
CS770/870 Fall 2008

Parallel Graphics

Bartz and Silva, Rendering and Visualization in Parallel Environments, EG 2001 Tutorial.

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

1

Preview

- Parallel environment characteristics
 - Architecture Taxonomy
 - Memory models
 - Programming models
 - Massive parallelism v. small scale parallelism
 - Graphics processor parallelism
- Classic algorithms
 - Polygon rendering (visibility, lighting)
 - Ray tracing
 - Radiosity
 - Volume rendering

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

2

Architecture Taxonomy

- Instruction stream and data stream are independent options for parallelization. This leads to 4 possible architectures:
 - SISD: Single Instruction, Single Data
 - the classic example was a desktop workstation
 - SIMD: Single Instruction, Multiple Data
 - array processing supercomputer
 - MIMD: Multiple Instruction, Multiple Data
 - most common parallel computing architecture; includes multiprocessing systems and workstation farms
 - MISD: Multiple Instruction, Single Data
 - not a viable model (unless you want to use this classification for *pipelining*, which is commonly used inside CPUs)

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

3

Memory Models

- Shared memory
 - All processors have direct (and equal) access to all memory; SMP: Symmetric Multiprocessing
 - Simplifies programming (or compiling)
 - Not scalable
- Distributed memory
 - Each processor has its own local memory; if needed data is somewhere else, must copy it, typically closer to I/O or network speed, rather than memory speed.
- Non-uniform Memory Access (NUMA)
 - Hybrid; some memory local, rest “appears” to be local; analogous to virtual memory.

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

4

Architecture Implications

- MIMD model for architecture
 - encourages the programmer to think in *coarse-grained* parallelism: large chunks of code that run in parallel
 - need to synchronize the execution of these chunks
 - code that is very hard to debug, especially with *race* conditions
 - code itself, however, is pretty familiar
- SIMD model leads to
 - more *fine-grained* parallelism in the program
 - often represents a novel program design challenge to get an effective solution

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

5

Memory Implications

- Shared memory is very attractive from a programming point of view
 - it's what we are used to, so it's familiar
 - parallelizing existing sequential algorithms is often easy
- Distributed memory often requires significant revision of existing sequential code to parallelize
 - inefficient use of the distributed memory can have a major impact on performance
- NUMA models can be a nice compromise
 - Memory *looks* as if it's shared, but it's not
 - but performance may be badly affected if program doesn't access remote memory efficiently

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

6

Message-based Programming Model

- Message Passing Interface (MPI)
 - processes exchange messages when information needs to be shared
 - motivated by cluster computing, but works on shared memory machines as well
 - However, algorithm must be tailored to the distributed model, which is the usually the more complex implementation
 - application programmer must design and program the parallelization explicitly

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

7

Thread-based Programming Model

- Threads are based on a shared memory model
 - One process, many execution threads
 - Inexpensive implicit communication
 - Only efficient with shared memory
 - Can be implemented on distributed memory however
 - Generally easier to program than MPI
- Very appropriate for low scale parallelization such as is becoming available in commodity workstations these days (dual quad cores, e.g.)

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

8

Parallelism Scale

- Large scale parallelism
 - Thousands of processors are pretty common now; at this scale it's by far easier to think about massively parallelizing the data stream than massively parallelizing the instruction stream
 - This leads to coarse-grained parallelism from the program point of view: relatively large computational units applied to thousands of data streams
- Small scale parallelism (dual quad cores, e.g.)
 - Too coarse a program view can lead to idle time in some or many processors, so
 - tend to partition computation more finely

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

9

Graphics Processors

- Today's graphics processors are highly parallelized at a fine scale: they have components that implement:
 - pipelining
 - parallel matrix operations
 - parallel pixel operations
 - and more

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

10

Parallel Algorithm Development

- For any problem, need to address 4 key issues:
 - what computation units should be distributed to what nodes?
 - Not an issue for SIMD
 - how does a processor get the data needed to complete each computation unit?
 - a major issue for distributed data systems
 - for very large problems, this becomes an I/O issue for shared memory systems
 - how might these decisions change over time?
 - *temporal parallelism*
 - load balancing
 - total time is time of the longest processing path
 - static vs. dynamic load balancing
 - computation vs. communication

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

11

Temporal Parallelism

- Can seem embarrassingly parallel if there is enough memory
- Consider a simple animation strategy
 - given n processors, distribute every n th frame to a processor
 - if each frame takes 1 second, only need 12 (or so) processors to get real-time animation
- Problems
 - shared memory: might need far more data than is available
 - distributed memory: distribution costs can be very heavy, need to assemble resulting image at display node
 - naïve approach loses frame-to-frame coherence

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

12

Rendering Pipeline

- Any graphics rendering can be represented by a (very simplified) processing pipeline:



- All the stages are candidates for parallelization
 - *Application*: generate object specifications
 - *Geometry*: transformation of objects to npc; clipping; lighting; texture calculations (but not mapping)
 - *Rasterization*: convert npc-specified objects to pixels
 - modern graphics cards are highly parallelized
 - *Display*: parallel display occurs with *wall* displays

Classic Parallel Graphics Algorithms

- Many of the classic graphics algorithms are “embarrassingly” parallel with shared memory systems, but present real challenges with distributed memory
- Let’s consider the classic algorithms for
 - polygon rendering (visible surface removal)
 - ray tracing
 - radiosity
 - volume rendering

