Lossy Scientific Data Compression With SPERR

Shaomeng Li
Nat'l Center for Atmospheric Research

Peter Lindstrom
Lawrence Livermore Nat'l Lab

John Clyne
Nat'l Center for Atmospheric Research

***Abstract*—As the need for data reduction in high-performance computing (HPC) continues to grow, we introduce a new and highly effective tool to help achieve this goal—SPERR. SPERR is a versatile lossy compressor for structured scientific data; it is built on top of an advanced wavelet compression algorithm, SPECK, and provides additional capabilities valued in HPC environments. These capabilities include parallel execution for large volumes and a compression mode that satisfies a maximum point-wise error tolerance. Evaluation shows that in most settings SPERR achieves the best rate-distortion trade-off among current popular lossy scientific data compressors.**

I. Introduction

Numerical simulations running on high-performance computers (HPCs) face a growing gap between data generation and storage—the ability to generate data is very much outpacing the ability to store data. As a result, simulation scientists are forced to take mitigation measures, including outputting fewer variables, outputting less frequently, and with an increasing popularity, adopting a data compression strategy. Between the two types of compression, *lossless* and *lossy* compression, the former is of limited use in scientific data compression, because it only achieves very modest data reduction. The latter, however, offers the potential for significant data reduction, and is viable as long as the information loss does not lead to misinterpretation of scientific results. In this paper, we focus on lossy compression of floating-point data.

Amongst the current leading lossy scientific data compression methods [1]–[10] there exists a good degree of variation in performance and capabilities, such as compression and decompression speed, support for an error guarantee, and rate-distortion trade-offs. Better rate-distortion trade-offs, for example using less storage for the same compression quality, may be prioritized in many applications. Consider large community data sets that can be analyzed by hundreds or even thousands of researchers, and have a lifetime lasting years. One example is the NCAR CESM LENS climate simulation data set [11], [12], which occupies 500 TB of on-premise storage. Another example is the Johns Hopkins Turbulence Database [13], [14], which hosts hundreds of terabytes of publicly accessible turbulence simulation outputs. In both cases the data sets may be written once, but accessed and/or transmitted repeatedly by researchers worldwide. Thus, achieved compression rates may trump other considerations such as computational speeds. Such scenarios serve as a motivation for our work.

When performing lossy compression, a criterion is needed to decide when to terminate the coding process. Most termination criteria are expressed as either a size bound or an error bound. With a size-bounding criterion, the compression algorithm terminates when its output reaches a specified size, which is often expressed as a bitrate measured by bit-per-point, or BPP. With an error-bounding criterion, however, the algorithm strives to achieve the maximum data reduction without the resulting reconstructed data exceeding an error bound. Note that no compressor can generally satisfy size and error bounds simultaneously. For the lossy compression of scientific data, a preferred termination criterion is often an error bound, particularly a point-wise error (PWE) tolerance. A PWE tolerance t ($t > 0$) means that any data point cannot deviate from its true value by more than t as a result of lossy compression; it often serves as an intuitive quality control in practice.

One of the most efficient lossy compressors from a rate-distortion perspective is SPECK [15], [16], which encodes coefficients produced by wavelet transforms. SPECK only operates as a *size-bounded* compressor. The development of SPECK, like most other transform-based schemes targetting multi-media (e.g., imagery and video) compression, was driven by the desire to minimize average error, not the maximum PWE, under a size constraint. However, it is also our observation that SPECK often proportionately reduces the maximum PWE while minimizing average error; this observation motivates our approach to augment SPECK to support a PWE guarantee while keeping its low average error.

We have implemented an improved version of the SPECK algorithm. After compressing input data with our SPECK implementation, we find all reconstructed data points that exceed a specified PWE tolerance, which are referred to as *outliers*. We then record their *positions* and *correction values* needed to restore the outlier values to be within the PWE tolerance, also using a SPECK-inspired algorithm. Our software taking on this approach is named SPERR, which stands for SPEck with ERRor-bounding. SPERR offers the best performance of any of the leading compression technologies we have evaluated from a rate-distortion perspective. The rest of this paper describes in detail our major contributions:

- an open-source scientific data compressor, SPERR, that features the ability to bound a maximum point-wise error (GitHub: `https://github.com/NCAR/SPERR`);
- thorough discussion and evaluation of design choices we have made when developing SPERR; and
- comparisons with leading scientific data compressors that provide a full profile of SPERR's characteristics.

II. Background

The random nature of the least significant bits in floating point numbers, and specialized requirements—such as the de-

sire to control point-wise errors—have led to the investigation and development of purpose-built scientific data compression technologies [1]–[10], [17]–[19]. Here, we briefly review only the current and most widely reported ones in the literature, which generally fall into two broad categories: prediction-based and transform-based.

The most widely reported prediction-based compressor is the SZ family of compressors [4]–[7], which have explored a variety of mathematical predictors, the most recent being splines [5]. The most widely reported transform-based compressor is ZFP [8], which employs a custom decorrelating transform, similar to the discrete cosine transform. SZ and ZFP were both developed in the last decade and have received great amounts of attention in both literature and practice.

In addition to these two established compression technologies, we reference two more recent additions: MGARD [2], [3] and TTHRESH [18]. The former is inspired by wavelet decompositions [20]–[24] and multi-grid methods used by linear solvers. TTHRESH is also transform-based, however, unlike most transform-based approaches that use predefined bases, TTHRESH uses the Tucker tensor decomposition [25] to generate *data-dependent* bases. The target application for TTHRESH is visualization, where bigger compression error is an acceptable trade-off for greater compression.

Our compressor, SPERR, is based on wavelet transforms. The result of applying wavelet transforms to an input is a set of wavelet coefficients having the property of *information compaction*: most information is stored in a small percentage of coefficients, whose information content is proportional to their magnitude. Wavelet transforms are reversible and achieve no data reduction themselves; it is during the coefficient coding process that data reduction occurs and information loss may be introduced. A wide range of sophisticated algorithms have been developed to efficiently code the addresses and values of the small amount of most information-rich coefficients [15], [16], [26]–[28]. Among them, SPECK [15], [16] serves as the basis of our own compressor (more discussion in Section III).

SPECK does not support a PWE guarantee though. Our approach to supporting such a guarantee is to *explicitly* encode *positions* and *correction values* of outlier residuals so we can later correct the offending reconstructed data points. The problem of encoding these positions and values is similar to compacting sparse matrices used in solving large linear systems. The most commonly employed methods for storing such matrices are Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) storage. However, CSR and CSC are far from optimal in our application because they still use naive storage to record element positions and values. Another approach is to record positions using bitmap coding [29], [30], and to handle correction values using, for example, variable-length coding (e.g., universal codes [31]). The SZ family of compressors takes this approach: quantized outlier correction values are stored as non-zero integers and then Huffman coded together with zero-valued inliers. As we shall show, our outlier coder accomplishes these two separate tasks in a unified manner with better efficiency.

III. Wavelet Transforms and Coding

This section describes the specifics of our implementation of a biorthogonal wavelet transform and the SPECK wavelet coding algorithm, paying particular attention to improvements that we have made to better control the amounts of resulting outliers. Given the close ties between SPECK and our outlier coding algorithm (more detail in Section IV), this section also serves as a quick review of the essence of SPECK.

A. Wavelet Transforms

We have chosen to use the CDF 9/7 biorthogonal wavelet transform [32] among a large selection of available wavelets, because it has proven to perform well in lossy scientific data compression, and possesses desirable properties such as near orthogonality and the ability to efficiently handle non-periodic input [20]–[24]. We have borrowed a lifting [33] implementation of the CDF 9/7 transform from QccPack, an excellent collection of signal processing tools [34]. This implementation uses symmetric boundary handling and wavelet basis functions with approximately unit norm. Because the CDF 9/7 wavelet functions are near-orthogonal and normalized, any L^2 error in wavelet coefficients introduced during encoding is approximately equal to the L^2 error in the reconstructed data.

