Image Compositing - Software

Dale Beermann

- Depth Complexity in Object-Parallel Graphics Architectures
- Pixel Merging for Object-parallel Rendering: A Distributed Snooping Algorithm
- Parallel Volume Rendering Using Binary-Swap Compositing
- A Comparison of Parallel Compositing Techniques on Shared Memory Multiprocessors

Depth Complexity in Object-Parallel Graphics Architectures

- Michael Cox, Pat Hanrahan - 1992
  – An analysis of average depth complexity in a scene in order to find a place to look for a better algorithm for compositing of pixels after object-parallel rendering.
    - Such an algorithm is presented in the next paper.

  – Model is based on graphics primitive size distribution.

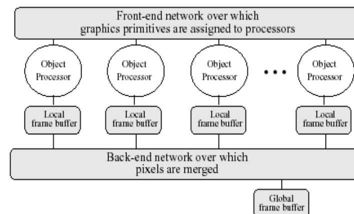
  – Use RenderMan and a software implementation of SGI's GL to generate traces of the pixels generated during the rendering process (I won't go into this much).

Some Definitions

- Vertical Parallelism
  - also known as instruction-level parallelism and pipelining
- Horizontal Parallelism
  - also known as data parallelism, multiprocessing, and parallel processing.
  - Two Types of Horizontal Parallelism:
    - Image Parallelism
      - also known as screen-space parallelism
      - each processor is assigned a region of the screen and must render all primitives that would fall in that region - partial local frame buffers are created
    - Object Parallelism
      - also known as image composition
      - primitives are assigned round-robin and a full frame buffer is created
      - The Problem: Pixel Merge
        - pixels rendered by all processors must be merged in the end

Object Parallelism

- The object parallel architecture has three phases:
  - object assignment to processors (round robin)
    - done by the *front-end network*
  - rendering (done locally on individual processors)
    - performs local hidden surface removal with z-buffering
    - produces a local frame buffer
  - pixel merging
    - done by the *back-end network*
    - composite local frame buffers from all processors
    - produces a global frame buffer with the final composited image



Some More Definitions

- *depth* at (x,y)
  - the number of primitives in a graphics scene to be rendered to this specific (x,y) position
- *depth complexity distribution* or *depth complexity*
  - in a graphics scene, this is the distribution of depth over the whole image (all (x,y) coordinates).
  - will use subsets of the scene and refer to the depth complexity of these subsets as well
    - ex. the 1/n primitives a processor may receive

Implications of Depth Complexity

- Scenes with larger depth complexity require more back-end pixel traffic.
  - Scenes require more processing for z-buffering when there is a large depth complexity because of increased comparisons.
  - Removing depth complexity at the local frame buffer will reduce the bandwidth required by the back-end network.
    - when done by local processors, it can be done in parallel
    - depending on the back-end network, there is a limited amount of parallelism that can be achieved (other papers acknowledge these algorithms)

The Analytic Model (More Definitions)

$A =$ the area of the screen, in pixels.

$R =$ the set of graphics primitives to be rendered.

$N = |R|$.

For each $r \in R$, $|r| =$ the number of pixels to which graphics primitive r renders. That is, $|r|$ is the *projected size* of primitive r.

$R_k = \{r | \; |r| = k\}$; that is, R_k is the subset of R comprising primitives of size k.

$N_k = |R_k|$.

$n =$ the number of processors on which the scene is rendered

$p_d =$ the probability that (after rendering) the depth at an arbitrary pixel in an arbitrary processor's local frame buffer is exactly d. When $n = 1$, this is the probability that the depth at an arbitrary pixel from a rendering of the initial graphics scene is exactly d.

The Analytic Model continued

- Assumptions:
  - screen coverage by primitives is uniformly and randomly distributed
  - primitives are uniformly and randomly distributed to processors
- A starting point (my attempt to explain the math):
  - The Generating Function for probability distribution
    - The probability that a primitive r with size k renders to (x,y) is k/(An)
      - this is just the size of the primitive divided by the frame buffer area times the number of processors
    - $H_k(v) = (1 - (k/An)) + (k/An)v$
    - The coefficient of v_o is the probability that r does not render to (x,y)
      - (1 - (k/An))
    - The coefficient of v_1 is the probability that r does render to (x,y)
      - (k/An)

The Analytic Model continued (some more math)

