CPU Ray Tracing Large Particle Data with Balanced P-k-d Trees

Ingo Wald[1] Aaron Knoll[2,3,5] Gregory P Johnson[1] Will Usher[2] Valerio Pascucci[2,5] Michael E Papka[3,4,5]

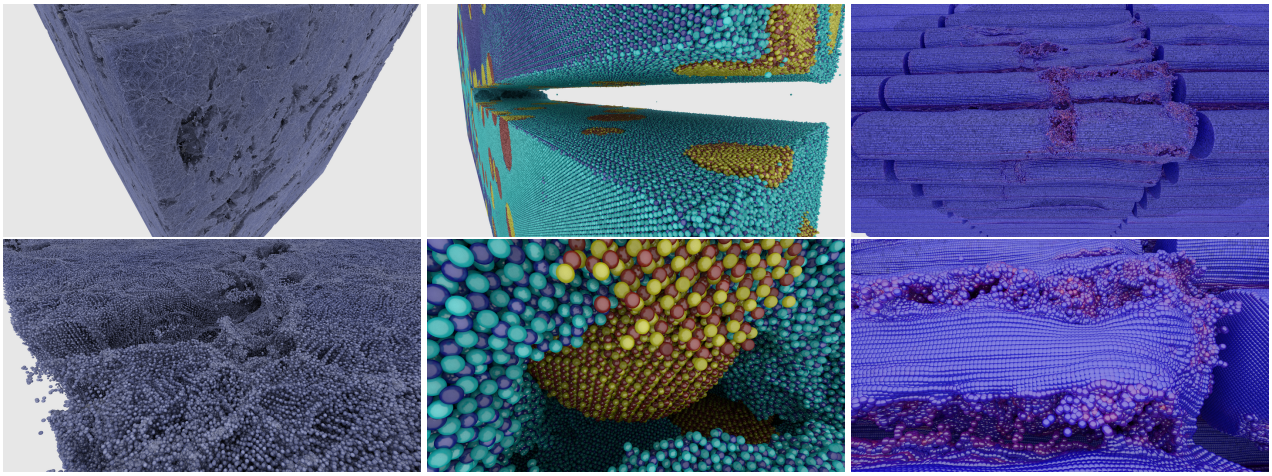

Fig. 1. Full-detail ray tracing of giga-particle data sets. From left to right: *CosmicWeb* early universe data set from a P3D simulation with 29 billion particles; a 100 million atom molecular dynamics $Al_2O_3 - SiC$ materials fracture simulation; and a 1.3 billion particle *Uintah MPM* detonation simulation. Using a quad-socket, 72-core 2.5 GHz Intel® † Xeon® E7-8890 v3 Processor with 3 TB RAM and path-tracing with progressive refinement at 1 sample per pixel, these far and close images (above and below) are rendered at 1.6 (far) / 1.0 (close) fps (left), 2.0 / 1.2 fps (center), and 1.0 / 0.9 fps (right), respectively, at 4K (3840×2160) resolution. All examples use our balanced P-k-d tree, an acceleration structure which requires little or no memory cost beyond the original data.

Abstract—We present a novel approach to rendering large particle data sets from molecular dynamics, astrophysics and other sources. We employ a new data structure adapted from the original balanced k-d tree, which allows for representation of data with trivial or no overhead. In the OSPRay visualization framework, we have developed an efficient CPU algorithm for traversing, classifying and ray tracing these data. Our approach is able to render up to billions of particles on a typical workstation, purely on the CPU, without any approximations or level-of-detail techniques, and optionally with attribute-based color mapping, dynamic range query, and advanced lighting models such as ambient occlusion and path tracing.

Index Terms—Ray tracing, Visualization, Particle Data, k-d Trees

1 Introduction

With ever increasing compute power, simulations produce increasingly large quantities of data to be visualized. The largest computational codes predominantly generate particle data: molecular dynamics materials computations, mesoscale or macroscale atomistic simulations, and cosmology and astrophysics n-body codes. The largest cosmology simulations now generate trillions of particles at scale; these petabytes of data are seldom even stored, let alone visualized. Examples of such data are shown in Figure 1.

At such scale, traditional rasterization-based approaches to rendering such data sets become problematic: simply rendering each particle with a tessellated sphere becomes prohibitive, and even splatting and impostor techniques are limited by rasterization performance, GPU memory limitations and PCI bandwidth. This becomes more challenging if the user desires to interact with multiple data time steps, apply different attribute color mappings, or perform interactive parameter range selection. State-of-the-art GPU techniques [17] can render up to 10 billion particles on a single GPU with level-of-detail (LOD). However, LOD approaches must be specifically tuned to individual data and rendering modalities. For extremely large datasets from cosmology, showing full-detail data is challenging but crucial to understanding both structure and scale of the simulation (Figure 2). Ideally, we wish to visualize data at full-resolution without LOD. GPU visualization clusters can render on the order of hundreds of billions of particles with no LOD in parallel [25]. However, repartitioning and compositing massive point data can be costly, and requires data-parallel software architectures and significant compute resources.

With the right algorithms, large-scale visualization is achievable on single-node CPU hardware. Visualization is a big data problem – the chief challenge is accessing large memory efficiently and directly. CPU memory is cheap, plentiful and fast: a laptop CPU has more memory (16 GB) than even a high-end GPU (12 GB), and a large-memory workstation with 768 GB can be acquired for less than $10,000. New vis clusters commonly feature nodes with 256 GB, and "fat" nodes are capable of 1–6 TB. Directly visualizing large data on a single resource is attractive, but requires fast memory-efficient rendering techniques for the CPU. CPU ray tracing has proven a viable approach for particle data (e.g., [8, 15]), but previous methods used standard acceleration structures with high overhead or algorithms specific to older SIMD architectures. General-purpose ray tracers like Embree [30] offer performance and portability, but incur high memory cost and lack the ability to efficiently query data.

[1] *Intel Corporation*
[2] *Scientific Computing and Imaging Institute, University of Utah*
[3] *Argonne National Laboratory*
[4] *Northern Illinois University*
[5] *Member, IEEE*

Manuscript received 31 Mar. 2014; accepted 1 Aug. 2014; date of publication xx xxx 2014; date of current version xx xxx 2014.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

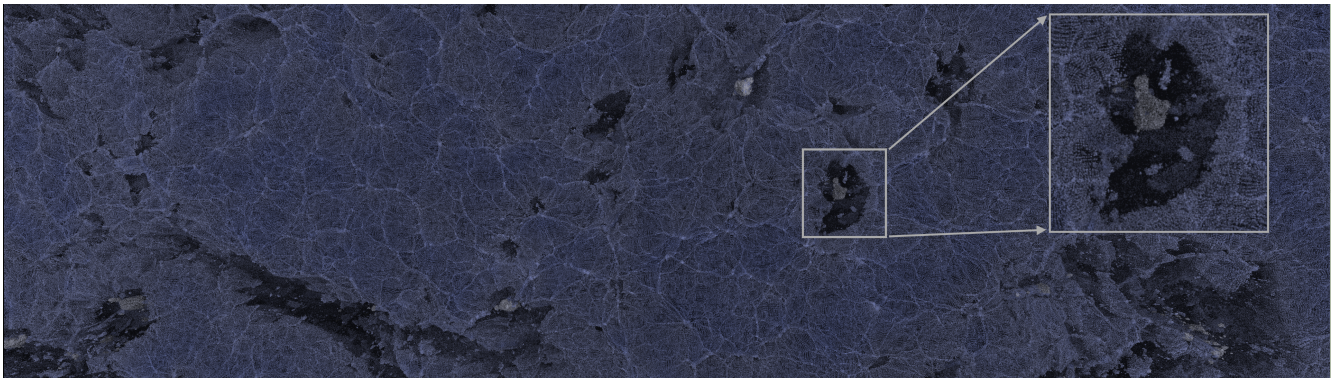

Fig. 2. Half-4K (3840x1080) rendering of the 432 GB Cosmic Web 8^3 dataset (29 billion particles), rendering at 6.4 frames per second (fps) with ambient occlusion – 1 sample per pixel (spp) with progressive refinement – on a 72-core 2.5 GHz Xeon E7-8890 v3 with 3 TB RAM.

