TAC: Optimizing Error-Bounded Lossy Compression for Three-Dimensional Adaptive Mesh Refinement Simulations

Daoce Wang
Washington State University
Pullman, WA, USA
daoce.wang@wsu.edu

Jesus Pulido
Los Alamos National Laboratory
Los Alamos, NM, USA
pulido@lanl.gov

Pascal Grosset
Los Alamos National Laboratory
Los Alamos, NM, USA
pascalgrosset@lanl.gov

Sian Jin
Washington State University
Pullman, WA, USA
sian.jin@wsu.edu

Jiannan Tian
Washington State University
Pullman, WA, USA
jiannan.tian@wsu.edu

James Ahrens
Los Alamos National Laboratory
Los Alamos, NM, USA
ahrens@lanl.gov

Dingwen Tao*
Washington State University
Pullman, WA, USA
dingwen.tao@wsu.edu

ABSTRACT

Today's scientific simulations require a significant reduction of data volume because of extremely large amounts of data they produce and the limited I/O bandwidth and storage space. Error-bounded lossy compression has been considered one of the most effective solutions to the above problem. However, little work has been done to improve error-bounded lossy compression for Adaptive Mesh Refinement (AMR) simulation data. Unlike the previous work that only leverages 1D compression, in this work, we propose to leverage high-dimensional (e.g., 3D) compression for each refinement level of AMR data. To remove the data redundancy across different levels, we propose three pre-process strategies and adaptively use them based on the data characteristics. Experiments on seven AMR datasets from a real-world large-scale AMR simulation demonstrate that our proposed approach can improve the compression ratio by up to 3.3× under the same data distortion, compared to the state-of-the-art method. In addition, we leverage the flexibility of our approach to tune the error bound for each level, which achieves much lower data distortion on two application-specific metrics.

CCS CONCEPTS

• **Theory of computation → Data compression**.

KEYWORDS

AMR; Lossy compression; scientific data; compression performance.

ACM Reference Format:
Daoce Wang, Jesus Pulido, Pascal Grosset, Sian Jin, Jiannan Tian, James Ahrens, and Dingwen Tao. 2022. TAC: Optimizing Error-Bounded Lossy Compression for Three-Dimensional Adaptive Mesh Refinement Simulations. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC '22), June 27-July 1, 2022, Minneapolis, MN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3502181.3531458

*Corresponding author: Dingwen Tao, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99163, USA.

HPDC '22, June 27-July 1, 2022, Minneapolis, MN, USA

ACM ISBN 978-1-4503-9199-3/22/06.
https://doi.org/10.1145/3502181.3531458

1 INTRODUCTION

Motivation. The increase in supercomputer performance over the last few years has been insufficient to solve many challenging modeling and simulation problems. For example, the complexity of solving evolutionary partial differential equations (PDEs) scales as $\Omega(n^4)$, where n is the number of mesh points per dimension. Thus, the performance improvement of about three orders of magnitudes over the past 30 years has meant just a 5.6× gain in spatio-temporal resolution [8]. To address this issue, many high-performance computing (HPC) simulation packages [15] (such as AMReX [41] and Athena++ [33]) use Adaptive Mesh Refinement (AMR)—which applies computation to selective regions of most interest—to increase resolution. Compared to the method where a high resolution is applied everywhere, the AMR method can greatly reduce the computational complexity and storage overhead; thus, it is one of most widely used frameworks for many HPC applications [2, 31, 34, 38] in various science and engineering domains.

Although AMR can save storage space to some extent, AMR applications running on supercomputers still generate large amounts of data, making the data transmission and storage challenging. For example, one Nyx simulation [30] with a resolution of 4096^3 (i.e., 0.5×2048^3 mesh points in the coarse level and 0.5×4096^3 in the fine level) can generate up to 1.8 TB of data for a single snapshot; a total of 1.8 PB of disk storage is needed assuming running the simulation 5 times with 200 snapshots dumped per simulation. Therefore, reducing data size is necessary to lower the storage overhead and I/O cost and improve the overall application performance for large-scale AMR applications running on supercomputers.

A straightforward way to address this issue is to use data compression. However, traditional lossless compression techniques such as GZIP [12] and Zstandard [44] can only provide a compression ratio up to 2 for scientific data [32]. On the other hand, a new generation of lossy compressors which can provide a strict error control (called "error-bounded" lossy compression) has been developed, such as SZ [14, 24, 35], ZFP [27], MGARD [1], and TTHRESH [6]. Using those error-bounded lossy compressors, scientists can achieve relatively high compression ratios while minimizing the quality loss of reconstructed data and post analysis, as demonstrated in many prior studies [4, 5, 9, 17, 18, 21, 28, 40].

Limitation of state-of-the-art approach. Only a few existing contributions have investigated error-bounded lossy compression for AMR applications and datasets. A common approach is to generate uniform resolution data by up-sampling the coarse-level data and

merging them with the finest-level data, and then to perform compression on the merged data. However, this approach introduces redundant information to the data, which will significantly degrade the compression ratio, especially when the up-sampling rate is high or there are multiple coarse levels to up-sample. Recently, Luo *et al.* introduced zMesh [29], a technique that groups data points that are mapped to the same or adjacent geometric coordinates such that the dataset is smoother and more compressible. However, since zMesh maps data points from different AMR levels to adjacent geometric coordinates and generates a 1D array, it cannot adopt 3D compression which most HPC simulations use. Moreover, zMesh is designed only for patch-based AMR applications. The patch-based AMR structure saves the data blocks that will be refined at the next level in the current level redundantly. While the state-of-the-art AMR framework AMReX provides quadtree/octree-based structure besides patch-based structure [3], many newly developed AMR applications such as Nyx adopt the tree-based structure to avoid redundancy by only saving each data point in the level of its finest refinement. For this scenario, the reorganization approach proposed by zMesh may not improve the data smoothness appropriately (will be demonstrated in Section 4).

Key contributions. To solve these issues, we propose an approach (called TAC) to optimize error-bounded Three-dimensional AMR lossy Compression. Specifically, we propose to adopt 3D compression for each AMR level. However, each level may contain many empty regions (i.e., zero blocks), where data points are saved in other levels; these empty regions (zero blocks) significantly decrease the data smoothness/compressibility and increase the data size (hence reduce the compression ratio). Thus, we propose to either remove these empty regions or partially pad them with appropriate values, based on the density of empty regions. Furthermore, we propose an optimization to reduce the time cost of removing empty regions. Finally, we evaluate TAC on seven datasets and compare it with the state-of-the-art approach. Our main contributions are summarized as follows.

- We propose to leverage 3D compression to compress each level of an AMR dataset separately. We propose a hybrid compression approach based on the following three preprocess strategies and data characteristics (e.g., data density).
- For sparse AMR data, we propose an optimized sparse tensor representation to efficiently remove empty regions.
- To reduce the time overhead of removing empty regions, we propose an optimization based on the enhanced k-d tree.
- For dense AMR data, we propose a padding approach to improve the smoothness and compressibility.
- We tune the error bound for each AMR level for Nyx cosmology simulation, which improves the compression quality in terms of two application-specific post-analysis metrics.
- Experiments show that, compared to the state-of-the-art approach zMesh, TAC can improve the compression ratio by up to 3.3× under the same data distortion on the tested real-world datasets.

Experimental methodology and artifact availability. We evaluate TAC on seven datasets from two real-world AMR simulation runs. The AMR simulations are well-known, open-source cosmology simulations—Nyx [30]. We compare TAC with three baselines including zMesh using generic metrics such as compression ratio and peak signal-to-noise ratio (PSNR) and application-specific metrics such as power spectrum and halo finder. Our code and datasets are available at https://github.com/hipdac-lab/3dAMRcomp.

Limitations of the proposed approach. Compared with the approach that up-samples the coarse-level data and then compresses the data with uniform resolution (denoted by "3D baseline"), TAC provides much better compression performance (i.e., rate-distortion), when the finest level of the AMR dataset has a relatively low density. However, when the finest level has a relatively high density, TAC is slightly worse than the 3D baseline. We will discuss this limitation in detail in Section 4.3.

In Section 2, we present background information about error-bounded lossy compression, AMR method, k-d tree, and related work on AMR data compression. In Section 3, we describe our proposed pre-process strategies and hybrid compression. In Section 4, we show the experimental results on different AMR datasets. In Section 5, we conclude our work and discuss the future work.

2 BACKGROUND AND RELATED WORK

In this section, we introduce background information about lossy compression for scientific data, AMR method and data, classic k-d tree used in particle data compression, and discuss the state-of-the-art method of AMR data compression and remaining challenges.

2.1 Lossy Compression for Scientific Data

There are two main categories for data compression: lossless and lossy compression. Compared to lossless compression, lossy compression can offer much higher compression ratio by trading a little bit of accuracy. There are some well-developed lossy compressors for images and videos such as JPEG [36] and MPEG [23], but they do not have a good performance on the scientific data because they are mainly designed for integers rather than floating points.

In recent years there is a new generation of lossy compressors that are designed for scientific data, such as SZ [14, 24, 35], ZFP [27], MGARD [1], and TTHRESH [6]. These lossy compressors provide parameters that allow users to finely control the information loss introduced by lossy compression. Unlike traditional lossy compressors such as JPEG [36] for images (in integers), SZ, ZFP, MGARD, and TTHRESH are designed to compress floating-point data and can provide a strict error-controlling scheme based on the user's requirements. Generally, lossy compressors provide multiple compression modes, such as error-bounding mode and fixed-rate mode. Error-bounding mode requires users to set an error type, such as the point-wise absolute error bound and point-wise relative error bound, and an error bound level (e.g., 10^{-3}). The compressor ensures that the differences between the original data and the reconstructed data do not exceed the user-set error bound level.

