Stallion tiled display in the TACC vislab, 512 MP image, 128 MIC's, 2 fps

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

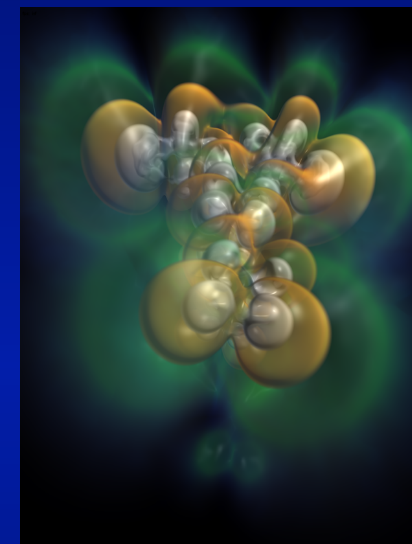
Aaron Knoll (TACC)
Ingo Wald (Intel)
Paul Navratil (TACC)
Michael E. Papka (ANL)
Kelly Gaither (TACC)

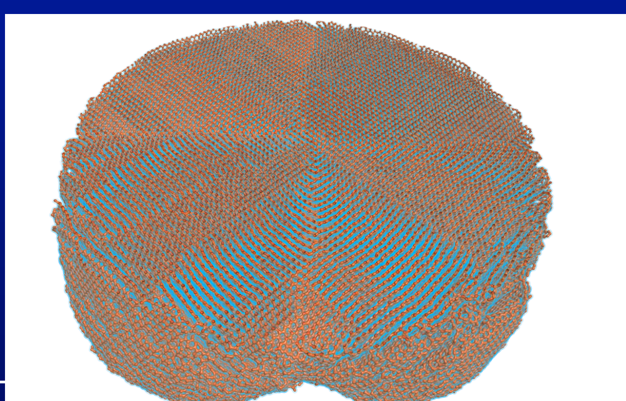
Motivation

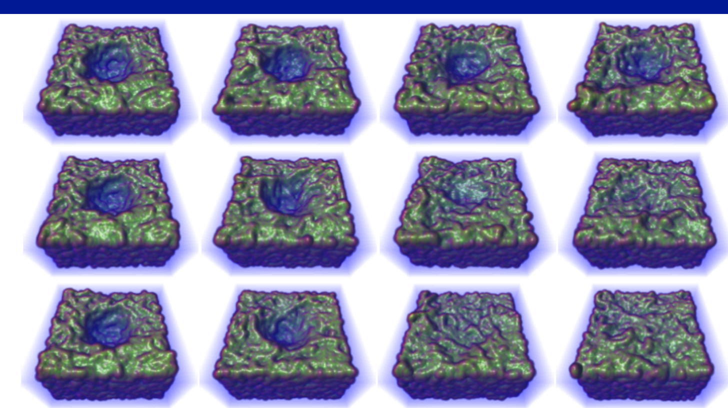
- "Direct" visualization
  - Glyphs and volume data, in particular applied to chem vis
  - No triangles: reduce memory footprint, improve quality
- Interactive vis without GPUs
  - Evaluate CPU, MIC and GPU performance
  - GPGPU-like code that works on CPU+MIC
  - "First steps" towards platform-abstract ray tracing for visualization
- Ray tracing
  - Better scalability to large data
  - Better image quality (for a price…)
  - One pipeline for both batch and interactive rendering

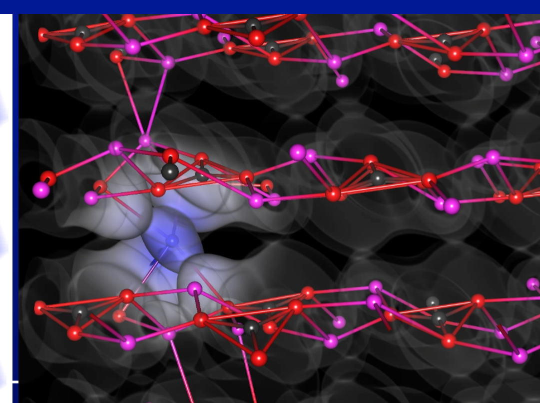
Chem vis

- Materials and biochem are increasingly significant HPC workloads
- Classical molecular dynamics, ab initio MD, DFT
- Volume rendering provides:
  - Continuous electron density/potential fields
  - Automatic LOD, transfer functions for illustrating uncertainty and contrast in surfaces/interfaces
- DVR is expensive, most chem packages don't do it
- Computational chemists want these features *and* ball-and-stick, for increasingly large data with many timesteps
  - They want to do vis locally and on clusters, with or without GPU's.









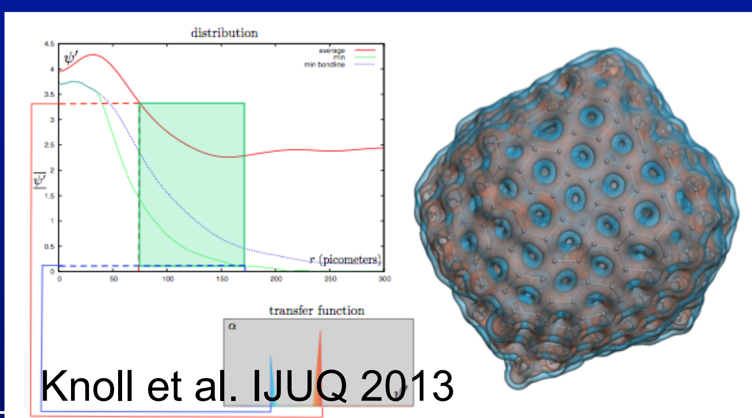
TACC

Data courtesy KC Lau, Maria Chan, Hakim Iddir,Julius Jellinek @ ANL

Related work: Nanovol

Aaron Knoll (TACC/ANL), Khairi Reda (EVL, UIC), Michael E. Papka (ANL)

- GPU ray casting for large MD data (up to 15M atoms)
- Ball-and-stick and volume rendering, nice lighting + filtering
- Compute RBF volume data from molecule statistics, bulk DFT
- Many, many other molecular vis solutions, why nanovol?
  - Support for volume rendering
  - Not built around triangle preprocess pipeline (e.g. VMD, PyMol, Paraview, Visit)
  - Not specifically built for fast LOD glyph rendering (e.g. MegaMol)
  - Ray casting using a grid acceleration structure; could support full ray tracing.
  - Straightforward GLSL implementation, easy to reproduce and compare against