In this paper, we describe a novel approach for visualizing large particle data sets using a ray tracing acceleration data structure based on the original balanced k-d tree [2]. Our approach rearranges data in-place into an acceleration structure that requires *no additional memory footprint*. Balanced k-d trees are different from the spatial k-d trees (BSP trees) commonly used in ray tracing, and to the best of our knowledge we are the first to employ this sturcture in directly ray tracing particle data at this scale. Moreover, our technical contributions are augmenting the balanced k-d tree into a *P-k-d tree* supporting additional range queries and efficient traversal and classification in a packet ray tracing environment. We achieve performance competitive with existing BVH ray tracing approaches, while requiring significantly less memory. We implement our approach in the OSPRay visualization framework [21], and use it to ray trace billion-particle data sets (Figures 1 and 2) on commodity workstations.

2 Background

2.1 Particle and Point Visualization

Efficient rendering of point data has been a popular topic in graphics (see, e.g., [9]). In this paper, we are principally interested in visualizing *volumetric* particle data from molecular dynamics and other Lagrangian simulations. Points fill interior space as opposed to defining a surface, and have one or more attributes, e.g. atom type, temperature, etc. Approaches for rendering such data vary, but can roughly be categorized into glyph, volumetric and implicit surface approaches.

Glyph techniques have long been explored on both GPU and CPU. Most relevant to our work, Gribble et al. [8] employ coherent ray tracing algorithms on the CPU to efficiently render millions of opaque sphere glyphs. Knoll et al. [16] employed BVHs in a predecessor of OSPRay for ray tracing megascale ball-and-stick models on CPU and Xeon Phi architectures. Megamol [11, 10] uses a combination of GPU rasterization, ray casting of sphere impostors, and image-space filtering to efficiently render millions of atoms. More recently, Le Muzic et al. [17] demonstrated a dynamic LOD system for rendering up to 10 billion atoms at 10 fps. With LOD, the system effectively renders on the order of 10^7 primitives on-device. They used this approach for real-time molecular animation, procedurally deformed with Brownian motion and user interaction. This approach works principally because the GPU controls both level-of-detail and movement of particles; it would likely face IO challenges if applied to large-scale, time-varying simulation data. In our work, we propose a CPU-based solution to rendering full-scale data, without LOD.

Volumetric approaches to rendering large point-based data vary widely. With LOD, systems have proven capable of rendering billions of particles; Fraedrich et al. [6] implemented an extremely fast out-of-core LOD particle renderer for real-time rendering of astrophysics data. However, image-space reconstruction is insufficient to reconstruct smooth surfaces classified from volume data. For that, one generally has the choice of resampling particle data onto a grid, or employing direct SPH/RBF volume rendering. Kähler et al. [14] demonstrate the former, using an octree to simultaneously splat particle data (simplified using LOD) and volume-render approximated data on a structured grid. Fraedrich et al. [5] dynamically resample from an octree into perspective-space uniform grids of predetermined size, and achieve nearly interactive performance on an NVIDIA 280 GTX for up to 42M particles (0.1 fps). Orthomann et al. [20] describe a similar system traversing an octree, using "packets" of rays computed on the GPU. Reda et al. [24] use the GPU to efficiently volume ray cast ball-and-stick glyphs, structured volumes, and RBF volume data. Knoll et al. [15] demonstrate direct RBF volume rendering on the CPU and Xeon Phi at interactive frame rates, using a BVH which incurs significant memory overhead.

Extracting surfaces from point data has been common in the molecular vis community, starting with the solvent-accessible surfaces of Connolly [3]. Wyvill et al. [31] employed the first implementation of marching cubes in polygonizing implicit surfaces from "blobby" RBF objects. For visualization, Navrátil et al. [19] use marching cubes to extract single iso-surfaces from multi-field cosmological data. Stone et al. [26] implement CUDA-accelerated iso-surface extraction from Gaussian RBF fields for fast computation of molecular surfaces. Though efficient, these approaches would sacrifice reconstruction quality and limit opportunities for dynamic classification.

Lastly, the astrophysics and cosmology communities have several tools for parallel batch visualization of particles [23, 4, 27, 1]. Generally, these do not take advantage of SIMD, have limited if any GPU acceleration, and are not designed for interactive rendering.

Irrespective of the type of data, rendering large numbers of particles can also be seen as a special case of more general large-model rendering techniques, for which we refer interested readers to state-of-the-art papers on CPU [30] and GPU [7] approaches, as well as more recent work involving large-model ray tracing on GPUs [18].

2.2 Ray Tracing Acceleration Structures

Ray tracing describes an entire family of algorithms that solve for the intersection of rays with geometric primitives, and the transport of light (or other properties) across a spatial media. At its core, ray tracing relies on spatial data structures, or *acceleration structures*, such as grids, binary space partitioning (BSP), (spatial) k-d trees, or bounding volume hierarchies (BVHs) (see Figure 3a-b).

Bounding Volume Hierarchies (BVHs, Figure 3a) are *object hierarchies* that store the bounds of all enclosed primitives in each node. Inner nodes specify tree topology; leaf nodes store primitives. Each primitive is referenced in exactly one leaf node, and nodes can spatially overlap. BVH trees are generally highly *unbalanced*.

Spatial k-d trees, in contrast (see Figure 3b), rely on hierarchical *space partitioning*. Spatial (subdivision) k-d trees are just a special case of more general *binary space partitioning (BSP)* trees, in which

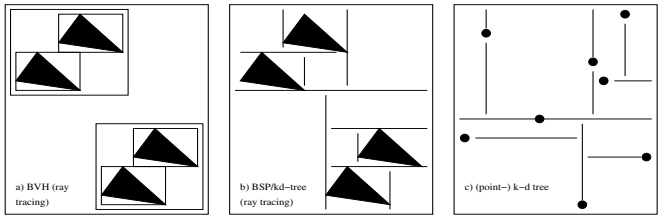

Fig. 3. BVHs (left) and k-d trees (center) as used in ray tracing to hierarchically organize geometric primitives. Right: Balanced point k-d trees (such as our P-k-d tree) encode k-dimensional points. Point k-d trees (right) and ray tracing k-d trees (center) share similarities, but also have important differences (see Section 3).

split planes are axis-aligned but may be placed anywhere in space. Inner nodes of a spatial k-d tree specify axis-aligned partitioning planes that recursively subdivide space; leaf nodes store references to geometric primitives. Nearly all "k-d trees" in ray tracing, collision and graphics are of this variety.

Range Trees BVHs, spatial k-d trees, etc can all be augmented with additional information to store, for each node, the min/max ranges of the attribute values associated with primitives in this sub-tree; this can then be used to, interactively reject sub-trees outside of a given parameter range (see, e.g., [28, 8]) These structures are often called min-max trees or interval trees.