In practice, wavelet transforms are applied recursively to an input array. The longer the array, the more transform passes can be applied, leading to better information compaction and thus compression. In our implementation, with an input array of length N, the number of recursive transforms applied is $\min(6, \lfloor \log_2 N \rfloor - 2)$. We choose to cap the recursion depth at six because of the diminishing benefit of deeply recursive wavelet transforms. Finally, for 3D volumes or 2D slices, transforms are separately applied along each axis to take advantage of data coherence in each dimension.

B. Overview of the Classic SPECK Algorithm

The classic SPECK algorithm is designed to locate wavelet coefficients, and encode them bitplane-by-bitplane, from most to least significand [15], [16]. Given a bitplane with place value 2^n, all coefficients with magnitude at least 2^n—deemed *significant coefficients*—are located first. SPECK performs this task by "zooming in" from the full data volume or slice to individual significant coefficients. Specifically, SPECK performs spatial divisions to partition 3D volumes into octrees or 2D slices into quadtrees. Each subtree is then examined and designated as *significant* or *insignificant* based on whether or not it contains at least one significant coefficient, during which one bit per significance test is recorded. Only significant subtrees are further partitioned, until individual significant coefficients (leaf nodes in the tree) are reached. With the storage of the significance information of subtrees at each level, this "zoom in" procedure can be replayed during decoding. In the second iteration, which processes bitplane 2^{n-1}, more coefficients will be deemed significant and recorded. Note that once a significant coefficient is discovered, it is stored together with all significant coefficients discovered from previous iterations, and does not partake in further significance tests.

Significant coefficients need to have their values recorded, which is accomplished with progressive precision using a quantized representation. Coefficients already found significant are maintained in a separate list, and a *refinement* pass appends the next significant bit of such coefficients before moving on to the next bitplane. The final reconstructed (non-zero) value usually has one additional one-bit appended as the least significant bit, resulting in mid-riser quantization that centers the value in its refined interval. Mid-tread quantization can also be achieved by rounding before applying SPECK if the place value of the last bitplane coded is known a priori.

The bitplane-by-bitplane fashion of SPECK coding naturally enables fixed-size compression, because the encoding process can terminate whenever a user-prescribed output size is reached, and the already-produced bitstream is always valid for decoding. This fixed-size compression capability is shared by most other transform-based compressors, such as ZFP.

C. Arbitrary Quantization Thresholds

When the original SPECK algorithm processes bitplanes with place value 2^n, n is always an integer. In our implementation, we have relaxed the integer requirement so any real value can be used in the algorithm, and more importantly, in the progressive quantization step. For example, denoting the *finest* quantization step size by q, the successive larger steps are $2q$, $4q$, $8q$, etc., none necessarily being an integer power of two. Note that one way to look at this modification is that we pre-scale all coefficients by the reciprocal of q and then apply the original SPECK algorithm with $n \geq 0$.

Supporting arbitrary quantization thresholds provides us with critical control of compression quality. The quantization scheme used in our implementation explains how the finest quantization step size q affects the compression quality. First, this scheme has a *dead zone* of $[-q, q]$; wavelet coefficients in the dead zone are not encoded and will be reconstructed as zeroes during decoding. Second, coefficients with a magnitude greater than q are quantized using a *mid-riser* quantizer, so that all values in the range of $(iq, (i+1)q]$ are quantized to $(i+\frac{1}{2})q$, where i is an integer. The maximum quantization error for these coefficients is thus $\frac{q}{2}$. Note that the error in the wavelet reconstruction may exceed $\frac{q}{2}$ as errors in overlapping wavelets may compound. Using this quantization scheme, a smaller q means both a smaller dead zone and smaller quantization error for every encoded coefficient, leading to a higher compression quality; a larger q, similarly, leads to a lower quality. Naturally, the better compression quality, the less outliers are produced, and vice versa. As a result, q controls not only SPECK compression quality, but also the amount of outliers need to be corrected. This relationship is demonstrated in Figure 2. Thus, allowing q to be an arbitrary value gives us great flexibility to direct SPECK to yield a desired amount of outliers, which is further explored in Section IV-D.

D. Embarrassingly Parallel Computation

Our implementation also includes an embarrassingly parallel strategy to take advantage of multi-core CPUs. Specifically, a big input volume is divided into multiple smaller chunks, and each chunk is processed individually in parallel. The output of each chunk is a separate bitstream, so these bitstreams are concatenated together to form the final compressed output. Our implementation uses OpenMP to orchestrate the parallel execution, and can support cases where the volume dimension is not divisible by the chunk dimension. Admittedly, this strategy imposes a limit on the degree of parallelization achievable—it cannot exceed the number of chunks. We plan to explore means to expose more parallelism in the future.

IV. Coding of Outliers

This section describes in detail a SPECK-inspired algorithm that efficiently encodes outliers; this algorithm enables SPERR to bound a prescribed PWE tolerance. This section also investigates how to achieve a desirable storage balance between coefficient coding and outlier coding in SPERR.

A. Algorithm Overview

The problem of outlier coding can be summarized as the following: by comparing the original data and wavelet reconstructed data, one can find a list of outliers that do not satisfy the user-defined PWE tolerance t, i.e., $|err_i| = |x_i - \tilde{x_i}| > t$, where $\tilde{x}$ is the wavelet reconstruction of the original value x. The task then is to record tuples of $(pos, corr)$ where pos is the outlier's position within the input data, and $corr$ is the correction value. A perfect correction value is $corr = x - \tilde{x}$, so the SPERR reconstruction z would be perfect with $z = \tilde{x} + corr$. However, an error up to t is allowed, so we look at the PWE with an imperfect correction value $\tilde{corr}$:

$$|z-x| = |\tilde{x}+\tilde{corr}-x| = |\tilde{corr}-(x-\tilde{x})| = |\tilde{corr}-err|. \quad (1)$$

This equation shows that bounding the SPERR reconstruction PWE $|z-x|$ is equivalent to bounding the correction value in the tuple $(pos, corr)$ so that $|\tilde{corr} - err| \leq t$.

Now the problem of outlier coding is very similar to what SPECK was designed to address: both encode a tuple of position and value, where value is a quantized approximation of the original (see Section III-B). Outlier coding has a very specific termination criterion though, which is the point where all outliers are corrected to be within the tolerance. Another difference is that in our implementation, we linearize 2D or 3D inputs to 1D arrays. We will discuss the decision to linearize higher-dimension data in Section IV-C.

B. Algorithm Description

We define the input to the outlier coding algorithm as:

- Length of the 1D array: N;
- A user-defined PWE tolerance: t ($t > 0$); and
- A list of outliers, each representing the position and correction value of an outlier: $(pos, corr)_i = (pos, x - \tilde{x})_i$.

The algorithm output is a compact bitstream that can be used at a later time to reconstruct individual outliers $(pos, \tilde{corr})_i$, where pos is exact and $\tilde{corr}_i$ is an approximation.

The algorithm also uses the concept of *significance* with respect to a quantization threshold $thrd$. If any one or more

members in a set of data points (in this case, outliers) have a value magnitude greater than $thrd$, then this set is *significant*. Otherwise, this set is *insignificant*. Note that as outlier coding is a completely separate procedure from the coding of wavelet coefficients, $thrd$ values used here are completely independent from those used in coefficient coding.

Listing 1: Overall outlier encoding algorithm. The three data structures $LSP, LNSP$ and LIS are globally accessible.