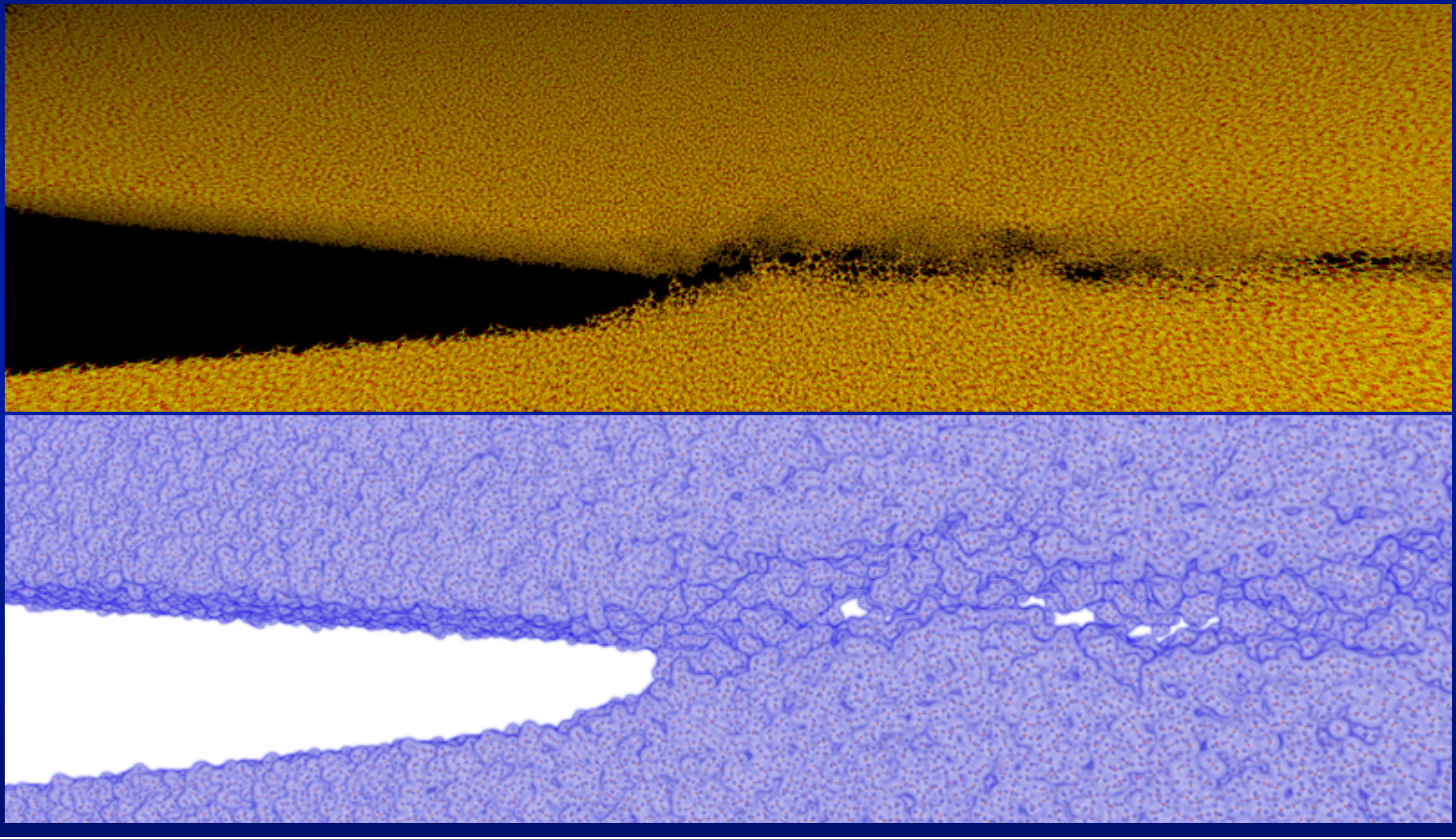
Knoll et al. IJUQ 2013

On CAVE2: Reda et al. LDAV 2013

SiO2 Fissure: VMD vs Nanovol

5M atoms, ~300 MB/ timestep

VMD: GPU choke(d) on 5-100 GB of ball+stick or surface geometry. But GL_LINES are very fast!
Nanovol: 1 voxel per Angstrom volume data (92 MB), analytical glyphs, 6 fps @ 4 MP



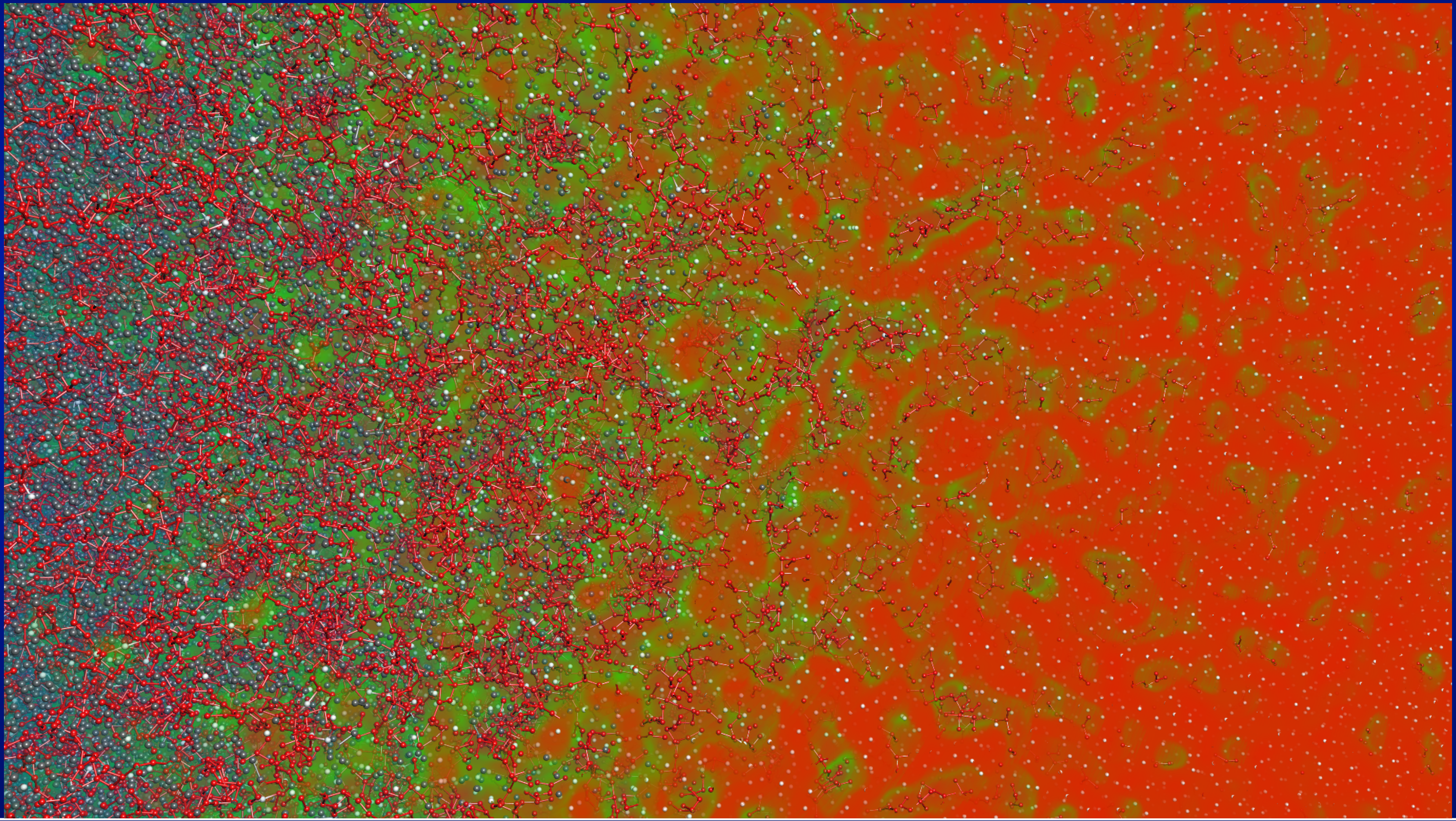
Data courtesy Ken-ichi Nomura, University of Southern California

ANP3 aluminum oxidation data

15M atom dataset (~1 GB / timestep)

Could only fit a 0.5 voxel-per-Angstrom volume in memory on a 680 GTX!