2.3 Balanced k-d Trees

In contrast to spatial k-d trees used in ray tracing, in this paper we build upon the original *balanced* k-d trees of Bentley [2], illustrated in Figure 3(c). Spatial k-d trees and balanced k-d trees sound similar, and both subdivide space via axis-aligned split planes. However, they are fundamentally different data structures with very different properties. In particular, balanced k-d trees are explicitly designed to store *points* (not geometric primitives), which make them not immediately applicable to ray tracing. They do, however, have some interesting properties, specifically that a balanced k-d tree is:

(i) complete: all but the lowest levels of the tree are filled,

(ii) left-balanced: all nodes are on the left side of the tree, and

(iii) pointer-less: children of node i live at $2i+1$ and $2i+2$

These allow balanced k-d trees to encode a spatial hierarchy *simply by reordering the original data,* requiring no memory at all other than for the particle positions themselves [2]. We discuss the properties of k-d trees, and our variant the *p*-k-d tree, in greater detail in Section 3.

2.4 ISPC, Embree, and OSPRay

Our implementation make use of several open-source projects that influence how exactly our technique was implemented.

The Intel® SPMD Program Compiler (ISPC) [22, `http://ispc.github.io`] is an open-source Single-Program Multiple Data (SPMD) compiler that performs vectorization by mapping different instances of a scalar program to different vector lanes. ISPC is similar in spirit to other SPMD languages like OpenCL or OpenMP 4.0, but is designed to better allow for using CPU-like programming models. ISPC supports all major CPU instruction set architectures (ISAs), including Intel® Streaming SIMD Extensions (SSSE), Intel® Advanced Vector Instructions (AVX, AVX2, and AVX-512), and the Intel® Many Core Instructions (IMCI) used by the current version of Intel ® Xeon Phi ™ processors.

Embree [30, `http://embree.github.io`] is a high-performance kernel framework for efficient CPU ray tracing. It offers a set of low-level kernels for building and traversing ray tracing data structures that are particularly optimized to exploit modern CPUs' vector instruction sets through highly optimized, hand-tuned kernels. Embree allows for using user-defined primitive types (e.g., spheres), but handles all acceleration structure kernels (e.g., type of BVH) internally and opaquely.

OSPRay (`ospray.github.io`) is a ray tracing based rendering engine for high-fidelity visualization. OSPRay builds on both Embree and ISPC, using Embree for everything related to tracing rays, and ISPC for everything involving rendering and shading. OSPRay generally achieves interactive performance even on a single laptop or workstation (depending on the number of rays traced per pixel) and supports effects such as shadows, reflections, transparency, or ambient occlusion. Improving on Embree, OSPRay allows for new shading models and user-defined data representations and acceleration structures.

2.5 Challenges in Ray Tracing Large Particle Models

Initially, when we set out applying ray tracing to particle visualization we did not expect this would require anything new; we initially took the OSPRay ray tracing engine (which internally builds on Embree), used Embree's *user defined geometry* functionality to add a sphere primitive to OSPRay. This enabled us to render complex models with many millions of particles, but soon revealed two major issues.

The first issue is the desire to interactively color-map and discard (i.e. query) particle data based on range or other attributes. This is impossible with the standard Embree BVH. Simply discarding deactivated particles during traversal is performance-prohibitive when large regions of particles are inactive; they are traversed but never actually intersected. One solution is to use min-max trees [28, 15]; but these cannot easily be realized within the Embree framework. We required a new data structure and traversal mechanism for efficient query.

The second problem is the memory overhead required to handle the ray tracing acceleration structures, in particular if the memory required for each primitive itself is relatively small. For example, using sphere glyphs in Embree's *user-defined geometry* requires 16 bytes for the primitive and 32 bytes per BVH leaf, leading to an overhead of up to $7\times$ for leaves with one primitive each. While copious CPU memory encourages waste if performance can be gained, in many cases inefficiency can mean the difference between being able to handle a data set or not. Also, in an *in situ* visualization context, the memory overhead required for rendering would leave only a fraction of memory for the actual simulation.

3 Ray Tracing using Balanced P-k-d Trees

In this paper, we propose a novel approach to efficiently handle large particle data in a ray tracer—in particular taking into account memory consumption and construction time of the acceleration structure. We apply this technique to interactive visualization of large particle data sets using the OSPRay CPU ray tracing framework. Given the challenges outlined in Section 2.5, we need a way of ray tracing large numbers of particles that:

- allows large numbers of particles to be rendered without losing ray tracing's ability to scale logarithmically in model size,
- has significantly lower overhead than $7\times$; and
- achieves competitive, interactive performance.

We take an approach based on balanced k-d trees which achieves these goals. Our method works by representing *each* primitive i via a single representative point x_i (in the case of spheres, their centers), and computing the maximum spatial extent R_{max} that *any* primitive deviates from this point (in the case of spheres, the maximum radius of any sphere). We then organize these points in a balanced "point"-k-d tree which has several nice properties for our particular application – in particular memory consumption – and devise a novel ray traversal scheme that, given the point-k-d tree and R_{max} can efficiently ray trace the primitives encoded in this tree. Our P-k-d tree is thus a balanced k-d tree with several modifications to enable efficient rendering of particle data, namely:

- we use maximum extent [13] instead of round robin [2] to pick split planes, we encode the split axis in the tree data itself
- we re-arrange particle attributes into separate balanced trees, so they may be queried separately
- we preprocess attributes into (optional) min-max trees for faster query, incurring only modest extra cost of 1–2 bytes per particle.

3.1 Balanced vs. Spatial k-d tree Traversal

As discussed in Section 3, balanced kd-trees are quite different from spatial k-d trees commonly used in ray tracing. This has important consequences for ray traversal.

In spatial k-d trees, splitting planes are used to separate different primitives that all have a spatial extent. Different sub trees' spatial extends do not overlap, and primitives will often be referenced in multiple leaf nodes. In the spatial k-d tree, inner nodes correspond to splitting planes and store a plane description; leaf nodes correspond to regions of space and store (references to) primitives. Being able to place partitioning planes at arbitrary locations allows the spatial k-d tree a wide variety of construction options [12]. Properly built spatial k-d trees can enclose their geometric primitives quite well, but require significant memory overhead for interior nodes, and pointers for both tree topology and primitives.

Balanced k-d trees, in contrast, always and exclusively encode *points* (not 3D primitives), and place planes always right *through* the data points. When building the k-d tree such that the resulting tree is binary, complete, and left-balanced, balanced k-d trees can be encoded in completely pointerless fashion: the children for the node n_i are always n_{2i+1} and n_{2i+2}, respectively, and a child exists if and only if its

index i is less than the number of points N. In each node n_i, the partitioning plane's dimension d_i is defined implicitly through the node's tree level $L_i = floor(log_2(i))$ (as $d_i = L_i \% k$, where k is the number of dimensions), and the plane's position—going through n_i then is x_i, d, where $x_i, i \in [0..k)$ are the k coordinates of point i. Stored in that way, a balanced k-d tree can encode N k-dimensional points with exactly zero memory other than for the points themselves.

The drawback of the balanced k-d tree is that a node does not partition its primitives into *exclusively* left and right sides, but instead *also* contains a primitive at each inner node itself. These inner nodes must be intersected by every ray entering this sub-tree, usually unsuccessfully. The root node, for example, is guaranteed to be intersected by almost every ray traversing the data structure (though early termination may avoid its traversal). This data structure would be impractical for geometry whose intersection tests are expensive. However, for visualization of spheres, the cost of unnecessary intersection is low.

3.2 Naïve Balanced P-k-d Tree Traversal

In order to efficiently allow 3D object queries such as ray traversal a data structure has must allow for recursive traversal that quickly rejects sub-trees that do not intersect the query (e.g., a ray, frustum, range, etc). A spatial k-d tree style traversal does not easily work for the P-k-d tree data structure; the balanced k-d tree is in fact closer to an *object* hierarchy (like a BVH), where each node partitions its objects into disjoint sets of those objects on the left, those on the right, and the root node—and even though the representative point of an object may lie to the left of the root node's plane it is not guaranteed that the entire object is exclusively on that side.

