zMesh: Exploring Application Characteristics to Improve Lossy Compression Ratio for Adaptive Mesh Refinement

Huizhang Luo*, Junqi Wang†, Qing Liu*, Jieyang Chen‡, Scott Klasky‡, Norbert Podhorszki‡
* New Jersey Institute of Technology, Newark, NJ, USA
† Rutgers University-Newark, Newark, NJ, USA
‡ Oak Ridge National Laboratory, Oak Ridge, TN, USA
huizhang.luo@njit.edu, junqi.wang@rutgers.edu, qliu@njit.edu, {chenj3, klasky, pnorbert}@ornl.gov

***Abstract*—Scientific simulations on high-performance computing systems produce vast amounts of data that need to be stored and analyzed efficiently. Lossy compression significantly reduces the data volume by trading accuracy for performance. Despite the recent success of lossy compression, such as ZFP and SZ, the compression performance is still far from being able to keep up with the exponential growth of data. This paper aims to further take advantage of application characteristics, an area that is often under-explored, to improve the compression ratios of adaptive mesh refinement (AMR) - a widely used numerical solver that allows for an improved resolution in limited regions. We propose a level reordering technique *zMesh* to reduce the storage footprint of AMR applications. In particular, we group the data points that are mapped to the same or adjacent geometric coordinates such that the dataset is smoother and more compressible. Unlike the prior work where the compression performance is affected by the overhead of metadata, this work re-generates restore recipe using a chained tree structure, thus involving no extra storage overhead for compressed data, which substantially improves the compression ratios. The results demonstrate that zMesh can improve the smoothness of data by 67.9% and 71.3% for Z-ordering and Hilbert, respectively. Overall, zMesh improves the compression ratios by up to 16.5% and 133.7% for ZFP and SZ, respectively. Despite that zMesh involves additional compute overhead for tree and restore recipe construction, we show that the cost can be amortized as the number of quantities to be compressed increases.**

***Index Terms*—High-performance computing (HPC), data storage, lossy compression, adaptive mesh refinement (AMR)**

I. Introduction

As the fidelity of scientific simulations continues to grow as empowered by the next generation high-performance computing (HPC) systems, a major challenge that domain scientists are faced with is how to store and analyze the vast volume of simulation outputs efficiently to extract new knowledge. For example, the Community Earth Simulation Model (CESM) running on the Yellowstone supercomputer produces a total of 170 terabytes of data for the fifth Coupled Model Intercomparison Project (CMIP5) run. Such a high data volume poses a multitude of storage and data analysis challenges, and renders even a simple step in its workflow (e.g., transposing the data format) a lengthy process.

To address this challenge, various approaches have been attempted, including floating-point data compression [1], [2], [3], [4], in-situ data analysis [5], [6], [7], new storage architecture and I/O methods [8], [9], [10], [11]. Among them, data compression is considered to be a fundamental yet effective method to lower the cost of data movement. Depending on whether there is information loss during compression, floating-point data compression can be either lossless [12], [13] or lossy [2], [3], [4], with the central theme of exploiting the correlation between neighboring data points and compressing either the residual or the decorrelated values. Compared to lossless compression that preserves the exact content of original data, lossy compression offers much higher compression ratios by trading accuracy for performance, and is deemed to be the path forward to handling extreme-scale data. Despite the recent success in this direction, such as ZFP [3] and SZ [4], [14], the compression ratios are still not attractive enough to be indispensable in scientific processes. Therefore, this work aims to further explore ideas and techniques to improve the compression ratios of lossy compressors. One root cause of the limited compression performance is that floating-point compressors are designed around the presumptive local smoothness in data [15], which may be insignificant.

This paper studies data reduction for adaptive mesh refinement (AMR), a widely used numerical technique for solving partial differential equations [16], [17]. The central idea of AMR is to place higher resolutions in selected regions that have higher local truncation error, thereby greatly reducing the overall computational complexity and storage overhead, as compared to the case where a high resolution is enforced everywhere. The motivation behind studying AMR data compression is threefold: First, AMR represents a large set of HPC applications that is highly data intensive, for which reducing data is necessary to lower the I/O cost. It is widely adopted in various science and engineering domains, and therefore we believe improving the AMR compression ratios is beneficial to a broad range of applications. Second, AMR is unique in its data model, featured by a set of hierarchical grids (often called boxes in AMR) with different grid spacing, and brings additional challenges and opportunities for reduction. Intuitively, there exists high information redundancy among grids across levels that can be exploited for compression. However, the native organization of AMR data does not easily expose this redundancy to compressors (Section II-A), thus leading to low compression ratios. Last but not least, the architectural trend that compute has become increasingly cheap as compared to I/O on HPC systems [18] motivates us to use additional compute cycles to preprocess data to improve the compressibility of data. This work aims to further improve

the compression ratios of AMR-based applications using lossy compression, leveraging the inherent information redundancy within the AMR output. Specifically, this paper makes the following contributions.

- We propose a level reordering technique for AMR, called *zMesh*, to improve the local smoothness within a dataset[1]. Unlike prior reordering methods that maintain a restore recipe as part of the compressed data, this work can regenerate the restore recipe using a chained tree structure during decompression. Therefore, there is no additional metadata associated with reordering in the compressed data, which substantially improves the compression ratios.
- We further develop zMesh in the context of space-filling curves to improve the compression ratios of existing compressors. While both zMesh and space-filling curve aim to improve the locality, zMesh further leverages the redundancy across levels and therefore its merits are orthogonal to space-filling curve.
- We evaluate the effectiveness of zMesh with real AMR applications in Chombo [19], a well adopted AMR framework developed at Lawrence Berkeley National Laboratory, and provide in-depth analyses with regard to its improvement over existing compressors, the introduced overhead, and data subsetting performance.

The remainder of this paper is organized as follows. Section II discusses the background and related work, along with motivation in Section III. Section IV presents the design and implementation of zMesh. Section V evaluates the proposed zMesh in terms of improvements of smoothness and compression ratios, overhead and the ability of accessing the compressed data. Section VI presents the conclusions.

II. Background and Related Work

A. Adaptive Mesh Refinement

In simulation-based scientific discovery, many science problems do not require a uniform accuracy across the entire problem domain, for example, those spatial regions in a simulation that have small truncation errors or do not contain interesting physical phenomena. AMR provides a framework level solution to generate and evolve a set of hierarchical grids in an adaptive manner. Compared to the uniform grid, the key advantages of AMR are the substantial computational and storage savings as well as the adaptive control of grid resolution. Fig. 1 illustrates an example of AMR output with two levels of fidelity. The grid in black (level 0) represents the original problem domain with a coarse grid spacing, whereas the grid in blue (level 1) is a fine grid which has a half grid spacing, covering a subset of the problem domain. A key motivation of this work is that, in AMR the data points of a finer level are identified and computed directly from the coarser level, and therefore there exists correlations between a fine and a coarse grid, which can be exploited for compression.

[1]The source code is available at https://github.com/HNUHPC/AMR.

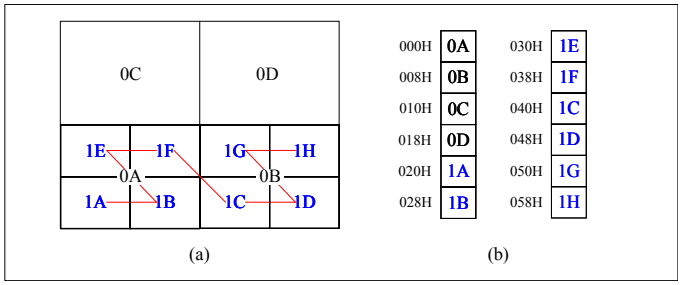


Fig. 1: AMR data layout (Z-ordering). (a) logical layout. (b) physical layout on storage. The level 0 grid consists of data points 0A to 0D, while the level 1 grid consists of 1A to 1H. After serialization, the data locality across levels is lost. For example, after Z-ordering, grid points 1A, 1B, 1E and 1F are not physically adjacent to 0A, despite that they capture physical quantities (e.g., temperature, pressure) over the similar region.