In this work, we focus on the SZ lossy compression (2021 R&D 100 Award Winner [39]) because SZ typically provides higher compression ratio than ZFP [28, 42] and higher (de)compression speeds than MGARD [26, 42] and TTHRESH [6]. SZ is a prediction-based error-bounded lossy compressor for scientific data. It has three main steps: (1) predict each data point's value based on its neighboring points by using an adaptive, best-fit prediction method; (2) quantize

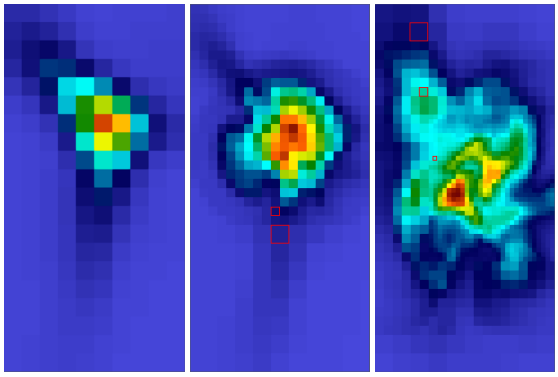


Figure 1: Visualization (one zoom-in 2D slice) of three key timesteps generated from an AMR-based cosmology simulation. The grid structure changes with the universe's evolution. The red boxes indicate different resolutions within one AMR level.

the difference between the real value and predicted value based on the user-set error bound; and (3) apply a customized Huffman coding and lossless compression to achieve a higher ratio.

2.2 AMR Method and AMR data

AMR is a method of adapting the accuracy of a solution (e.g., solving hydrodynamics equations) by using a non-uniform grid to increase computational and storage savings while still achieving the desired accuracy. AMR applications change the mesh or spatial resolution based on the level of refinement needed by the simulation and use *finer mesh in the regions with more importance/interest* and *coarser mesh in the regions with less importance/interest*. Figure 1 shows that during an AMR run, the mesh will be refined when the value meets the refinement criteria, e.g., refining a block when its norm of the gradients or maximum value is larger than a threshold [20].

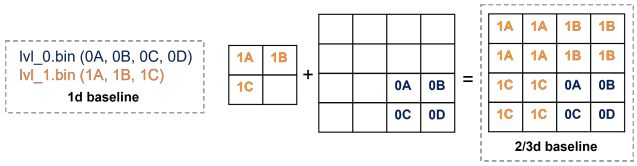


Figure 2: A typical example of AMR data storage and usage.

Clearly, the data generated by an AMR application are hierarchical data with different resolutions. The data of each AMR level are usually stored separately (e.g., in a 1D array). For example, Figure 2 (left) shows a simple example of two-level AMR data; "0" means high resolution (the fine level) and "1" for low resolution (the coarse level). When the AMR data are needed for post analysis or visualization, users will typically covert the data from different levels to a uniform resolution. In the previous example, we will up-sample the data in the coarse level and combine it with the data in the fine level, as shown in Figure 2 (right).

2.3 Existing AMR Data Compression

2.3.1 1D AMR Compression. The main challenge for AMR data compression is that the AMR data is comprehensive and hierarchical with different resolutions. A naive approach is to compress the 1D data of each AMR level separately. However, this approach loses most of the topological/spatial information, which is critical for data compression. zMesh [29] is a state-of-the-art AMR data compression based on the 1D approach. Different from the naive 1D approach, zMesh re-organizes the 1D data based on each point's coordinate in the 2D layout; in other words, zMesh puts the points neighbored in the 2D layout closer in the 1D array. It can increase the data smoothness/compressibility to benefit the following 1D compression such as SZ on the traditional patch-based [37] AMR data with redundancy. However, zMesh does not leverage high-dimensional compression, while many previous studies [35, 43] proved that leveraging more dimensional information (e.g., spatial/temporal information) can significantly improve the compression performance (e.g., compression ratio). Moreover, it only focuses on 2D patch-based AMR data. TAC aims to leverage high-dimensional data compression and supports 3D AMR data.

2.3.2 High-dimensional AMR Compression. Similar to the idea described in Section 2.2, a straightforward way to leverage 3D compression on 3D AMR data is to compress different levels together by up-sampling coarse levels. However, this approach must handle extra redundant data generated by the up-sampling process. As shown in Figure 2, *1A*, *1B*, and *1C* are redundant points in the compression. Note that the storage overhead of these redundant points will be higher when more data are in the coarse levels or up-sampling rate is higher, especially for 3D AMR data. This is because we only need to duplicate one point from the coarse level for 4 times for 2D AMR data but 8 times for 3D AMR data, with an up-sampling rate of 2. Another limitation of this approach is that it cannot apply different compression configurations (e.g., error bound) to different AMR levels, because after up-sampling all data points will have the same importance. However, the purpose of using the AMR method is to set different interests to different AMR levels, so the error bound for each AMR level can be chosen adaptively based on the analysis.

2.4 k-D Tree for Particle Data Compression

k-d tree [7] is a binary tree in which every node represents a certain space. Without loss of generality, for the 3D case, every non-leaf node in a k-d tree splits the space into two parts by a 2D plane associated with one of the three dimensions. The left subspace is associated with the left child of the node, while the right subspace is associated with the right child. k-d tree is commonly used in particle data compression [10, 13, 19] to locate each particle and remove empty regions. Specifically, a k-d tree keeps dividing the space in between along one dimension until the space is empty or contains only one particle. We will propose to optimize the classic k-d tree and use it to remove empty regions and increase the data compressibility for each AMR level (to be detailed in Section 3.2).

3 OUR PROPOSED DESIGN

In this section, we propose a pre-process approach for AMR data to leverage high-dimensional data compression algorithms in each AMR level. Specifically, we propose three pre-process strategies to

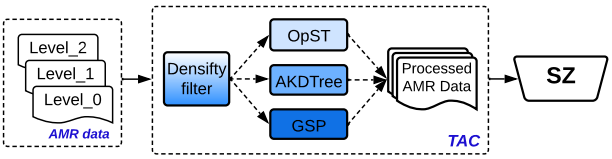
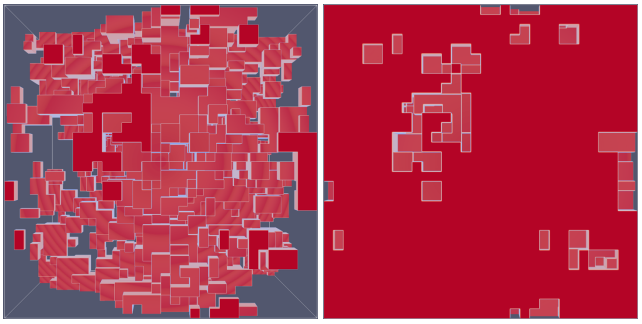


Figure 3: Workflow overview of our proposed TAC.



(a) z10 fine level (b) z10 coarse level

Figure 4: Visualization of data distributions of an example AMR data "z10", where z = redshift. Non-empty regions are shown in red.

mitigate the issue of irregular data distribution. We also propose an adaptive approach to select the best-fit pre-process strategy based on the data characteristic (e.g., density) of each AMR level. Figure 3 show the overview of our proposed TAC. It has a density filter that determines the best-fit pre-process strategy for each AMR level in the AMR dataset before compression. We will now illustrate our proposed three strategies in Section 3.1, 3.2, and 3.3, respectively.

3.1 Optimized Sparse Tensor Representation for Low-density Data

To compress the AMR data in 3D, besides the aforementioned 3D baseline, we can also compress each level separately in 3d. However, in that way, the data will be split into multiple levels, and each level will have many empty regions and an irregular data distribution, as shown in Figure 4. A naive solution to handle the irregular 3D data is to fill the empty regions with zeros and pass a large 3D block to the compressor. However, when most of the regions in the data are empty (e.g., about 77% of the data is empty in Figure 4a), we have to fill up many zeros, which would greatly increase the size of data for compression, resulting in a low compression ratio.

To solve this issue, we propose to use a naive sparse-tensor-based approach (called **NaST**) to remove the empty regions, as shown in Figure 5. NaST includes four main steps in the compression process: (1) partition the 3D data into multiple unit blocks, (2) remove the empty blocks, (3) linearize the remaining 3D blocks into a 4D array, and (4) pass the 4D array to the compressor. Note that in the decompression process, we will put the unit blocks from the decompressed 4D array back to the original data.

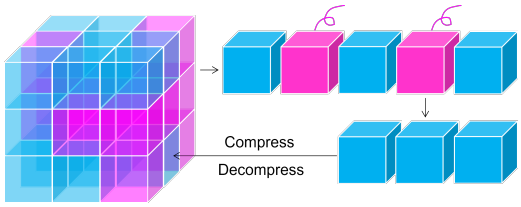


Figure 5: Workflow of the naive sparse tensor (NaST) method (empty regions marked in pink and non-empty regions marked in blue).

However, in order to completely remove the empty regions to form a sparse representation, the unit block size needs to be relatively small compared to the input data size (e.g., 16^3 vs. 512^3),

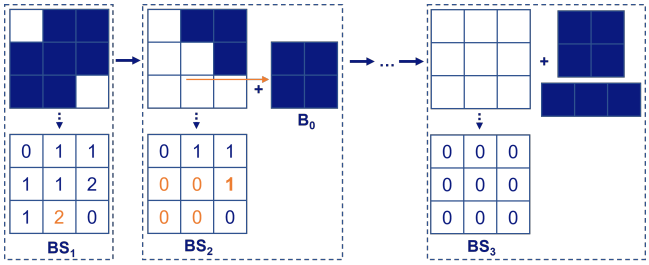


Figure 6: A 2D example of our proposed OpST approach. The sub-blocks are extracted according to our optimized sizes saved in BS. E.g., a 2-by-2 sub-block B_0 is extracted according to $BS_1[2][1]$.

resulting in a high proportion of data on the boundary. While boundary data have less information of neighboring data than non-boundary data, thus, it is harder for prediction-based compressors such as SZ to predict the boundary data values. As a result, the NaST method without optimizing the boundary data would have low compression performance.