Algorithm: $OutlierCoder()$
Require: A PWE tolerance t, a list of outlier tuples, $(pos, corr)_i$, and the array length N.
1: Save the signs of $corr_i$ separately, then $corr_i \leftarrow |corr_i|$.
2: Create two global lists representing existing and newly-found significant points, LSP and $LNSP$. Both are initially empty.
3: Create a global list representing insignificant sets, LIS, which is initialized to be the entire array $[0, N)$.
4: Find the maximum integer $n \geq 0$ so that $2^n t < \max(corr_i)$.
5: **while** $n \geq 0$ **do**
6: $thrd \leftarrow 2^n t$
7: $SortingPass(thrd)$ {details in Listing 2}
8: $RefinementPass(thrd)$ {details in Listing 3}
9: $n \leftarrow n - 1$
10: **end while**

Listing 1 contains the pseudo-code of the encoding algorithm. At a high-level, this algorithm starts from the largest possible threshold that is an integer power-of-two multiple of the tolerance, locates significant data points with respect to that threshold ($SortingPass()$), and then refines them using quantization and bitplane coding ($RefinementPass()$). At the end of this iteration, outliers deemed significant with respect to $thrd$ are located and refined by $thrd$. In the next iteration, the threshold is halved, thus more outliers will be deemed significant. New outliers, along with those identified from previous iterations, will be further refined with respect to the halved threshold. This operation iterates with smaller and smaller thresholds until $thrd$ equals t, at which point all outliers will be deemed significant and encoded, and their reconstruction values will not deviate from the true values by more than $\frac{t}{2}$, satisfying the PWE tolerance.

Listing 2: Algorithm for $SortingPass()$.

Algorithm: $SortingPass(thrd)$
1: In increasing order of their sizes, for every set S that is already in LIS, do $Process(S, thrd)$.
2: ▷ $Process(S, thrd)$
3: Output a bit indicating if S is significant w.r.t. $thrd$.
4: **if** output was **true then**
5: **if** S is a single point **then**
6: Output the sign of S.
7: Put S in $LNSP$.
8: **else**
9: $Code(S)$
10: **end if**
11: Remove S from LIS.
12: **end if** {End of $Process(S, thrd)$}
13: ▷ $Code(S)$
14: Equally divide S into two disjoint subsets: sub_1 and sub_2.
15: Put sub_1 and sub_2 in LIS.
16: $Process(sub_1, thrd)$
17: $Process(sub_2, thrd)$ {End of $Code(S)$}

Listing 2 presents the pseudo-code of $SortingPass()$, a subroutine that locates and moves from LIS to $LNSP$ all previously insignificant points that become significant with respect to the current $thrd$. This process involves repeated binary set partitions until reaching individual significant data points (recursive function calls between $Process()$ and $Code()$). During this process, more insignificant sets are generated and added to LIS, waiting to be processed in the next iteration, when a smaller threshold may test some of them significant.

Listing 3: Algorithm for $RefinementPass()$.

Algorithm: $RefinementPass(thrd)$
1: **for all** $p = (pos, corr) \in LSP$ **do**
2: Output a bit indicating if $corr > thrd$.
3: **if** output was **true then**
4: Encoder: $corr \leftarrow corr - thrd$
5: Decoder: $corr \leftarrow corr + \frac{thrd}{2}$
6: **else**
7: Decoder: $corr \leftarrow corr - \frac{thrd}{2}$
8: **end if**
9: **end for**
10: **for all** $p = (pos, corr) \in LNSP$ **do**
11: Encoder: $corr \leftarrow corr - thrd$
12: Decoder: $corr \leftarrow \frac{3}{2} thrd$
13: **end for**
14: Append $LNSP$ to the end of LSP, then reset $LNSP$ to empty.

Listing 3 presents the pseudo-code of $RefinementPass()$, a subroutine that performs one level of refinement to the previously identified significant points, and also quantizes newly-discovered significant points, both with respect to the current threshold $thrd$. When invoked repeatedly with decreasing thresholds, it refines these points with each iteration specifying a narrower range. During decoding, though the reconstructed value can be anywhere within a specified range, it is chosen at the middle of the range, same as the mid-riser quantizer described in Section III-C. Lines 5, 7, and 12 show how this quantization approach reconstructs correctors in a decoder.

Note that all the algorithm output is in binary form, thus taking exactly one bit of storage. Every eight bits are then packed into a byte in the output bitstream. In this bitstream, every bit contains one of the following three types of information, depending on which step it was output from:

1) set significance (Line 3 of Listing 2);
2) outlier sign (Line 6 of Listing 2); or
3) direction in which to refine a value (Line 2 of Listing 3).

A decoder then uses the same execution paths as the encoder, with significance test results coming from the bitstream. Quantized outlier values are also restored along the way, and decoding terminates when the bitstream is exhausted.

C. Choice of Linearization

Our outlier coding algorithm is inspired by SPECK, which is designed to take advantage of clustering of significant wavelet coefficients. As seen in Figure 1, unlike wavelet coefficients, very little spatial correlation exists between outliers; rather, they tend to appear at random positions. With little or no spatial correlation to exploit, we choose to flatten the sparse, multi-dimensional outlier arrays into a 1D array prior to encoding, which simplifies the software implementation.

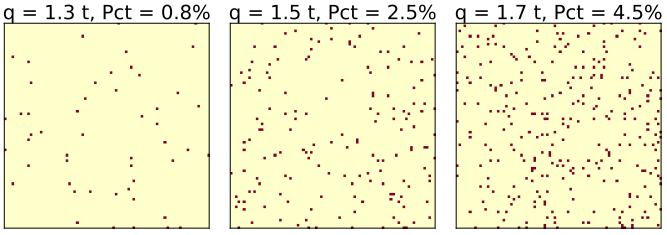

Fig. 1: A typical outlier heat map (produced from the Lighthouse image from the Kodak Image Suite [35]) where brown dots represent the positions of outliers. Little, if any, correlation between outlier positions can be observed. The subfigures represent three outlier percentage levels (noted above figures) which are controlled by the coefficient-outlier coding balance from $q = 1.3t$ to $q = 1.7t$ (more details in Section IV-D).

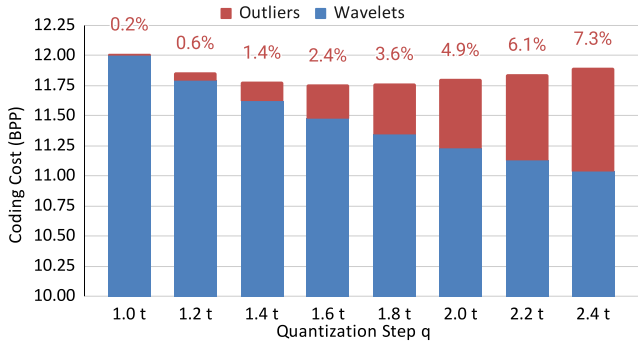

Fig. 2: Total coding cost as a function of quantization step, q, expressed in units of the error tolerance, t, for the Miranda Pressure field at tolerance level $t = 3.64 \times 10^{-11}$. The coding cost is broken out into the cost to code wavelet coefficients and outliers; the percentage labels indicate the portion of the coding cost associated with outliers. Note that only the cost above 10 BPP is shown in this plot.

Though there is little correlation for our outlier encoder to exploit, we do benefit from another important property of SPECK: the ability to efficiently encode position information along with variable-length coding for values. Section V-A and VI-E provide quantitative evaluations of this approach.

D. Balance Between Wavelet Coefficient and Outlier Coding

SPERR's primary goal is to minimize the storage cost for a given PWE tolerance. Secondarily, it also strives for a low average error. The total storage cost of SPERR consists of two components: the coding cost for wavelet coefficients and for outliers. This section investigates how to balance the relative storage allocation between the two to best achieve our goals.

