on $0 < \delta < 1$, we can get $2M - 1 < 2M - 1 + 2\delta < 2M + 1$. From Equation (6), we can deduce

$$(1+\varepsilon)^{2M-1} < f < (1+\varepsilon)^{2M+1}. \tag{7}$$

Equation (7) gives us a hint that we can probably separate the range of f into some intervals and acquire the quantization factor M for a floating-point number k by finding out which interval that the k belongs to.

Naive model (Model A) of SZ_P

Inspired by the hint, we propose a naive model (we call it Model A) to realize the above idea, and the general description of our first model is shown in Fig. 4, in which we use a series of intervals to cover the range of f. Now, we have three issues to resolve. **The** 1st **issue** is what the format of the interval should be. **The** 2nd **issue** is whether any number in an interval can be compressed into one value with an error smaller than the user-specified error bound. **The** 3rd **issue** is whether any number can be included within the above mentioned neighbouring intervals. We will solve the three issues as below.

Considering the decompression, we want to get a decompressed value f′ from M, and f′ should satisfy the error bound requirement. So, we have

$$1-\varepsilon \leq \frac{(1+\varepsilon)^{2M}}{f} \leq 1+\varepsilon, \tag{8}$$

where $(1+\varepsilon)^{2M}$ is f' and f'/f should be within $(1-\varepsilon) \sim (1+\varepsilon)$. From Equation (8) we can further deduce that

$$(1+\varepsilon)^{2M-1} \leq f \leq \frac{(1+\varepsilon)^{2M}}{1-\varepsilon}. \tag{9}$$

We call this interval $[(1+\varepsilon)^{(2M-1)}, \frac{(1+\varepsilon)^{2M}}{1-\varepsilon}]$ as M's present interval, denoted by **PI(M)**. So, *the* 1st *issue and* 2nd *issue are solved*. **PI(M)** is a reasonable format of the intervals, and any float in **PI(M)** can be compressed into $(1+\varepsilon)^{2M}$ with an error smaller than the user-specified error bound.

For any pair **PI(M)** and **PI(M+1)**, we can get their top and bottom boundaries (see the example shown in Fig. 4) as follows:

$$\begin{aligned} PI(M)_{top} &= \frac{(1+\varepsilon)^{2M}}{1-\varepsilon} \\ PI(M+1)_{bottom} &= (1+\varepsilon)^{2M+1}. \end{aligned} \tag{10}$$

Also, we can get

$$\frac{PI(M)_{top}}{PI(M+1)_{bottom}} = \frac{\frac{(1+\varepsilon)^{2M}}{1-\varepsilon}}{(1+\varepsilon)^{2M+1}} = \frac{1}{1-\varepsilon^2} > 1. \tag{11}$$

Further, we can get the following relations.

$$PI(M)_{bottom} < PI(M+1)_{bottom} < PI(M)_{top} < PI(M+1)_{top}$$

These indicate that two neighboring **PI**s are intersecting (see the example in Fig. 4), although the intersections are extremely narrow. Therefore, **the** 3rd **issue** is also solved, there is no any floating-point number existing between two neighboring **PI**s while all the **PI**s cover a bounded continuous interval, called $\boldsymbol{V}$.

We can then rasterize $\boldsymbol{V}$ to build Table **T1**, where each grid corresponds to a table entry. Thus, we can build a mapping relation between f (i.e., **PI**(n)) and M (i.e., n). During compression, for each input floating-point f based on Equation (4), we can calculate in which grid it is located, then getting the value M from Table **T1**. As a result, f can be represented by $(1+\varepsilon)^{2M}$, which can restore the value of f during decompression respecting the predefined error bound ε.

Obviously, if we can acquire that a floating-point number f belongs to which **PI**, we can directly know which M f maps to. However, there is another problem to resolve: how can we know which **PI** the floating-point number f belongs to. Since brute-force search by traversing the **PI**s is very time-consuming, we must explore other more efficient solutions.

We decide to build a precomputed table to avoid the traversal cost. Fig. 4 summarizes this method, and we call this the basic method Model **A** in the paper. In this model, we separate the range $\boldsymbol{V}$ into several grids with equal size. For each grid, we find a **PI** to maps to. In this way, for any float f, we first find out which grid the f belongs to and second find out which **PI** the grid maps to, and finally using $(1+\varepsilon)^{2M}$ as the compressed value of f. Only the integer M needs to be saved for decompression.

To summarize, the challenge of adopting Model **A** is that there exist some grids divided by two **PI**s (as shown in Fig. 4). Thus, no matter how we decide these grids belong to which **PI**s, we always arrive at a situation in which a grid B maps to **PI(K)** but a number d in grid B does not belong to **PI(K)**. This results in higher relative errors than the error bound ε, which makes our compressor *non-error-bounded*. Model **A** therefore should be further improved to strictly respect the error bound.

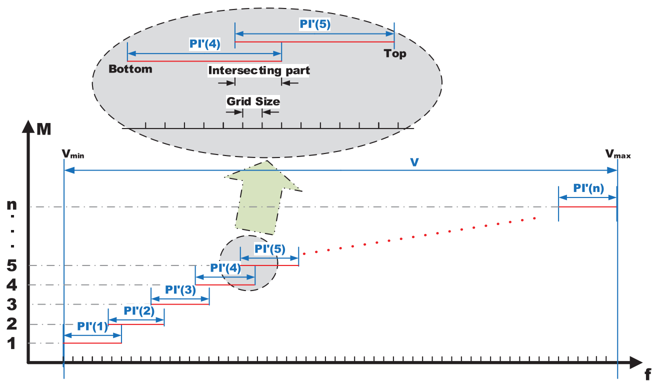


Fig. 5. General description of Model B of SZ_P.

Improved model (Model B) of SZ_P

To address the issue that Model **A** does not strictly respect error bounds, we propose a new design of **PI**′, namely, Model **B**, that has more intersecting parts to ensure data are strictly error bounded after decompression. In Model **B**, we shift **PI**s from Equation (9) and enlarge their intersections. More specifically, we define the new f' and **PI**′ as

$$(1+\varepsilon)^{M(2-\theta)-1} \leq f' \leq \frac{(1+\varepsilon)^{M(2-\theta)}}{1-\varepsilon}, \quad 0 < \theta \leq 1, \tag{12}$$

where θ is introduced for enlarging the intersection ratio. Because $\mathbf{PI'(M)}_{top} > \mathbf{PI'(M)}_{bottom}$, **PI**′s are intersecting, so they can also cover a bounded continuous interval V (as shown in Fig. 5). Concerning the relation between a grid and a **PI**′, we present the following lemma.

Lemma 1. *If a grid size* **G** *is smaller than the size of any intersecting part of* **PI**′*, a* **PI**′ *completely including the grid always exists.*

Proof. Consider the intersecting size of two neighboring **PI**′s.

$$\begin{aligned} PI'(M)_{top} &= \frac{(1+\varepsilon)^{M(2-\theta)}}{1-\varepsilon} \\ PI'(M+1)_{bottom} &= \frac{(1+\varepsilon)^{(M+1)(2-\theta)}}{1+\varepsilon} \end{aligned} \tag{13}$$

$$\begin{aligned} \Delta &= PI'(M)_{top} - PI'(M+1)_{bottom} \\ &\geq (1+\varepsilon)^{M(2-\theta)+1}(1-(1+\varepsilon)^{-\theta}). \end{aligned} \tag{14}$$

As the condition provided by Lemma 1 (shown in Fig. 5), we assume that all the grid sizes are smaller than any of the Δ. We can further prove the lemma with all three possible cases of the grids as follows.

1) A grid in which there is no **PI**′'s boundary, is completely included in a **PI**′. Since **PI**′s cover a bounded continuously interval, this kind of grid is included in a **PI**′.
2) A grid in which there is only one of the **PI**′'s boundaries, is completely included in a **PI**′. Without loss of generality, we assume that it is **PI**′(n+1)′s bottom boundary and it is a float number K, and the grid is G. So **PI**′(n)′s top boundary is K+Δ. Because the size of grid G is smaller than Δ, G is completely included in **PI**′(n). Thus, this kind of grid is also completely included in a **PI**′. The same conclusion can be made assuming that the boundary is **PI**′(n-1)′s top boundary.
3) A grid in which there are two or more **PI**′'s boundaries does not exist. The reason is that the size of **PI**′ is bigger than Δ and Δ is bigger than the grid size. □

Therefore, Lemma 1 demonstrates that the mapping relation between f (i.e., the grid) and M (i.e., n) in Model **B** is $N \rightarrow 1$ and that every grid could be completely included by a **PI**′, which Model **A** cannot do. Model **B** can avoid the problem appearing in Model **A**, that is, relative errors higher than the predefined bound.