To address the above problem, we propose an optimized sparse tensor representation (called **OpST**) to effectively remove the empty regions as well as maintain a relatively large unit block size so as to reduce the portion of boundary data. The detailed description of our algorithm can be found in Algorithm 1. We use a 2D example to demonstrate our approach, as illustrated in Figure 6. Specifically, (1) we partition the data into many small unit blocks. (2) For each unit block, we use the dynamic programming method to initiate an array BS to save the dimension/size of the maximum square whose bottom-right corner is that unit block (line 6, will be discussed in the next paragraph). (3) We extract the sub-blocks (composing of multiple unit blocks) from the original data according to the sizes saved in BS (lines 6 and 7). (4) Since the original data will be changed after the extraction, we need to partially update BS based on *maxSide* (will be discussed later). We loop (3) and (4) from the bottom-right corner to the top-left corner until the original data is empty. (5) After extracting all the sub-blocks, we put them into multiple 3D arrays (to be compressed) based on their sizes. Note that the sub-blocks with the same size will be merged into the same array for easy compression.

When initializing the BS in the step (2), we start with the $b'[i][j]$ with $i = 0$ or $j = 0$ (i.e., on the top-left edge), where $b'[\cdot][\cdot]$ are the unit blocks: if $b'[i][j]$ is empty, we will set $BS[i][j]$ to 0 otherwise 1. For the remaining unit blocks, if it is empty, $BS[i][j]$ will be 0; otherwise, $BS[i][j]$ will be set to 1 plus the minimum value among its three neighboring blocks (i.e., upper block, left block, and upper-left block). In other words, we have $BS[i][j] = 1 + \min(BS[i][j-1], BS[i-1][j], BS[i-1][j-1])$ for the 2D case. For example, $BS_1[2][1]$ is 2 because all its upper-left neighbors are 1 (as shown in Figure 6). However, both $BS_1[1][1]$ and $BS_2[1][2]$ can only reach 1 because one of their neighbors are set to 0, having no chance to form a sub-block with the size of 2.

Moreover, as mentioned in the step (3), we need to update BS after each extraction. Specifically, for each sub-block we extract, we have to set its corresponding values in BS to zeros. For instance, as shown in Figure 6, after we extract a 2-by-2 sub-block B_0 at $BS_1[2][1]$, we need to set $BS_2[1][0]$, $BS_2[1][1]$, $BS_2[2][0]$, and $BS_2[2][1]$ to zeros. In addition, we also need to recalculate a part of

Algorithm 1: Proposed Optimized Sparse Tensor Method

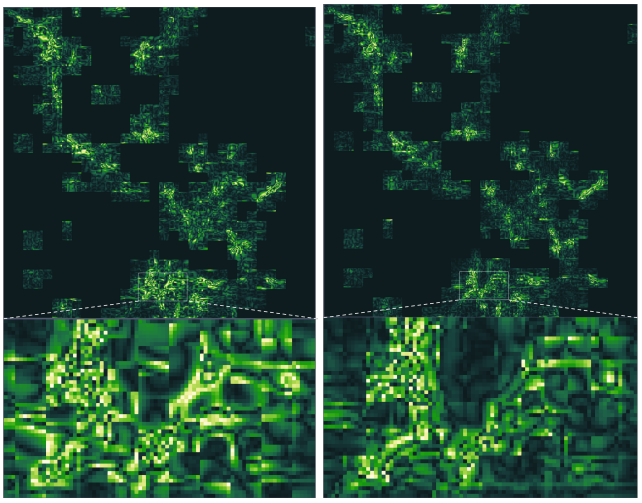
Input: Sparse 3D data S
Output: multiple 4D array D_n

1 **for** *each unit block* $b(x, y, z)$ **do**
2 **if** $b(x, y, z)$ *is non-empty* **then**
3 **if** x *is 0* ***or*** y *is 0* ***or*** z *is 0* **then**
4 $BS(x, y, z) = 1$
5 **else**
6 $BS(x, y, z) = \min(BS(x-1, y, z),\ BS(x, y-1, z),\ BS(x, y, z-1),\ BS(x-1, y-1, z),\ BS(x, y-1, z-1),\ BS(x-1, y, z-1),\ BS(x-1, y-1, z-1)) + 1$; /* BS(x,y,z) is the dimension size of the maximum cube whose bottom right rear corner is the unit block with index (x,y,z) in the original data */
7 $maxSide = \max(maxSide,\ BS(x, y, z))$
8 **end**
9 **end**
10 **end**
11 **for** *each unit block* $b(x, y, z)$ **do**
12 **if** $BS(x, y, z) \geq 1$ **then**
13 $size = BS(x, y, z)$
 $D_{size} \leftarrow S((x - size : x) * blkSize,\ (y - size : y) * blkSize,\ (z - size : z) * blkSize)$; /* put the sub-block to the according 4D array */
14 $b(x - size : x,\ y - size : y,\ z - size : z) \leftarrow empty$
 $BS(x - size : x,\ y - size : y,\ z - size : z) = 0$
 $BS = updateBs(BS,\ x,\ y,\ z,\ maxSide)$
15 **end**
16 **end**
17 return D_n

BS (line 17 in Algorithm 1) because the extraction could influence other *BS* values. For example, we need to recalculate $BS_2[1][2]$ (marked in bold orange) after extracting B_0. Note that this update is a partial update as the *BS* values to be updated will be bounded by *maxSide* which is the dimension size of the largest cube in the dataset (line 7).

Similar to the NaST method, during decompression we will put the sub-blocks back to reconstruct the data based on the saved coordinates. Note that after our optimization, each sub-block size will be relatively large (e.g., 96^3 versus the original data size of 512^3), the metadata overhead of saving the coordinates of all the sub-blocks will be negligible (e.g., 0.1%).

Finally, we show a visual comparison of the compression quality between NaST and OpST in Figure 7. Note that both use the same compressor with the same error bound. Brighter means more error. We can observe that compared to the NaST method, OpST can significantly reduce the overall compression error, especially for the data points on the boundary. It is worth noting that even with lower error, our OpST can still provide a higher compression ratio than NaST. This is because our proposed optimization will generate larger sub-blocks, which provide more information for prediction-based lossy compressors such as SZ to achieve better rate-distortion. A detailed evaluation will be shown in Section 4.



(a) NaST (CR = 233.8, PSNR = 76.9 dB) (b) OpST (CR = 241.1, PSNR = 77.8 dB)

Figure 7: Visual comparison (one slice) of compression errors of two approaches using SZ based on Nyx "baryon density" field (i.e., z10's fine level, 23% density). Brighter means higher compression error. The error bound is the relative error bound of 4.8×10^{-4}.

3.2 Adaptive k-D Tree for Medium-density Data

The OpST approach proposed for low-density data, however, has a high computation overhead, especially when the data is relatively dense. This is because, on one hand, OpST needs to update *BS* based on *maxSide* for each extraction of a sub-block, while the larger the *maxSide*, the more values in *BS* that need to be updated; on the other hand, *maxSide* is the dimension size of the largest non-empty cube in the dataset, which is highly related to the density of the dataset. Thus, the time complexity of OpST can be expressed as $O(N^2 \cdot d)$, where N is the unit block number and d is the density. Note that here density describes how dense the data is. For example, the density of 77% means that 23% of the data is empty. Clearly, when the density of an AMR level is relatively high, using OpST for compression will be relatively time-consuming.

To address the above high overhead issue of OpST, we propose an adaptive k-d tree, called **AKDTree**, to remove empty regions and extract sub-blocks (containing multiple unit blocks). AKDTree has a lower time complexity of $O(\frac{1}{3}N \cdot \log N)$ (will be discussed later). Figure 8 shows a simple 2D example. Specifically, (1) we partition the data into small unit blocks. (2) We use a tree to hierarchically represent the whole data. Each node in the tree is associated with a sub-block of the data. Moreover, each node stores the number of non-empty unit blocks in the sub-block associated with the node. (3) For each node, we split its associated sub-block from the middle along one dimension to form two sub-blocks for its two children. Note that we select one dimension which can maximize the difference of the numbers of non-empty unit blocks of the two children (will be discussed in the next two paragraphs). (4) We keep splitting a node until it has no empty unit block or itself is empty. (5) Once finishing the construction of the tree, we collect all the leaf nodes and send them to the compressor. Note that a non-empty leaf node does not have any empty unit block; otherwise, it will keep

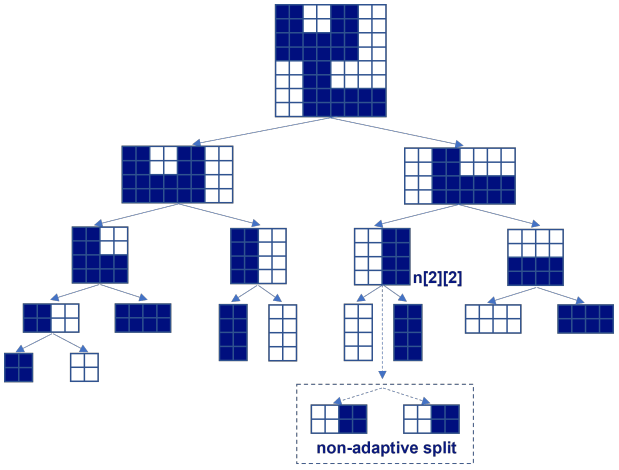


Figure 8: 2d Example of adaptive k-d tree, the sub-block will be adaptively split to in order to effectively remove the empty region as well as get bigger full sub-block.

splitting. Thus, a leaf node must be an empty or full node, as shown in Figure 8. The detailed algorithm is described in Algorithm 2.

