Scalable high performance visualization on Discovery Cluster.

The session is structured as follows:
• Need for Parallelization in Rendering and Visualization
• The problem of large data sets
• Solutions
• Available software on Discovery for parallel distributed rendering
• Using GPU queue for scalable rendering and visualization
• Using general compute queue for scalable rendering and visualization
• Some example runs on the cluster – GPU queue, general compute queue
• Questions
Why Parallelization in Rendering and Visualization...?
The Data Tsunami

Visualization is needed to analyze the flood of data from petascale computers.
Visualization Allows Us to “See” the Science

Raw Data

01001101011001
11001010010101
00101010100110
11101101011011
00110010111010

Application

Geometric Primitives

Render

Pixels
Huge Data Overwhelms Workstations

How do we solve this problem?

Big Data, Big Memory, Lots of CPU, Lots of Graphics Power
Keep Heat and Noise in the Server Room, Secure

Change the model!
Transition Point Has Been Reached
Time to Send Images Rather than Data
Getting from Data to Insight

Data Representation → Visualization Primitives → Graphics Primitives → Display

Iteration and Refinement
Pipelining v/s Parallelism

- Pipelining: the problem is decomposed into individual steps that are mapped onto processors. Data travel through the processors and are transformed by each stage in the pipeline.
- Pipelining: offers only a limited amount of parallelism
- Pipelining: difficult to achieve good load-balancing amongst the processors in the pipeline as the different functions in the pipeline vary in computational complexity
- Pipelining: offers only a limited amount of parallelism
- Pipelining: for many problems like rendering pipeline, such a partitioning is natural

Solution:

- To overcome such constraints pipelining is often augmented by replicating some or all pipeline stages
- Data are distributed amongst those processor and worked on in parallel
- If the algorithms executed by each of the processors are identical, the processors can perform their operation in lockstep, thus forming a SIMD (single-instruction, multiple-data) engine
- If the algorithms contain too many data dependencies thus making SIMD operation inefficient, MIMD (multiple-instruction, multiple-data) architectures are more useful
Object Partitioning v/s Image Partitioning

- Object-space partitioning is commonly used for the geometry processing portion of the rendering pipeline, as its operation is intrinsically based on objects.
- The data objects distributed amongst parallel processors belonged into object space, e.g., polygons, edges, or vertices.
- Most parallelization strategies for rasterizers employ image-space partitioning.
- Rasterisation: task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer, or for storage in a bitmap file format.
- Compared with other rendering techniques such as ray tracing, rasterisation is extremely fast. However, rasterization is simply the process of computing the mapping from scene geometry to pixels and does not prescribe a particular way to compute the color of those pixels. Shading, including programmable shading, may be based on physical light transport, or artistic intent.
- **Object-space partitioning**
  - Most often used for geometry processing
  - Each task handles part of the scene, e.g. high-level objects, triangles etc.

- **Image-space partitioning**
  - Most often used for rasterization
  - Each task processes parts of the image, e.g. pixels, scanlines, tiles, etc.

---

**Temporal Partitioning**

- **Popular choice for rendering of animations**
- **Process**
  - Broadcast data to all processors
  - Assign frames (i.e. viewpoint etc.) to \( n \) processors
  - Render \( n \) frames
  - Display or store \( n \) frames
- **Considered to be *embarrassingly parallel***
  - Rendering performance scales linearly with number of processors

- But: Data distribution and collection (communication), forms serial overhead and may overshadow computation and rendering ... no speedup & bad scalability!!
Types of Input Data

- **Point / Particle**
  - N-body simulation
- **Regular grid**
  - Medical scan
- **Curvilinear grid**
  - Engineering model
- **Unstructured grid**
  - Extracted surface

**Grid** – regular structure, all voxels (cells) are the same size and shape

**Curvilinear** – regularly grided mesh shaping function applied

**Unstructured grid** – irregular mesh typically composed of tetrahedra, prisms, pyramids, or hexahedra.
Parallel Geometry Processing

- **Geometry Processing**
  - Transformations, Lighting, Clipping, Texture Calculations

- **Round-robin distribution**
  - Each processor works on $1/n$ objects
  - Objects are sent to rasterizers as soon as possible
  - Load balancing problems as objects may require different amounts of work (clipping, lighting, ...)