TABLE I: Given a field with a data range $Range$, translate a label idx to an actual PWE tolerance t.

idx	PWE Tolerance t	Understanding of t
10	$\frac{Range}{2^{10}} \approx Range \times 10^{-3}$	One thousandth of the data range
20	$\frac{Range}{2^{20}} \approx Range \times 10^{-6}$	One millionth of the data range
30	$\frac{Range}{2^{30}} \approx Range \times 10^{-9}$	One billionth of the data range
40	$\frac{Range}{2^{40}} \approx Range \times 10^{-12}$	One trillionth of the data range

A basic property of SPECK coding for wavelet coefficients is that the more bits are produced, the less average error it introduces. Most often, less average error leads to fewer outliers and less storage cost for outlier coding. This inverse relationship between the coding cost for wavelet coefficients and outliers is demonstrated in Figure 2, with the technique to adjust the storage balance formally explained later in this section. Given that both coding costs (coefficients and outliers) constitute the total SPERR storage and their inverse relationship, it is reasonable to hypothesize that there is a sweet spot where the total storage is minimal, as suggested by Figure 2. However, we have found that it is nontrivial to analytically model this relationship because it depends on the characteristics of the data set and on the user-provided PWE tolerance. As a result, we resort to empirical solutions.

We run a series of experiments with multiple data sets and compression settings to better understand where the sweet spot might lie. Here we report results of four fields from two data sets, which are representative of all of the data sets we have tested. Section VI-B provides more detail on these data sets. For each data field, we test four (for single-precision input) or five (for double-precision input) tolerance levels indicated by idx; the actual PWE tolerance t is derived as $t = \frac{Range}{2^{idx}} = \frac{\max(f)-\min(f)}{2^{idx}}$ of a data field f. Table I provides an intuitive understanding of this translation. The parameter passed to the compressor is the actual PWE tolerance, t, while idx merely serves as a label facilitating discussion in this paper. For each tolerance t, we then shift the balance between wavelet coefficient coding and outlier coding by adjusting q, the quantization step in coefficient coding (see Section III-C). The smaller q is, the larger coefficient coding output will be, using more coefficient coding. The bigger q is, the smaller coefficient coding output will be, using more outlier coding. In practice, we have observed that the optimal q is very close in magnitude to t, so for convenience, we express q in multiples of t. This experiment tests q ranging from $q = t$ to $q = 3t$.

The top row of Figure 3 plots the relationship between q (horizontal axis) and the overall bitrate (vertical axis). The vertical axis represents the *increased cost* in overall bitrate relative to the minimum bitrate observed over all q settings. Not surprisingly, most curves take on a U shape, indicating the existence of a sweet spot that gives the lowest bitrate. The exact sweet spot varies from one field to another and from one compression level to another, but they are mostly in between $q = 1.4t$ and $q = 1.8t$. There are also occasional occurrences where bitrate drops unexpectedly with large q values in fields Miranda Viscosity and Nyx Dark Matter Density. This behavior happens when as q increases, the storage reduced by wavelet coefficient coding outpaces the additional storage incurred by outlier coding. In our experience, this phenomenon appears to occur rather rarely with larger q values only, and has no practical impact on the location of the minimum.

A secondary consideration of SPERR is the average error. Because outlier coding is almost always less efficient than

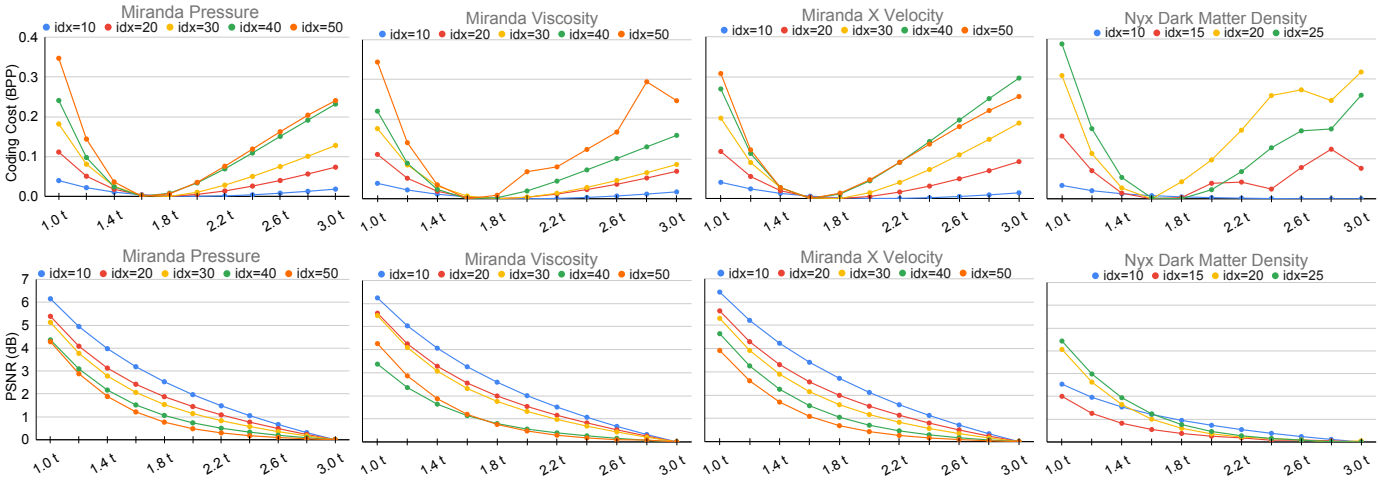

Fig. 3: Relative difference of bitrate (top) and average error (bottom) compared to the lowest observed values. The x-axis is quantization step q in coefficient coding expressed in units of the error tolerance t, and the y-axis is difference in BPP (top) and decibel, dB (bottom). For each field, multiple tolerance levels, $t \propto 2^{-idx}$, are tested, as the idx numbers indicate.

wavelet coefficient coding at reducing average error, a modest increase in coefficient coding bitrate can potentially lead to a substantial average error reduction. The bottom row of Figure 3 examines this consideration by plotting the *difference* in achieved peak-signal-to-noise ratio (PSNR), which is inversely proportional to the logarithm of root-mean-square error, at the same experiment settings (q at the same steps). As before, these curves are drawn in comparison with the lowest achieved PSNR for each idx setting; the vertical axis shows the *increase* in PSNR (higher is better). This time, all the curves are monotonically decreasing, suggesting that shifting the balance to use more outlier coding only increases the average error. These plots also show that though two different points on the U-shaped curves in the top row can be using the same amount of storage to satisfy a PWE tolerance, they will yield quite different average errors. Combining these two considerations, and the sweet spot range of $q = 1.4t$ to $q = 1.8t$, we conservatively choose $q = 1.5t$ in our software implementation. Further evaluation in Section VI-C will show that this choice is satisfactory at providing competitive rate-distortion curves.

V. Evaluation: Aspects of SPERR

Our compressor SPERR comprises coding of wavelet coefficients and outliers (Section IV and III, respectively), with each step producing a bitstream. These two bitstreams are then concatenated and losslessly compressed by ZSTD [36] to become the final SPERR output. This section evaluates aspects of SPERR and our design choices.

A. Outlier Coding Efficiency

We evaluate coding efficiency of the outlier coding algorithm, which is measured by bitrate of the outliers, the number of bits encoding all outliers over the number of outliers. Note that our implementation uses a header of a fixed size of twenty bytes; this cost is included in all evaluations in this paper.

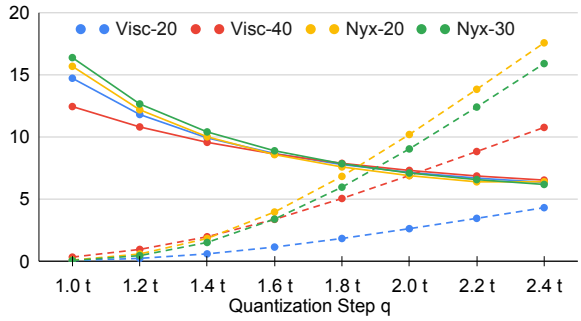

Fig. 4: Outlier bitrate (solid lines) and percentage (dashed lines) at different q values expressed in units of the PWE tolerance t. The bitrate is calculated over the number of outliers. Visc-20 and Visc-40 are Miranda Viscosity at tolerance level $idx = 20$ and $idx = 40$, and Nyx-20 and Nyx-30 are Nyx Dark Matter Density at $idx = 20$ and $idx = 30$.