As mentioned in the step (3), we are distributing the non-empty unit blocks unevenly to two children for each node because we attempt to get as many leaf nodes with large sub-block sizes as possible. If we keep splitting sub-blocks in a fixed way, for instance, first split along the x-axis, second split along the y-axis, third split along the x-axis, fourth split along the y-axis, and so on, we will

Algorithm 2: Dynamic k-D Tree

Input: data block d, counts information
Output: k-d tree

```
node.count ← counts information;
if d is empty or d is full then
    continue ;                                  /* stop splitting */
else
    if d is a cube then
        split d equally into 8 oct-blocks: s_1, ··· , s_8;
        get the counts c_1, ...c_8 for s_1, ··· , s_8;
        find the maxDiff partition d_1,d_2;
        node.left  = AKDTree (d_1, four c_i of d_1);
        node.right = AKDTree (d_2, four c_i of d_2);
    else if d is a flat cuboid then
        get the counts c_1, ··· , c_4 from counts information;
        find the maxDiff partition d_1, d_2;
        node.left  = AKDTree (d_1, two c_i of d_1);
        node.right = AKDTree (d_2, two c_i of d_2);
    else if d is a slim cuboid then
        get the counts c_1, c_2 from counts information;
        split d along the largest dimension to get d_1,d_2;
        node.left  = AKDTree (d_1, c_1);
        node.right = AKDTree (d_2, c_2);
end
return node;
```

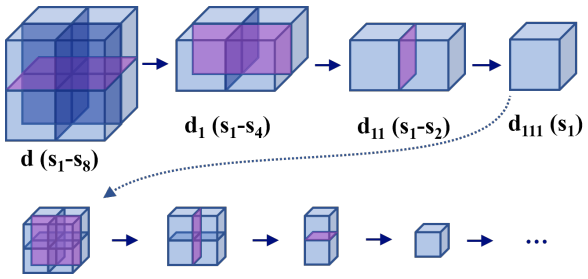


Figure 9: Example of the adaptive splitting, different shapes will have different number of choices for splitting. The process will be looped until a node is empty or full.

get a 2-by-2 sub-block for the node $n[2][2]$ as shown in the dashed box, while its largest possible sub-block could be 4 by 2.

To select one of the dimensions to unevenly distribute its non-empty unit blocks to the two children. We now present our dynamic splitting approach. We categorize nodes into three different types: "cube" nodes, "flat" nodes, and "slim" nodes, whose dimension ratios are 1:1:1, 2:2:1, 2:1:1, respectively. First of all, for the cube node d, we first divide it into eight oct-blocks, i.e., $s_1, s_2, \cdots, s_8$ (as shown in Figure 9), each sized $\frac{n}{2}^3$. Here n is the dimension size of the original data. Then, we can get the counts of non-empty unit blocks of the eight oct-blocks, i.e., $c_1, c_2, \cdots, c_8$. After that, We will decide along which dimension to split the cube node d based on the counts. Specifically, we can calculate the following three difference values:

$$\begin{aligned}\text{diff}_x &= |c_1 + c_3 + c_5 + c_7 - c_2 - c_4 - c_6 - c_8|,\\ \text{diff}_y &= |c_1 + c_2 + c_5 + c_6 - c_3 - c_4 - c_7 - c_8|,\\ \text{diff}_z &= |c_1 + c_2 + c_3 + c_4 - c_5 - c_6 - c_7 - c_8|.\end{aligned}$$

Finally, we compare these three values and choose the dimension with the maximum difference to split. For example, if the maximum difference is diff_z, we will split d along z-axis (i.e., the pink 2D plane shown in Figure 9) and get two flat nodes d_1 and d_2.

Then, for the flat nodes such as d_1, we can reuse $c_1, \cdots, c_4$ to decide whether to split d_1 along x-axis or y-axis by choosing the larger one among the following two difference values.

$$\text{diff}_x = |c_1 + c_3 - c_2 - c_4|,\ \text{diff}_y = |c_1 + c_2 - c_3 - c_4|.$$

Finally, for the slim nodes such as d_{11}, we simply split it along x-axis to get two cube nodes s_1 and s_2. This process (i.e., cube nodes→flat nodes→slim nodes) in the step (3) will be looped until the node becomes to a leaf node (i.e., empty or full).

Note that based on the above description, the counting process is required every three nodes in each three path (i.e., only for the "cube" nodes). Thanks to this dynamic splitting approach, we can lower the time complexity of the AKDTree algorithm to $O(\frac{1}{3} \cdot N \cdot \log N)$, where N is the number of unit blocks, while extracting as many relatively large sub-blocks without empty unit block as possible.

In addition, after the dynamic splitting, we will have a series of sub-blocks with the same size but different directions (e.g., 2:2:1, 2:1:2, 1:2:2). We will align the sub-blocks with the same size based on their splitting dimensions (instead of transposing them in the memory), merge them into an array, and feed multiple merged arrays to the following compression.

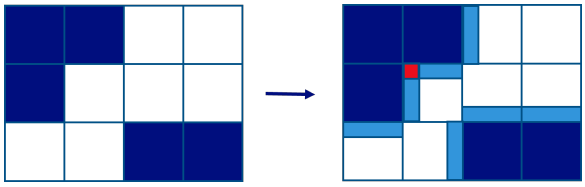


Figure 10: A 2D example of GSP approach. Non-empty blocks are in navy blue; padded blocks are in light blue/red; padded blocks based on more than one non-empty neighbors are in red.

3.3 Ghost-Shell Padding for High-density Data

For high-density data such as z10's coarse level shown in Figure 4b (i.e., about 77% density), the benefit of using our proposed OpST or AKDTree is minimal because there is not much room for removing empty regions. Meanwhile, due to the data partition/reorganization, OpST and AKDTree will hurt the data locality/smoothness.

To this end, we propose to pad zeros into the few empty regions, instead of removing them, followed by compression. However, these padded zeros can greatly reduce the performance of compression, especially for prediction-based lossy compression such as SZ, because these zeros can significantly affect the prediction accuracy of SZ, resulting in high compression errors on the boundaries, as shown in Figure 12a. More specifically, as mentioned in Section 3.1, SZ uses each point's neighboring points' values to predict its value. Thus, for those boundary points which are adjacent to padded zeros, SZ will involve zero(s) into the prediction, while the actual values of these empty regions are typically non-zeros (saved in other AMR levels), which will seriously mislead the prediction.

To eliminate the above issue of padding zeroes, we propose to use a ghost-shell padding strategy (**GSP**) to diffuse neighboring values to a padding layer. Figure 10 illustrates the high-level idea, and the detailed algorithm is described in Algorithm 3. Specifically, we still partition the data into unit blocks. Then, we will pad each empty unit block by using the average of its non-empty neighbors' boundary data values. Note that some empty unit blocks can have more than one non-empty neighbors such as the red box shown in Figure 10. For these blocks, we will use the average value of all its neighbors for padding. Correspondingly, we will remove these padded values during the decompression based on the saved padding information. Note that since the padding process is only for non-empty blocks, this metadata overhead is almost negligible for high-density data (e.g., 0.1%).

After padding, each boundary point will be predicted using the average of all the boundary data in the unit block(s) to which it belongs or is neighbored. As shown in Figure 12, compared to the zero filling (ZF) approach, GSP can significantly reduce the overall compression error, especially for the boundary data. Moreover, the GSP approach can provide a similar compression ratio to the ZF approach on this high-density data and hence a better rate-distortion. A detailed evaluation will be presented in Section 4.

3.4 Hybrid Compression Strategy

In this section, we propose a solution to adaptively choose a best-fit compression strategy from on our proposed *OpST*, *AKDTree*, and *GSP* based on the data characteristics (i.e., data density). According to Section 3.1, 3.2, and 3.3, the OpST approach is more suitable for sparse (i.e., low-density) data, while the AKDTree approach is designed to address the high time overhead of OpST when the density of data increases. When the data density is very high, the GSP approach will be used to maintain the data smoothness/locality compared to the AKDTree and OpST approaches. Therefore, we propose to use two data-density thresholds to determine when to use OpST, AKDTree, or GSP.

Algorithm 3: Proposed Ghost Shell Padding Method

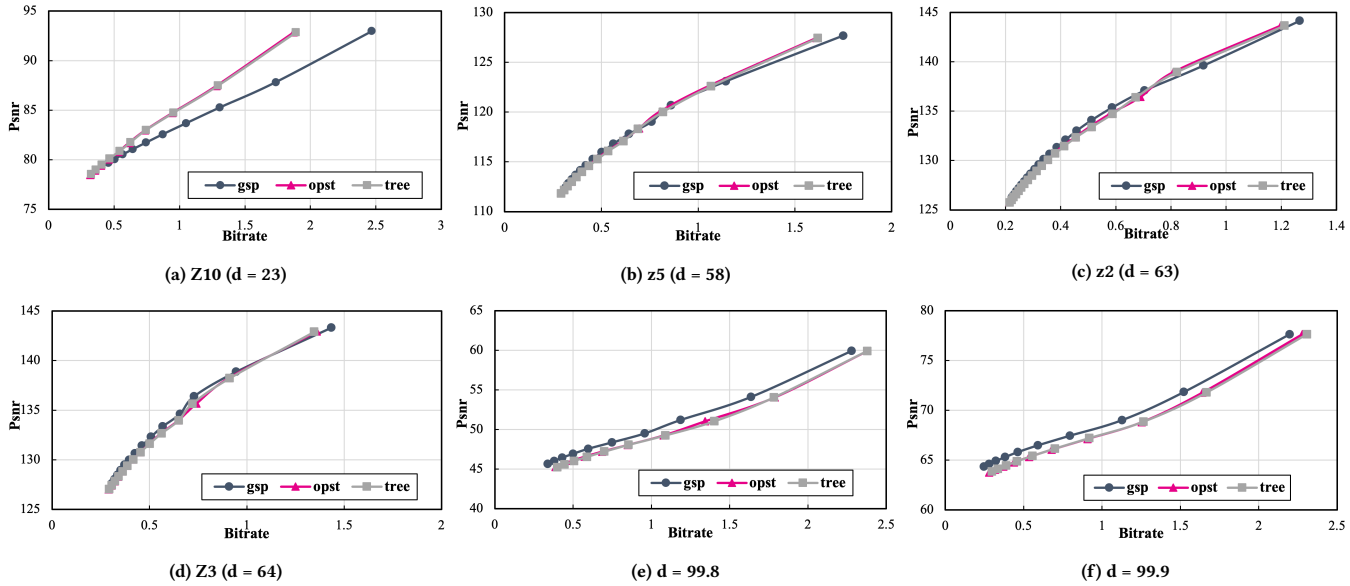
```
Input: Data, x, y
Output: Data after padding
for each unit block b_i do
    if b_i is empty and b_i has non-empty neighbor then
        for each non-empty neighbor n_j do
            pad slice = avg (first y slices of n_j next to b_i);
            if overlap edge then
                pad = pad/2;
            else if overlap corner then
                pad = pad/3;
            else
                continue;
            end
            add an x-layers pad slice to b_i next to n_j;
        end
    end
end
return padded Data
```

