Parallel Rendering In the GPU Era

Orion Sky Lawlor
olawlor@acm.org
U. Alaska Fairbanks
2009-04-16

http://lawlor.cs.uaf.edu/

Importance of Computer Graphics

- "The purpose of computing is insight, not numbers!" R. Hamming

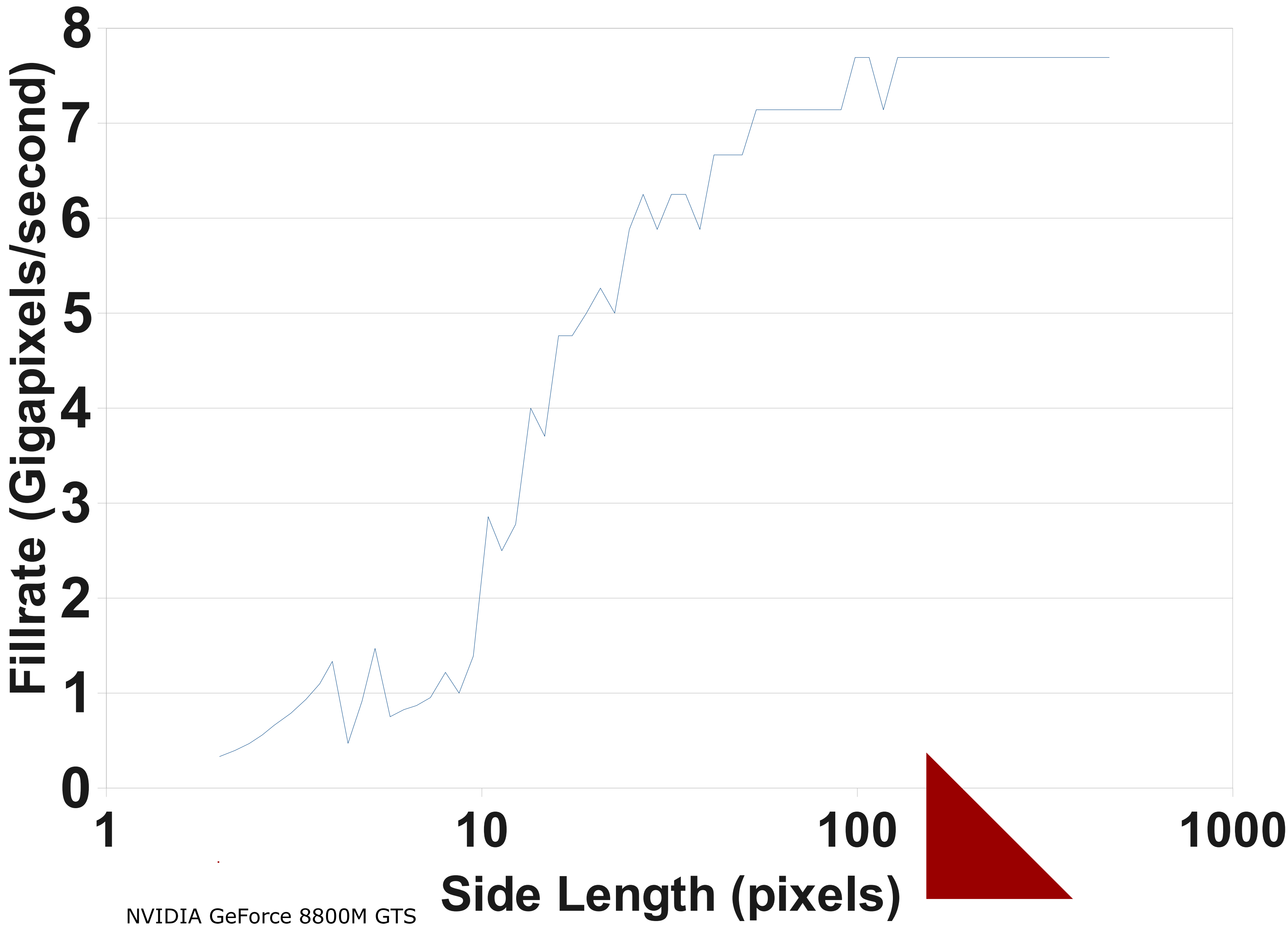
- Vision is a key tool for analyzing and understanding the world
- Your eyes are your brain's highest bandwidth input device
  - Vision: >300MB/s
    - 1600x1200 24-bit 60Hz
  - Sound: <1 MB/s
    - 44KHz 24-bit 5.1 Surround sound
  - Touch: <1 KB/s (?)
  - Smell/taste: <10 per second
- Plus, pictures look really cool…

Prior work:
GPUs, NetFEM, impostors

GPU Rendering Drawbacks

- Graphics cards <u>are</u> fast
  - But not at rendering lots of tiny geometry:
    - 1M primitives/frame OK
    - 1G pixels/frame OK
    - 1G primitives/frame not OK
- Problems with billions of primitives do <u>not</u> utilize current graphics hardware well
- Graphics cards only have a few gigabytes of RAM (vs. parallel machine, with terabytes of RAM)

Graphics Card: Usable Fill Rate



NVIDIA GeForce 8800M GTS

Parallel Rendering Advantages

- Multiple processors can render geometry simultaneously

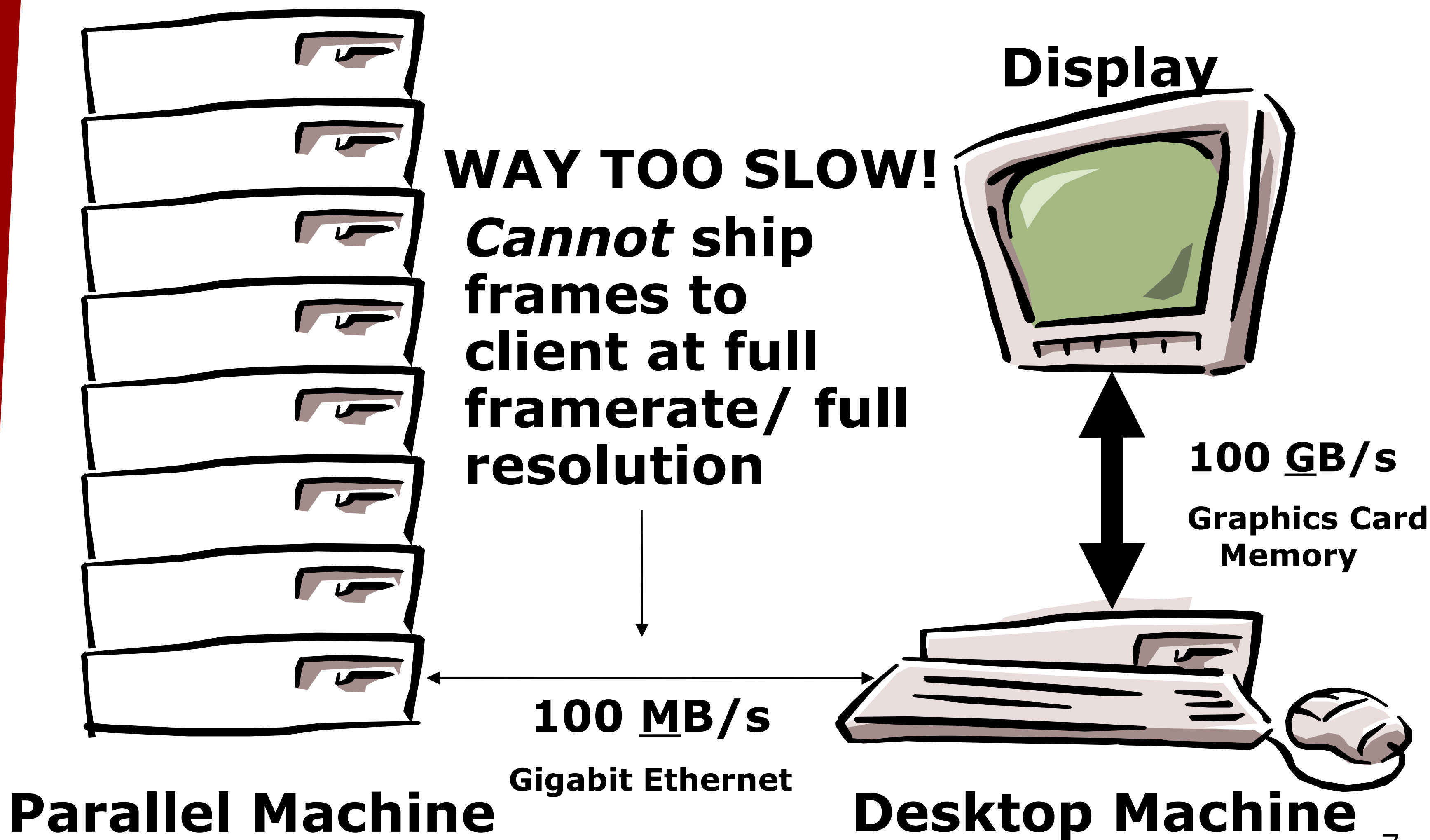
Processors	4	8	16	24	32	48
MParticles/second	7.14	15.71	32.71	49.18	65.49	81.68

48 nodes of Hal cluster: 2-way 550MHz Pentium III nodes connected with fast ethernet

- Achieved rendering speedup for large particle dataset
- Can store huge datasets in memory
- BUT: No display on parallel machine!
- Ignores cost of shipping images to client

Parallel Rendering Disadvantage

- Link to client is too slow!

