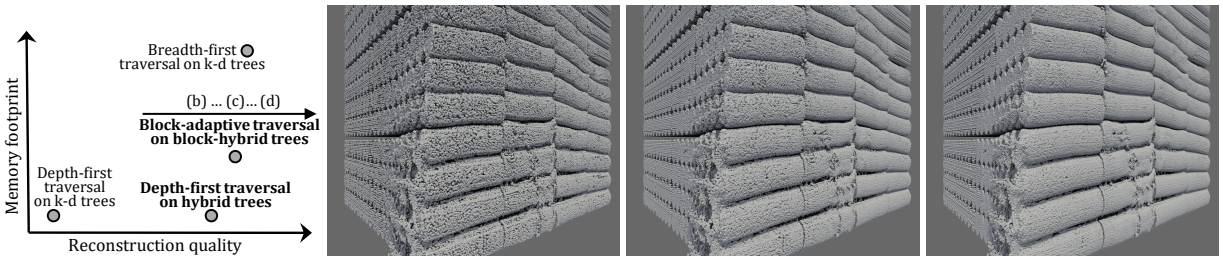
High-Quality and Low-Memory-Footprint Progressive Decoding of Large-Scale Particle Data

Duong Hoang* Harsh Bhatia† Peter Lindstrom† Valerio Pascucci*
SCI Institute, University of Utah CASC, Lawrence Livermore National Laboratory SCI Institute, University of Utah



(a) Memory-quality space (b) 138×, n = 33M at ≈ 50 MB (c) 46×, n = 106M at ≈ 50 MB (d) 23×, n = 215M at ≈ 50 MB

Figure 1: (a) Our particle compression approaches based on the proposed *hybrid trees* and *block-hybrid trees* achieve better memory-quality trade-offs for lossy reconstruction compared to traditional approaches based on k-d trees. (b, c, d) Three approximations of a *detonation* simulation dataset, compressed and then decoded with our *block-hybrid tree* approach (compression ratio $k\times$ and corresponding number of particles, n, given). All three are snapshots of a single progressive decompression process, and all use a small and constant amount of memory for decoding (only about 50 MB, measured using the maximum number of elements in the data structures used for tree traversal with 64 bits per element). Rendering is done with OSPRay [74] after a subset of particles of the original 968M particles is decoded in each case. The original dataset could not be rendered on our machine with 64 GB RAM (previously rendered using 3 TB RAM [75]).

Abstract

Particle representations are used often in large-scale simulations and observations, frequently creating datasets containing several millions of particles or more. Due to their sheer size, such datasets are difficult to store, transfer, and analyze efficiently. Data compression is a promising solution; however, effective approaches to compress particle data are lacking and no community-standard and accepted techniques exist. Current techniques are designed either to compress small data very well but require high computational resources when applied to large data, or to work with large data but without a focus on compression, resulting in low reconstruction quality per bit stored. In this paper, we present innovations targeting tree-based particle compression approaches that improve the tradeoff between high quality and low memory-footprint for compression and decompression of large particle datasets. Inspired by the lazy wavelet transform, we introduce a new way of partitioning space, which allows a low-cost depth-first traversal of a particle hierarchy to cover the space broadly. We also devise novel data-adaptive traversal orders that significantly reduce reconstruction error compared to traditional data-agnostic orders such as breadth-first and depth-first traversals. The new partitioning and traversal schemes are used to build novel particle hierarchies that can be traversed with asymptotically constant memory footprint while incurring low reconstruction error. Our solution to encoding and (lossy) decoding of large particle data is a flexible block-based hierarchy that supports progressive, random-access, and error-driven decoding, where error heuristics can be supplied by the user. Finally, through extensive experimentation, we demonstrate the efficacy and the flexibility of the proposed techniques when combined as well as when used independently with existing approaches on a wide range of scientific particle datasets.

Keywords: particle datasets, data compression, coarse approximation, progressive decompression, tree traversal, multiresolution

*{duong, pascucci}@sci.utah.edu
†{hbhatia, pl}@llnl.gov

1 Introduction

As a common form of data representation, particles are used in multiple scientific applications, such as molecular dynamics and atomistic simulations [15,23,31,38,49], fluid dynamics simulations [5,67,79], computational cosmology [19,24,60,66], scanning and imaging of objects and environments [4,36,45], and plasma physics [9]. With rapid advances in computational capabilities, simulations now can generate trillions of particles at scale [9,52,60]. Although such large-scale simulations promise immense value for understanding complex scientific phenomena [14], the reality of relatively poor scaling of network, memory, and disk bandwidths often limits appropriate utilization and analysis of the resulting data, due to its sheer size. Compression is a promising solution to the problem of ever-expanding data. However, most compressors in high-performance computing (HPC) are designed for structured data [42,70], and no widely accepted compressors for large-scale scientific particle data currently exist. Some initial attempts to adapt grid-based compressors for particles [30,69] have seen limited success, partly because particle data tend to be much more random and, thus, difficult to compress. Outside of HPC, techniques designed to compress point clouds representing scans of objects [37,43,47,63] focus largely on improving compression ratios, often at the expense of scalability in performance and memory footprint, making them suitable for only relatively small data. On the other hand, multiresolution rendering systems [19,39,58–60,64] that can scale to very large data typically do not target high-quality compression.

Whereas both approaches have advantages, we note a lack of techniques that balance the two goals — high compression ratios together with fast and low-memory-footprint data reconstruction. Toward bridging this gap, we present novel hierarchy construction and traversal methods. Our techniques improve on the state-of-the-art tree-based compression techniques, and achieve, for the first time, high-quality progressive reconstructions using an asymptotically constant memory footprint. We focus on compressing particle *positions*, since they are needed in almost all applications and, in many applications, are the only attributes needed.

2021 IEEE 11th Symposium on Large Data Analysis and Visualization (LDAV) | 978-1-6654-3283-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/LDAV53230.2021.00011

978-1-6654-3283-2/21/$31.00 ©2021 IEEE
DOI 10.1109/LDAV53230.2021.00011

Scientific particle data are difficult to compress losslessly, since most are sparse relative to the precision of the particles, *i.e.,* many lower-precision bits are essentially random. Nevertheless, valuable trade-offs can be made in the space of lossy compression and progressive, partial decompression, in which the decompression can be paused at any point to obtain an approximation, and resumed to refine such approximations by consuming more bits. This property is useful since instead of waiting for a long decompression process, users can always work with (lossy) reconstructions of data, at improving quality levels that adapt to the available computational resources and time. Here, reconstruction quality depends greatly on the order in which the particle position bits are decoded, which also affects the costs of keeping a state in memory for resuming the decompression. Achieving a balance between decoding costs and reconstruction quality often manifests as a choice between (1) spatially limited but complete representation of particles and (2) quantized but uniform coverage of space — or, in a way, between a *depth-first* (DT) and a *breadth-first* traversal (BT) of a particle hierarchy. We explore this trade-off from the perspectives of both tree traversal and tree construction.

Contributions. Our specific technical contributions are:

- We present a new mechanism to partition space, the *odd-even splits* (Section 3.1), which can be used in conjunction with the standard k-d splits (*i.e.,* splits that create a k-d tree) to selectively convert a DT of a subtree into a BT of the corresponding space.
- We propose a particular way of combining odd-even and k-d splits to create *hybrid trees* (Section 3.2) that allows a low-memory-footprint DT to also have the power of BT (high-quality reconstruction), while being conducive to compression.
- We introduce an *adaptive traversal* (AT) (Section 4.1) that allows better dynamic guiding of tree refinement and with respect to a given error metric; we propose two such metrics by heuristics.
- We combine the strengths of both k-d trees and hybrid trees in *block-hybrid trees* (Section 3.3), which can be traversed with *block-adaptive traversal* (BAT, 4.2), for improved memory-quality trade-off and error-guided, spatially progressive refinement with random access of large, compressed particle data.

2 Background and Related Work

We give an overview of the literature on particle data management, then discuss the method of Devillers and Gandoin [13] (DG), which serves as a base upon which our technical contributions are built.

Particle hierarchies. One of the most common ways to introduce structure to a particle dataset – to facilitate compression – is to impose a spatial hierarchy (a tree) on the particles. Many state-of-the-art compressors follow this approach, where the tree can be one of many types, *e.g.,* binary trees [21], quadtrees [62], octrees [2, 6, 20, 27, 39, 43, 47, 50, 55, 61, 63], k-d trees [12, 13], and bounding-volume hierarchies [59]. A hierarchy helps compression in two ways. First, the higher position bits are "distributed" into coarser tree levels and shared among particles in the form of coarse tree nodes. Thus, in finer nodes, one needs to store only the lower order bits for the particles within, possibly with truncation [26, 28]. Second, regions with no particles (empty space) are quickly identified and carved away, further reducing the number of bits needed to accurately locate particles — a key property that helps both compression and rendering [60, 75, 76]. An octree where each node stores the occupancy of its children is by far the most common approach, and it has been noted [55, 57] that at coarse levels, occupancy-based octrees are better than the k-d tree used by DG [13], since encoding the number of particles in child nodes often requires several bits compared to at most one bit for occupancy. However, here we show that knowing the number of particles in each node can help drive a traversal to make better decisions as to where to refine next.