To decide the first threshold T_1 for switching between OpST and AKDTree, we perform a series of experiments, as shown in Figure 11. The figure shows that OpST and AKDTree have almost identical compression performance in terms of bit-rate and PSNR on all six datasets/levels (from different timesteps) with different densities. Moreover, Figure 13 shows the time costs of OpST and AKDTree (excluding compression). The figure demonstrates that the time of AKDTree is relatively stable, while the time of OpST increases linearly with the increase of data density. Overall, the only criterion for selecting OpST or AKDTree is the time cost rather than the compression performance. This is consistent with our previous design aim, that is, AKDTree is mainly designed to address the high time overhead issue of OpST. Since OpST and AKDTree have a similar speed when the density is around 50%, we propose to choose $T_1 = 50$ for choosing OpST or AKDTree.

Next, to determine the threshold T_2 for switching between AKDTree and GSP, we also evaluate them on different datasets with different densities. As shown in Figure 11, when the density is relatively low, AKDTree outperforms GSP with respect to both bit-rate and PSNR; when the density gets higher and higher, GSP gradually outperforms AKDTree. We can also observe that AKDTree and GSP have similar compression performance when the density is around 60%. Thus, we use $T_2 = 60\%$ for choosing AKDTree or GSP.

In summary, our proposed hybrid compression approach is described as follows.

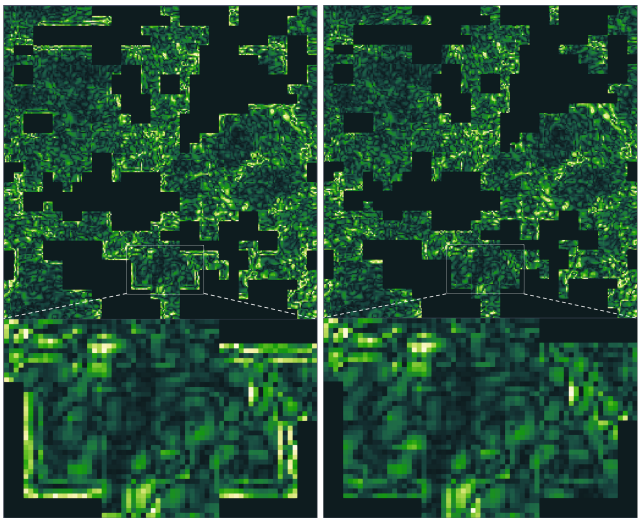
(1) When the density is smaller than $T_1 = 50\%$, we will use OpST to remove empty regions and then perform the compression;



(a) Z10 (d = 23) (b) z5 (d = 58) (c) z2 (d = 63)

(d) Z3 (d = 64) (e) d = 99.8 (f) d = 99.9

Figure 11: Compression performance comparison of GSP, OpST and AKDTree on six datasets with different densities.



(a) ZF (CR = 156.7, PSNR = 32.8 dB) (b) GSP (CR = 161.3, PSNR = 33.5 dB)

Figure 12: Visual comparison (one slice) of compression errors of two approaches using SZ based on Nyx "baryon density" field (i.e., z10's coarse level, 77% density). Brighter means higher compression error. The error bound is the relative error bound of 6.7×10^{-3}.

(2) When the density is between $T_1 = 50\%$ and $T_1 = 60\%$, we will use AKDTree to remove empty regions and then compress;
(3) When the density is larger $T_1 = 60\%$, we will use GSP to pad appropriate values and then compress the padded data.

4 EXPERIMENTAL EVALUATION

In this section, we first present our experimental setup and evaluation metrics. We then demonstrate and discuss the effectiveness of TAC in terms of both compression ratio and data quality. After that, we show the benefit of using adaptive error bound in TAC regarding post-analysis quality. Finally, we show that TAC has comparable throughput compared to comparison baselines.

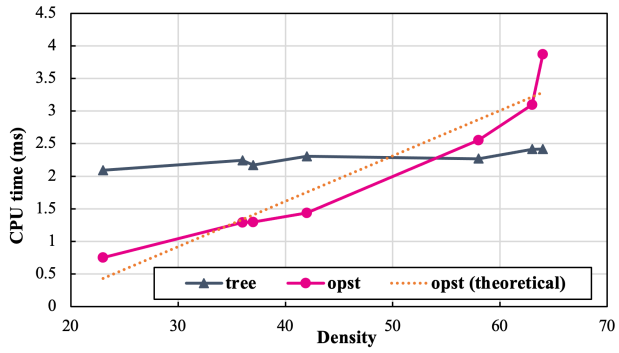


Figure 13: Time overhead comparison of OpST and AKDTree on different datasets with different densities.

4.1 Experimental Setup

Test data. Our evaluation mainly focuses on the AMReX framework [41], particularly the Nyx cosmology simulation [30]. Nyx is a state-of-the-art extreme-scale cosmology code using AMReX, which generates six fields including baryon density, dark matter density, temperature, and velocities (x, y, and z). We use seven datasets generated by two real-world simulation runs with different numbers of AMR levels, simulating a region of 64 megaparsecs (Mpc). For this data, Z is equal to the redshift, i.e. the displacement distant galaxies and celestial objects, as seen in Tab 1.

Specifically, the first run has two levels of refinement, with the coarse level of 256^3 grids and the fine level of 512^3 grids. We've collected five timesteps with the finest level density from 23% to 64%. The second run has a maximum of four levels of refinement. It was initially configured at the resolution of 128^3 and gradually refined to 1024^3. This run collected three timesteps with the coarsest-level

resolution of 256^3 (two levels), 512^3 (three levels), and finest 1024^3 (four levels), respectively. The density of the finest level varies from 0.2% to 0.003%. Note that the density of the finest level describes how much of the data in the dataset is at the highest resolution; a higher density of the finest level means that more data is refined to the highest resolution. Usually, the data density is gradually increasing at the finest level, within a single run.

Evaluation platform. The test platform is equipped two 28-core Intel Xeon Gold 6238R processors and 384 GB DDR4 memory.

Table 1: Our tested datasets.

Dataset	# Levels	Grid Size of Each Level (Fine to Coarse)	Density of Each Level (Fine to Coarse)
Run1_Z10	2	512, 256	23%, 77%
Run1_Z5	2	512, 256	58%, 42%
Run1_Z3	2	512, 256	64%, 36%
Run1_Z2	2	512, 256	63%, 37%
Run2_T2	2	256, 128	0.2%, 99.8%
Run2_T3	3	512, 256, 128	0.02%, 0.56%, 99.42%
Run2_T4	4	1024, 512, 256, 128	3E-5, 0.02%, 2.2%, 97.7%

Comparison baselines. As discussed in Section 2, we have three 1D or 3D comparison baselines. Specifically, (1) the *1D baseline (naive)*: each AMR level is compressed separately as a 1D array; (2) the *1D baseline (zMesh)* [29]: we refer readers to Section 2 for more details about how the zMesh approach reorganize the AMR data for 1D compression; and (3) the *3D baseline*: Different AMR levels are unified to the same resolution for 3D compression.

4.2 Evaluation Metrics

We will evaluate the compression performance based on the following metrics including generic and application-specific metrics.

(1) Compression ratio or bit-rate (generic, Section 4.3)
(2) Distortion quality (generic, Section 4.3)
(3) Compression throughput (generic, Section 4.6)
(4) Rate-distortion (generic, Section 4.3)
(5) Power spectrum (cosmology specific, Section 4.5)
(6) Halo finder (cosmology specific, Section 4.5)

Metric 1: To evaluate the size reduction as a result of the compression, we use the compression ratio, defined as the ratio of the original data size compared to the compressed data size, or bit-rate (bits/value), representing the amortized storage cost of each value. For a single-/double-precision floating-point data, the bit-rate is 32/64 bits per value before compression. The compression ratio and bit-rate has a mathematical relationship as their product is 32/64 so that a lower bit-rate means a higher compression ratio.

Metric 2: Distortion is another important metric used to evaluate lossy compression quality in general. We use the peak signal-to-noise ratio (PSNR) to measure the distortion quality.

$$\text{PSNR} = 20 \cdot \log_{10}(R_X) - 10 \cdot \log_{10}\left(\sum_{i=1}^{N} e_i^2 / N\right),$$

where e_i is the difference between the original and decompressed values for the point i, N is the number of points, and R_X is the value range of the dataset X. Note that higher PSNR less error.

Metric 3: Similar to prior work [21, 22, 24–26, 35, 43], we plot the rate-distortion curve to compare the distortion quality with the same bit-rate, for a fair comparison between different compression approaches, taking into account diverse compression algorithms.

Metric 4: (De)compression throughputs are critical to improving the I/O performance. We will calculate the throughput based on the original data size and (de)compression time.