AMR employs sophisticated data layout to manage the hierarchical grids. For example, rather than row-wise ordering, various space-filling curves, such as Z-ordering (Morton) [20], Hilbert curve [21] and Moore curve [22], were proposed to improve the performance of spatiotemporal data retrieval. However, the current methods will traverse data level by level, and none of them have considered the similarities among levels when compressing data. Fig. 1 shows an example of a 2-level AMR layout using Z-ordering. It maps the multi-dimensional data to one dimension through a Z-shaped traversal, as shown in Fig. 1(a). Despite that Z-ordering improves the spatial locality versus the simple row or column ordering, it fails to capture the locality among AMR levels. For example, grid points 1A-1B-1E-1F and 0A capture the quantities (e.g., pressure, temperature) over the same physical regions, and therefore should have similar values. However, after Z-ordering, these grid points are dispersed, which can harm the local smoothness.

B. Lossy Compression

Lossy compression of scientific data [2], [3], [4] has received renewed interest recently, largely due to the need to manage the ever-increasing data volume and the promising performance achieved by lossy compression. The main idea of lossy compression is to reduce the data volume with inexact approximations and partial data discarding. ZFP [3] transforms the original data to the frequency domain so that the unimportant components can be selectively discarded within a given error bound. In particular, it partitions the entire dataset into fixed-sized blocks and each block is further converted to mantissas along with a common exponent. The mantissas are then converted into fixed-point signed integers, followed by an orthogonal transform to generate near-zero coefficients, which are encoded using embedded coding for each bit plane. In contrast, SZ uses a 1D curve-fitting predictor [4] or a multi-dimensional Lorenzo predictor [23] to predict the data value for each data point using its neighboring points. If a data point can be curve-fitted, i.e., the difference between the original data and the prediction is within the given error bound, a linear-scaling quantization method is used to encode the data value. The quantized data are further compressed

TABLE I: AMR applications tested.

Application	Description and configuration
PineIsland-Glacier	The simulation of the fastest melting glacier in Antarctica, the application uses real data of Pine Island. Problem domain: 256km×384km; Size of the base grid: 2km×h2km; # of AMR levels: 5; # of boxes of each level: 24, 27, 52, 136, 326; # of data points each level: 27744, 15868, 32032, 64256, 128440; Timestep size: 39 MB.
MISMIP3D	Marine ice sheet model inter-comparison project for plan view models (3D), incorporating two horizontal dimensions. Problem domain: 800km×100km; Size of the base grid: 6.25km ×6.25km; # of AMR levels: 9; # of boxes of each level: 4, 2, 4, 8, 13, 41, 70, 143, 311; # of data points each level: 2448, 2312, 4080, 5984, 10436, 19716, 38136, 72940, 144220; Timestep size: 35 MB.
Greenland	The simulation of the melting glacier in Greenland. Domain size: 1440km×2800km; Size of the base grid: 20km× 20km; # of AMR levels: 6; # of boxes each level: 15, 95, 199, 740, 1953, 5281; # of data points each level: 11700, 41628, 128140, 398096, 1127972, 2935300; Timestep size: 638 MB.

using Huffman coding, motivated by the observation that a number of data points can be quantized into the same level. For curve-missed data points, binary representation analysis will be performed to reduce the storage footprint.

Overall, ZFP and SZ are very effective in taking advantage of the information redundancy in scientific data. However, they both are designed to be generic, without leveraging the application characteristics. Particularly, in the context of AMR, they are oblivious to the hierarchical layout and the redundancy among AMR levels.

III. Motivation

In this section, we aim to understand the similarity among AMR levels and provide grounds for reordering AMR data to improve the compression ratios. To this end, we test a number of AMR applications (Table I). Each application calculates a set of six physical quantities in the simulation, labeled as P1-P6 for *PineIslandGlacier*, M1-M6 for *MISMIP3D*, and G1-G6 for *Greenland*, respectively. For example, the six physical quantities of *PineIslandGlacier* are *thickness*, *xVel*, *yVel*, *topography*, *dThickness/dt*, and *fVel*, respectively.

Our method is inspired by the observation that general-purpose lossy compressors exploit the presumptive local smoothness in data through curve-fitting or discarding high frequency components. However, the information redundancy in data may exist well beyond the local smoothness that only relates to the neighboring data points. Fig. 2 shows the data features of *PineIslandGlacier*, *MISMIP3D*, and *Greenland*. In particular, Fig. 2(a), (b), (d), (e), (g) and (h) show the values of data points at levels 0 and 1, respectively. To quantify the smoothness of data points at level i, we calculate *mean of absolute change* (MAC), which is defined as

$$MAC = \frac{\sum_{j=1}^{N[i]-1} |V[i][j] - V[i][j-1]|^2}{N[i]}$$

where $V[i][j]$ denotes the value of data point j, $0 \leq j < N[i]$, of level i, and $N[i]$ denotes the total number of data points of level i, where $0 \leq i < L$ and L is the number of levels. Intuitively, the smaller the MAC is, the smoother the data is. Fig 2(c), (f) and (i) calculate the difference between levels

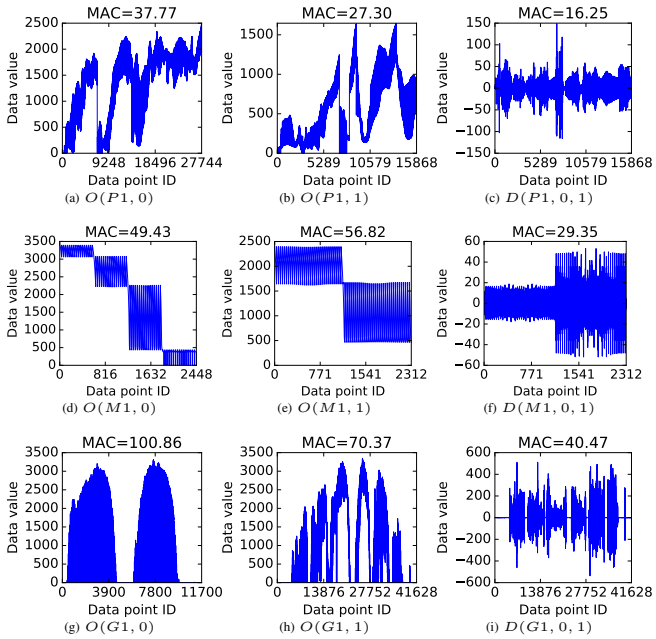


Fig. 2: AMR data feature. Note $O(\cdot)$ is an operator that retrieves a particular level of a quantity. For example, $O(P1, 0)$ denotes the level 0 of P1. Similarly, $D(\cdot)$ is an operator that calculates the difference between two levels of a quantity. For example, $D(P1, 0, 1)$ denotes the delta between levels 0 and 1 of P1.