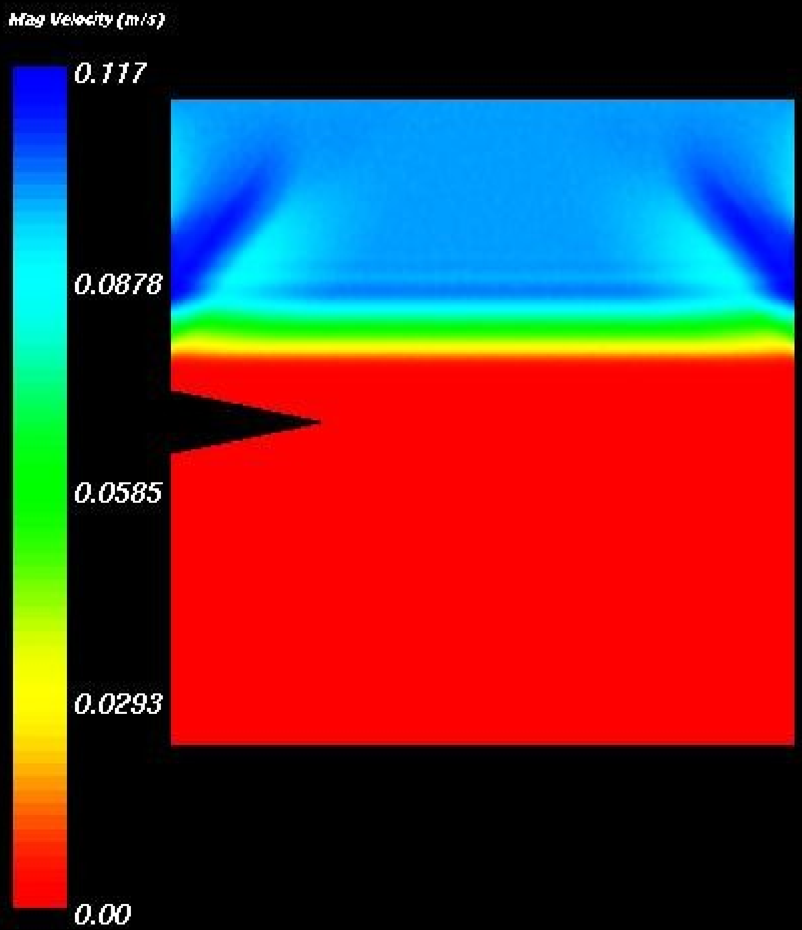


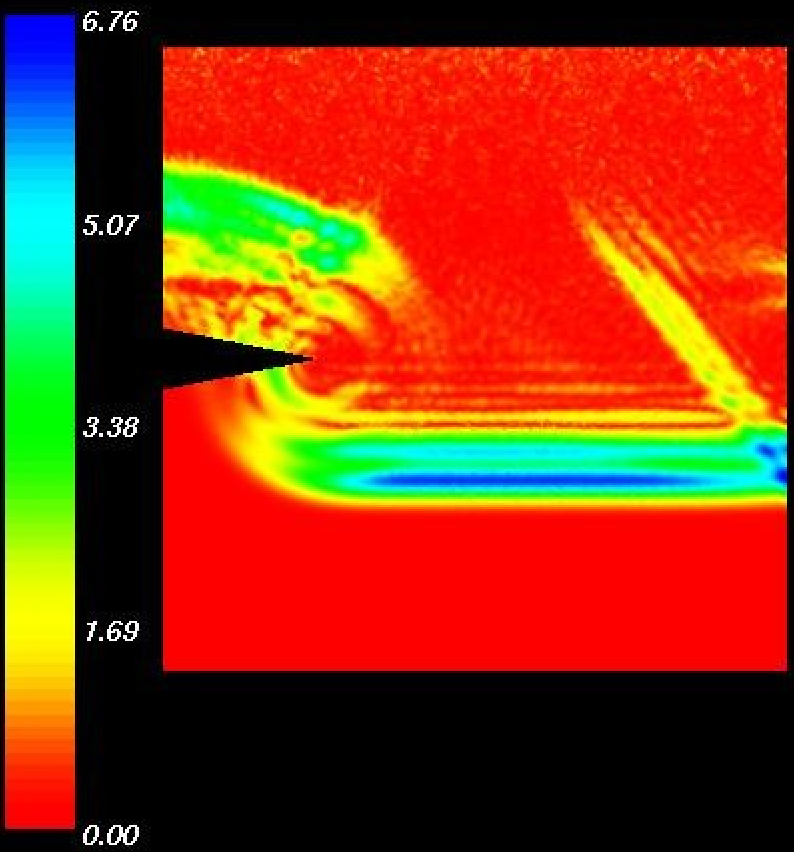
Basic model: NetFEM

- Serial OpenGL Client
- Parallel FEM Framework Server
- Client connects
- Server sends client the current FEM mesh (nodes and elements)
  - Includes all attributes
  - Client can display, rotate, examine
  - Not just for postmortem!
    - Making movies on the fly
    - Dumping simulation output
    - Monitoring running simulation