Level-of-detail. Although a tree naturally provides a progressive coarse-to-fine structure, from which representative particles can be decoded and viewed [19, 59], some techniques generate levels of detail through subsampling [21, 27, 58, 64, 71, 78], which requires no data duplication at coarse levels, and is often faster to compute. Random subsampling [21, 64, 71] may seem a reasonable choice, but leads to suboptimal compression because the bounding volumes for coarse particle subsets are not easily bounded. This is not the case with our lazy wavelet inspired *odd-even* subsampling, which exactly halves the bounding volume at each level. Wavelet-based downsampling is common for compressing mesh vertices [32,44,73]. When a mesh is not readily available, connectivity can be introduced by building a graph [11], local graphs [65,77], or a resampled signed distance field [35] from the particles. Instead, we use a regular grid, which is simple and fast to compute.

Error-guided tree construction and traversal. Minimizing approximation error can be cast as a (hierarchical) clustering problem, where, at each level, particles are clustered and represented with points chosen to minimize some error metric [18,22,26,27,41,53,54]. More data-adaptive hierarchies reorder child nodes based on their predicted occupancy [2], or make planes of k-d divisions adaptive to local variations [12]. The trade-off between quantization (imprecise particles) error and discretization (low particle count) error has been studied both in theory [34] and practice [40, 72] for triangle meshes, where refinement heuristics are given based on geometric distortion measures, including a progressive reconstruction that ranks octree nodes by a priority value [56]. Our *adaptive traversal* instead assumes no connectivity information and works on generic particle data. For reconstructing point-sampled geometry, DT has been shown to be memory efficient whereas BT gives better progressive reconstruction [8]. In fact, BT is by far the more preferred traversal order in the literature. However, we show that the reconstruction quality of DT can be vastly improved through our *odd-even* decomposition of space. Finally, some studies have focused on task-based error metrics for point clouds beyond PSNR [6, 16]. Our *block-adaptive traversal* also facilitates a user-specified error heuristic at decoding time independently of how the data are encoded.

Large-scale and out-of-core techniques. Techniques that handle large data usually organize the data into blocks, so that each block can be randomly accessed and decoded independently as needed [60, 64]. Multilevel hierarchies that treat subtrees as blocks are also not uncommon [10, 17, 19, 33], but previous approaches traverse both the coarse-level tree and the fine-level subtrees (blocks) using BT, which restricts the traversal to a *single progressive order*, where blocks are traversed one by one with potential memory reuse in between. In contrast, by using DT within the blocks, our *block-hybrid trees* allow for *simultaneous, independent, and progressive decoding* of all blocks (*i.e.,* not one block at a time). This approach provides excellent computational gains because the DT puts a hard bound on the memory footprint of traversal.

Standard k-d tree coder. The DG's k-d-tree-based coder [13] (implemented in Google's Draco [1]) has competitive compression ratios while being very fast and general. This method constructs a k-d tree where each node stores the number of particles, n, encapsulated by a bounding box, $\mathbf{B}$. A given node $(\mathbf{B}, n)$ is split into two children $(\mathbf{B}_1, n_1)$ and $(\mathbf{B}_2, n_2)$, with $\mathbf{B}_1$ and $\mathbf{B}_2$ formed by splitting B exactly in the middle along one of the dimensions, and n_1 and n_2 being the numbers of particles bound by $\mathbf{B}_1$ and $\mathbf{B}_2$. By construction, only n_1 needs to be encoded at each node, since n_2, $\mathbf{B}_1$, and $\mathbf{B}_2$ can be inferred. Furthermore, n_1 can be encoded using approximately $\log_2(n+1)$ bits (since $0 \le n_1 \le n$), which becomes smaller toward the leaf level (n decreases). The tree can be implicitly built, traversed, and encoded at the same time, by having the encoder partition an array of particles in-place, following a certain traversal order, which the decoder also follows. In this paper, the term *k-d tree* always refers to a tree constructed with this method.

3 TREE CONSTRUCTION

Most tree-based compression techniques work by encoding tree nodes that implicitly give quantized particle positions, having both the encoder and decoder follow the same traversal order. When run to completion, all tree nodes are visited (in different orders depending on the traversal strategy). Large trees, however, are often only partially decoded to support the task at hand. Algorithm 1 gives a general template for a decoder that can be stopped at any point to produce an approximation to the original particles. The inputs include the total number of particles n_0, an initial bounding box $\mathbf{B}_0$, and a bitstream Bs storing the encoded bits. A data structure C, supporting PUSH and POP operations (*e.g.,* a stack or queue), keeps track of nodes at the traversal front. In each iteration, a node $(\mathbf{B},n)$ is popped from C, and a callback is invoked. If the node is a leaf (as determined by ISLEAF on line 8), a LEAFCALLBACK (line 9) can append $\mathbf{B}$ (now a single point) to a list of output particles. If the node is an inner node, an INNERCALLBACK (line 12) can be used to reconstruct the tree in memory, although this is not needed for decoding. For inner nodes, n_1 (the number of particles in the left child) is decoded from Bs using the knowledge of n (line 14). $\mathbf{B}$ is then split into two halves using a SPLIT procedure (line 16); the resulting two children nodes $(\mathbf{B}_1,n_1)$ and $(\mathbf{B}_2,n_2)$ are pushed back into C, and the process continues until either C is empty or when DONE(Bs) is true (*e.g.,* when enough bits have been read).

When data is reconstructed approximately, the shape of the tree can profoundly affect the quality of reconstruction. For example, on a traditional k-d tree constructed with the typical *k-d split*, which splits a node along a certain dimension (one of x,y,z in 3D), BT often gives coarse representations of the whole space whereas DT reconstructs a spatial region perfectly but completely misses the rest. A k-d split thus offers two contrasting choices: high-cost and coarse reconstructions for both children (with BT), or low-cost and perfect reconstruction for one child but none of the other (with DT). Cost mostly means memory footprint, but a high memory footprint often translates to lower cache utilization and accordingly lower speed. To alleviate this problem, we introduce new ways of constructing a hierarchy that is better suited for a DT than a k-d tree, by allowing the SPLIT procedure on line 16 to be either a regular k-d or an *odd-even split*, which we describe next.

Algorithm 1 Generic traversal framework for a tree-based decoder

Require: n_0 particles in bounding box $\mathbf{B}_0$, bitstream Bs, node container C (*e.g.,* stack, queue), stopping criteria DONE, leaf criteria ISLEAF, callbacks LEAFCALLBACK, and INNERCALLBACK

```
1: function DECODETREE
2:     C.PUSH((B_0, n_0))                ▷ push the node (B_0, n_0) into C
3:     while not DONE(Bs) and not C.ISEMPTY do
4:         (B, n) ← C.POP
5:         if n = 0 then
6:             continue
7:         end if
8:         if ISLEAF(B, n) then
9:             LEAFCALLBACK(B, n)        ▷ callback for leaf nodes
10:            continue
11:        else
12:            INNERCALLBACK(B, n)       ▷ callback for inner nodes
13:        end if
14:        n_1 ← DECODE(n, Bs)           ▷ particles in the left child
15:        n_2 ← n − n_1                 ▷ particles in the right child
16:        B_1, B_2 ← SPLIT(B)           ▷ left and right bounding boxes
17:        C.PUSH((B_1, n_1))
18:        C.PUSH((B_2, n_2))
19:    end while
20: end function
```

3.1 Odd-Even Splits

An *odd-even split* interleaves the child boxes $\mathbf{B}_1$ and $\mathbf{B}_2$ of a box $\mathbf{B}$ by having each contain many disjoint "slices" instead of being a whole contiguous region. This scheme is inspired by multiresolution techniques invented for regular grids, such as hierarchical indexing [51] and the lazy wavelet transform [68]. To realize the odd-even splits, we impose (but do not build) a regular grid on top of the particles by recursively subdividing the bounding box of the particles in the longest dimension, and stop when every cell contains at most one particle. This is the same grid as would be implied by a full k-d tree built over the particles, but here we need the grid dimensions up front before tree construction. To avoid an infinitely large grid and potential numerical floating-point issues caused by particles being too close, we first quantize the particle positions if they are initially expressed in floating point. The full grid, $\mathbf{G}^*$, is associated with the root node, and all other nodes are associated with a different subgrid $\mathbf{G}$ of $\mathbf{G}^*$ as well as the particle subset contained in $\mathbf{G}$. If $\mathbf{G}$ is of dimensions $G_x \times G_y \times G_z$, we index its cells from $(0,0,0)$ to (G_x-1,G_y-1,G_z-1), with the directions of the three axes fixed throughout. In between the cells of a node $(\mathbf{G},n)$, there may be cells of its ancestor or sibling nodes (*i.e.,* $\mathbf{G}$ has strides greater than 1), but we do not consider those for indexing or splitting of $\mathbf{G}$. An odd-even split decomposes a node $(\mathbf{G},n)$ into two child nodes $(\mathbf{G}_e,n_e)$ and $(\mathbf{G}_o,n_o)$, such that $(\mathbf{G}_e,n_e)$ contains the even-indexed grid cells in $\mathbf{G}$ (along the dimension of splitting) along with the n_e particles occupying those cells, and $(\mathbf{G}_o,n_o)$ contains the odd-indexed cells which hold the rest of the particles ($n_o = n - n_e$). When used exclusively for tree construction, the odd-even splits result in an *odd-even tree*, illustrated in Fig. 2 in contrast to a regular k-d tree.