0 and 1 of the three datasets. Note that since the number of data points at levels 0 and 1 are different as a result of refinement, we utilize piecewise constant interpolation to interpolate the correction from level 0 to level 1. For example, the delta between data point 0A in Fig. 1 and its corresponding data points {1A, 1B, 1E, 1F} at level 1 is calculated as $Delta[1][j] = V[1][j] - V[0][0]$, $j = 0, 1, 4, 5$. The results show that the difference between the two levels are much smoother than the original data, as evidenced by the reduced MAC. For example, the MAC of levels 0 and 1 in *PineIslandGlacier* are 37.77 and 27.30, respectively, while that of delta is reduced to 16.25. This motivates us to further expose the similarity among levels to improve the compression performance.

IV. Level Reordering

Based upon the observation that there exists similarity among AMR levels, the idea of zMesh is to restructure AMR data in a way that those data points belong to the same geometric locations in the simulation will be made adjacent prior to lossy compression.

A. Overall Design

For the convenience of discussion, we denote a box in AMR as b, which can be represented by a pair of coordinates, $(b.smallEnd, b.bigEnd)$, where $b.smallEnd$ and $b.bigEnd$ are the two diagonally corners of the rectangular region. The set of boxes constructed by AMR is denoted as $boxes$, where $boxes[i]$ denotes all boxes at level i. Fig. 3(a) illustrates a dataset of three levels, with one, two, and one boxes at each

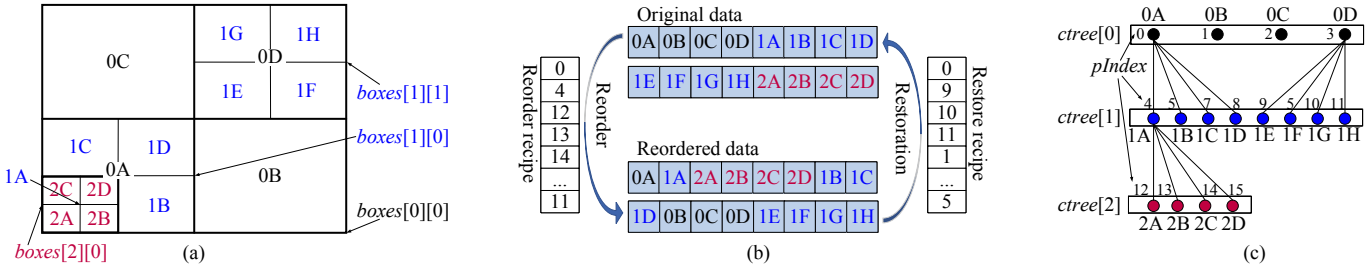


Fig. 3: An example of zMesh. (a) an example of AMR data layout. (b) reordering and restoration. The reorder/restore recipes maintain the indexes of data to allow for reordering/restoration. For example, 0 and 9 in the restore recipe indicate that the first data point is at position 0 in the reordered data, which is 0A, and the second data point is at position 9, which is 0B. The restore recipe will add to the total size of compressed data, thereby reducing the overall compression ratio. (c) the proposed recipe-free compression enabled by computing a chained tree structure on-the-fly. The root node of each tree belongs to level 0 and is maintained by $ctree[0]$. $pIndex$ denotes the original position, which is used later to generate the reorder recipe.

level. Each box has four data points, where each point is denoted by the concatenation of its level ID and sequence index within its level. For example, 0A denotes the first data point at level 0. As shown in Fig. 3(b), without reordering, the original data will be stored level by level as

0A-0B-0C-0D-1A-1B-1C-1D-1E-1F-1G-1H-2A-2B-2C-2D

In contrast, with zMesh, the segment of 1A-1B-1C-1D is moved between 0A and 0B due to the similarity between 0A and segment 1A-1B-1C-1D. Likewise, the segment of 1E-1F-1G-1H is moved after 0D. Therefore, we have

0A-1A-1B-1C-1D-0B-0C-0D-1E-1F-1G-1H-2A-2B-2C-2D

Next, we move the segment of 2A-2B-2C-2D after 1A, again due to the similarity between levels 1 and 2. The reordered data will be stored as

0A-1A-2A-2B-2C-2D-1B-1C-1D-0B-0C-0D-1E-1F-1G-1H

as shown in the lower portion of Fig. 3(b). The reordering process is facilitated by a reorder recipe, which is

0, 4, 12, 13, 14, 15, 5, 6, 7, 1, 2, 3, 8, 9, 10, 11

where each number indicates the original position for each data point. Similarly, to allow data to be reconstructed later, we need to construct a restore recipe as

0, 9, 10, 11, 1, 6, 7, 8, 12, 13, 14, 15, 2, 3, 4, 5

where each number indicates the index of an original data point. For example, the first two elements in the restore recipe, 0 and 9, indicate that after restoration, the 0-th element 0A and the 9-th element 0B will be recovered as the first two data points.

Chained tree structure. A key step associated with realizing level reordering is to build up the geometric mapping between data points in fine and coarse levels. To this end, we implement a chained tree structure *ctree*, as shown in Fig. 3(c), which can be constructed based upon $boxes$. In particular, $ctree[i]$ maintains all data points at level i, and $ctree[i][j]$ represents the j-th data point at level i. Essentially, the idea is that if a data point at level i is maintained as a tree node, those that fall within its geometric boundary at level $i+1$ will be maintained as its direct child nodes. For example, the first box at level 2, $boxes[2][0]$, which contains {2A, 2B, 2C, 2D} is geometrically mapped to the first box at level 1, $boxes[1][0]$.

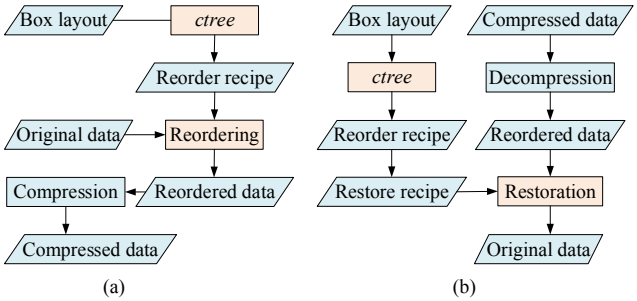


Fig. 4: Workflow of zMesh. (a) compression. (b) decompression.

Since the coordinate of 2A is within the boundary of 1A, we set 1A in $boxes[1][0]$ as the parent and 2A as the child. Then, other data points within $boxes[2][0]$, i.e., 2B, 2C and 2D, will be maintained as siblings to 2A, sharing 1A as the common parent. More details on using $ctree$ for reordering are presented in Section IV-B.

Recipe-free restoration. A challenge we aim to address in this work is how to reduce the storage overhead associated with maintaining the restore recipe, which has been a key problem for reordering based methods. As shown in prior work, such as ISABELA [2] and Migratory Compression [24], the compression ratios can be hurt by the recipe overhead, which adds to the size of compressed data. Therefore, one contribution we make in zMesh is that, instead of storing the recipe alongside the compressed data, we compute $ctree$ on-the-fly during decompression using $boxes$, based upon which we then construct the recipe. We note that the storage overhead of $boxes$ is negligible - only a pair of coordinates per box is stored, and the box layout is typically maintained by AMR applications for its numerical calculations. Therefore, our method essentially does not involve storage overhead for maintaining the restore recipe in the compressed data. The reorder recipe is constructed by traversing $ctree$ with depth first search (DFS), and the restore recipe is the inverse of the reorder recipe [24].

Fig. 4 shows the key steps in zMesh for compression and decompression. During the compression, $ctree$ is generated according to the box layout based upon the geometric mapping

between adjacent levels, and the reorder recipe is constructed by traversing $ctree$. With the reorder recipe, the original data are reordered and then further compressed by lossy compressors. Similarly, during decompression, the restore recipe is generated by $ctree$ based upon the box layout. Then, the compressed data are inflated and reconstructed using the restore recipe. Note that for decompression, the process of constructing the restore recipe can be done in parallel with the inflation process.