Coarse macrocell grid, slow performance (0.2 fps @ 2 MP)



TACC

Data courtesy Ken-ichi Nomura, University of Southern California

3 solutions to GPU memory limits:

- Go parallel
- Out-of-core/LOD
- Use hardware with more memory

- Our solution: use the CPU/MIC before going parallel.
  - Ideally, we'd like to do all 3. But first things first.

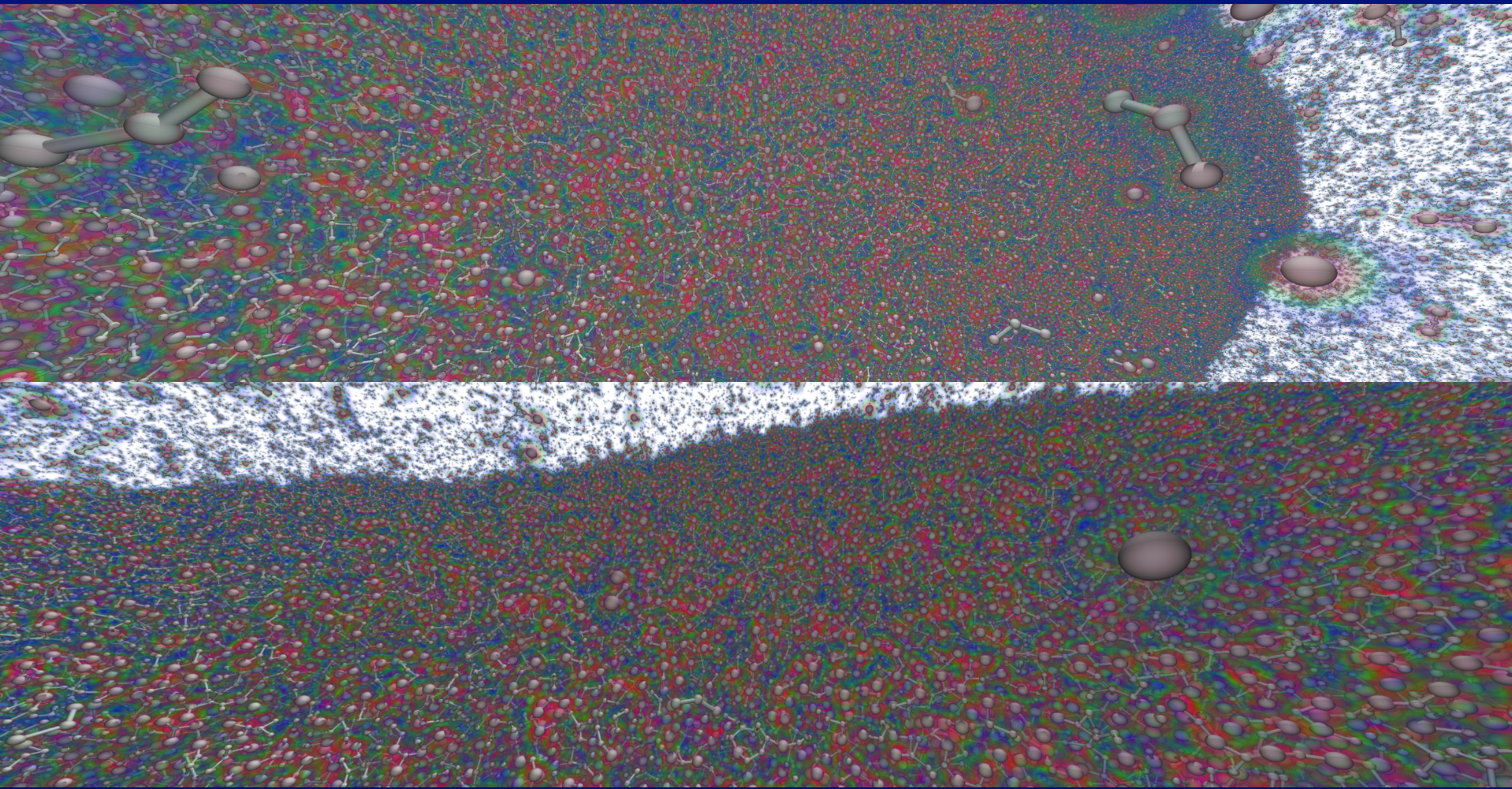
Data courtesy Ken-ichi Nomura, University of Southern California

ANP3 data in bnsView

more memory on both CPU and MIC

BVH performs more gracefully than grid

(3-4 fps at 4 MP on 1 SE10P Xeon Phi®, ray tracing with hard shadows)