Besides facilitating the odd-even splits, an underlying grid allows us to effectively encode not only sparse particle sets (relative to the size of the grid) but also dense ones. Whenever the number of particles, n, is greater than half the number of cells in $\mathbf{G}$, we can switch from encoding the number of particles in the left child (n_1) to encoding the number of empty cells in the left child, *i.e.,* $|\mathbf{G}|/2 - n_1$, and thus more quickly bound the values to be encoded further down the tree. In the extreme case where every cell contains a particle, our method simply stops after encoding the number of particles at the root node, since the number of empty cells is now 0, whereas DG [13] keeps refining this dense grid until the individual cells.

Picking either the odd or the even subgrid can also be viewed as a sampling method. For odd-even sampling, the subgrids $\mathbf{G}_o$ and $\mathbf{G}_e$ are half the size of $\mathbf{G}$ (unlike for random sampling), which is important for locating the particles using fewer bits. Still, any subsampling method potentially compromises compression quality. An odd-even tree, which consists entirely of odd-even splits, is not conducive to compression since the density of particles for many datasets is sparse relative to the imposed grid cells (*i.e.,* most grid cells are empty). In such cases, empty cells are "distributed" into the odd and even subtrees, effectively increasing the number of tree nodes to be coded to locate the particles. Instead, a k-d split could be used to quickly cull away whole empty subtrees. In practice, we have seen an almost 100% increase in compressed size with odd-even trees compared to regular k-d trees. We discuss next how to use both split types to build a hybrid tree that facilitates high-quality reconstructions with DT and is also conducive to compression.

3.2 Hybrid Trees

To reduce the adverse impact of odd-even splits on compression, we can restrict the application of odd-even splits to only one type of child node (left or right). Here, we borrow a technique from the wavelet literature, where multiresolution decomposition is done by recursively transforming only the low-pass filtered subband in every iteration. Similarly for hybrid trees, without loss of generality, odd-even splits are used recursively only on the left child node, and k-d splits exclusively on the other. Furthermore, once a k-d split is

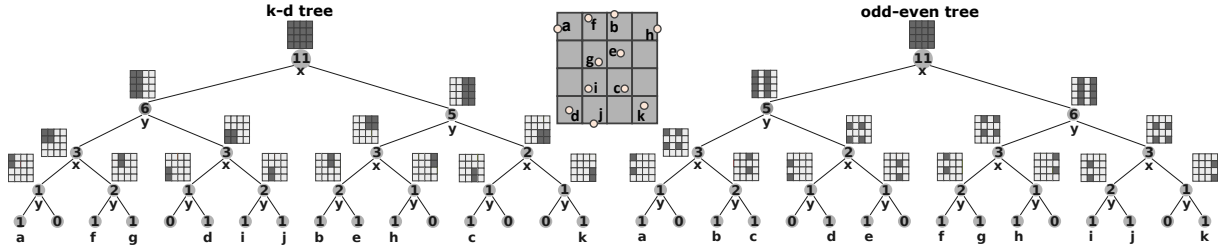
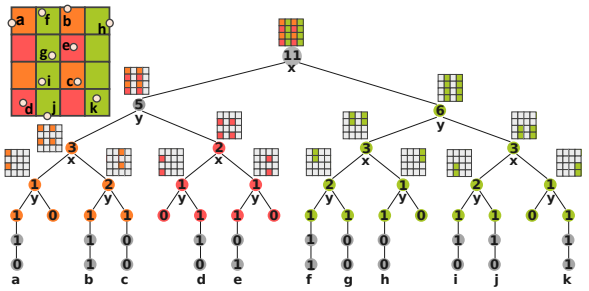
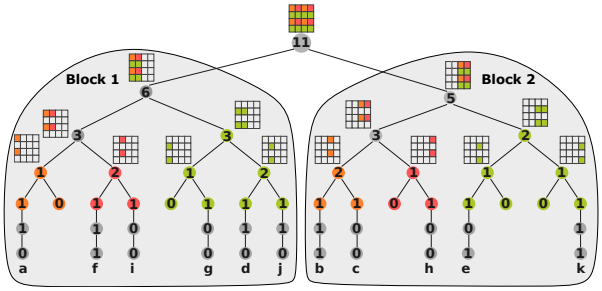


Figure 2: An example dataset with 11 particles (denoted a to k) on a 4×4 grid, from which we build a k-d tree (left) and an odd-even tree consisting of purely odd-even splits (right). Our odd-even splits partition space by interleaving odd-indexed and even-indexed grid cells at each tree level.



(a) A hybrid tree with three resolution levels, created with two odd-even splits (at the root and its left child)



(b) A block-hybrid tree with two blocks, each of which is a hybrid tree with three resolution levels

Figure 3: (a) A *hybrid tree* created using a particular combination of odd-even splits (with different colored child nodes) and the standard k-d splits (same colored child nodes). (b) A *block-hybrid tree* created by exclusive uses of k-d splits at shallow depths and hybrid trees further down. Both trees are constructed for the same 11-particle in Fig. 2, with additional (conceptual) tree nodes for in-cell refinement bits, shown in gray.

used, subsequent descendant splits are all k-d splits. We also use the convention that the left child contains the even-indexed cells of a parent's grid (*i.e.,* it is $(\mathbf{G}_e, n_e)$). From top to bottom, every odd-even split creates a new, coarser *resolution level*, which consists of nodes in the even-indexed subtree. We build a hybrid tree by applying the odd-even splits for a sequence of exactly $R-1$ nodes (on $R-1$ depth levels) by traversing down the left child $R-2$ times from the root, resulting in R *resolution levels* in total (see Fig. 3a for an example with $R=3$). In practice, R can be automatically set so that the chain of odd-even splits ends when no particles are left in the last left child, or when it contains one particle located to full quantized precision. Constructed this way, the impact of our hybrid trees on compression is minimal; in the worst case, we have noticed only a 5% increase in compressed size compared to k-d trees.

By design, a DT on a hybrid tree visits the resolution levels from coarse to fine (assuming that left children are always visited first), which now produces not a depth-first walk but a breadth-first sampling of space, still at the same memory footprint as a DT on a k-d tree. Although hybrid trees are designed with DT in mind, they also support BT (see Fig. 6), noting that BT is best used only within each resolution level and not across resolution levels (note that nodes at the same *depth level* may belong to different *resolution levels*, see *e.g.,* Fig. 3a). Our proposed hybrid tree is also only one of the many possible combinations of k-d and odd-even splits, and perhaps different combinations may be useful for different purposes. Although hybrid trees with DT work well in providing a low-cost sampling of space, they still require the encoder and decoder to follow the same steps, which, if applied to the whole tree, can hinder task-oriented, random-access, and parallel decoding, all important when working with large data. We next discuss a solution for decoupling the encoder and decoder to support all the above.

3.3 Block-Hybrid Trees

Here, we combine k-d splits and odd-even splits across the depth of the spatial hierarchy to create a *block-hybrid tree*, which contains a *coarse* portion (several k-d splits at the top), a *medium* portion (several hybrid trees, or *blocks*), and a *fine* portion (refinement bits). After the coarse and medium portions, no cell of the full grid $\mathbf{G}^*$ contains more than one particle. The inner-cell refinement bits further locate individual particles within the respective cells. These bits are technically also present for the hybrid tree previously described, but we discuss them here since they are not relevant to odd-even splitting. We start by building a k-d tree from the top, but only to a certain depth. The leaves of this coarse k-d tree create multiple subtrees (blocks); for each leaf, we build a local hybrid tree. Although a k-d tree can also be made block-based, our block-hybrid tree enables low-cost DT of the blocks by turning each block into a hybrid tree. Fig. 3b shows an example of how a block-hybrid is built for our running example with 11 particles in 2D.

At encoding time, the compressed bitstream for the coarse portion is stored once. The bitstreams for individual blocks are stored and indexed separately (see Fig. 4) so that at decoding time we can freely pause and jump to any of them to continue decoding/traversal, which enables random, parallel access as well as task-oriented decoding. Unlike the coarse and medium portions, the in-cell refinement bits are stored verbatim since they are more random and difficult to compress. Even though these fine bits can conceptually be considered part of the tree (as depicted in Fig. 3a), we treat them separately. We store the fine bits last in a block's bitstream and in bit plane order (*i.e.,* BT order), where each bit plane contains one refinement bit for each particle in the block, in the order that a medium-phase DT visits the particle. The fine bits are not stored depth-first as linear chains following respective particles, since a DT on such chains can have a high coding cost when the input particles are sparse but specified with high precision (thus, a majority of bits are fine bits), which is common for scientific data (see *e.g., molecule* in Fig. 8).

To decode a particle's position by traversing a tree is to reconstruct the bits of its quantized integer coordinates. Assuming the coordinate bits in x, y, z are interleaved into a Morton code, with a k-d tree, the bits are reconstructed from left to right (MSB to LSB, see Fig. 5). Indeed, a DT on a k-d tree is well known to correspond to sorting or indexing the particles by their Morton codes. If we ignore the fine bits that locate particles within individual cells, a DT on an odd-even tree corresponds to indexing the particles by their *reversed* Morton codes (LSB to MSB), because we use the LSB, which determines whether a particle is "odd" or "even", to assign it to a resolution

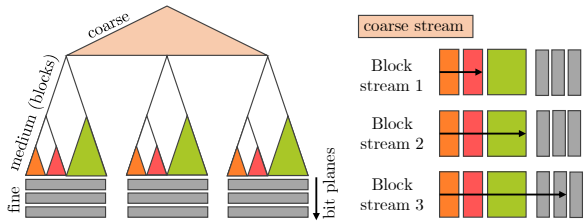


Figure 4: A schematic depiction of our block-hybrid tree, with subtrees colored by resolution level (left). The blocks' bitstreams are stored separately to support independent decoding of blocks (right).

