Ray Tracing and Volume Rendering Large Molecular Data on Multi-Core and Many-Core Architectures

Aaron Knoll
Texas Advanced Computing Center
knolla@tacc.utexas.edu

Ingo Wald
Intel Corporation
ingo.wald@intel.com

Paul A. Navrátil
Texas Advanced Computing Center
pnav@tacc.utexas.edu

Michael E. Papka
Argonne National Laboratory
papka@anl.gov

Kelly P. Gaither
Texas Advanced Computing Center
kelly@tacc.utexas.edu

ABSTRACT

Visualizing large molecular data requires efficient means of rendering millions of data elements that combine glyphs, geometry and volumetric techniques. The geometric and volumetric loads challenge traditional rasterization-based vis methods. Ray casting presents a scalable and memory- efficient alternative, but modern techniques typically rely on GPU-based acceleration to achieve interactive rendering rates. In this paper, we present bnsView, a molecular visualization ray tracing framework that delivers fast volume rendering and ball-and-stick ray casting on both multi-core CPUs and many-core Intel® Xeon Phi™ co-processors, implemented in a SPMD language that generates efficient SIMD vector code for multiple platforms without source modification. We show that our approach running on co- processors is competitive with similar techniques running on GPU accelerators, and we demonstrate large-scale parallel remote visualization from TACC's Stampede supercomputer to large-format display walls using this system.

1. INTRODUCTION

Large scientific data generated by today's supercomputers demand more capable analysis and visualization methods. Molecular dynamics (MD) simulations in particular challenge current visualization techniques since they frequently combine glyphs, geometric structures and volume rendering in a single image. Ball-and-stick representations remain the most popular means of molecular visualization, bearing similarity to physical erector-set models. However, a one million atom nanosphere dataset could produce 100 gigabytes of tessellated ball-and-stick geometry data; a fifteen million atom dataset would produce nearly a terabyte of geometry. Generating molecular surfaces adds to both the precomputation cost and the geometric data load. These data test the functional limits of traditional molecular modeling software (e.g. VMD [17]) and general visualization packages (e.g. ParaView [3], VisIt [9]), limiting the ability for scientists to explore such models effectively. To enable better interactive exploration, and ultimately in-transit visualization and computational steering, it is necessary to increase the efficiency of rendering ball-and-stick models, and employ volumetric models of molecular data similar to the underlying charge density fields from quantum physics.

Ray casting in particular is an attractive option for rendering molecular data, since ball-and-stick glyphs can be rendered directly without tessellation, and direct volume rendering can be used in place of molecular surfaces and isosurfaces in illustrating a charge density or potential field. By employing ray casting, we can reduce both the computational cost and the memory footprint needed for rendering, allowing large MD data to be rendered efficiently on a single node. However, existing ray casting approaches [28] rely on GPUs to achieve interactive performance. While GPUs are the most common accelerator in the world's top 500 supercomputers, most (89% in the June 2013 Top 500 list [30]) use only CPU-based processing. Out of the current top ten, only two include GPUs, six systems use only CPUs, and two include Intel® Xeon Phi™ coprocessors (MICs) including Tianhe-2, the current world #1. An efficient MD visualization technique that removes the GPU dependency, allowing for rendering on any compute resource, would broaden the insight and discovery potential for this important class of simulations.

In this paper, we present bnsView, a molecular visualization system for efficient ray casting and ray tracing of molecular data on multi-core CPU and many-core MIC (Xeon Phi™) architectures. As a central component of our approach, we present an efficient parallel ray tracing framework written in a custom SPMD language (similar to NVIDIA CUDA [1]), which enables efficient auto-vectorization of code for both CPUs and MICs. This paper makes the following contributions:

- a technique for efficient molecular visualization that combines both volume data and ball-and-stick geometry in the same ray tracing framework, all suitable for both multi-core and many-core architectures;
- a comparison of our technique with a state-of-the-art accelerator (GPU-based) technique; and
- a demonstration of our system used for parallel remote visualization from the Stampede supercomputer to a large-format tiled display, enabling in-transit visualization of large MD simulations.

The rest of the paper is organized as follows. In Section 2 we place our approach in context with related work. We provide background essential to evaluating our technique in Section 3, and we present our implementation in detail in Section 4. In Section 5 we discuss our methodology and results, and we consider future work and con-

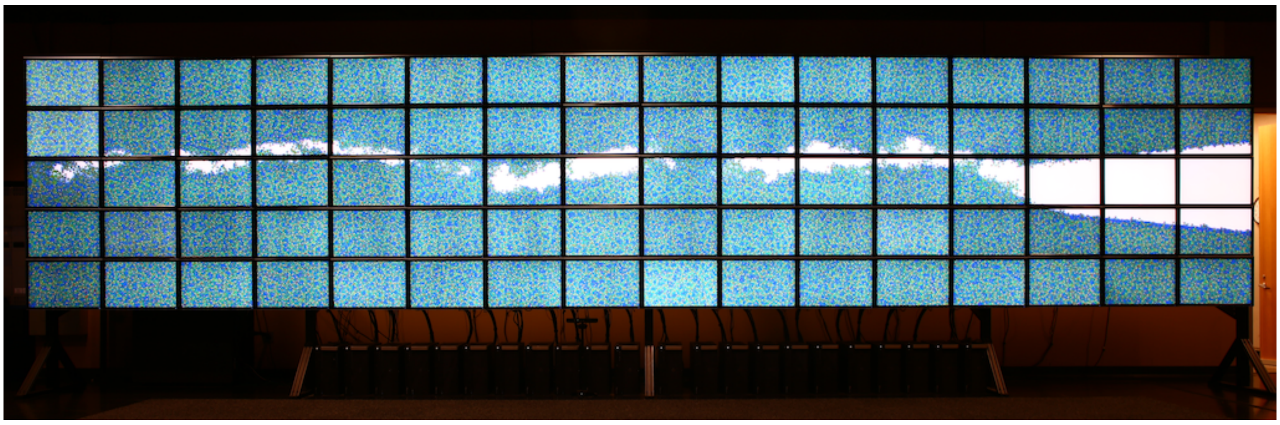


Figure 1: A 65536 $\times$ 8192 (512 megapixel) frame buffer rendered on TACC *Stampede* using 128 Intel$^{®}$Xeon Phi^{TM}'s and streaming at 2 fps to the TACC *Stallion* 328 megapixel tiled display.

clude the paper in Section 6.