- **Dynamic assignment**
  - First available processor receives next object

**Limitations of these strategies:**
- Temporal object ordering is lost
Classification by Sorting
a) Sort First
b) Sort Middle
c) Sort Last

Considering the two main steps in rendering, i.e. geometry processing and rasterization, there are three principal locations for the sorting step:
1. Early during geometry processing (sort-first)
2. Between geometry processing and rasterization (sort-middle)
3. And after rasterization (sort-last).
Load Balancing:

Tasks are the basic units of work that can be assigned to a processor, e.g. objects, primitives, scanlines, pixels etc.

Granularity quantifies the minimum number of tasks that are assigned to a processor, e.g. 10 scanlines per processor or 128x128 pixel regions.

Coherence describes the similarity between neighboring elements like consecutive frames or neighboring scanlines. Coherence is exploited frequently in incremental calculations, e.g. during scan conversion. Parallelization may destroy coherence, if neighboring elements are distributed to different processors.

Load balance describes how well tasks are distributed across different processor with respect to keeping all processors busy for all (or most) of the time.

Objective is to ...
- distribute work evenly among all processors
- have all processors finish their work at the same time

(One) Definition of load balance:

\[ LB = \frac{T_f}{T} \]

- \( LB \) Load balance
- \( T_f \) Time fastest processor finishes
- \( T \) Total processing time
Workload Characterization:

Clipping. Objects clipped by the screen boundaries incur more work than objects that are trivially accepted or rejected. It is difficult to predict whether an object will be clipped and load-imbalances can result as a consequence of one processor receiving a disproportionate number of objects requiring clipping.

Tessellation. Some rendering APIs use higher order primitives, like NURBS, that are tesselated by the rendering subsystem. The degree of tessellation, i.e. the number of triangles per object, determines the amount of data expansion occurring during the rendering process. The degree of tessellation is often view-dependent and hence hard to predict a priori.

Primitive distribution. In systems using image-space partitioning, the spatial distribution of objects across the screen decides how many objects must be processed by each processor.

Primitive size. The performance of most rasterization algorithms increases for smaller primitives. (It takes less time to generate fewer pixels.) The mix of large and small primitives therefore determines the workload for the rasterizer.
- **Static**
  - Fixed assignment of tasks to processors

- **Dynamic**
  - On-the-fly assignment of tasks to processors

- **Adaptive**
  - Assign tasks such that all processors have approximately the same load
Static Load Balancing

- All tasks assigned before start of rendering, e.g.
  - Round-robin object assignment in sort-middle architectures
  - Assignment of screen regions to rasterizers (SGI RE/IR, Pixel-planes 4)

- Relies on assumptions about statistics of the model to achieve load balancing, e.g.
  - Most objects require the same amount of work to process
  - Interleaving of pixels will give each processor an equal share of busy and less busy screen regions
    **But**: Reduces coherence between pixels within a processor
  - All frames of an animation incur approximately the same workload
Dynamic Load Balancing

- Task are assigned on demand, i.e. the next task goes to the first available processor
  - Assume that there are more tasks than processors
    \[ \text{Granularity Ratio} = \frac{\#\text{tasks}}{\#\text{processors}} > 1 \]
  - Upper bound for load imbalance is difference between largest and smallest task
  - Simple optimization: (if known) assign largest tasks first

- Task-processor assignment not known a priori
  - Maintains a task list that is depleted by processors
  - Requires dynamic (sic) distribution of tasks during runtime
  - Tasks may not complete in same order as issued
  - Some APIs require temporal ordering
Adaptive Load Balancing

- Create tasks which will require (approximately) the same amount of processing time

**Static adaptive load balancing**
- Predictive: Estimate the processing time for each task
- Reactive: Deduce processing time from previous frame
- Requires separate step to determine task assignments

**Dynamic adaptive load balancing**
- Monitor workload of processors
- Reassign tasks from busy processors to idle processors
- Requires concurrent monitoring process
Adaptive Load Balancing Algorithms:

Usually performed in two steps:

Load estimation (Count primitives per screen region, Estimate cost per primitive)
Work distribution (Subdivide / tile screen to create regions with approx. equal load, or assign fixed-sized regions (cells) to processors to create approximate load balance.

Some algorithms –

Roble’s Method
Whelan’s Method
Whitman’s Method
Muellers’s Method (MAHD)
Ellsworth Method
Princeton Display Wall Method
Parallel Geometry Processing

- **Geometry Processing**
  - Transformations, Lighting, Clipping, Texture Calculations

- **Round-robin distribution**
  - Each processor works on $1/n$ objects
  - Objects are sent to rasterizers as soon as possible
  - Load balancing problems as objects may require different amounts of work (clipping, lighting, ...)