Figure 4 presents this evaluation. It uses two data fields and two tolerance levels on each field. Again we vary the quantization step q for wavelet coefficient coding to adjust its quality, consequently the number of outliers produced. The percentage of outliers at each q is also plotted.

This evaluation shows that the cost of outlier coding (solid lines) is mostly between 6 to 16 bits *per outlier*. As q increases, more data points are identified as outliers, as the percentage curves (dashed lines) show. However, the amortized cost to encode individual outliers decreases, as the bitrate lines show, because each set significance test (Line 3 in Listing 2) catches a greater number of outliers that share the cost of that test. At $q = 1.5t$, the value our software implementation uses (see the last part of Section IV-D), this cost is approximately 10 bits per outlier. In our experience, this number is quite consistent across data sets. Section VI-E will provide a more detailed comparison between our strategy and the outlier coding method used by the SZ family of compressors.

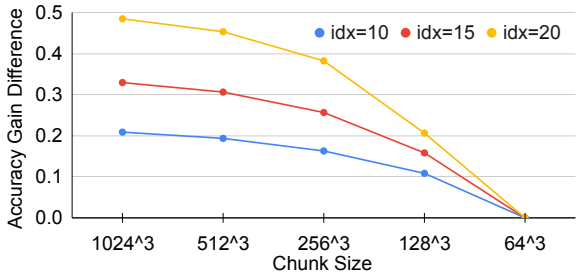

Fig. 5: Difference in accuracy gain with various chunk sizes. Three idx values represent three tolerance levels: $t = \frac{Range}{2^{idx}}$.

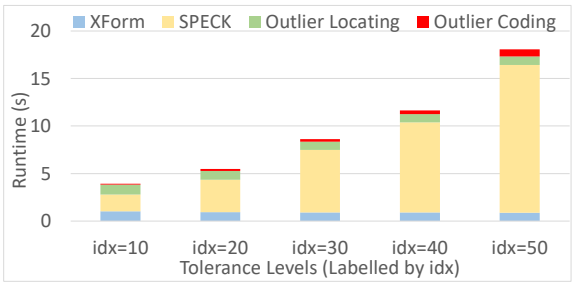

Fig. 6: Execution time breakdown on field Miranda Viscosity.

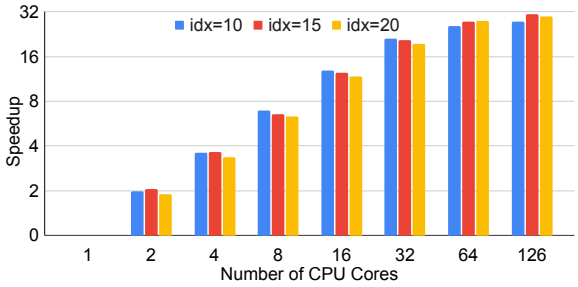

Fig. 7: Scalability test using up to 126 CPU cores with three tolerance levels. Note that the y-axis is on a logarithmic scale.

B. Chunk Size Impact on Compression Efficiency

SPERR achieves parallelization by dividing large 3D volumes into smaller chunks. The size of chunks, however, has impact on compression efficiency: the smaller the chunks are, the more boundaries they produce, and wavelet compression tends to handle boundaries less effectively. Further, small chunk sizes limit the number of passes of wavelet transforms that can be performed (see Section III-A), which also negatively impacts compression efficiency. This section evaluates the impact on compression efficiency by chunk sizes.

We use *accuracy gain*, explored by Lindstrom [37], to measure compression efficiency. Accuracy gain is defined as

$$gain = \log_2 \frac{\sigma}{E} - R \tag{2}$$

where σ is the standard deviation of the original data, E is the root-mean-square error, and R is the bitrate in BPP. A high accuracy gain is preferred and achieved when either the error, E, or rate, R, is low; intuitively, it measures the amount of information inferred by a compressor that need not be stored. Accuracy gain relates to the more common signal-to-noise ratio (SNR) by $gain = \frac{SNR}{20\log_{10} 2} - R \approx \frac{SNR}{6.02} - R$ and essentially flattens the 6.02 dB/bit slope commonly exhibited by SNR plots, thus compacting the vertical range and emphasizing differences in quality. Another benefit of accuracy gain is that it incorporates both average error and the storage used into a single number, allowing comparisons of two lossy-compressed data sets when neither their rate nor error matches.

Figure 5 plots *differences* in accuracy gain with various chunk sizes, using a cutout of $1,024^3$ at the center of the $3,072^3$ Miranda Density field. Unsurprisingly, bigger chunks bring higher accuracy gain. At the same time, the benefit diminishes with very large chunks. One may also observe that efficiency is even more impacted by chunk sizes for smaller error tolerances (bigger idx). Given that the chunk size also dictates the degree of parallelism that may be achieved (see Section III-D), in practice, we find that the range between 128^3 and 256^3 is preferable because it provides good compression efficiency and also enables multi-way parallelization. Our software uses a default chunk size of 256^3, but it needs not to be powers of two nor divisible by the volume dimensions.

C. Compression Time Breakdown

SPERR has four major steps in its data processing pipeline: 1) wavelet transform on the input data; 2) SPECK coding of wavelet coefficients; 3) locating outliers, which consists of an inverse wavelet transform and a comparison with the original input data; and 4) encoding located outliers. This section reports computational time on these four steps separately. We run this experiment on a compute node equipped with two 64-core AMD Epyc 7763 CPUs (128 cores in total), 256 GB of system memory, and SPERR compiled using GCC 11.2.0.

Figure 6 reports this time breakdown in a serial compression of Miranda Viscosity at $384^2 \times 256$. Though results on only one field are reported here, the time breakdown is representative of all fields that we have tested. There are five PWE tolerance levels tested, which are labeled by idx (see idx explanation in the third paragraph of Section IV-D). The result shows an increasing total compression time as the PWE tolerance tightens, mostly due to the increased SPECK coding time. SPECK time increases because tighter PWE tolerances lead to both more wavelet coefficients being located and that they are encoded with finer quantization precisions (see Section III-C). Wavelet transform time remains constant because those transforms are performed regardless of the tolerance level. Outlier locating and coding time also remains quite stable because the number of outliers does not fluctuate much, which is a design objective and is achieved by our strategy of choosing the quantization step size q, as discussed in Section IV-D.

D. Scalability Test

This section explores how well SPERR scales on modern multi-core CPU architectures by conducting a strong scaling experiment. The test platform is the same 128-core compute node described in Section V-C. This experiment uses a cutout of $2,048^3$ from the $3,072^3$ Miranda Density field in single precision; the cutout is due to memory capacity limits on the test platform. With SPERR's default chunk size of 256^3,

512-way parallelization is possible. We test on three PWE tolerance levels labeled as $idx = 10, 15$, and 20. For each tolerance level, we increase the number of OpenMP threads, thus CPU cores in use, from one to 126. (We do not use up all 128 CPU cores because it is recommended practice to leave a few CPU cores for system processes.) Figure 7 reports achieved parallel speedups compared to the serial execution time in a compression task. The result shows a close-to-linear speedup with up to 16 cores, and a gradually slower increase afterwards. Further, the speedup is seemingly approaching a plateau after 64 cores, signaling limitations of our embarrassingly parallel strategy.

VI. Evaluation: Comparing With Other Tools

A. Other Compressors And Test Environment

We have used SPERR version `0.4` for all comparisons in this study. Other scientific data compressors used in these comparisons are SZ3 [5], ZFP [8], TTHRESH [18], and MGARD [2]. All compressors, including our SPERR, are open source and publicly available. Section II has a brief description of these compressors, and this section provides the specifics in our experiment.

- SZ3: version `3.1.5.4` with the default config file except for enabled OpenMP.
- ZFP: version `1.0.0` with default configurations.
- TTHRESH: commit number `ae58002` (the latest code as of Sep. 9th 2022) with default configurations.
- MGARD: version `1.3.0` with default configurations and OpenMP backend.