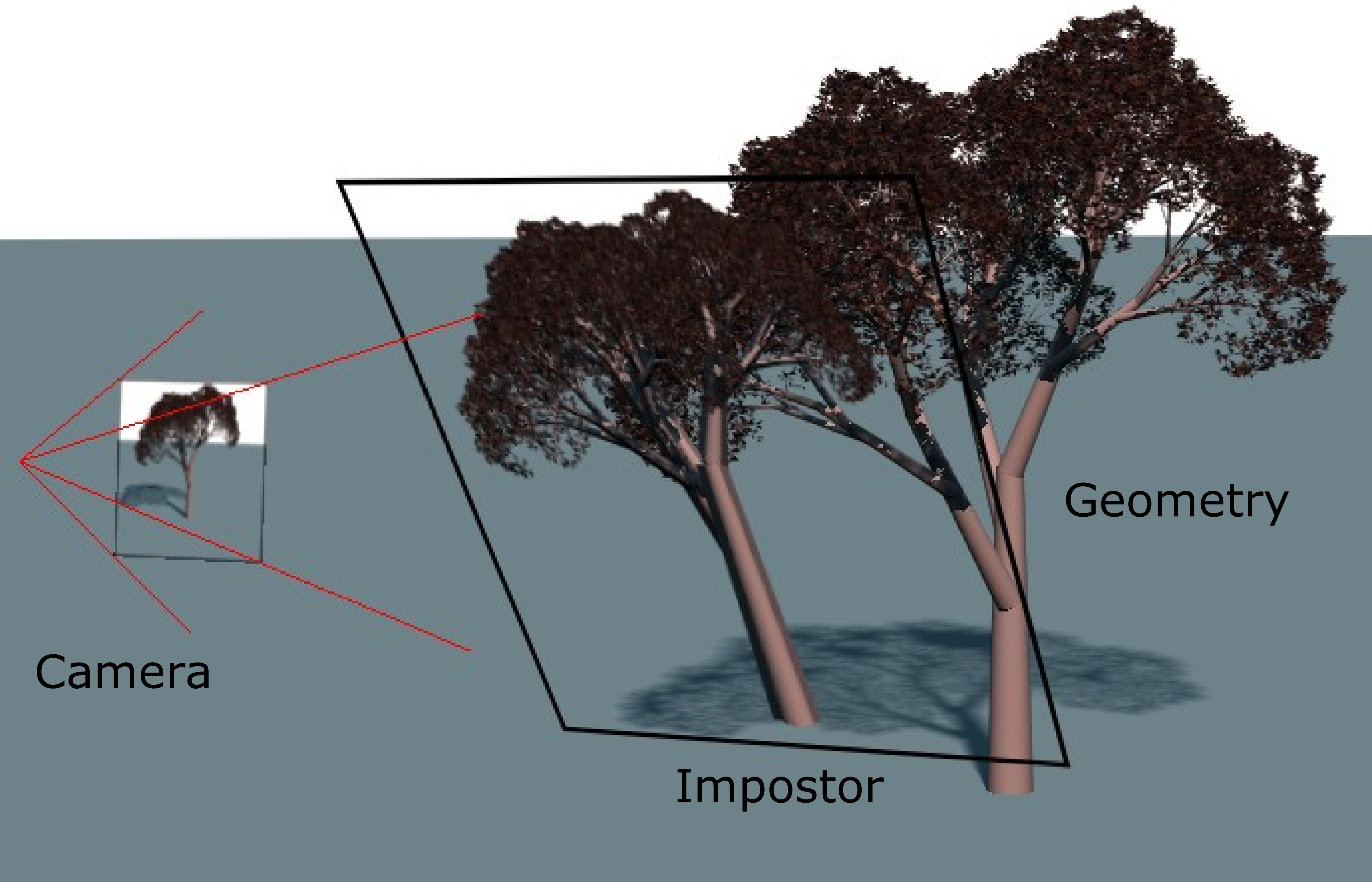
NetFEM: visualization tool

- Connect to running parallel machine
- See, e.g., wave dispersion off a crack





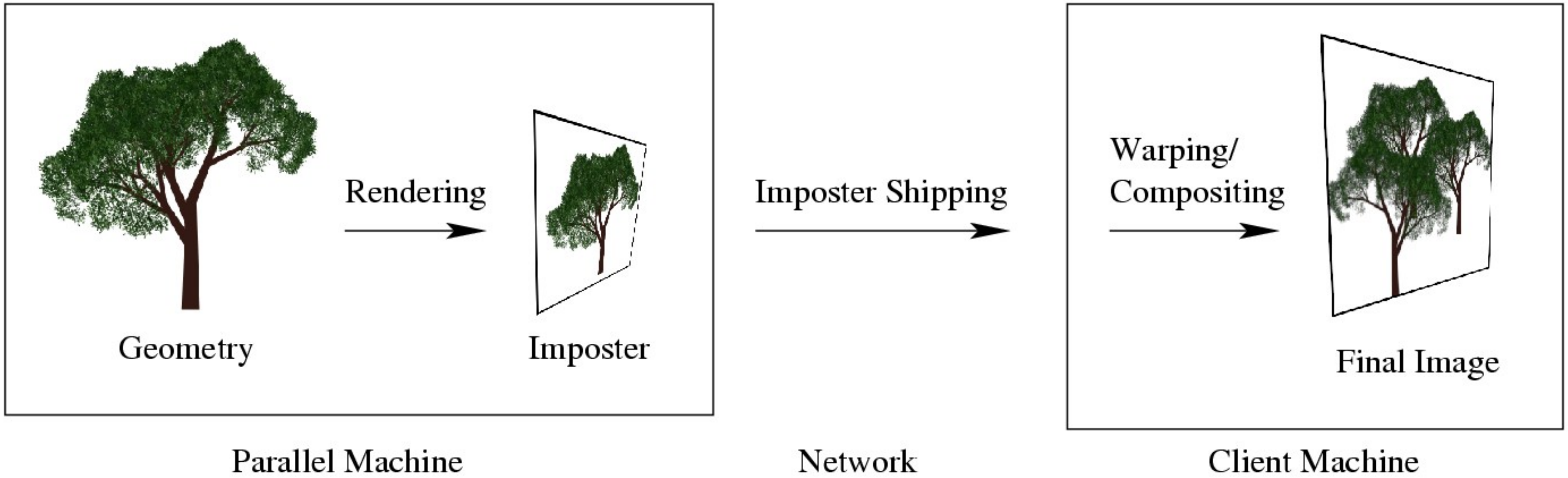
Impostors : Basic Idea



Parallel Impostors Technique

- Key observation: impostor images don't depend on one another
- So render impostors in parallel!
  - Uses the speed and memory of the parallel machine
    - Fine grained-- lots of potential parallelism
  - Geometry is partitioned by impostors
    - No "shared model" assumption
- Reassemble world on serial client
  - Uses rendering bandwidth of client graphics card
  - Impostor reuse cuts required network bandwidth to client
    - Only update images when necessary
  - Impostors provide latency tolerance

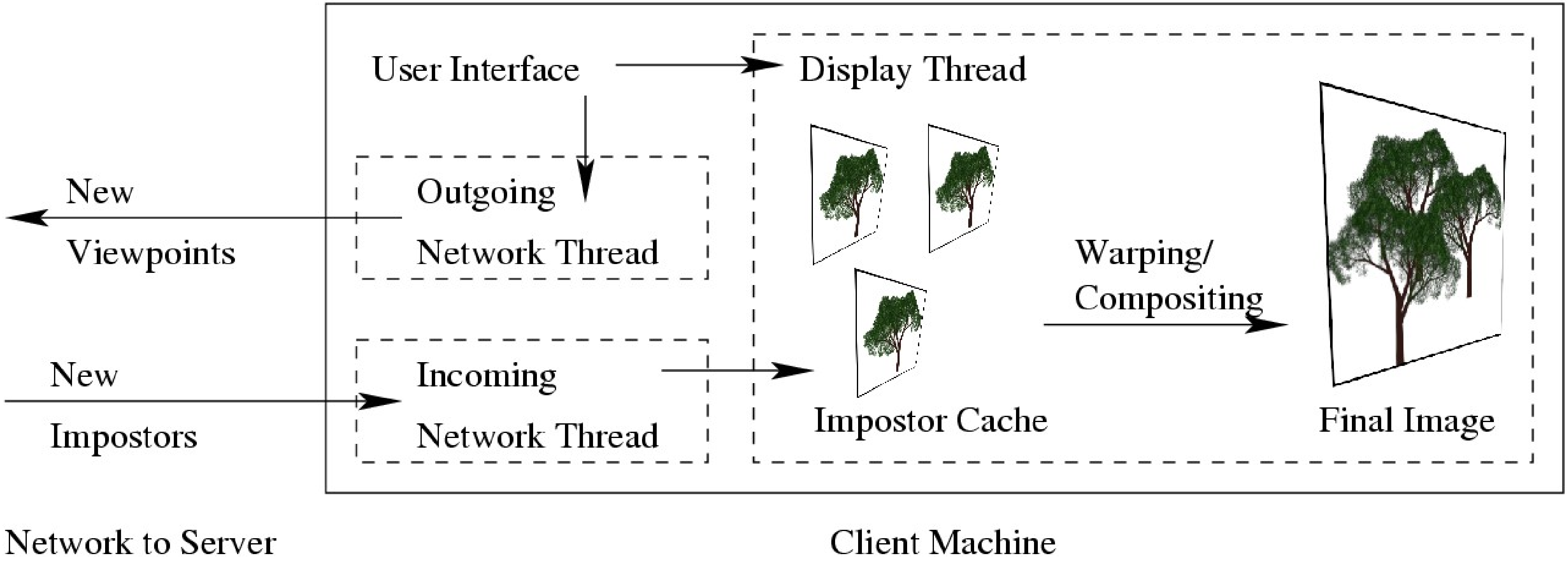
Client/Server Architecture



- **Parallel machine can be anywhere on network**
  - **Keeps the problem geometry**
  - **Renders and ships new impostors as needed**
- **Impostors shipped using TCP/IP sockets**
  - **CCS & PUP protocol [Jyothi and Lawlor 04]**
  - **Works over NAT/firewalled networks**
- **Client sits on user's desk**
  - **Sends server new viewpoints**
  - **Receives and displays new impostors**

Client Architecture

- Latency tolerance: client *never* waits for server
  - Displays existing impostors at fixed framerate
    - Even if they're out of date
  - Prefers spatial error (due to out of date impostor) to temporal error (due to dropped frames)
- Implementation uses OpenGL for display
  - Two separate kernel threads for network handling



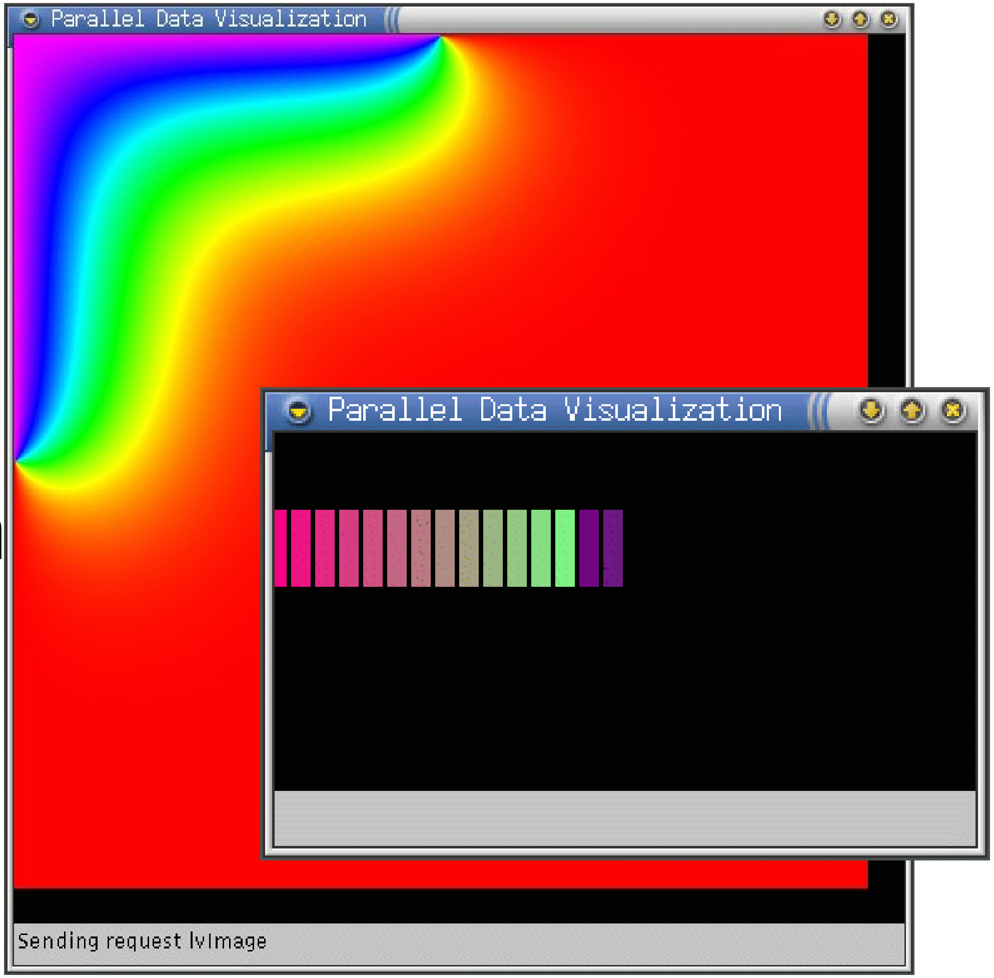
New work:
liveViz pixel transport

Basic model: LiveViz

- **Serial 2D Client**
- **Parallel Charm++ Server**
- **Client connects**
- **Server sends client the current 2D image pixels (just pixels)**
  - **Can be from a 3D viewpoint (liveViz3D mode)**
  - **Can be color (RGB) or grayscale**
  - **Recently extended to support JPEG compressed network transport**
    - **Big win on slow networks!**

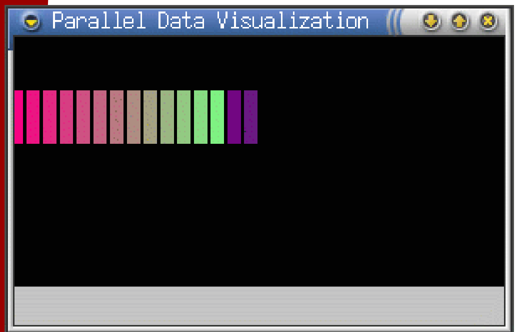
LiveViz – What is it?