- **Dynamic assignment**
  - First available processor receives next object

- **Limitations of these strategies:**
  - Temporal object ordering is lost
Temporal Ordering

- Required by some APIs, e.g. OpenGL
- Important for algorithms that rely on the order in which objects are drawn onto the screen
  - Non z-buffer hidden surface removal, e.g. painter's algorithm
  - Multi-pass algorithms, e.g. transparency, overlays, solid-modeling, priority algorithm

Possible solutions
- Sequence numbers (time stamps) enforce strict ordering
- Barriers to ensure ordering between groups of objects (see
Visualization Techniques

• Surface Rendering, is an indirect geometry based technique

• Direct Volume Rendering, is a technique for the visualization of 3D scalar data sets without a conversion to surface representations
Surface Shading (Pseudocolor)

Given a scalar value at a point on the surface and a color map, find the corresponding color (and opacity) and apply it to the surface point.

Most common operation, often combined with other ops
Isosurfaces (Contours)

- Surface that represents points of constant value with a volume
- Plot the surface for a given scalar value.
- Good for showing known values of interest
- Good for sampling through a data range
Volume Rendering

Expresses how light travels through a volume. Color and opacity controlled by transfer function. Smoother transitions than isosurfaces.
Clipping / Slicing Planes

Extract a plane from the data to show features
Hide part of dataset to expose features
Particle Traces (Streamlines)

Given a vector field, extract a trace that follows that trajectory defined by the vector.

\[ P_{\text{new}} = P_{\text{current}} + V_P \Delta t \]

Streamlines – trace in space
Pathlines – trace in time
**Medical Data**

- Male Visible Human (512-by-512-by-1877) 500MB CT, 14GB RGB

PVR [Silva-et-al 96]
But what about large, distributed data?
Or distributed rendering?

01001101011001
11001010010101
00101010100110
11101101011011
00110010111010

[Diagram of geometric shapes and a head image]
Or distributed displays?
Or all three?
## Moving Data

- How long can you wait?

<table>
<thead>
<tr>
<th>File Size</th>
<th>10 Gbps</th>
<th>54 Mbps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GB</td>
<td>1 sec</td>
<td>2.5 min</td>
</tr>
<tr>
<td>1 TB</td>
<td>~17 min</td>
<td>~43 hours</td>
</tr>
<tr>
<td>1 PB</td>
<td>~12 days</td>
<td>~5 years</td>
</tr>
</tbody>
</table>
Visualization Scaling Challenges

- Moving data to the visualization machine
- Most applications built for shared memory machines, not distributed clusters
- Image resolution limits in some software cannot capture feature details
- Displays cannot show entire high-resolution images at their native resolution
Visualization scales with HPC

Large data produced by large simulations require large visualization machines and produce large visualization results.

- Terabytes of Data
- AT LEAST Terabytes of Vis
- Resampling, Application, ...
- Gigapixel Images
- Resolution to Capture Feature Detail
Analyzing Data

• Visualization programs **only beginning to efficiently** handle ultrascale data
  – 650 GB dataset -> 3 TB memory footprint
  – Allocate HPC nodes for RAM not cores
  – N-1 idle processors per node!

• Stability across many distributed nodes
  – Rendering clusters typically number N <= 64
  – Data must be dividable onto N cores
GPU Considerations

• Parallelism – kernel should be highly SIMD/SIMT
  – Switching kernels is expensive!
  – Fermi *hardware* supports multiple kernel execution

• Control Flow – avoid conditionals in kernels
  – Implemented with predication, harms utilization

• Job size – high workload per thread + many threads
  – amortize thread initialization and memory transfer costs
  – GPU is a throughput machine, must keep it busy!

• Memory footprint – task must decompose well
  – local store per GPU core is low (16 KB on Tesla)
  – card-local RAM is limited (1 – 4GB on Tesla)
  – access to system RAM is slow (treat like disk access)
Rasterization v/s Ray tracing