- Generating functions can be multiplied to express convolution of the probability density functions (yeah I copied that from the paper).
  - So the coefficient of v^2 in $H_k(v)^2$ is the probability the *two* primitives of size k render to (x,y)
  - Moving along, $H_k(v)^{N_k} = [(1 - (k/An)) + (k/An)v]^{N_k}$
    - N_k is the number of primitives with size k
    - So the probability that d are rendered to (x,y) is represented by the coefficient of v^d
- Now we can find the generating function of the probability density of depth complexity by finding the product over all possible primitive sizes.

$$G(v) = \prod_{k=1}^{A} \left[\left(1 - \frac{k}{An}\right) + \frac{k}{An} v \right]^{N_k}$$

The Analytic Model continued (some more math)

- The probability p_d that an arbitrary pixel in an arbitrary processor's local frame buffer has depth exactly d is the coefficient of v^d of G(v).
- Since they tell you to look in the appendix for this one, so will I.
  - For all d, d > 0:

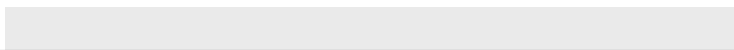
$$p_d = \sum_{j=0}^{d-1} \frac{(-1)^j}{d} \left[\sum_{k=1}^{A} N_k \left(\frac{k}{An-k}\right)^{j+1} \right] p_{d-1-j}$$

  - and

$$p_0 = \prod_{k=1}^{A} \left(1 - \frac{k}{An}\right)^{N_k}$$

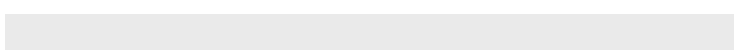
The Analytic Model continued

- The point?
  - With the graphics primitive size distribution for a given scene and the number of processors, they can predict the depth complexity probability density function.
  - Then they can also predict the expected depth complexity and variance, and, what they are really aiming at, the expected number of active pixels per local frame buffer.
  - I omit the rest of the math
- Some notes, p_0 is the probability that no primitive renders to some pixel
- Pr[arbitrary pixel (x,y) is active] is simply $(1-p_0)$



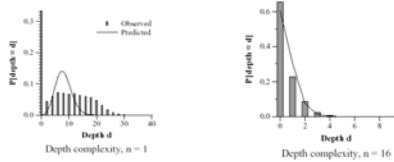
The Analytic Model continued

- Answers to some questions:
  - A sparse frame buffer is one for which a small portion of the frame buffer has had primitives rendered to it.
    - it is pointless to send all of the data across the wire when you only need to send a small portion of it
    - active vs. inactive pixels denote which ones have had primitives rendered to them



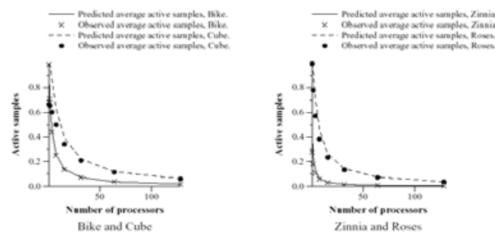
Some Depth Complexity Results

- As the number of processors increase, the depth complexity goes down
  - with 32 processors, for the Wash.ht scene, the depth complexity was never greater than 1
- A problem: not all graphics primitives are uniformly distributed
  - *cube* - for small numbers of processor, the model does not fit, but as the number of processors increases, the primitives are distributed better, getting rid of spatial coherence, and the model looks much nicer



Some Depth Complexity Results

- Screen coverage steeply declines when primitives are distributed across many processors.
  - most of the frame buffers represent unused hardware.
    - there is a potential here for reduced cost
    - moving only those pixels rendered could help a lot
- Average active sample prediction was very good.



Depth Complexity Conclusions

- The fit of the model for uniprocessors can be good, but is much better for multiprocessors where primitives are distributed widely resulting in less spatial coherence among primitives for a processor.
- Again, the model predicts the number of active samples for a scene pretty well.
- Z-Buffering pixels locally has much less of an advantage when there are few processors as opposed to when there are many.
- Basically, there has to be some way to take advantage of the sparse frame buffers in not transmitting so much pixel data.

Pixel Merging for Object-Parallel Rendering: A Distributed Snooping Algorithm

- Michael Cox, Pat Hanrahan - 1993
  - Analyzes and compares a few different pixel merging algorithms and presents a new distributed snooping algorithm.
  - A good part of the paper is based on the previous one.
    - Describe different types of parallelism are described, as well as the pixel-merge problem
    - Describe use of RenderMan and GL for tracing pixels
    - Define the same bunch of terms

$d_{x,y}$ = the *depth* at pixel location (x, y).

