Progressive Tree-Based Compression of Large-Scale Particle Data

Duong Hoang, Harsh Bhatia, Peter Lindstrom, and Valerio Pascucci

Abstract—Scientific simulations and observations using particles have been creating large datasets that require effective and efficient data reduction to store, transfer, and analyze. However, current approaches either compress only small data well while being inefficient for large data, or handle large data but with insufficient compression. Toward effective and scalable compression/decompression of particle positions, we introduce new kinds of particle hierarchies and corresponding traversal orders that quickly reduce reconstruction error while being fast and low in memory footprint. Our solution to compression of large-scale particle data is a flexible block-based hierarchy that supports progressive, random-access, and error-driven decoding, where error estimation heuristics can be supplied by the user. For low-level node encoding, we introduce new schemes that effectively compress both uniform and densely structured particle distributions. Our proposed methods thus target all three phases of a tree-based particle compression pipeline, namely tree construction, tree traversal, and node encoding. The improved efficacy and flexibility of these methods over existing compressors are demonstrated through extensive experimentation, using a wide range of scientific particle datasets.

Index Terms—particle datasets, compression (coding), data compaction and compression, hierarchical, progressive decompression, coarse approximation, tree traversal, multiresolution, visualization

1 Introduction

As a common discrete representation beside grids, particles — moving points in space that carry attributes — are frequently used in scientific applications, including molecular dynamics [1], [2], [3], fluid dynamics [4], [5], [6], computational cosmology [7], [8], [9], imaging of objects and environments [10], [11], [12], and plasma physics [13]. With rapid advances in computational capabilities, simulations and equipment can generate datasets with trillions of particles [8], [13], [14], posing serious challenges to studying such datasets for scientific insights. Compression is a promising solution to the problem of ever-expanding data. However, no widely accepted compressors for particle data currently exist, and attempts to adapt grid-based compressors for particles [15], [16] have seen limited success. Outside of HPC, techniques designed to compress point clouds representing scans of objects [17], [18], [19] focus largely on improving compression ratios at the expense of scalability in performance, making them unsuitable for large datasets. On the other hand, multiresolution rendering systems [7], [8], [20], [21], [22] can handle large data but do not aim for effective compression.

Toward bridging the gap between high compression ratios and low-memory-footprint compression, we introduce novel methods for hierarchy construction, traversal, and encoding that improve on the state-of-the-art tree-based compression methods. We introduce novel tree-based particle compression methods that enable high-quality progressive reconstructions without requiring excessive computational or memory costs. We focus on compressing particle *positions*, since they are needed in almost all applications and, in many applications, are the only attributes needed. Particle positions in scientific applications are difficult to compress losslessly, since they are often specified to such precision that many lower order bits are essentially random. Nevertheless, valuable trade-offs can be made in the space of lossy and progressive (de)compression, in which a decompressor produces approximations that can be progressively refined by decoding more bits that are streamed from the disk or over the network. Progressive decompression allows the user to immediately work with data approximations that improve over time without having to wait for the full data to load or decompress, which can greatly enhance the user experience and accelerate the rate of at which insights are obtained. A progressive decoder can also adapt to the computational resources and time available since decompression can stop as soon as a certain time or data size threshold is reached.

In a progressive setting, 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. At the center of our contributions is a node splitting scheme called *odd-even split*, which we utilize to construct novel hierarchies that can be traversed with asymptotically constant memory footprints to produce high-quality progressive approximations.

- *Duong Hoang and Valerio Pascucci are with the Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, 84112.*
- *Harsh Bhatia and Peter Lindstrom are with Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, 94550.*

Contributions. Specifically, we propose:

- A new mechanism to partition space, the *odd-even split* (subsection 4.1), 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.
- A particular way of combining odd-even and k-d splits to create *hybrid trees* (subsection 4.2) that allows a low-memory-footprint DT to also have the power of BT (high-quality reconstruction), while being conducive to compression.
- An *adaptive traversal* scheme (AT) (subsection 5.1) that allows dynamic guiding of tree refinement, with respect to a given error metric; we propose two such metrics by heuristics.
- *Block-hybrid trees* (subsection 4.3), which combine the strengths of both k-d trees and hybrid trees, to be traversed with *block-adaptive traversal* (BAT, subsection 5.2), for improved memory-quality trade-off and error-guided, progressive refinement with random access.
- A *binomial coding* scheme (subsection 6.1) that improves the compression of uniformly distributed particles by modeling the distribution of child node values using the Binomial distribution.
- An *odd-even context coding* scheme (subsection 6.2) that improves the compression of dense surface data by leveraging the similarity between the two subtrees under an odd-even split.

A preliminary discussion on tree construction and traversal was presented in our previous work [23]. Here, we further analyze and expand upon those ideas by combining them with novel node encoding schemes. Together with tree construction and traversal, these coding schemes complete the tree-based particle compression pipeline (Fig. 1). Our contributions are flexible – they can be utilized either in conjunction with each other or independently, where suitable with existing frameworks. We discuss and compare many such cases through experimentation on a wide range of particle datasets. Finally, note that our work focuses purely on the tasks of particle encoding and decoding, which serve as foundations for followed-up tasks such as rendering. The full source code to our implementation is available online [24].

2 Related Work

In this section we give an overview of the literature on particle (point cloud) data management and compression.

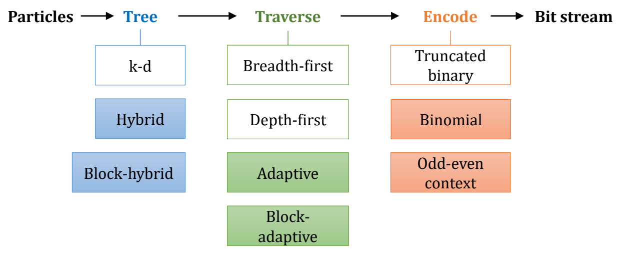


Fig. 1: Our contributions (in color-filled boxes) cover all three main stages of the generic tree-based particle compression pipeline.

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 [25], quadtrees [26], octrees [17], [19], [27], [28], [29], [30], [31], [32], [33], [34], [35], k-d trees [36], [37], and bounding-volume hierarchies [20]. An octree where each node stores the occupancy of its children is by far the most common approach. 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 [38], [39]. 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 [8], [40], [41].

Level-of-detail. Although a tree naturally provides a progressive coarse-to-fine structure, from which representative particles can be decoded and viewed [7], [20], some techniques generate levels of detail through subsampling [21], [22], [25], [30], [42], [43], which requires no data duplication at coarse levels, and is often faster to compute. Random subsampling [21], [25], [42] 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 [44], [45], [46]. When a mesh is not readily available, connectivity can be introduced by building a graph [47], local graphs [48], [49], or a resampled signed distance field [50] 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 [30], [38], [51], [52], [53], [54], [55]. More data-adaptive hierarchies reorder child nodes based on their predicted occupancy [32], or make planes of k-d divisions adaptive to local variations [37]. The trade-off between quantization (imprecise particles) error and discretization (low particle count) error has been studied both in theory [56] and practice [57], [58] 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 [59].

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 [60]. 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 [33], [61]. 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 [8], [21]. Multilevel hierarchies that treat subtrees as blocks are also not uncommon [7], [62], [63], [64], 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, not one at a time. This approach provides excellent computational gains because thanks to DT's low memory footprint.

Modeling for compression. For effective compression, techniques often assume some model for the particle data. The model can be prescribed, using *e.g.,* local planes [31], [34], [49], [53], [65], [66], [67], higher order surfaces [26], [68], self-similarity of patches [69], [70], grid-based or graph-based transforms [71], [72], [73], or learned from training data [61], [74], [75], [76], [77]. The model can also be statistical [18], [19], [28], which often means using a frequency histogram to drive an arithmetic coder [78]. It is also common to sort particles to introduce coherency, either with a graph-based traversal [79], [80] or by directly using particle coordinates [15], [16], [81], [82]. Our *odd-even context coding* assumes a statistical model but is unique in that it relies on similarity between subsampled versions of the same point set, which is an idea not previously explored.

3 Background

Here we discuss the method of Devillers and Gandoin [36] (DG), which serves as a base upon which our technical contributions are built. The DG k-d-tree-based coder (implemented in Google's Draco [83]) has competitive compression ratios while being very fast and general, partly due to the coding scheme being nonstatistical (*i.e.,* it does not rely on knowing the distribution of the particles). 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 $0 \le n_1 \le n$). As n decreases toward the leaf level, the number of bits needed for encoding each node gets smaller, resulting in compression. The tree can be implicitly built, traversed, and encoded at the same time, by having the encoder partition an array of particles inplace, 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.

Fig. 2 gives an example for the DG coder. In their paper, the authors give a theoretical analysis on the number of bits required to separate the particles. Assuming the tree

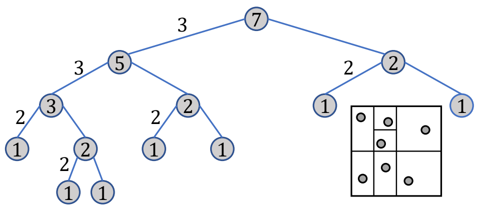


Fig. 2: A k-d tree built for 7 particles in 2D (bottom right). For simplicity, the subdivision stops when the particles are all separated. Each node contains the number of particles in its bounding box. Numbers on the edges specify the number of bits required to encode the corresponding left children nodes (right children are inferred). The numbers written to the bit stream are (in BT order): $7, 5, 3, 1, 1, 1, 1$, using a total of 14 bits.

is balanced and every split divides the number of particles in half, on tree depth k, the total number of bits needed is approximately $2^k \log_2(\frac{N}{2^k}+1)$, with N being the total number of particles. The total number of bits needed to separate the particles is therefore $\sum_{k=0}^{\log_2 N-1} 2^k \log_2(\frac{N}{2^k}+1) \le 2.4N$. Using this result, the paper also gives a lower bound on the number of bits saved using the k-d tree coder compared to verbatim encoding of the particle positions, which is $N \log_2 N$. Since $O(N \log_2 N)$ is also the number of bits needed to encode the relative ordering of the particles (of which there are $N!$), the k-d tree compresses by discarding the original order of particles, on top of compression achieved by quickly separating particles from empty space.

4 Tree construction

Most tree-based compression techniques work by encoding (and decoding) nodes that implicitly give quantized particle positions. A general template for a tree-based decoder is given in Algorithm 1 in the Appendix. Encoding the number of particles may at first seem wasteful: it has been noted [29], [84] that at coarse levels, occupancy-based octrees are better than the k-d tree used by DG [36], since encoding the number of particles in child nodes often requires several bits compared to at most one bit for occupancy. However, occupancy encoding requires both children of a node to be coded instead of just the left child. Toward the leaf level, past the particle separation stage, encoding n_1 requires a single bit, whereas encoding the occupancy for both children requires 2 bits. Since there are approximately as many leaf nodes as there are internal nodes, encoding occupancy for both children ends up not providing a saving overall compared to encoding the number of particles in only the left child for each internal node.

Furthermore, as seen in subsection 5.1, encoding the number of particles also allows us to perform adaptive tree traversal to minimize reconstruction error, which is estimated using the number of particles and the spatial extent of a node. Finally, by knowing the number of particles, we can employ a grid-based approach and switch to encoding the number of empty grid cells when the grid has more particles than empty cells, which significantly saves coding cost (as discussed in subsection 4.1).

4.1 Odd-Even Splits and Odd-Even Trees

When decoding is run to completion, all tree nodes are visited, in an order that depends on the traversal strategy.

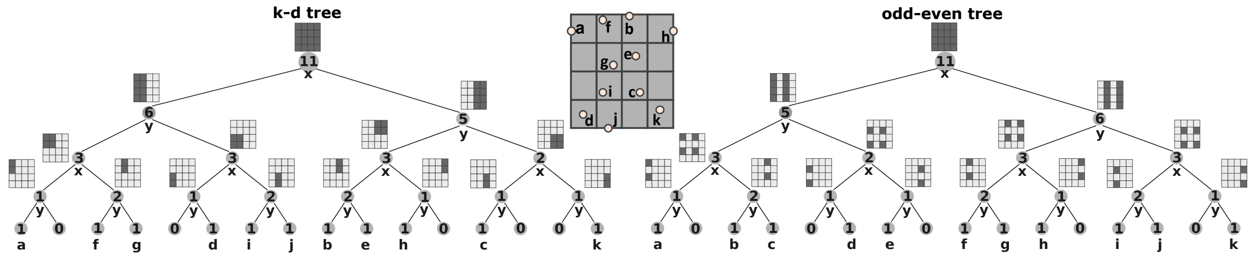


Fig. 3: An example with 11 particles (a to k) on a 4^2 grid, from which we build a k-d tree (left) and an odd-even tree (right). Our odd-even splits partition space by interleaving odd-indexed and even-indexed grid cells at each tree level. For simplicity, the trees are built only until every particle is located to its own cell. The numbers encoded in the bit stream (using DT) are $\{11,6,3,1,1(a),1(f),1,0,1(i),3,2,1(b),1(h),1,1(c),0\}$ for the k-d tree, and $\{11,5,3,1,1(a),1(b),1,0,1(e),3,2,1(f),1(h),2,1(i),0\}$ for the odd-even tree.

In practice, however, it is often desirable to traverse and decode large trees only partially to save I/O bandwidth, memory, and decoding time, reconstructing particles whose positions only approximate the original particles' positions. In this context, the shape of the tree and the traversal order can profoundly affect the accuracy of the approximation.

Trade-offs between depth-first and breadth-first traversals. On a traditional k-d tree constructed with the typical *k-d split*, a node's bounding box $\mathbf{B}$ is split along a certain dimension (one of x, y, z in 3D) to give children boxes $\mathbf{B}_1$ and $\mathbf{B}_2$. BT visits $\mathbf{B}_2$ after $\mathbf{B}_1$ on each tree level, whereas DT only visits $\mathbf{B}_2$ after $\mathbf{B}_1$ has been fully visited. Therefore, when stopped midway, BT often gives coarse representations of the whole space whereas DT reconstructs a spatial region perfectly but completely misses the rest. In most cases, the former behavior is preferred.