All five compressors are compiled using GCC 11.2.0. All comparisons in this section are conducted on a compute node equipped with two 18-core Intel Xeon 6240 processors and 394 GB system memory, running CentOS 7 operating system.

B. Testing Data Sets

Our tests use openly available data sets from SDRBench, a collection of representative scientific data sets frequently used in literature for compression evaluation [38]. Specifically, we have chosen four simulations briefly described blow. The SDRBench website has more details on each of them.

- Miranda: hydrodynamics turbulence simulation with double-precision output at $384^2 \times 256$ and single-precision output at $3,072^3$;
- S3D: combustion, double-precision output at 500^3;
- Nyx: cosmology, single-precision output at 512^3; and
- QMCPACK: ab initio quantum Monte Carlo simulation, single-precision output at $69^2 \times 115 \times 288$ (288 orbitals).

The QMCPACK data set is essentially a stack of 3D volumes of size $69^2 \times 115$, which is best to be compressed as 288 individual volumes. SPERR is configured to do so with its chunk size specified as $69^2 \times 115$. The other compressors are configured to process a single 3D volume of size $69^2 \times 33120$ ($33120 = 115 \times 288$), which is less than ideal, but is also listed as example configurations on SDRBench.

C. Comparison Of Compression Efficiency

This section compares the efficiency of SPERR with other leading compressors. We again use accuracy gain (see Section V-B) as the main metric, and an integer idx to label PWE tolerance levels (see Table I). Note that it is the PWE tolerance t, rather than idx, passed in as the quality control parameter to compressors. We increment idx from zero to the point where t is approaching machine epsilon. For single-precision input (Nyx and QMCPACK), the upper bound on idx reaches between 25 and 35, and for double-precision input (S3D and Miranda), the upper bound reaches between 50 and 60. We note that when t is tight MGARD cannot bound the error tolerance [1] and TTHRESH starts to use significantly more bits without reducing average error nor the maximum PWE. When such unexpected behavior occurs, the offending test is terminated, and not included in the presentation here. (TTHRESH requires some special attention: it supports a target average error (e.g., PSNR) but not a PWE guarantee. As a result, at each idx we prescribe for TTHRESH $PSNR = (20 \log_{10} 2) \times idx$, which results in a halving of root-mean-square error for each increment to idx. Lastly, TTHRESH was not able to finish computation on data set QMCPACK, so is not included in results for QMCPACK.)

Figure 8 presents comparison results over nine data fields using rate-distortion curves. We use logarithmic scale on the x-axis so low-rate, low-quality compression regions are clearly shown. Compression in these regions, for example most compressors achieve around 50 dB in PSNR at $\frac{1}{32}$ BPP, may already provide sufficient quality for applications such as visualization. There are a few interesting observations. First, the curves increase at low rates, indicating that significant compression occurs, where each halving in error, E, incurs less than one additional bit of compressed storage, R (see Equation 2). Most curves then reach a stable plateau, where each additional bit encoded halves the error, indicating that random trailing significant bits have been reached. This plateau spans a wide rate range, which is often 10- to 20-bit wide, though it appears horizontally "compacted" by the logarithmic scale in these plots. Second, SPERR exhibits a clear advantage in achieving high accuracy gains at mid-to-high rates (i.e., more than 2 BPP), while remains competitive at low rates (i.e., less than 1 BPP). In most scientific applications involving quantitative analyses, we argue that high-rate, high-quality compression is much more useful and frequently requested than low-rate, low-quality compression. Third, not all curves exhibit the same level of smoothness, which indicates different degrees of *predictability* of a compressor's behavior. On this account, SPERR is one of the more predictable compressors. Overall, these rate-distortion plots suggest SPERR's outstanding compression efficiency over a wide range of bitrates.

Another aspect of the efficiency comparison is how many bits a compressor needs to satisfy a PWE tolerance, *regardless* of the overall average error. We perform this test using the same data fields as in Figure 8 with a selection of tolerance

[1] Suspected bugs in MGARD have been reported to MGARD authors.

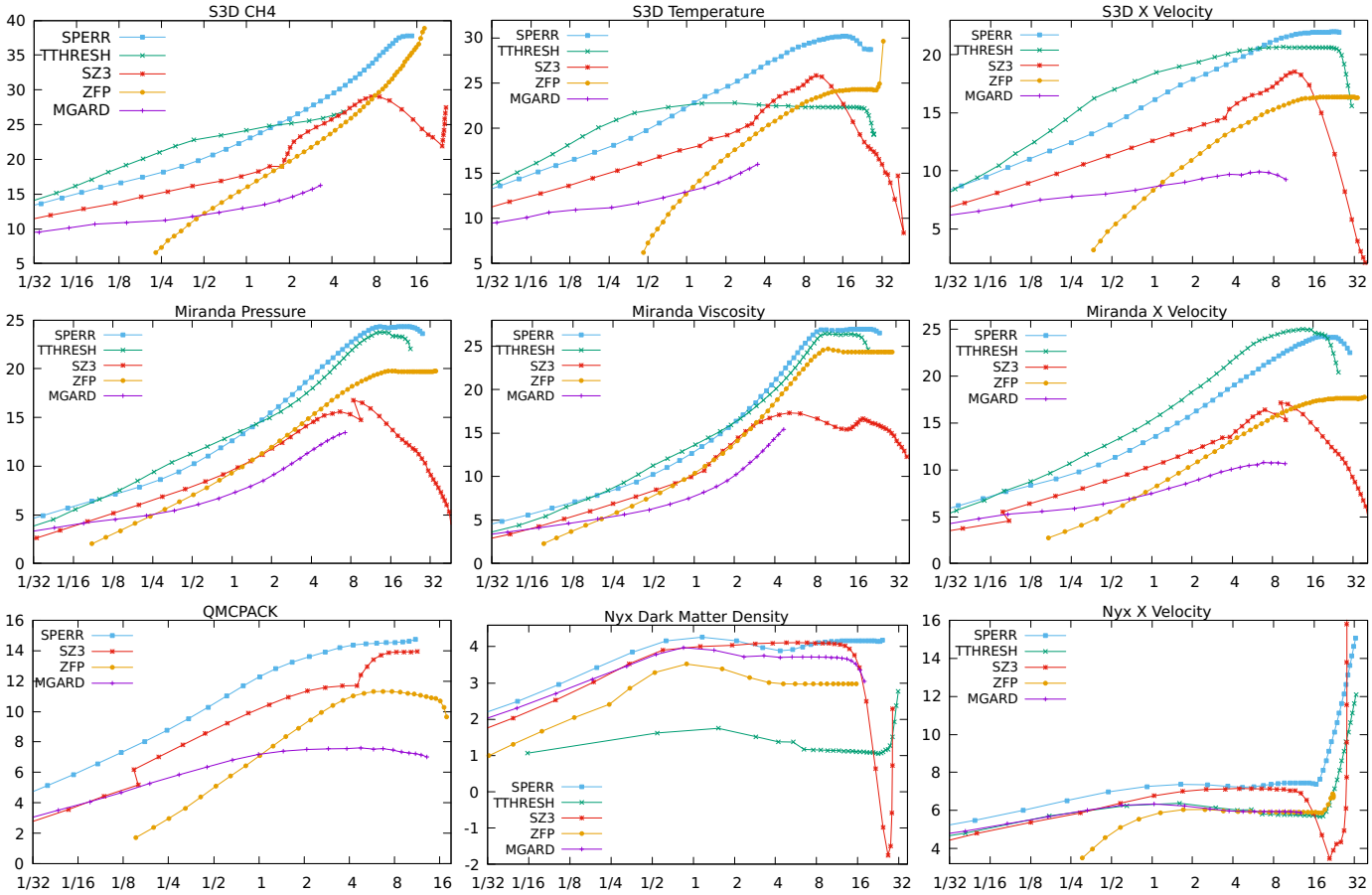

Fig. 8: Rate-distortion curves comparing five compressors on nine data fields. The x-axis is the achieved bitrate measured in BPP, and the y-axis is accuracy gain as defined in Section V-B. Note that the x-axis is plotted in logarithmic scale.