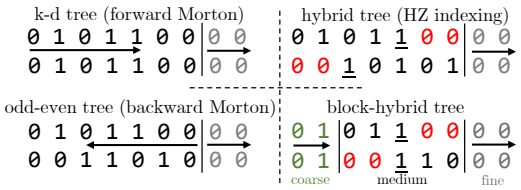


Figure 5: A tree implies an indexing of particles by manipulating Morton codes. Input and output Morton codes are at the top and bottom, with arrows indicating directions of the output bits. K-d trees and odd-even trees use forward and backward Morton codes. Hybrid trees use HZ indexing [51] where the position of the least significant 1-bit (underlined) indicates the resolution level. Block-hybrid trees replace forward Morton codes with HZ indexing for the medium portion.

level. When its chain of odd-even splits is carried to the end, a hybrid tree corresponds to Hierarchical-Z (HZ) indexing, proposed by Pascucci and Frank [51] and generalized by Hoang *et al.* [25]. In this scheme, the position (from the right) of the least significant one-bit indicates the particle's resolution level, and the bits to its left constitute its index within the level. Finally, a block-hybrid tree performs HZ indexing but only for the medium portion.

4 Tree Traversal

Once a tree is constructed, the space of possible traversal strategies contains many valuable trade-offs (*e.g.,* the one between BT and DT as discussed). We introduce *adaptive traversal* (AT) and *block-adaptive traversal* (BAT), both of which improve qualities of partial reconstruction by being more adaptive than BT and DT. AT is primarily designed for k-d trees but works also for hybrid trees, whereas BAT is specifically designed for block-hybrid trees.

4.1 Adaptive Traversal

Although universally used, both BT and DT are static traversal orders that do not take into account the actual node values. To achieve better reconstruction, the traversal should be more adaptive, *i.e.,* nodes with a potentially low cost of traversal (in terms of number of bits to decode) and high gains (in terms of reduction of error) ought to be prioritized. We therefore generalize the container used for traversal (C in Algorithm 1) from a stack (for DT) or a queue (for BT) to a priority queue, which allows prioritizing nodes that are more important for traversal with respect to some error metric and coding cost. That is, the priority queue allows us to perform rate-distortion optimization during traversal.

Our framework is agnostic to different metrics; here we use a heuristic that assigns an importance score to a given node based on how "densely packed" the node is (in number of particles) and how many bits are required to decode the node. Concretely, given a node $(\mathbf{B}, n)$ (see Section 2), we define its importance score as

$$\frac{n(d/2)^2}{\log_2(n+1)}, \tag{1}$$

where d is the length of $\mathbf{B}$ along the axis of splitting at the given node. The denominator captures the cost of decoding the node $(\mathbf{B}, n)$, and the $(d/2)^2$ term captures the (squared) error reduction per-particle obtained by decoding this node, assuming the extreme case where all n particles fall into either the left or the right child. Intuitively, given the same number of nodes, larger nodes (in terms of $\mathbf{B}$) are traversed first; conversely, given the same bounding box size, denser nodes are prioritized (because errors are reduced for more particles). We expect AT with this heuristic to work best (compared to BT) when the particles are highly nonuniformly distributed and therefore the importance scores of same-depth nodes are notably different.

Our importance score is simple yet works well in practice to improve the rate-distortion trade-off over BT for a wide range of datasets (see Section 5). Regardless, this score is still a heuristic and thus is not guaranteed to work for all datasets. We also demonstrate modifications (see Fig. 6) to the importance function by reducing the emphasis on node density (*i.e.,* by removing n from the numerator), which we have observed to work better for particles representing a surface. We anticipate that in future work, many more importance functions can be devised depending on the data and task at hand, but all should be supported by AT. Although AT improves on BT in reconstruction quality, it has a similarly high memory footprint in practice (see Section 5.3). Furthermore, AT works with individual nodes and not blocks, so it cannot be used as is to efficiently traverse a block-hybrid tree. In the next section, we generalize AT to *block-adaptive traversal* (BAT), which is still data-adaptive but works with entire blocks instead of individual nodes, and has asymptotically constant memory footprint similarly to that of DT.

4.2 Block-Adaptive Traversal

For a block-hybrid tree, both the encoder and the decoder completely traverse the coarse portion (using BT or AT) before starting on the medium portion. For the medium portion, our decoder ranks blocks using a global heap, where each element represents a block and each block's priority can be dynamically updated. For each block, we also keep a stack so that decoding of any block (with DT) can be paused and resumed as needed. BAT happens in multiple iterations; in each iteration, the block at the top of the heap is traversed for either a certain number of decoded bytes or a certain number of particles. At the end of an iteration, the priority of the current block is updated using a scoring function. We find it nontrivial to generalize the scoring function designed for AT (Formula 1), since the front of each block (a single node) contains too little information to estimate either the reconstruction error or the remaining coding cost for the entire block. Although it may be possible to estimate this rate-distortion trade-off by combining the information stored at all nodes of each stack, a thorough investigation is left for future work. Instead, we opt for a simpler criterion to rank blocks during traversal, which works well in practice. Given two partially decoded blocks, we always prioritize the one at a coarser resolution level. If two blocks are at the same resolution level, we prioritize the one with a smaller value of n_l^*/n_l, where n_l is the total number of particles in the block on level l, and n_l^* is the number of those already visited by the per-block DT. By construction, nodes on the same resolution level have subgrids with the same strides (*i.e.,* spacing between neighboring cells). Since the distance between two particles in the same subgrid is bound by its stride, the stride itself is a good proxy for the error of the reconstruction. Therefore, forcing the blocks to refine to the same resolution level (with subgrids of the same stride) effectively forces approximately the same bound for reconstruction error everywhere. Once the blocks are at the same resolution level, the ratio n_l^*/n_l indicates how much of the given level has been traversed. Compared to the simple hybrid tree, our block-hybrid tree thus distributes error more uniformly (see Fig. 8).

Pseudocode for BAT is given in Algorithm 2. B denotes the current block being traversed, with B.l being the block's resolution level, B.n its total number of particles and B.n^* its number of visited particles, as described above. In the TraverseCoarse

and TRAVERSEMEDIUM functions, Node denotes the current node being traversed, with Node.l storing its resolution level and Node.n its number of particles. It may seem that, to support progressive decoding of the fine bits, we need to maintain an array of particle positions per block (B.*particles* on line 38) that get refined with each coming bit. However, these extra states are not required if only medium-phase decoding is needed. To decode the fine bits for very large datasets, regardless of the approach, special care has to be taken at the application (not decoder) level regarding how to work with a large set of output particles with limited memory. Examples include out-of-core techniques such as streaming, caching, or memory mapping. For these reasons, we do not consider any data structures used for storing the output particles as part of the internal memory footprint for our progressive decoder.

Given a tree of height L, the memory footprint of BAT is controlled by L_c, the height of the coarse k-d tree. Since there are at most 2^{L_c-1} blocks and each block contains a stack of size at most $L-L_c$, the number of elements in the different containers is bound by 2^{L_c-1} (queue) $+2^{L_c-1}(L-L_c)$ (stacks) + 2^{L_c-1} (heap). In contrast, the memory footprint for BT, if used exclusively for the whole hierarchy, is bound by 2^{L-1}, which is often several orders of magnitude larger, since a typical L_c is only half of L. In practice, the L_c chosen should be large enough so that the error is more uniformly distributed and that random access is more fine-grained, but also small enough to not turn BAT into BT and also to not create too many blocks, which require extra indexing bits for random access. 32 bits are needed to keep track of the number of particles n in the node, provided L_c is chosen so that each block has at most 2^{32} particles. A node's grid $\mathbf{G}$ and its resolution level l can be deduced from the path connecting it to the tree's root, which can be stored once per block using at most 32 bits. The encoder would also need to keep track of the range of particles that each node encompasses, for a total of 64 additional bits per node. As mentioned in Section 3, we do not explicitly construct the tree in memory, as node values are simply encoded to and decoded from a bitstream, following a certain traversal order. Therefore, only the size of the data structures used for traversal, and not that of the tree itself, count toward our memory footprint. Nevertheless, the INNERCALLBACK and LEAFCALLBACK functions can be used to construct the tree itself in memory, which might serve as an acceleration data structure for tasks such as ray tracing.

In contrast to any previously discussed traversal order, with BAT, the encoder and decoder need to follow the same traversal order only for the coarse portion (*i.e.,* BT or AT for the k-d subtree) and for each of the blocks (*i.e.,* DT within each hybrid subtree). Otherwise, the scoring function for ranking blocks can be supplied at decoding time, and different scoring functions may be used depending on the task at hand. For example, a renderer may prioritize blocks closer to the camera; another example is when the user specifically wants to refine a block because it contains a feature of interest.

5 Evaluation and Results

We evaluate the efficacy of our proposed solutions through various experiments. Here, we quantify the reduction in data as *bits-per-particle* (bpp), measured by dividing the number of bits decoded by the total number of particles in the original dataset. All datasets discussed here have 96 bpp when uncompressed; each particle is originally specified using 32-bit floating point coordinates, which are then quantized to 32-bit integers prior to experiments. We also use the notation $|C|$ to refer to the size of container(s) used for traversal, in terms of number of elements. We use both the standard peak-signal-to-noise ratio (PSNR) and rendered images, when appropriate, to assess the quality of partial reconstructions. To generate an approximation when a traversal stops midway, we output one (random) particle within a node's bounding grid ($\mathbf{G}$) for each node in the traversal front (in container C). Finally, both "BT on k-d tree" and "DT on k-d tree" are the baseline DG [13] methods; all other traversal-tree combinations are contributions of this paper.