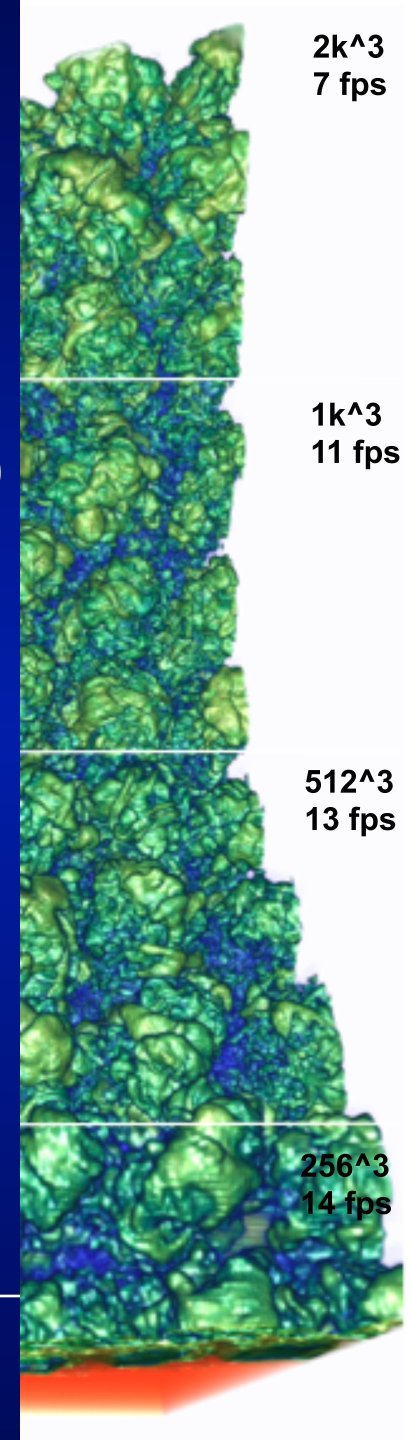


TACC

Data courtesy Ken-ichi Nomura, University of Southern California

Benefits of CPU ray tracing

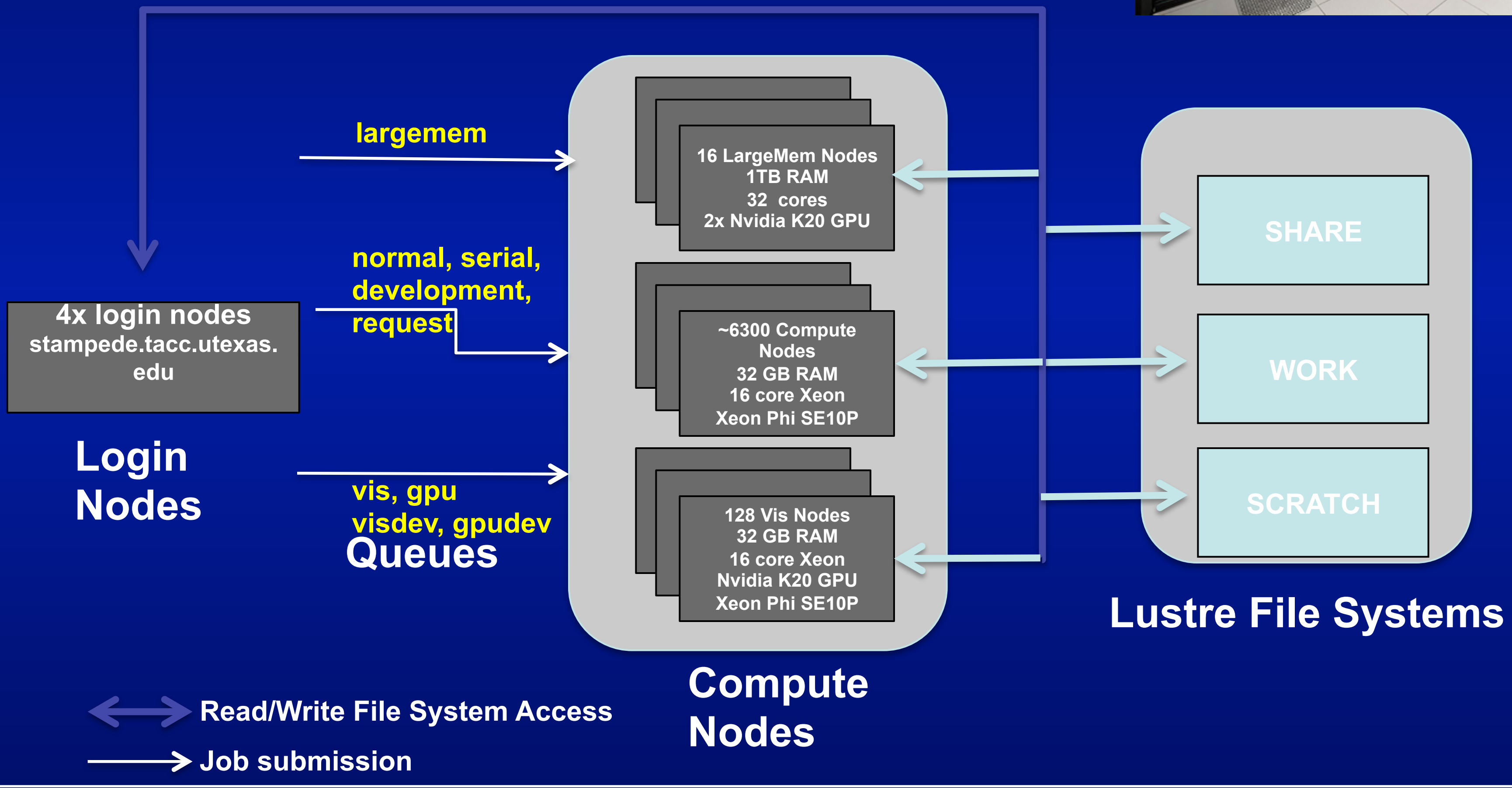
- CPU ray tracing can be made fast, has great weak scaling
  - Ingo Wald papers from 2001-
  - Brownlee et al. EGPGV 13 (fast distributed image-parallel RT)
  - Navratil et al. EGPGV 12 (data-parallel RT at scale)
  - Knoll et al. PacificVis 11 (structured volumes on SMP)
- Nanovol is written in OpenGL
  - Industry standard, but not everyone has a high-end GPU
  - No suitably fast OpenGL/OpenCL/ for CPU's
  - Nanovol limited by volume rendering
  - ray tracing would be nice
- Potential for in situ / in transit vis on HPC systems without GPU's on every node
  - Lots of data movement required for million+ atom MD



TACC

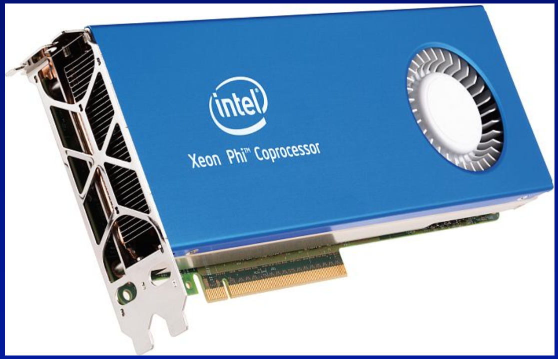
TACC Stampede

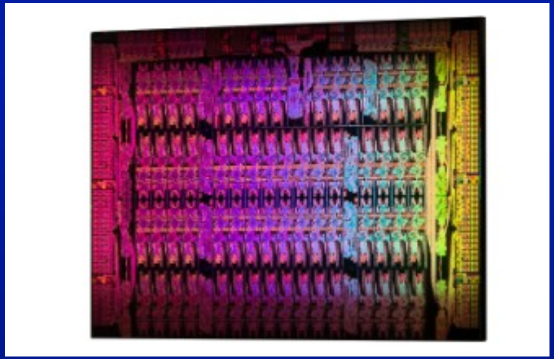
128 GPU's
6400+ dual-SandyBridge CPU's
6400 MIC's (8 GB each)



Intel MIC (Xeon Phi ®)