2. RELATED WORK

In this section, we present work related to our technique to place our contribution in context.

2.1 Molecular Visualization

Popular molecular visualization applications such as VMD [17] and PyMol [12] focus on biomolecular problems such as protein docking; others such as Avogadro [15] emphasize building molecular geometry for *ab initio* computations. Generally, these applications are designed for molecules with up to thousands of atoms, and are not suited for materials problems with hundreds of thousands or millions of atoms. A large body of work on polygonal molecular surfaces exists, surveyed by Connolly [10]. For our data in VMD, molecular surfaces and isosurfaces were prohibitively slow to compute for the 740K nanosphere data, and the 5M atom silicon fissure data was only representable as rasterized points and lines.

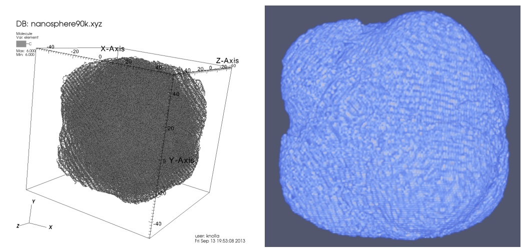


Figure 2: Left: VisIt can generate ball-and-stick models, but fails for data larger than the 90K nanosphere. Right: ParaView can render larger volume data such as the 740K nanosphere, but is slow and uses pre-classification, resulting in poor image quality compared to post-classified volume rendering.

General parallel visualization toolkits such as ParaView [3] and VisIt [9] have molecular visualization capability. Paraview and VisIt have means of rendering sphere impostors, but not cylinders. Volume rendering in these packages is relatively slow (VisIt) or poor-quality due to preclassification in (ParaView), as shown in Figure 2. Neither ParaView nor VisIt was capable out-of-the-box of generating bonds for our 740K atom nanosphere or larger data. Clearly, it would be possible to construct a parallel reader and rendering pipeline for larger MD data in ParaView and VisIt. However, for reasons of image quality, single node performance and lower memory utilization, direct ray casting approaches are compelling.

Many specialized molecular visualization tools have been developed for the GPU. Bajaj et al. [5] employ particle and impostor-based methods for fast rendering on the GPU using the rasterization and shader pipeline. They use 3D texture-based volume rendering at relatively small scales for LOD and illumination purposes. Tarini et al. [29] use similar techniques to approximate global illumination and retain real-time performance. The MegaMol framework [14] employs several different techniques to ray cast millions of atoms as impostors at real-time rates. Compared to our work, MegaMol's only limitation is that it employs rasterization and relies on LOD for performance, and would not support ray tracing for high-quality rendering.

The limitations of both general-purpose and molecular vis tools served as motivation for development of Nanovol [20, 28], a GPU ray casting application built on an efficient OpenGL volume ray caster for structured grids [21]. It uses a single-level uniform grid of "macrocells" to classify and skip empty space as defined by a transfer function. Macrocells also contain pointers to a list of ball and stick glyphs, denoted by atom indices and pairs of indices, respectively. GLSL was chosen for its compatibility across multiple GPUs, and (at the time of implementation) better volume rendering performance than CUDA. Similarly, the grid acceleration structure was chosen for its efficiency on the GPU (e.g. [11]). Recently, Nanovol has been extended to run in a multi-GPU environment, rendering million-atom MD data at real-time frame rates in the 70-megapixel stereo 3D CAVE2 environment [28].

2.2 CPU Ray Tracing

Parker et al. [25] demonstrated interactive performance for direct ray tracing of isosurfaces on a 128-CPU SMP supercomputer. The algorithms employed by bnsView are based on coherent ray tracing [35], in which rays are traced together in groups or *packets* [35], reducing both traversal and intersection costs by exploiting common memory access patterns and vector instructions. Subsequent work employed these techniques for interactive CPU visualization systems for structured [23, 22] and unstructured [33] isosurfacing and volume rendering. Gribble et al [13] employed coherent ray tracing for efficient rendering of multi-million atom sphere glyphs. Work integrating the Manta interactive ray tracer [6] into ParaView [8] and more generally for interception of polygonal geometry from OpenGL [7] has proven the advantages of ray tracing methods in visualizing large data in parallel, particularly with weak scaling. The recent Embree 2.0 framework [36] delivers interactive ray tracing of polygonal models on both CPUs and Intel®Xeon Phi™ accelerators, in part by using the Intel SPMD Program Compiler (ISPC) [26]. This framework is an open-source counterpart to the ray tracing framework we developed in IVL [24] for molecular visualization.

3. BACKGROUND

This section provides background on techniques used in our rendering approach. Readers experienced with molecular visualization concepts and vectorized ray tracing can proceed to Section 4.