DT however is significantly less resource intensive, since it requires only a small stack whose size is at most the height of the tree ($O(\log_2 n)$), whereas BT requires a queue large enough to keep all nodes at the current depth, which can grow as large as the total number of particles ($O(n)$). 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). Here, cost mostly means memory footprint, but a high memory footprint often also translates to lower cache utilization and accordingly lower speed.

Odd-even splits. To alleviate the main drawback of DT while retaining its main benefit, we introduce the notion of an *odd-even split*, which spatially "interleaves" the children boxes $\mathbf{B}_1$ and $\mathbf{B}_2$ by having each contain many disjoint slices instead of being a whole contiguous region. This scheme is inspired by the hierarchical indexing scheme [85] and the lazy wavelet transform [86], multiresolution techniques invented for data sampled on regular grids.

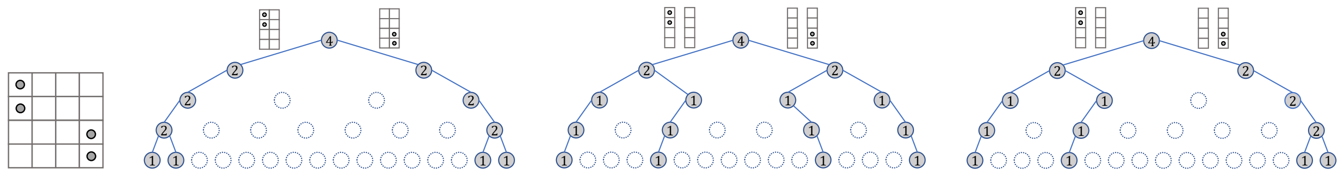
We first impose (but do not build) a regular grid on top of the particles such that each cell contains at most k particle. One way to build such a grid is to recursively subdivide the particles' bounding box into equal halves along the longest dimension, stopping when the target k is met. For the odd-even splitting scheme to work best, k should ideally be 1. However, when particle coordinates are given in floating point, $k = 1$ may produce a grid that is too large if any two particles have almost exactly the same coordinates. In this paper we use $k = 1$ in all experiments, but in general k is a parameter that can be set by the user. In addition, to avoid potential rounding errors when multiplying and dividing floating point numbers, we work with quantized particle positions in deciding which grid cell a particle belongs to, but note that the original particles' positions can still be encoded losslessly if needed.

After the full grid is defined, we associate the root of the tree with the full grid, and associate every other node with a different subgrid $\mathbf{G}$ and the particles 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), along three fixed axes. An odd-even split decomposes a node $(\mathbf{G}, n)$ into $(\mathbf{G}_e, n_e)$ and $(\mathbf{G}_o, n_o)$, such that $(\mathbf{G}_e, n_e)$ contains the even-indexed cells in $\mathbf{G}$ (along the dimension of splitting) and the n_e particles occupying those cells, while $(\mathbf{G}_o, n_o)$ contains the rest of the (odd-indexed) cells and particles.

Odd-even trees. A tree constructed exclusively from odd-even splits is called an *odd-even tree*, illustrated in Fig. 3 in contrast to a k-d tree. The idea of the odd-even split is that either the odd or the even child node represents a coarse approximation of the particle set associated with the parent node, so that a DT can never miss an entire region as with k-d splits. Odd-even trees carry this idea to an extreme where every node is split in the odd-even scheme, and therefore DT on an odd-even tree provides the best coarse-to-fine refinement of the full data with respect to the number of particles reconstructed, but not in terms of coding cost (or compression ratio) which will be discussed later.

Odd-even subsampling. Picking either the odd or the even subgrid to traverse can be viewed as a subsampling method. It may at first seem that random subsampling (*e.g.*, as done in [87]) achieves the same effects as odd-even subsampling while being simpler. However, unlike random sampling, odd-even sampling produce subgrids ($\mathbf{G}_o$ and $\mathbf{G}_e$) that are half the size of the parent grid $\mathbf{G}$ in number of cells, which is important for locating the particles using fewer bits. Unfortunately, an odd-even tree is still not conducive to compression. This is due to the fact that for most datasets, particles do not scatter randomly in space but form clusters and structures that can be well separated from empty cells. With odd-even splits, the empty cells are "distributed" into the odd and even subtrees, effectively increasing the number of tree nodes to be coded. Instead, k-d splits could be used to quickly cull away entire empty subtrees (as can be seen in Fig. 4).

Coding costs. For a more quantitative analysis, we calculate the number of bits required to locate, using a k-d tree, n particles in a grid $\mathbf{G}$ with G cells, of which n contain particles and $G - n$ are empty. Denote the answer



(a) 4 particles in 2D (b) The particles encoded as a k-d tree. The total number of nodes is 11. (c) The particles encoded as an odd-even tree. The total number of nodes is 15. (d) The particles encoded as a hybrid tree. The total number of nodes is 13.

Fig. 4: An example that demonstrates how odd-even trees (c) can compress not as well as k-d trees (b), and how hybrid trees (d) alleviate this problem. The odd-even tree uses odd-even splits exclusively, whereas the hybrid tree only uses odd-even splits on the path connecting the root to the left-most leaf node. Odd-even trees do not compress well since they create too many nodes to fully locate the particles among all cells.

as $T(n, G)$. In the best case, a k-d split will put most particles in one child and empty space in the other, leading to $T(n, G) = (\log_2 n) + T(n, G/2)$, *i.e.*, n stays the same but G is reduced by half. After $i = \log_2 (G/n)$ such iterations, $G/2^i$ and n are approximately equal, *i.e.*, $G/2^i < 2n$. The k-d tree now requires $\approx 2.4n$ bits to separate the n particles (see section 3), and an additional n bits to finally locate the particles (assuming at the leaf level, each particle needs to be separated from one empty cell). In contrast, an odd-even split implies a different recurrence relation: $T'(n, G) = (\log_2 n) + 2T'(n/2, G/2)$ (*i.e.*, both n and G are halved but two substrees are created instead of one). After the particles are separated from one another (after $\approx 2.4n$ bits), each particle needs to be further located among G/n cells, for a cost of $n \log_2 (G/n)$ additional bits. Therefore, the difference between $T(n, G)$ and $T'(n, G)$ is that between $n + \log_2 n \log_2 (G/n)$ (for the k-d tree) and $n \log_2 (G/n)$ (for the odd-even tree). The two are similar if G is close to n (the grid is dense in particles), but in most cases, G is significantly larger than n, making the odd-even tree worse. In experiments, we have seen odd-even trees that are almost twice as large as a k-d tree for the same input. We later discuss a solution in subsection 4.2.

Encoding dense particle distributions. Besides facilitating the odd-even splits, an underlying grid allows us to effectively encode sparse as well as dense particle sets (relative to the size of the grid). This situation happens when the majority of grid cells contain a particle, instead of being empty. 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.*, $G/2 - n_1$, and thus more quickly bound the values to be encoded further down the tree. Note that n_1 is always at most $G/2$ since there can only be at most one particle per grid cell. In the extreme case where every cell contains a particle, our method simply stops after encoding the number of particles

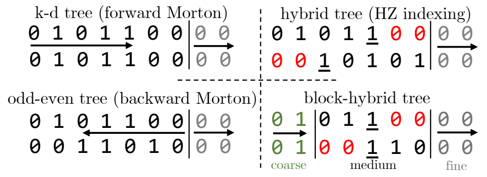


Fig. 5: A tree implies an ordering of particles following their transformed Morton codes. Input Morton bits are shown the top and arrows indicate directions of the output bits at the bottom. K-d trees and odd-even trees use forward and backward Morton codes. Hybrid trees use HZ indexing [85]. Block-hybrid trees use HZ indexing for the medium portion. In-cell refinement bits are shown in gray.

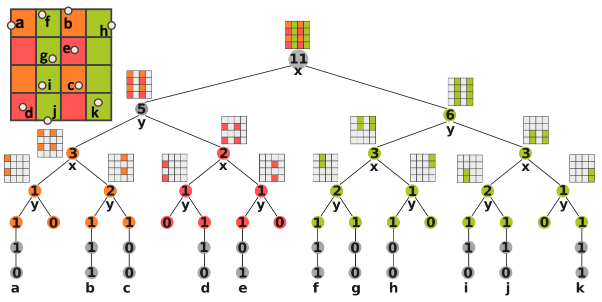
at the root node, since the number of empty cells is now 0, whereas other methods, such as DG [36] or MPEG [17] which encodes node's occupancy, will keep refining this grid until the individual cells.

Tree traversal as particle indexing. To decode particles' positions by traversing a tree is to reconstruct the bits of their quantized integer coordinates, or, equivalently, to index (order) the particles using their coordinate bits. It is well known that a k-d tree sorts the particles using their Morton codes, which interleave particle coordinate bits in x, y, z, with an interleaving pattern that depends on the order of the dimensions along which nodes are split. In other words, a k-d tree reconstructs the interleaved coordinate bits from left to right (MSB to LSB) if traversed using DT, whereas an odd-even tree reconstructs them from right to left (LSB to MSB) (see Fig. 5), since the LSB determines whether a particle is "odd" or "even".

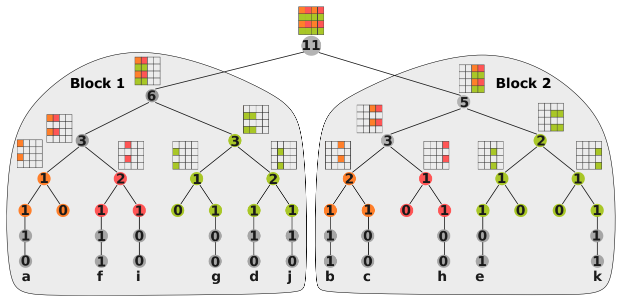
4.2 Hybrid Trees

As seen in 4.1, odd-even trees create too many nodes because every odd-even split distributes both the particles and the empty cells into two children, instead of (mostly) particles in one and empty cells in the other. To reduce this adverse impact on compression while retaining most of their benefits, we need to reduce the use of odd-even splits. 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, we restrict the use of odd-even splits to only left child nodes, with k-d splits used everywhere else. Furthermore, once a k-d split is used, subsequent descendant splits will all be k-d splits. We also use the convention that the left child node is $(\mathbf{G}_e, n_e)$, *i.e.*, it contains the even-indexed cells. The dimension of splitting is the largest dimension of the parent's grid $\mathbf{G}$, as is also the case for all the other trees discussed in this paper. 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.

Resolution levels. From top to bottom, every odd-even split creates a new, coarser *resolution level*, which consists of nodes in the even-indexed subtree. A hybrid tree with L resolution levels contains a sequence of exactly $L - 1$ odd-even splits, at nodes found by traversing down the left child $L-2$ times from the root (see Fig. 6a for an example with $L = 3$). L can be automatically set so that the chain of odd-even splits ends when no particles or cells are left to split. Assuming that left children are always visited first, DT on hybrid trees visits the resolution levels from coarse to fine, producing a



(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

Fig. 6: (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. 3, with additional (conceptual) tree nodes for in-cell refinement bits, shown in gray.

"breadth-first" walk of space similar to BT on k-d trees but with much a smaller memory footprint.

Although hybrid trees are designed with DT in mind, they also support BT (see Fig. 12 for an example), 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. 6a). Our proposed hybrid tree is also only one of the many possible combinations of k-d and odd-even splits, which may be useful for different purposes.

Particle indexing. From the perspective of particle indexing using their interleaved quantized coordinates, hybrid trees correspond to the hierarchical Z (HZ) ordering [85] of particles. An HZ ordering sorts the particles first by their resolution level, then by their index within the level. This is done by swappping the least significant one bit in a particle's Morton code (whose position from the LSB determines the particle's resolution level) and the bits to its left (which constitute the particle's intra-level index). Fig. 5 gives an example of this scheme. The HZ indexing scheme was first proposed by Pascucci and Frank [85] and generalized by Hoang *et al.* [88] for multiresolution decomposition of regular grids. Here, we adapt the scheme to construct a multiresolution particle tree.

Refinement bits. Refinement bits are bits that further locate each particle in its corresponding cell (see Fig. 6, gray nodes at the bottom). All refinement bits at the same tree depth form a *bit plane*. Once each particle is located to its cell, further refinement bit planes recursively half the cell in one dimension at a time, and the bit values indicate which half the particle belongs to. The number of refinement bit planes vary across datasets. For some, there are no refinement bits, *i.e.,* particles are specified with precision low enough that the particles are exactly located just by the grid that separates them. In the other spectrum, scientific simulation data are often dominated by refinement bits due to the particles being specified with relatively high precision compared to their density. For hybrid trees, the refinement bits are stored in depth-first order: particles are completely refined one by one, in the (depth-first) order that they appear in the tree. For example, in Fig. 6a, the refinement bit stream is $10(a)11(b)00(c)10(d)01(e)11(f)00(g)00(h)10(i)10(j)11(k)$.