Metric 5: Matter distribution in the Universe has evolved to form astrophysical structures on different physical scales, from planets to larger structures such as superclusters and galaxy filaments. The two-point correlation function $\xi(r)$, which gives the excess probability of finding a galaxy at a certain distance r from another galaxy, statistically describes the amount of the Universe at each physical scale. The Fourier transform of $\xi(r)$ is called the matter power spectrum $P(k)$, where k is the comoving wavenumber. The matter power spectrum describes how much structure exists at each physical scales. We run power spectrum on the baryon density field by using a cosmology analysis tool called Gimlet. We compare the power spectrum $p'(k)$ of decompressed data with the original $p(k)$ and accept a maximum relative error within 1% for all k < 10.

Metric 6: Halo finder aims to find the halos (over-densities) in the dark matter distribution and output the positions, the number of cells, and mass for each halo it finds, respectively. Specifically, the halo-finder algorithm [11] searches for the halos from all the simulated data, with the following two criteria: (1) the mass of a data point must be greater than a threshold (e.g., 81.66 times the average mass of the whole dataset) to become a halo cell candidate [16, 21, 22], and (2) there must be enough halo cell candidates in a certain area to form a halo. For decompressed data, some of the information (mass and cells of halos) can be distorted from the original.

4.3 Evaluation on Rate-distortion

We first evaluate the rate-distortion of TAC and compare it with the baselines on different datasets.

For the 1D baseline, as shown in Figure 14 and 15, TAC (top-left curve) outperforms the 1D baseline across all the 7 datasets. Furthermore, the performance of TAC is more stable (i.e., smoother curve) than the 1D baseline. We can also find that zMesh is slightly worse than the 1D baseline on our tested data, which will be explained in the next section.

For the 3D baseline, we can observe that TAC has much better performance when the finest level has a relatively low density or the decompressed data has a high PSNR, as shown in Figure 15. However, when the finest level has a relatively high density, TAC cannot dominate the 3D baseline as shown in Figure 14c and 14d. Specifically, in Figure 14a (the finest level density is 23%), TAC outperforms the 1D baseline when the bit-rate is larger than 1.6; in Figure 14b (the finest level density is 58%), the intersection is the bit-rate of 1.9; as the finest level density continues to grow up to 63 and 64 in Figure 14c and 14d, TAC is slightly worse than the 3D baseline until the bit-rate is larger than 2.5. In the next section, we will discuss why the 3D baseline is slightly better in the datasets of which finest level has a very high density in detail later and will also propose a solution to adaptively use the 3D baseline and TAC.

4.4 Discussion on Comparison with Baselines

On compression, zMesh is meant to improve the smoothness of the block-structured AMR datasets by taking advantage of the

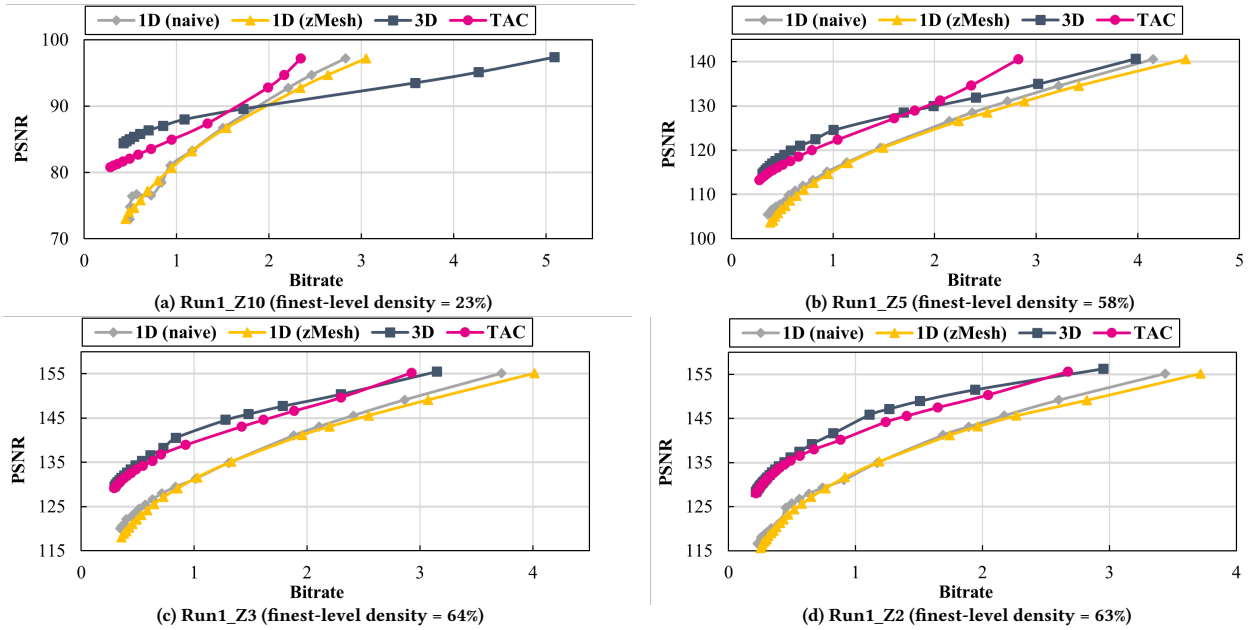


Figure 14: Rate-distortion comparison of TAC and baselines on the early time-step (Z10) to the late time-step (Z2) from run1.

data redundancy between each AMR level (as described in the introduction).

Thus, zMesh cannot improve the smoothness if there is no data redundancy in the tree-structured AMR datasets (i.e., our tested datasets). A simple example is used to illustrate this in Figure 16b, where the finer-level data has higher values because a grid will be refined only if its value is larger than a certain threshold. For block-based AMR, when a grid needs to be refined because of its high value, the value will still remain in the level, resulting in a redundant value saved (i.e., the red 8). If one uses the original z-ordering to traverse the data level-by-level (shown in Figure 16b), the reordered data will have three significant value changes (i.e., from 2 to 8, from 8 to 1, and from 1 to 9).

To solve this issue, zMesh traverses the two AMR levels together based on the layout of the 2D array. The reordered data are "1-2-8-9-8-7-8-1", which only has two significant value changes (i.e., from 2 to 8 and from 8 to 1). Thus, zMesh can improve the smoothness/compressibility for block-structured AMR data. However, as shown in Figure 16a, for tree-structured AMR data (without saving a redundant "8"), compared to the 1D baseline that compresses each level separately, zMesh introduces two significant data changes (i.e., from 2 to 9 and from 8 to 1) as it traverses between two AMR levels. This explains why zMesh is slightly worse than the 1D baseline on our tested AMR datasets.

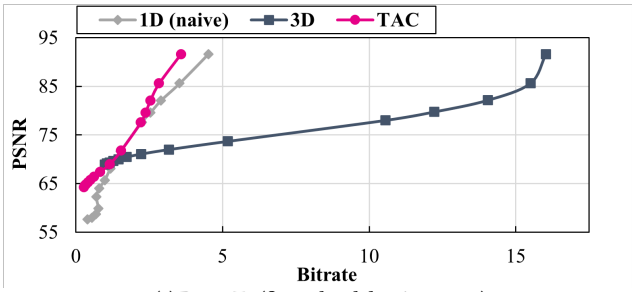
When considering a 3D baseline, we found that it works slightly better than an adaptive compression approach. First, from the high level, if the finest level of an AMR dataset has a very high density, it means that this dataset is not much different from a non-AMR dataset with uniform resolution. Thus, there is no need to use TAC. Instead, we can directly use the 3D baseline that up-samples coarse-level data and compresses the merged uniform data. This is because the main disadvantage of the 3D baseline is the redundant data generated by the up-sampling process; however, when the finest level is very dense, the coarse levels do not have much data to up-sample, thus the overhead of redundant upsampled data is almost negligible. On the other hand, compression on the uniform-resolution data (the 3D baseline) can better leverage the spatial information than the level-wise compression (TAC).

For an example of a two-level 2D AMR dataset as shown in Figure 17. Its finest and coarse levels have the grids of 512^2 and 256^2, respectively. If the density of the finest level is larger than 60% (e.g., 75% in the 2D example), TAC applies the GSP strategy to the finest level. In that way, the data points to compress in the finest level and the coarse level are 512^2 and $0.25 \cdot 256^2$, respectively. However, by simply using the 3D baseline, after up-sampling and merge, there are totally 512^2 data points to compress. Therefore, instead of padding values to the finest level (i.e. GSP), we can simply fill in the up-sampled coarse levels to save the extra space of compressing the coarse levels separately and increase the smoothness/compressibility of the dataset.

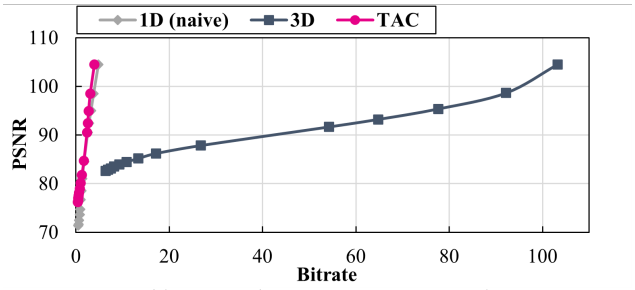
Overall, we propose to adaptively use the 3D baseline and TAC based on the density of the finest level of an AMR dataset as follows: (1) check the finest level's density; (2) use the 3D baseline to compress the data if the density meets the threshold T_2 we set, and (3) use TAC (OpST, AKDTree, and GSP) if the density does not meet the threshold.