3.1 Volumetric modeling of molecular data

In volume visualizing molecular data, our goal is to approximate a charge density or potential model that would provide insight into the presence of molecular surfaces or materials interfaces. Volumetric representation lets us represent this continuously, allowing for viewing from distances at which ball and stick representation would be impractical. We model this using a radial basis function (RBF), a continuous, real-valued function ϕ whose value decays with respect to distance from the atom. The global field function Φ consists of summing the radial basis functions for all particles i contributing to a point in space,

$$d_i = |\mathbf{x} - \mathbf{x_i}| \qquad \Phi = \sum_i \phi_i(d_i) \tag{1}$$

Common choices of ϕ are Gaussians and polynomials with compact support. While more sophisticated models are possible [20], in this work for simplicity we use a Gaussian with height scaled by radius

$$\phi_i(r) = \sqrt{r}e^{-d^2/r_i^2} \tag{2}$$

where r_i is the covalent radius. The $\sqrt{r}$ prefactor dampens the height of the Gaussian, distributing charge density more evenly across large and small atoms (though a rough approximation, this is similar in behavior to charge density plots from Density Functional Theory computation). We use $\sqrt{2}r_i$ as the support width for the basis function corresponds, which corresponds roughly to the Van der Waals radius. Outside of this outer radius we can clamp the contribution of the basis function to zero.

To precompute a structured volume using this model, we create a grid with a fixed resolution (typically 1 to 4 voxels per Ångström), depending on accuracy and memory requirements. We then iterate over each atom in the molecule, applying the radial basis function to each voxel it overlaps within the outer radius.

3.2 SPMD Coherent Ray Tracing

Achieving good ray tracing performance on modern architectures requires making good use of SIMD vector units. On GPUs this is typically achieved through languages such as CUDA, OpenCL, or GLSL using specialized kernels e.g. [4, 16]. On CPUs in "coherent" ray tracing, rays are bundled together into packets that simultaneously compute traversal, intersection, and shading routines using SIMD vector instructions [35]. To take advantage of vector intrinsics, coherent ray tracers must organize data in structure-of-arrays layout corresponding to vector width (e.g., SSE operating on 4 floats at once). However, SIMD width and vector instruction set vary across different CPU and accelerator architectures. To better exploit both vector- and thread-parallelism across all these different architectures and instruction set, a mechanism for abstracting SIMD lanes and multiple threads is increasingly in order.

IVL [24] is an experimental single-program, multiple-data (SPMD) compiler that can transparently target a variety of instruction sets including Intel®SSE, Intel®AVX, and Intel®Xeon Phi™. It automatically generates structure-of-arrays (SOA) layout for varying data across SIMD lanes and ensures properly vectorized control flow via masking and conditional execution, thus enabling efficient use of SIMD vector instructions. Crucially, it allows the programmer to write code once and emit for multiple vector backends (SSE, AVX, MIC and potentially others). IVL is closely related to the publicly available ISPC [26], but in addition supports dynamic polymorphism, operator overloading, verbatim low-level C++ code embedded within its SPMD code, and support for transparently "offloading" code and data to an accelerators where applicable. At the time of implementation, IVL offered concrete feature and performance advantages over ISPC. However, performance and behavior of our system should be similar and reproducible in an ISPC implementation as well.

4. IMPLEMENTATION

In this section we describe the implementation of bnsView on the CPU and MIC. At a high level, bnsView uses a bounding volume hierarchy (BVH) to accelerate opaque ball and stick geometry, and a single level grid of macrocells to accelerate volume rendering (similar to Nanovol). We chose a BVH based approach because (in part inspired by our experiences with Embree [36]) we anticipated better performance using coherent ray tracing algorithms on SIMD hardware. At the same time, volume rendering is accelerated using a single-level structured grid, similar to Nanovol. These acceleration structures are shown in Figure 3.

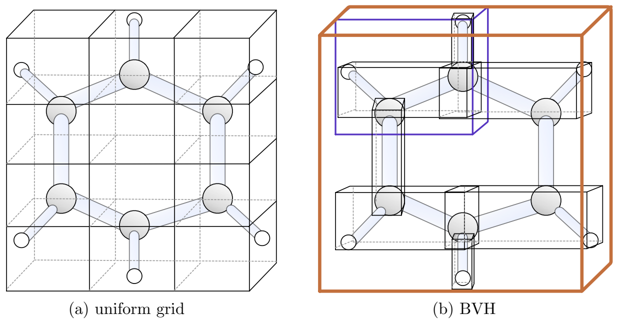


Figure 3: Acceleration structures. (a) Uniform grid uniformly decomposes world space. In Nanovol on the GPU, a grid is used for both volume and polygonal data: each macrocell stores indices to ball and stick primitives that overlap it; a primitive can be referenced by more than one macrocell. (b) In bnsView, we use a macrocell grid to accelerate traversal of the volume data, but use a bounding volume hierarchy (right) for the polygonal data: Each BVH leaf node contains up to 4 primitives (either balls or sticks); nodes may overlap but no object will be contained by more than one leaf.

4.1 Preprocess pipeline

Data is input as a set of atoms, each with a type (e.g., carbon) and 3D position in Ångströms. The following preprocess steps then occur:

1. Atoms are inserted into a uniform grid of particles (independent from the macrocell grid associated with the volume data). The dimensions of this grid are chosen by dividing the bounds of the molecule by minimum bond length (specifically, the spacing between any two atoms).

2. Bonds are built using nearest-neighbor lookup from this grid structure.

3. Compute bounding boxes and centroids for ball and stick primitives; compute global bounds in the same pass.

4. BVH construction, described below.

5. The structured volume is precomputed using the RBF model. This is implemented using multiple threads (OpenMP), with resolution chosen by a "voxels per Ångström" parameter (default to four).

6. If necessary, data are transferred across the bus to the MIC. In our implementation, this data transfer is done transparently by the IVL compiler's runtime.