Coding costs. As in subsection 4.1, let $T(n, G)$ denote the number of bits needed to locate n particles in G grid cells using a k-d tree. For a hybrid tree, the number of bits to code a subtree under node $(\mathbf{G}, n)$ is $(\log_2 n) + T_l(n/2, G/2) + T_r(n/2, G/2)$. The term $T_r(n/2, G/2)$ (for the right subtree) is just $T(n/2, G/2)$ since the right subtree is always a k-d tree. The term $T_l(n/2, G/2)$ (for the left subtree) can again be decomposed into $\log_2(n/2) + T_l(n/4, G/4) + T(n/4, G/4)$. Following the recurrence to the end and ignoring the various $\log_2(n/2^i)$ terms that are insignificant, we see that the cost of encoding the whole hybrid tree is approximately $\sum_i T(n/2^i, G/2^i)$. Since $T(n, G)$ is approximately linear in n (see subsection 4.1), $T(n, G) = 2T(n/2, G/2)$. The sum $\sum_i T(n/2^i, G/2^i)$ therefore is approximately just $T(n, G)$, meaning the coding cost of a hybrid tree is approximately the same as that of a k-d tree for the same input (the difference is of order $O(\log_2 n \log_2(G/n))$ bits which is essentially the number of resolution levels multiplied by the cost to cull the empty cells on each level). This analysis also shows that if a hybrid tree is traversed with DT, so that the resolution levels are visited from coarse to fine, the (partial) coding cost doubles after each resolution level as the number of particles, n, presumably also doubles.

Reconstruction error. To quantify the reconstruction error for each original particle, we find the nearest particle to it among the reconstructed particles. We give an upper bound for the reconstruction error in both cases: BT on a k-d tree (BT-kd) and DT on a hybrid tree (DT-hybrid). Suppose the two particles form two opposite corners of a box of dimensions $d_x \times d_y \times d_z$, we can bound the values of d_x, d_y, d_z using k, understood to be either the number of tree depths not yet traversed (for BT-kd), or the number of resolution levels not yet traversed (for DT-hybrid). For both cases, it is guaranteed that $d_x d_y d_z \leq w_x w_y w_z 2^k$, with w_x, w_y, w_z being the dimensions of each cell at the leaf level. From the analysis in the **Coding costs** paragraph, we know that the total coding cost for the k-d tree is approximately the same as that for the hybrid tree. Moreover, this cost doubles after each tree depth level (for the k-d tree) and after each resolution level (for the hybrid tree). Therefore, BT-kd and DT-hybrid tree have similar coding costs as well as similar reconstruction error bounds. Note that when all particles are located to their respective cells, the reconstruction error is bounded by the dimensions of a cell *i.e.,* $w_x \times w_y \times w_z$, and each refinement bit plane reduces this bound by half.

In terms of reconstruction error, the main difference between the two schemes is that DT-hybrid puts the reconstructed particles exactly where the corresponding original particles are, whereas BT-kd puts them in the middle of the bounding boxes at the traversal front. Depending on the dataset, one choice may be preferred over another (section 7). Because both schemes have approximately the same total coding costs but DT-hybrid partially reconstructs particles to a higher precision, it also tends to generate significantly fewer particles compared to BT-kd when both are stopped midway at the same decoding bit budget.

Memory footprint with DT. We do not explicitly construct the tree in memory, as node values are simply encoded to and decoded from a bit stream, following a certain traversal order. Therefore, only the size of the data structures used for traversal, and not that of the tree itself, counts toward our memory footprint. Let G denote the total number of grid cells and N the total number of particles. Using DT, a hybrid tree can be traversed using a stack whose size is the bounded by the height of the tree, which is $\log_2 G + 1$. For each element in the stack, $\log_2 (N+1)$ bits are needed to keep track of the number of particles. The other information required for traversal, namely a node's associated grid, its resolution level, the dimension of splitting, and the type of split can all be deduced from the path connecting the node to the tree's root, encoded with $\log_2 G + 1$ bits. The encoder (but not decoder) would also need to keep track of the range of particles that each node encompasses, for a total of $\log_2 (G+1)$ additional bits per node. In short, the memory footprint of the encoder is $(\log_2 G + 1)^2 + \log_2 (N+1)$ bits, while that of the decoder is $(\log_2 G + 1)^2$ bits. Even when $N = G = 2^{64} - 1$, both the encoder and the decoder require trivial amounts of memory (less than 600 bytes).

4.3 Block-Hybrid Trees

A problem wit hybrid trees is that each resolution level is still traversed region-by-region, resulting in uneven error distribution (see Fig. 14, a5 for an example). To mitigate this problem, we split the whole tree into multiple blocks (subtrees) and interleave their traversals to reduce error more uniformly. The resulting *block-hybrid tree* contains multiple blocks that can be decoded independently. By adaptively allocating bits across blocks, we can lower the overall reconstruction error, or prioritize certain blocks for the task at hand. Furthermore, blocks can be randomly accessed or decoded in parallel.

To construct a block-hybrid tree, we first use k-d splits to form a *coarse* portion (a k-d subtree at the top), then combine k-d splits with odd-even splits to form a *medium* portion (several hybrid subtrees), and finally use the in-cell refinement bits to form a *fine* portion. Each leaf of the coarse-portion k-d subtree creates a hybrid sub-tree, or *block*. The coarse and medium portions together refine the full grid until at least the cell level (*i.e.,* no leaf node contains more than one particle). The fine portion further locates individual particles within the respective cells. Fig. 6b shows an example of a block-hybrid built for our running example with 11 particles in 2D. From a particle indexing perspective, block-hybrid trees use hierarchical-Z indexing for the middle portion, and forward Morton for the rest (see Fig. 5).

Refinement bits. Since one main goal of block-hybrid trees is to distribute reconstruction error more uniformly in space, we store the in-cell refinement bits verbatim in bit plane order, *i.e.,* in breadth-first instead of depth-first order as done for hybrid trees. In particular, each bit plane contains one refinement bit for each particle in the block, in the order that a medium-phase DT visits the particle. To decode each refinement bit, we need to subdivide a bounding box that encompasses the current particle. To avoid buffering such bounding boxes for later refinement in typical breadth-first manner, we compute them on-the-fly from the current position of each particle and the dimensions of grid cells at the current tree depth. Therefore, no extra memory is needed in addition to an array storing the positions of the decoded particles, which is presumably always present.

Flexible decoding. A block-hybrid tree, once encoded, can be decoded in different ways; in particular, the blocks can be decoded independently and to different extents. To support independent decoding, the compressed bit streams for individual blocks are stored separately (see Fig. 7). Decoding of higher resolution particles can also be skipped in favor of more refinement bits for lower resolution ones. Such a strategy, which trades resolution for precision, may be desired if the number of output particles needs to be limited due to resource constraints. Since blocks are encoded in independent bit streams, a decoder can freely jump to any block to continue decoding/traversal if needed. The user can also supply a scoring function to rank blocks during traversal. In subsection 5.2, we introduce one such function, which interleaves traversal of blocks to lower the average reconstruction error. Other criteria are possible, for example, during rendering, certain blocks may be prioritized if they are closer to the camera, or since they are known to contain features of interest.

5 Tree Traversal

To achieve better progressive reconstruction than BT and DT, 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 introduce two such adaptive orders: *adaptive traversal* (AT) for k-d/hybrid trees and *block-adaptive traversal* (BAT) for block-hybrid trees.

5.1 Adaptive Traversal

For k-d trees, we generalize the container C in Algorithm 1 from either a stack or a queue to a priority queue, which

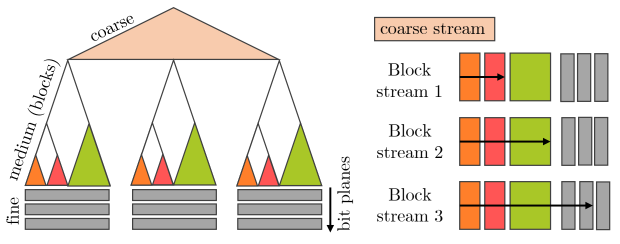


Fig. 7: Left: a block-hybrid tree with subtrees colored by resolution level. Right: the blocks' bit streams are stored separately, so that blocks can be decoded independently, indicated by the arrows. In a block, medium-portion bits are in depth-first order (by resolution level), whereas in-cell refinement bits (gray) are in breadth-first order (by bit plane). At decoding time, any of the resolution levels (colored triangles) can be skipped in favor of more refinement bits for the coarser resolution levels.

allows us to perform rate-distortion optimization during traversal, *i.e.,* prioritizing nodes that are more important with respect to some error metric and coding cost. Concretely, the importance score of a given node $(\mathbf{B}, n)$ is

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

where d is the length of $\mathbf{B}$ along the current axis of splitting. The denominator captures the cost of decoding the current node, while 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, spatially larger and denser nodes are prioritized so that reconstruction 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.

Alternative score function. 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 7). Regardless, this score is still a heuristic and thus is not guaranteed to work for all datasets. We also demonstrate modifications (see Fig. 12) 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.

5.2 Block-Adaptive Traversal

Although AT improves on BT in reconstruction quality, it has a similarly high memory footprint in practice (see subsection 7.3). Furthermore, AT works with individual nodes and not blocks, so it cannot be used as is to efficiently traverse a block-hybrid tree. Here, we generalize AT to *block-adaptive traversal* (BAT), which is also data-adaptive but works with entire blocks of nodes, and has asymptotically constant memory footprint similarly to that of DT.

With BAT, the coarse portion of a block-hybrid tree is traversed with either BT or AT. Traversal of the medium portion only begins after traversal of the coarse portion completes. The medium portion is traversed in iterations in a data-dependent round-robin manner. Each iteration consists of two steps: first, we pick a block to traverse using a priority queue that ranks blocks based on some criterion, then, we traverse the chosen block using DT. The block at the top of the priority queue is traversed for either a certain number of decoded bytes or a certain number of particles, then its priority is updated in the queue, and the process repeats with the next iteration.

Heuristic for ranking blocks. The ranking of blocks is handled by a user-supplied scoring function; here we propose one. Between two partially decoded blocks, we always prioritize the one at a coarser resolution level. If the 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 resolution level l, and n_l^* is the number of those already visited by the per-block DT. The idea behind this heuristic is to distribute reconstruction error across blocks as uniformly as possible, so that the average error is reduced.

Reconstruction error. By construction, nodes on the same resolution level are associated with subgrids with the same internal spacing (*i.e.,* spatial distances in x, y, z between neighboring cells). This spacing gives an upper bound on the reconstruction error, since particles that fall in between neighboring cells in the current subgrid have not been reconstructed (they belong to finer resolution levels). For example, when all particles in the even subtree under an odd-even split have been reconstructed, but particles in the odd subtree have not (because the odd subtree belongs to the next finer resolution level), the spacing between cells of the grid corresponding to the even subtree is an error upper bound. Therefore, forcing the blocks to be refined to the same resolution level effectively forces approximately the same upper 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.

Memory footprint. Given a tree of height H, the memory footprint of BAT is controlled by H_c, the height of the coarse k-d subtree. Since there are at most 2^{H_c-1} blocks and each block contains a stack of size at most $H - H_c$, the total number of elements in the different containers is bound by 2^{H_c-1} (queue) $+2^{H_c-1}(H-H_c)$ (stacks) + 2^{H_c-1} (priority queue). In contrast, the size of the queue for BT, if used exclusively for the whole hierarchy, is bounded above by 2^{H-1}, which is often several orders of magnitude larger than 2^{H_c-1}, since a typical H_c is only half of H. In practice, the H_c chosen should be large enough so that the error is more uniformly distributed and that random access to the blocks is more fine-grained, but also small enough to not turn BAT into BT and also to not create too many blocks.

6 Encoding Node Values

During decompression, at a node $(\mathbf{G}, n)$, a decoder needs to decode n_1 with the knowledge of n and the fact that $0 \le n_1 \le n$. The state-of-the-art method [36] uses arithmetic coding [78] or truncated binary coding [89], [90], assuming that n_1 is uniformly distributed in $\{0, \dots, n\}$. However, this assumption is often not true in practice, and thus better encoding methods are possible. We present two such methods here that better predict n_1, namely a nonstatistical *binomial coding* scheme and a statistical *odd-even context coding* scheme, targeting two extreme particle distributions: uniform and highly structured.

6.1 Binomial Coding

For data that exhibit approximately uniformly spatial distribution of particles, n_1 is not uniformly distributed in $\{0, \dots, n\}$ but is more likely to be close to $\frac{n}{2}$ — a property that we will exploit to improve the encoding. Given a node with n particles, there are 2^n possible configurations (each of the n particles can fall in either of the two child nodes with probability $\frac{1}{2}$), and there are $\binom{n}{n_1}$ ways for the left child node under consideration to contain exactly n_1 particles out of the n particles of the parent. Therefore, n_1 follows the binomial distribution with parameters n and $\frac{1}{2}$ (see Fig. 8), *i.e.*, $P(n_1|n) = \binom{n}{n_1}2^{-n} = B(n, \frac{1}{2})$.

Arithmetic coding for small n. For small values of n, this binomial distribution can be effectively coded using arithmetic coding [78] with a precomputed binomial table. Our arithmetic coder supports integer probabilities whose sum is at most 2^{31}, which means the distribution $B(n, \frac{1}{2})$ is exactly modeled for $n \leq 30$. For every $n \in \{1, \ldots, 30\}$, we precompute a table where the entries are $\binom{n}{n_1}$ for every $n_1 \in \{0, \ldots, n\}$ (we scale $P(n_1|n)$ by 2^n to represent the probabilities with integers). We then compute a prefix sum on each table to obtain a (scaled) cumulative distribution function (CDF) ready to be used by our arithmetic coder.

Binary-search coding for large n. When $n > 30$, arithmetic coding with exact probabilities fail because our arithmetic coder uses 32-bit values for its internal states, which have insufficient precision to distinguish all possible values of $P(n_1|n)$, since the scaled CDF grows exponentially with n, *i.e.*, $\sum_{n_1=0}^{n}\binom{n}{n_1} = 2^n$. Note that simply using 64-bit internal states would not solve the problem, due to potential integer overflows under multiplications. Instead, we leverage the *de Moivre-Laplace* theorem [91] to approximate the binomial distribution with a Normal distribution for large n, *i.e.*, $B(n,p) \simeq N(np, np(1-p))$, where N is the Normal distribution with mean $\mu = np$ and variance $\sigma^2 = np(1-p)$). When $p \approx \frac{1}{2}$, *i.e.*, assuming an approximately uniform distribution of particles, the theorem states that $P(n_1|n)$ follows $N(\frac{n}{2}, \frac{n}{4})$.