Compared with **PI**, the tight arrangement of **PI**′ can lead to a better error control but also result in a smaller size of the covered range V. It depends on the parameter θ: a smaller θ can lead to a compression ratio similar to that of Model **A** but also a larger total grid count, which means higher memory cost. Empirically, θ should be set to 1, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and so on, which makes it easier to decide grid size in Model **B**. We adopt Model **B** in SZ_P, which brings us better error control but a slight decrease in the compression ratio. We note that in both Model **A** and Model **B**, once the table is generated, it can be saved for repeated future use, because the table is related only to the user-set error bound ε, instead of the values of the input datasets.

4.2 Table Construction Using Model B

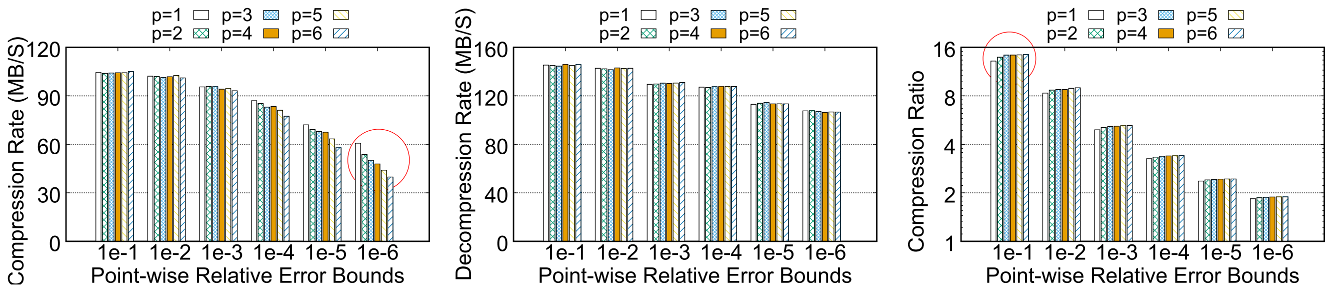
The next problem is how to build Table **T1** with Model **B**, namely, building the mapping relation between f (i.e., $\frac{X_i}{X_i'}$) and M (i.e., the error-controlled quantization code). We mainly discuss the construction of Table **T1** here since Table **T2** can be easily calculated and constructed, as shown in Algorithm 2 (see Section 4.5).

According to Lemma 1, we can know the condition that the size of grids should meet. There are also some challenges in building Table **T1**. The first is that the exponential model such as $(1+\varepsilon)^{M(2-\theta)}$ in Equations (13) and (14) has a huge growth rate, and a fixed grid size will lead to a huge amount of grids that can make Table **T1** have a huge size. Thus, controlling the size of the table is important. Next we introduce how we overcome this.

In Table **T1**, for a given error bound ε and an interval capacity $\boldsymbol{L}$, we divide the range **V** into many equal-sized grids. In Model **B** we divide the range **V** into several segments with **PI**′'s exponent, process them separately, and set different sizes for grids in different segments. In each segment, as discussed in Lemma 1, since the grid size must be smaller than the intersecting size of two smallest neighboring **PI**′s, we define the size of smallest **PI**′ in a segment is S_{seg}, because the size of **PI**′ and intersection in a segment is the same.

Therefore, if θ is 1, the grid size should be half of S_{seg}, which is the size of smallest intersection. Meanwhile, θ determines the size of intersecting parts of **PI**′s.

If θ is 1, any **PI**′(n) should be covered by two neighbors (i.e., **PI**′(n+1) and **PI**′(n-1)), so the grid size should be set smaller than $\frac{S_{seg}}{2}$. If θ is $\frac{1}{2}$, the grid size should be set smaller than $\frac{S_{seg}}{4}$, and so on. Each segment will be separated into several grids



(a) Compression Rate (b) Decompression Rate (c) Compression Ratio

Fig. 6. Performance comparison of SZ_P with different plus_bits (p) on the NYX dataset.

depending on its grid size. Furthermore, the index number of f in a segment of Table **T1** can be calculated as $\frac{f-Seg_{min}}{gridsize}$.

Consider the cost of division operations. In order to save time getting the index number of f as calculated above, the grid size should be simple in the binary system. Hence, we keep just one valid number for the grid size in IEEE 754 format [30]. For example, we can transform the $\frac{f-Seg_{min}}{1.101*2^{-5}}$ to $\frac{f-Seg_{min}}{1.000*2^{-5}}$ (in order to make the grid size smaller and also satisfy Lemma 1), which is equal to $(f - Seg_{min}) * 2^5$, in which the multiply operation does not change the mantissa of $f - Seg_{min}$. Therefore, we can get the index from the mantissa of $f - Seg_{min}$ directly and no longer need a division operation.

Now that we have determined the way to solve the problem of controlling the size of Table **T1**, we discuss how to separate the range V into segments and decide the size of grids for each segment.

Since floating-point data in cmputer systems are saved in IEEE 754 format [30], we divide range V into several segments according to f's exponent, and we build subtables for each segment separately. For the segment where f's exponent is k and the smallest $(1+\varepsilon)^{M(2-\theta)}$ in Seg_k is S, we define the grid size $G = 1.0 \times 2^k \times \varepsilon \times \theta$.

Since the exponent part of any $(1+\varepsilon)^{M(2-\theta)}$ in Seg_k is k, any S is larger than 1.0×2^k. For Seg_k, we can get the value of Δ, which we have mentioned in Lemma 1, as

$$\begin{aligned}\Delta &= (1+\varepsilon)^{(M(2-\theta)+1)}(1-(1+\varepsilon)^{(-\theta)}) \\ &= S(1+\varepsilon)(1-(1+\varepsilon)^{(-\theta)}).\end{aligned}$$

Since $S > 1.0 \cdot 2^k$ and $(1+\varepsilon)(1-(1+\varepsilon)^{(-\theta)}) \geq \varepsilon \times \theta$, we know $\Delta > \mathbf{G}$, so $\mathbf{G}$ is a reasonable grid size that satisfies the requirement discussed in Lemma 1. Accordingly, if the grid size is $\mathbf{G}$, any grid in Seg_k can always find a $\mathbf{PI}'$ that completely includes grid. So, $\mathbf{G}$ can be used as the grid size for each segment. However, if we use $\mathbf{G}$ as the grid size, which always has many mantissas, we usually need a divide operation to know which grid a floating-point number should belong to, and it is also time-consuming. As such, we need a grid size, which has only one mantissa, and in this way we can know which grid a floating-point number should belong to according to its binary format. Therefore, we use a value $\mathbf{G'}$, which is smaller than $\mathbf{G}$ and also satisfies Lemma 1, as the grid size for each segment. As a result, we define $\mathbf{G'}$ as below:

$$G' = 1.0 \times 2^k \times \varepsilon' \times \theta.$$

Here ε' is the result of ε keeping one valid number in binary, and if we choose a θ which has only one mantissa in binary format, we can know that $\mathbf{G'}$ also has only one mantissa in binary format.

We can determine that the size of the subtable of Seg_k is $\frac{the\ size\ of\ Seg_k}{\mathbf{G'}}$. To better illustrate this, we calculate it separately as follows:

1) The size of Seg_k is $1.111... \times 2^k - 1.0 \times 2^k = 1.0 \times 2^k$.
2) Convert the ε to binary representation, and keep one valid number (e.g., if $\varepsilon = 0.01$ in decimal, $\varepsilon' = 0.0000001$ in binary and could be present by 2^{-7}).
3) Choose a power of 2 that less than 1.0 as θ, such as $1/2, 1/4, 1/8$ etc., denoted as 2^{-p}.

Thus, $G' = 1.0 \times 2^k \times \varepsilon' \times 2^{-p}$, and the size of the subtable for Table **T1** is $2^p/\varepsilon'$. Obviously the size is determined by θ and ε. For example, if $\theta = 0.25$ and $\varepsilon = 0.01$, the size of the subtable is $2^2 \times 2^7 = 2^9 = 512$. Therefore, the scale of the subtable will be in control. On the other hand, the count of the subtable in Table **T1** is determined by **L** (i.e., the range of quantization code **M**), which is about 10 in general.

Overall, by setting the different grid sizes for different segments (i.e., the subtables), we can control the total size of Table **T1** at an acceptable level. Generally, in our final implementation and evaluation in Section 5, the sizes of Table **T1** are only about 131 KB, 884 KB, 2,848 KB, and 4,608 KB when ε = 0.1, 0.01, 0.001, and 0.0001, respectively. These memory footprints are ignorable for HPC servers.

4.3 The Adaptive Parameter Selection in Model B

As mentioned in Section 4.2, the value of θ determines the size of the table, in which we denote θ as 2^{-p} and p is called **'plus_bits'** here. Now we discuss the value selection of plus_bits, which determines the range covered by all the $\mathbf{PI}$'s and the table size (affecting the compression ratio and rate, respectively).