B. Building Reorder and Restore Recipe

Algorithm 1 Building reorder and restore recipe.

Require: A set of boxes $boxes$, the number of AMR levels L
Ensure: Generate the reorder and restore recipe
1: **for** b in $boxes[i]$ of level i, $i = 0, 1, \ldots, L-1$ **do**
2: $j \leftarrow 0$
3: **for** each coordinate (x, y) within b **do**
4: Allocate $tree[i][j]$ with structure members of (x, y), $pIndex$, $firstChild$ and $nextSibling$
5: $j \leftarrow j + 1$
6: **end for**
7: **end for**
8: Establish the geometric mapping $P_{map}[i][j]$
9: **for** i from $L-1$ to 1 **do**
10: **for** $tree[i][j]$ in $ctree[i]$ **do**
11: $parent \leftarrow ctree[i-1][P_{map}[i][j]]$
12: **if** $parent$ has no first child **then**
13: Set $tree[i][j]$ as the first child of $parent$
14: **else**
15: Append $ctree[i][j]$ to the sibling list of the first child
16: **end if**
17: **end for**
18: **end for**
19: Traverse the trees with DFS and output $pIndex$ as reorder recipe
20: Restore recipe is the inverse of reorder recipe

Algorithm 1 shows how to build the reorder and restore recipe of zMesh by maintaining $ctree$. The algorithm is divided into three main steps. **Step 1.** Initialize $tree$. We initialize $ctree$ level by level, and within each level box by box. For a box $b \in boxes[i]$, we allocate a tree node $ctree[i][j]$ for each data point that falls between $b.smallEnd$ and $b.bigEnd$, where i is the level ID and j is the data point ID within the level. The tree node $ctree[i][j]$ maintains the associated data point's coordinate (x, y), the original position index $pIndex$ (shown in Fig. 3) in the level data, and pointers to the first child $firstChild$ and the next sibling $nextSibling$ (Line 1-7).

Step 2. Establish the geometric mapping. Essentially, we need to identify the parent node of $ctree[i][j]$ that will be later reordered and compressed together (Line 8). We denote the data point ID of the parent node as $P_{map}[i][j]$, and therefore the parent node is $ctree[i-1][P_{map}[i][j]]$. Then, we compare the coordinate of $ctree[i][j]$ with those of nodes in $ctree[i-1]$ to identify a match. In particular, if the distance between two neighboring data points at level i, e.g., between $ctree[i][j+1]$ and $ctree[i][j]$, is within the refinement ratio, which is a parameter of AMR applications to define the ratio of grid sizes between two adjacent levels, we directly set the parent of $ctree[i][j+1]$ to $ctree[i-1][P_{map}[i][j]]$, without looping through all data points in $ctree[i-1]$. We take $ctree[1]$ in Fig. 3(c) as an example. For the first box $boxes[1][0]$, we start with its first data point, $smallEnd$, that is 1A. By comparing 1A with the level 0 boxes, in this case only $boxes[0][0]$, we have the parent of 1A as 0A. For the other three nodes 1B, 1C and 1D within $boxes[1][0]$, since their distances to 1A are within the refinement ratio of 2×2, they share the same parent with 1A. We next perform a similar procedure for $boxes[1][1]$, and obtain $P_{map}[1] = \{0, 0, 0, 0, 3, 3, 3, 3\}$. If $ctree[i-1][P_{map}[i][j]]$ does not have a child, $ctree[i][j]$ is then set as its first child. Otherwise, $ctree[i][j]$ is appended to the sibling list $firstChild.nextSibling$ (Line 9-18).

Step 3. Build the reorder and restore recipe. To build the reorder recipe, we traverse $ctree$ and output $pIndex$ (Line 19). From each tree within $ctree$, we traverse the nodes from its root using DFS. Take Fig. 3(c) as an example, the $pIndex$ list of $ctree$ is $\{0, 1, 2, \cdots, 15\}$, and by traversing $ctree$ with DFS, the reorder recipe is {0, 4, 12, 13, 14, 15, 5, 6, 7, 1, 2, 3, 8, 9, 10, 11}. Then, we can build the restore recipe as {0, 9, 10, 11, 1, 6, 7, 8, 12, 13, 14, 15, 2, 3, 4, 5}, which is the inverse of the reorder recipe and regenerated by $restore\ recipe[reorder\ recipe[j]] = j$ (Line 20). The memory overhead of zMesh stems from maintaining $ctree$, P_{map}, and reorder/restoration recipe. The sizes of a $ctree$ node, a P_{map} element and a recipe element are 5, 1 and 1 integers, respectively.

C. zMesh with Space-filling Curve

In this section, we further develop zMesh in the context of space-filling curve - a well known technique to preserve data locality [20] for a multi-dimensional dataset. While both zMesh and space-filling curve aim to improve the locality, zMesh further leverages the redundancy across levels and therefore its merits are orthogonal to space-filling curve. In particular, space-filling curve improves the data locality from the aspect of intra-level, while zMesh is from inter-level. The key challenge is that when space-filling curve is applied, $P_{map}[i][j]$ will accordingly change.

We present the idea of combining zMesh with space-filling curve through an example of 2-level AMR layout, where there is only one box at each level. Both the sizes of $boxes[0][0]$ and $boxes[1][0]$ are 4×2, with data points 0A-0B-0C-0D-0E-0F-0G-0H at level 0, and 1A-1B-1C-1D-1E-1F-1G-1H at level 1. Data points 1A, 1B, 1E and 1F are mapped to 0C, and data points 1C,1D,1G and 1H are mapped to 0D. Thus, we have $P_{map}[1] = \{2, 2, 3, 3, 2, 2, 3, 3\}$. The following are the steps of zMesh under Z-ordering. **Step 1**: Do the space-filling curve and the resulting $ctree[0]$ is 0A-0B-0E-0F-0C-0D-0G-0H. **Step 2**: We denote the inverse transformation of space-filling curve as R_{De}, which allows $ctree[0]$ to be reconstructed in original order. Based on the reordering of $ctree[0]$, we have $R_{De} = \{0, 1, 4, 5, 2, 3, 6, 7\}$. **Step 3**: Calculate the new parent index of $ctree[1][j]$ as $P'_{map}[1][j] = R_{De}[P_{map}[1][j]]$. For example, the new parent index of 1A is $P'_{map}[1][0] = R_{De}[P_{map}[1][0]] = R_{De}[2] = 4$. Similarly, for 1C, we

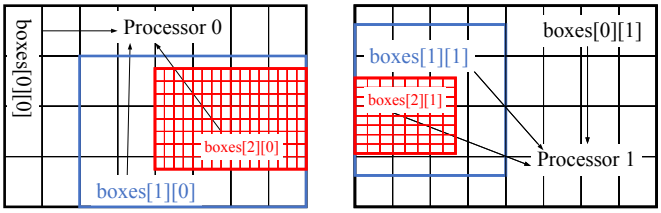


Fig. 5: A schematic of zMesh parallel compression.

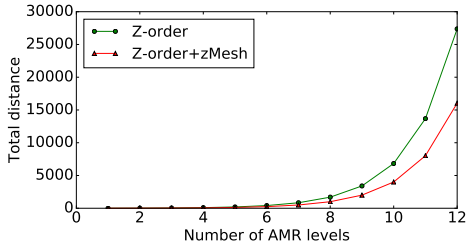


Fig. 6: Total distance of standard Z-ordering (Z-order) and Z-ordering in conjunction with zMesh (Z-order+zMesh).

have $P'_{map}[1][2] = R_{De}[P_{map}[1][2]] = 5$. Finally, we have $P'_{map}[1] = \{4, 4, 5, 5, 4, 4, 5, 5\}$.