Denoting the CDF of $N(\frac{n}{2}, \frac{n}{4})$ as F, we use a binary search that locates n_1 by halving F in the search range $[a_i, b_i]$ for each iteration i, outputting a bit to indicate which half contains n_1. The point of division can be computed using the inverse of F, namely $F^{-1}(x) = \frac{n}{2} + \sqrt{\frac{n}{2}}\,\mathrm{erf}^{-1}(2x - 1)$, where erf is the error function. Initially, $[a_0, b_0] = [0, n]$, and our search stops when either the value is found (*i.e.*, $n_1 \leq a_i < b_i < n_1 + 1$ for some i) or the range stops converging, indicating that we run out of numerical precision. In the latter case, we assume equal probabilities for all values in $[a_i, b_i]$ and encode $n_1 - a_i$ using truncated binary coding. We use a *mid-short* (or *centered-minimal*) code [90] that assigns shorter codewords for values near the middle of $[0, \ldots, b_i - a_i]$. The pseudocode for our binomial encoder is given by Algorithm 4 in the Appendix.

Code size gain over truncated binary coding. The theoretical gain achievable with binomial coding can be assessed using the entropy of the binomial distribution, *i.e.*, $H \simeq \frac{1}{2}\log_2(2\pi e np(1-p))$ (the full derivation is in Appendix D). For $p = \frac{1}{2}$, we get $H \simeq \frac{1}{2}\log_2(2\pi e\frac{n}{4}) \approx 1 + \frac{1}{2}\log_2 n$. The normalized entropy (dividing H by $\log_2 n$) thus approaches $\frac{1}{2}$ as n tends to infinity and 1 as n tends to 1. In contrast, the normalized entropy of the uniform distribution is always 1, which means that in the best case (when n is very large) binomial coding reduces the code length by half compared to uniform arithmetic coding or truncated binary coding. This gain makes binomial coding attractive for large data. Finally, binomial coding also works well with odd-even splits, because such splits tend to produce approximately equal numbers of particles on the two sides, regardless of the actual particle distribution.

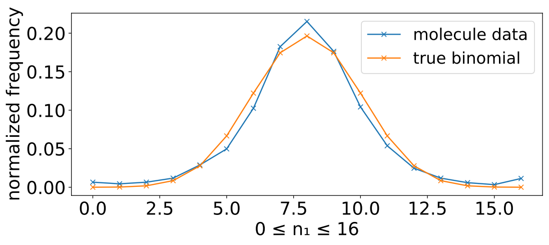


Fig. 8: (Normalized) frequencies of n_1 (number of particles in the left child), given $n = 16$, for both the *molecule* dataset and a true binomial distribution *i.e.*, $B(16, \frac{1}{2})$. The empirical distribution tracks the theoretical distribution well, showing that n_1 is clearly not uniformly distributed.

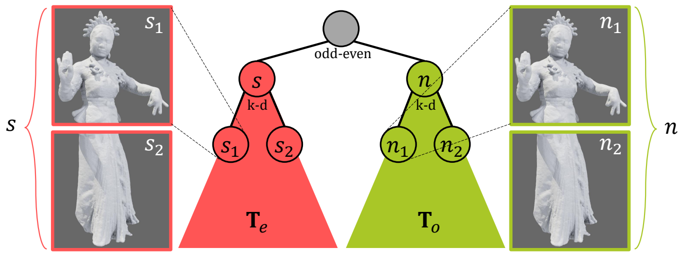


Fig. 9: The left (red) and right (green) subtrees under an odd-even split can have similar particle distributions, and one can be used to predict the other. Here, n_1 can be inferred from n, s, and s_1 (*e.g.*, $n_1 \approx ns_1/s$).

6.2 Odd-Even Context Coding

For datasets where the particles are not approximately uniformly distributed – and thus binomial coding does not apply – we propose a prediction scheme based on DT on a hybrid tree to improve compression. Since the even (left) and odd (right) subtrees under an odd-even split interleave spatially, they can have very similar distributions of particles (Fig. 9). We can therefore leverage their spatial correlations and use one to predict the other. An odd-even split creates an odd and an even subtree, denoted as $\mathbf{T}_o$ and $\bar{\mathbf{T}}_e$, respectively. We use different notations to indicate that $\mathbf{T}_o$ is always a k-d tree, while $\bar{\mathbf{T}}_e$ is almost always a hybrid tree by definition. Using DT, we traverse and code $\bar{\mathbf{T}}_e$ first, and then use it as a reference to predict $\mathbf{T}_o$.

Lock-step traversal. Since $\bar{\mathbf{T}}_e$ and $\mathbf{T}_o$ are different kinds of tree, we first need to transform $\bar{\mathbf{T}}_e$ to a k-d tree $\mathbf{T}_e$. We do so by invoking a k-d tree building routine on the cells to which the particles of $\bar{\mathbf{T}}_e$ have been located. The k-d trees $\mathbf{T}_e$ and $\mathbf{T}_o$ can now be traversed in lockstep using DT to maintain spatial correlations between respective nodes at the traversal front. After $\mathbf{T}_o$ is fully coded, the hybrid subtree combining $\mathbf{T}_e$ and $\mathbf{T}_o$ is converted to a k-d tree to serve as the reference for the next resolution level (Fig. 10). Algorithm 5 in the Appendix gives the full pseudocode for our odd-even context encoder. Note that we never explicitly create and store any of $\bar{\mathbf{T}}_e$, $\mathbf{T}_e$ and $\mathbf{T}_o$ in memory. The conceptual transformation of $\bar{\mathbf{T}}_e$ from an odd-even to a k-d subtree can be performed inplace (by partitioning the array storing the input particles) and inline (computing node values for $\mathbf{T}_e$ on-the-fly as we traverse $\mathbf{T}_o$).

Context coding. During the lockstep DT, the traversal front typically contains two nodes: $(\mathbf{G}_s, s)$ on $\mathbf{T}_e$ and $(\mathbf{G}, n)$ on $\mathbf{T}_o$ (see Fig. 10). If the number of particles in the left child of $(\mathbf{G}_s, s)$ and $(\mathbf{G}, n)$ are s_1 and n_1, respectively, then n, s, and s_1 are known, while n_1 needs to be coded. To predict n_1 using n, s and s_1, we leverage context-based arithmetic coding [92], in which the knowledge of a vector $\mathbf{c}_1$ (the context) helps narrow down the possible values for n_1. We do not use $\mathbf{c}_1 = [n, s, s_1]$ to encode n_1 directly, since any of these numbers can be so large that keeping track of all possible contexts is impractical. Instead, we work with the log values, namely $m = \lfloor \log_2 (n+1) \rfloor, m_1 = \lfloor \log_2 (n_1+1) \rfloor, r = \lfloor \log_2 (s+1) \rfloor$, and $r_1 = \lfloor \log_2 (s_1+1) \rfloor$. The use of log values also make all contexts more reliable, since each context now appears enough times to be statistically significant. However, in place of n_1, we now must encode both m_1 and $m_2 = \lfloor \log_2 (n_2+1) \rfloor$, with n_2 being the number of particles in the right child of the current node (*i.e.,* $n_2 = n - n_1$). The reason is that in general m_2 in general cannot be inferred from m and m_1, except in a few special cases, namely when $m = m_1 = 1, m_2 = 0$, and when $m_1 = 0, m_2 = m$.

Context update. To encode m_1, our context vector $\mathbf{c}_1$ contains more information than just $[m, r, r_1]$. In particular, $\mathbf{c}_1 = [m, r, r_1, r_2, l, h]$, with r_2 being the log of the number of particles in the right child of the reference node (*i.e.,* $r_2 = \lfloor \log_2 (s - s_1 + 1) \rfloor$), l being the current node's resolution level and h being its current tree depth. We use a *context table* H to maintain and update the conditional probabilities $P(m_1|\mathbf{c}_1)$ on the fly for all combinations of m_1 and $\mathbf{c}_1$ encountered during traversal. H is a hashtable, that, when indexed with a key $\mathbf{c}_1$, return a frequency array that gives the conditional distribution of m_1 given $\mathbf{c}_1$, *i.e.,* $P(m_1|\mathbf{c}_1) = \mathrm{H}[\mathbf{c}_1][m_1] \,/\, \sum_i \mathrm{H}[\mathbf{c}_1][i]$.

During traversal and coding, we increment $\mathrm{H}[\mathbf{c_1}][m_1]$ whenever the $(\mathbf{c_1}, m_1)$ pair occurs. However, since $\mathbf{T}_e$ and $\mathbf{T}_o$ in general have different shapes, a full context may not exist, in which case we fall back to the shorter context $[m, l, h]$ for m_1. When a $(\mathbf{c}_1, m_1)$ pair occurs for the first time, $\mathrm{H}[\mathbf{c}_1][m_1] = 0$ and thus m_1 cannot be coded using $\mathbf{c}_1$. We instead encode an empty symbol at index -1 with frequency 1 (*i.e.,* $\mathrm{H}[\mathbf{c}_1][-1] = 1$) to signify to the decoder that $\mathbf{c}_1$ cannot be used, then encode m_1 with uniform probability *i.e.,* $1/(m+1)$. At the same time, we still increment $\mathrm{H}[\mathbf{c}_1][m_1]$ to avoid this zero-probability problem the next time the same $(\mathbf{c}_1, m_1)$ pair occurs. Finally, m_2 is also encoded with a context, which combines $\mathbf{c}_1$ and m_1, since m_1 is already known before m_2 is decoded.

7 Evaluation and Results

We evaluate the efficacy of our proposed solutions through various experiments. In the discussion that follows, both "BT on k-d tree" and "DT on k-d tree" are the baseline DG [36] methods; all other traversal-tree combinations are our contributions. 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 originally. Particle are always specified using 32-bit floating point coordinates, which are then quantized to 32-bit (96 bpp) integers prior to experiments. To generate an approximation when a traversal stops midway, for each node $(\mathbf{G}, n)$ at the traversal front, we output one (random) particle within $\mathbf{G}$. We use $|\mathrm{C}|$ 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. PSNR is defined as $20 \log_{10}(W/E)$, where E is the root mean square point-wise distance between every reference particle and its closest reconstructed particle, and W is the maximum dimension of the bounding box for the reference particles. A PSNR or 50 dB means that E is about $\frac{W}{300}$, and an improvement of 1 dB corresponds to a reduction of E by 10 percent. The rendered images are produced using OSPRay [93] after the particles have been decoded (*i.e.,* we do not decode and render simultaneously); as previously mentioned, this work focuses purely on encoding and decoding, and we hope that future work can extend the ideas presented here to perform direct rendering from compressed data. Note that unless explicitly mentioned otherwise (as in subsection 7.4 and subsection 7.5), truncated binary coding is used.

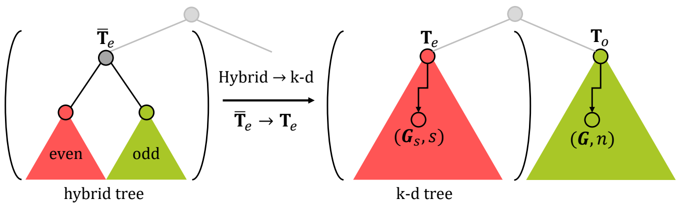


Fig. 10: The even subtree $\bar{\mathbf{T}}_e$ is transformed from a hybrid tree to a k-d tree $\mathbf{T}_e$. The odd subtree $\mathbf{T}_o$ is coded using a lockstep traversal with $\mathbf{T}_e$. Local information at the two front nodes ($(\mathbf{G}_s, s)$ and $(\mathbf{G}, n)$) are used for context coding. When $\mathbf{T}_o$ is fully coded, it is combined with $\mathbf{T}_e$ and transformed into a k-d tree for next-level prediction.

7.1 Adaptive Traversal of k-d Trees

AT (with the proposed scoring heuristic, Equation 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. 11). 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. Visual demonstration of the differences between low-bit-rate reconstructions using BT and AT is provided in Fig. 14 (see the first green-highlighted column pair). We render at low bit rates the outputs of the various traversal and tree combinations with OSPRay [93]. 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 occurs 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 (Equation 1) does not always work well for all datasets. For example, the rendering of the *coal* dataset (which contains simulated coal particles) in Fig. 12 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. 12b and Fig. 12c), indicating that PSNR does not always capture visual quality. When 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.

7.2 Traversals of Hybrid and Block-Hybrid Trees

In Fig. 14, we encode six datasets with different characteristics (rows) and decode them using five combinations of tree and traversal orders (columns) discussed in the paper. For each row, all columns are decoded at the same bit rate, but note that the number of decoded particles can be different for each method. It can be seen 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). Compared to BT on k-d tree (DG), DT on hybrid tree tend to produce better results for dense surface data (*girl*, *soldier*), and worse results for sparse scientific data. A specially difficult case for DT on hybrid tree is *molecule*, where the distribution is very sparse but the particles are specified with high precision. In such cases, refining a coarse subset of particles to high precision is not useful (see Fig. 14 (c5)).

For most datasets, 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). Interestingly, 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*). BAT can also fail visually (*cosmic web*) compared to DT on hybrid tree (see (f5) and (f6)) since when dense regions are clearly preferred, uniform refinement is not a good strategy. Finally, our hybrid and block-hybrid trees often generate significantly fewer particles at the same bit rates compared to BT on k-d trees (see *fissure*, *dam break*, and *cosmic web*), which should benefit downstream processing tasks. The most striking example can be seen by comparing (e6) with (e2), where BAT produces an almost identical-looking approximation to BT using only one-eighth the number of particles.