$\bar{d}$ = the *depth complexity* of a given scene; that is the average depth over the screen.

p_d = the probability that at arbitrary pixel location (x, y), $d_{x,y} = d$.

A = the resolution of the screen or window, in pixels.

n = the number of processors in the multiprocessor.

A Straightforward Pixel Merging Algorithm

- Each node is assigned a subset of A (screen area) and is responsible for any pixels in that subset.
  - either as the node renders them or after it is done rendering all pixels.
    - could z-buffer pixels locally so it didn't have to send so many off
    - simulations show that few pixels will be deleted locally, so there really isn't much of a benefit
- After *any* node finishes rendering a pixel, those pixels are sent to the node responsible for them.
- Each node then z-buffers the pixels it receives and the images from each node can be tiled.

A Straightforward Pixel Merging Algorithm cont.

- Computing the network traffic is straight forward
  - $d_a A$ pixels must be sent, received and compared, this is the total network traffic
  - this is simply the average depth of the pixels multiplied by the screen area

The PixelFlow Algorithm

- Each processor has an associated full frame buffer it renders to.
- After rendering, the first node sends its pixels to the second node, which performs z-buffering. The second node then sends its pixels to the third, etc.
  - Parallelism is taken advantage of, so processors aren't waiting while others are busy sending and z-buffering (once they've sent their pixels they are inactive though).
- The final node will have the final image.

The PixelFlow Algorithm

- The total number of pixels transferred will be nA, where n is the number of processors in the system.
  - bandwidth required on each link is only A
- In aggregate, O(nA) processing is required to read, compare, send and receive pixels

A Distributed Snooping Merge Algorithm

- Each processor renders to a local frame buffer all the primitives it has received.
- n processors share a global frame buffer to which pixels can be broadcast.
- Processor 1 broadcasts all of its pixels to the global frame buffer while everyone else "listens" to the broadcast.
  - keeps track of active vs. inactive pixels
- For processors that have an active pixel for each (x,y) another processor broadcasts, they do a z-buffer check and discard the pixel if the need to.
- The global frame buffer z-buffers the incoming pixels and ends up with the final image.

A Distributed Snooping Merge Algorithm cont.

- Answers to some questions:
  - Pixels that are broadcast include (screen-x, screen-y, eye-coordinate-z, red, green, blue, *parent-primitive)
- Calculating expected network traffic:
  - The first processor sends its pixels with probability 1, the next sends each pixels with probability 1/2, etc.
  - so Expected Traffic at (x,y) is

$$ET_{x,y} = \sum_{1 \le d \le d_{x,y}} 1/d = H_{d_{x,y}}$$

  - from the last paper, they use the distribution of depth, p_d, to find the expected traffic from all locations

$$ET = A \sum_{1 \le d} p_d H_d$$

A Distributed Snooping Merge Algorithm cont.

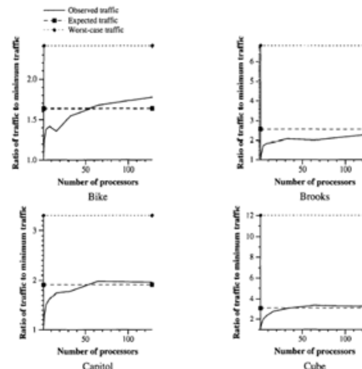
- They bound the equation for Estimated Traffic:

$$ET \leq A\left[(1-\alpha)H_{\lfloor \bar{d} \rfloor} + \alpha H_{\lceil \bar{d} \rceil}\right]$$

- Basically, the idea is that you take the lower bound (floor) and use it for the probability that the pixel won't be rendered (1-alpha). Similarly, you take the upper bound (ceiling) and use it for the probability that the pixel will be rendered.
- Alpha is the (average depth complexity - the floor of the average depth complexity). I don't know exactly what that means.
- So, as the depth increases, the ET grows about at the same rate as log() of the average depth because the Harmonics grow so slowly.

A Distributed Snooping Merge Algorithm Results

- Expected traffic is close to Observed traffic for most scenes.
- Minimum traffic is as observed on a uniprocessor, so the ratio is the comparison for multiprocessors to the uniprocessor result.
- Ratio grows because z-buffering has been ignored. From before, it can be noted that it can be ignored for many processors, but is not an accurate assumption for few processors.