- Charm++ library
- Visualization tool
- Inspect your program's current state
- Java client runs on any machine
- You code the image generation
- 2D and 3D modes



LiveViz Request Model

Client GUI



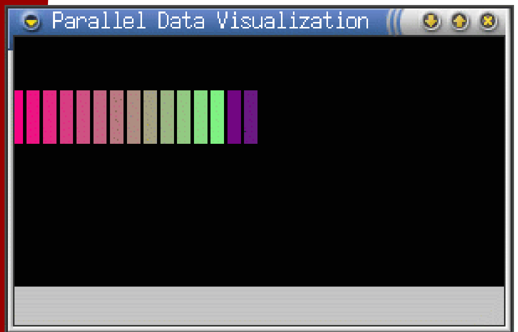
LiveViz Server Library

LiveViz Application

- Client sends request
- Server code broadcasts request to application
- Application array element render image pieces
- Server code assembles full 2D image
- Server sends 2D image back to client
- Client displays image

LiveViz Request Model

Client GUI



LiveViz Server Library

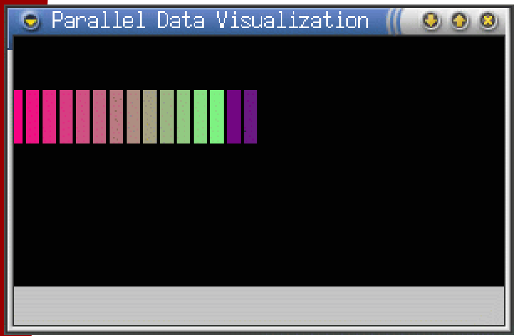
LiveViz Application

- Client sends request
- Server code broadcasts request to application
- Application array element render image pieces
- Server code assembles full 2D image
- Server sends 2D image back to client
- Client displays image

Bottleneck!

LiveViz Compressed requests

Client GUI



LiveViz Server Library

LiveViz Application

- Client sends request
- Server code broadcasts request to application
- Application array element render image pieces
- Server code assembles full 2D image
- Server compresses 2D image to a JPEG
- Server sends JPEG to client
- Client decompresses and displays image

LiveViz Compressed requests

Window Size	No Compression	Compression
256x256	333 fps	25 fps
512x512	166 fps	24 fps
1024x1024	50 fps	15 fps
2048x2048	13 fps	4 fps

- **On a gigabit network, JPEG compression is CPU-bound, and just slows us down!**
- **Compression hence <u>optional</u>**

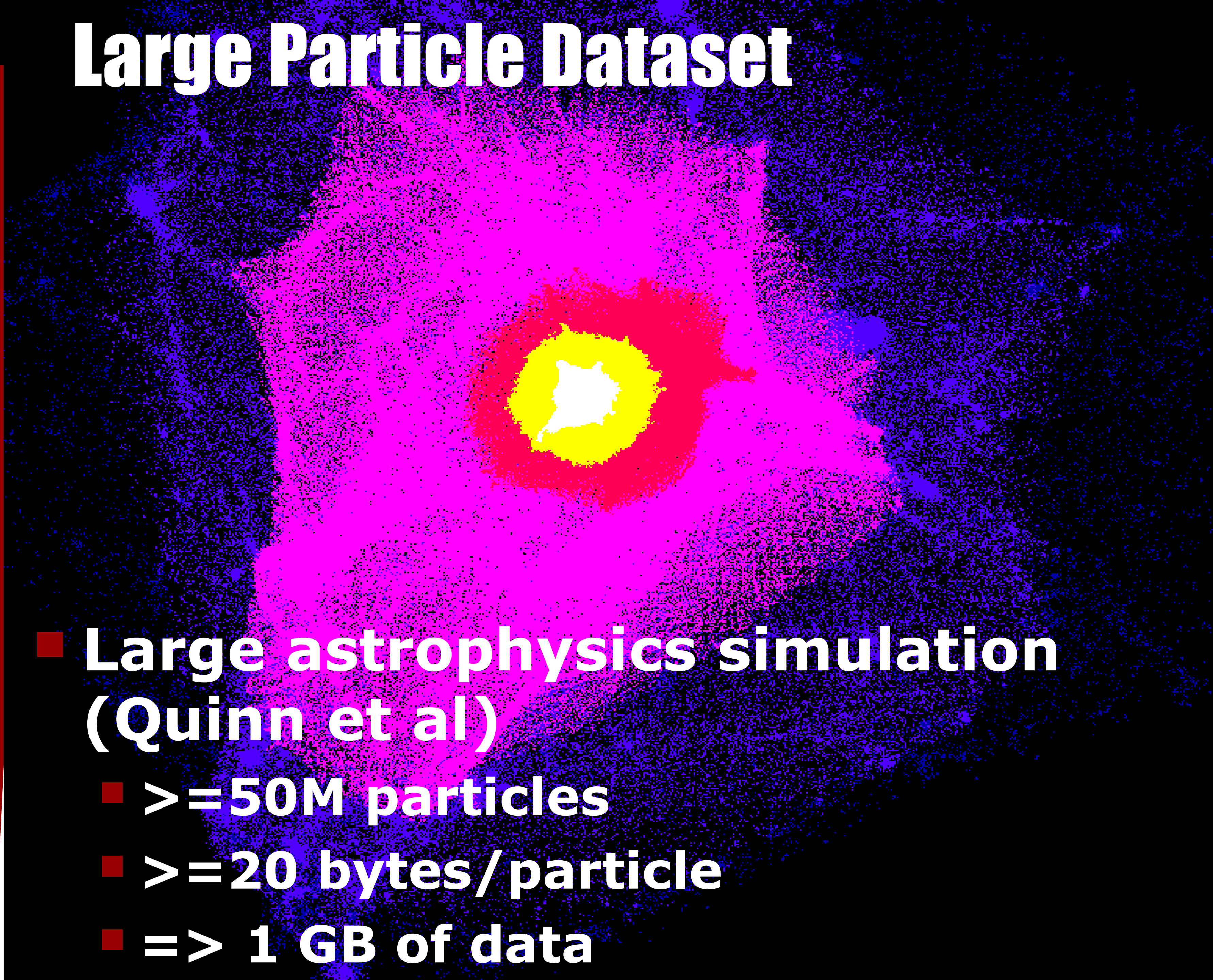
LiveViz Compressed requests

Window Size	No Compression	Compression
256x256	6 fps	22 fps
512x512	2 fps	15 fps
1024x1024	< 1 fps	13 fps
2048x2048	<< 1 fps	4 fps

- On a slow 2MB/s wireless or WAN network, uncompressed liveViz is network bound
- Here, JPEG data transport is a big win!

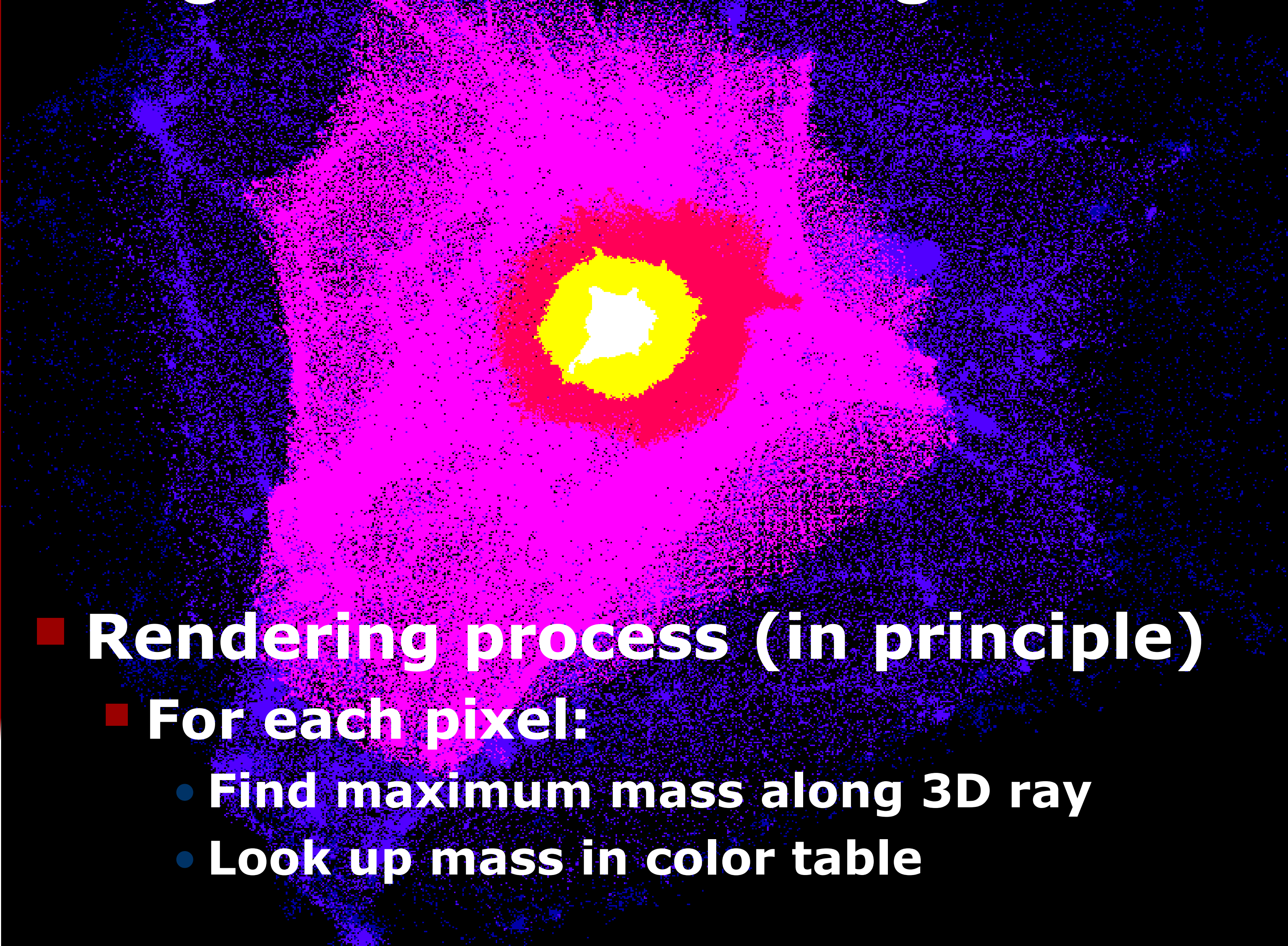
New work:
Cosmology Rendering

Large Particle Dataset



- Large astrophysics simulation (Quinn et al)
  - >=50M particles
  - >=20 bytes/particle
  - => 1 GB of data