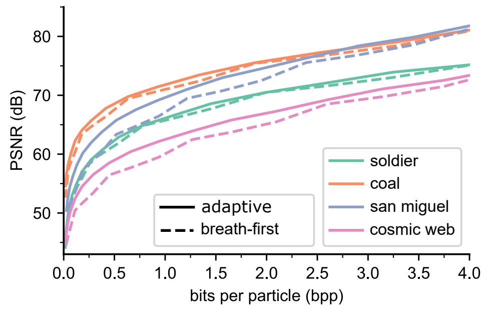


Fig. 11: Rate-distortion curves for AT and BT on k-d trees. AT not only outperforms BT on all datasets tested, but also produces significantly "smoother" rate-distortion curves.

7.3 Speed and Memory Footprint

Fig. 13 (a, b) 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 subsection 7.1). 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. 13b). The trade-off is higher reconstruction quality (Fig. 14). 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 test the scalability of BAT on block-hybrid tree against the state-of-the-art octree compressor, MPEG [17], using the TMC3 [94] reference implementation. We encode eight datasets in increasing numbers of particles, and record the encoding time and memory usage of both methods. Fig. 13 (c) 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 in this experiment, which consists 400 M particles.

We also encode a dataset with almost one billion particles (*detonation-large*, with 968M particles) using the block-hybrid tree, and then progressively decode and render three approximations from that same encoding (Fig. 15). Rendering is done with OSPRay [93] after a subset of particles of the original 968M particles is decoded in each case. Since OSPRay constructs its own acceleration data structure for rendering which inflates the memory requirement, without reducing the number of particles, the original dataset could not be rendered on our machine with 64 GB RAM (it was previously rendered using 3 TB of RAM [41]). With block-hybrid tree, high-quality reconstructions are possible at significantly lower particle counts, decoded progressively using a constant memory footprint (50 MB of RAM).

7.4 Binomial Coding

In Fig. 16, we plot rate-distortion curves for both truncated binary coding [89], [90] and our binomial coding using BT

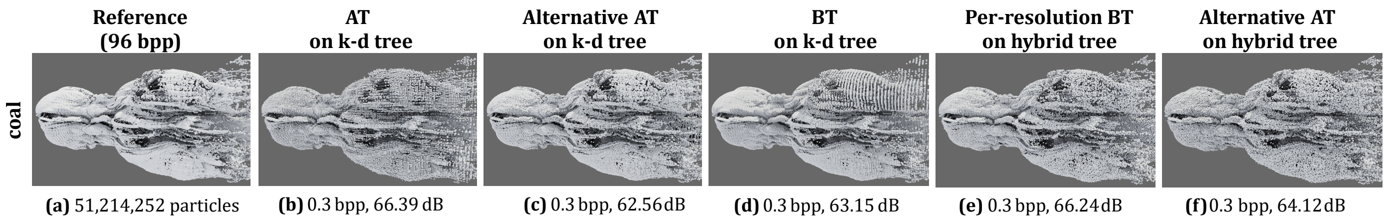
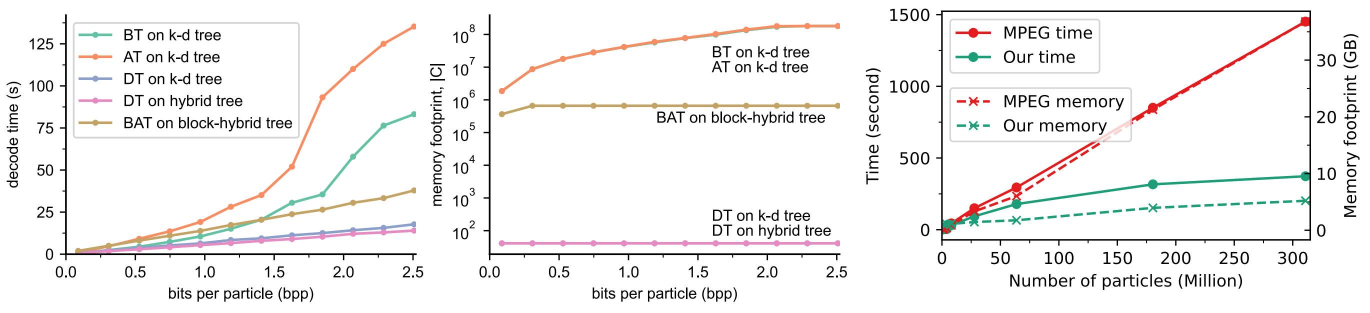


Fig. 12: Reconstruction results for alternative combinations of traversal orders and trees, including the use of an alternative scoring function for AT to obtain a better reconstruction visually (c), even at a lower PSNR. All reconstructions are at 0.3 bpp. Although not canonical, BT and AT on hybrid trees are very possible combinations, which may sometimes be preferable than BT on k-d trees, as is perhaps the case here.



(a) Decode times (b) Memory footprints (c) Comparison with MPEG

Fig. 13: (a, b): Decode times and memory footprints for combinations of trees and traversal methods, plotted for the *detonation* 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. (c): Compared to MPEG [17], our block-based encoder (BAT on block-hybrid tree) is almost $5\times$ to $7\times$ less expensive, and our time and memory costs also grow at much slower rates.

on k-d trees. We use three real-world datasets with approximately uniform distributions of particles, and a synthetic dataset where the distribution is truly uniform. It can be seen that, at the same data quality, binomial coding consistently improves the compression ratio by a factor between 10 and 20 percent. Conversely, at the same compression ratio, binomial coding on average improves the PSNR by about 0.5 dB. Fig. 17 visually demonstrates the difference in data quality between binomial and truncated binary coding, using the *fissure* dataset, which shows that even a seemingly small difference of 0.68 dB can translate to a significant visual difference. The most difficult dataset to compress is unsurprisingly the *random* one.

To further evaluate the coding efficiency of our implementation, we compress the synthetic dataset consisting of randomly generated particles, and compare the size of our compressed bitstream to a theoretically calculated code size. The theoretical code size is calculated by summing the theoretically smallest number of bits needed to encode n_1 under every k-d tree node $(\mathbf{G}, n)$, assuming n_1 is binomially distributed given n. Fig. 18 shows that our binomial coding implementation achieves code sizes that are virtually the same as the theoretically calculated ones across all tree depths, meaning our average code size for each tree node is close to the entropy of the binomial distribution. The same figure also shows that this (normalized) entropy approaches 0.5 as n gets larger toward the root of the tree, and 1 as n approaches 1 toward the leaves, consistent with our analysis in subsection 6.1. This result is encouraging for increasingly larger (and denser) datasets of the future, since progressive refinements will stop more toward the root, resulting in better compression for binomial coding, approaching a reduction ratio of 0.5 compared to truncated binary coding. In terms of performance, binomial coding runs about 1.5 times slower than truncated binary coding.

7.5 Odd-Even Context Coding

To test the efficacy of the odd-even context coding method, we compress several datasets with two methods: truncated binary coding and context coding. For this comparison, we always use DT on hybrid trees because odd-even context coding is designed to work with this combination. On a hybrid tree, DT visits the resolution levels from coarse to fine; we therefore record the ratio between the two bitstreams as each resolution level is processed, and plot these ratios in Fig. 19 (a ratio less than 1 means context coding is better). The figure shows that context coding improves on truncated binary coding in compression ratio for several datasets, compressing up to 40% better (for *dancer*), and in several cases up to 20% better at the last resolution level (lossless). Context coding also works better as the resolution gets finer, likely because sibling subtrees under an odd-even split are more correlated at finer resolution levels due to them being less spatially separated. On the other hand, as this distance increases toward coarser resolution levels, the correlation reduces, and thus compression suffers.

Fig. 19 also shows that our context coding does not work as well for some datasets, but is never significantly worse than truncated binary coding. We distinguish two kinds of datasets: densely sampled surfaces (dashed lines) and sparse but high-precision particles in scientific simulations (solid lines). It can be seen that our scheme works better for the surface datasets, where the particles form very distinct shapes and there are enough particles that the shapes are relatively well preserved by odd-even subsampling. Such datasets contain densely distributed particles, therefore they have significantly fewer in-cell refinement bits. Since such bits tend to be more random, datasets with fewer of them

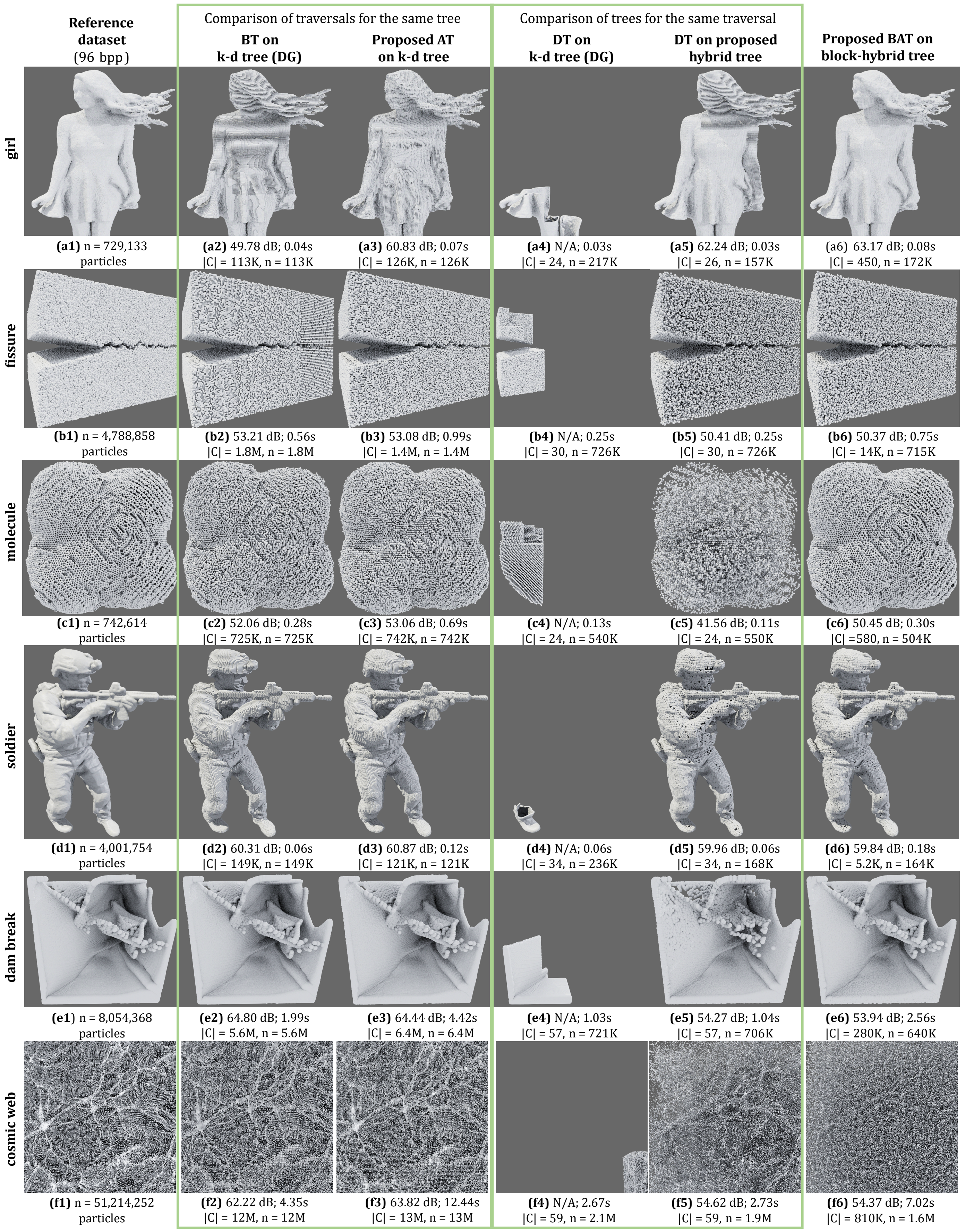

Reference dataset (96 bpp)
Comparison of traversals for the same tree
BT on k-d tree (DG)
Proposed AT on k-d tree
Comparison of trees for the same traversal
DT on k-d tree (DG)
DT on proposed hybrid tree
Proposed BAT on block-hybrid tree
girl
(a1) n = 729,133 particles
(a2) 49.78 dB; 0.04s |C| = 113K, n = 113K
(a3) 60.83 dB; 0.07s |C| = 126K, n = 126K
(a4) N/A; 0.03s |C| = 24, n = 217K
(a5) 62.24 dB; 0.03s |C| = 26, n = 157K
(a6) 63.17 dB; 0.08s |C| = 450, n = 172K
fissure
(b1) n = 4,788,858 particles
(b2) 53.21 dB; 0.56s |C| = 1.8M, n = 1.8M
(b3) 53.08 dB; 0.99s |C| = 1.4M, n = 1.4M
(b4) N/A; 0.25s |C| = 30, n = 726K
(b5) 50.41 dB; 0.25s |C| = 30, n = 726K
(b6) 50.37 dB; 0.75s |C| = 14K, n = 715K
molecule
(c1) n = 742,614 particles
(c2) 52.06 dB; 0.28s |C| = 725K, n = 725K
(c3) 53.06 dB; 0.69s |C| = 742K, n = 742K
(c4) N/A; 0.13s |C| = 24, n = 540K
(c5) 41.56 dB; 0.11s |C| = 24, n = 550K
(c6) 50.45 dB; 0.30s |C| =580, n = 504K
soldier
(d1) n = 4,001,754 particles
(d2) 60.31 dB; 0.06s |C| = 149K, n = 149K
(d3) 60.87 dB; 0.12s |C| = 121K, n = 121K
(d4) N/A; 0.06s |C| = 34, n = 236K
(d5) 59.96 dB; 0.06s |C| = 34, n = 168K
(d6) 59.84 dB; 0.18s |C| = 5.2K, n = 164K
dam break
(e1) n = 8,054,368 particles
(e2) 64.80 dB; 1.99s |C| = 5.6M, n = 5.6M
(e3) 64.44 dB; 4.42s |C| = 6.4M, n = 6.4M
(e4) N/A; 1.03s |C| = 57, n = 721K
(e5) 54.27 dB; 1.04s |C| = 57, n = 706K
(e6) 53.94 dB; 2.56s |C| = 280K, n = 640K
cosmic web
(f1) n = 51,214,252 particles
(f2) 62.22 dB; 4.35s |C| = 12M, n = 12M
(f3) 63.82 dB; 12.44s |C| = 13M, n = 13M
(f4) N/A; 2.67s |C| = 59, n = 2.1M
(f5) 54.62 dB; 2.73s |C| = 59, n = 1.9M
(f6) 54.37 dB; 7.02s |C| = 810K, n = 1.6M

Fig. 14: 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*).