BVH construction is a binned variant of a surface area heuristic (SAH) build used for axis-aligned kd-trees or bounding volume hierarchies of polygonal data [31]. First, we build the list of base primitives (balls and sticks), their centroids and bounding boxes. We sort primitives into three separate indexed lists, along X, Y and Z axes. Then a recursive routine proceeds as follows: if the number of primitives in the node is less than the "leaf" threshold (we use four primitives per leaf), we return. Otherwise, we initialize the "best cost" as the SAH of the parent node. To find the partition split plane, we sweep from left to right over the chosen axes, extending the bounds of the partition and computing the the heuristic, and choosing the partition with the minimal cost. We then recursively call the routine for left and right children, passing their already sorted sub-lists of primitives. For reasons of convenience we used the already-implemented SAH builder in the RIVL polygonal ray tracer. Since this builder has never been designed or optimized for real-time rebuilds we are currently limited to static scene geometry. However, the Embree framework [36] contains several BVH builders that are applicable to our needs and would enable interactive rebuilds as well.

4.2 Ray tracing framework

BnsView is built on top of RIVL, a ray tracing engine written in IVL [24], and a precursor of the SPMD module that is now part of Embree 2.0 (which is written using the publicly available ISPC [26]). As with other SPMD languages, kernels are expressed in scalar form that perform traversal, intersection, and rendering in SIMD parallel on rays. Semantically, these rays are expressed as *varying* data IVL and ISPC, and are distributed across SIMD lanes by the compiler. Scene data, such as camera, geometry, acceleration structure, transfer function, are *uniform*, thus constant across SIMD lanes. Like ray tracers in GLSL, OpenCL or CUDA, task-parallelism is expressed by subdividing the screen into 'tiles' that are then scheduled to be rendered in parallel. Inside each tile rendering task the SPMD compiler then gangs multiple rays together into SIMD-sized chunks/warps that are processed in SPMD fashion. All tiles of a task are scheduled together into a tasking system in which all threads pick tasks from a shared task pool while available; all SPMD control flow is handled by the compiler.

Like GPU programming, IVL maintains the concept of host and device. The first step is to perform host-to-device copy of all data precomputed on the CPU (the BVH, balls, sticks, and structured data if it exists). For CPU backends, pointers to the data are shared with the kernel and no explicit data copy is made. To generate rays, we pass a frame buffer, camera and scene data to a device-side kernel, which in turn divides the frame buffer into tile tasks. In IVL/ISPC, this involves writing a `renderTile` task, which processes all of a tile's pixels in SIMD width-sized chunks, generates a ray per pixel (all SIMD-parallel), and passes this ray to our framework, thus beginning the ray casting or tracing process.

Tracing a ray consists of two separate passes:

1. opaque geometry traversal, using a bounding volume hierarchy around ball and stick geometry. This stores a single opaque hit position and gradient, which can then be shaded.

2. volume rendering of the RBF-modeled structured data (front-to-back, from camera viewpoint to first opaque hitpoint), using a uniform grid for acceleration. This stores an integrated color and opacity.

Generally, the volume rendering pass is more costly, so we use the opaque geometry pass for occlusion and early termination of the volume rendering pass. Either pass can be enabled or disabled, depending on the need of the shader calling it. This approach limits secondary bounces to occuring at opaque geometry intersections or where the ray terminated in the volume. However, by calling this `trace` function iteratively and integrating color in between, one can fully implement various ray tracing effects.

4.3 BVH traversal, ball and stick intersection

Our BVH traversal is similar in practice to coherent BVH traversal [32], with fewer optimizations (e.g. no interval arithmetic and per-packet culling tests) but simpler implementation. Rays start at the root node and descend into children when any ray in the *varying* packet intersects the child bounding box. Active rays, automatically masked by conditionals, intersect ball and stick geometry at the leaf nodes. Then, the rays traverse up the stack to the subsequent node, repeating the process until no more nodes remain and traversal ends. With coherent traversal, nodes are traversed when any single ray intersects, thus the stack maintained for BVH traversal (as well as all traversal control flow associated with it) is *uniform*. This stands in contrast to a GPU implementation where each separate thread (corresponding to SIMD lane on the CPU/MIC) would have a separate stack. Packet-based traversal typically suffers from somewhat lower SIMD utilization than GPU-style independent ray traversal (in particular for complex models as we are using), but in turn suffers less from control flow divergence, and in particular does not require the frequent scatter/gather required by GPU methods; when weighed against each other, packet approaches are often preferable on CPUs.

Intersecting balls and sticks employs standard ray-sphere and ray-cylinder intersection tests. Like nodes in the BVH, spheres and cylinders are also *uniform* data. Thus, the tests are performed in SIMD parallel for multiple (active) varying rays on a single geometric primitive.

4.4 Structured volume rendering and shading

Structured volume rendering is implemented in straightforward fashion, using ray casting with uniform sampling, trilinear interpolation of the structured grid, and post-classification. We transform the ray into volume grid coordinates, determine entry and exit coordinates of the volume bounding box, then march along at a fixed step size. Trilinear interpolation is implemented as seven linear interpolations, with care taken to reduce the cost of address translation. We

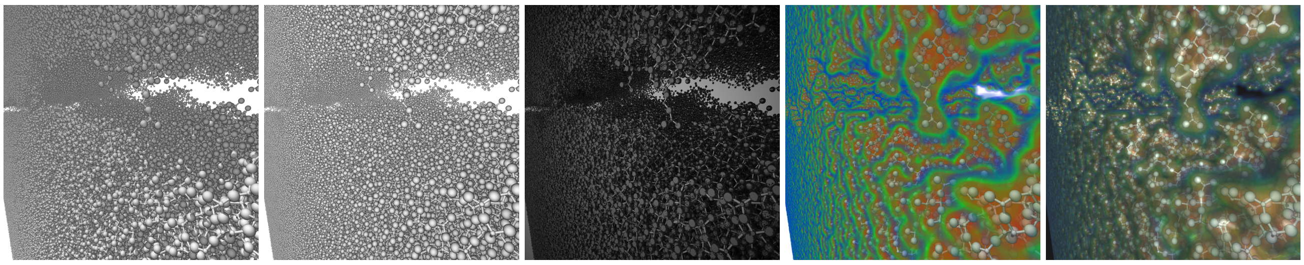


Figure 4: Ray tracing modalities with bnsView. From left to right: ball-and-stick with diffuse illumination and shadows (58 fps on the Xeon Phi), ambient occlusion (1.8 fps), and path traced with a metal BRDF (0.25 fps); unlit volume rendering with volumetric shadows (16 fps), and the same with gradient lighting (3.7 fps).

note that volume data are not bricked for maximal cache efficiency; performance could be improved with better cache strategies.