As shown in Fig. 6, we compare the results of different plus_bits on the NYX dataset to study its influence on the performance of SZ_P using Model B. From Fig. 6c, we see that the compression ratios are often in direct proportion to the plus_bits. As described above, plus_bits decides the intersection size of neighboring $\mathbf{PI}$'s: a smaller 'plus_bits' leads to a smaller intersection. Therefore, if we use a bigger plus_bits, $\mathbf{PI}$'s can cover a larger range with smaller intersections, resulting in fewer unpredictable values and thus higher compression ratios. Tuning of plus_bits is more critical to the compression use-case with large error bound than with small error bound. On the other hand, a bigger plus_bits also will lead to a bigger precomputed table. The table size doubles every time the value of plus_bits increases by one. A bigger table size will probably result in more cache misses, which would cause worse performance in table lookup. Fig. 6a shows that the compression rate differs with different values of plus_bits. The compression rate is a decreasing function of the plus_bits value, which is reasonable. The difference is insignificant in lower accuracy requirements, such as 1e-1, 1e-2, and 1e-3. In use cases demanding small error bound, such as 1e-4, 1e-5, and 1e-6, a bigger plus_bits results in a clearly different time cost. Hence, if the user accepts compression with a relatively large error bound, we do not need to choose a large plus_bits. If a small error bound is required, we should choose a small plus_bits, in order to get a high compression rate. From Fig. 6b, we also observe negligible difference on decompression rate brought by the value of plus_bits.

Based on this analysis, we can decide the best-fit plus_bits according to the range of L and the user-specified error bound, which determines the size of the table. If the table size is fairly large, the lookup performance can easily be degraded significantly, leading to a low compression rate. In this situation, using a small plus_bits can effectively mitigate this issue, obtaining a better compression rate with an insignificant loss of the compression ratio. If the table size is small, it is better to use a bigger plus_bits to get a higher compression ratio with an insignificant loss of the compression rate.

Algorithm 1. Adaptive selection of the plus_bits p.

```
Input: pointwise relative error bound $\varepsilon$, interval capacity $L$
Output: the parameter $p$ to define $\theta$;
 1: $p = 0$
 2: coefficient_L = 1
 3: for i=1 to 1024 do
 4:    if $L \gg$ i == 0 then
 5:       coefficient_L = $2^{i-15}$
 6:    end if
 7: end for
 8: expo_$\varepsilon$ = exponent of $\varepsilon$
 9: coefficient_$\varepsilon$ = $2^{-expo\_\varepsilon}$
10: coefficient_total = coefficient_L * coefficient_$\varepsilon$
11: if coefficient_total $< 2^7$ then
12:    $p = 3$
13: else if coefficient_total $\leq 2^{10}$ then
14:    $p = 2$
15: else if coefficient_total $< 2^{16}$ then
16:    $p = 1$
17: else
18:    $p = 0$
19: end if
```

In addition, not only is the table size relative to plus_bits and ε, but it also involves the user-specified parameter L. Therefore, we also need to take L into consideration carefully. To this end, we propose an adaptive method, as shown in Algorithm 1, to make use of these features for best performance by choosing a reasonable plus_bits under different provided conditions. We set the table size with ε = 1e-1 and L = 32768 as the base size. ①, When we get a series of user-specified parameters as configuration, we estimate the table size in this situation, in which we compare the differences of ε and L, predict how many times each parameter will enlarge the table size, and give a total result. ②, we calculate the $coefficient\_total$ with these two sizes to predict how many times the table size is larger than the base size. ③, According to the results of a series of experiments, as shown in Fig. 6a, we note that the performance will be significantly slowed down in error bound 1e-4 using different plus_bits. The reason is the overlarge table size. According to Fig. 6c, when plus_bits is larger than 3, the compression ratios differ little with different plus_bits. Combining these results, we define three boundaries for $coefficient\_total$ to set a suitable $plus\_bits$ for achieving a good trade-off between the compression ratio and compression rate.

4.4 Implementation Details about Model B

In this subsection, we provide the implementation details of SZ_P in Algorithms 2 and 3 with pseudocode descriptions.

First, we build an Table **T2**, saving all the $(1+\varepsilon)^{M(2-\theta)}$, and an inverse *table* **T1** which is used to find a M for a float f, as described in the preceding subsection. To save subtables from different segments for Table **T1**, we design the following data structures:

```
struct TopTable{
  uint16_t topIdx; /*the biggest exponent in V*/
  uint16_t btmIdx; /*the smallest exponent in V*/
  int bits; /*define the relative size of grid size*/
  struct SubTable * subTablePtr;
  /*sub-table array*/
}
struct SubTable{
 uint32_t * grids; /*array of grid to M*/
}
```

For the TopTable in Table **T1**, it should save the biggest and smallest exponent of $(1+\varepsilon)^{M(2-\theta)}$ as topIdx and btmIdx, as well as the pointer subTablePtr to its subtables. Here bits is the result of $-(exponent\ of\ \varepsilon')+p$, where p is the plus_bits parameter mentioned in the preceding subsection regarding how to choose the value of θ adaptively.

Based on these data structures (also as shown in Algorithm 1), we first build Table **T2** and then build Table **T1** according to **T2**. The size of **T1** is determined by the amount of different exponents in the Table **T2**. The size of the subtables of **T1** is determined as 2^{bits}. We scan the grids in **T1** and calculate each grid's bottom and top boundaries to determine that which **PI**$'$ the processed grid should belong to. The building of Table **T1** is completed as all the grids are processed.

Algorithm 2 describes the point-by-point processing stage in SZ_P using Model B. We get the error-controlled quantization code M by looking up Table **T1**, if the data $Ds[i]$ is a predictable value. Thus, SZ_P avoids many logarithmic transformation operations in SZ_T while achieving nearly the same compression efficiency on scientific data. Fig. 7 shows how the tables are used. When we get a floating-point number f, we acquire its exponent part with bit operations to find out which SubTable we should use. Next, we acquire its several bits from its mantissa to lookup in the SubTable, and get the M which f should map to. The number of bits acquired from mantissa is decided by the user-specified precision and value of θ.

Table **T2** can also be used in decompression since decompression is an inverse process of compression. After using Zstd as well as Huffman decoding to get the quantization factor array *M[]*, the decompression process also requires a

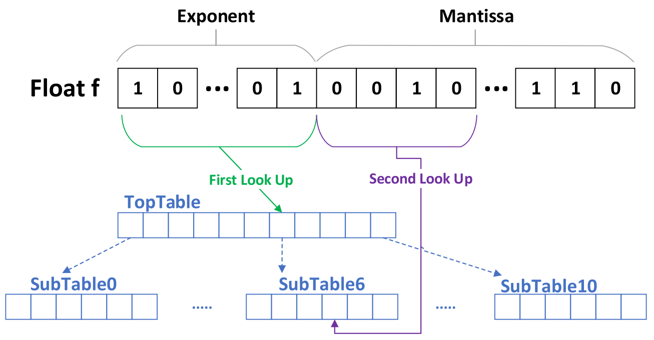


Fig. 7. An general example for looking up tables to get M with a specified f.

point-by-point processing stage, in which Table **T2** can be used to restore data. SZ_P uses quantization factors as indices to look up Table **T2** to get the floating-point values, which have been precomputed according to Equation (7). These floating-point values are not the values of original data but the quotients of neighboring points, which can be further used to restore the original data by using several simple multiply operations.

Algorithm 2. Building Tables *T1* and *T2* using Model B.

```
Input: pointwise relative error bound $\varepsilon$, interval capacity $L$,
       and the parameter $p$ to define $\theta$;
Output: table $T2$ ($M \to f$) and table $T1$ ($f \to M$);
 1: T2={0};
 2: $\theta = 2^{-p}$;
 3: for i=0 to L-1 do
 4:    T2[i] = $(1+\varepsilon)^{(i-L/2)\times(2-\theta)}$;
 5: end for
 6: TopTable T1;
 7: T1.btmIdx = T2[0]'s exponent; T1.topIdx = T2[L-1]'s exponent;
 8: T1.bits = -($\varepsilon$'s exponent) + $p$;
 9: index = 0; state = false;
10: subBoundary = (1 $\ll$ T1.bits) - 1;   /*to define the last index
    of sub-table*/
11: for i=0 to T1.topIdx - T1.btmIdx do
12:    for j=0 to subBoundary do
13:       PI'top = T2[index] / (1-$\varepsilon$);
14:       PI'btm = T2[index] / (1+$\varepsilon$);
15:       gridBtm =((i+T1.btmIdx) $\ll$ 23) + (j $\ll$ (23-T1.bits));
16:       gridTop = ((i+T1.btmIdx) $\ll$ 23) + ((j+1) $\ll$ (23-T1.bits));
17:       /*In IEEE 754 format, 32bit float has 23 bits mantissa, i +
          T1.btmIndex is gridTop's exponent and j is its mantissa.*/
18:       if gridTop $<$ PI'top && gridBtm $>$ PI'btm then
19:          T1.subTablePtr[i].grids[j] = index;
20:          state = true;
21:       else if index $<$ L-1 and state == true then
22:          index ++;
23:          T1.subTablePtr[i].grids[j] = index;
24:       else
25:          T1.subTablePtr[i].grids[j] = 0;
26:       end if
27:    end for
28: end for
```

In addition, we can find that the tables are used to build a map between a floating-point numbers f and integers M, and the map is determined by the required error bound ε and the value of θ; it has nothing to do with specific dataset. Therefore, the tables are deterministic, thus they can be precomputed and saved into files by serialiazation and reused for different datasets.