Given a maximum radius R_{max} we can compute conservative bounding boxes for its two children as depicted in Figure 4. Offsetting the node x's plane P_x into $+R_{max}$ and $-R_{max}$ yields two implicit planes $P_{x,lo}$ and $P_{x,hi}$ that, when clipped against the bounding box for the given sub-tree, yields two bounding boxes that can easily be guaranteed to conservatively bound all the primitives in that sub-tree. This then leads to the simple recursive traversal depicted in Algorithm 1.

```
function recurse(Object Q, Node n, box aabb)
  if (Q does not intersect aabb)
    return;
  Q.process(n);
  box aabb_l = aabb.clip(n.Plane + Rmax);
  box aabb_r = aabb.clip(n.Plane - Rmax);
  if (has left child)
     recurse(Q,leftChild,aabb_l);
  if (has right child)
     recurse(Q,rightChild,aabb_r);

function traverse(Object Q, K-D tree tree)
  box aabb_root = tree.bounds().extend(Rmax);
  recurse(Q,tree.root,aabb_root);
```

Algorithm 1: A naïve recursive traversal algorithm applicable for 3D object query (ray, frustum, range, etc.).

3.3 Optimized Ray Traversal

Several optimizations to such a naïve algorithm can significantly improve performance. In particular, for ray traversal we want to traverse as much as possible in front-to-back order, in order to maximize the likelihood of skipping entire subtrees once a close hit is found. Like in a spatial k-d tree, we can actually guarantee a front-to-back traversal using only local traversal decisions. However, unlike in a spatial k-d tree, we can *not* early-terminate a ray upon the first found intersection:

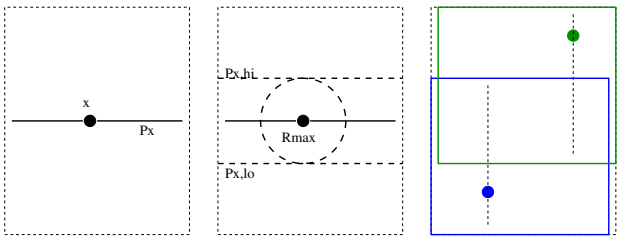

Fig. 4. Recursive traversal of a balanced k-d tree assuming each primitive is represented by a point, and bounded by a sphere with a radius of, at most, R_{max} . Given a bounding box known to enclose all the primitives in a sub-tree, the plane P_x associated with the subtree's root node x implicitly defined two planes $P_{x,lo}$ and $P_{x,hi}$. Clipping the subtree's bounds against those two planes gives two (conservative) bounding boxes for the left and right child trees, allowing for recursive traversal.

since sub-trees can overlap, our data structure and traversal are actually closer to a BVH, and other sub-trees may still contain a possibly closer intersection.

Another important observation to make is that when traversing from one node to any of its children, only one of the six box sides will actually change. Thus, performing full ray-box intersection tests would redundantly re-compute 5 of 6 values every time. Instead, we can adopt an idea from spatial k-d tree traversal that, rather than tracking the bounding box itself, instead tracks only the ray interval that overlaps the current box, and incrementally changes that based on the distance to the respective plane(s). We do not employ actual recursion, but emulate the recursion using a manually-maintained node stack as commonly done for both BVHs and spatial k-d trees. In addition, we add some logic for traversal of the min-max range trees, optionally computed for each particle attribute we wish to query (described in Section 3.6.2). The final pseudo-code is given in Algorithm 2.

```
function traverse(Ray R, K-D tree tree) {
  box aabb_root = tree.bounds().extend(Rmax);
  (t_in,t_out) = clip R to aabb_root;
  if (t_out >= t_in) return DONE;
  Node n = tree.root;
  while (true)
    while (true)
      if (node is leaf)
        intersect particle n; break;
      // particle range culling:
      if (n out of valid range)
         break;
      // compute dist to near and far plane
      t_lo,t_hi = distance to P_lo,P_hi
      // node IDs for near/far child
      k = split_dim(n)
      s = sign(R.dir[k]);
      (n_nr,n_fr) = (2n+2-s,2n+1+s)
      // entry/exit dist for nr/fr child
      t_fr_in  = max(t_in,min(t_lo,t_hi))
      t_nr_out = min(t_out,max(t_lo,t_hi))

      if (t_in > t_near_out)
        // entire [t_in,t_out] on far side
        n = t_far; continue;
      if (t_fr < t_far_in)
        // entire [t_in,t_out] on near side
        n = t_near; continue;
      push(n,n_far,t_far_in,t_out)
      (n,t_out) = (n_nr,t_nr,out)

    // pop from stack:
    while (true)
      if (stack is empty) return;
      (n,n_far,t_in,t_out) = pop();
      if (ray.t_hit < t_in) continue;
      intersectPrim(n);
      break; // go on traversing
```

Algorithm 2: Optimized algorithm for traversing rays through a P-k-d tree, including attribute range selection. The algorithm is a hybrid between spatial k-d-tree and BVH traversal.

3.4 Handling Particles With Different Radii

Though all our data sets use a fixed radii for all particles, it would be possible to also support different radii (for example, by storing a radius per particle, or by deriving a particle from a mapped attribute), and even non-spherical shapes such as balls-and-sticks, triangles, etc. All the P-k-d tree needs to guarantee correctness is a *conservative* R_{max} value that, when used to shift a subtree's planes, properly bounds all primitives in that subtree. The *tightness* of the bounding primitive (i.e., how tightly the sphere with radius R_{max} bounds the actual primitive) will impact traversal performance. In cases where a handful of large particles are mixed with many tiny particles, performance will suffer. There are ways of addressing this (e.g., storing a maximum radius per sub-tree); we leave them outside the scope of this paper.

3.5 Ray Tracing and Shading

By implementing the P-k-d traversal routine within OSPRay [21], we are automatically able to use the material, rendering and shading pipeline of that ray tracing engine. When a ray terminates in traversal, the OSPRay renderer is given a geometry ID (a pointer to the particle), from which it can look up the material via the chosen attribute and transfer function. This material is then passed to the chosen OSPRay renderer (ray cast, ambient occlusion, path tracing, etc.), which integrates the color accordingly and generates secondary rays as necessary. Like Embree [30], OSPRay allows for progressive refinement an option, ensuring consistent interactivity and allowing path-traced images to converge to production-quality renderings. Examples of

diffuse-only ray casting, ambient occlusion and path tracing are shown in Figure 5.

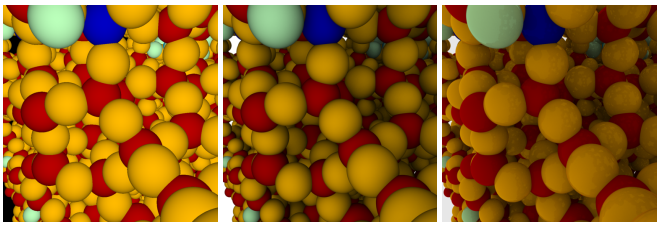

Fig. 5. Rendering modalities, illustrated on a 3500-atom *zeolite* structure. Left to right: ray casting (106 fps), ambient occlusion (5 fps at 16 spp; 45 fps at 1 spp with progressive refinement) and path tracing (0.041 fps at 512 spp; 18 fps with progressive refinement).

3.6 Tree Construction

Generally, balanced k-d trees rely on a dimensional sort, and pick the literal median element as the pivot point. Unlike spatial k-d trees [12], they offer no flexibility in placing split planes: once the split dimension has been chosen, the balance of the tree dictates exactly which particle along that axis has to be the root node. Nevertheless, there are a variety of choices in particular with respect to data layout that we want to briefly discuss.