To improve volume rendering performance, bnsView employs a macrocell grid on top of volume data similar to Nanovol. This grid contains only minimum and maximum range values of the volume data, not pointers to ball and stick geometry. Unlike with coherent BVH traversal, rays traverse the volume grid and intersect voxels independently. Nonetheless, we found this provided up to 3× performance improvement on data with significant empty space such as the large nanosphere.

We implemented several options for volumetric shading. With no lighting, we simply return the trilinear interpolant and classify. With central differences, we fetch the interpolated values at six neighboring voxels in the X,Y and Z directions. This ensures smooth gradients but is costly in particular on architectures that do not contain hardware 3D texture units and dedicated texture caches. All interpolation and shading operations are performed in parallel (for *varying* ray data).

4.5 Ray tracing modalities

The ray tracer can determine how to shade the results of both passes for a range of effects. BnsView can fully ray-trace molecular data using opaque geometry, volume data, or both for a wide range of visual effects. For example, we can render ball-and-stick only with ambient occlusion or a reflective metal, or use volume rendering for occlusion rays, achieving a soft-shadow effect. Several examples are shown in Figure 4.

5. RESULTS

We present the results of our experiments below. Unless stated otherwise, the following benchmarks were conducted using a 1024 × 1024 frame buffer on a visualization node of Stampede with dual 8-core (16 cores total) 2.7 GHz Intel®Xeon ™ E5-2680 with 32 GB RAM, an Intel®Xeon Phi™ SE10P with 61 cores at 1.1 GHz with 8 GB RAM, and an NVIDIA K20 (Kepler) GPU with 6 GB RAM. All computations were carried out in single-precision floating point. On the CPU, we used the 8-wide AVX instruction set.

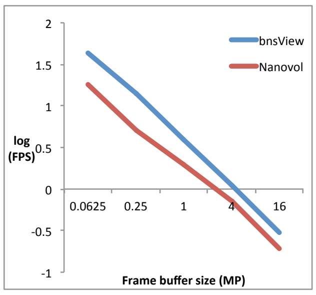
5.1 Overall performance comparison

We evaluated bnsView on the CPU, bnsView on MIC, and Nanovol on the GPU for five molecular datasets ranging from 20 thousand to 15 million atoms. We considered a "far" and "near" reference view, and report performance. For both volume rendering methods, we used a fixed step size of 0.5 samples per voxel unit. We use the same pre-integrated transfer function with a 256-bin lookup table. For our RBF volume data we used 4 voxels per Ångström for all data sets except the silicon fissure (SiO_2) and alumina nanospheres (ANP3), which used 1 voxel per Ångström to fit the memory constraints of the GPU and MIC. To save space, we also cast volume data from float to one-byte scalars.

BnsView and Nanovol ray cast molecular data similarly, using different algorithms. These algorithms were chosen to maximize performance and maintain flexibility in these respective systems, not explicitly to make the fairest comparison for this paper. We chose coherent BVH traversal in bnsView because it is a known efficient method for handling large geometry on the CPU. Likewise, Nanovol was developed using a uniform grid because that performed well for traversal of structured data on the GPU [21]. In development of Nanovol [28], it was assumed that volume rendering would invariably be the performance bottleneck, and that a grid acceleration structure would be sufficient. This generally holds true, but for large molecular data when a coarse grid is chosen, the cost of ball-and-stick rendering increases relative to the cost of volume rendering. More efficient ball-and-stick ray tracing, for example using BVH traversal [4] would be possible on the GPU, but would likely require a reimplementation of Nanovol in CUDA or OpenCL, and falls outside the scope of our work. Despite their different choice of algorithm, bnsView and Nanovol are functionally (and intentionally) very similar and exhibit shared behavior. Compared to other systems such as MegaMol, ParaView or VisIt, comparison of these two systems, while acknowledging their differences, is fair.

In all, we are able to achieve interactive performance for ray casting on all platforms. We show results in Table 3 and reference scenes in Figure 6. For ball-and-stick geometry alone, bnsView on the CPU and MIC performs better than Nanovol on the GPU by up to 2× – 5×. This can be explained by the efficiency of coherent BVH traversal employed by bnsView, compared to the grid implementation of Nanovol. In particular, building coarser grids due to GPU memory constraints decreases efficiency, and performance suffers when many balls and sticks reside in each grid cell, as discussed above.

For volume rendering alone, Nanovol on the GPU is generally faster than bnsView on both the CPU (2× – 7×, average 5×) and the MIC (0.65× – 5.7×, average 2.5 ×). BnsView and Nanovol employ the same volume rendering algorithm (grid traversal with macrocells), with the GPU having the advantage of optimally cache-aligned 3D texture and built-in hardware for texture sampling, interpolation, and caching. Despite this, GPU volume-only performance is not always better: with larger volume data such as the nanosphere740k (1 GB) and ANP3 (2 GB) datasets, the MIC outperforms the GPU by up to 1.5×. This is especially surprising given that bnsView uses no special bricking and paging of large volume data, compared to the native 3D texture format of the GPU; we suspect the performance advantage is due to better behavior of MIC for cache-



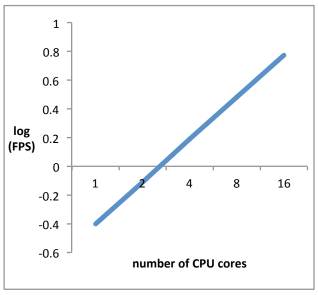


Figure 5: Log-scale frame rate plotted against frame buffer size (left) and number of cores (right).