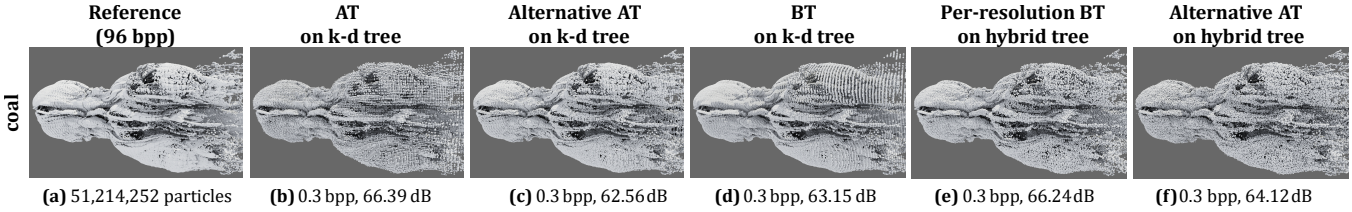
Algorithm 2 Functions for block-adaptive traversal

Require: Heap, Stacks, CoarseStream, BlockStreams, BitsRead, MaxBits are global variables

```
 1: function TRAVERSEBLOCKADAPTIVE
 2:     if Heap.EMPTY then                          ▷ still in coarse phase
 3:         TRAVERSECOARSE(CoarseStream)
 4:     end if
 5:     while BitsRead < MaxBits and not Heap.EMPTY do
 6:         B ← Heap.POP    ▷ now traverse block B at Heap's top
 7:         S ← Stacks[B.Id]                        ▷ block B's current stack
 8:         Bs ← BlockStreams[B.Id]                 ▷ block B's bitstream
 9:         if B.n* < B.n then                      ▷ not all particles visited
10:             TRAVERSEMEDIUM(B, S, Bs)
11:             Heap.PUSH(B)
12:         else  ▷ all particles in B visited, do fine phase traversal
13:             TRAVERSEFINE(B, Bs)
14:             if not Bs.EMPTY then
15:                 Heap.PUSH(B)
16:             end if
17:         end if
18:     end while
19: end function
20:
21: function TRAVERSECOARSE
    ▷ This is Algorithm 1 with a queue container, stopping criteria
    BitsRead ≥ MaxBits, bitstream CoarseStream, and the following LEAFCALLBACK
22:     B.l ← 0; B.n ←Node.n; B.n* ←0
23:     Heap.PUSH(B)                                ▷ push leaf block B into Heap
24:     S ← empty stack; S.PUSH(Node.left); S.PUSH(Node.right)
25:     Stacks[B.id] ← S
26: end function
27:
28: function TRAVERSEMEDIUM(B, S, Bs)
    ▷ This is Algorithm 1 with the stack S as container, stopping
    criteria BitsRead ≥ MaxBits, bitstream Bs, and the following
    LEAFCALLBACK
29:     B.n* ← B.n* + 1; B.n*_l ← B.n*_l + 1 ▷ a new particle visited
    ▷ and the following INNERCALLBACK
30:     if Node.l ≠ B.l then          ▷ reaching a finer resolution level
31:         B.l ← Node.l; B.n_l ← Node.n    ▷ update current level
32:         B.n*_l ← 0                ▷ reset particle count for current level
33:     end if
34: end function
35:
36: function TRAVERSEFINE(B, Bs)
    ▷ For simplicity, always read an entire bit plane of B from Bs
37:     for each particle P ∈ B.particles do
38:         Bit ← READONEBIT(Bs)
39:         P ← REFINE(P, Bit, B)
40:         BitsRead ← BitsRead + 1
41:     end for
42:     B.l ← B.l + 1
43: end function
```

5.1 Adaptive Traversal of k-d Trees

AT (with the proposed scoring heuristic, Formula 1) on k-d trees improves the rate-distortion trade-off over BT on k-d trees for a wide range of datasets (see Fig. 7, left). We do not include DT in the same figure since the root-mean-square error for DT is often exceptionally high due to whole regions missing, rendering L_2-norm-based quality metric such as PSNR less meaningful. Here, PSNR is computed from the mean-square point-wise distance between every particle



(a) 51,214,252 particles **(b)** 0.3 bpp, 66.39 dB **(c)** 0.3 bpp, 62.56 dB **(d)** 0.3 bpp, 63.15 dB **(e)** 0.3 bpp, 66.24 dB **(f)** 0.3 bpp, 64.12 dB

Figure 6: Reconstruction results for alternative combinations of traversal orders and trees, including the use of an alternative scoring function for AT to obtain a better reconstruction visually (c), even at a lower PSNR. All reconstructions are at 0.3 bpp. Although not canonical, BT and AT on hybrid trees are very possible combinations, which may sometimes be preferable than BT on k-d trees, as is perhaps the case here.

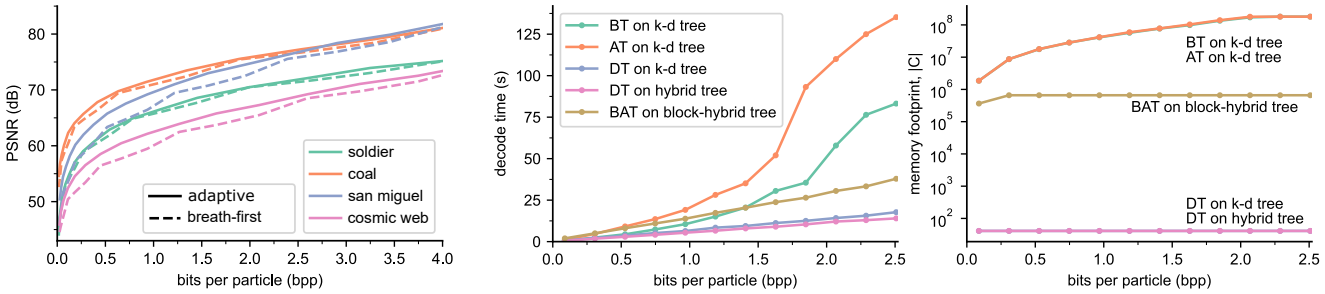


Figure 7: Left: Rate-distortion curves demonstrate that AT outperforms BT. Middle and right: decode time and memory footprints for combinations of trees and traversal methods, plotted for the *detonation-small* dataset. DT and BAT achieve constant memory footprint and linearly scaled decode time in number of bits, whereas AT and BT require orders of magnitude more memory, and also much faster growing decode time.

in the reference and the closest particle in the reconstruction, with the signal range being the maximum dimension of the bounding box for the reference particles. Visual demonstration of the differences between low-bit-rate reconstructions using BT and AT is provided in Fig. 8 (see the first green-highlighted column pair). We render at low bit rates the outputs of the various traversal and tree combinations with OSPRay [74]. The bit rates are chosen so that visual differences among the combinations are most apparent. For the *girl* dataset, AT (a3) provides a better covering of space compared to BT (a2), which follows a strict order on each tree depth level, creating a visible seam where the resolution changes. The same phenomenon happens for *fissure* (comparing b2 and b3). For *soldier*, although less noticeable, AT (d3) generates a smoother surface as well. For *cosmic web*, AT (f3) captures the points of interest — clusters of particles (galaxies) — better by favoring densely packed nodes. Overall, by being more data-adaptive, AT can provide significant improvements over BT, both visually and quantitatively (in PSNR).

Alternative AT. Our default scoring function for AT (Formula 1) does not always work well for all datasets. For example, the rendering of the *coal* dataset (which contains simulated coal particles) in Fig. 6 contains occlusion because particles on the "surface" are given more importance. Because of occlusion, however, the majority of particles in dense tree nodes are hidden from view, but these are also nodes that our scoring function deems important. To improve visual quality, we instead use an alternative scoring function, removing n from the numerator, to prevent an overemphasis on dense nodes. The result is a reconstruction with lower PSNR but improved visual quality (*i.e.,* more similar to the reference, compare Fig. 6b and Fig. 6c), indicating that PSNR does not always capture visual quality. For datasets where the particles are intended to be viewed as surfaces, our alternative scoring function often produces better visualizations, because nodes containing surface particles are given higher priority, even though they tend to be more sparse.

5.2 Traversals of Hybrid and Block-Hybrid Trees

Fig. 8 shows that DT on our hybrid tree is able to recover coarse reconstructions of the whole space instead of very fine reconstructions of only parts of the data, as is the case with DT on a k-d tree (see the second green-highlighted column pair). A specially difficult case for DT on hybrid tree is *molecule*, where the distribution is very sparse (few but precise particles). In such cases, precisely refining a coarse subset of particles is not useful (see Fig. 8 (c5)). In general, BAT on block-hybrid tree often improves upon DT on hybrid tree visually by distributing error more uniformly throughout space. This observation is most visible when comparing (a5) with (a6), (c5) with (c6), and (e5) with (e6). Note that in terms of PSNR, BAT on block-hybrid tree tends to perform worse than BT or AT on k-d trees and sometimes even DT on hybrid trees. Visually, however, BAT typically outperforms all other methods (most strikingly in the case of *molecule*), often producing a less blocky look on densely sampled surfaces compared to BT or AT (see *girl* or *soldier*). We also note the case where BAT fails visually (*cosmic web*) compared to DT on hybrid tree (see (f5) and (f6)) because when dense regions are clearly preferred over sparse ones, aiming for uniform refinement (in terms of gaps between grid cells) is not a good strategy. Finally, we note that our hybrid and block-hybrid trees can often generate significantly fewer particles at the same bit rate compared to BT on k-d trees (see *fissure*, *dam break*, and *cosmic web*), which should greatly benefit downstream processing tasks (*e.g.,* rendering) because fewer particles often means faster speed.