- Stampede has 6400+ of these (and dual-MIC nodes)
  - Tianhe 2, 3 MIC's per node!
  - Knights Landing: no longer just a discrete GPU!
- SE10P: a special TACC-only preproduction Xeon Phi,
  - 61 cores at 1.1 GHz, 8 GB RAM
  - 16 wide SP/ 8-wide DP SIMD vector instructions
- similar to the 5120D official product
- 1.2 TF theoretical peak – comparable to NVIDIA K20
- How does it stack up in practice?
- 16-wide vector ops are nasty.
  - Intel compiler + OpenMP won't solve this (yet)
  - OpenCL on MIC… not quite.
- We need to write SIMD intrinsics and SOA code for MIC
  - How?
  - Can we re-use SIMD algorithms written for CPU / GPU?





```
_mm_prefetch((const char *)&(a[q+224]), _MM_HINT_T0);
_mm_prefetch((const char *)&(a[q+240]), _MM_HINT_T0);

// For KNF, cheaply emulated to KNC
__m512 a_0 = _mm512_load_ps(&(a[q]));
__m512 a_1 = _mm512_load_ps(&(a[q+16]));
__m512 a_2 = _mm512_load_ps(&(a[q+32]));
__m512 a_3 = _mm512_load_ps(&(a[q+48]));
__m512 a_4 = _mm512_load_ps(&(a[q+64]));
__m512 a_5 = _mm512_load_ps(&(a[q+80]));
__m512 a_6 = _mm512_load_ps(&(a[q+96]));
__m512 a_7 = _mm512_load_ps(&(a[q+112]));

b_0 = _mm512_add_ps(b_0, a_0);
b_1 = _mm512_add_ps(b_1, a_1);
b_2 = _mm512_add_ps(b_2, a_2);
b_3 = _mm512_add_ps(b_3, a_3);
```

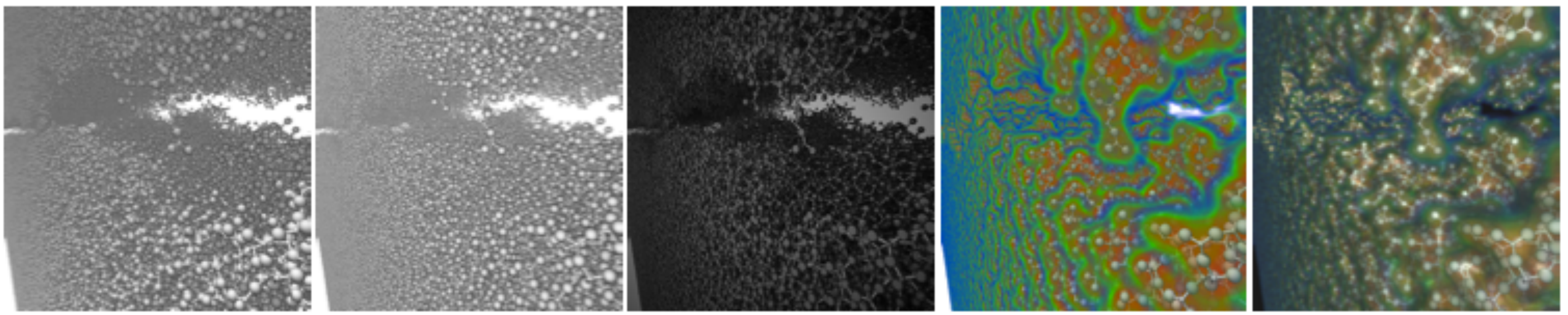
TACC

ISPC and IVL

- Single Process Multiple Data (SPMD) compilers for CPU vector instructions
  - Write "single thread" code once, automatically create vectorized structure-of-arrays (SOA) C++ code with SIMD vector instrinsics
  - similar to GPU languages (OpenCL, CUDA, GLSL)
- Different from GPU's:
  - Abstraction of SIMD intrinsics, not
  - explicit control over "uniform" vs "varying" data across multiple threads
- ISPC: Intel SPMD Program Compiler
  - http://ispc.github.io
  - Official maintained Intel product built on clang/llvm
  - ISPC authors write all backends for you, including a "generic-16" backend for MIC
- IVL: "Ingo Wald's vector language"
  - Built on flex, supports operator overloading, virtual functions
  - Better support/performance on MIC
  - Closed source, but accessible to TACC and ANL collaborators
  - opportunity to write your own intrinsics (non-Intel hardware – BlueGene/ARM?)
- We chose IVL when work on bnsView started…

bnsView

- Uses RIVL (the predecessor of Intel's Embree 2.0 ray tracer)
- Packet-based ray tracer, coherent BVH traversal
- Support for multiple vector backends (SSE, AVX, MIC) using IVL
  - Code runs on Stampede CPUs and MICs, as well as my Mac.
- Started out as a fast ball-and-stick ray tracer
  - Hard shadows, ambient occlusion, full path tracing
- Volume rendering added later



diffuse + shadows 58 fps | **Ambient occlusion 1.8 fps** | **Path tracing ~0.25 fps** | **Volume + shadows (unlit), 16 fps** | **Volume + shadows (lit), 3.4 fps**

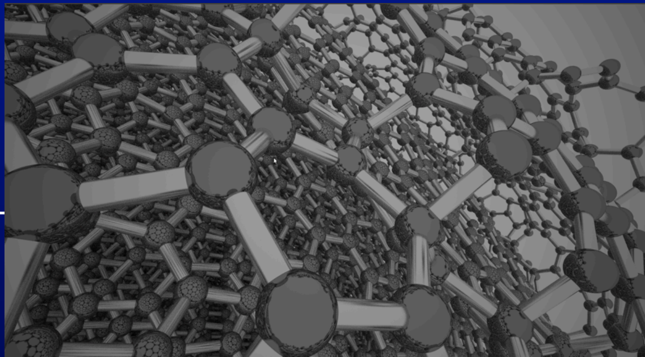
Preprocess

- For each data timestep
  - Read data
  - Create a coarse grid of balls
  - Build sticks
  - Build BVH from both balls and sticks
  - Build structured volume using radial basis functions
  - Build macrocell grid from structured volume (contains min-max values over range)
  - Optionally, offload to MIC using Intel COI libraries