FB size (MP)	.0625	.25	1	4	16
bnsView	44	14	4	1.1	.3
nanovol	18	5	2	0.7	.19

Table 1: Performance in fps scaling to frame buffer size (megapixels).

N. cores	1	2	4	8	16
bnsView	.394	.779	1.54	3.05	5.93

Table 2: Performance in fps scaling to number of cores.

incoherent memory access. Gradient lighting with central differences is expensive for bnsView, incurring a $1.5\times - 3\times$ performance penalty compared to unlit volume rendering. In contrast, lighting is only slightly (10-20%) more costly for Nanovol on the GPU. Although volume lighting is not always used in practice, it is worth noting that it is faster on the GPU, likely due to that platform's efficiency for algorithms with good cache locality. Comparing CPU to GPU and Xeon Phi™ performance, we note that while the CPU renders more slowly than accelerators or co-processors (on average, a dual-socketed CPU configuration is roughly $2\times$ slower than one Xeon Phi™ accelerator board), it is still capable of interactive performance, and in particular is not subject to the same scene size limits that GPUs and accelerators are. Generally, the CPU is better at handling geometry and worse at volume rendering, arguably due to our *SandyBridge*-based CPUs' lack of hardware-support for gather which is available on both GPU and Xeon Phi™ (as well as on some newer CPU generations).

5.2 Frame buffer size and number of cores

In Figure 5-left, we benchmark the nanosphere740k "far" scene using volume rendering at varying frame buffer sizes (256^2 to 4096^2). Generally, we see GPU performance increase at a faster rate with respect to frame buffer size, up until the 4 MP mark where both systems are roughly $3\times$ slower than 1 MP (5 fps vs 1.5 fps for bnsView, 2 fps vs 0.70 fps for Nanovol). This implies the GPU implementation depends more strongly on coherence for performance, and at high enough resolution it makes less difference.

In Figure 5-right, we examine bnsView performance (again with the nanosphere740k far scene) with one to sixteen cores of a 2.7 GHz E5-2680 Sandy Bridge Xeon CPU. We achieve 94% scalability at 16 cores, likely due to non-uniform memory access (NUMA) effects.

5.3 Remote visualization with bnsView

We integrated bnsView with the DisplayCluster [18] framework to enable remote in-transit visualization of a live molecular dynamics simulation (a one million atom version of the large nanosphere model in Figure 6). Using eight nodes of Stampede, and using MPI for image-parallel rendering with replicated data, we performed visualization on the Intel® Xeon Phi™ co-processors while the simulation (using LAMMPS [27]) ran on the nodes' CPUs. We were able to achieve 20 fps for a 4096×2048 frame buffer streaming to Stallion, TACC's 80-panel, 328 megapixel tiled display at the University of Texas at Austin. This corresponds closely to the per-node performance at 1 MP from Table 3. Using sixteen nodes of Stampede, we generated a 32768×8192 image at 0.5 fps, successfully utilizing most of the display at native resolution. Using 128 Stampede nodes, we were able to fill a 512 megapixel frame buffer at 2 fps (see Figure 1).

BnsView updates geometry in a separate thread whenever a new timestep arrives, allowing scientists to interact with the run as it progresses. The use of a platform-independent SPMD compiler opens up new possibilities for remote, in-transit and in situ visualization of this kind, with either the CPUs or the MIC co-processors tasked with rendering and the other with computation, using the same rendering algorithm and SPMD code independent of the underlying hardware or instruction set.

6. CONCLUSION

We have presented an efficient method for large molecular visualization on CPU and MIC architectures, capable of ray casting millions of ball and stick glyphs and efficiently volume rendering both from precomputed structured data. We used coherent BVH traversal in a ray tracer implemented in an SPMD compiler to achieve high performance across both CPUs and Intel® Xeon Phi™ co-processors without having to write dedicated code for each platform. We find our method to be competitive with Nanovol on the GPU for similar tasks. For volume and ball-and-stick ray casting, bnsView is up to $2\times$ faster on the Intel® Xeon Phi™'s, and competitive on the CPU with a ray caster performing similar tasks on a state-of-the-art K20 GPU. With the implementations we have chosen, the GPU is more capable for some tasks (volumetric lighting), and the CPU/Intel® Xeon Phi™ at others (ray casting opaque geometry).

The main barrier to using our implementation for in-transit visualization is preprocess time. In particular, precomputing structured data is the most significant preprocess bottleneck. This could be avoided altogether by directly rendering from the particle data; an approach we would like like to further investigate in future work. If the bottleneck around computing the structured data could be avoided the next biggest bottleneck would be BVH construction time, which in our current implementation is clearly non-interactive. Integrating our framework with some of the fast BVH builders that have recently been added to the Embree framework interactive rebuilds should be possible even for non-trivial data sets. We would also be interested in lazy (implicit) BVH builds, e.g. [2, 19, 34].

In addition, in future work we would also like to investigate larger distributed data with our system. Since both IVL and RIVL are closed-source experimental codebases, in order to make our software more accessible we also plan future development around the open-source ISPC and Embree ray tracing frameworks [36]. Ultimately, we would like to construct a common framework with which to evaluate similar visualization algorithms on CPU, Intel® Xeon Phi™, and GPU, using Stampede as a testbed.