5.3 Speed and Memory Footprint

Fig. 7 (right) shows that DT on any tree and BAT on block-hybrid tree achieve a constant memory footprint, whereas AT and BT require orders of magnitude more memory. Compared to DT and BAT, BT and AT also become slower very quickly. Compared to BT, our AT requires the same memory footprint and is slower, but can improve reconstruction quality by a good margin (as discussed in Section 5.1). The decode time for BAT grows faster than that of DT (on both k-d and hybrid trees) and its memory footprint is also higher, while still being asymptotically constant (Fig. 7, middle). The trade-off is higher reconstruction quality (Fig. 8). Notwithstanding its lack of features compared to BAT on block-hybrid tree, perhaps the best trade-off is had with DT on hybrid tree, which vastly improves reconstruction quality over DT on k-d tree almost for free. Based on these results, we recommend AT on k-d trees for small data and BAT on block-hybrid trees for large data, with AT limited to only the coarse k-d portion at the top.

We also test the scalability of BAT on block-hybrid tree against the state-of-the-art octree compressor, MPEG [63], using the TMC3 [3]

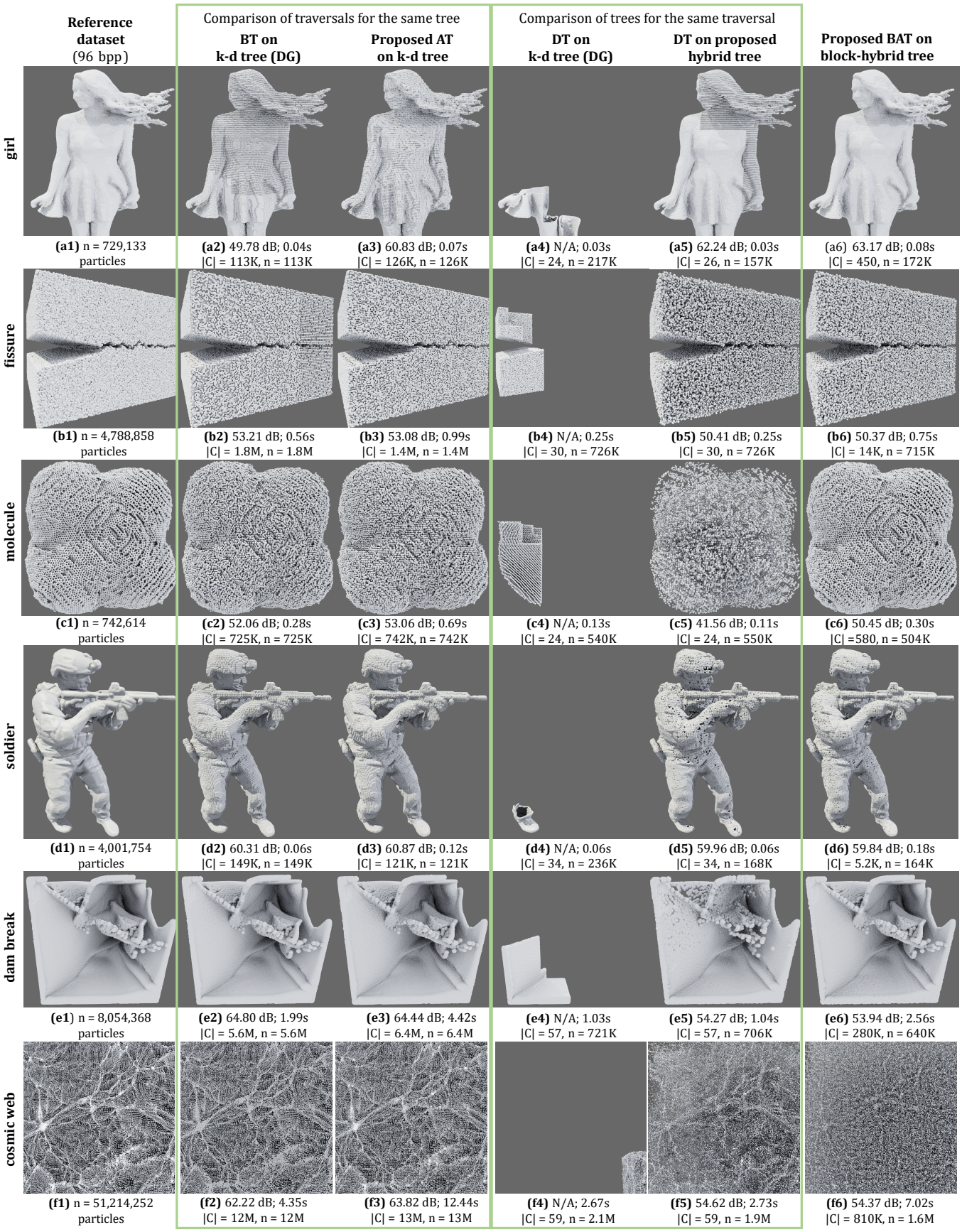


Figure 8: Visual comparison of the different traversal-tree combinations (columns) discussed in this paper for six datasets (rows). The reduced datasets are shown at 1.1 bpp (*girl*), 1.3 bpp (*fissure*), 4.4 bpp (*molecule*), 0.4 bpp (*soldier*), 3.1 bpp (*dam break*), and 1.3 bpp (*cosmic web*).

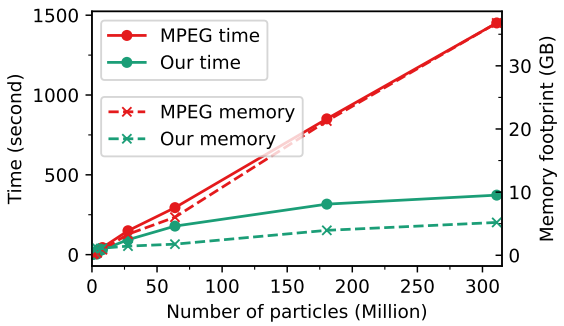


Figure 9: Our block-based encoder is almost 5× to 7× less expensive than MPEG's [63] and the costs also grow at much slower rates.

reference implementation. We encode eight datasets in increasing numbers of particles (the largest one has about 400 M particles) and record the encoding time and memory usage of both methods. Fig. 9 shows that our block-based encoder is several times faster than MPEG's encoder and, at the same time, uses an order of magnitude less memory for the larger datasets. Furthermore, our method's time requirement and memory footprint grow at much slower rates. For decoding, a fair comparison is difficult to obtain since MPEG decodes and outputs one block at a time, whereas we maintain all the states necessary for simultaneous progressive decoding of all blocks (important for cross-block bit allocation). Nevertheless, MPEG crashes while decoding the largest dataset here.

5.4 Lossless Compression Ratio

For completeness, we compare lossless compression ratios among four methods: DG [13], our block-hybrid tree, MPEG [63], and LASZip [29] for several datasets in Table 1. For lossless compression of point clouds, LASZip is an industry standard, and MPEG represents the state-of-the-art in compression ratio. Table 1 shows that our block-based tree achieves practically the same lossless compression ratios against that of DG (with differences of at most 5%), which means our use of the odd-even splits does not degrade compression significantly. Note that while being comparable with DG in lossless compression ratio – which matters most for storage – our method achieves much better data quality for lossy reconstruction with better overall memory-quality trade-offs (*e.g.,* as shown in Fig. 8). Both DG and our method compress significantly better than LASZip in most cases, and MPEG compresses the best with its sophisticated context modeling, although for many cases its compression ratio is no better than ours. Three datasets deserve further comments: *soldier*, *detonation-small*, and *random-80*. Unlike the other high-precision scientific datasets, *soldier* is a densely sampled scanned surface, a case that MPEG is specifically designed for, so it unsurprisingly performs well here. *detonation* contains highly regular, repeating particle arrangements (see Fig. 1), which MPEG and LASZip (to a lesser extent) take advantage of, while ours and DG's do not. However, with further dictionary compression applied on top of our compressed bitstream, our compression ratio increases from 3.13 to 10.3, which is the same as that of LASZip's. Finally, *random-80* is a synthetically generated dataset where 80% of the grid cells contain particles. Since our grid-based approach scales gracefully from sparse to dense data by switching to coding empty cells when there are relatively more particles (see Section 3.1), it compresses twice better than DG's and four times better than LASZip, whereas MPEG simply crashes. Most particle datasets in practice are sparse relative to the grid size, but future data will likely become denser as more particles are captured and simulated.