Rendering

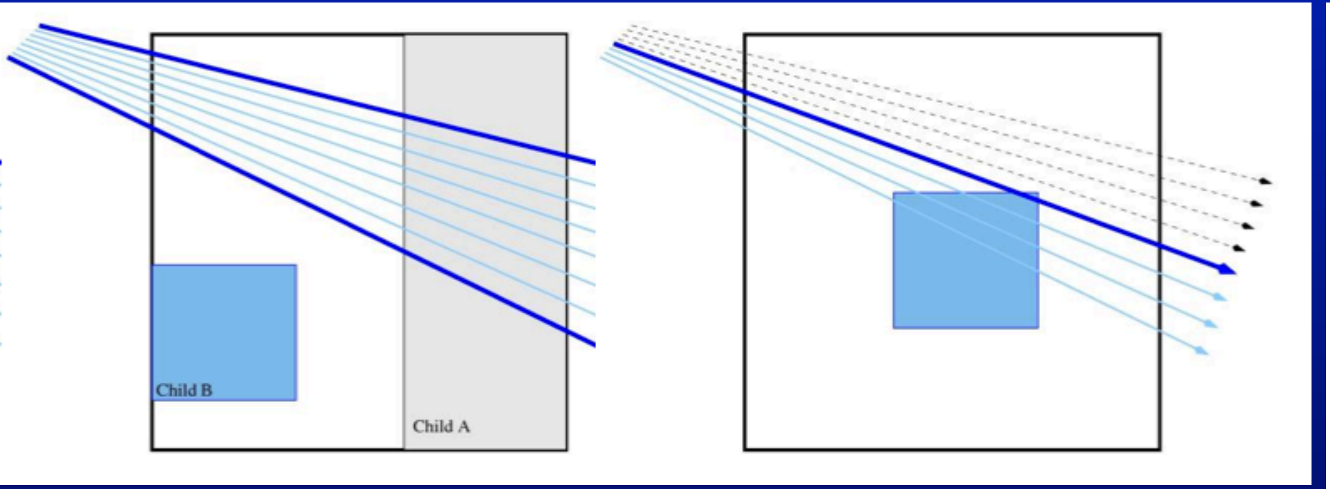
- For each frame
  - Update camera and all user params (transfer function, etc.)
  - Ray generation and distribution (partition a frame buffer into strips of rays determined by SIMD width)
  - As determined by renderer (volume renderer, shadow ray caster, AO renderer, path tracer):
    - while (ray hasn't terminated)
      - trace_ray(), with two separate traversals:
      - Ball and stick ray tracing, using BVH traversal
        - » Computes hit position `t`, hit primitive and `opaque_color`
      - Direct volume rendering, using macrocell grid traversal
        - » Starting from the eye, ending at the opaque hit position
        - » Computes DVR termination position `t`, DVR integrated color
      - Shade this ray, spawn secondary rays or terminate

    Write integrated ray to frame buffer

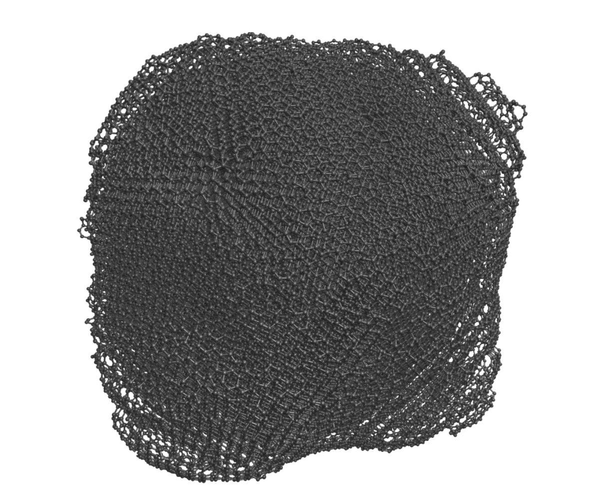


Coherent BVH traversal

- Acceleration structure traversal is the dominant cost for most ray tracing
- Trace packets of rays together: multiple rays, 1 BVH node
  - Fast min/max SIMD intrinsics
  - Exploit memory locality
- BVH is ideal for primitives whose boxes *overlap* (e.g. sticks)



Wald et al. ACM TOG 07



Coherent BVH traversal in IVL

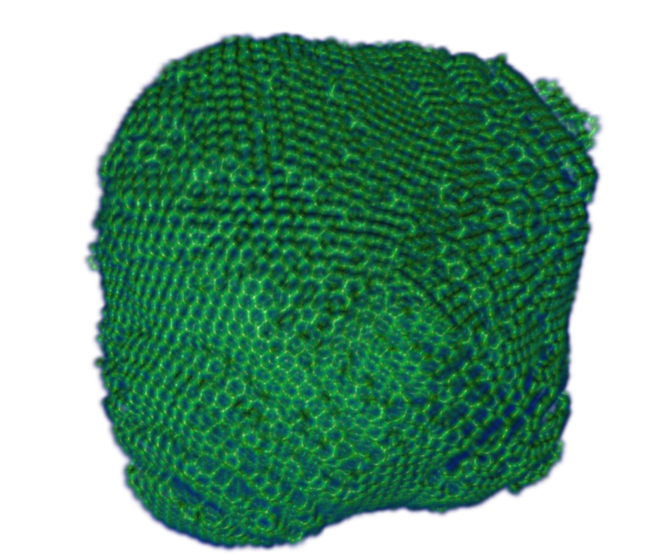
- Fast (Packet) CPU ray tracing algorithms can be written <u>as if for single rays</u>
  - Compiled to multiple vector architectures (SSE, AVX, MIC, potentially BlueGeneP/Q)
- Fuller vector utilization than using OpenMP
  - On MIC, ray-bounding box tests are trivial for 16 rays at once trivial
  - Much cleaner than writing intrinsics
- This would be coded differently on the GPU
  - E.g. Aila & Laine HPG 2009

```
varying bool BNS::traverse_bvh_geometry(varying Ray reference ray)
{
  uniform uint nodeStack[STACK_DEPTH];
  uniform uint stackPtr = 0;
  uniform uint nodeID = 0;
  while (1) {
    const uniform uint count = node[nodeID].count;
    const uniform uint offset = node[nodeID].offset;
    if (count != 0) {
      const uniform uint leafBegin = offset;
      const uniform uint leafEnd = leafBegin + count;
      for (uniform uint itemID=leafBegin;itemID<leafEnd;itemID++) {
        uniform int primID = primIDs[itemID];
        if (primID >= numBalls)
          intersect(ray,stick[primID-numBalls]);
        else
          intersect(ray,ball[primID]);
      }
      if (stackPtr == 0)
        break;
      nodeID = nodeStack[--stackPtr];
    }
    else {
```

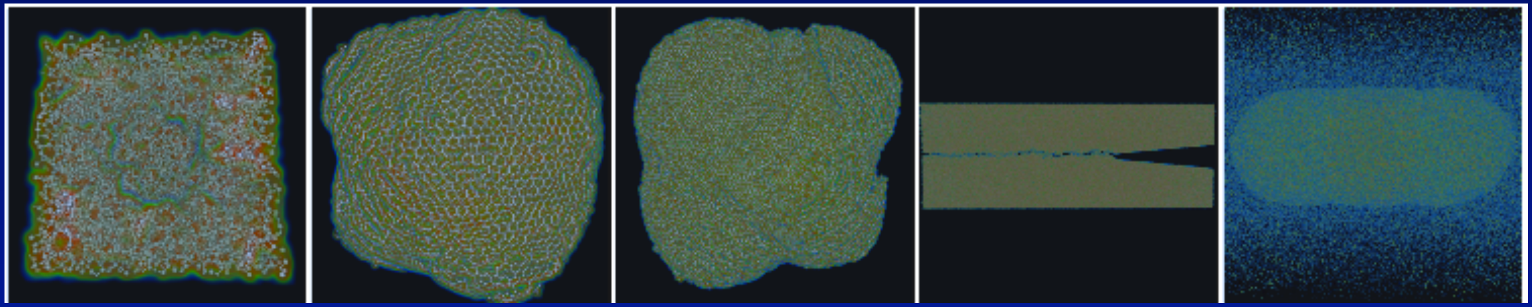
- All ball & stick ray tracing in bnsView uses this.