Polygon Rendering

- Mapping objects to pixels and visible surface problem
- Characterized by identifying when to sort the polygons (or parts of polygons) usually in *z*-order or *xy*-order
- *sort-first*: prior to geometry processing
 - partition screen; determine object space region that projects to each image region; sort objects into those regions
- *sort-middle*: between geometry and rasterization
 - sort in image space, usually also by screen region
- *sort-last*: after rasterization
 - essentially *z*-buffer and compositing

Polygon Rendering Data Issues

- Data distribution is primarily an distributed memory issue
- *sort-first*: easy
 - distribute objects by their spatial region
 - can even do it dynamically; could have entire geometry database on local disk; each node only reads what it needs
- *sort-middle*: just a bit more complicated
 - could do *sort-first* pre-processing to identify spatial regions and distribute based on that
- *sort-last*: can do almost any kind of partitioning

Ray Tracing: Parallelize on Rays

- Shared memory
 - embarrassingly parallel
 - distribute rays to processors
 - load balancing best if distribute rays alternately, or even randomly
 - easy and good if all processors have access to all objects
- Distributed memory
 - partition space; distribute data by partition
 - partitions should be parallelepipeds, but could be sized so each partition has about the same # of objects
 - rays get passed from node to node as they leave a partition
 - very heavy communications costs
 - can be somewhat reduced by bundling ray passing
 - every node does computation for all rays, then passes them all at once to all neighbors; hard to balance load

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

17

Ray Tracing Optimizations

- Multiresolution object representation
 - Create spatial partitioning (octree or uniform grid)
 - Create rough bounding volume representations of the scene that is available to every processor; this is a low resolution representation of the scene
 - Distribute the the full resolution scene and the rays among the nodes
 - When tracing a ray in one node, it only needs to send the ray to its neighbor if it intersects a low resolution object in the neighbor
 - Reduces ray distribution at run-time, but still an issue
- Lots of other ideas

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

18

Radiosity

- Spatial partitioning works somewhat better than ray tracing with a notion of *virtual wall*
 - partitioning creates a wall between regions that is divided into patches
 - these patches gather light on one side and shoot on the other side
 - Works really well on shared memory system
 - Would work well if there were some shared and some distributed memory; the shared memory would have the virtual walls
 - Communications is more efficient than sending rays

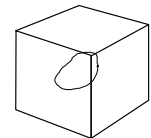
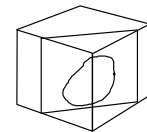
09:55

CS770/870 Fall 2008 Bergeron/Plumlee

19

Volume Visualization Techniques

- Planar Slicing
 - move slice through space
- Isosurface —surface from equal valued cells
- Direct volume rendering
 - change value over time
 - viewing “gas” using color/opacity
 - ray casting and splatting



09:55

CS770/870 Fall 2008 Bergeron/Plumlee

20

Volume Data

- Volume data is a set of data points in 3D
 - regularly spaced sampling is common from medicine
 - irregular sampling sometime occurs with finite element analysis problems
- Assume sampling from a *continuous* phenomena
- Regular sampling leads to division of volume into rectilinear *voxels* (volume data elements)
 - sometimes view the sample value as the center of a voxel, sometimes as a corner

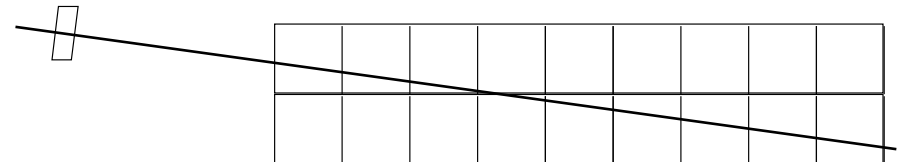
09:55

CS770/870 Fall 2008 Bergeron/Plumlee

21

Direct Volume Rendering Ray Casting

- Figure out how each pixel is built from data values
- Ray intersects each voxel
 - value inside voxel is a *density*
 - distance ray travels through voxel determines *opacity* that is added to pixel's opacity value
 - if pixel opacity reaches 1, ray traversal terminates



09:55

CS770/870 Fall 2008 Bergeron/Plumlee

22

Direct Volume Rendering Splatting

- Figure out how each data value contributes to each pixel
- Treat each voxel as a solid object, project its faces onto the display area (from back to front)
 - the color and opacity of the projected polygons are determined from the voxel's values
 - the projected polygons are *composited* according to their depths, opacities and colors

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

23

Parallel DVR: Ray casting

- Ray casting is much simpler than ray tracing
- No bouncing
- If view is along major axis, data partitioning is trivial
 - problem is embarrassing parallel in both shared and distributed memory systems
- Lacroute developed a very efficient mechanism, called *shear warp* for transforming a volume with an oblique view to a new volume with a view along principal axis

09:55

CS770/870 Fall 2008 Bergeron/Plumlee

24

Parallel DVR: Splatting

- Partition space by uniform grid
 - each partition does the splatting of its voxels onto a virtual image
 - all the virtual images are then splatted together via compositing
 - it's especially straightforward with view along a principal axis, but can also be done with an arbitrary view

Review

- Very superficial review
- Parallel computation environments
- Parallel algorithm development
- Parallelization of graphics rendering algorithms
 - polygon rendering
 - ray tracing
 - radiosity
 - direct volume rendering