6 Conclusion and Future Work

We have presented novel tree construction and traversal techniques that achieve a better balance between data quality and resource requirements compared to other state-of-the-art particle compressors. Our *adaptive traversal* approach improves over the static breadth-first traversal with respect to a user-defined error heuristic. Compared to k-d trees, our *hybrid trees* enable high-quality depth-first traversal. The *block-hybrid tree* allows not only independent, low-footprint encoding and decoding of blocks, but also higher reconstruction quality compared to all other approaches. Our *block-adaptive traversal* approach allows flexible, error-guided reconstructions at decoding time independent of how data is compressed. All of our proposed techniques benefit equally the encoder and decoder. Working together, our contributions amount to a highly flexible and scalable particle compression system, which compares favorably to the state-of-the-art MPEG standard in memory and speed, both in absolute terms and in rates of growth.

datasets	# particles	DG	Ours	MPEG	LASZip
crystal [49]	16,384	2.10	2.08	2.44	1.56
molecule [75]	742,614	2.19	2.18	2.18	1.64
salt [5]	1,832,808	2.36	2.36	2.33	1.51
fissure [75]	4,788,858	2.54	2.54	3.50	2.13
*soldier [36]	4,001,754	13.70	13.40	21.00	5.56
san miguel [46]	3,693,102	3.36	3.20	3.46	2.02
dam break [67]	8,054,368	2.71	2.69	2.69	1.85
coal [48]	27,693,140	3.45	3.42	3.36	2.20
cosmic web [75]	51,214,252	3.17	3.12	3.06	1.72
*detonation-small [7]	180,426,240	3.08	3.13	24.20	10.30
*random-80	1,678,152	42.70	95.50	CRASHED	27.40

Table 1: Comparison of lossless compression ratios across four compression methods for the several datasets used in our experiments. We give further comments in the text for the datasets marked with *.

Our method uses the same low-level node encoding scheme as DG's [13] and does not take advantage of global redundancy. It therefore does not improve lossless compression ratios. To realize the odd-even splitting scheme, we need to quantize particle positions to avoid inaccuracy caused by floating-point operations, but there may exists techniques that maintain accuracy without quantization. We also do not tackle compression of attributes other than positions, although odd-even splitting – being based on the lazy wavelet transform – might suggest a wavelet-based compression scheme for attributes. We see opportunities for more in-depth studies of the trade-offs between odd-even and k-d splits, as well as between various possible combinations of tree and traversal types. The idea of odd-even splits may be generalized to octrees, although perhaps with different trade-offs. In this paper, we assume particles are decompressed into an output buffer to be consumed by downstream visualization or analysis tasks. However, our inherent tree structures can also be used to accelerate certain tasks such as nearest-neighbor queries, occlusion culling, or empty-space skipping in rendering. There, it remains to be seen how our odd-even splitting mechanism affects such application-level concerns, and more generally, to what extent our hybrid and block-hybrid trees can be used for noncompression purposes. Finally, it is also important to study task-oriented error metrics/heuristics and their utility to drive either tree construction, tree traversal, or both.

Acknowledgments

This research is supported in part by the U.S. Department of Energy (DOE) under Award Number(s) DE-FE0031880 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the DOE Office of Science and the National Nuclear Security Administration. This work is supported in part by NSF OAC award 1842042, NSF OAC award 1941085, and NSF CMMI award 1629660. This work was also performed under the auspices of the DOE by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 and supported by the LLNL-LDRD Program under Project No. 17-SI-004. LLNL release number: LLNL-CONF-821057.

REFERENCES

[1] Draco: Geometric coding for dynamic voxelized point clouds. https://github.com/google/draco. Accessed: 2021-03-31.

[2] A generic scheme for progressive point cloud coding.

[3] MPEG-PCC-TMC13: Geometrybased point cloud compression. https://github.com/MPEGGroup/mpeg-pcc-tmc13. Accessed: 2021-03-31.

[4] OpenTopography. https://portal.opentopography.org/datasets. Accessed: 2021-03-31.

[5] Scientific visualization contest 2016. https://www.uni-kl.de/sciviscontest/. Accessed: 2021-03-31.

[6] E. Alexiou and T. Ebrahimi. On the performance of metrics to predict quality in point cloud representations. In *Applications of Digital Image Processing XL*, volume 10396, page 103961H, 2017.

[7] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight. Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices. *SIAM Journal on Scientific Computing*, 38(5):S101–S122, 2016.

[8] M. Botsch, A. Wiratanaya, and L. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Eurographics Workshop on Rendering*, EGRW '02, pages 53–64, 2002.

[9] S. Byna, A. Uselton, D. Knaak, and H. He. Trillion particles, 120,000 cores, and 350 TBs: Lessons learned from a hero I/O run on Hopper. In *Cray User Group conference*, CUG '13, 2013.

[10] K. Cai, Y. Liu, W. Wang, H. Sun, and E. Wu. Progressive out-of-core compression based on multi-level adaptive octree. In *ACM International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCIA '06, pages 83–89, 2006.

[11] S. Chen, D. Tian, C. Feng, A. Vetro, and J. Kovačević. Fast resampling of three-dimensional point clouds via graphs. *IEEE Transactions on Signal Processing*, 66(3):666–681, 2018.

[12] G. Cirio, G. Lavoué, and F. Dupont. A Framework for Data-Driven Progressive Mesh Compression. In *International Conference on Computer Graphics Theory and Applications*, GRAPP '10, pages 5–12, 2010.

[13] O. Devillers and P.-M. Gandoin. Geometric compression for interactive transmission. In *IEEE Visualization*, VIS '00, pages 319–326, 2000.

[14] F. Di Natale, H. Bhatia, T. S. Carpenter, C. Neale, S. K. Schumacher, T. Oppelstrup, L. Stanton, X. Zhang, S. Sundram, T. R. W. Scogland, G. Dharuman, M. P. Surh, Y. Yang, C. Misale, L. Schneidenbach, C. Costa, C. Kim, B. D'Amora, S. Gnanakaran, D. V. Nissley, F. Streitz, F. C. Lightstone, P.-T. Bremer, J. N. Glosli, and H. I. Ingólfsson. A massively parallel infrastructure for adaptive multiscale simulations: Modeling RAS initiation pathway for cancer. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 1–16, 2019.

[15] J. Diemand, R. Angélil, K. K. Tanaka, and H. Tanaka. Large scale molecular dynamics simulations of homogeneous nucleation. *The Journal of Chemical Physics*, 139(7):074309, 2013.

[16] O. Dovrat, I. Lang, and S. Avidan. Learning to sample. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR '19, pages 2755–2764, 2019.

[17] Z. Du, P. Jaromersky, Y. Chiang, and N. Memon. Out-of-core progressive lossless compression and selective decompression of large triangle meshes. In *Data Compression Conference*, DCC '09, pages 420–429, 2009.

[18] Y. Fan, Y. Huang, and J. Peng. Point cloud compression based on hierarchical point clustering. In *Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, APSIPA '13, pages 1–7, 2013.

[19] R. Fraedrich, J. Schneider, and R. Westermann. Exploring the millennium run - scalable rendering of large-scale cosmological datasets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1251–1258, 2009.

[20] D. C. Garcia, T. A. Fonseca, R. U. Ferreira, and R. L. d. Queiroz. Geometry coding for dynamic voxelized point clouds using octrees and multiple contexts. *IEEE Transactions on Image Processing*, 29:313–322, 2020.

[21] E. Gobbetti and F. Marton. Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models. *Computers & Graphics*, 28(6):815–826, 2004.

[22] P. Goswami, F. Erol, R. Mukhi, R. Pajarola, and E. Gobbetti. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. *The Visual Computer*, 29(1):69–83, 2013.

[23] S. Grottel, P. Beck, C. Müller, G. Reina, J. Roth, H. Trebin, and T. Ertl. Visualization of electrostatic dipoles in molecular dynamics of metal oxides. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2061–2068, 2012.

[24] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka, V. Vishwanath, Z. Lukić, S. Sehrish, and W.-k. Liao. HACC: Simulating sky surveys on state-of-the-art supercomputing architectures. *New Astronomy*, 42:49–65, 2016.

[25] D. Hoang, B. Summa, H. Bhatia, P. Lindstrom, P. Klacansky, W. Usher, P. Bremer, and V. Pascucci. Efficient and flexible hierarchical data layouts for a unified encoding of scalar field precision and resolution. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):603–613, 2021.

[26] M. Hopf, M. Luttenberger, and T. Ertl. Hierarchical splatting of scattered 4D data. *IEEE Computer Graphics and Applications*, 24(4):64–72, 2004.

[27] M. Hosseini and C. Timmerer. Dynamic adaptive point cloud streaming. In *Packet Video Workshop*, PV '18, pages 25–30, 2018.

[28] E. Hubo, T. Mertens, T. Haber, and P. Bekaert. The quantized kd-tree: efficient ray tracing of compressed point clouds. In *IEEE Symposium on Interactive Ray Tracing*, RT '06, pages 105–113, 2006.

[29] M. Isenburg. LASzip: lossless compression of LiDAR data. *Photogrammetric Engineering & Remote Sensing*, 79(2), 2013.

[30] S. Jin, P. Grosset, C. M. Biwer, J. Pulido, J. Tian, D. Tao, and J. P. Ahrens. Understanding GPU-based lossy compression for extreme-scale cosmological simulations. In *IEEE International Parallel & Distributed Processing Symposium*, IPDPS '20, pages 105–115, 2020.

[31] K. Kadau, T. C. Germann, and P. S. Lomdahl. Large-scale molecular-dynamics simulation of 19 billion particles. *International Journal of Modern Physics C*, 15(01):193–201, 2004.