D. Parallel zMesh Compression

zMesh is embarrassingly parallel in nature and each processor compresses data locally without collective communication, solely based upon its local box layout constructed by AMR. As shown in Fig. 5, processor 0 computes three boxes, $boxes[0][0]$, $boxes[1][0]$, $boxes[2][0]$, while processor 1 computes $boxes[0][1]$, $boxes[1][1]$, $boxes[2][1]$. The zMesh compression of data at processor 0 is done solely based upon the layout of the its resident boxes, completely independent of the compression at processor 1, without any communications. We carefully note that the numerical calculation in AMR itself typically involves communications for grid synchronization, but the zMesh compression does not incur additional communications.

E. Data Regularization of zMesh

In this section, we show that zMesh in conjunction with space-filling curve can further improve the data locality with a typical AMR example (complete refinement). We focus on the total distance, which is a common metric for data locality [25]. Consider a dataset of levels $0, 1, \ldots, L-1$ with $N[i]$ data points at level i, let $N = \sum_{i=0}^{L-1} N[i]$ be the total number of data points. The refinement ratio is 2^r, where r is a positive integer. We introduce two ways of sequentializing these N points. Let $A = \{a_j\}_{j=0}^{N-1}$ and $B = \{b_j\}_{j=0}^{N-1}$ be the sequences of points generated by standard Z-ordering (level by level) and Z-ordering in conjunction with zMesh, respectively, where a_j and b_j are points on the plane where data points in the AMR structure are taken. Let $\mathcal{E} = \{E_j = (a_j, a_{j+1}), j = 0, 1, \ldots, N-2\}$ and $\mathcal{F} = \{F_j = (b_j, b_{j+1}), j = 0, 1, \ldots, N-2\}$ be sets of edges.

Fig. 6 shows results of total distance across different number of AMR levels. Due to the space limit, we only take Z-ordering as an example. In this example, all the cells in a coarser level are refined, and r is set as 1. There are $2^{2(i+1)r}$ cells at level i, and the side length of a cell at level i is 2^{-ir}. The results shows that the total distance of zMesh is obviously smaller than that of Z-ordering.

Recall that in AMR algorithms, the data are assumed to be second differentiable over the underlying space. Thus, the gradient exists and is locally bounded. Since the edge-length distribution is regularized (length locally shortened) in zMesh, the data difference distribution (which locally can be estimated as the product of local gradient and edge length) is also tamed. This helps reduce the total variation of the sequence of data, as observed in empirical results. Moreover, detailed analysis shows that since zMesh makes a number of long distance adjacent pair of data points to closer pairs, the sequence is more likely to behave better under lossy compression (better for curve-fitting or data transformation) while at the same time fewer adjacent pairs are made slightly farther.

V. Evaluation

A. Experimental Setup

We evaluate the effectiveness of zMesh through experiments conducted on a Linux server with Ubuntu 16.04.5 LTS. The processor is Intel CoreTM i5-7500 that has 4 cores with frequency of 3.4 GHz and 16 GB of memory. The parallel experiments were done on Summit at Oak Ridge National Laboratory. We use ZFP (0.5.5) and SZ (2.1.8) as the backend lossy compressors. In particular, we use the absolute error bound for both compressors. To test a range of error bounds that covers the entire value range of a dataset, we use the product of the relative error bound and the range of data to determine the absolute error bound for each dataset. For SZ, the number of quantization intervals is set to 0, which allows SZ to search for an optimized setting. In our experiments, the default relative error bound is 1E-3, unless otherwise specified. We evaluate four setups across this paper, including the standard Z-ordering (*Z-order*), Z-ordering in conjunction with zMesh (*Z-order+zMesh*), Hilbert curve (*Hilbert*), and Hilbert curve in conjunction with zMesh (*Hilbert+zMesh*).

B. Compression Performance

1) Compression Ratio: As the very first step, we examine whether the local smoothness is improved through zMesh as compared to Z-order and Hilbert. Fig. 7 calculates the MAC across all 18 datasets as described in Table I. On average, zMesh reduces the MAC of Z-order and Hilbert by 67.9% and 71.3%, respectively. This suggests that after reordering, data are significantly smoother. Fig. 8 further measures the resulting compression ratios using the four setups. Overall zMesh yields higher compression ratios for both ZFP and SZ, as compared to the standard space-filling curves. The compression ratios are improved by 4.4% to 16.4% for Z-order with ZFP, 5.7% to 16.5% for Hilbert with ZFP, 25.2% to 113.3% for Z-order with SZ, and 28.4% to 133.7% for Hilbert with SZ, respectively.

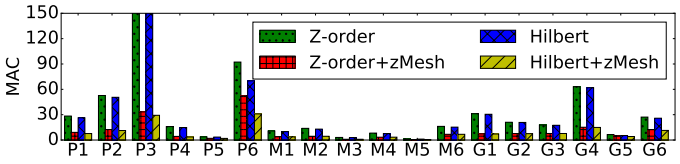
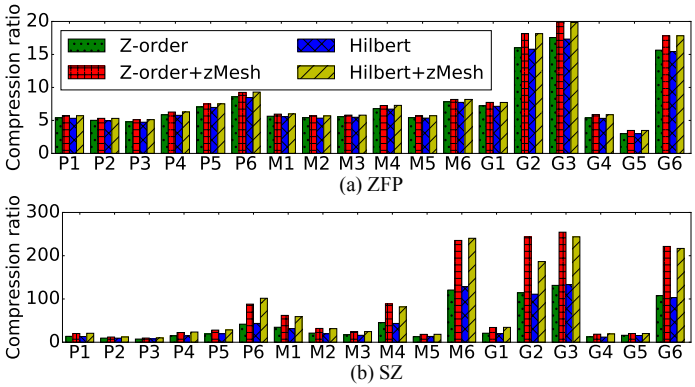


Fig. 7: Local smoothness measured in MAC. The x-axis is the quantity to be compressed and the y-axis is MAC.



(a) ZFP

(b) SZ

Fig. 8: Compression ratio.

2) Internal Compression Metrics: To further understand how the compressors react to the improved smoothness and in turn increase the compression ratio, we collect the internal compression metrics of ZFP and SZ, as described in Table II. For ZFP, the size of compressed data is the sum of sizes of all compressed blocks, given that ZFP partitions the entire dataset into fixed-sized blocks. The important metrics are $ZeroCnt$, $MaxExp$, $MaxPrec$, $BitsBitplane$, and $BlockSize$ (detailed in Table II). For the case that a block contains all zeros, $BlockSize$ is one since only a single bit of zero is written. Otherwise, $BlockSize$ is the sum of $BitsBitplane$ and $BitsExp$, where $BitsExp$ is a fixed number of bits representing the exponents (11 for double-precision floating-point data). For a non-zero block, $BitsBitplane = \sum_{j=0}^{MaxPrec-1} BitsPerBitplane_j$, where $BitsPerBitplane_j$ is the j-th bit plane to encode for the block, and $MaxPrec$ is calculated as $MaxPrec = min(64, MaxExp - log_2(Errorbound) + 2 * (dim + 1))$, where dim is the number of dimensions. For SZ, the size of compressed data is the sum of the sizes of curve-hit points and curve-missed points. The former further consists of Huffman tree and Huffman encoding for quantized data. Therefore, the important metrics are $HitRatio$, $QuantIntv$, $TreeSize$, $EncodeSize$, and $OutlierSize$ (detailed in Table II). Further details regarding the compression algorithms can be found in prior work [3], [4].