4.5 Evaluation on Post-analysis Quality with Adaptive Error Bound

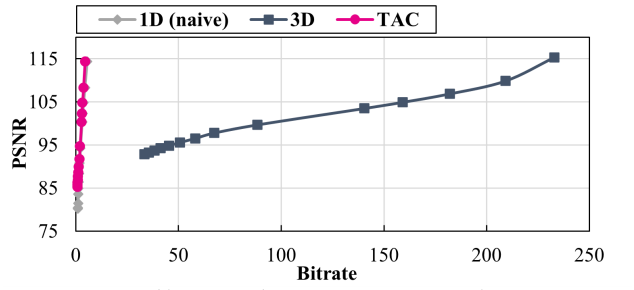
We now evaluate TAC with the two cosmology-specific post-analysis metrics (i.e., metrics 5 and 6: power spectrum and halo finder) to demonstrate the benefit of the adaptive error bound method. When factoring level-wise compression, TAC can apply different error bounds to different AMR levels based on (1) the post-analysis metrics, (2) the up-sampling rates of coarse levels, and (3) the rate-distortion trade-off between different AMR levels.We choose the



(a) Run2_T2 (finest-level density = 0.2%)

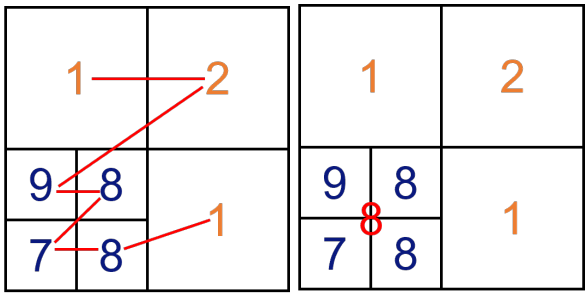


(b) Run2_T3 (finest-level density = 0.02%)



(c) Run2_T4 (finest-level density = 3E-5)

Figure 15: Rate-distortion comparison of TAC (top-left) and baselines on different time-steps from run2.



(a) Tree-structured AMR data (b) Block-structured AMR data

Figure 16: An example of how the 1D baseline, zMesh, and original z-order reorder a simple 2D AMR data without and with redundancy. Orange: coarse level , blue: fine level, red: redundant data.

dataset run1-Z2 for evaluation because TAC is slightly worse than the 3D baseline on this dataset.

Figure 18 shows the motivation of performing rate-distortion trade-off between different AMR levels. As the error bounds for

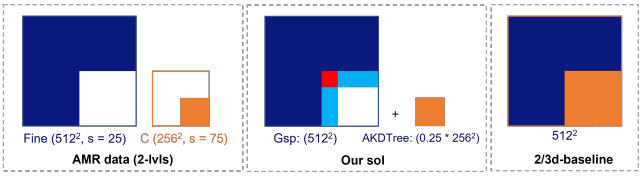


Figure 17: Comparison between the 3D baseline and TAC on an example AMR dataset with the dense finest level.

the fine and coarse levels increase, their bit rates will converge to a similar value. This means that when the error bound is relatively large, the reduction in data size will be insignificant compared to the compression error increment (i.e., the slopes of both curves are very small). Therefore, we can say that when the error is large, it is not worth trading data quality for size reduction.

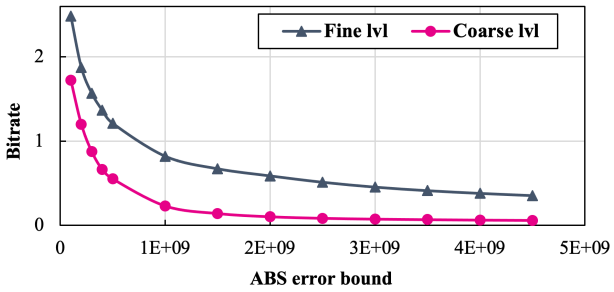


Figure 18: Bit-rates with different error bounds using SZ lossy compression for fine and coarse levels on Run1_Z2 dataset.

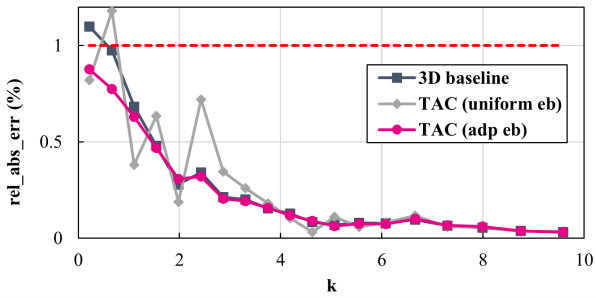


Figure 19: Power spectrum error (in relative) of the 3D baseline and TAC (the same error bound for all AMR levels) and TAC (different error bounds for different AMR levels) on baryon density field on run1-Z2. The red dashed line is the 1% limit of acceptable power spectrum error.

Power Spectrum. Figure 19 shows that, under the (almost) same compression ratio, TAC (with the uniform error bound) has a similar power-spectrum error compared to the 3D baseline.

Now, let us follow the three steps mentioned at the beginning of this section to adjust the error bound for each AMR level. First, the post-analysis metric–power spectrum—needs to be run on the uniform-resolution data and focuses on the global quality of data. Thus, the ideal error-bound configuration/ratio for the fine and coarse levels on the uniform-resolution data would be 1:1.

Table 2: Overall compression/decompression throughput (MB/s) of different approaches with different absolute error bounds.

EB_{abs}	Run1_Z2			Run1_Z3			Run1_Z5			Run1_Z10			Run2_T2			Run2_T3			Run2_T4		
	1D	3D	TAC	1D	3D	TAC	1D	3D	TAC	1D	3D	TAC	1D	3D	TAC	1D	3D	TAC	1D	3D	TAC
1E+08	169	94	97	166	90	94	161	76	99	160	40	95	152	17	76	143	2.4	60	125	0.4	30
1E+09	219	115	121	213	120	127	208	109	123	208	63	117	193	27	91	184	3.9	66	159	0.5	32
1E+10	259	125	135	256	125	136	253	117	137	250	65	135	242	30	102	229	4.0	72	197	0.5	34

As aforementioned, the coarse level of the AMR dataset needs to be up-sampled to uniform the resolution. As a result, the compression error of the coarse level will be up-sampled as well, resulting in more error to the post analysis. Thus, we then need to give the coarse level a smaller error bound based on the up-sample rate. Here the up-sample rate for Z2's coarse level is 2^3, leading to an ideal error-bound ratio of the fine and coarse levels changed to 8:1.

Finally, this 8:1 ratio needs to be adjusted based on the rate-distortion trade-off as aforementioned. As shown in Figure 19, when using the error-bound ratio of 8:1 (e.g., 4E+9 for the fine level and 5E+8 for the coarse level), the error bound of the fine level is too large, resulting in an ineffective rate-distortion trade-off. Thus, we can balance two levels by increasing the error bound for the coarse level (to gain compression ratio) and decreasing the error bound for the fine level (to add compression error), which can achieve an overall rate-distortion benefit. Based on our experiments, we adjust the error-bound ratio from 8:1 to 3:1 and can observe that TAC has a significant improvement in the power spectrum error and outperforms the 3D baseline.

Halo finer. We evaluate the mass change, and the number of cells change for the biggest halo identified using the 3D baseline, TAC (with uniform error bound), and TAC (with adaptive error bound), as shown in Table 3. We can see that TAC with adaptive error bound produces better halo-finer analysis quality than the 3D baseline.

Table 3: Halo finder analysis with different methods.

	CR	Rel Mass Diff	Cell Nums Diff
3D baseline	198.5	6.66E-04	39.00
TAC (1:1)	198.5	4.97E-04	28.00
TAC (2:1)	**198.6**	**4.49E-04**	**25.00**

Similar to the error-bound configuration analysis done for power spectrum, let us now adjust the error-bound ratio between the fine and coarse levels for halo finder. The halo-finer analysis also requires a uniform-resolution data as input. However, different from the power-spectrum analysis, the halo-finder analysis focuses more on high-value points in the fine level, since only high-value data points qualify as halo candidates, as described in Section 4.2. Note that this does not mean we can directly discard the coarse-level data with small values as they still contribute to the average value of the dataset, which is also an important parameter for the halo finder [11]. Therefore, we set the ideal error-bound ratio to 1:2 (i.e., fine level v.s. coarse level) for the uniform-resolution data based on our massive experiments.After that, considering the up-sampling rate of 2^3, the error-bounded ratio is changed to 4:1. Finally, we adjust the ratio to 2:1 based on the rate-distortion trade-off. Overall, as we can see in Table 3, TAC with adaptive error bound obtains the minimal differences of the mass and cell numbers.

4.6 Evaluation on Time Overhead

We evaluate the overall throughput (including pre-processing, compression, and decompression) on all the datasets with different error bounds. As shown in Table 2 compared to the 3D baseline, the throughput of TAC is up to 75× higher than on the Run2 datasets and 2.4× higher on the Run1 datasets. This is because the Run2 datasets have lower density than the Run1 datasets in the finest level, resulting in a higher overhead of redundant data for the 3D baseline, which is consistent with our discussion in Section 4.4. Moreover, TAC is slightly worse than the 1D baseline on the Run1 datasets due to the pre-possessing overhead. While we note that on the T3 and T4 datasets, our throughput drops due to a relatively heavy launching time (for compressing multiple 4D arrays generated by OpST) compared to the overall time on the small-sized datasets. Note that we exclude zMesh during the evaluation as it is theoretically slower than the 1D baseline due to the extra z-ordering and provides worse rate-distortion according to our evaluation.

5 CONCLUSION AND FUTURE WORK

In conclusion, this paper proposes an error-bounded lossy compression for 3D AMR data, called TAC. It leverages 3D compression for AMR data on a systemic level. We propose three pre-processing strategies that can adapt based on the density of each AMR level. Our approach improves the compression ratio compared to the state-of-the-art approach by up to 3.3× under the same data quality loss. With our level-wised compression approach, we are able to tune the error-bound ratio of fine and coarse levels to be 3:1 and 2:1 for better power-spectrum and halo-finder analyses, respectively, under the same compression ratio.

In future work, we will apply our hybrid compression approach to more AMR simulations. We will also address the issue of relatively low throughput on small AMR datasets.