[32] J. E. S. Khalil, A. Munteanu, L. Denis, P. Lambert, and R. V. d. Walle. Scalable feature-preserving irregular mesh coding. *Computer Graphics Forum*, 36(6):275–290, 2017.

[33] T.-J. Kim, B. Moon, D. Kim, and S.-E. Yoon. RACBVHs: random-accessible compressed bounding volume hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):273–286, 2010.

[34] D. King and J. Rossignac. Optimal bit allocation in compressed 3D models. *Computational Geometry*, 14(1):91–118, 1999.

[35] M. Krivokuća, P. A. Chou, and M. Koroteev. A volumetric approach to point cloud compression – Part II: Geometry compression. *IEEE Transactions on Image Processing*, 29:2217–2229, 2020.

[36] M. Krivokuća, P. A. Chou, and P. Savill. 8i voxelized surface light field (8ivslf) dataset. *ISO/IEC JTC1/SC29 WG11 (MPEG) input document m42914*, pages 61–70, 2018.

[37] S. Lasserre, D. Flynn, and S. Qu. Using neighbouring nodes for the compression of octrees representing the geometry of point clouds. In *ACM Multimedia Systems Conference*, MMSys '19, pages 145–153, 2019.

[38] M. Le Muzic, L. Autin, J. Parulek, and I. Viola. cellVIEW: a tool for illustrative and multi-scale rendering of large biomolecular datasets. VCBM '15, pages 61–70, 2015.

[39] H. Lee, M. Desbrun, and P. Schröder. Progressive encoding of complex isosurfaces. *ACM Transactions on Graphics*, 22(3):471–476, 2003.

[40] H. Lee, G. Lavoué, and F. Dupont. Rate-distortion optimization for progressive compression of 3D mesh with color attributes. *The Visual Computer*, 28:137–153, 2012.

[41] P. Lindstrom. Out-of-core construction and visualization of multiresolution surfaces. In *Symposium on Interactive 3D Graphics and Games*, I3D '03, pages 93–102, 2003.

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

[43] H. Liu, H. Yuan, Q. Liu, J. Hou, and J. Liu. A comprehensive study

and comparison of core technologies for MPEG 3D point cloud compression. *IEEE Transactions on Broadcasting*, 66(3):701–717, 2020.

[44] A. Maglo, C. Courbet, P. Alliez, and C. Hudelot. Progressive compression of manifold polygon meshes. *Computers & Graphics*, 36(5):349–359, 2012.

[45] O. Martinez Rubi, S. Verhoeven, M. van Meersbergen, M. Schütz, P. Oosterom, R. Goncalves, and T. Tijssen. Taming the beast: Free and open-source massive point cloud web visualization. In *Capturing Reality Forum*, 2015.

[46] M. McGuire. Computer graphics archive. `https://casual-effects.com/data`. Accessed: 2021-03-31.

[47] R. Mekuria, K. Blom, and P. Cesar. Design, implementation, and evaluation of a point cloud codec for tele-immersive video. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(4):828–842, 2017.

[48] Q. Meng, A. Humphrey, and M. Berzins. The UINTAH framework: a unified heterogeneous task scheduling and runtime system. In *SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 2441–2448, 2012.

[49] A. Metere, S. Sarman, T. Oppelstrup, and M. Dzugutov. Formation of a columnar liquid crystal in a simple one-component system of particles. *Soft Matter*, 11(23):4606–4613, 2015.

[50] S. Park and S. Lee. Multiscale representation and compression of 3D point data. *IEEE Transactions on Multimedia*, 11(1):177–183, 2009.

[51] V. Pascucci and R. J. Frank. Global static indexing for real-time exploration of very large regular grids. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '01, pages 45–45, 2001.

[52] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey. BD-CATS: big data clustering at trillion particle scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 1–12, 2015.

[53] M. Pauly, M. Gross, and L. P. Kobbelt. Efficient simplification of point-sampled surfaces. In *IEEE Visualization*, VIS '02, pages 163–170, 2002.

[54] J. Peng, Y. Huang, C.-C. J. Kuo, I. Eckstein, and M. Gopi. Feature oriented progressive lossless mesh coding. *Computer Graphics Forum*, 29:2029–2038, 2010.

[55] J. Peng and C. C. J. Kuo. Octree-based progressive geometry encoder. In *Internet Multimedia Management Systems IV*, volume 5242, pages 301–311, 2003.

[56] J. Peng and C.-C. J. Kuo. Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition. In *ACM International Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '05, pages 609–616, 2005.

[57] J. Peng and C.-J. Kuo. Progressive geometry encoder using octree-based space partitioning. In *IEEE International Conference on Multimedia and Expo*, ICME '04, pages 1–4, 2004.

[58] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath. Large-scale parallel visualization of particle-based simulations using point sprites and level-of-detail. In *Eurographics Symposium on Parallel Graphics and Visualization*, PGV '15, page 1–10, 2015.

[59] S. Rusinkiewicz and M. Levoy. QSplat: a multiresolution point rendering system for large meshes. In *ACM International Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 343–352, 2000.

[60] K. Schatz, C. Müller, M. Krone, J. Schneider, G. Reina, and T. Ertl. Interactive visual exploration of a trillion particles. In *IEEE Symposium on Large Data Analysis and Visualization*, LDAV '16, pages 56–64, 2016.

[61] R. Schnabel and R. Klein. Octree-based point-cloud compression. In *Eurographics Conference on Point-Based Graphics*, SPBG'06, pages 111–121, 2006.

[62] R. Schnabel, S. Möser, and R. Klein. A parallelly decodeable compression scheme for efficient point-cloud rendering. SPBG '07, pages 214–226, 2007.

[63] S. Schwarz, M. Preda, V. Baroncini, M. Budagavi, P. Cesar, P. A. Chou, R. A. Cohen, M. Krivokuća, S. Lasserre, Z. Li, J. Llach, K. Mammou, R. Mekuria, O. Nakagami, E. Siahaan, A. Tabatabai, A. M. Tourapis, and V. Zakharchenko. Emerging MPEG standards for point cloud compression. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(1):133–148, 2019.

[64] M. Schütz, S. Ohrhallinger, and M. Wimmer. Fast out-of-core octree generation for massive point clouds. *Computer Graphics Forum*, 39(7):155–167, 2020.

[65] J. M. Singh and P. J. Narayanan. Progressive decomposition of point clouds without local planes. In *Indian Conference on Computer Vision, Graphics and Image Processing*, ICVGIP'06, pages 364–375, 2006.

[66] S. W. Skillman, M. S. Warren, M. J. Turk, R. H. Wechsler, D. E. Holz, and P. M. Sutter. Dark sky simulations: Early data release. *arXiv e-prints*, page arXiv:1407.2600, 2014.

[67] S. Slattery, C. Junghans, D. L-G, rhalver, G. Chen, S. Reeve, ascheinb, C. Smith, and R. Bird. ECP-copa/Cabana: Cabana version 0.2.0, 2019.

[68] W. Sweldens. Wavelets and the lifting scheme: A 5 minute tour. *Journal of Applied Mathematics and Mechanics*, 76:41–44, 1996.

[69] D. Tao, S. Di, Z. Chen, and F. Cappello. In-depth exploration of single-snapshot lossy compression techniques for N-body simulations. In *IEEE International Conference on Big Data*, pages 486–493, 2017.

[70] D. Tao, S. Di, Z. Chen, and F. Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '17, pages 1129–1139, 2017.

[71] W. Usher, X. Huang, S. Petruzza, S. Kumar, S. R. Slattery, S. T. Reeve, F. Wang, C. R. Johnson, and V. Pascucci. Adaptive spatially aware I/O for multiresolution particle data layouts. In *IEEE International Parallel and Distributed Processing Symposium*, IPDPS '21, pages 547–556, 2021.

[72] S. Valette, R. Chaine, and R. Prost. Progressive lossless mesh compression via incremental parametric refinement. *Computer Graphics Forum*, 28(5):1301–1310, 2009.

[73] S. Valette and R. Prost. Wavelet-based progressive compression scheme for triangle meshes: wavemesh. *IEEE Transactions on Visualization and Computer Graphics*, 10(2), 2004.

[74] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Gunther, and P. Navratil. OSPRay - a CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):931–940, 2017.

[75] I. Wald, A. Knoll, G. P. Johnson, W. Usher, V. Pascucci, and M. E. Papka. CPU ray tracing large particle data with balanced P-k-d trees. In *IEEE Visualization*, VIS '15, pages 57–64, 2015.

[76] I. Wald and H.-P. Seidel. Interactive ray tracing of point-based models. In *Eurographics Symposium on Point-Based Graphics*, SPBG'05, pages 9–16, 2005.

[77] M. Waschbüsch, M. Gross, F. Eberhard, E. Lamboray, and S. Würmlin. Progressive compression of point-sampled models. In *Eurographics Symposium on Point-Based Graphics*, SPBG'04, pages 95–103, 2004.

[78] J. Woodring, J. Ahrens, J. Figg, J. Wendelberger, S. Habib, and K. Heitmann. In-situ sampling of a large-scale particle simulation for interactive visualization and analysis. In *Eurographics Conference on Visualization*, EuroVis'11, pages 1151–1160, 2011.

[79] D. Z. Zhang, Q. Zou, W. B. VanderHeyden, and X. Ma. Material point method applied to multiphase flows. *Journal of Computational Physics*, 227(6):3159–3173, 2008.