Fig. 9 measures the internal metrics of ZFP. Due to the space limit, we only show the results of P1, M1 and G1. As compared to Z-order, zMesh increases $ZeroCnt$ from 1,050 to 1,282 for P1, and from 67,372 to 84,403 for G1, respectively. Despite the significant improvement in $ZeroCnt$, the percentage of all-zero blocks is low across all datasets, and therefore the contribution to the overall improvement of compression ratio is negligible. Meanwhile, $MaxExp$ and $MaxPrec$ remain almost identical since the average of the maximum exponent will not change substantially after reordering. The average $BitsBitplane$ of zMesh has a substantial reduction of 2.73, 6.27 and 2.63, for P1, M1 and G1, respectively. The reason is that zMesh improves the data smoothness, and more bits in a bit plane after the orthogonal transform will likely be zero, resulting in the reduced $BitsBitplane$. Similar results also observed for Hilbert with zMesh. By and large, zMesh yields less improvement for ZFP versus SZ (shown next in Fig. 10). The reason is that ZFP is intended to a high-throughput compressor and is conservative in exploiting the improved smoothness and losing accuracy for higher compression ratios.

TABLE II: A list of internal compression metrics.

Symbols	Description
	ZFP
BlockCnt	Number of blocks
ZeroCnt	Number of blocks that are with all zeros
MaxExp	The common (largest) exponent of each block
MaxPrec	Maximum number of bit planes to encode in order to meet the accuracy demand
BitsBitplane	Number of bits used in encoding bitplane
BlockSize	Size of each block data (in bits)
	SZ
HitRatio	Curve-fitting hit ratio
QuantIntv	Number of quantization intervals
TreeSize	Size of Huffman tree (in bytes)
EncodeSize	Total size of Huffman coding (in bytes)
OutlierSize	Total size of curve-missed points (in bytes)

Fig. 10 measures the internal metrics of SZ. As compared to both Z-order and Hilbert, $HitRatio$ is improved for all three applications using zMesh as a result of the improved smoothness. For example, for Z-order, $HitRatio$ is improved from 0.9667 to 0.9878, from 0.9857 to 0.9968, and from 0.9645 to 0.9952 for P1, M1 and G1, respectively. This in turn lowers $OutlierSize$, which is the size of curve-missed points that are typically hard to compress, for zMesh. For example, $OutlierSize$ of M1 is reduced from 8 940 by more than 50%. For some cases, SZ may apply smaller quantization intervals under zMesh as a result of improved smoothness, which further reduces the tree size and encode size. With the optimized mode of SZ, the number of tree nodes in the Huffman tree is $2 \cdot QuantIntv - 1$, and therefore we observe similar outcome of $TreeSize$. More importantly, with the improved data smoothness, $EncodeSize$ is substantially smaller as compared to Z-order and Hilbert. Overall, zMesh improves the compression ratios as a result of the reduction of $EncodeSize$ and $OutlierSize$.

We further test the effectiveness of zMesh across error bounds from 1E-5 to 1E-2 in Fig. 11. It is found that as the error bound relaxes, the improvement of compression ratio by zMesh becomes more pronounced. For example, for Hilbert, as the error bound increases from 1E-5 to 1E-2, the compression ratio improvement on P1 using SZ are 21.5%, 23.1%, 56.9%, and 109.3%, respectively.

3) zMesh vs. Improved Locality from Compressors: There has also been similar effort in the compressors to improve

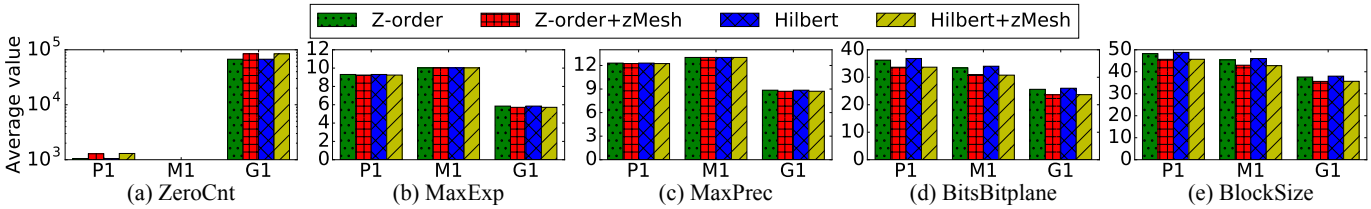


Fig. 9: Internal compression metrics (ZFP).

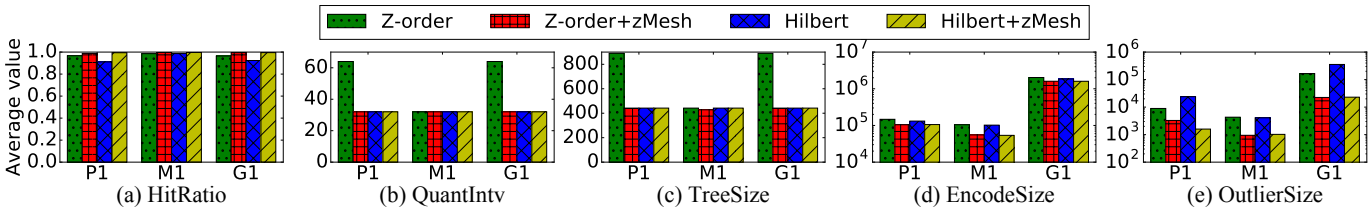


Fig. 10: Internal compression metrics (SZ).

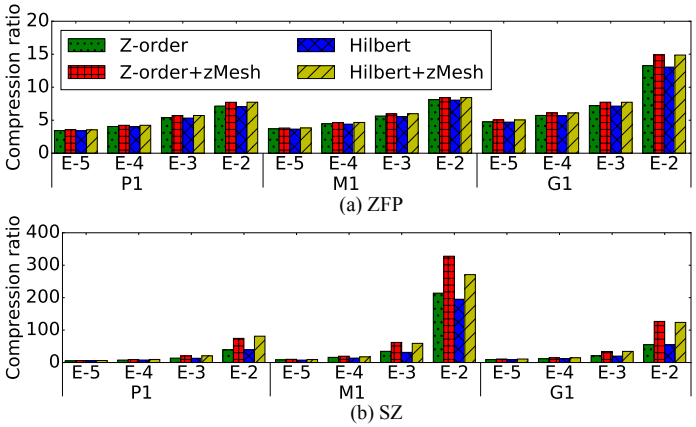


Fig. 11: Comparison of compression ratios across different error bounds.