Algorithm 3. Point-by-point processing stage in SZ_P.

```
Input: the dataset $Ds\{\}$, a user-specified point-wise relative
       error bound $\varepsilon$ and interval capacity $L$;
Output: compressed data stream $M$ and the unpredicted bytes;
 1: Build Tables T1 and T2;
 2: M = {0};
 3: pred = 0;
 4: Process Ds[0] as an unpredicted float in SZ, pred = Ds[0]'; /*Ds
    [0]' is truncated from Ds[0] according to $\varepsilon$ as SZ does [22].*/
 5: length = size of Ds;
 6: for i=1 to length-1 do
 7:    ratio = Ds[i]/pred;
 8:    index = 0;
 9:    expoIdx = ratio's exponent;
10:    if expoIdx $>$ T1.btmIdx && expoIndex $<$ T1.topIdx then;
11:       mantiIdx = ratio's mantissa $\gg$ (23-T1.bits);
12:       subtable = T1.subTablePtr[expoIdx-T1.btmIdx]
13:       index = subtable.grids[mantiIdx];
14:    end if
15:    if index != 0 then
16:       M[i] = index;
17:       pred = pred * T2[index];
18:    else
19:       M[i] = 0;
20:       process Ds[i] as unpredicted in SZ, pred = Ds[i]'; /*Ds[0]'
          is truncated from Ds[0] according to $\varepsilon$ as SZ does [22].*/
21:    end if
22: end for
```

4.5 Optimizing Huffman Decoding

As discussed in Section 3, Huffman encoding/decoding is a critical step in SZ. In SZ_P, we optimize the Huffman decoding performance in particular. Generally, Huffman decoding parses the bit stream according to a Huffman tree, which means the decoding process acts as a state machine (bit by bit processing). With small error bound, quantization factor values will no longer converge, but scatter, leading to a longer average Huffman code length, thus longer decoding time (see results in Fig. 2b). Based on our observation, there are many repeated calculations in Huffman decoding. Traversing always begins from the root of the Huffman tree, and different Huffman codes with the same prefix will have the same or partially overlapped traverse path. Many repeated traversing operations can be avoided if we can also design such a precomputation-based mechanism for Huffman decoding.

Inspired by our precomputed tables for logarithmic transformation, we process multiple bits as a prefix at one time, instead of processing bit by bit. As a result, we also design the precomputation-based tables for Huffman decoding in SZ_P as shown in Fig. 8. We describe the tables in details as below.

- *Node Table*: It records *prefix bits* $\to$ *Huffman node address* (or called subtree). Here we configure the length of prefix bits as a fixed number Y, and we construct the *Node Table* for all Y prefix bits in the Huffman tree. Fig. 8 provides an example of this

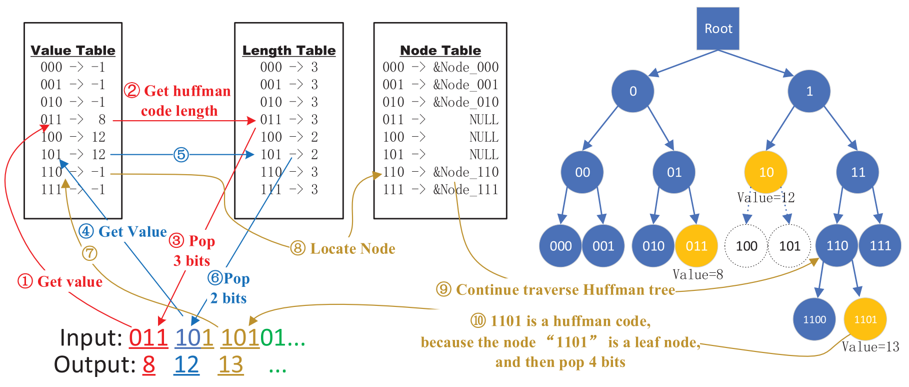


Fig. 8. Example of building precomputation-based tables to accelerate Huffman decoding in SZ_P.

Node Table (Y=3 in this example), which records all possibilities of nodes mapping to 3 prefix bits in the Huffman tree. Thus, by looking up this table, we can directly read 3 bits each time and access the corresponding node, avoiding the traversal cost from level 0 to level 3 in the Huffman tree.

- *Value Table*: It records *prefix bits* $\rightarrow$ *a possible Huffman code*. To build this table, we traverse all the Y prefix bits in Huffman tree; and if we find a leaf node in the traversal, we record the value of the leaf node in the entry mapping to the corresponding prefix, and otherwise record *-1*.
- *Length Table*: It records *prefix bits* $\rightarrow$ *the length of a Huffman code*. The reason we construct this table is that Huffman codes are variable lengths such that some Huffman codes can be shorter than Y.

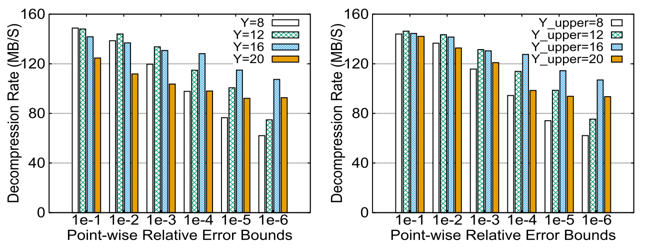
In the example in Fig. 8, we process 3 bits each time (Y=3). For the first 3 bits '011', we first look up the *Value Table* to obtain '011' $\rightarrow$ '8', knowing that it may be a Huffman code and its corresponding value is 8. We further get its *length*=3 by looking up the *Length Table*. As a result, we output the value '8' for the prefix '011', and move the bit stream pointer 3 bits forward accordingly. For the next 3 bits '101', we look up the *Value Table* to obtain '101' $\rightarrow$ '12', and we then look up the *Length Table* to obtain *length*=2. Thus, we output the value '12' for the prefix '10'. Because *length*=2, we move the bit stream pointer 2 bits forward. The last 1 (of '101') and its next 2 bits '10' are treated as the next 3 bits '110'. We find '110' is not a Huffman code in the *Value Table* ('110' $\rightarrow$ '-1'), so we get the address of Huffman tree's node '110' from the *Node Table* and traverse the Huffman tree according to the following (input) bits to get a longer Huffman code.

With these precomputed tables, we can significantly accelerate the decompression process of SZ_P by reducing tree traversing operations during Huffman decoding. Note that the size of the precomputed table is determined by a key parameter Y, which will affect the final performance. Specifically, the total size of the three precomputed tables can be calculated according to the length of the prefix bits (i.e., the value Y) as: $2^Y \times (4+2+8)$. If the table size is too large, it will also cause lots of cache misses during Huffman decoding, thus decreasing the overall decompression speed. We discuss this issue in detail as follows.

As shown in Fig. 9a, the best choice of Y is also different for different user-specified error bound. With error bounds 1e-1 and 1e-2, the average length of Huffman codes is short, and we do not need a large table. In the smaller error bounds cases such as 1e-5 and 1e-6, the average length of Huffman codes is long, such that a small table cannot make full use of this kind of schema. We can observe that the value of Y is deeply involved in the decompression rate. It is closely related to the average lengths of Huffman codes, which is decided by the user-specified error bound and the distribution of data and is hard to predict only in decompression processing.

Therefore, we also propose an adaptive method to select a suitable parameter Y for our Huffman decoding. We record 95 percent of the lengths of Huffman codes, which reflect the lengths of most Huffman codes, in the compression output file. During the decompression processing, we will use this record as the value of Y to adapt different situations on different datasets.