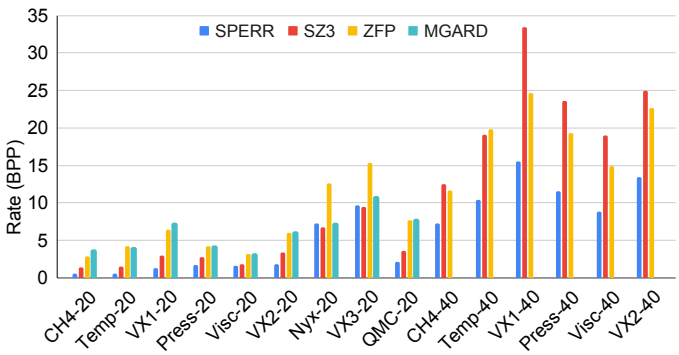

Fig. 9: Achieved bitrate on multiple data sets and tolerance levels. Abbreviations on the x-axis are explained in Table II.

TABLE II: Explanation of abbreviations for data fields and compression levels used in Figures 9, 10, and 11.

CH4-20, CH4-40	S3D CH4, $idx = 20$ and 40
Temp-20, Temp-40	S3D Temperature, $idx = 20$ and 40
VX1-20, VX1-40	S3D X Velocity, $idx = 20$ and 40
Press-20, Press-40	Miranda Pressure, $idx = 20$ and 40
Visc-20, Visc-40	Miranda Viscosity, $idx = 20$ and 40
VX2-20, VX2-40	Miranda X Velocity, $idx = 20$ and 40
QMC-20	QMCPACK, $idx = 20$
Nyx-20	Nyx Dark Matter Density, $idx = 20$
VX3-20	Nyx X Velocity, $idx = 20$

levels. This time TTHRESH is not tested because it does not have an error-bounded compression mode. MGARD is also not presented at $idx = 40$ tolerance levels because it gives results obviously exceeding the error tolerance. Figure 9 presents our test results, using abbreviations for field names and tolerance levels explained in Table II. These results show that SPERR uses the least number of bits to guarantee a given PWE tolerance in all but two cases, again highlighting the superior compression efficiency of SPERR.

Finally, we note that the main evaluation metric used here, accuracy gain, is generic in nature and aims to provide an overview of SPERR's characteristics. Evaluations using more domain-specific metrics (e.g., SSIM [39]) are likely necessary to determine SPERR's applicability in a particular use case.

D. Comparison of Runtime Performance

This section compares parallel runtime performance on multi-core CPUs. All five compressors support parallelization through OpenMP, which we have enabled during compilation. This experiment uses the same data fields and tolerance levels summarized in Table II. There are two considerations: first, MGARD is not presented at $idx = 40$ tolerance levels, because it gives results obviously exceeding the error tolerance. Second, TTHRESH takes in PSNR targets, instead of PWE

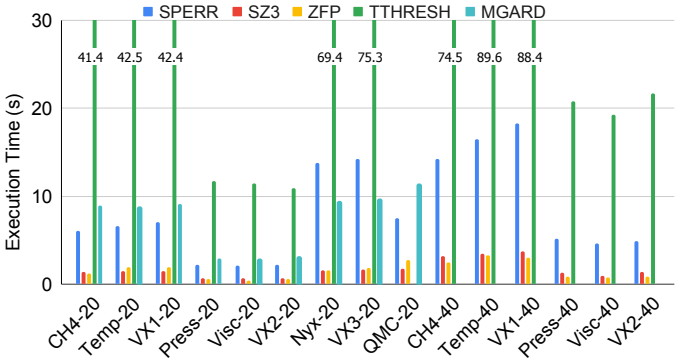

Fig. 10: Compression time on multiple data sets and tolerance levels. Abbreviations on the x-axis are explained in Table II.

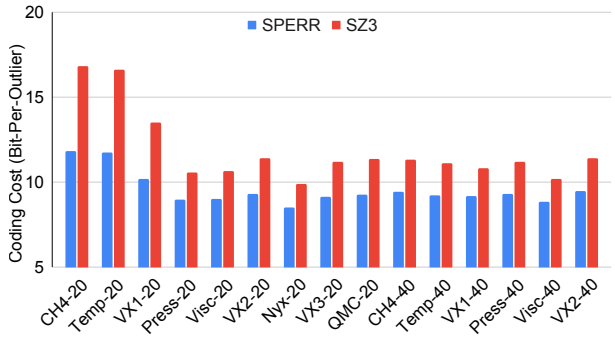

Fig. 11: Comparison of outlier coding efficiency between SPERR and SZ. The x-axis is different data fields and tolerance levels (see Table II), and the y-axis is the coding cost of outliers. Note that only the cost above 5 bits is shown.

tolerances, because average errors are the only parameters that TTHRESH accepts. Using the same formula discussed and used in Section VI-C, these PSNR targets for TTHRESH are determined as 120.41 dB at $idx = 20$, and 240.82 dB at $idx = 40$. In fact, these PSNR targets are quite close to the PSNR values SPERR achieved at respective tolerance levels.

Figure 10 presents this runtime comparison using four OpenMP threads. The reported numbers are wall clock times of invoking respective command-line tools to perform a compression task. Note that eight runs have time exceeding the graph scale, so their respective bars are truncated with a text label indicating the actual time. These results show that SZ3 and ZFP are comparable and are extremely fast compared to the rest compressors. SPERR runs a few times slower than SZ3 and ZFP, but is also significantly faster than TTHRESH. Finally, SPERR performs comparably with MGARD.

E. Comparison Of Outlier Coding Efficiency

To the best of our knowledge, SPERR's approach to providing a PWE guarantee—explicitly encoding corrections to outliers that violate the PWE tolerance t—is shared by only one other lossy compressor: SZ. SZ uses a different scheme to encode outliers. It first quantizes prediction errors to integer multiples of $2t$, analogous to SPERR's correction values. SZ then uses Huffman coding to efficiently encode these integers [6]. The Huffman codes and tree are finally compressed by ZSTD [36]. We perform experiments to compare outlier coding efficiency between SPERR and SZ by first intercepting SPERR's data processing pipeline to obtain the list of outliers, and then feeding the same list of outliers to respective outlier coding schemes (SPERR's and SZ's).

To prepare the outliers for SZ, we first quantize the SPERR outlier correction values as multiples of the PWE tolerance. SZ encodes a correction value for *every* data point, where inliers are represented as zero-valued correctors. Hence, SZ does not code the positions of outliers. Unlike in SZ, which usually uses thousands of quantization bins, the SPERR correctors are usually small; we observed no corrector outside the range of $\{-4, \ldots, 4\}$. Finally, we note that SZ provides a separate tool, *compressQuantBins*, in its QCAT package [40] to perform this coding task. We used *compressQuantBins* in this experiment.

Figure 11 presents this comparison using the same data fields and compression levels described in Table II. It uses bit-per-outlier, the average number of bits to encode an outlier, to measure the coding efficiency. This figure shows that SPERR uses around ten bits to encode one outlier across all experiment settings, consistent with our findings in Section V-A. It also shows that SPERR consistently uses fewer bits than SZ to encode the same set of outliers, usually by a 1- to 2-bit margin.

VII. Future Work and Conclusion

SPERR has great potential to grow its capabilities. First, the property of roughly equal root-mean-square error between wavelet coefficients and their inversely transformed reconstruction (see Section III-A) enables estimating compression error without much computational overhead, thus compression targeting an average error is feasible. Second, wavelet transforms naturally represent data as hierarchies with self-similarities, i.e., each coarsened hierarchy level resembles the full-resolution data. This hierarchy enables multi-level reconstruction that is useful in areas such as explorative analysis. Third, the bitplane-by-bitplane quantization scheme in SPECK means that the output bitstream is *embedded*, so any prefix of the bitstream can reconstruct a less-accurate version of the data, in addition to the error-bounded version provided by the full bitstream. This embedded property makes SPERR suitable for streaming applications where data reconstruction using a partially transmitted bitstream, though less accurate, may still be appreciated. Fourth, as we have alluded, there is still great opportunity to improve SPERR's runtime performance, potentially through parallelization schemes that are more sophisticated and allow for porting the algorithm to GPUs.