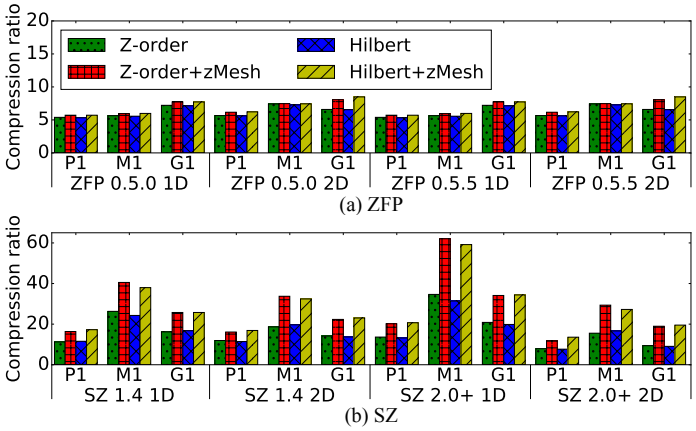


Fig. 12: zMesh under various compressor internal locality. For all 2D configurations, the size of the first dimension is set to 1024.

the data locality, e.g., using Lorenzo predictor to improve the multi-dimensional curve-fitting. We next evaluate zMesh under various compressor internal locality, aiming to understand whether zMesh is orthogonal to these optimizations or duplicates these efforts. For ZFP, we test two major versions 0.5.0 and the latest 0.5.5 for both 1D and 2D. We anticipate 2D will have higher locality due to the increased block size, and more importantly, the better row-column locality. For SZ, we test 1.4 and the latest 2.0+, with the latter using the Lorenzo predictor [23]. Fig. 12 shows the compression ratios with different configurations of ZFP and SZ. The results show that zMesh is completely complementary to other locality improvement in compressors - even with the 2D setup, the compression ratios of zMesh are higher than those of standard space-filling curves.

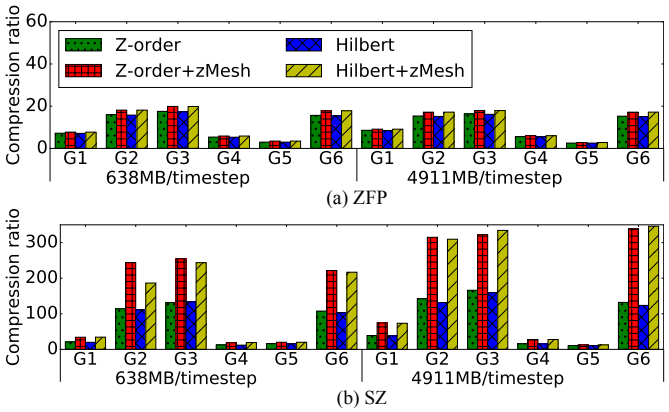


Fig. 13: Compression ratio of zMesh on different size configurations of *Greenland*.

4) Impacts of Dataset Size and Parallelism: We further evaluate the results on larger datasets up to 4911 MB per timestep for *Greenland*, and Fig. 13 shows the compression ratio of *Greenland*. For the dataset configuration, each timestep output has a size of 638MB, while the large dataset has a size of 4911MB per timestep. the finding is that zMesh yields higher compression ratios for SZ due to the higher chance of information redundancy. Meanwhile, for ZFP, since it is block-based compression and does not fully take advantage of the global data, the gain is insignificant.

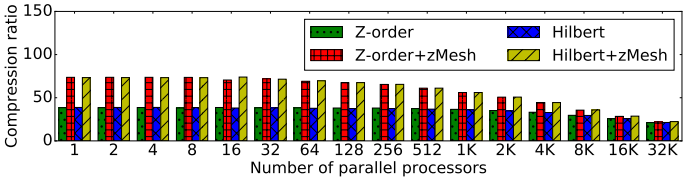


Fig. 14: Parallel compression using 1 to 32K processors (*Greenland* with SZ).

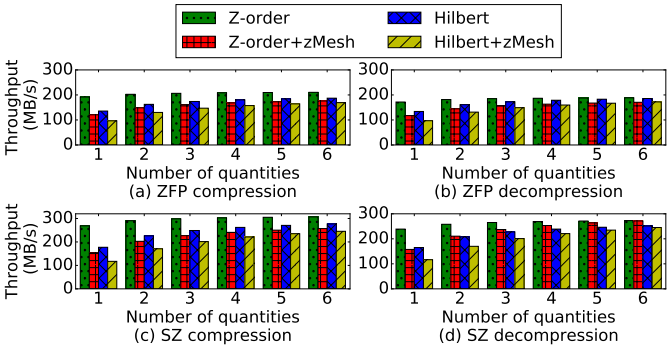


Fig. 15: Compression and decompression throughput (*PineIslandGlacier*).

As aforementioned, zMesh is embarrassingly parallel, and therefore paralleling zMesh is trivial. Fig. 14 shows the compression results using 1 to 32K processors. In the experiment, each processor compresses only its local data by maintain its own $ctree$. Note that some boxes in the finer levels may have no parent boxes. In this case, the nodes in these boxes are regarded as root nodes. The results show that, using small numbers of processors (smaller than 512), zMesh has almost the same compression ratio improvement compared to the serial compression (1 processor). However, for huge numbers of processors, such as 16K and 32K processors, the improvement of zMesh is not as obvious. The reason is as aforementioned, zMesh has a limited space for compression optimization. In the huge number cases, there is only one box on each processor, zMesh is the same as standard space-filling curves.

5) Throughput: We further evaluate how much the overhead of zMesh would impact the overall compression and decompression throughput. As shown in Fig 15, zMesh has a non-trivial reduction of throughput as a result of building $ctree$ and reorder/restore recipe, if only one quantity is processed. However, as the number of quantities increases, the overhead is clearly amortized, since all quantities in an application would have identical $ctree$, and therefore $ctree$ and reorder/restore recipe only need to be built once. When processing 6 quantities, the average throughput degradation is 13.0%, 8.4%, 13.8% and 1.5% for ZFP compression, ZFP decompression, SZ compression and SZ decompression, respectively.

C. Subsetting Compressed Data

Herein we discuss the performance of extracting a subset of the compressed datasets using zMesh - a scenario that is particularly important to data analytics with the goal of avoiding inflating the entire data as much as possible. In Table III, we summarize the key access patterns of AMR data analysis, and use *fetching efficiency*, defined as the ratio of the size of a query to the size of fetched data, to characterize the efficiency of accessing the compressed data [26]. Ideally, a compressor with a high fetching efficiency should allow for high random access and retrieve only the target region. However, in reality, compressors often have limited support towards random access. For example, ZFP compresses data block by block, and therefore a block needs to be fully inflated in order to access the target region. Meanwhile, the curve-fitting in SZ is built involving the neighboring points, and retrieving a small subset necessitates decompression all data first. Therefore, in what follows, we only focus on ZFP.

TABLE III: AMR access pattern.

Pattern	Geometric Region	Level
Pattern 1	Retrieve all data points that are geometrically within a given box	All levels
Pattern 2	Retrieve one row that is geometrically within the coarsest box	All levels
Pattern 3	Retrieve all data points that are geometrically within a given box	Finest level
Pattern 4	Retrieve one row that is geometrically within a given box	Finest level

Fig. 16 shows the fetching efficiency under all access patterns for P1. In particular, Fig. 16(a)-(d) show the fetching efficiency of cross-level queries on the compressed data, while Fig. 16(e)-(h) show the fetching efficiency of queries issued to the finest level of the compressed data. For Pattern 1, we retrieve the geometric region covered by a box and the region may relate to one or more levels. Herein, we examine all 565 boxes in P1 with one query issued per box. In contrast in Pattern 2, we retrieve each of the 192 rows of the coarsest box. Note that for Pattern 2, there is no difference between Z-order (or Hilbert) and Z-order (or Hilbert)+zMesh for some rows. The reason is that if there is only one AMR level involved, zMesh is the same as the standard space-filling curves. In Pattern 3, we query all 326 boxes at the finest level in P1, and in Pattern 4, we uniformly query 10% of the rows within all 326 boxes. It is found that zMesh has a higher fetching efficiency than the standard space-filling curves for Pattern 1 and 2. However, it yields a lower fetching efficiency for Pattern 3 and 4, and the reason is that zMesh has to fetch more ZFP blocks from the compressed data, since the data points of other AMR levels are included in the fetched blocks as a result of reordering.