The table size should not be too large because the size of cache in computer is limited and the bigger table will lead to many cache misses. Thus, the value of Y also needs an upper limit Y_{upper}. As shown in Fig. 9b, we run a series of experiments on NYX dataset with plus_bits = 3 to show the difference in the decompression rate with different Y_{upper}. If Y_{upper} is too small, the precomputed table cannot fully realize their potential. On the other hand, if Y_{upper} is too large, cache misses will also make the compressor slower. With error bounds of 1e-1, 1e-2, and 1e-3, different Y_{upper}s may lead to similar results. However, we observe that in error bounds of 1e-4, 1e-5, and 1e-6, the decompression rate increases as Y_{upper} increases from 8 to 16. However, for the situation of Y_{upper} = 20, its decompression rate is much slower with error bounds of 1e-4 and 1e-5. With an error bound of 1e-6 that may lead to a long average huffman code length, Y_{upper} = 20 seems able to make full use of this



(a) Using fixed value Y for Huffman decoding (b) Using the adaptive method with Y and different Y_{upper}

Fig. 9. Performance comparisons of SZ_P on NYX dataset using different Y (with adaptive selection of '$plus\_bits$').

precomputed mechanism and achieves better performance than with Y_{upper} = 16, but the result is contrary to what we expected. The reason is that the tables constructed by this mechanism are too large, leading to many cache misses in Huffman decoding.

Based on this observation, Y_{upper} = 16 is always a reasonable setting, which means that each table has 2^{16} entries at most. If the lengths of 95 percent of Huffman codes during the compression processing are larger than Y_{upper}, we set Y = Y_{upper} as the prefix length; otherwise, we follow the results of the statistics summary. The size of the three tables is only ~851 KB (using Y_{upper}=16) at most in our design for Huffman decoding, which is also ignorable for HPC servers.

4.6 Discussions

In summary, the key points of our idea in designing SZ_P are as follows.

- Since the predicted values (X'_i) are always expected to be close to the real values $(X_i)(i.e., \frac{X_i}{X'_i}$ is always distributed in a zone close to 1.0), we can get a new method with a precomputed table, which is mathematically equivalent to the logarithmic transformation used in SZ_T; and the size of the precomputed table can be well controlled.
- By our careful design of Model B, we make the error caused by the table look-up always smaller than the error bound ε, which means that the accuracy of compressing data by SZ_P could be well controlled.

Therefore, our optimization includes two parts: the log transformation and the calculation of the quantization factors (i.e., the relationships of the neighboring floats). We avoid the logarithmic transformation in SZ_T and optimized the quantization process with table lookup operations. That is the key reason we speed up the pointwise relative error bounded lossy compression significantly. Because ZFP [31] does not have the quantization factor calculation procedure, our current design cannot be directly applied to ZFP. More specifically, ZFP saves the transformation results of the original values instead of the derived values of the neighboring points.

5 Evaluation

In this section, we compare our approach (denoted by SZ_P) with two state-of-the-art methods, SZ_T and ZFP_T, which were proposed in the literature [20]. We chose ZFP_T because it is an improved version in terms of the original ZFP by leveraging logarithm transformation for point-wise relative error bound. Our prior work [20] shows that ZFP_T exhibits much better results than the original ZFP. Compression and decompression rate, ratio, and data fidelity are all used to assess lossy compressors quantitatively.

5.1 Experimental Setup

We conduct our tests on a server with two Intel Xeon Gold 6130 processors running at 2.1 GHz and a total of 128 GB of memory. We perform the evaluation with as many datasets as possible downloaded from the scientific data reduction benchmark (sdrbench) [32], including HACC cosmology simulation (1D), NYX cosmology simulation (3D), Hurricane ISABEL simulation (3D), CESM-ATM climate simulation (2D), EXAALT moledular dynamic simulation (1D), Miranda turbulence simulation (3D), S3D combustion simulation (3D), and SCALE-LETKF climate simulation(3D). The sizes of these eight datasets are 6.3 GB, 3.1 GB, 1.9 GB, 2.0 GB, 33 MB, 1.0 GB, 26 GB, and 6.4 GB per snapshot for each application, respectively. Other datasets on the sdrbench either are unstructured grid datasets (unable to be represented by regular mesh grid, such as XGC and Brown samples) or require absolute error bound instead of pointwise relative error bound focused by our paper (such as EXAFEL).

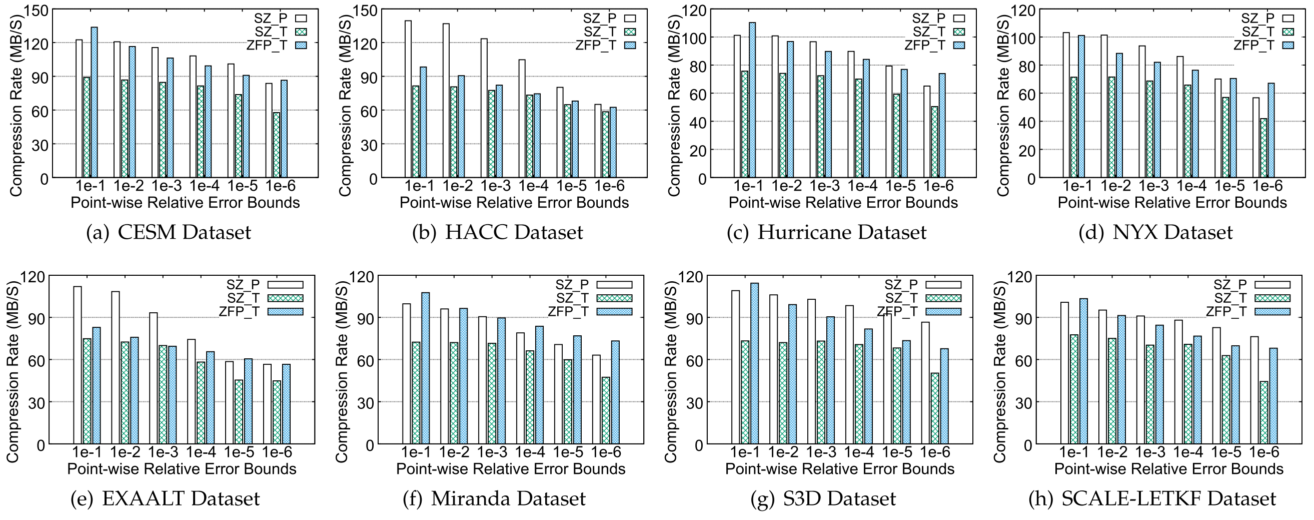
The data fidelity of lossy compression approaches is measured with multiple metrics, including the max pointwise relative error (MAX E), which shows whether a method is respecting the error bound; MRE (mean relative error; the smaller the better), which shows the mean relative difference between original values and the decompressed values; and RMSRE (root mean squared relative error; the smaller the better), which measures the degree of dispersion of relative error.

Compression rate and decompression rate indicate the throughput of compression and decompression. For compression and decompression rate, we run each experiment five times to calculate the average. Since each application involves many fields, each in a data file, we use the aggregated file size divided by the total compression or decompression time to calculate the rate. The compression ratio is the ratio of the original data size to the compressed size.

5.2 Compression Rate and Decompression Rate

In this section, we compare compression and decompression rates for different approaches. For SZ_P, the time cost of building tables is not counted in the total time cost. The reason is that the tables are determined by the user-specified error bounds and θ value, so they are actually computed priori and can reused by different datasets in practice, as discussed in Section 4.4. Figs. 10 and 11 present the compression rate and decompression rate of SZ_P, SZ_T and ZFP_T on the eight datasets, respectively. Generally, SZ_T has the lowest compression rate among the three approaches, because of its logarithmic transformation and floating-point quantization time cost. The compression rate is a particular advantage of ZFP_T, since the implementation of ZFP itself has been optimized for the speed purpose. As observed in the figure, ZFP_T exhibits a 20 ~30 percent higher compression rate than SZ_T. In most cases, SZ_P's speed is about $1.2\sim 1.5\times$ as fast as that of SZ_T on compression, which is attributed to the performance gain of our new table lookup method. As Fig. 11 demonstrates, the decompression rate of SZ_P is also about $1.3\sim 3.0\times$ as high as that of SZ_T.