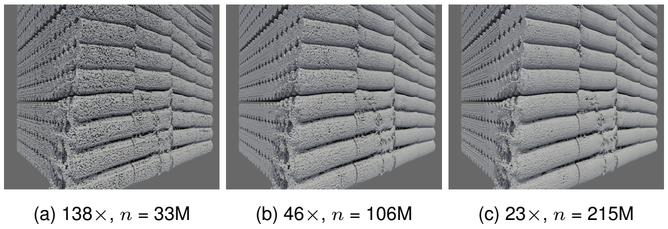


Fig. 15: Three approximations of a *detonation-large* simulation dataset, compressed and then decoded with our block-hybrid tree approach (compression ratio $k\times$ and corresponding number of particles, n, given). All three are snapshots of a single progressive decompression process, and all use only 50 MB for decoding.

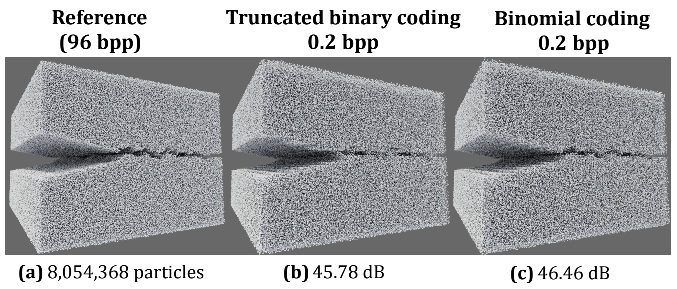


Fig. 17: At the same bit rates, binomial coding more faithfully reconstructs features in the original data: for the *fissure* dataset, the shape of the crack is more clearly defined with binomial coding.

often compress better. In contrast, the scientific simulation datasets are more difficult to compress because they are dominated by in-cell refinement bits, due to the particles being relatively sparse but stored with high precision.

There is one nonsurface dataset (*detonation*) for which our context-based scheme also works well. Here, the particles mostly follow a very regular arrangement as they represent arrays of explosives, and context modeling can exploit such global repetitions. Using the *dancer* dataset, Fig. 20 demonstrates that context coding can result in significant improvements in PSNR over truncated binary coding for progressive decompression. Visually, the improvements in PSNR translate to better reconstructed surface at low bit rates with significantly fewer artifacts (Fig. 20, bottom). In experiments, our implementation of odd-even context coding is often two times slower (for the encoder), and between three to eight times slower (for the decoder) than truncated binary coding. The extra cost mostly comes from the re-partitioning of the particles in the even subtree, which (as expected) doubles the computational cost for the encoder. Without odd-even context coding, the decoder is about four times faster than the encoder since it does not have to partition an array of particles to obtain node values (instead, node values are decoded from the bit stream). With odd-even context coding, which adds an extra partitioning step, both the encoder and the decoder run at similar speeds.

7.6 (Near) Lossless Compression Ratio

We compare near lossless compression ratios (lossless with respect to quantized particles) among four methods: DG [36], our proposed techniques, MPEG [17], and LASZip [95] for several datasets in Table 1. To achieve the best compression, we use k-d trees with binomial coding for *crystal*, *molecule*, *salt*, *fissure*, *detonation* and *random-80*, block-hybrid trees with odd-even context coding for *girl*, *dancer*, and *sodier*, and hybrid trees with truncated binary coding for the rest. 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 methods mostly achieve comparable lossless compression ratios against that of DG, which means our use of the odd-even splits does not degrade compression (while achieving much better quality-memory trade-offs as previously shown). For dense surface datasets (*girl*, *dancer*, *soldier*), our odd-even context coding results in significantly better lossless compression ratios over DG. LASZip produces the

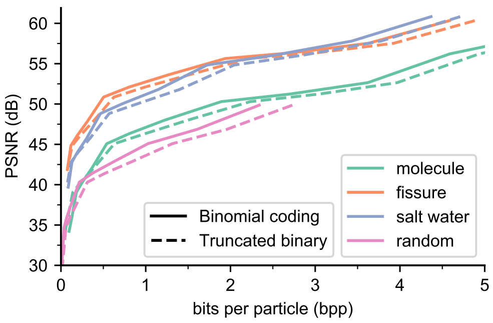


Fig. 16: Rate-distortion curves demonstrate that our proposed binomial encoding outperforms the standard truncated-binary coding [89] for datasets with approximately uniform distributions of particles. We also include a synthetic *random* dataset, with random particle distribution.

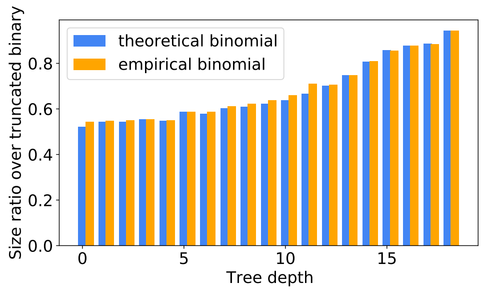


Fig. 18: Ratios of code sizes (binomial coding over truncated binary coding), both theoretically calculated and empirically measured, for a synthetic dataset with randomly generated particles. Our binomial coding implementation achieves almost perfect coding efficiency.

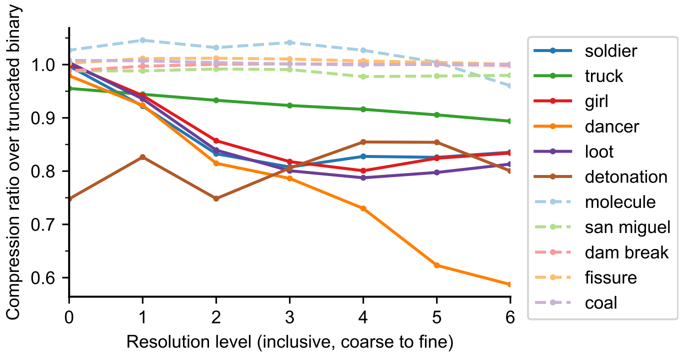


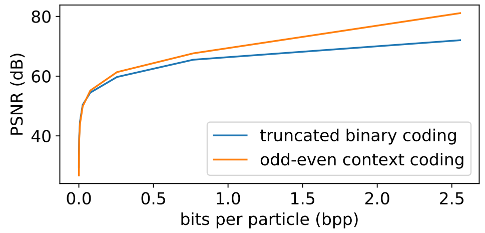
Fig. 19: Ratio between the compressed code sizes (context coding over truncated binary coding) at progressively finer resolution levels, with the last level corresponding to lossless compression. Our context coder works very well for dense surface datasets (solid lines), and less well for high-precision but sparse datasets (dashed lines).

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 *.

datasets	# particles	DG	Ours	MPEG	LASZip
crystal [2]	16K	2.10	2.11	2.44	1.56
girl [11]	729K	13.90	39.57	67.29	9.92
molecule [41]	742K	2.19	2.20	2.18	1.64
salt [4]	1.8M	2.36	2.38	2.33	1.51
fissure [41]	4.7M	2.54	2.56	3.50	2.13
dancer [11]	3.1M	22.81	35.42	65.02	6.89
soldier [11]	4M	13.70	16.14	21.00	5.56
san miguel [96]	3.6M	3.36	3.20	3.46	2.02
dam break [6]	8M	2.71	2.69	2.69	1.85
coal [97]	27M	3.45	3.42	3.36	2.20
cosmic web [41]	51M	3.17	3.12	3.06	1.72
*detonation [98]	180M	3.08	3.85	24.20	10.30
*random-80	1.6M	42.70	95.50	crashed	27.40

worst compression ratios among all methods in most cases, while MPEG compresses the best with its sophisticated context modeling. Unsurprisingly, MPEG also performs the best for the dense surface datasets (*girl*, *dancer*, *soldier*), since it is designed specifically for this kind of data. For many of the coarse but high-precision scientific datasets (*molecule*, *salt*, *dam break*, *coal* and *cosmic web*), however, MPEG's compression ratios are no better than ours.

detonation contains highly regular, repeating particle arrangements, which MPEG and LASZip take advantage of, whereas DG and ours do not. However, with additional dictionary-based compression, our compression ratio increases from 3.85 to 10.3, comparable to that of LASZip's. *random-80* is a synthetically generated dataset where a random 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 particles are densely distributed, it compresses twice better than DG and four times better than LASZip, whereas MPEG simply crashes. Most scientific datasets in practice are sparse relative to the grid size, but future data will likely become denser as more particles are captured and simulated.



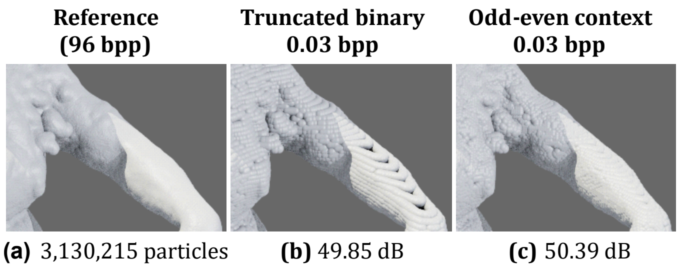


Fig. 20: Top: rate-distortion plots for the *dancer* dataset shows that odd-even context coding significantly outperforms truncated binary coding. Bottom: visual comparison between the two coding methods at the same bit rate of 0.03 bpp. Odd-even context coding reproduces the reference data more faithfully (with fewer artifacts).

8 Conclusion and Future Work

We have presented novel techniques along a tree-based particle compression pipeline, centered around the concept of an *odd-even split*. 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. Our proposed *binomial coding* and *odd-even context coding* significantly improve the compression ratio for datasets they are designed for by as much as 20% (for uniform distributions) and 40% (for densely sampled surfaces). All of the proposed techniques benefit the encoder and decoder equally. 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.

Like DG [36], our method does not take advantage of global redundancy, which could be useful to compress certain regular arrangements of particles, albeit at the expense of coding complexity and speed. To realize the odd-even splitting scheme, we need to quantize particle positions to avoid the inaccuracy caused by floating-point operations, but techniques may exist 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.

For tasks such as such as nearest-neighbor queries, occlusion culling, or empty-space skipping in rendering, it remains to be seen how our odd-even splitting mechanism affects application-level concerns, and to what extent our hybrid and block-hybrid trees can be used for noncompression purposes. For some datasets, neither binomial coding nor odd-even context coding may be applicable. Such datasets tend to contain nonuniform, relatively sparse but high-precision particles, which are common in scientific simulations. Better coding schemes might be invented to better target these cases, for which we hope the ideas presented here provide good starting points. Finally, it is also important to study task-oriented error metrics/heuristics and their utility to drive either tree construction or tree traversal, or both.

Acknowledgment

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.

References

[1] 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*, vol. 18, no. 12, pp. 2061–2068, 2012.

[2] 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*, vol. 11, no. 23, pp. 4606–4613, 2015.

[3] M. Le Muzic, L. Autin, J. Parulek, and I. Viola, "cellview: a tool for illustrative and multi-scale rendering of large biomolecular datasets," in *Eurographics Workshop on Visual Computing for Biomedicine*, vol. 2015, 2015, p. 61.

[4] "Scientific visualization contest 2016," https://www.uni-kl.de/sciviscontest/, accessed: 2021-03-31.

[5] D. Z. Zhang, Q. Zou, W. B. VanderHeyden, and X. Ma, "Material point method applied to multiphase flows," *Journal of Computational Physics*, vol. 227, no. 6, pp. 3159–3173, 2008.

[6] 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.

[7] 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*, vol. 15, no. 6, pp. 1251–1258, 2009.

[8] 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)*, 2016, pp. 56–64.

[9] 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*, vol. 42, pp. 49–65, 2016.

[10] 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.

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

[12] "OpenTopography," https://portal.opentopography.org/datasets, accessed: 2021-03-31.

[13] 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)*, 2013.

[14] 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)*, 2015, pp. 1–12.

[15] 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)*, 2020, pp. 105–115.

[16] 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*, 2017, pp. 486–493.

[17] 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*, vol. 9, no. 1, pp. 133–148, 2019.

[18] 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)*, 2019, pp. 145–153.

[19] 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*, vol. 27, no. 4, pp. 828–842, 2017.

[20] 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)*, 2000, pp. 343–352.

[21] M. Schütz, S. Ohrhallinger, and M. Wimmer, "Fast out-of-core octree generation for massive point clouds," *Computer Graphics Forum*, vol. 39, no. 7, pp. 155–167, 2020.

[22] 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)*, 2015, p. 1–10.

[23] D. Hoang, H. Bhatia, P. Lindstrom, and V. Pascucci, "High-quality and low-memory-footprint progressive decoding of large-scale particle data," in *2021 IEEE 11th Symposium on Large Data Analysis and Visualization (LDAV)*, 2021, pp. 32–42.

[24] D. Hoang, "Progressive particle compression," https://github.com/hoang-dt/particle-compression, 2022.

[25] E. Gobbetti and F. Marton, "Layered point clouds: a simple and efficient multiresolution structure for distributing and rendering gigantic point-sampled models," *Computers & Graphics*, vol. 28, no. 6, pp. 815–826, 2004.