• Real time issues
• GPU v/s FPGA
• PowerVR OpenRL ray tracing API
• Ray tracing's popularity stems from its basis in a realistic simulation of lighting over other rendering methods (such as scanline rendering or ray casting). Effects such as reflections and shadows, which are difficult to simulate using other algorithms, are a natural result of the ray tracing algorithm. The computational independence of each ray makes ray tracing amenable to parallelization.
• A serious disadvantage of ray tracing is performance. Scanline algorithms and other algorithms use data coherence to share computations between pixels, while ray tracing normally starts the process anew, treating each eye ray separately. However, this separation offers other advantages, such as the ability to shoot more rays as needed to perform spatial anti-aliasing and improve image quality where needed.
• True photorealism occurs when the rendering equation is closely approximated or fully implemented. Implementing the rendering equation gives true photorealism, as the equation describes every physical effect of light flow. However, this is usually infeasible given the computing resources required.

https://www.youtube.com/watch?v=3yH8OQTKpy8
DISCOVERY CLUSTER

Software on Discovery Cluster:

1) VisIt (https://wci.llnl.gov/simulation/computer-codes/visit/)
2) ParaView (http://www.paraview.org/)
3) Vaa3D (http://home.penglab.com/proj/vaa3d/home/index.html)
4) Tachyon (http://jedi.ks.uiuc.edu/~johns/raytracer/)
5) Other options include parallel visualization and rendering features in commercial packages on Discovery Cluster like Matlab, Mathematica, SAS, Ansys-Fluent.

Hardware on Discovery Cluster:

1) GPU Queue
2) Hadoop HDFS 50TB
3) Large Mem Queue
4) General Purpose Queues 10G and restriced IB
LET'S FOCUS ON PARAVIEW for now and for runs on the DISCOVERY CLUSTER
ParaView Application Architecture

ParaView Client | pvpython | ParaWeb | Catalyst | Custom App
---|---|---|---|---

UI (Qt Widgets, Python Wrappings)

ParaView Server

VTK

OpenGL | MPI | IceT | Etc.
ParaView Architecture

• Three tier
  – Data Server
  – Render Server
  – Client
Standalone

Client

Data Server  Render Server
Client-Server

Data Server

Render Server

Client
Client-Render Server-Data Server
Connecting to a ParaView Server
Data Parallel Pipelines

• Duplicate pipelines run independently on different partitions of data.
Data Parallel Pipelines

• Duplicate pipelines run independently on different partitions of data.
Data Parallel Pipelines

• Some operations will work regardless.
  – Example: Clipping.
Data Parallel Pipelines

• Some operations will work regardless.
  – Example: Clipping.
Data Parallel Pipelines

• Some operations will work regardless.
  – Example: Clipping.
Data Parallel Pipelines

• Some operations will have problems.
  – Example: External Faces
Data Parallel Pipelines

- Some operations will have problems.
  - Example: External Faces
Data Parallel Pipelines

• Ghost cells can solve most of these problems.
Data Parallel Pipelines

- Ghost cells can solve most of these problems.
Data Partitioning

• Partitions should be load balanced and spatially coherent.
Data Partitioning

• Partitions should be load balanced and spatially coherent.
Data Partitioning

• Partitions should be load balanced and spatially coherent.
Load Balancing/Ghost Cells

• Automatic for Structured Meshes.
• Partitioning/ghost cells for unstructured is “manual.”
• Use the D3 filter for unstructured
  – (Filters → Alphabetical → D3)
Job Size Rules of Thumb

• Structured Data
  – Try for max 20 M cell/processor.
  – Shoot for 5 – 10 M cell/processor.

• Unstructured Data
  – Try for max 1 M cell/processor.
  – Shoot for 250 – 500 K cell/processor.
Avoiding Data Explosion

• Pipeline may cause data to be copied, created, converted.

• This advice only for dealing with very large amounts of data.
  – Remaining available memory is low.
Topology Changing, No Reduction

- Append Datasets
- Append Geometry
- Clean
- Clean to Grid
- Connectivity
- D3
- Delaunay 2D/3D
- Extract Edges
- Linear Extrusion
- Loop Subdivision
- Reflect
- Rotational Extrusion
- Shrink
- Smooth
- Subdivide
- Tessellate
- Tetrahedralize
- Triangle Strips
- Triangulate
Topology Changing, Moderate Reduction

- Clip
- Decimate
- Extract Cells by Region

- Extract Selection
- Quadric Clustering
- Threshold

Similar: Extract Subset
Topology Changing, Dimension Reduction

- Cell Centers
- Contour
- Extract CTH Fragments
- Extract CTH Parts
- Extract Surface
- Feature Edges
- Mask Points
- Outline (curvilinear)
- Slice
- Stream Tracer
Adds Field Data