VI. Conclusion

This paper focuses on further improving the compression ratios by exploiting the information redundancy in AMR. We illustrate in this paper that there exists high similarities among AMR levels, and the native organization of AMR data does not easily expose this redundancy to compressors, thus leading to low compression ratios. The idea of the proposed zMesh is to re-arrange data so that data points corresponding to the same (or adjacent) geometric coordinates can be compressed together, thus greatly improving the local smoothness of the

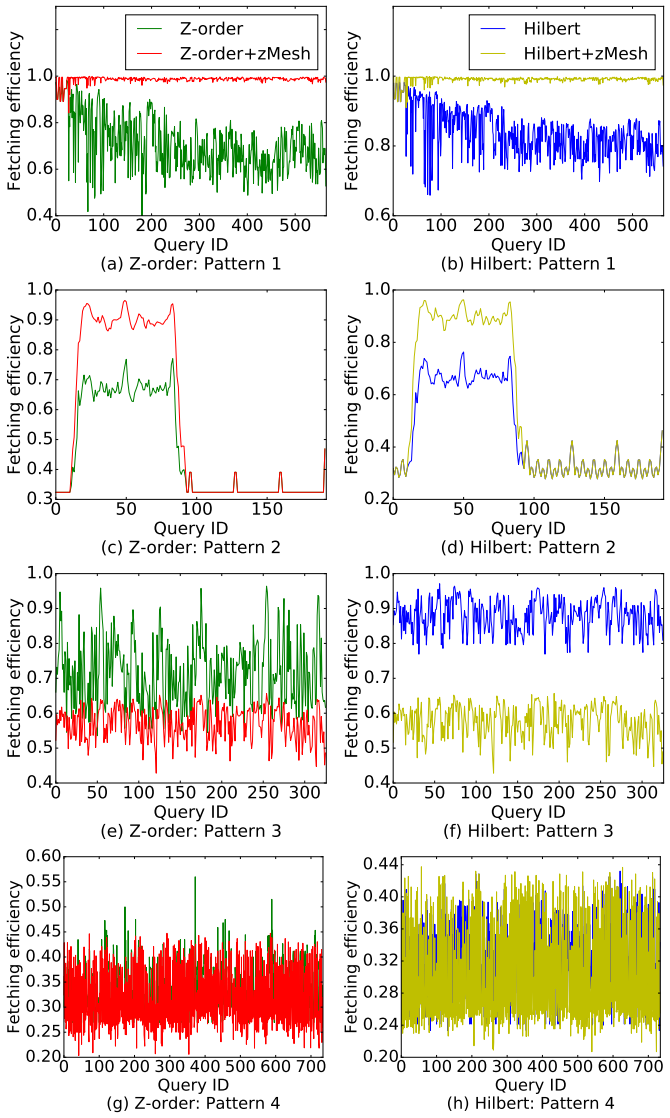


Fig. 16: Fetching efficiency of the compressed data. We tested zMesh over P1 compressed by ZFP.

dataset. A key contribution of this work is that, unlike any prior work that maintain the restore recipe as part of the compressed data, zMesh can compute the restore recipe on-the-fly using a chained tree structure. The results demonstrate that zMesh can reduce the mean of absolute change of data by 67.9% and 71.3% for Z-ordering and Hilbert, respectively. Overall zMesh improves the compression ratios by up to 16.5% and 133.7% for ZFP and SZ, respectively. Despite that zMesh involves additional compute overhead for tree and restore recipe construction, we show that the cost can be amortized as the number of quantities to be compressed increases.

Acknowledgments

The authors wish to acknowledge the support from the US NSF under Grant No. CCF-1718297, CCF-1812861, and DOE CODAR project. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

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

[2] S. Lakshminarasimhan *et al.*, "Compressing the incompressible with ISABELA: In-situ reduction of spatio-temporal data," in *EURO-PAR '11*. Springer, 2011, pp. 366–379.

[3] P. Lindstrom, "Fixed-rate compressed floating-point arrays," *T-VCG*, vol. 20, no. 12, pp. 2674–2683, 2014.

[4] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *IPDPS '16*, 2016, pp. 730–739.

[5] J. C. Bennett *et al.*, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *SC '12*, 2012, p. 49.

[6] J. Ahrens *et al.*, "An image-based approach to extreme scale in situ visualization and analysis," in *SC '14*. IEEE Press, 2014, pp. 424–434.

[7] U. Ayachit *et al.*, "Performance analysis, design considerations, and applications of extreme-scale in situ infrastructures," in *SC '16*. IEEE Press, 2016, p. 79.

[8] T. Wang, S. Oral, Y. Wang, B. Settlemyer, S. Atchley, and W. Yu, "Burstmem: A high-performance burst buffer system for scientific applications," in *BigData '14*, 2014, pp. 71–79.

[9] S. Herbein *et al.*, "Scalable i/o-aware job scheduling for burst buffer enabled hpc clusters," in *IPDPS '16*. ACM, 2016, pp. 69–80.

[10] N. Liu *et al.*, "On the role of burst buffers in leadership-class storage systems," in *MSST 12*, 2012, pp. 1–11.

[11] T. Lu *et al.*, "Canopus: Enabling extreme-scale data analytics on big hpc storage via progressive refactoring," in *HotStorage '17*, Santa Clara, CA, 2017.

[12] M. Burtscher and P. Ratanaworabhan, "Fpc: A high-speed compressor for double-precision floating-point data," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 18–31, 2008.

[13] P. Lindstrom and M. Isenburg, "Fast and efficient compression of floating-point data," *T-VCG*, vol. 12, no. 5, pp. 1245–1250, 2006.

[14] J. Tian, S. Di, C. Zhang, X. Liang, S. Jin, D. Cheng, D. Tao, and F. Cappello, "wavesz: a hardware-algorithm co-design of efficient lossy compression for scientific data," in *PPoPP '20*, 2020, pp. 74–88.

[15] A. M. Gok *et al.*, "Pastri: Error-bounded lossy compression for two-electron integrals in quantum chemistry," in *CLUSTER '18*, 2018.

[16] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[17] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler *et al.*, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, 2014.

[18] I. Foster *et al.*, "Computing just what you need: online data analysis and reduction at extreme scales," in *EURO-PAR '17*, 2017, pp. 3–19.

[19] P. Colella *et al.*, "Chombo software package for amr applications design document," *Available at the Chombo website: http://seesar. lbl. gov/ANAG/chombo/*, 2009.

[20] S. Kumar *et al.*, "Efficient i/o and storage of adaptive-resolution data," in *SC '14*. IEEE Press, 2014, pp. 413–423.

[21] A. Lintermann *et al.*, "Massively parallel grid generation on hpc systems," *Computer Methods in Applied Mechanics and Engineering*, vol. 277, pp. 131–153, 2014.

[22] E. H. Moore, "On certain crinkly curves," *Transactions of the American Mathematical Society*, vol. 1, no. 1, pp. 72–90, 1900.

[23] D. Tao *et al.*, "Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization," in *IPDPS '17*. IEEE, 2017, pp. 1129–1139.

[24] X. Lin *et al.*, "Migratory compression: Coarse-grained data reordering to improve compressibility," in *FAST '14*, 2014, pp. 256–273.

[25] C. Pyo, K.-W. Lee, H.-K. Han, and G. Lee, "Reference distance as a metric for data locality," in *HPC Asia'97*. IEEE, 1997, pp. 151–156.

[26] S. Kreft and G. Navarro, "Lz77-like compression with fast random access," in *DCC '10*. IEEE, 2010, pp. 239–248.