[26] R. Schnabel, S. Möser, and R. Klein, "A parallelly decodeable compression scheme for efficient point-cloud rendering," in *Eurographics Conference on Point-Based Graphics (SPBG)*, 2007, pp. 214–226.

[27] 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*, vol. 66, no. 3, pp. 701–717, 2020.

[28] 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*, vol. 29, pp. 313–322, 2020.

[29] J. Peng and C. C. J. Kuo, "Octree-based progressive geometry encoder," in *Internet Multimedia Management Systems IV*, vol. 5242, 2003, pp. 301–311.

[30] M. Hosseini and C. Timmerer, "Dynamic adaptive point cloud streaming," in *Packet Video Workshop (PV)*, 2018, pp. 25–30.

[31] R. Schnabel and R. Klein, "Octree-based point-cloud compression," in *Eurographics Conference on Point-Based Graphics (SPBG)*, 2006, pp. 111–121.

[32] Y. Huang, J. Peng, C.-J. Kuo, and M. Gopi, "A generic scheme for progressive point cloud coding," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 2, pp. 440–453, 2008.

[33] E. Alexiou and T. Ebrahimi, "On the performance of metrics to predict quality in point cloud representations," in *Applications of Digital Image Processing XL*, vol. 10396, 2017, p. 103961H.

[34] S. Park and S. Lee, "Multiscale representation and compression of 3D point data," *IEEE Transactions on Multimedia*, vol. 11, no. 1, pp. 177–183, 2009.

[35] H. Lee, M. Desbrun, and P. Schröder, "Progressive encoding of complex isosurfaces," *ACM Transactions on Graphics*, vol. 22, no. 3, pp. 471–476, 2003.

[36] O. Devillers and P.-M. Gandoin, "Geometric compression for interactive transmission," in *IEEE Visualization (VIS)*, 2000, pp. 319–326.

[37] 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)*, 2010, pp. 5–12.

[38] M. Hopf, M. Luttenberger, and T. Ertl, "Hierarchical splatting of scattered 4D data," *IEEE Computer Graphics and Applications*, vol. 24, no. 4, pp. 64–72, 2004.

[39] 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)*, 2006, pp. 105–113.

[40] I. Wald and H.-P. Seidel, "Interactive ray tracing of point-based models," in *Eurographics Symposium on Point-Based Graphics (SPBG)*, 2005, pp. 9–16.

[41] 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)*, 2015, pp. 57–64.

[42] 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)*, 2021, pp. 547–556.

[43] 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)*, 2011, pp. 1151–1160.

[44] J. E. S. Khalil, A. Munteanu, L. Denis, P. Lambert, and R. V. d. Walle, "Scalable feature-preserving irregular mesh coding," *Computer Graphics Forum*, vol. 36, no. 6, pp. 275–290, 2017.

[45] S. Valette and R. Prost, "Wavelet-based progressive compression scheme for triangle meshes: wavemesh," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 2, pp. 123–129, 2004.

[46] A. Maglo, C. Courbet, P. Alliez, and C. Hudelot, "Progressive compression of manifold polygon meshes," *Computers & Graphics*, vol. 36, no. 5, pp. 349–359, 2012.

[47] 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*, vol. 66, no. 3, pp. 666–681, 2018.

[48] 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)*, 2006, pp. 364–375.

[49] 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)*, 2004, pp. 95–103.

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

[51] 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)*, 2013, pp. 1–7.

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

[53] M. Pauly, M. Gross, and L. P. Kobbelt, "Efficient simplification of point-sampled surfaces," in *IEEE Visualization (VIS)*, 2002, pp. 163–170.

[54] 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*, vol. 29, no. 1, pp. 69–83, 2013.

[55] P. Lindstrom, "Out-of-core construction and visualization of multiresolution surfaces," in *Symposium on Interactive 3D Graphics and Games (I3D)*, 2003, pp. 93–102.

[56] D. King and J. Rossignac, "Optimal bit allocation in compressed 3D models," *Computational Geometry*, vol. 14, no. 1, pp. 91–118, 1999.

[57] S. Valette, R. Chaine, and R. Prost, "Progressive lossless mesh compression via incremental parametric refinement," *Computer Graphics Forum*, vol. 28, no. 5, pp. 1301–1310, 2009.

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

[59] 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)*, 2005, pp. 609–616.

[60] M. Botsch, A. Wiratanaya, and L. Kobbelt, "Efficient high quality rendering of point sampled geometry," in *Eurographics Workshop on Rendering (EGRW)*, 2002, pp. 53–64.

[61] O. Dovrat, I. Lang, and S. Avidan, "Learning to sample," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 2755–2764.

[62] 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*, vol. 16, no. 2, pp. 273–286, 2010.

[63] 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)*, 2006, pp. 83–89.

[64] 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)*, 2009, pp. 420–429.

[65] J. Smith, G. Petrova, and S. Schaefer, "Progressive encoding and compression of surfaces generated from point cloud data," *Computers & Graphics*, vol. 36, no. 5, pp. 341–348, 2012.

[66] T. Golla and R. Klein, "Real-time point cloud compression," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2015, pp. 5087–5092.

[67] T. Ochotta and D. Saupe, "Image-Based Surface Compression," *Computer Graphics Forum*, vol. 27, no. 6, pp. 1647–1663, 2008.

[68] S. Fleishman, D. Cohen-Or, M. Alexa, and C. T. Silva, "Progressive point set surfaces," *ACM Transactions on Graphics*, vol. 22, no. 4, pp. 997–1011, 2003.

[69] E. Hubo, T. Mertens, T. Haber, and P. Bekaert, "Self-similarity based compression of point set surfaces with application to ray tracing," *Computers & Graphics*, vol. 32, no. 2, pp. 221–234, 2008.

[70] J. Digne, R. Chaine, and S. Valette, "Self-similarity for accurate compression of point sampled surfaces," *Computer Graphics Forum*, vol. 33, no. 2, pp. 155–164, 2014.

[71] P. d. O. Rente, C. Brites, J. Ascenso, and F. Pereira, "Graph-Based Static 3D Point Clouds Geometry Coding," *IEEE Trans. on Multimedia*, vol. 21, no. 2, pp. 284–299, 2019.

[72] D. Thanou, P. A. Chou, and P. Frossard, "Graph-Based Compression of Dynamic 3D Point Cloud Sequences," *IEEE Transactions on Image Processing*, vol. 25, no. 4, pp. 1765–1778, 2016.

[73] S. Milani, E. Polo, and S. Limuti, "A Transform Coding Strategy for Dynamic Point Clouds," *IEEE Transactions on Image Processing*, vol. 29, pp. 8213–8225, 2020.

[74] L. Huang, S. Wang, K. Wong, J. Liu, and R. Urtasun, "Octsqueeze: Octree-structured entropy model for lidar compression," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1313–1323.

[75] A. F. Guarda, N. M. Rodrigues, and F. Pereira, "Adaptive deep learning-based point cloud geometry coding," *IEEE Journal of Selected Topics in Signal Processing*, vol. 15, no. 2, pp. 415–430, 2020.

[76] J. Wang, H. Zhu, H. Liu, and Z. Ma, "Lossy point cloud geometry compression via end-to-end learning," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 12, pp. 4909–4923, 2021.

[77] D. T. Nguyen, M. Quach, G. Valenzise, and P. Duhamel, "Lossless coding of point cloud geometry using a deep generative model," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 31, no. 12, pp. 4617–4629, 2021.

[78] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, 1998.

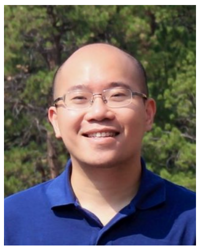
[79] S. Gumhold, Z. Kami, M. Isenburg, and H.-P. Seidel, "Predictive point-cloud compression," in *ACM SIGGRAPH Sketches*, 2005, p. 137.

[80] B. Merry, P. Marais, and J. Gain, "Compression of dense and regular point clouds," in *International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH)*, 2006, pp. 15–20.

[81] J. Krüger, J. Schneider, and R. Westermann, "Compression and rendering of iso-surfaces and point sampled geometry," *The Visual Computer*, vol. 22, pp. 517–530, 2006.

[82] A. Omeltchenko, T. J. Campbell, R. K. Kalia, X. Liu, A. Nakano, and P. Vashishta, "Scalable i/o of large-scale molecular dynamics simulations: A data-compression algorithm," *Computer Physics Communications*, vol. 131, no. 1-2, pp. 78–85, 2000.

[83] "Draco: Geometric coding for dynamic voxelized point clouds," https://github.com/google/draco, accessed: 2021-03-31.
[84] J. Peng and C.-J. Kuo, "Progressive geometry encoder using octree-based space partitioning," in *IEEE International Conference on Multimedia and Expo (ICME)*, 2004, pp. 1–4.
[85] 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)*, 2001, pp. 45–45.
[86] W. Sweldens, "Wavelets and the lifting scheme: A 5 minute tour," *Journal of Applied Mathematics and Mechanics*, vol. 76, pp. 41–44, 1996.
[87] 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 Distributed Processing Symposium (IPDPS)*, 2021.
[88] 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*, vol. 27, no. 2, pp. 603–613, 2021.
[89] J. Teuhola, "Tournament Coding of Integer Sequences," *The Computer Journal*, vol. 52, no. 3, pp. 368–377, 2009.
[90] A. Moffat and L. Stuiver, "Binary interpolative coding for effective index compression," *Information Retrieval*, vol. 3, pp. 25–47, 2004.
[91] Y. Dodge, *The Concise Encyclopedia of Statistics*, 2008.
[92] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, p. 256–294, jul 1998.
[93] 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*, vol. 23, no. 1, pp. 931–940, 2017.
[94] "MPEG-PCC-TMC13: Geometrybased point cloud compression," https://github.com/MPEGGroup/mpeg-pcc-tmc13, accessed: 2021-03-31.
[95] M. Isenburg, "LASzip: lossless compression of LiDAR data," *Photogrammetric Engineering & Remote Sensing*, vol. 79, no. 2, 2013.
[96] M. McGuire, "Computer graphics archive," https://casual-effects.com/data, accessed: 2021-03-31.
[97] 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)*, 2012, pp. 2441–2448.
[98] 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*, vol. 38, no. 5, pp. S101–S122, 2016.



Duong Hoang is a Ph.D. student in computer science at the Scientific Computing and Imaging (SCI) Institute, University of Utah. Before joining Utah, he obtained Bachelor and Masters degrees in computer science from the National University of Singapore. His research interests include data compression, scientific data visualization and analysis, computer graphics and scientific computing.

Harsh Bhatia is a Computer Scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research spans the broad area of visualization and computational topology, with special focus on scientific data. Harsh is also interested in ML-based approaches for scientific applications. Prior to joining LLNL, Harsh earned his Ph.D. from Scientific Computing & Imaging Institute at the University of Utah in 2015, where he worked on the feature extraction for vector fields.

Peter Lindstrom Peter Lindstrom is a Computer Scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory, where he leads several research efforts that span data compression, scientific visualization, and scientific computing. He received a Ph.D. in computer science from Georgia Institute of Technology in 2000 and B.S. degrees in computer science, mathematics, and physics from Elon University in 1994. Peter previously served as Editor in Chief for Graphical Models, Associate Editor for IEEE Transactions on Visualization and Computer Graphics, and Paper Chair for IEEE VIS. He is a Senior Member of IEEE.

Valerio Pascucci received the EE laurea (master's) degree from the University La Sapienza, Rome, Italy, in December 1993, as a member of the Geometric Computing Group, and the PhD degree in computer science from Purdue University, in May 2000. He is a faculty member at the Scientific Computing and Imaging (SCI) Institute, University of Utah. Before joining SCI, he served as a project leader at the Lawrence Livermore National Laboratory (LLNL), Center for Applied Scientific Computing (from May 2000) and as an adjunct professor at the Computer Science Department of the University of California, Davis (from July 2005). Prior to his tenure at CASC, he was a senior research associate at the University of Texas at Austin, Center for Computational Visualization, CS, and TICAM Departments. He is a member of the IEEE.

Appendix A
Generic Tree-Based Decoder

We give a generic 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 bit stream BS storing the encoded bits. A data structure C, supporting a PUSH and a POP operations (*e.g.*, a stack or queue), keeps track of the traversal front. In each iteration, a front node $(\mathbf{B}, n)$ is popped from C, followed by an optional callback, which, depending on whether $(\mathbf{B}, n)$ is a leaf, can append it to an output list of particles or to a tree. For inner nodes, the number of particles in the left child (*i.e.*, n_1) is decoded from BS with the knowledge of n. $(\mathbf{B}, n)$ is then SPLIT into two children, which are then pushed back into C, and the whole process repeats until either C is empty or when DONE(BS) is true (*e.g.*, when enough bits have been read from BS).