3.6.1 Round-Robin vs Maximum-Extent Partitioning

Traditional balanced k-d trees [2] chose the partitioning dimension in a round-robin (RR) manner, in which case each node's dimension is implicit in the node's depth in the tree. As shown by Jensen [13], it is often advantageous to instead partition along the axis of the current subtree's maximum extent, and since such a maximum extent (ME) splitting scheme will minimize the surface area of the two child nodes, this will also be advantageous for ray traversal. Generally, we found that ME splitting gave a 30% performance advantage over round robin.

Maximum-extent partitioning also simplifies our algorithm, as we no longer have to track the tree depth on the stack. However, we now must store the chosen split axis. We currently squeeze this two-bit information into the particle position, i.e. the lowest two bits of the x, etc. Alternately, one could employ unused bits of the min-max tree, or of the atom type attribute, etc. In the worse case, one could store these bit explicitly in an separate array, requiring two additional bits per particle.

3.6.2 Range Trees, Queries and Multi-Attribute Data

The balanced P-k-d tree is different from standard balanced k-d trees in that it is designed for volumetric particle data with queryable attributes. Our goal is to efficiently traverse the tree and cull unwanted branches based on a transfer function or other range query. This is useful, for example, in materials science when isolating atoms of one or more types, or in cosmology to filter out low-density particles to better reveal structure (Figure 6).

In the P-k-d tree, each attribute is its own array of attribute values. Attributes are ordered in the same way as particles, i.e. for given attributes M, D, and V, the V value for particle i is stored at pkd.attribute[V].value[i]. Range trees are built on top of attributes, and traversed alongside the P-k-d tree as in Algorithm 2. To build the range tree, we first build the P-k-d tree, then simply compute min-max information of the component attributes. To store the min-max tree, we currently use one integer per inner node of the tree, which gives us a 32-bit mask of which attribute values are present in the given sub-tree. While this adds some overhead, it is typically small compared to the size of the attribute data in the inner nodes; moreover one mask suffices for multi-attribute data. This mask is computed as a preprocess every time the transfer function changes. While the added cost of the mask is relatively small (13%), it is purely optional; the user can traverse all data without culling sub-trees.

3.6.3 Construction Algorithm

For actual tree construction we use an in-place partitioning scheme inspired by the well-known *quick-sort* algorithm. The method proceeds as follows: first, using either round-robin or maximum extent, we pick the axis on which to sort. Then, using the current root particle as a pivot we iterate through left and right sub-trees (in heap indexing), to find "wrong" particles in the left and right sub-tree; i.e., a particle on the left that is larger than, and a particle on the right that is smaller than, that pivot. If these exist (at, say, positions i and j) we swap these two particles, and continue scanning at $i+1$ and $j+1$.

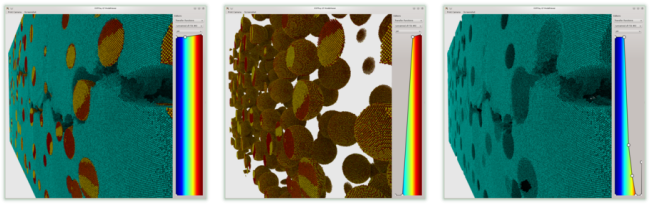

Fig. 6. Attribute-based query, based on atom type, in the Al_2O_3-SiC fissure data set. Left to right: full data set; silica carbide particles only; indentations in the alumina.

If a wrong particle could only be found on one side—say, i on the left—then i becomes the new pivot by swapping with the root, and we again search for wrong nodes in both subtrees (but noting that from now on, we will no longer have to scan any earlier than i on the left); the right-side case is analogous. If no wrong particle could be found on either side, then the current root is the proper pivot, the tree is properly partitioned in that node, and we can recursively build its children.

For thread parallelization, we fork a new thread to handle the left sub-tree when sub-trees contain more than a certain number of (currently, 16K) elements. Though this strategy does not achieve perfect scalability, it works reasonably well for larger data for which scalability is needed most, e.g., delivering around 70% scalability to 16 threads on a 16-core 2.7 GHz SandyBridge CPU (Table 5).

The P-k-d tree is currently built in a pre-processing step when we convert from the external input file formats to OSPRay's internal XML-based binary data format; this saves re-building the tree every time a model is loaded, and since our data structure is very compact there was no obvious reason not to store readily built trees.

3.7 OSPRay Implementation

We implemented our P-k-d method in the OSPRay [21] framework. With its object-oriented design, all ray-intersectable geometric objects are derived from a common, abstract `ospray::Geometry` type. We subclass this into a new ospray `PKDGeometry` type that performs the ray traversal, range culling, and particle intersection. This allows all other components of OSPRay—camera, materials, renderers, parallel rendering mode, etc. — to work "out of the box", requiring no new implementation.

OSPRay internally implements all objects in a hybrid C++/ISPC fashion, where components that aren't performance critical (bookkeeping, reference counting, etc) are implemented in C++, while performance-critical traversal routines are implemented in ISPC; the interface to all such functions—ray generation, traversal, intersection, shading, etc—is based on *varying* rays (i.e., they internally operate on N rays in parallel, one per vector lane, where N is the architecture's vector width). A naïve implementation of Algorithm 2 in ISPC would—just like in other SPMD languages like OpenCL or CUDA—lead to each lane simply executing its own independent traversal, with each lane having (and maintaining) its own stack, traversing its own path through the tree, typically traversing different nodes and intersecting different particles, etc. However, as ISPC allows a programmer to explicitly specify which data is *uniform* (i.e., scalar) vs which is *varying* (i.e., once per vector lane) it is relatively easy to also implement a "packet" traversal in which the different rays remain ganged together (see e.g., [29]), and either all rays skip a sub-tree, or the entire packet enters this sub-tree (even those rays that would not have needed to). Despite worse SIMD utilization, packet traversal is consistently faster: The naïve SPMD implementation is limited by the the efficiency of gathering of up to N different particles (in general very different memory locations) in each traversal step. Moreover, stack operations and masked traversal logic, executed in *uniform*, are far simpler for the packet code. For the 180 million atom *Uintah* data set, for example, the packet code is roughly $2\times$ faster than the naïve SPMD method for primary rays, and even $3\times$ faster for ambient occlusion rays. We thus use the packet variant in all our experiments.

4 RESULTS

We evaluate our implementation on several different hardware platforms enumerated in Table 1. These cover a wide range of machines from personal laptops to high-end workstations and HPC nodes. In all cases, we only use the systems' CPUs, the GPUs (where available) are only used to display the final rendered image. Unless otherwise noted, all benchmarks are performed at 1024×1024 (1 megapixel) resolution, on the 72-core Intel®Xeon®E7-8890 v3 workstation with 3 TB RAM, using ambient occlusion with 1 sample per pixel and progressive refinement.

name	#CPUs	cores	memory
Laptop (Macbook Pro)	Core i7	4 x 2.7GHz	16GB
Xeon Phi Workstation (CPU)	2x Xeon E5-2650	16 x 2.7 GHz	64GB
Xeon Phi Workstation (Phi)	Xeon Phi SE7110P	61 x 1.24 GHz	16GB
Xeon E7-8890 v3 Workstation	4x Xeon E7-8890 v3	72 x 2.5 GHz	3 TB

Table 1. Hardware used to evaluate P-k-d performance.