Large Particle Rendering

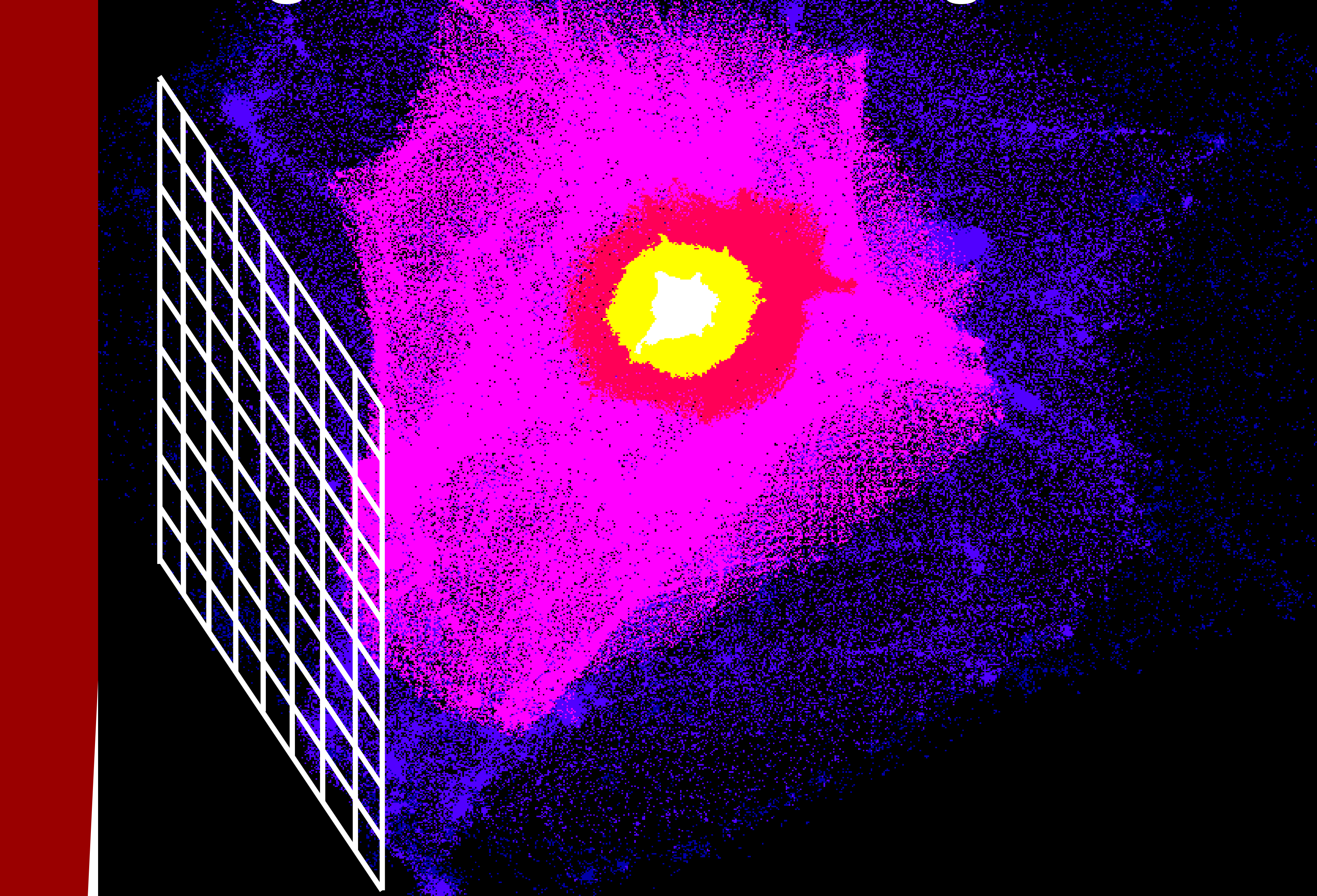


- Rendering process (in principle)
  - For each pixel:
    - Find maximum mass along 3D ray
    - Look up mass in color table

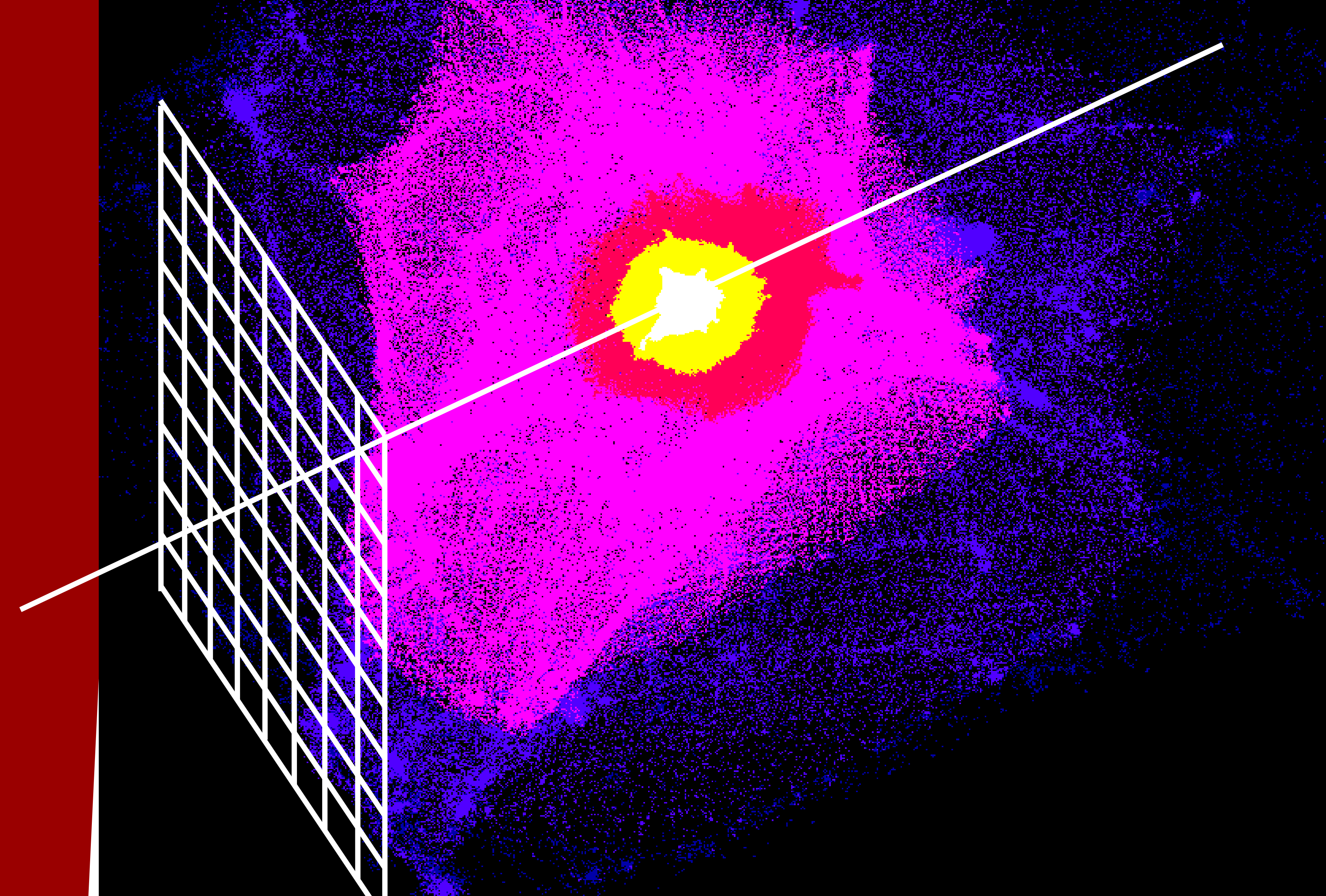
Large Particle Rendering

- Rendering process (in practice)
  - For each particle:
    - Project 3D particle onto 2D screen
    - Keep maximum mass at each pixel
    - Ship image to client
    - Apply color table to 2D image at client