Algorithm 1 Generic tree-based decoder. Inputs: n_0 particles in bounding box $\mathbf{B}_0$, bitstream BS, node container C (*e.g.*, stack, queue).

```
 1: function DecodeTree($n_0, \mathbf{B}_0$, BS, C)
 2:     C.Push($\mathbf{B}_0, n_0$)              ▷ push node $(\mathbf{B}_0, n_0)$ to C
 3:     D ← X                           ▷ initial dimension of splitting
 4:     while not Done(BS) and not C.IsEmpty do
 5:         $(\mathbf{B}, n) \leftarrow$ C.Pop
 6:         if $n = 0$ then
 7:             continue
 8:         end if
 9:         if IsLeaf($\mathbf{B}, n$) then
10:             LeafFunc($\mathbf{B}, n$)          ▷ e.g., output a particle
11:             continue
12:         else
13:             InnerFunc($\mathbf{B}, n$)        ▷ e.g., create a tree node
14:         end if
15:         $n_1 \leftarrow$ Decode($n$, BS)   ▷ particles in the left child
16:         $n_2 \leftarrow n - n_1$               ▷ particles in the right child
17:         $\mathbf{B}_1, \mathbf{B}_2 \leftarrow$ Split($\mathbf{B}$, D)           ▷ split $\mathbf{B}$ along D
18:         D ← Next(D)             ▷ e.g., if D = X, Next(D) = Y
19:         C.Push($\mathbf{B}_1, n_1$)
20:         C.Push($\mathbf{B}_2, n_2$)
21:     end while
22: end function
```

Appendix B
Hybrid Tree Encoding

We give the pseudocode for a hybrid tree encoder in Algorithm 2. Compared to Algorithm 1, this algorithm is written in recursive form for simplicity, includes details on when to use odd-even splits, and how we switch to encoding empty cells if the current grid is dense in particles (lines 9 – 11). A new EncodeRefinementBits step is invoked to output a particle's in-cell refinement bits (*i.e.,* gray nodes in Fig. 6). Finally, we further specify the actual low-level encoding method to be the commonly used truncated binary coding [89], [90].

Algorithm 2 Depth-first, recursive hybrid tree encoding. Inputs: a set of particles P residing in a grid $\mathbf{G}$, split type S which is either ODD-EVEN or K-D (it is ODD-EVEN initially), dimension of splitting D, and bitstream BS.

```
1:  function DtHybridEncode($P$, $\mathbf{G}$, S, D, BS)
        ▷ step 1: encode in-cell refinement bits
2:      if $G = 1$ then                          ▷ one particle in a single cell
3:          EncodeRefinementBits($P$, $\mathbf{G}$, D, BS)
4:          return
5:      end if
        ▷ step 2: split the current node
6:      $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow$ Partition($P, \mathbf{G}$, S, D)
        ▷ step 3: encode the current node
7:      $n \leftarrow |P|$              ▷ number of particles in current node
8:      $n_1 \leftarrow |P_1|$            ▷ number of particles in the left child
9:      if $G - n < n$ then                   ▷ grid is dense in particles
10:         $n \leftarrow G - n$            ▷ encode number of empty cells
11:         $n_1 \leftarrow |\mathbf{G}_1| - n_1$
12:     end if
13:     TruncatedBinaryEncode($n_1, n$, BS)
        ▷ step 4: recurse
14:     D $\leftarrow$ Next(D)                 ▷ next dimension of splitting
15:     if $|P_1| > 0$ then                                   ▷ left child
16:         DtHybridEncode($P_1, \mathbf{G_1}$, S, D, BS)
17:     end if
18:     if $|P_2| > 0$ then                                  ▷ right child
19:         S $\leftarrow$ K-D                  ▷ by definition of hybrid-trees
20:         DtHybridEncode($P_2, \mathbf{G_2}$, S, D, BS)
21:     end if
22: end function
23:
24: function Partition($P$, $\mathbf{G}$, S, D)
25:     $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow \varnothing$
26:     if S = ODD-EVEN then
27:         $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow$ OddEvenPartition($P, \mathbf{G}$, D)
28:     else
29:         $P_1, P_2\mathbf{G}_1, \mathbf{G}_2 \leftarrow$ KdPartition($P, \mathbf{G}$, D)
30:     end if
31:     return $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2$
32: end function
```

APPENDIX C
BLOCK-ADAPTIVE TRAVERSAL

The pseudocode for BAT is given in Algorithm 3. 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 COARSETRAVERSE and MEDIUMTRAVERSE functions, N denotes the current node at the traversal front, with N.l storing its resolution level and N.n its number of particles. The FINETRAVERSE function performs fine-phase (breadth-first) traversal, which decodes the in-cell refinement bits.

Algorithm 3 Block-adaptive traversal. Inputs: a priority queue of blocks Q, coarse bit stream BS.

```
1:  function BLOCKADAPTIVETRAVERSE(Q, BS)
2:      if Q.ISEMPTY then              ▷ still in coarse traversal phase
3:          COARSETRAVERSE(CS)
4:      end if
5:      while not Q.ISEMPTY do
6:          B ← Q.POP                  ▷ B is the block with the largest error
7:          if B.$n^* < $ B.$n$ then   ▷ not all of B's particles visited
8:              MEDIUMTRAVERSE(B)
9:              Q.PUSH(B)
10:         else ▷ all particles visited, do fine traversal phase
11:             FINETRAVERSE(B)
12:             if not ALLBITSREAD(B.BS) then
13:                 Q.PUSH(B)
14:             end if
15:         end if
16:     end while
17: end function
18:
    ▷ Coarse-phase traversal: Algorithm 1 with a queue
    container, bitstream BS and the following callback.
19: function COARSETRAVERSE(BS)
    ▷ LEAFFUNC                         ▷ from each leaf we create a block B
20:     B.$l \leftarrow 0$             ▷ with resolution level 0,
21:     B.$n \leftarrow$ N.$n$         ▷ and appropriate number of particles,
22:     B.$n^* \leftarrow 0$           ▷ and number of visited particles
23:     B.ST ← empty stack
24:     B.ST.PUSH(N)                   ▷ N is the root node of block B
25:     Q.PUSH(B)
26: end function
27:
    ▷ Medium-phase traversal: Algorithm 1 with B.ST as
    the (stack) container, B.BS as the bitstream, and the
    following callbacks.
28: function MEDIUMTRAVERSE(B)                  ▷ DT of block B
    ▷ LEAFFUNC                                  ▷ each leaf is a particle
29:     B.$n^* \leftarrow$ B.$n^* + 1$          ▷ a new particle visited
30:     B.$n_l^* \leftarrow$ B.$n_l^* + 1$
    ▷ INNERFUNC
31:     if N.$l \neq$ B.$l$ then       ▷ reached a finer resolution level
32:         B.$l \leftarrow$ N.$l$     ▷ update the current resolution level
33:         B.$n_l \leftarrow$ N.$n$   ▷ update number of particles
34:         B.$n_l^* \leftarrow 0$     ▷ reset number of visited particles
35:     end if
36: end function
37:
    ▷ Fine-phase traversal: for simplicity, always read an
    entire bit plane of the input block B.
38: function FINETRAVERSE(B)
39:     for each particle P $\in$ B.$P$ do
40:         BIT ← READONEBIT(B.BS)
41:         P ← REFINE(P, BIT)
42:     end for
43:     B.$l \leftarrow$ B.$l + 1$
44: end function
```

Appendix D
Binomial Coding and Entropy

To get a better understanding of the theoretical gain achievable with binomial coding, we look at the entropy of the binomial distribution, H, which is

$$H = -\sum_{k=0}^{n} \binom{n}{k} p^k (1-p)^{n-k} \log_2 \left(\binom{n}{k} p^k (1-p)^{n-k} \right)$$

We use the de Moivre-Laplace theorem to get

$$H \simeq - \int_{-\infty}^{\infty} \frac{dx}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \log_2 \left(\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \right)$$

$$\simeq - \int_{-\infty}^{\infty} \frac{dx}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \left(-\log_2 (\sigma\sqrt{2\pi}) - \frac{(x-\mu)^2}{2\sigma^2} \log_2 e \right)$$

By the definitions of a normal distribution and its variance, we have, respectively, $\int_{-\infty}^{\infty} \frac{dx}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} = 1$ and $\int_{-\infty}^{\infty} \frac{dx}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} (x-\mu)^2 = \sigma^2$. Therefore,

$$H \simeq \log_2 (\sigma\sqrt{2\pi}) + \frac{1}{2}\log_2 e = \frac{1}{2}\log_2 (2\pi e \sigma^2)$$
$$= \frac{1}{2}\log_2 (2\pi e n p (1-p))$$

Algorithm 4 Binomial coding. Inputs: CDF tables CDFS, arithmetic bit stream BS_a, truncated binary bit stream BS_b, number of particles in the current node n, number of particles in the left child n_1.

```
1: function BinomialEncode(CDFS, $\text{BS}_a$, $\text{BS}_b$, $n, n_1$)
2:     $\mu \leftarrow n/2$
3:     $\sigma^2 \leftarrow n/4$
4:     small $\leftarrow n \leq 30$
5:     while True do
6:         $a \leftarrow \lceil a \rceil$
7:         $b \leftarrow \lfloor b \rfloor$
8:         if $a = b$ then          ▷ $n_1 = a = b$, no need to encode
9:             return
10:        end if
11:        $n \leftarrow b - a$
12:        if small then                  ▷ exact binomial modeling
13:            ArithmeticEncode($n_1 - a$, CDFS[$n$], $\text{BS}_a$)
14:            return
15:        end if
16:        $f_a \leftarrow F_{\mu,\sigma^2}(a)$       ▷ $F_{\mu,\sigma^2}$ is the CDF of $N(\mu, \sigma^2)$
17:        $f_b \leftarrow F_{\mu,\sigma^2}(b)$
18:        $m \leftarrow F^{-1}_{\mu,\sigma^2}((f_a + f_b)/2)$
19:        if $m = a$ or $m = b$ then          ▷ ran out of precision
20:            TruncatedBinaryEncode($n, n_1 - a$, $\text{BS}_b$)
21:            return
22:        end if
23:        if $n_1 < m$ then
24:            WriteBit($\text{BS}_b$, 0)
25:            $b \leftarrow m$
26:        else
27:            WriteBit($\text{BS}_b$, 1)
28:            $a \leftarrow m$
29:        end if
30:    end while
31: end function
```

APPENDIX E
ODD-EVEN CONTEXT CODING

Algorithm 5 gives the pseudocode for our odd-even context encoder, including details on how to perform the lockstep traversal inline and inplace.

Algorithm 5 Odd-even context encoding. The inputs P, D, $\mathbf{G}$, S, BS$_a$, BS$_b$ are the same as in previous algorithms. l and h are, respectively, the resolution level and tree depth of the current node during the recursive DT. R is a sorted array containing particles of the current reference subtree, and $\mathbf{G}_R$ is the corresponding subgrid where these particles reside.

```
function OeEncode($R$, $\mathbf{G}_R$, $P$, $\mathbf{G}$, D, S, $l$, $h$, BS$_a$, BS$_b$)
    ▷ first two steps are the same as Algorithm 2 . . .
    ▷ step 3: encode the current node
    $m \leftarrow \log_2(n+1)$
    $m_1 \leftarrow \log_2(n_1+1)$
    $m_2 \leftarrow \log_2(n-n_1+1)$
    $R_1, R_2, \mathbf{G}_{R_1}, \mathbf{G}_{R_2} \leftarrow$ KdPartition($R$,$\mathbf{G}_R$,D)
    $P_1, P_2, \mathbf{G}_1, \mathbf{G}_2 \leftarrow$ Partition($P$,$\mathbf{G}$,S,D)   ▷ see Alg. 2
    if $|R| > 0$ then                              ▷ reference node is present
        $r \leftarrow \log_2(|R|+1)$
        $r_1 \leftarrow \log_2(|R_1|+1)$
        $r_2 \leftarrow \log_2(|R_2|+1)$
        $\mathbf{c}_1 \leftarrow [m, r, r_1, r_2, l, h]$
        ContextEncode($m_1, \mathbf{c}_1$, BS$_a$, BS$_b$)
        if CannotInfer$m_2$From($m, m_1$) then
            $\mathbf{c}_2 \leftarrow [m, r, r_1, r_2, l, h, m_1]$
            ContextEncode($m_2, \mathbf{c}_2$, BS$_a$, BS$_b$)
        end if
    else                          ▷ no reference node, minimal context
        $\mathbf{c}_1 \leftarrow [m, l, h]$
        ContextEncode($m_1, \mathbf{c}_1$, BS$_a$, BS$_b$)
        if CannotInfer$m_2$From($m, m_1$) then
            $\mathbf{c}_2 \leftarrow [m, l, h, m_1]$
            ContextEncode($m_2, \mathbf{c}_2$, BS$_a$, BS$_b$)
        end if
    end if
    ▷ step 4: recurse
    D $\leftarrow$ Next(D)                          ▷ next axis of splitting
    $h \leftarrow h + 1$                               ▷ next tree depth
    if $|P_1| > 0$ then                                    ▷ left child
        $l_n \leftarrow l +$ (S = ODD-EVEN)   ▷ next resolution level
        $R_1 \leftarrow$ (S = ODD-EVEN) ? $\varnothing$
        OeEncode($R_1$, $\mathbf{G}_{R_1}$, $P_1$, $\mathbf{G}_1$, D, S, $l_n$, $h$, BS$_a$, BS$_b$)
    end if
    if $|P_2| > 0$ then                                   ▷ right child
        $(R_2, \mathbf{G}_{R_2}) \leftarrow$ (S $\neq$ K-D) ? $(R_1, \mathbf{G}_{R_1})$
        OeEncode($R_2$, $\mathbf{G}_{R_2}$, $P_2$, $\mathbf{G}_2$, D, S, $l$, $h$, BS$_a$, BS$_b$)
    end if
end function

    ▷ encode $m_1$ using context $\mathbf{c}_1$ into bit streams BS$_a$, BS$_b$
function ContextEncode($m_1$, $\mathbf{c}_1$, BS$_a$, BS$_b$)
    if H[$\mathbf{c}_1$][$m_1$] $> 0$ then              ▷ context can be used
        ArithmeticEncode($m_1$, H[$\mathbf{c}_1$], BS$_a$)
    else      ▷ context cannot be used due to 0 probability
        H[$\mathbf{c}_1$][$-1$] $\leftarrow 1$
        ArithmeticEncode($-1$, H[$\mathbf{c}_1$], BS$_a$)
        TruncatedBinaryEncode($m_1$, $\mathbf{c}_1.m$, BS$_b$)
    end if
    H[$\mathbf{c}_1$][$m_1$] $\leftarrow$ H[$\mathbf{c}_1$][$m_1$] $+ 1$
end function
```