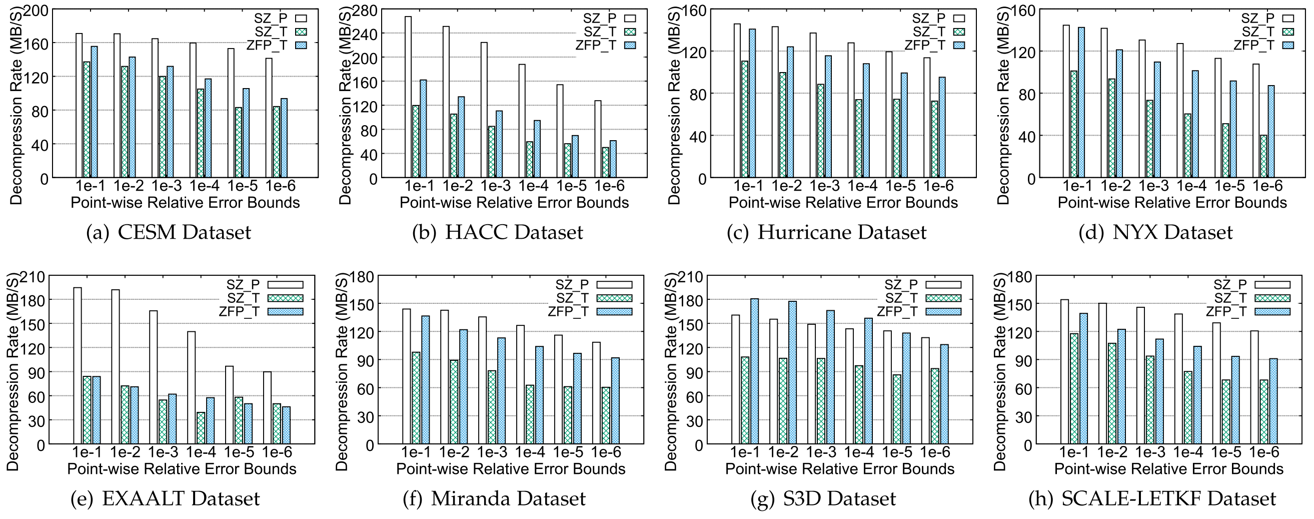
From Fig. 11, we can also observe that the decompression performance gain of SZ_P over other lossy compressors differs with error bound, also depending on the datasets. Specifically, SZ_P's decompression rate is equivalent to or even higher than ZFP_T's decompression rate in most of cases. The key reason is that the exponential transformation and Huffman decoding takes the major portion (more than $\frac{2}{3}$) of the total decompression time for SZ, as can be confirmed in Fig. 2b, and SZ_P greatly reduces the time cost of these two steps in decompression. Note that SZ_P has the highest decompressing rate on the HACC dataset. This is because



(a) CESM Dataset (b) HACC Dataset (c) Hurricane Dataset (d) NYX Dataset

(e) EXAALT Dataset (f) Miranda Dataset (g) S3D Dataset (h) SCALE-LETKF Dataset

Fig. 10. Compression rate on given point relative error bound.



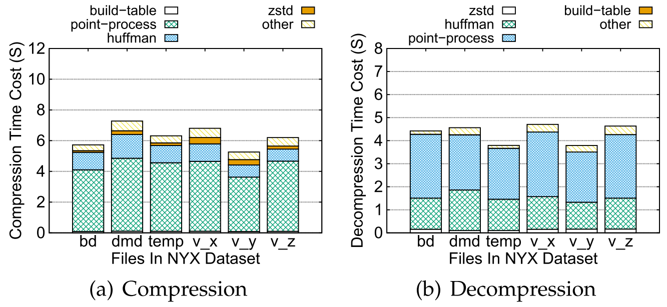
(a) CESM Dataset (b) HACC Dataset (c) Hurricane Dataset (d) NYX Dataset

(e) EXAALT Dataset (f) Miranda Dataset (g) S3D Dataset (h) SCALE-LETKF Dataset

Fig. 11. Decompression rate on given pointwise relative error bound.

HACC has the lowest compression ratio and thus needs longer time for Huffman decoding, whereas SZ_P greatly accelerates Huffman decoding as discussed earlier.

We present breakdown of SZ_P time consumption in Fig. 12. From the figure we can see that SZ_P eliminates the time cost on logarithmic transformation in comparison with SZ_T (see Fig. 2 in Section 3). This explains why SZ_P achieves much higher compression and decompression rates than does SZ_T. Fig. 12a also suggests that the time cost in building tables of our precomputation-based mechanism occupies less than 5 percent of the total time. In addition, the constructed tables in SZ_P are independent of the dataset size, which means this overhead can be amortized and hence is negligible when compressing large datasets. From Fig. 12b, we can see that SZ_P also greatly reduces the time cost of Huffman decoding, achieving a much higher decompression rate (see Fig. 11).



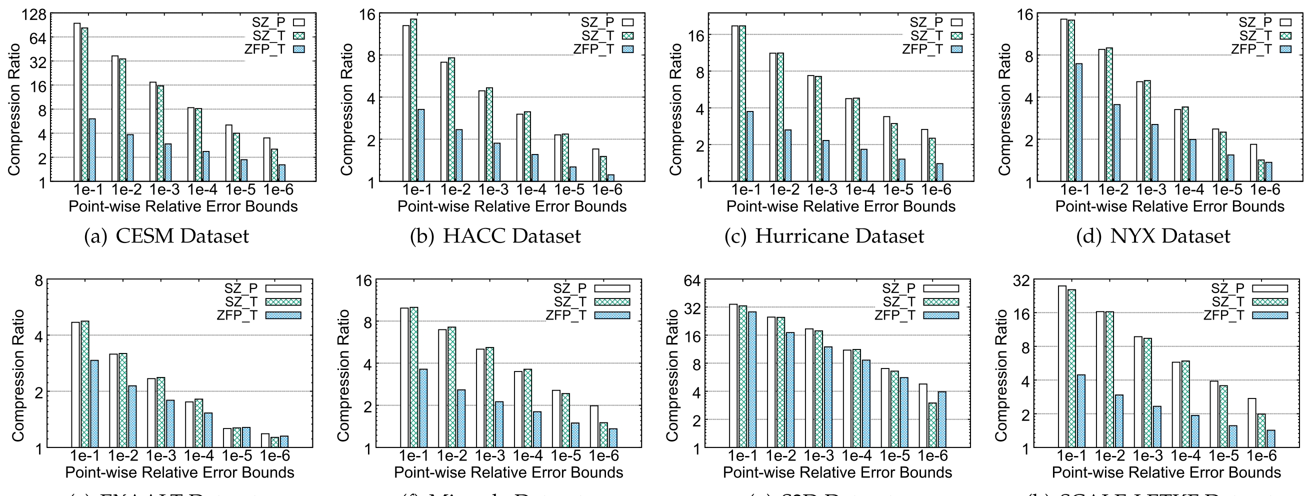
(a) Compression (b) Decompression

Fig. 12. Breakdown of the compression time for SZ_P using 1E-2 pointwise relative error bound on NYX dataset.

5.3 Compression Ratio

Fig. 13 shows the compression ratio of ZFP_T, SZ_T, and SZ_P with the six commonly-used pointwise error bounds (0.1, 0.01, 0.001, ..., 0.000001) on the eight datasets. From this figure, we can see that SZ_P has compression ratios similar to those of SZ_T, which are often much higher (even up to one order of magnitude in some cases) than those of ZFP_T. We explain the key reasons as follows.

Unlike SZ_T whose interval size covered by a quantization code is twice as large as the error bound (i.e., exactly 2ε), SZ_P has a smaller interval size of quantization code, which is $(2-\theta)\varepsilon$, which thus slightly decreases the compression ratio.



(a) CESM Dataset (b) HACC Dataset (c) Hurricane Dataset (d) NYX Dataset

(e) EXAALT Dataset (f) Miranda Dataset (g) S3D Dataset (h) SCALE-LETKF Dataset

Fig. 13. Compression ratio on given pointwise relative error bound.

The degree of decrease depends on the θ value: using a smaller θ would achieve a smaller decrease, but the table size would be increased (because θ determines the grid size). As discussed in Section 4.3, we use an adaptive method in choosing θ for SZ_P to achieve a tradeoff between compression rate and compression ratio. SZ_P achieves a similar compression ratio as SZ_T.

On the other hand, we note that SZ_P achieves a higher compression ratio (about 1.20×) on small error bounds, such as 1e-5 and 1e-6. As our prior work [20] indicated, in order to avoid the round-off error caused by logarithmic transformation for compression and its inverse transformation for decompression, a correction (i.e., a subtraction) must be introduced for the error bound requirement. When the error bound is relatively large, such as 0.1, the subtrahend is usually much smaller than the tolerable accuracy loss; the correction on user-specified error bound would be not obvious, and it does not affect the compression ratio. However, in the case of setting extremely small error bounds, the derived error bound value can also be very small, so the round-off value could be at the same order of magnitude as the error bound.

As a result, subtracting the round-off value from the user-specified error bound can dramatically reduce the error bound, causing an impact similar to the situation with tight error bounds, leading to a lower compression ratio. Our approach, SZ_P, does not have the preproccessing stage (i.e., X_i are transformed to $\log(X_i)$), and thus it does not need to do the error bound correction. Therefore, SZ_P can avoid the compression ratio decrease caused by error bound correction, having it achieve better compression ratios than does SZ_T in low-error-bound cases. In addition, we note that the compression and decompression rates of SZ_P and SZ_T with low error bounds are nearly the same. The reason is that compared with SZ_P, the tighter error bounds of SZ_T would lead to a lower prediction accuracy (or a higher amount of unpredictable data) in the point-to-point processing stage of SZ, such that many zero bits (M[i]=0) would be generated, as shown in Algorithm 2, resulting in less time spent for Huffman coding.