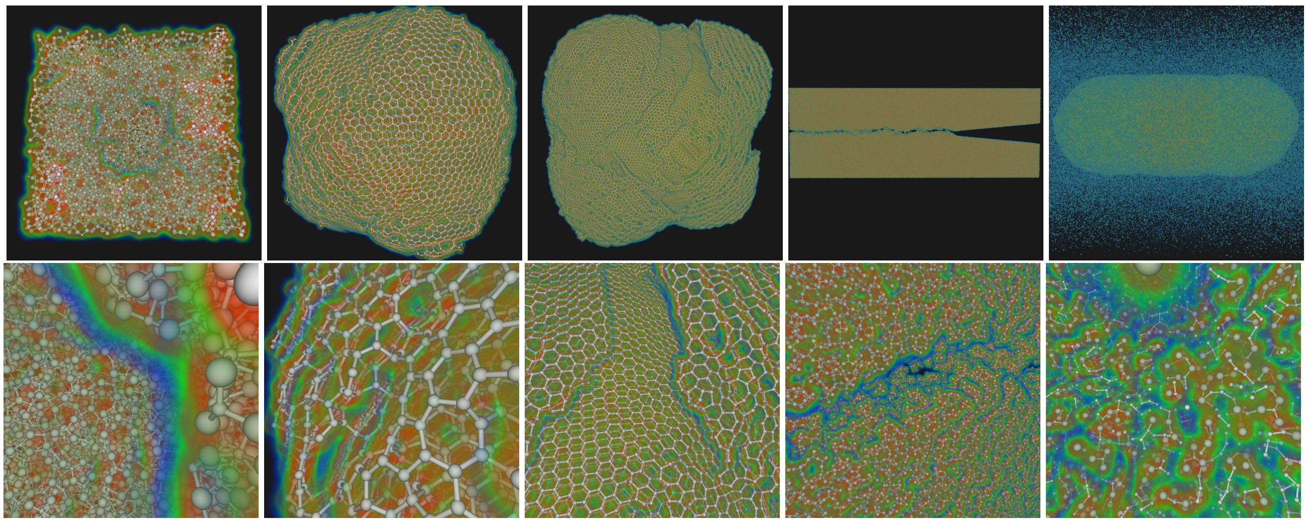


Figure 6: Reference scenes. From left to right: Al_2O_3 nanobowl (20K atoms), small carbon nanosphere (90K atoms), large carbon nanosphere (740K atoms), SiO_2 fissure (5M atoms), and ANP3 alumina nanoparticle combustion (15M atoms).

Dataset		nanobowl	ns90k	ns740k	SiO_2	ANP3
	num. atoms (M)	0.020	0.092	0.742	4.8	14.7
	data size per timestep (MB)	0.8	3	40	160	950
	geometry size (MB)	0.7	6	52	130	504
	BVH size (MB)	0.5	4	34	160	430
	str. vol size (MB)	1.1	11	720	92	2012
	voxels per Ångström	4	4	4	1	1
	bond build time (s)	0.03	0.18	2	10	28
	BVH build time (s)	0.08	0.9	7.5	50	128
	str. vol build time (s)	0.6	1.1	128	35	70
bnsView (CPU)	b&s	63 / 55	42 / 45	39 / 40	33 / 45	17.4 / 30
Intel® Xeon™	volume	15.1 / 7.94	8.75 / 9.31	6.82 / 7.91	15.3 / 7.67	1.3 / 4.0
E5-2680	vol+b&s	22 / 15	8.6 / 10.2	6.07 / 7.85	13.3 / 7.65	1.31 / 6.4
	vol+b&s, lit	6.15 / 4.02	2.42 / 2.97	1.57 / 2.46	4.51 / 2.02	0.35 / 1.91
bnsView (MIC)	b&s	**160 / 130**	**90 / 95**	**70 / 41**	**72 / 91**	**33 / 48**
Intel® Xeon Phi™	volume	53 / 32	26 / 28	**19.6** / 22.5	36 / 40	**3.59** / 12.3
SE10P	vol+b&s	**71 / 46**	**23.3** / 28.3	**18.1 / 23.8**	**39 / 33**	**3.22 / 28.0**
	vol+b&s, lit	36 / 22	12.4 / 14.8	**9.98 / 14.1**	**20.3** / 10.7	1.18 / 14.1
nanovol (GPU)	b&s	78 / 66	40 / 74	21 / 40	36 / 41	6.80 / 26
NVIDIA	volume	**73 / 60**	**36 / 54**	12.9 / **24.3**	**105 / 88**	3.51 / **70**
Tesla K20	vol+b&s	34 / 30.5	20.7 / **29.9**	7.0 / 11.6	20.1 / 24.4	2.63 / 19.5
	vol+b&s, lit	**41 / 32.5**	**19.5 / 26**	6.0 / 10.7	19.6 / **20.9**	**2.50 / 17.3**

Table 3: Performance for (far / close) views, in frames per second, at 1024×1024 **resolution for five molecules ranging from 20K to 15M atoms. The fastest fps result for each render type is in bold font. Reference images (using vol+b&s) are shown in Figure 6.**

7. ACKNOWLEDGMENTS

This work was funded in part by National Science Foundation grants OCI-1134872 and ACI-1339863. Additional funding was provided from the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357, and the Computational Postdoctoral Fellowship at Argonne National Laboratory under the American Reinvestment and Recovery Act (ARRA). At Argonne National Laboratory, we would like to thank Maria Chan at the Center for Nanoscale Materials and Kah Chun Lau at the Materials Science Division for the nanobowl and nanospheres data sets, respectively. We thank Ken-ichi Nomura at the University of Southern California for the SiO_2 fissure and ANP3 data sets, as well as the INCITE program at Argonne National Laboratory that made their computation possible.

8. REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] A. T. Áfra. Incoherent ray tracing without acceleration structures. In *Eurographics (Short Papers)*, pages 97–100, 2012.

[3] J. Ahrens, B. Geveci, and C. Law. Paraview: An end user tool for large data visualization. *the Visualization Handbook. Edited by CD Hansen and CR Johnson. Elsevier*, 2005.

[4] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[5] C. Bajaj, P. Djeu, V. Siddavanahalli, and A. Thane. Texmol: Interactive visual exploration of large flexible

multi-component molecular complexes. In *Proceedings of the conference on Visualization'04*, pages 243–250. IEEE Computer Society, 2004.

[6] J. Bigler, A. Stephens, and S. G. Parker. Design for parallel interactive ray tracing systems. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 187–196. IEEE, 2006.

[7] C. Brownlee, T. Fogal, and C. D. Hansen. GLuRay: Enhanced ray tracing in existing scientific visualization applications using OpenGL interception. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 41–50. The Eurographics Association, 2012.

[8] C. Brownlee, J. Patchett, L.-T. Lo, D. DeMarle, C. Mitchell, J. Ahrens, and C. D. Hansen. A study of ray tracing large-scale scientific data in two widely used parallel visualization applications. In *Eurographics Symposium on Parallel Graphics and Visualization*, pages 51–60. The Eurographics Association, 2012.

[9] H. Childs, E. Brugger, K. Bonnell, J. Meredith, M. Miller, B. Whitlock, and N. Max. A contract based system for large data visualization. In *Visualization, 2005. VIS 05. IEEE*, pages 191–198. IEEE, 2005.

[10] M. Connolly. Molecular Surfaces: A Review. *Network Science Online*, 1996. http://www.netsci.org/Science/Compchem/feature14.html.

[11] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22. ACM, 2009.

[12] W. L. DeLano. The PyMOL molecular graphics system, http://www.pymol.org. 2002.

[13] C. P. Gribble, T. Ize, A. Kensler, I. Wald, and S. G. Parker. A coherent grid traversal approach to visualizing particle-based simulation data. *Visualization and Computer Graphics, IEEE Transactions on*, 13(4):758–768, 2007.

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

[15] M. D. Hanwell, D. E. Curtis, D. C. Lonie, T. Vandermeersch, E. Zurek, and G. R. Hutchison. Avogadro: an advanced semantic chemical editor, visualization, and analysis platform. *Journal of cheminformatics*, 4(1):1–17, 2012.

[16] D. Hughes, I. Lim, M. Jones, A. Knoll, and B. Spencer. Ink-compact: In-kernel stream compaction and its application to multi-kernel data visualization on general-purpose gpus. In *Computer Graphics Forum*. Wiley Online Library, 2013.

[17] W. Humphrey, A. Dalke, K. Schulten, et al. VMD: visual molecular dynamics. *Journal of molecular graphics*, 14(1):33–38, 1996.

[18] G. P. Johnson, G. D. Abram, B. Westing, P. Navratil, and K. Gaither. Displaycluster: An interactive visualization environment for tiled displays. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 239–247. IEEE, 2012.

[19] T. Karras and T. Aila. Fast parallel construction of high-quality bounding volume hierarchies. *Proc. High-Performance Graphics*, 2013.

[20] A. Knoll, M. Chan, K. Lau, B. Lui, J. Greeley, L. Curtiss, M. Hereld, and M. Papka. Uncertainty classification and visualization of molecular interfaces. *International Journal of Uncertainty Quantification*, 3(2):157–169, 2013.

[21] A. Knoll, Y. Hijazi, R. Westerteiger, M. Schott, C. Hansen, and H. Hagen. Volume ray casting with peak finding and differential sampling. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6):1571–1578, 2009.

[22] A. Knoll, S. Thelen, I. Wald, C. D. Hansen, H. Hagen, and M. E. Papka. Full-resolution interactive CPU volume rendering with coherent BVH traversal. In *Pacific Visualization Symposium (PacificVis), 2011 IEEE*, pages 3–10. IEEE, 2011.

[23] A. M. Knoll, I. Wald, and C. D. Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, 2009.

[24] R. Leißa, S. Hack, and I. Wald. Extending a c-like language for portable SIMD programming. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 65–74. ACM, 2012.

[25] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P. Sloan. Interactive ray tracing for isosurface rendering. In *Visualization'98. Proceedings*, pages 233–238. IEEE, 1998.

[26] M. Pharr and W. Mark. ispc: A SPMD compiler for high-performance CPU programming. *Proceedings of Innovative Parallel Computing (InPar)*, 2012.

[27] S. Plimpton. Lammps user manual. *Sandia National Laboratory*, 2005.

[28] K. Reda, A. Knoll, K. Nomura, M. Papka, A. Johnson, and J. Leigh. Visualizing large-scale atomistic simulations in ultra-high resolution immersive environments. In *IEEE LDAV (to appear)*, 2013.

[29] M. Tarini, P. Cignoni, and C. Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 12(5):1237–1244, 2006.

[30] TOP500.org. Architecture Share for 6/2013, June 2013.

[31] I. Wald. On fast construction of SAH-based bounding volume hierarchies. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 33–40. IEEE, 2007.

[32] I. Wald, S. Boulos, and P. Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics (TOG)*, 26(1):6, 2007.

[33] I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen. Interactive isosurface ray tracing of time-varying tetrahedral volumes. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1727–1734, 2007.

[34] I. Wald, T. Ize, and S. G. Parker. Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes. *Computers & Graphics*, 32(1):3–13, 2008.

[35] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum (Proceedings of EUROGRAPHICS)*, 20(3):153–164, 2001.

[36] S. Woop, C. Benthin, and I. Wald. Intel embree 2.0: Photorealistic ray tracing kernels, http://embree.github.io. 2013.