Volume rendering in bnsView

- Macrocell grid traversal
  - Very similar to nanovol
  - Standard 3D-DDA (e.g., Amanatides and Woo 87)
  - Poor coherence, but grid is coarse enough that it shouldn't matter
    - 1.5x-3x improvements vs without the grid, similar to nanovol
    - Could be improved (coherent grid traversal, Wald et al. SIGGRAPH 06)
- Direct volume rendering
  - Preintegrated transfer function
  - Default step size of 0.5 voxels, uniform sampling
  - Optional gradient shading
- This IVL code looks virtually identical to GLSL.
  - Except we have to write and use our own tex3D() and tex3Dgrad()
  - Compare to GPU built-in 3D texture interpolation
  - Nothing clever being done here (yet) – room for improvement!



GPU vs CPU vs MIC, 1 Stampede vis node
b&s + structured volume rendering

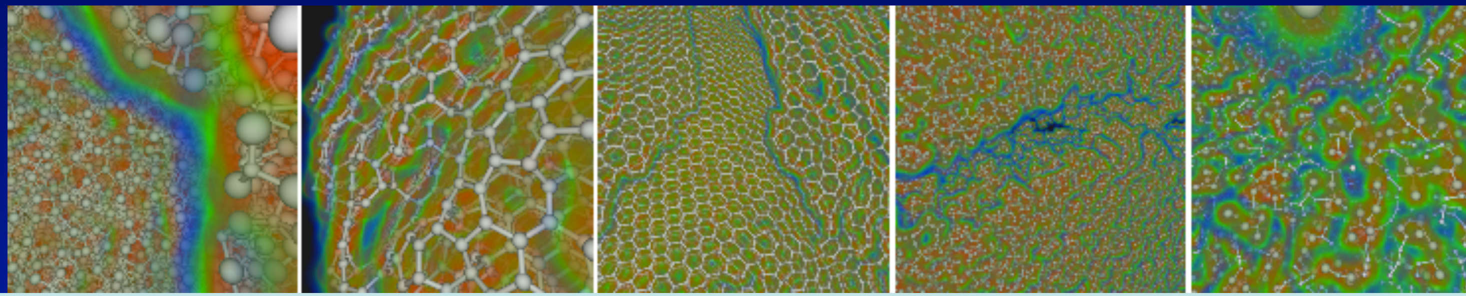


Dataset	*Nanobowl*	*Nanosphere*	*Nanosphere*	*SiO2 fissure*	*ANP3*
#atoms	20K	90K	740K	5M	15M
Size	800 KB	3 MB	40 MB	160 MB	1 GB
Volume size	1.1 MB	11 MB	720 MB	92 MB	263 MB
Voxels/Ang.	4	4	4	1	.5
GPU fps	34	21	7	20	2.63
CPU fps	22	8.6	6.1	13.3	1.31
MIC fps	**71**	**23.3**	**18.1**	**39**	**3.22**
MIC/GPU	2.1x	1.1x	2.5x	2.0x	1.2x
MIC/CPU	3.2x	2.7x	3x	2.9x	2.5x

GPU: NVIDIA K20 (Kepler) GPU (2496 cuda cores)
CPU: dual 8-core 2.7 GHz Intel Xeon E5-2680,
MIC: 61-core SE10P 1.1 GHz Intel Xeon Phi

TACC

GPU vs CPU vs MIC, 1 Stampede vis node b&s + structured volume rendering



Dataset	*Nanobowl*	*Nanosphere*	*Nanosphere*	*SiO2 fissure*	*ANP3*
#atoms	20K	90K	740K	5M	15M
Size	800 KB	3 MB	40 MB	160 MB	1 GB
Volume size	1.1 MB	11 MB	720 MB	92 MB	263 MB
Voxels/Ang.	4	4	4	1	.5
GPU fps	30.5	**29.9**	11.6	24.4	19.5
CPU fps	15	10.2	7.85	7.65	6.4
MIC fps	**46**	28.3	**23.8**	**33**	**28.0**
MIC/GPU	1.5x	.95x	2.1x	1.35x	1.4x
MIC/CPU	3.1x	2.8x	3.0x	4.3x	4.4x

GPU: NVIDIA K20 (Kepler) GPU (2496 cuda cores)
CPU: dual 8-core 2.7 GHz Intel Xeon E5-2680,
MIC: 61-core SE10P 1.1 GHz Intel Xeon Phi

TACC

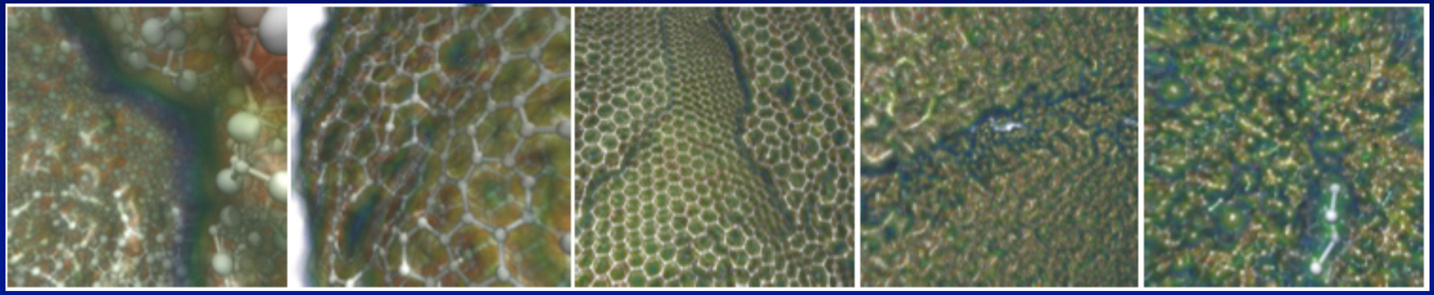
with volumetric lighting (far)…

Dataset	*Nanobowl*	*Nanosphere*	*Nanosphere*	*SiO2 fissure*	*ANP3*
#atoms	20K	90K	740K	5M	15M
Size	800 KB	3 MB	40 MB	160 MB	1 GB
Volume size	1.1 MB	11 MB	720 MB	92 MB	263 MB
Voxels/Ang.	4	4	4	1	.5
GPU fps	**41**	**19.5**	6	19.6	**2.50**
CPU fps	6.15	2.42	1.57	4.51	0.35
MIC fps	36	12.4	**9.98**	**20.3**	1.18
MIC/GPU	.87x	.63x	1.6x	1.03x	.47x
MIC/CPU	5.9x	5.1x	6.4x	4.5x	3.4x

GPU: NVIDIA K20 (Kepler) GPU (2496 cuda cores)
CPU: dual 8-core 2.7 GHz Intel Xeon E5-2680,
MIC: 61-core SE10P 1.1 GHz Intel Xeon Phi