5.4 Quality of Data From Decompression

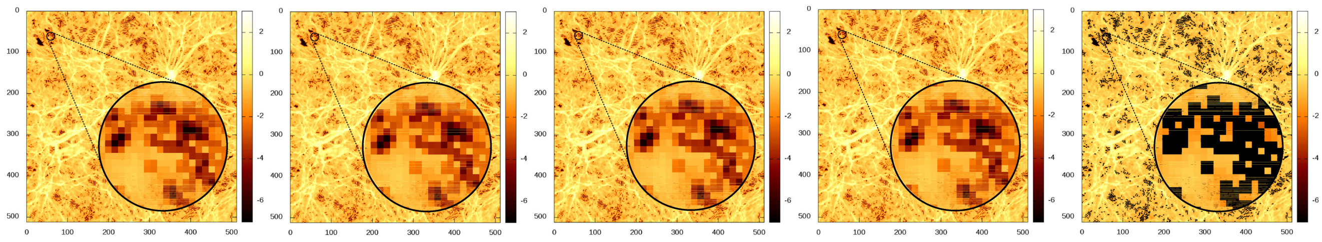
In this subsection, we evaluate the data distortion of SZ_P and compare it with that of other compressors. Since they are evaluated in previous work SZ_T [20], we also select dark_matter_density, velocity_x fields in NYX, as well as a temperature field, to evaluate data fidelity of each approach. Dark_matter_density is a typical use case for pointwise relative error in which a large majority of values are distributed in [0, 1] and the rest are in [1, 1.378E+4]. The velocity_x includes large values with positive/negative signs indicating directions. Maximum pointwise relative error, RMSRE, MRE, and compression ratio are evaluated. Model A (denoted by SZ_P′) is also evaluated to demonstrate that it cannot strictly respect the error bound.

Table 2 shows the data quality results of SZ_P, SZ_T, SZ_P′, and ZFP_T with four most widely used pointwise error bounds (0.1, 0.01, 0.001, 0.0001). SZ_P′ does not strictly respect the error bound, achieving about 2 times the error bound in some cases. It also does not perform well on MRE and RMSRE. Overall, the data distortion of SZ_P′ is higher than that of either SZ_T′ or SZ_P due to its design limitation in error control.

On the other hand, SZ_P (using Model B), an advanced version of SZ_P′ (using Model A), strictly respects the error bound as SZ_T does and thus works better than SZ_P′ and achieves data accuracy similar to that of SZ_T. Benefiting from its smaller interval size of quantization code ($(2-\theta)\theta$ in SZ_P and 2θ in SZ_T), SZ_P achieves slightly better accuracy than does SZ_T, but it also suffers from a little reduction in the compression ratio for the same reason, causing few more unpredictable data points. However, with these slight differences, as Fig. 14 shows, in practice SZ_P and SZ_T achieve almost the same decompressed data quality. The data loss of SZ_P′ is about 2 times higher than that of SZ_P, but for accuracy tolerant applications such as visualization, the SZ_P′ performs as well as SZ_P and SZ_T. In comparison, the visualization effect of ZFP_T decompressed data is obviously different from the original data; thus, the fidelity loss caused by ZFP_T has an obvious impact at the application level.

TABLE 2
Pointwise Relative Error Bound on 3 Representative Fields in NYX

pwr	Type	dark_matter_density				Temperature				Velocity_x			
		MAX E	RMSRE	MRE	CR	MAX E	RMSRE	MRE	CR	MAX E	RMSRE	MRE	CR
1E-01	SZ_P′	1.52E-01	4.71E-02	2.65E-02	6.25	1.50E-01	8.26E-02	5.72E-02	23.48	1.52E-01	8.26E-02	5.54E-02	18.83
	SZ_T	1.00E-01	3.14E-02	2.20E-02	6.19	1.00E-01	5.51E-02	4.76E-02	24.97	1.00E-01	5.50E-02	4.62E-02	19.85
	SZ_P	9.99E-02	2.98E-02	2.08E-02	6.13	9.97E-02	5.32E-02	4.51E-02	25.97	9.97E-02	5.22E-02	4.50E-02	21.90
	ZFP_T	5.07E-02	3.73E-03	2.84E-03	3.32	4.80E-02	3.49E-03	2.72E-03	18.40	5.17E-02	2.83E-03	2.13E-03	14.00
1E-02	SZ_P′	1.75E-02	5.56E-03	2.99E-03	3.85	1.70E-02	9.76E-03	6.44E-03	13.46	1.70E-02	9.74E-03	6.02E-03	14.37
	SZ_T	1.00E-02	3.27E-03	2.30E-03	3.85	1.00E-02	5.73E-03	4.96E-03	14.06	1.00E-02	5.72E-03	4.75E-03	13.55
	SZ_P	1.00E-02	3.07E-03	2.16E-03	3.80	9.96E-03	5.39E-03	4.66E-03	13.21	9.96E-03	5.37E-03	4.64E-03	13.09
	ZFP_T	3.02E-03	2.33E-04	1.78E-04	2.35	3.33E-03	2.37E-04	1.83E-04	6.59	3.16E-03	2.34E-04	1.81E-04	5.21
1E-03	SZ_P′	1.96E-03	5.90E-04	3.25E-04	2.75	1.95E-03	9.89E-04	6.72E-04	6.75	1.95E-03	9.81E-04	6.68E-04	8.02
	SZ_T	9.97E-04	3.27E-04	2.30E-04	2.74	9.98E-04	5.75E-04	4.97E-04	6.61	9.98E-04	5.74E-04	4.74E-04	7.63
	SZ_P	1.00E-03	2.90E-04	2.03E-04	2.69	9.99E-04	5.09E-04	4.39E-04	6.30	9.99E-04	5.08E-04	4.38E-04	7.35
	ZFP_T	3.90E-04	2.92E-05	2.22E-05	1.92	3.95E-04	2.96E-05	2.28E-05	4.08	3.97E-04	2.96E-05	2.28E-05	3.50
1E-04	SZ_P′	1.60E-04	4.89E-05	2.97E-05	2.12	1.60E-04	7.73E-05	6.57E-05	3.93	1.60E-04	7.72E-05	5.86E-05	4.39
	SZ_T	9.80E-05	3.15E-05	2.81E-05	2.09	9.90E-05	5.65E-05	4.89E-05	3.92	9.90E-05	5.63E-05	4.86E-05	4.38
	SZ_P	1.00E-04	2.58E-05	1.76E-05	2.05	1.00E-04	4.59E-05	3.90E-05	3.74	1.00E-04	4.60E-05	3.90E-05	4.15
	ZFP_T	5.08E-05	3.65E-06	2.77E-06	1.63	4.99E-05	3.71E-06	2.86E-06	2.95	5.33E-05	3.73E-06	2.89E-06	2.63



(a) Original (b) SZ_P′ (Model A) (c) SZ_P (Model B) (d) SZ_T (e) ZFP_T

Fig. 14. Visualization of decompressed dark_matter_density dataset (slice 200) at the compression ratio of 2.75.

Table 2 shows that with the same relative error bound, ZFP_T achieves relatively high data qualities but low compression ratios. The reason is that ZFP is difficult to control the data quality accurately compared with the target user-set error bound because of the overpreserved error estimation design [20].

6 Conclusion

In this paper, we propose an effective approach to accelerate the pointwise relative-error-bounded lossy compression of SZ, leading to an optimal lossy compressor for users with respect to both compression ratio and compression rate in most cases. Our optimization strategy originates from an important observation that the logarithmic transformation and Huffman decoding are the performance bottlenecks in SZ. We develop a precomputation-based mechanism, called SZ_P, with the fast table-lookup methods for logarithmic transformation and most of traversing operations in Huffman decoding. This solution can improve the compression rate significantly in most cases. The key findings of our performance evaluation with eight well-known application datasets are as follows:

- Compression/decompression rate: SZ_P improves the compression rate by about 40 percent and the decompression rate by about 80 percent compared with SZ_T. It has performance comparable to or even higher than that of ZFP_T in most cases.
- Compression ratio: SZ_P and SZ_T have similar compression ratios, which are significantly higher (even up to one order of magnitude in some cases) than that of ZFP_T on the tested datasets.
- Respecting user-required error control: Similar to SZ_T, SZ_P can always respect user requirements on pointwise relative error bounds. SZ_P also has the same level of data distortion in RMSRE or MRE with SZ_T.

We have integrated our codes into the official SZ package (version 2.1) [33].