Algorithm Comparisons

d_a represents average depth

Uniprocessor z-buffering

- d_aA pixels must be read and compared
- $\log(d_a)A$ pixels must be written
- total cost is $O(d_aA + \log(d_a)A)$

Simple Algorithm

- aggregate network traffic is d_aA pixels
- $O(d_aA)$ processor cycles required to route, send, receive, compare

PixelFlow Algorithm

- total bandwidth required is nA pixels
- $O(d_aA)$ processor cycles required to route, send, receive, compare

Snooping Algorithm

- requires network bandwidth $\log(d_a)A$ in the expected case

A Distributed Snooping Merge Algorithm Conclusion

- $\log(d_a)A$ is better than d_aA.
- There is still room for improvement. They mention that hardware modification for snooping support could be desirable.
- The Zinnia scene exceeded expected traffic (not shown) because the largest primitive, the background, appeared first in the database, was thus given to the first processor, and thus *every* pixel had to be written to the global frame buffer and written over where necessary by other pixels.
- The higher the depth complexity the better the reduction in traffic.

Parallel Volume Rendering Using Binary-Swap Image Compostion

- Ma, Painter, Hansen, Krogh - 1994
  - Present an algorithm where sub-volumes are raytraced independently to a local frame buffer, and resulting images are composited in parallel.
  - Based on the fact that there are many available high perfomance workstations that could be used for rendering.
    - no need for nodes to communicate while rendering data
      - A Good Question: How does raytracing do lighting effects such as shadows when other primitives are located on other processors?
  - Also go over image- and data-space subdivision.
    - Data-space subdivision is usually applied to distributed computing while image-space subdivision generally involves a shared-memory multiprocessing environment

Divide and Conquer

- Don't mention how raytracing works in a distributed environment where nodes don't have access to all primitives.
- Use a kd-tree for data subdivision.
- Data on the boundaries must be duplicated.

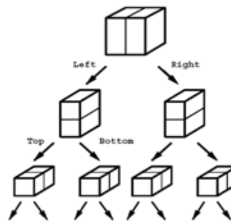


Figure 1: k-Dtree Subdivision of a Data Volume

Parallel Rendering

- Local rendering is done on each processor independently (again, that question…)
  - only rays that fall within the subvolume are cast
  - use an identical view position
- Note that you must user consistent sampling locations so that “we can construct the original volume.”
  - you could see volume boundaries
    - get two primitives on boundaries

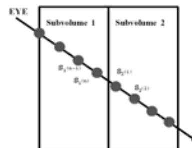


Figure 2: Correct Ray Sampling

Image Composition

- Need to composite images in back to front order
- Porter and Duff - **over**

- In the naïve process many processors become idle
  - With Binary Swap every processor participates in every step of the compositing process
- Each processor swaps its image with another, getting two coherent halves.
- Division is then done along the opposite axis.
- End up with 1/n of the image

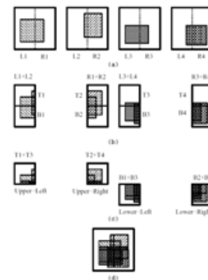


Image Composition cont.

- The final image is constructed by simply tiling the images from each processor
- “Sparsity” can be exploited.
  - don’t composite parts of the image that aren’t there
    - processors keep a bounding box of their non-blank sub-image area
    - is a bounding box the best way here?
      - Could be very sparse but if two pixels far apart would create a big bounding box
      - my suggestion - would it be possible to keep an ordered list of pixels and do a cross product on the two sets?
- Number of processors had to be a power of two
  - algorithm been expanded?

Communication Costs

- Mathematical Massaging:
  - after rendering each processor has approximately $pn^{-2/3}$ pixels so the total number of pixels is $pn^{1/3}$
    - don’t say where they come up with the exponent
  - some number of pixels will be deleted in compositing
  - massaging …. pixels transmitted is $\le 2.43n^{1/3}p$
    - the way they come up with it doesn’t exactly make sense to me (anyone else?)

Comparison to Other algorithms

- Direct Send
  - assign a subset of the image pixels to each processor
  - pixels transmitted is $n^{1/3}p(1-1/n)$
  - may require n(n-1) messages to be transmitted
    - every pixel a processor renders needs to be sent to another processor, and this happens for each processor - one processpor sends p(n-1) pixels, n processors, get pn(n-1)
- Binary Swap
  - each processor sends log(n) messages, so the total number of messages sent is nlog(n)
    - can use nearest neighbor paths when they exist, when number of pixels transmitted is largest early in the compositing

Comparison to Other algorithms

- Projection method
  - rays propagate through a 3D grid decomposition of the volume data
  - rays propagate back to front through the volumes from one processor holding a sub-volume to the processor holding the neighboring sub-volume
  - transmits $O(n^{1/3}p)$ pixels like the other two methods
  - Must move through $n^{1/3}$ processor nodes so the message latency grows by $O(n^{1/3})$ as opposed to (n-1) for direct send and log(n) for binary swap

Implementation

- Used the CM-5, a 1024 node supercomputer
  - each node had 32MB of RAM and 64-bit wide vector units (unused).
- Also experimented with a shared network of workstations.