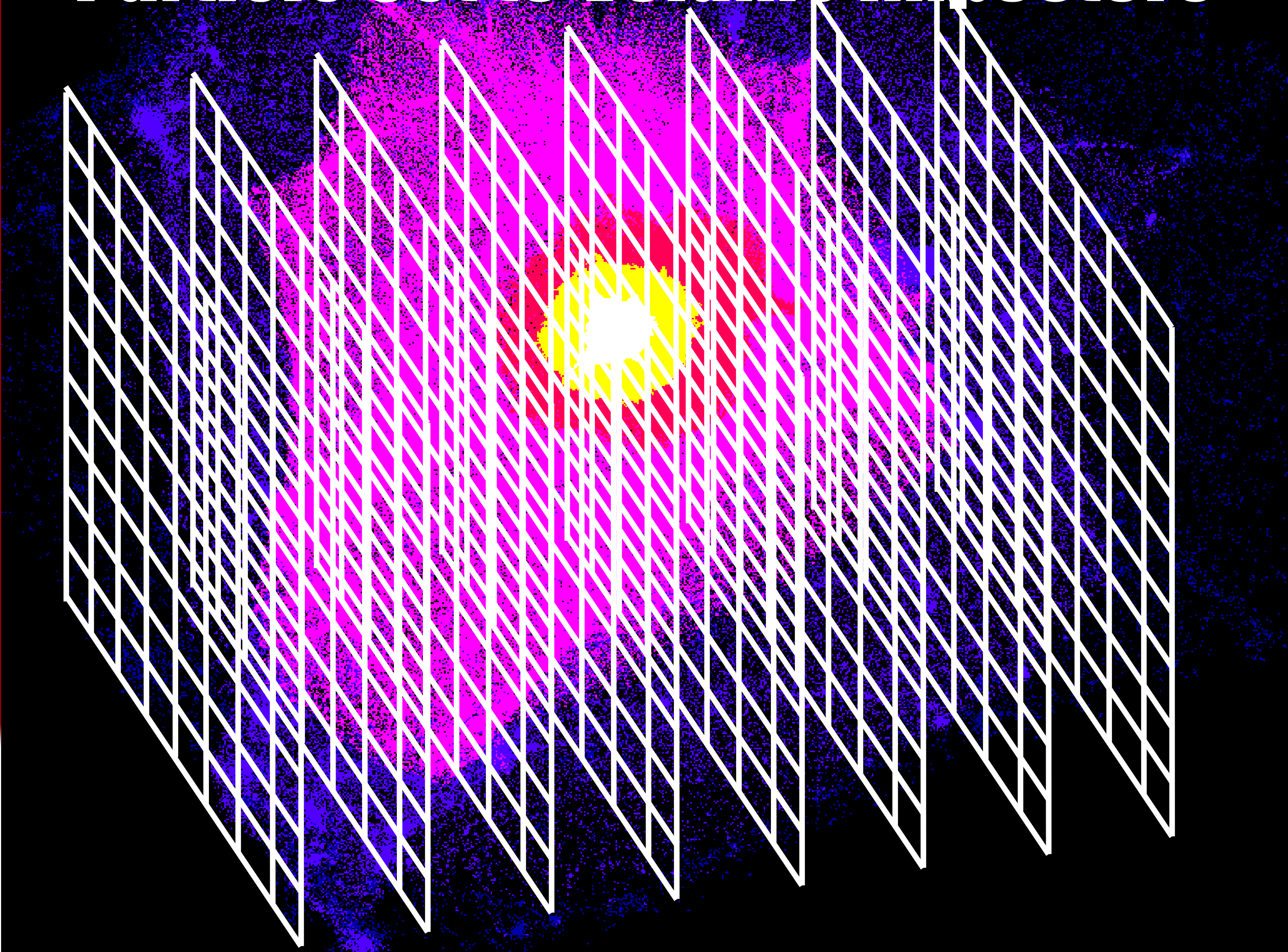
Large Particle Rendering (2D)



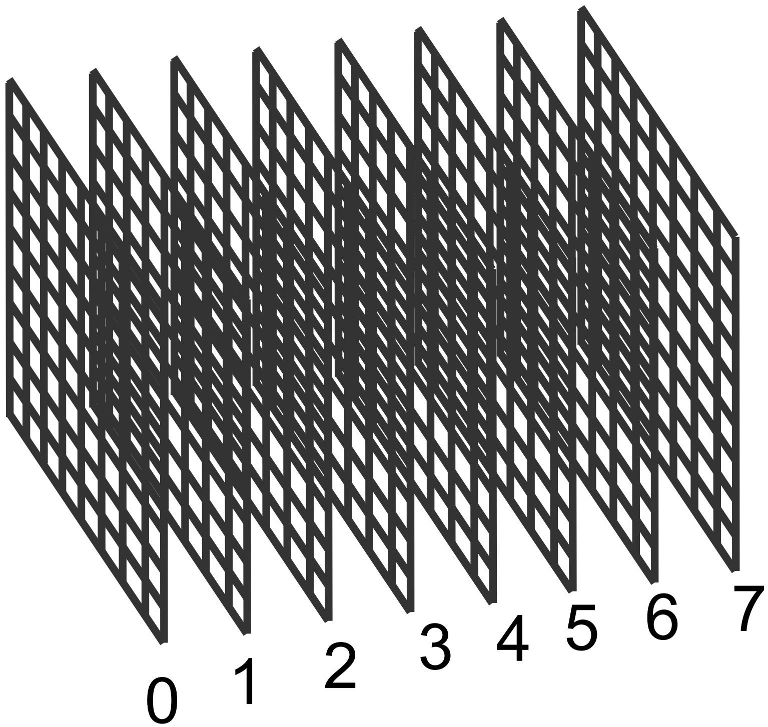
Large Particle Rendering (2D)



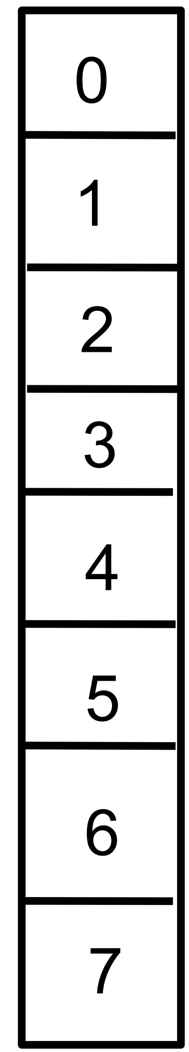
Particle Set to Volume Impostors



Shipping Volume Impostors



Slices of 3D Volume



Stack of 2D Slices

Shipping Volume Impostors

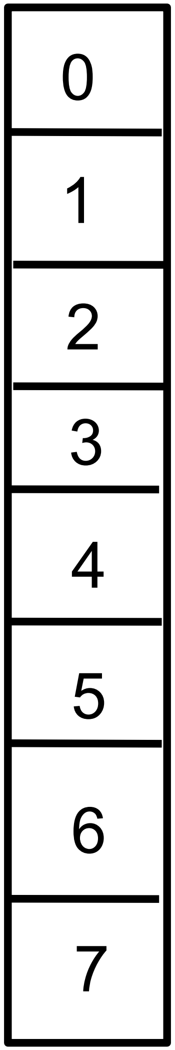
- Hey, that's just a 2D image!
- So we can use liveViz:

  Render slices in parallel

  Assemble slices across processors

  (Optionally) JPEG compress image

  Ship across network to (new) client



Stack of 2D Slices

Volume Impostors Technique

- 2D impostors are flat, and can't rotate
- 3D voxel dataset can be rendered from any viewpoint on the client
- Practical problem:
  - Render voxels into a 2D image on the client by drawing slices with OpenGL
  - Store maximum across all slices: glBlendEquation(GL_MAX);
  - To look up (rendered) maximum in color table, render slices to texture and run a programmable shader

Volume Impostors: GLSL Code

- **GLSL code to look up the rendered color in our color table texture:**

```
varying vec2 texcoords;
uniform sampler2D rendered, color_table;
void main()
{
  vec4 rend=texture2D(rendered,texcoords);
  gl_FragColor = texture2D(color_table,
        vec2(rend.r+0.5/255,0));
}
```