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and feedback on this work. This research was partly supported by National Key R&D Program of China under Grant No. 2018YFB1003800 and No. 2018YFB1003805, NSFC No. 61972441, No. 61832004, No. 61972112, No. 61672186, and No. 61872110, the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2018WNLOKF008, Key R&D Program for Guangdong Province under Grant No.2019B010136001, the Shenzhen Science and Technology Program under Grant No. JCYJ20170413105929681 and No. JCYJ20170811161545863. The material was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357, and also by National Science Foundation under Grant No. 1619253. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the author(s) and do not necessarily reflect the views of the funding agencies. The preliminary version of this article appeared in the proceedings of IEEE MSST 2019. X. Zou and T. Lu equally contributed to this work.

References

[1] T. Lu *et al.*, "Canopus: A paradigm shift towards elastic extreme-scale data analytics on HPC storage," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2017, pp. 58–69.

[2] D. Tao *et al.*, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 1129–1139.

[3] W. Austin, G. Ballard, and T. G. Kolda, "Parallel tensor compression for large-scale scientific data," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 912–922.

[4] I. Foster, "Computing just what you need: Online data analysis and reduction at extreme scales," in *Proc. Eur. Conf. Parallel Process.*, 2017, pp 3–19.

[5] D. Ghoshal and L. Ramakrishnan, "MaDaTS: Managing data on tiered storage for scientific workflows," in *Proc. HPDC*, 2017, pp. 41–52.

[6] B. Dong *et al.*, "ArrayUDF: User-defined scientific data analysis on arrays," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 53–64.

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

[8] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp.1681–1710, Sep. 2016.

[9] D. Meister *et al.*, "A study on data deduplication in HPC storage systems," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.

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

[11] N. Sasaki *et al.*, "Exploration of lossy compression for application-level checkpoint/restart," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2015, pp. 914–922.

[12] T. Lu *et al.*, "Understanding and modeling lossy compression schemes on HPC scientific data," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2018, pp. 348–357.

[13] J. Kunkel *et al.*, "Toward decoupling the selection of compression algorithms from quality constraints," in *Proc. Int. Conf. High Perform. Comput.*, 2017, pp. 3–14.

[14] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Optimizing lossy compression rate-distortion from automatic online selection between SZ and ZFP," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 8, pp. 1857–1871, Aug. 2019.

[15] A. Poppick *et al.*, "A statistical analysis of compressed climate model data," in *Proc. DRBSD*, 2018, pp. 1–6.

[16] Q. Liu, "Hello ADIOS: The challenges and lessons of developing leadership class I/O frameworks," *Concurrency Comput.: Practice Experience*, vol. 26, no. 7, pp. 1453–1473, 2014.

[17] T. E. Fornek, "Advanced photon source upgrade project preliminary design report," 2017. [Online]. Available: https://www.aps.anl.gov/files/download/Aps-Upgrade/PDR.pdf

[18] G. Marcus *et al.*, "High fidelity start-to-end numerical particle simulations and performance studies for LCLS-II," in *Proc. 37th Int. Free Electron Laser Conf.*, 2015, pp. 342–346.

[19] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2674–2683, Dec 2014.

[20] X. Liang *et al.*, "An efficient transformation scheme for lossy data compression with point-wise relative error bound," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2018, pp. 179–189.

[21] S. Lakshminarasimhan *et al.*, "Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data," in *Proc. Eur. Conf. Parallel Process.*, 2011, pp 366–379.

[22] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2016, pp. 730–739.

[23] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Process.*, vol. 41, no. 12, pp. 3445–3462, Dec. 1993.

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

[25] X. He and P. Shi, "Monotone B-spline smoothing," *J. Amer. Statist. Assoc.*, vol. 93, no. 442, pp. 643–650, 1998.

[26] X. Liang *et al.*, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *Proc. IEEE Int. Conf. Big Data*, 2018, pp. 438–447.

[27] S. Di, D. Tao, X. Liang, and F. Cappello, "Efficient lossy compression for scientific data based on pointwise relative error bound," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 2, pp. 331–345, Feb. 2019.

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

[29] "Zstandard - fast real-time compression algorithm," 2016. [Online]. Available: https://github.com/facebook/zstd

[30] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[31] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *IEEE Trans. Vis. Comput. Graph.*, vol. 20, no. 12, pp. 2674–2683, Dec. 2014.

[32] "Sdrbench," [Online]. Available: https://sdrbench.github.io/, [Online].

[33] "SZ," [Online]. Available: https://github.com/disheng222/SZ, [Online].

Xiangyu Zou is currently working toward the PhD degree majoring in computer science with the Harbin Institute of Technology, Shenzhen, China. His research interests include data deduplication, lossy compression and storage systems. He has published several papers in major journals and conferences including Future Generation Computer Systems, MSST, and HPCC.

Tao Lu received the BS and MS degrees in computer science from the Huazhong University of Science and Technology, China, in 2009 and 2012, respectively, and the PhD degree in electrical and computer engineering from Virginia Commonwealth University, in 2016. He is currently a senior software engineer with Marvell Semiconductor Inc. His research interests include computer systems, virtualization and cloud computing, high-performance computing, and computer system security. He has published several papers in major international conferences including INFOCOM, IPDPS, and MASCOTS.

Wen Xia received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014. He is currently an associate professor with the School of Computer Science and Technology at Harbin Institute of Technology, Shenzhen. His research interests include data reduction, storage systems, and cloud storage. He has published more than 40 papers in major journals and conferences including *the IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Proceedings of the IEEE*, USENIX ATC, FAST, HotStorage, MSST, DCC, IPDPS, INFOCOM, ICDCS, etc.

Xuan Wang received the PhD degree in computer sciences from the Harbin Institute of Technology, Harbin, China, in 1997. He is currently a professor and dean of the School of Computer Science and Technology with the Harbin Institute of Technology, Shenzhen, China. His research interests include artificial intelligence, computer network security, computational linguistics, and computer vision. He has published more than 120 academic papers in major journals and conferences.

Weizhe Zhang (Senior Member, IEEE) is currently a professor with the School of Computer Science and Technology at Harbin Institute of Technology, China. His research interests include parallel computing, distributed computing, cloud and grid computing, and computer networks. He has published more than 100 papers in major journals and conferences including *IEEE Trans. on Computers*, *Future Generation Comp. Syst.*, *Journal of Supercomputing*, IPDPS, Cluster, and CIKM.

Haijun Zhang received the PhD degree from the Department of Electronic Engineering, City University of Hong Kong, in 2010. He is currently a professor of computer science with the Harbin Institute of Technology, Shenzhen, China. His current research interests include multimedia data mining, machine learning, and computational advertising. He is currently an associate editor of *Neurocomputing*, *Neural Computing and Applications*, and *Pattern Analysis and Applications*. He has published more than 50 academic papers in major journals and conference proceedings.

Sheng Di (Senior Member, IEEE) received the PhD degree from the University of Hong Kong, in 2011. He is currently an assistant computer scientist with Argonne National Laboratory. His research interests include resilience on high-performance computing (such as silent data corruption, optimization checkpoint model, and in situ data compression) and broad research topics on cloud computing (including optimization of resource allocation, cloud network topology, and prediction of cloud workload/hostload). He is currently working on multiple HPC projects, such as detection of silent data corruption, characterization of failures and faults for HPC systems, and optimization of multilevel checkpoint models.

Dingwen Tao received the bachelor's degree in mathematics from the University of Science and Technology of China, in 2013, and the PhD degree in computer science from the University of California, Riverside, in 2018. He is currently an assistant professor with the Department of Computer Science at the University of Alabama. Before joining the university as faculty, he interned at Pacific Northwest National Laboratory, Argonne National Laboratory, and Brookhaven National Laboratory. His research interests include high-performance computing, parallel and distributed systems, and big data analytics. Specifically, His research interests include scientific data reduction and management, resilience and fault tolerance, and large-scale machine learning and deep learning. He has published more than 30 peer-reviewed high-quality papers in prestigious HPC and Big Data conferences and journals, such as ACM ICS, HPDC, PPoPP, SC, IEEE BigData, CLUSTER, IPDPS, MSST, TPDS, IJHPCA, including two best paper awards.

Franck Cappello (Fellow, IEEE) is currently a program manager and a senior computer scientist with Argonne National Laboratory. From 2009, he held a joint position at INRIA and the University of Illinois at Urbana-Champaign, where he initiated and codirected the INRIAIllinois Joint Laboratory on Petascale Computing. Until 2008, he led a team at INRIA, where he initiated the XtremWeb (Desktop Grid) and MPICH-V (fault-tolerant MPI) projects. From 2003 to 2008, he initiated and directed the Grid5000 project, a nationwide computer science platform for research in large-scale distributed systems. He has authored more than 200 papers in the domains of fault tolerance, high-performance computing, desktop Grids, and Grids and contributed to more than 70 program committees. He is an editorial board member of the *International Journal on Grid Computing*, *Journal of Grid and Utility Computing*, and *Journal of Cluster Computing*. He is the recipient of the 2018 IEEE TCPP Outstanding Service Award.