4.1 Overall Performance

In Tables 2 and 4, we evaluate our technique by comparing a range of data sets from 740 thousand atoms to 29 billion particles on the 72-core Xeon E7-8890 v3 workstation. Overall, we achieve solidly interactive results for ray casting (29–135 fps) and both ambient occlusion (24–90 fps) and path tracing (5–33 fps) at 1 sample per pixel with progressive refinement rendering. More expensive ray tracing modalities (ambient occlusion at 16 spp, path tracing at 64 spp) perform non-interactively, but at rates suitable for efficient batch rendering. We note that performance is only loosely dependent on data size; the structurally similar SiO_2 and Al_2O_3-SiC *fissure* simulations perform at nearly identical frame rates despite a factor of 20 difference in data size. We note similar behavior for the *Uintah* data sets.

4.2 Memory Consumption Relative to the Embree BVH

At the bottom of Table 3, we compare our P-k-d technique with the standard Embree [30] quad-BVH in OSPRay. The BVH is built directly on top of a single sphere per particle/atom; Embree chooses its own optimal number of primitives per leaf node (generally, 1). With an Embree quad-BVH at default (i.e., performance-optimal) settings (1 primitive per leaf), the total memory overhead is over 5 ×; however performance is on average 1.48× faster than our P-k-d tree. For far views, this difference is not as great as for close views (7% slower vs 43% slower); and in some cases the P-k-d traversal is actually *faster*. This is particularly interesting given that Embree uses hand-tuned, low-level SIMD routines for BVH traversal (the ray-sphere intersection is the same in both variants), while our traversal is, so far, coded exclusively in ISPC, thus leaving potential for low-level optimizations.

Embree by default uses a single primitive per leaf. In principle, this leaves another, simpler, way of reducing memory consumption by adopting shallower BVHs with more primitives per leaf. To quantify this effect we modified our version of Embree to use different threshold for leaf generation (up to a maximum of 8, which is the smallest that Embree can internally encode), and measured both performance and memory consumption of the resulting BVH relative to our P-k-d tree. As seen in Table 3, using the shallower BVH can indeed cut Embree's memory overhead by half, but it is still 2.7× larger than the P-k-d tree's. Even shallower BVHs would reduce this overhead further; however, we already achieve performance parity at a leaf threshold of 8 (see Table 3). For shallower BVHs, our P-k-d tree would thus have *still* lower memory consumption, and *also* yield higher performance.

We note that the Embree BVH is built only around raw particles; it does not encode any range information. Prior to implementation of the P-k-d tree, we experimented with standard range trees in a binary BVH, both performance and memory proved worse with this approach than with the P-k-d tree; generally by about 2× and 3×, respectively.

4.3 Range Tree and Query Costs

From Table 2, we see that range trees cost a relatively minor 13% of the original data size. However, the time to compute the range tree is $O(N)$, which is interactive for megascale data but can take on the order of minutes for 29 billion CosmicWeb dataset. Once computed, the range tree accelerates traversal of a subsets of data by skipping entire P-k-d subtrees. Frame rate is generally slower when performing these queries than when rendering full data, due to the extra branching and worse overall cache behavior. When changing the transfer function, in addition to the time required to rebuild the range tree, one may see temporary performance hits of 2x, which recover quickly as cache is filled appropriately. This behavior varies with data set and chosen classification. While it falls outside the scope of our work in this paper, it would be worth further investigation.

4.4 Build time

Table 5 shows scalability for several data sets, and Table 4 shows construction time for our reference data sets. We achieve roughly 70% scalability up to 16 threads; while we did not explicitly measure the time to build larger data sets we anticipate their scalability would be comparable. Overall, P-k-d construction times are small compared to the time to read from disk. Surprisingly, for the data we tested all build times were slightly lower than those of the Embree BVH.

data set	# particles	1 core	16 cores	scalability
nanosphere	740K	220ms	50ms	28%
SiO_2	5M	1.75s	330ms	33%
CosmicWeb 1^3	51M	41.6s	3.6s	72%
Al_2O_3-SiC	100M	80.4s	7.2s	70%

Table 5. Build time, and scalability up to 16 cores, for select data sets measured on a 2.7 GHz 16-core E5-2650 workstation with 64 GB RAM.

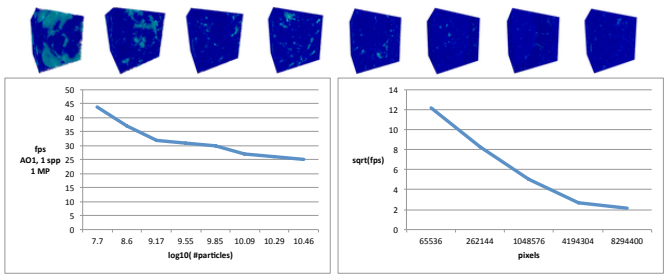

Fig. 8. Top: CosmicWeb subsets from 1^3 to 8^3. Left: frame rate vs $\log_{10}$(data size). Right: square root of frame rate vs number of pixels.

4.5 Scalability to Data and Image Size

In Figure 8 (left), we examine the scalability of our algorithm to data size, rendering varying subsets of the CosmicWeb data from 1^3 (51 million particles) up to 8^3 (29 billion particles) on the 4-socket, 72-core Xeon E7-8890 v3 workstation. We used default camera positions that render the full extents of the particle volume into a full-screen window. As expected, performance varies linearly with logarithm of data size, with some drops in performance reflecting NUMA.

In Figure 8 (right), considering performance with respect to frame buffer size, we plot number of pixels against the square root of frame rate, and find a mostly linear curve, rendering the 51M Cosmic 1^3 subset. Effective throughput increases as we process larger frame buffers, likely due to improved memory coherency.

4.6 Comparison to GPU Techniques

There is no readily-available GPU ray tracer for large molecular and particle data with which to compare. State-of-the-art GPU methods [10] employing OpenGL-rasterized impostors are capable of real-time performance – for kiloscale or megascale data these would be faster than our ray tracing approach, particularly on laptop or tablet CPUs. To render gigascale and larger particle data on the GPU, one must use simplification, out-of-core approaches, or distributed-parallel rendering. Geometric simplification methods on the GPU [17] have handled up to 10 billion (simplified to 200 million) atoms at 10 fps at 2 MP, combining illustrative rendering with screen-space ambient occlusion; the p-k-d technique on our workstation exhibits roughly comparable performance (10–20 MRays/sec) on the *full, unsimplified data*. Out-of-core LOD methods (e.g., [6]) are capable of even greater performance, but more are difficult to explicitly compare to. In more recent work involving parallel compositing on the GPU [25], the authors render 32 billion particle imposters without LOD on a 128-GPU cluster in roughly 3 seconds for an 18 MP image (6 million primary rays/second). With a LOD strategy picking 10% of the full number of particles, they achieve linearly better performance (60 MRays/second). While comparison between splatting transparent imposters and ray

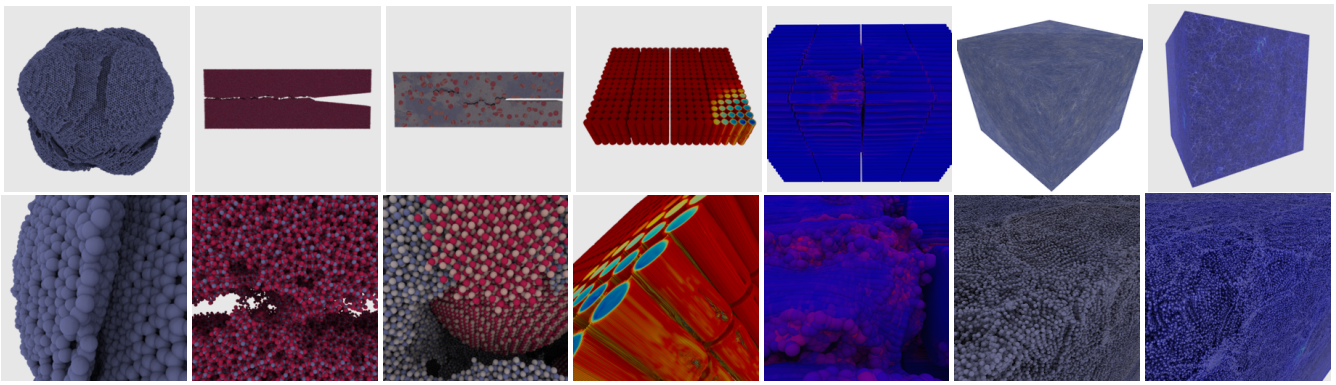

Fig. 7. Reference scenes (ambient occlusion, 16 spp). From left to right: *nanosphere* (740K), SiO_2 *fissure* (5M), $Al_2O3-SiC$ *fissure* (105M atoms), *Uintah* detonation (180M particles), large *Uintah* detonation (1G particles), *CosmicWeb* 6^3 (12.2G particles), *CosmicWeb* 8^3 (29G particles).