In conclusion, this paper has introduced SPERR, a new tool for lossy scientific data compression on HPCs. SPERR can provide both size-bounded and maximum point-wise error (PWE) bounded compression; the latter ability is provided by a novel use of an outlier coding algorithm that explicitly corrects data points exceeding a prescribed PWE tolerance. We have thoroughly evaluated SPERR and compared it against leading scientific data compressors; these studies demonstrate that SPERR has one of the most competitive compression efficiencies with the drawback of a higher computational cost.

ACKNOWLEDGMENT

This material is based upon work supported by the National Center for Atmospheric Research, which is a major facility sponsored by the National Science Foundation under Cooperative Agreement No. 1852977.

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research.

REFERENCES

[1] C. S. Zender, "Bit grooming: statistically accurate precision-preserving quantization with compression, evaluated in the netcdf operators (nco, v4. 4.4.8+)," *Geoscientific Model Development*, vol. 9, no. 9, pp. 3199–3211, 2016.

[2] M. Ainsworth, O. Tugluk, B. Whitney, and S. Klasky, "Multilevel techniques for compression and reduction of scientific data—the univariate case," *Computing and Visualization in Science*, vol. 19, no. 5, pp. 65–76, 2018.

[3] ——, "Multilevel techniques for compression and reduction of scientific data—the multivariate case," *SIAM Journal on Scientific Computing*, vol. 41, no. 2, pp. A1278–A1303, 2019.

[4] X. Liang, K. Zhao, S. Di, S. Li, R. Underwood, A. M. Gok, J. Tian, J. Deng, J. C. Calhoun, D. Tao *et al.*, "Sz3: A modular framework for composing prediction-based error-bounded lossy compressors," *IEEE Transactions on Big Data*, 2022.

[5] K. Zhao, S. Di, M. Dmitriev, T.-L. D. Tonellot, Z. Chen, and F. Cappello, "Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1643–1654.

[6] D. Tao, S. Di, Z. Chen, and F. Cappello, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *2017 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2017, pp. 1129–1139.

[7] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, "Error-controlled lossy compression optimized for high compression ratios of scientific datasets," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 438–447.

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

[9] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1245–1250, 2006.

[10] L. Hayne, J. Clyne, and S. Li, "Using neural networks for two dimensional scientific data compression," in *2021 IEEE International Conference on Big Data (Big Data)*, 2021, pp. 2956–2965.

[11] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. Bates, G. Danabasoglu, J. Edwards *et al.*, "The community earth system model (cesm) large ensemble project: A community resource for studying climate change in the presence of internal climate variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333–1349, 2015.

[12] J. Kay and C. Deser, "The community earth system model (cesm) large ensemble project," 2016. [Online]. Available: http://www.cesm.ucar.edu/projects/community-projects/LENS

[13] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, and G. Eyink, "A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence," *Journal of Turbulence*, no. 9, p. N31, 2008.

[14] E. Perlman, R. Burns, Y. Li, and C. Meneveau, "Data exploration of turbulence simulations using a database cluster," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007, pp. 1–11.

[15] W. A. Pearlman, A. Islam, N. Nagaraj, and A. Said, "Efficient, low-complexity image coding with a set-partitioning embedded block coder," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 11, pp. 1219–1235, 2004.

[16] X. Tang and W. A. Pearlman, "Three-dimensional wavelet-based compression of hyperspectral images," in *Hyperspectral Data Compression*. Springer, 2006, pp. 273–308.

[17] J. Clyne, "Progressive data access for regular grids," in *High Performance Visualization*, E. W. Bethel, H. Childs, and C. Hansen, Eds. CRC, 2012, pp. 145–169.

[18] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "TTHRESH: Tensor compression for multidimensional visual data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 9, pp. 2891–2903, 2019.

[19] S. Li, N. Marsaglia, C. Garth, J. Woodring, J. Clyne, and H. Childs, "Data reduction techniques for simulation, visualization and data analysis," *Computer Graphics Forum*, vol. 37, no. 6, pp. 422–447, 2018.

[20] J. D. Villasenor, B. Belzer, and J. Liao, "Wavelet filter evaluation for image compression," *IEEE Transactions on Image Processing*, vol. 4, no. 8, pp. 1053–1060, 1995.

[21] R. A. DeVore, B. Jawerth, and V. Popov, "Compression of wavelet decompositions," *American Journal of Mathematics*, vol. 114, no. 4, pp. 737–785, 1992.

[22] B. E. Usevitch, "A tutorial on modern lossy wavelet image compression: foundations of JPEG 2000," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 22–35, 2001.

[23] S. Li, K. Gruchalla, K. Potter, J. Clyne, and H. Childs, "Evaluating the efficacy of wavelet configurations on turbulent-flow data," in *IEEE 5th Symposium on Large Data Analysis and Visualization*, 2015, pp. 81–89.

[24] S. Li, J. Clyne, and H. Childs, "In situ wavelet compression on supercomputers for post hoc exploration," in *In Situ Visualization for Computational Science*, H. Childs, J. C. Bennett, and C. Garth, Eds. Cham: Springer International Publishing, 2022, pp. 37–59.

[25] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[26] A. Islam and W. A. Pearlman, "Embedded and efficient low-complexity hierarchical image coder," in *Visual Communications and Image Processing'99*, vol. 3653. International Society for Optics and Photonics, 1998, pp. 294–305.

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

[28] D. Taubman, "High performance scalable image compression with EBCOT," *IEEE Transactions on Image Processing*, vol. 9, no. 7, pp. 1158–1170, 2000.

[29] P. Howard, F. Kossentini, B. Martins, S. Forchhammer, and W. Rucklidge, "The emerging JBIG2 standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 8, no. 7, pp. 838–848, 1998.

[30] K. Wu, E. J. Otoo, and A. Shoshani, "Optimizing bitmap indices with efficient compression," *ACM Transactions on Database Systems*, vol. 31, no. 1, pp. 1–38, 2006.

[31] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[32] A. Cohen, I. Daubechies, and J.-C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, vol. 45, no. 5, pp. 485–560, 1992.

[33] I. Daubechies and W. Sweldens, "Factoring wavelet transforms into lifting steps," *Journal of Fourier analysis and applications*, vol. 4, no. 3, pp. 247–269, 1998.

[34] J. E. Fowler, "QccPack: An open-source software library for quantization, compression, and coding," in *Applications of Digital Image Processing XXIII*, vol. 4115. International Society for Optics and Photonics, 2000, pp. 294–301.

[35] R. Franzen, "Kodak lossless true color image suite," *source: https://r0k.us/graphics/kodak*, vol. 4, no. 2, 1999.

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

[37] P. Lindstrom, "MULTIPOSITS: Universal coding of $\mathbb{R}^n$," in *Conference on Next Generation Arithmetic*, J. Gustafson and V. Dimitrov, Eds. Springer International Publishing, 2022, pp. 66–83.

[38] K. Zhao, S. Di, X. Lian, S. Li, D. Tao, J. Bessac, Z. Chen, and F. Cappello, "SDRBench: Scientific data reduction benchmark for lossy compressors," in *IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2716–2724. [Online]. Available: https://sdrbench.github.io/

[39] Z. Wang and A. C. Bovik, "A universal image quality index," *IEEE signal processing letters*, vol. 9, no. 3, pp. 81–84, 2002.

[40] S. Di, "Quick Compression Analysis Toolkit (QCAT)." [Online]. Available: https://github.com/szcompressor/qcat