- Block Scalars
- Calculator
- Cell Data to Point Data
- Compute Derivatives
- Curvature
- Elevation
- Generate Ids
- Gen. Surface Normals
- Gradient
- Level Scalars
- Median
- Mesh Quality

- Octree Depth Limit
- Octree Depth Scalars
- Point Data to Cell Data
- Process Id Scalars
- Random Vectors
- Resample with dataset
- Surface Flow
- Surface Vectors
- Texture Map to...
- Transform
- Warp (scalar)
- Warp (vector)
Total Shallow Copy or Output Independent of Input

- Annotate Time
- Append Attributes
- Extract Block
- Extract Datasets
- Extract Level
- Glyph
- Group Datasets
- Histogram
- Integrate Variables
- Normal Glyphs
- Outline

- Outline Corners
- Plot Global Variables Over Time
- Plot Over Line
- Plot Selection Over Time
- Probe Location
- Temporal Shift Scale
- Temporal Snap-to-Time-Steps
- Temporal Statistics
Special Cases

• Temporal Filters
  – Temporal Interpolator
  – Particle Tracer
  – Temporal Cache

• Programmable Filter
Culling Data

• Reduce dimensionality early.
  – Contour and slice “see” inside volumes.

• Prefer data reduction over extraction.
  – Slice instead of Clip.
  – Contour instead of Threshold.

• Only extract when reducing an order of magnitude or more.
  – Can still run into troubles.
Culling Data

• Experiment with subsampled data.
  – Extract Subset

• Use caution.
  – Subsampled data may be lacking.
  – Use full data to draw final conclusions.
Memory Inspector
Rendering Modes

• Still Render
  – Full detail render.

• Interactive Render
  – Sacrifices detail for speed.
  – Provides quick rendering rate.
  – Used when interacting with 3D view.
Level of Detail (LOD)

- Geometric decimation
- Used only with Interactive Render

Original Data
Default Decimation
Maximum Decimation
3D Rendering Parameters

Edit → Settings, Render View

---

**Geometry Mapper Options**
- Enable use of display lists when rendering surface meshes.

**Interactive Rendering Options**
- Set the data size (in megabytes) beyond which to employ decimation, if possible, to speed renders when interacting. 0 implies the use of decimation for all interactive renders.

**Remote/Parallel Rendering Options**
- Set the data size (in megabytes) beyond which to render data remotely (or in parallel) when connected to parallel rendering capable server. In that case the rendered images are delivered to the client. Otherwise, the geometry is delivered to the client and rendering happens locally.

**Miscellaneous**
- When opening a dataset or creating a new filters, use Outline representation, if possible, when showing datasets with number of cells greater than thus threshold (in millions).
Parallel Rendering
Parallel Rendering
Tiled Displays
Image Size LOD

- ParaView’s parallel rendering overhead proportional to image size.
- Can use smaller images for interactive rendering.

Original Data  |  Subsample Rate: 2 pixels  |  Subsample Rate: 4 pixels  |  Subsample Rate: 8 pixels
Parallel Rendering Parameters

Edit → Settings, Render View
Parameters for Large Data

- Use display lists for GPU, off for CPU.
- Try outline instead of decimated geometry.
  - Also try lowering decimation factor.
- Avoid shipping large data back to client.
- Turn on subsampling.
- Image Compression on.
Parameters for Low Bandwidth

• Try larger subsampling rates.
• Try Zlib compression and fewer bits.
Parameters for High Latency

• Turn up Remote Render Threshold.
  – Allow more data to go to client.

• Play with the LOD Threshold and LOD Resolution to control geometry sent to client.

• Turn on Use outline for LOD rendering if decimated geometry too big for client.

• Try turning on interactive render delay.
Vector format image captures

- File->Export Scene->EPS ...
  Dumps image out in vector (scalable) format ideal for inclusion in technical papers

- File->Export Scene->WebGL ...
  Dumps out a web page that you can view in a browser and easily share.
LET'S RUN SOME PARAVIEW EXAMPLES ON DISCOVERY CLUSTER
ABOVE: Asteroid Golevka measures about 500 x 600 x 700 meters. In a CTH shock physics simulation on GPU nodes, a 10 Megaton explosion was initiated at the center of mass. The simulation ran for about 15 hours on 7200 nodes of Red Storm (Sandia National Labs) and provided approximately 0.65 second of simulated time. The resolution was 1 meter, with a 1 cubic kilometer mesh that contained 1.1 billion cells. The remarkable resolution of this simulation provides realism in crack formation and propagation not seen in lower-resolution models.