Dataset		nanosphere	SiO_2	Al_2O_3-SiC	Uintah180M	Uintah1B	CosmicWeb6^3	CosmicWeb8^3
	#particles	740K	5M	105M	180M	.97G	12.2G	29.0G
	#time steps	1	121	400	188	72	-	-
	P-k-d = raw data size	11.3 MB	73 MB	1.58 GB	2.67 GB	14.4 GB	182 GB	432 GB
	Range tree (optional)	1.4 MB	9.1 MB	.20 GB	.336 GB	1.8 GB	23 GB	54 GB
	P-k-d build time	220 ms	575 ms	6.6 s	10.2 s	110 s	1700 s	3900 s
ray cast fps	1 spp	125 / 90	135 / 70	111 / 95	80 / 75	46 / 29	37 / 55	35 / 54
AO fps	1 spp	67 / 49	90 / 49	75 / 44	53 / 37	26 / 24	28 / 15	26 / 28
	16 spp	14 / 6.6	21 / 6.9	20 / 6.0	13.1 / 5.12	3.70 / 2.56	5.9 / 6.2	5.2 / 4.6
path tracing fps	1 spp	20 / 10.2	28 / 9.1	33 / 9.4	19.3 / 9.0	7.45 / 5.0	11.5 / 10.9	10.2 / 6.0
	16 spp	1.46 / .690	2.37 / .622	3.0 / .642	1.50 / .638	.520 / .222	.945 / .905	.830 / .505
	64 spp	.377 / .174	.591 / .157	.811 / .162	.385 / .160	.130 / .052	.220 / .215	.191 / .120

Table 2. Memory cost and frame rate for (far / close) views, on the 72-core Xeon E7-8890 v3 workstation, rendered at 1024x1024 pixels. We report numbers for ray casting, ambient occlusion (AO), and path tracing (PT, using a diffuse-metal material), at 1–64 samples per pixel (spp).

OSPRay using P-k-d tree (our technique), including range trees and parameter sub-range selection							
Dataset		nanosphere	SiO_2	Al_2O_3-SiC	Uintah180M	Uintah1B	average
	raw data=P-k-d size	11 MB	73 MB	1.7 GB	2.67 GB	14.4 GB	
P-k-d AO fps	1 spp	67 / 49	90 / 49	75 / 44	53 / 37	26 / 24	
OSPRay using Embree quad-BVH (Embree "best performance" mode - 1 particle / leaf)							
	qBVH size	58 MB	347 MB	8.8 GB	13.2 GB	71 GB	
	qBVH+particles	69 MB	420 MB	10.5 GB	15.8 GB	85 GB	
	qBVH build time(s)	226 ms	3.15 s	9.1 s	14.5 s	140 s	
	qBVH/P-k-d size ratio	5.2x	5.8x	6.2x	5.9x	5.9x	**5.8x**
BVH AO fps	1 spp	81 / 72	68 / 71	60 / 67	47 / 52	45 / 74	
BVH/P-k-d fps ratio		1.21x/ 1.46x	.76x / 1.45x	.80x / 1.52x	.89x / 1.41x	1.73x/ 3.0x	**1.07x / 1.76x (1.41x)**
OSPRay using Embree quad-BVH ("performance parity" at roughly 8 particles / leaf)							
	qBVH size	20 MB	120 MB	2.7 GB	4.8 GB	24.2 GB	
	qBVH+particles	31 MB	193 MB	4.4 GB	7.5 GB	38.6 GB	
	qBVH build time(s)	120 ms	4.0 s	7.5 s	10.1 s	52 s	
	qBVH/P-k-d size ratio	2.8x	2.6x	2.6x	2.8x	2.6x	**2.7x**
BVH AO fps	1 spp	70 / 62	72 / 55	70 / 67	53 / 55	28 / 30	
BVH/P-k-d fps ratio		1.04x/ 1.26x	.80x / 1.12x	.93x / 1.42x	1.0x / 1.48x	1.08x/ 1.25x	**.97x / 1.3x (1.14x)**

Table 3. Memory usage and performance of our P-k-d tree relative to an Embree quad-BVH built around raw particle data (no range trees). For Embree, we report data for both "best performance" BVH settings (targeting 1 particle per leaf) and for a "performance parity" setting in which Embree is about as fast as our P-k-d tree. At best performance Embree is slightly faster than our P-k-d tree, but at 5x memory overhead; at performance parity, it still has 2x overhead; memory parity is not possible (Embree always requires more memory).

Data Set	nanosphere	SiO_2	Al_2O_3-SiC	Uintah180M	Uintah1B
#particles	740K	5M	105M	180M	.96B
data/P-k-d size	11 MB	73 MB	1.7 GB	2.7 GB	36 GB
Xeon E7-8890 v3 (2.5 GHz, 72 cores, 3 TB)	67 / 49	90 / 49	75 / 44	53 / 37	26 / 24
IvyBridge laptop (2.7 GHz, 4 cores, 16 GB)	6.4 / 4.3	10.3 / 3.6	9.6 / 3.5	2.7 / 1.4	- / -
SandyBridge 2.7 GHz (2.7 GHz, 16 cores, 64 GB)	18 / 13	30 / 12	27 / 11	15 / 9.1	5.5 / 5.1
Xeon Phi 7110P (1.33 GHz, 61 cores, 16 GB)	14.3 / 12.5	27 / 13.5	20 / 10.1	14.5 / 8.3	- / -

Table 4. Performance for (far / close) views, in frames per second at 1024x1024 resolution for the reference data sets shown in Figure 7, on different CPU and Xeon Phi coprocessors. All results shown for ambient occlusion, 1 sample per pixel.

tracing (mostly opaque) sphere glyphs is not completely fair, we are able to achieve comparable performance for primary rays on our single 72-core workstation. Moreover, though current GPUs support up to 12 GB RAM per device (750 million particles), Rizzi et al. [25] suggest rasterization performance would drop below interactive rates at 50-200 million particles depending on the GPU and technique used. Though outside the scope of this work, ray tracing with the P-k-d structure may prove useful on future distributed-memory GPU architectures.

5 Summary and Discussion

We have presented a method for fast ray tracing of massive particle data on CPU architectures, with virtually no memory overhead. With this approach, a suitably fast CPU with enough memory can handle extremely large particle data, with no level-of-detail. Data size has relatively little impact on rendering performance: kilo-scale and giga-scale particle data exhibit similar frame rates. The data chosen in our experiments include some of the largest molecular and materials sim-

ulations data, and significant subsets of cosmology simulation runs. With the balanced P-k-d structure, we are able to handle 5× larger data than BVH methods, with performance (using a naïve ISPC implementation for our P-k-d tree) on average 67% that of state-of-the-art BVH ray tracing (using hand-tuned kernels). Our performance is competitive with state-of-the-art distributed-parallel, out-of-core and LOD-based GPU methods. Thanks to its implementation in OSPRay and ISPC, it can be deployed on a wide range of CPUs.