- Didn't implement other algorithms, have no basis for comparison of timed results. Simulations do show that with more processors rendering time goes down while the percentage of time spent on distribution increases.

function	32	64	128	256	512
dist	89.87	93.516	83.185	94.326	49.157
rend	48.2005	24.4303	12.697	6.3434	3.1878
comp	0.6309	0.5579	0.4091	0.3736	0.3213
comm	0.0843	0.0231	0.0181	0.0138	0.0097
send	0.9918	0.965	0.9645	1.0151	0.9849

Table 1: CM-5 Time (in seconds) Breakdown, Vorticity Data Set, 512×512 image size

Conclusions

- They say that binary swap can be faster.

- It *does* keep the all the processors busy during compositing.

- Networked workstations did not have linear speedup because they were on a shared network, data is somewhat inconsistent. A LAN doesn't scale either, more nodes doesn't mean more bandwidth.

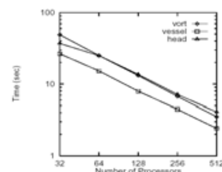


Figure 8: CM5 Run Times by Data Set, 512×512 image size

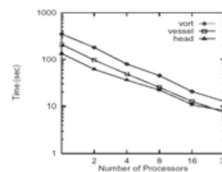


Figure 10: PVM Run Times by Data Set, 512×512 image size

Comparison of Parallel Compositing Techniques on Shared Memory Architecures

- Reinhard and Hansen - 2000
  - Mention the Parallel Pipeline
  - Review Binary Swap
  - Review Direct Send

Parallel Pipeline

- The basic idea is that the images and z-buffers to be composited are divided into P sub-images, one for each processor.
- The sub-images flow around a “ring” of processors,
- Each processor composites its sub-image k with the same sub-image that belongs to processor p_k There are P-1 steps

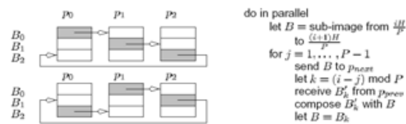


Fig. 1. Parallel pipeline compositing on distributed memory architectures (after [6]). Left: graphical representation. Right: pseudo-code. The variables are coded as follows: P is the total number of processors, H is total image height, B and Z are sub-images and p_i is processor number i.

Parallel Pipeline

- There are P-1 steps
  - each processor composites N/P pixels during each iteration
  - time complexity is O(N)
    - N is number of pixels

Binary Swap

- We already know how this works.

- The algorithm iterates $\log_2(p)$ times and each time each processor composites $N/(2^i)$ pixels
  - N is number of pixels
  - i is iteration number
  - Total number of composites is N(1-1/p)

Binary Swap

- Shared memory compositor (Direct Send)
  - send all partial results for a given pixel to the processor that manages the pixel
  - pixels are composited as they come in
  - result is in a single block of memory, allowing direct display
  - also O(N) for number of pixels
    - P-1 steps, compositing N/P pixels at each step

Results

- Everything takes about the same time
  - differences are due to overhead and cache efficiency

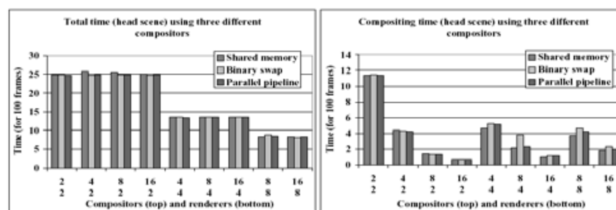


Fig. 6. Total time and compositing time for the head scene, using a resolution of 512^2 pixels.

Conclusions

- As long as the number of compositors and renderers is the same, there will be an optimum number of each
  - adding one or the other will create a bottleneck
- To reduce compositing time, must increase the number of compositors relative to the number of renderers.
  - For a given number of renderers, doubling the compositors should have the compositing time
  - We know every algorithm is linear, so this is no surprise

Conclusions

- The big conclusion
  - For a 1024^2 image, it takes 4 times the compositors it would for 512^2
  - good work boys