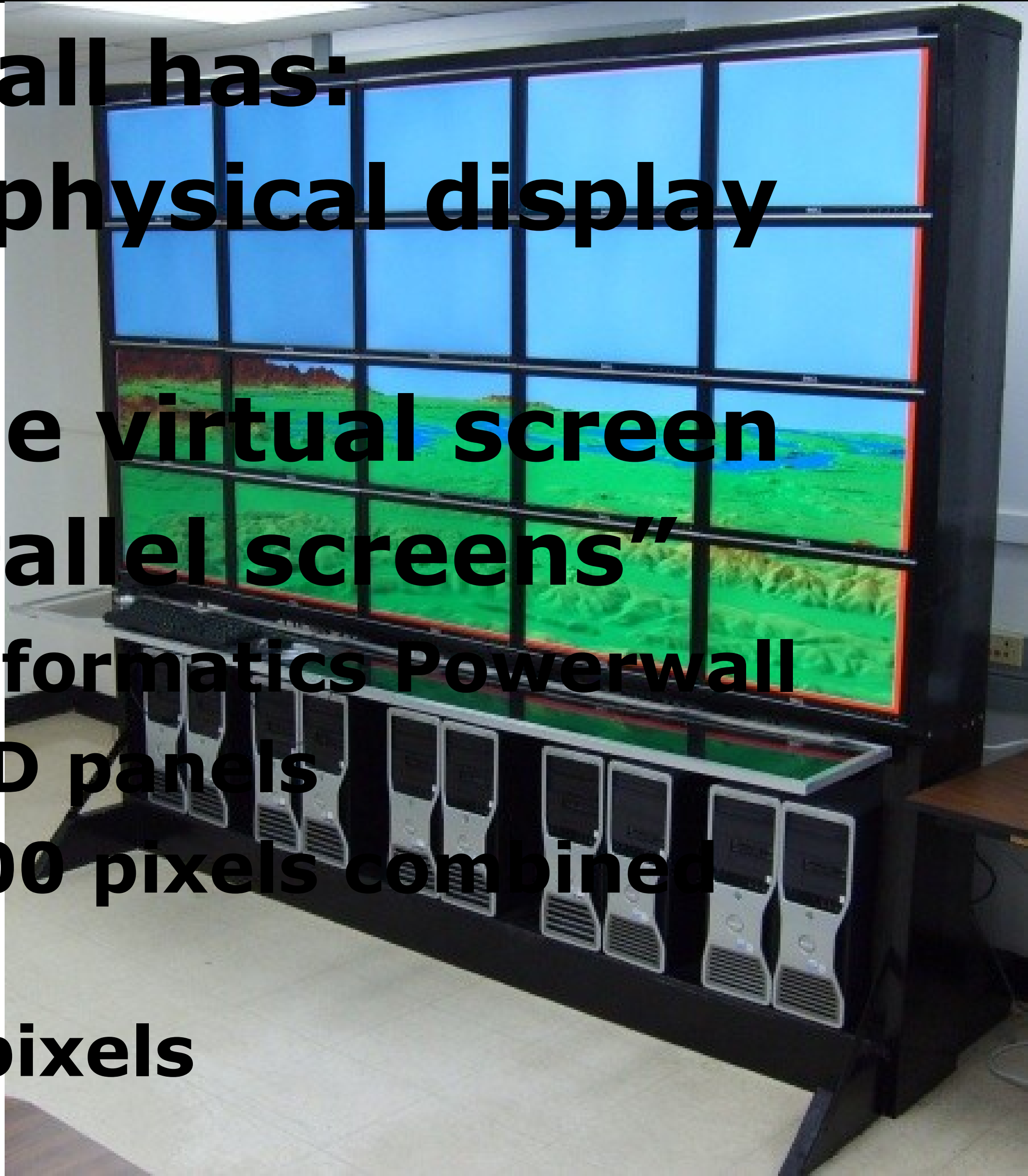
New Work:
MPIglut

MPIglut: Motivation

- All modern computing is <u>parallel</u>
  - Multi-Core CPUs, Clusters
    - Athlon 64 X<u>2</u>, Intel Core2 <u>Duo</u>
  - Multiple Multi-Unit GPUs
    - nVidia SLI, ATI CrossFire
  - Multiple Displays, Disks, ...
- But languages and many existing applications are <u>sequential</u>
  - Software problem: run existing serial code on a parallel machine
  - Related: easily write parallel code

What is a "Powerwall"?

- A powerwall has:
  - Several physical display devices
  - One large virtual screen
  - I.E. "parallel screens"
- UAF CS/Bioinformatics Powerwall
  - Twenty LCD panels
  - 9000 x 4500 pixels combined resolution
  - 35+ Megapixels




DELL
Backspace
Shift


DELL

MPIglut: The basic idea

- **Users compile their OpenGL/glut application using MPIglut, and it "just works" on the powerwall**
- **MPIglut's version of glutInit runs a separate copy of the application for each powerwall screen**
- **MPIglut intercepts glutInit, glViewport, and broadcasts user events over the network**
- **MPIglut's glViewport shifts to render only the local screen**

MPIglut uses glut sequential code

- GL Utilities Toolkit
  - Portable window, event, and GUI functionality for OpenGL apps
  - De facto standard for small apps
  - Several implementations: Mark Kilgard original, FreeGLUT, ...
  - Totally sequential library, until now!
- MPIglut intercepts several calls
  - But many calls still unmodified
  - We run on a patched freeglut 2.4
    - Minor modification to window creation

Parallel Rendering Taxonomy

- Molnar's influential 1994 paper
  - Sort-first: send geometry across network before rasterization (GLX/DMX, Chromium)
  - Sort-middle: send scanlines across network during rasterization
  - Sort-last: send rendered pixels across the network after rendering (Charm++ liveViz, IBM's Scalable Graphics Engine, ATI CrossFire)

Parallel Rendering Taxonomy

- **Expanded taxonomy:**
  - **Send-event (MPIglut, VR Juggler)**
    - **Send only user events (mouse clicks, keypresses). Just kilobytes/sec!**
  - **Send-database**
    - **Send application-level primitives, like terrain model. Can cache/replicate data!**
  - **Send-geometry (Molnar sort-first)**
  - **Send-scanlines (Molnar sort-middle)**
  - **Send-pixels (Molnar sort-last)**

MPIglut Code & Runtime Changes

MPIglut Conversion: Original Code

```
#include <GL/glut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void reshape(int x_size,int y_size) {
   glViewport(0,0,x_size,y_size);
   glLoadIdentity();
   gluLookAt(...);
}
...
int main(int argc,char *argv[]) {
   glutInit(&argc,argv);
   glutCreateWindow("Ello!");
   glutMouseFunc(...);
   ...
}
```

MPIglut: Required Code Changes

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void
   glV
   gl
   gluLookAt(...);
}
...
int main(int argc,char *argv[]) {
   glutInit(&argc,argv);
   glutCreateWindow("Ello!");
   glutMouseFunc(...);
   ...
}
```

This is the only source change. Or, you can just copy mpiglut.h over your old glut.h header!

MPIglut Runtime Changes: Init

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void reshape(int x_size,int y_size) {
   glViewport(0,0,x_size,y_size);
   glLoac
   gluLo
}
...
int main(int argc,char *argv[]) {
   glutInit(&argc,argv);
   glutCreateWindow("Ello!");
   glutMouseFunc(...);
   ...
}
```

MPIglut starts a <u>separate</u> copy of the program (a "backend") to drive each powerwall screen

MPIglut Runtime Changes: Events

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void reshape(int x_size,int y_size) {
   glViewport(0,0,x_size,y_size);
   glLo
   gluL
}
...
int ma
   glut
   glutCreateWindow("Ello!");
   glutMouseFunc(...);
   ...
}
```

Mouse and other user input events are collected and sent across the network.

Each backend gets <u>identical</u> user events (collective delivery)

MPIglut Runtime Changes: Sync

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void reshape(int x_size,int y_size) {
   glViewport(0,0,x_size,y_size);
   glLo
   gluL
}
...
int main(int argc,char *argv[]) {
   glutInit(&argc,argv);
   glutCreateWindow("Ello!");
   glutMouseFunc(...);
   ...
}
```

Frame display is (optionally) synchronized across the cluster

MPIglut Runtime Changes: Coords

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void reshape(int x_size,int y_size) {
   glViewport(0,0,x_size,y_size);
   glLoadIdentity();
   gluLookAt(...);
}
...
int ma
   glu
   glu
   glutMouseFunc(...);
   ...
}
```

User code works only in <u>global</u> coordinates, but MPIglut adjusts OpenGL's projection matrix to render only the <u>local</u> screen

MPIglut Runtime Non-Changes