TACC

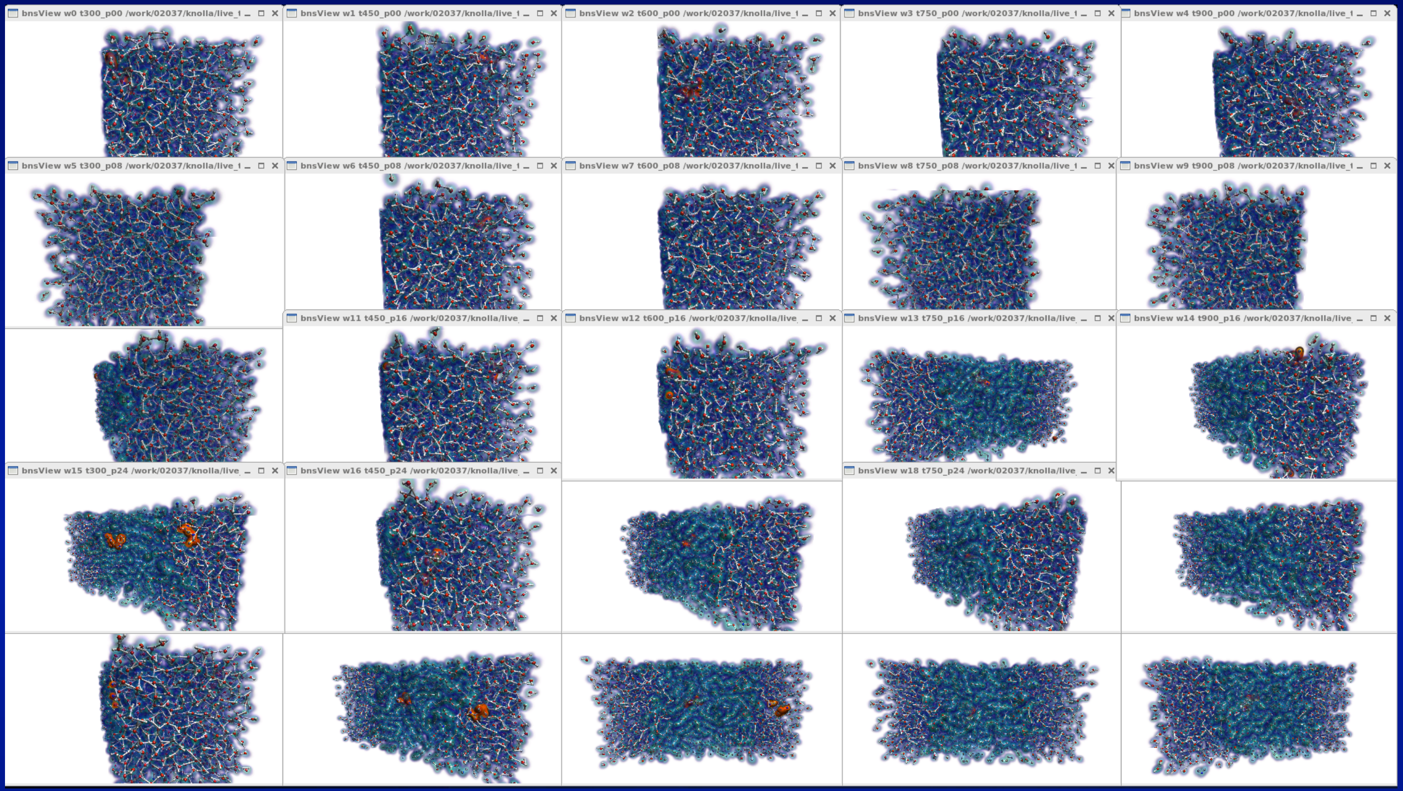
with volumetric lighting (close)



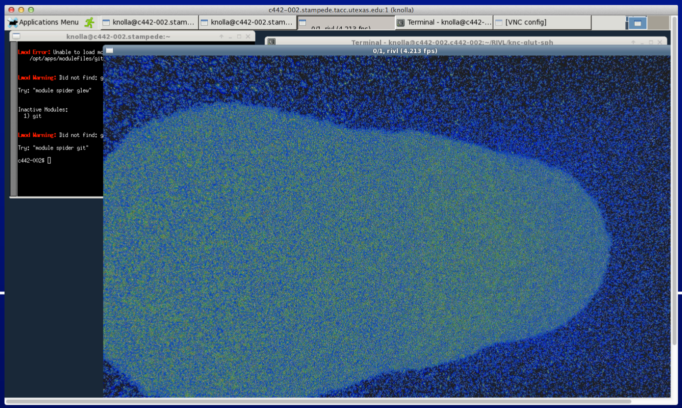
Dataset	*Nanobowl*	*Nanosphere*	*Nanosphere*	*SiO2 fissure*	*ANP3*
#atoms	20K	90K	740K	5M	15M
Size	800 KB	3 MB	40 MB	160 MB	1 GB
Volume size	1.1 MB	11 MB	720 MB	92 MB	263 MB
Voxels/Ang.	4	4	4	1	.5
GPU fps	**32.5**	**26**	10.7	**20.9**	**17.3**
CPU fps	4.02	2.97	2.46	2.02	1.91
MIC fps	22	14.8	**14.1**	10.7	14.1
MIC/GPU	.67x	.56x	1.3x	.51x	.82x
MIC/CPU	5.5x	5.0x	5.7x	5.3x	7.4x

GPU: NVIDIA K20 (Kepler) GPU (2496 cuda cores)
CPU: dual 8-core 2.7 GHz Intel Xeon E5-2680,
MIC: 61-core SE10P 1.1 GHz Intel Xeon Phi

Remote vis with bnsView



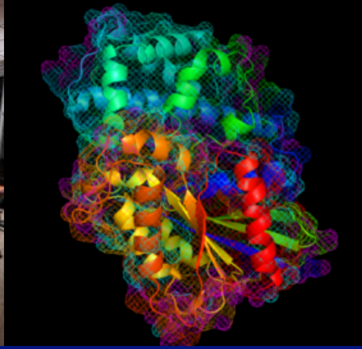
- VNC on Stampede
- DisplayCluster (~20 fps for a 8 MP window)
- Live in-transit demo, Intel booth @ SC13



TACC

Conclusions

- For these similar volume + ball & stick ray casting implementations, MIC is competitive with GPU's
  - CPU also competitive, but suffers from lack of gather
  - Opportuntity for improvement in DVR code, lighting
- Volume rendering is the big bottleneck
  - More so on CPU/MIC
- Potential for in-situ vis on Intel and non-Intel CPU's
- Can programming models be merged?
  - IVL/ISPC language (e.g. `uniform`) syntax needed for performance
  - Syntax is similar, but optimized kernels look very different
  - At least, one can write common host-side code and use either a IVL/ ISPC or GPU render



Thank you!

Aaron Knoll
knolla@tacc.utexas.edu