Some remaining problems merit discussion. Balanced k-d tree addressing leads to huge jumps in memory address when traversing up and down the tree (in particular close to the leaves), posing a challenge for any memory system. A hybrid structure, such as a separate grid or BVH containing P-k-d trees, may greatly improve performance at some small memory cost. At large scale, the time to read and write from disk remains the main bottleneck for both interactive and batch visualization. The time to build a P-k-d tree on a single machine is also non-trivial. While it would be possible to improve on our thread-parallel build, a better strategy might be a data-parallel approach in which different nodes build P-k-d trees over their own data, thus enabling distributed-parallel IO, distributed build of the acceleration structure, and interactive ray tracing – offline, in coprocessing or in situ. Alternately, one could pursue true in-memory in situ rendering, i.e., distributed parallel ray tracing on the compute resource.

Rendering up to billions of particles on millions of pixels means that thousands of particles can project to a single pixel. Though a ray tracer can handle this cost-wise, it creates challenges in terms of aliasing, and ultimately one must question whether glyph representation is the best way of visualizing such data. We believe the capability to visualize full particle data at high resolution (Figure 2) is compelling, particularly for production-quality still images. However, antialiasing solutions borrowing from LOD techniques would be desirable to remove artifacts and improve overall image quality, particularly during rapid camera movement and animation.

Lastly, the balanced P-k-d structure's optional range trees enable fast query and implicit classification of multi-attribute data. Simulations such as Uintah or CosmicWeb may have tens to hundreds of attributes. Choosing how to efficiently traverse and classify multi-field particle data, for example detecting halos or correlating combustion variables, could be of interest from an applications standpoint.

Acknowledgments

This research was supported by the Argonne Leadership Computing Facility under the U.S. Department of Energy, Office of Science, Office of Basic Energy (Award Number DE-AC02-06CH11357); by the NSF CISE ACI-0904631 at the University of Utah; and by the Intel® Parallel Computing Center program. The authors would like to thank Paul Navratil at Texas Advanced Computing Center for access to the CosmicWeb data. We thank the DOE INCITE program for access to data sets at ALCF, as well as Martin Berzins, Todd Harman and Jacqueline Beckvermitt at the University of Utah for the Uintah data sets; Ken-ichi Nomura, Aiichiro Nakano and Ying Li at University of Southern California for the SiO_2 and Al_2O_3-SiC data sets; and Kah Chun Lau and Larry Curtiss at Argonne National Laboratory for the Nanosphere. We also thank Mark West at Intel, and Janene Ellefson and Ryan Baxter at Micron, for their generous donations of hardware.

References

[1] G. Altay, R. A. Croft, and I. Pelupessy. SPHRAY: a smoothed particle hydrodynamics ray tracer for radiative transfer. *Monthly Notices of the Royal Astronomical Society*, 386(4):1931–1946, 2008.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[3] M. L. Connolly. Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221(4612):709–713, 1983.

[4] K. Dolag, M. Reinecke, C. Gheller, and S. Imboden. Splotch: visualizing cosmological simulations. *New Journal of Physics*, 10(12):125006, 2008.

[5] R. Fraedrich, S. Auer, and R. Westermann. Efficient high-quality volume rendering of SPH data. *IEEE Transòn Visualization and Computer Graphics (TVCG)*, 16(6), 2010.

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

[7] E. Gobbetti, D. Kasik, and S.-E. Yoon. Technical strategies for massive model visualization. In *Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 405–415. ACM, 2008.

[8] C. P. Gribble, T. Ize, A. Kensler, I. Wald, and S. G. Parker. A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Trans. on Visualization and Computer Graphics (TVCG)*, 13(4), 2007.

[9] M. Gross and H. Pfister. *Point-based graphics*. Morgan Kaufmann, 2011.

[10] S. Grottel. MegaMol Projektseite, Universität Stuttgart VISUS, 2012.

[11] S. Grottel, P. Beck, C. Müller, G. Reina, J. Roth, H.-R. Trebin, and T. Ertl. Visualization of Electrostatic Dipoles in Molecular Dynamics of Metal Oxides. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 18(12):2061–2068, 2012.

[12] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech TU in Prague, 2001.

[13] H. W. Jensen. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001. ISBN 1-56881-147-0.

[14] R. Kähler, T. Abel, and H.-C. Hege. Simultaneous GPU-assisted raycasting of unstructured point sets and volumetric grid data. In *Proceedings of the Sixth Eurographics/IEEE VGTC conference on Volume Graphics*, pages 49–56. Eurographics Association, 2007.

[15] A. Knoll, I. Wald, P. Navratil, A. Bowen, K. Reda, M. E. Papka, and K. Gaither. RBF Volume Ray Casting on Multi-core and Many-core CPUs. *Computer Graphics Forum*, 33(3):71–80, 2014.

[16] A. Knoll, I. Wald, P. A. Navrátil, M. E. Papka, and K. P. Gaither. Ray tracing and volume rendering large molecular data on multi-core and many-core architectures. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, 2013.

[17] M. Le Muzic, J. Parulek, A.-K. Stavrum, and I. Viola. Illustrative visualization of molecular reactions using omniscient intelligence and passive agents. *Computer Graphics Forum*, 33(3):141–150, 2014.

[18] O. Mattausch, J. Bittner, A. Jaspe, E. Gobbetti, M. Wimmer, R. Pajarola, R. Avetisyan, M. Willert, S. Ohl, O. Staadt, et al. CHC+RT: Coherent Hierarchical Culling for Ray Tracing. *Computer Graphics Forum*, 33:2329–2334, 2014.

[19] P. A. Navrátil, J. L. Johnson, and V. Bromm. Visualization of cosmological particle-based datasets. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(6):1712–1718, 2007.

[20] J. Orthmann, M. Keller, and A. Kolb. Topology-caching for dynamic particle volume raycasting. In *Proceedings of Vision, Modeling and Visualization 2010, Siegen, Germany*, pages 147–154. Eurographics, 2010.

[21] OSPRay: A Ray Tracing based rendering engine for High-Fidelity Visualization. `https://ospray.github.io`, 2015.

[22] M. Pharr and B. Mark. ISPC - A SPMD compiler for high-performance CPU programming. In *Proceedings of Innovative Parallel Computing (inPar)*, 2012.

[23] D. J. Price. SPLASH: An interactive visualisation tool for Smoothed Particle Hydrodynamics simulations. *Publications of the Astronomical Society of Australia*, 24(3):159–173, 2007.

[24] K. Reda, A. Knoll, K. Nomura, M. Papka, A. Johnson, and J. Leigh. Visualizing large-scale atomistic simulations in ultra-resolution immersive environments. In *IEEE Symposium on Large Scale Data Analysis and Visualization (LDAV)*, pages 59–65, 2013.

[25] S. Rizzi, M. Hereld, J. Insley, M. E. Papka, T. Uram, and V. Vishwanath. Large-Scale Parallel Visualization of Particle Datasets using Point Sprites and Level-of-Detail. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)*, 2015.

[26] J. Stone, D. Hardy, I. Ufimtsev, and K. Schulten. GPU-accelerated molecular modeling coming of age. *Journal of Molecular Graphics and Modeling*, 29(2):116–125, 2010.

[27] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. yt: A multi-code analysis toolkit for astrophysical simulation data. *The Astrophysical Journal Supplement Series*, 192(1), 2011.

[28] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–573, 2005.

[29] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3), 2001.

[30] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree–A Ray Tracing Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 2014.

[31] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.