ACKNOWLEDGMENTS

This work has been authored by employees of Triad National Security, LLC which operates Los Alamos National Laboratory under Contract No. 89233218CNA000001 with the U.S. Department of Energy/National Nuclear Security Administration. This research was supported by the Exasky Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. This research was supported by the U.S. National Science Foundation under Grants OAC-2042084 and OAC-2104024. We would like to thank Dr. Zarija Lukić from the NYX team at Lawrence Berkeley National Laboratory for granting us access to cosmology datasets.

REFERENCES

[1] M Ainsworth, O Tugluk, B Whitney, and S Klasky. 2017. MGARD: A Multilevel Technique for Compression of Floating-Point Data. In *DRBSD-2 Workshop at Supercomputing*.

[2] Ann S Almgren, John B Bell, Mike J Lijewski, Zarija Lukić, and Ethan Van Andel. 2013. Nyx: A massively parallel amr code for computational cosmology. *The Astrophysical Journal* 765, 1 (2013), 39.

[3] AMReX: Building a Block-Structured AMR Application (and More). 2020. https://extremecomputingtraining.anl.gov/files/2019/08/ATPESC_2019_Track-5_5_8-6_11am_Almgren-AMReX.pdf.

[4] Allison H Baker, Haiying Xu, John M Dennis, Michael N Levy, Doug Nychka, Sheri A Mickelson, Jim Edwards, Mariana Vertenstein, and Al Wegener. 2014. A methodology for evaluating the impact of data compression on climate simulation data. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 203–214.

[5] Allison H Baker, Haiying Xu, Dorit M Hammerling, Shaomeng Li, and John P Clyne. 2017. Toward a multi-method approach: Lossy data compression for climate simulation data. In *International Conference on High Performance Computing*. Springer, 30–42.

[6] Rafael Ballester-Ripoll, Peter Lindstrom, and Renato Pajarola. 2020. TTHRESH: Tensor Compression for Multidimensional Visual Data. *IEEE Transactions on Visualization and Computer Graphics* 26, 9 (2020), 2891–2903. https://doi.org/10.1109/TVCG.2019.2904063

[7] Jon Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *communications of the ACM September, 1975. vol. 18: pp. 509-517 : ill. includes bibliography.* 18 (01 1975).

[8] Carsten Burstedde, Omar Ghattas, Georg Stadler, Tiankai Tu, and Lucas C Wilcox. 2008. Towards adaptive mesh PDE simulations on petascale computers. *Proceedings of Teragrid* 8 (2008).

[9] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. 2019. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications* (2019).

[10] Gabriel Cirio, Guillaume Lavoué, and Florent Dupont. 2010. A Framework for Data-driven Progressive Mesh Compression. *GRAPP 2010 - Proceedings of the International Conference on Computer Graphics Theory and Applications*, 5–12.

[11] Marc Davis, George Efstathiou, Carlos S Frenk, and Simon DM White. 1985. The evolution of large-scale structure in a universe dominated by cold dark matter. *The Astrophysical Journal* 292 (1985), 371–394.

[12] Peter Deutsch. 1996. *GZIP file format specification version 4.3.* Technical Report.

[13] Olivier Devillers and Pierre-Marie Gandoin. 2000. Geometric Compression for Interactive Transmission. *Proc. Visualization '00* (01 2000). https://doi.org/10.1109/VISUAL.2000.885711

[14] Sheng Di and Franck Cappello. 2016. Fast error-bounded lossy HPC data compression with SZ. In *2016 ieee international parallel and distributed processing symposium (ipdps)*. IEEE, 730–739.

[15] Anshu Dubey, Ann Almgren, John Bell, Martin Berzins, Steve Brandt, Greg Bryan, Phillip Colella, Daniel Graves, Michael Lijewski, Frank Löffler, et al. 2014. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3217–3227.

[16] Bo Fang, Daoce Wang, Sian Jin, Quincey Koziol, Zhao Zhang, Qiang Guan, Surendra Byna, Sriram Krishnamoorthy, and Dingwen Tao. 2021. Characterizing Impacts of Storage Faults on HPC Applications: A Methodology and Insights. 409–420. https://doi.org/10.1109/Cluster48925.2021.00048

[17] Ali Murat Gok, Sheng Di, Yuri Alexeev, Dingwen Tao, Vladimir Mironov, Xin Liang, and Franck Cappello. 2018. Pastri: Error-bounded lossy compression for two-electron integrals in quantum chemistry. In *2018 IEEE international conference on cluster computing (CLUSTER)*. IEEE, 1–11.

[18] Pascal Grosset, Christopher Biwer, Jesus Pulido, Arvind Mohan, Ayan Biswas, John Patchett, Terece Turton, David Rogers, Daniel Livescu, and James Ahrens. 2020. Foresight: analysis that matters for data reduction. In *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 1171–1185.

[19] Duong Hoang, Harsh Bhatia, Peter Lindstrom, and Valerio Pascucci. 2021. High-Quality and Low-Memory-Footprint Progressive Decoding of Large-Scale Particle Data. 32–42. https://doi.org/10.1109/LDAV53230.2021.00011

[20] IS&T Co-Design Summer School. 2021. https://www.lanl.gov/projects/codesign/codesign-summer-school/research-areas/adaptive-mesh-refinement.php.

[21] Sian Jin, Pascal Grosset, Christopher M Biwer, Jesus Pulido, Jiannan Tian, Dingwen Tao, and James Ahrens. 2020. Understanding GPU-based lossy compression for extreme-scale cosmological simulations. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 105–115.

[22] Sian Jin, Jesus Pulido, Pascal Grosset, Jiannan Tian, Dingwen Tao, and James Ahrens. 2021. Adaptive configuration of in situ lossy compression for cosmology simulations via fine-grained rate-quality modeling. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*. 45–56.

[23] Didier Le Gall. 1991. MPEG: A video compression standard for multimedia applications. *Commun. ACM* 34, 4 (1991), 46–58.

[24] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2018. An Efficient transformation scheme for lossy data compression with point-wise relative error bound. In *CLUSTER*. IEEE, Belfast, UK, 179–189.

[25] Xin Liang, Sheng Di, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2018. An efficient transformation scheme for lossy data compression with point-wise relative error bound. In *2018 IEEE International Conference on Cluster Computing*. IEEE, 179–189.

[26] Xin Liang, Qian Gong, Jieyang Chen, Ben Whitney, Lipeng Wan, Qing Liu, David Pugmire, Rick Archibald, Norbert Podhorszki, and Scott Klasky. 2021. Error-controlled, progressive, and adaptable retrieval of scientific data with multilevel decomposition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[27] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014), 2674–2683.

[28] Tao Lu, Qing Liu, Xubin He, Huizhang Luo, Eric Suchyta, Jong Choi, Norbert Podhorszki, Scott Klasky, Matthew Wolf, Tong Liu, and Zhenbo Qiao. 2018. Understanding and modeling lossy compression schemes on HPC scientific data. In *2018 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 348–357.

[29] Huizhang Luo, Junqi Wang, Qing Liu, Jieyang Chen, Scott Klasky, and Norbert Podhorszki. 2021. zMesh: Exploring Application Characteristics to Improve Lossy Compression Ratio for Adaptive Mesh Refinement. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 402–411.

[30] Nyx. 2021. https://github.com/AMReX-Astro/Nyx.

[31] Brandon Runnels, Vinamra Agrawal, Weiqun Zhang, and Ann Almgren. 2021. Massively parallel finite difference elasticity using block-structured adaptive mesh refinement with a geometric multigrid solver. *J. Comput. Phys.* 427 (2021), 110065.

[32] Seung Woo Son, Zhengzhang Chen, William Hendrix, Ankit Agrawal, Wei-keng Liao, and Alok Choudhary. 2014. Data compression for the exascale computing era-survey. *Supercomputing Frontiers and Innovations* 1, 2 (2014), 76–88.

[33] James M Stone, Kengo Tomida, Christopher J White, and Kyle G Felker. 2020. The Athena++ adaptive mesh Refinement framework: Design and magnetohydrodynamic solvers. *The Astrophysical Journal Supplement Series* 249, 1 (2020), 4.

[34] Knut Sverdrup, Nikolaos Nikiforakis, and Ann Almgren. 2018. Highly parallelisable simulations of time-dependent viscoplastic fluid flow with structured adaptive mesh refinement. *Physics of Fluids* 30, 9 (2018), 093102.

[35] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. 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 (IPDPS)*. IEEE, 1129–1139.

[36] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE Transactions on Consumer Electronics* 38, 1 (1992), xviii–xxxiv.

[37] Feng Wang, Nathan Marshak, Will Usher, Carsten Burstedde, Aaron Knoll, Timo Heister, and Chris Johnson. 2020. CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data. *Computer Graphics Forum* 39 (06 2020), 1–12. https://doi.org/10.1111/cgf.13958

[38] S Whitman, J Brasseur, and P Hamlington. 2018. Simulation of Bluff-Body Stabilized Flames with PeleC, an Exascale Combustion Code.

[39] R&D World. 2021. 2021 R&D 100 Award Winners - SZ: A Lossy Compression Framework for Scientific Data. https://www.rdworldonline.com/rd-100-2021-winner/sz-a-lossy-compression-framework-for-scientific-data/.

[40] Xin-Chuan Wu, Sheng Di, Emma Maitreyee Dasgupta, Franck Cappello, Hal Finkel, Yuri Alexeev, and Frederic T Chong. 2019. Full-state quantum circuit simulation by using data compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–24.

[41] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, et al. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 4, 37 (2019), 1370–1370.

[42] Kai Zhao, Sheng Di, Maxim Dmitriev, Thierry-Laurent D Tonellot, Zizhong Chen, and Franck Cappello. 2021. Optimizing error-bounded lossy compression for scientific data by dynamic spline interpolation. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1643–1654.

[43] Kai Zhao, Sheng Di, Xin Liang, Sihuan Li, Dingwen Tao, Zizhong Chen, and Franck Cappello. 2020. Significantly improving lossy compression for HPC datasets with second-order prediction and parameter optimization. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. 89–100.

[44] Zstandard. 2020. http://facebook.github.io/zstd/.