```
#include <GL/mpiglut.h>
void display(void) {
   glBegin(GL_TRIANGLES); ... glEnd();
   glutSwapBuffers();
}
void re
   glVi
   glLo
   gluL
}
...
int ma
   glut
   glutCreateWindow("Ello!");
   glutMouseFunc(...);
   ...
}
```

MPIglut does NOT intercept or interfere with rendering calls, so programmable shaders, vertex buffer objects, framebuffer objects, etc all run at full performance

MPIglut Assumptions/Limitations

- Each backend app must be able to render its part of its screen
  - Does not automatically imply a replicated database, if application uses matrix-based view culling
- Backend GUI events (redraws, window changes) are collective
  - All backends must stay in synch
  - Automatic for applications that are deterministic function of <u>events</u>
    - Non-synchronized: files, network, time

MPIglut: Bottom Line

- Tiny source code change
- Parallelism hidden inside MPIglut
  - Application still "feels" sequential
- Fairly major runtime changes
  - Serial code now runs in parallel (!)
  - Multiple synchronized backends running in parallel
  - User input events go across network
  - OpenGL rendering coordinate system adjusted per-backend
  - But rendering calls are left alone

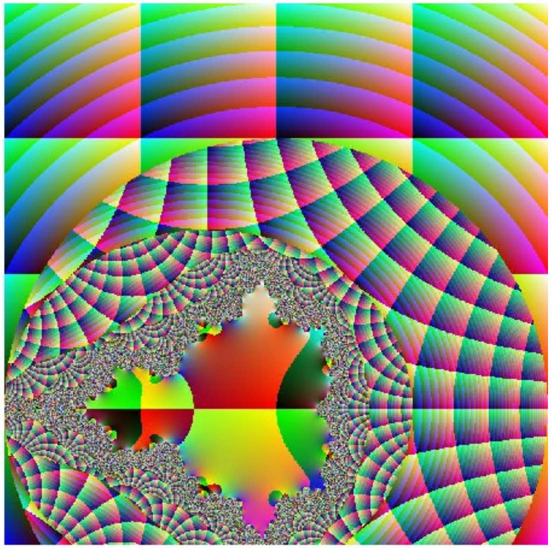
MPIglut Application Performance

Performance Testing

- MPIglut programs perform about the same on 20 screens as they do on 1 screen
- We compared performance against two other packages for running unmodified OpenGL apps:
  - DMX: OpenGL GLX protocol interception and replication (MPIglut gets screen sizes via DMX)
  - Chromium: libgl OpenGL rendering call interception and routing

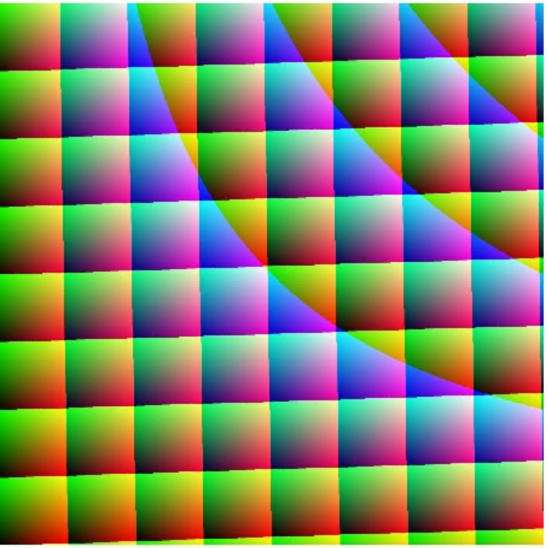
Benchmark Applications

basic

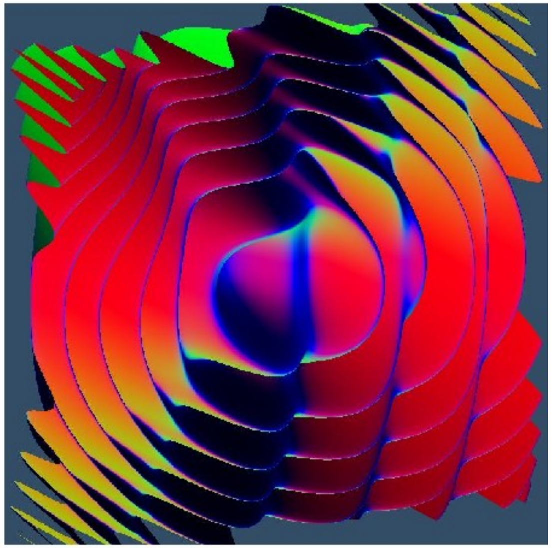


mandel

soar



tex, tex_obj



vtx, vtx_obj

UAF CS Bioinformatics Powerwall

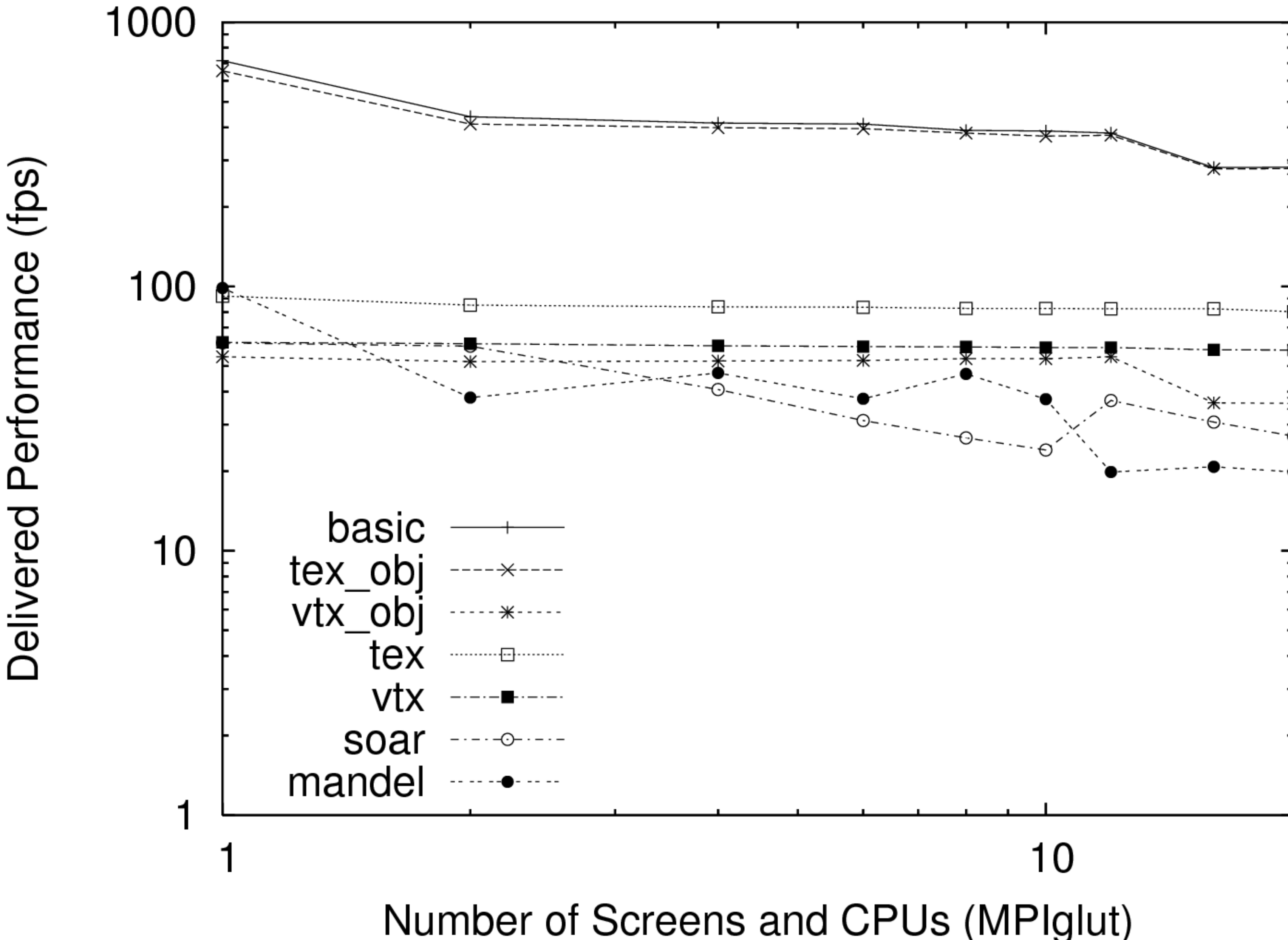
Switched Gigabit Ethernet Interconnect

10 Dual-Core 2GB Linux Machines:

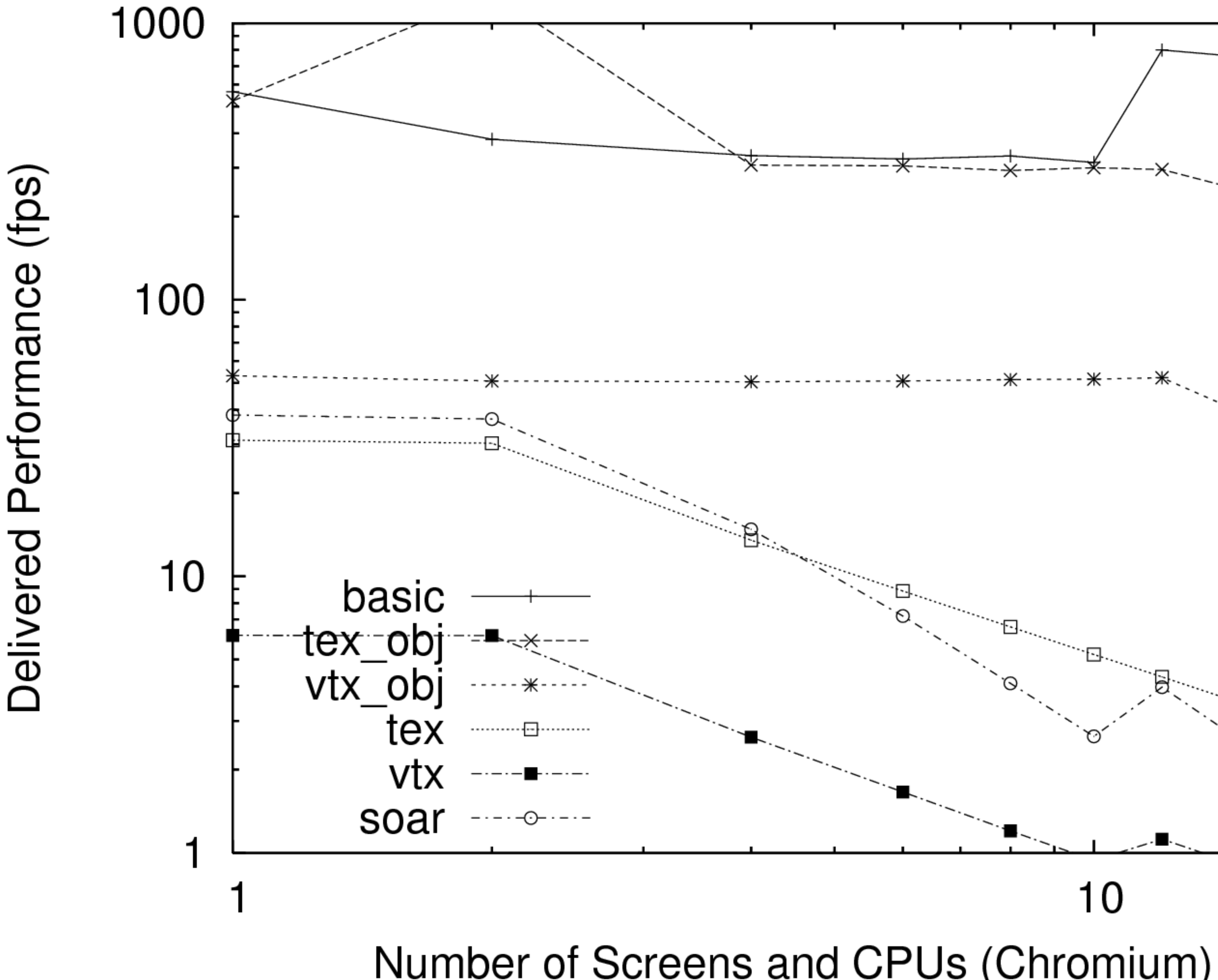
7 nVidia QuadroFX 3450

3 nVidia QuadroFX 1400

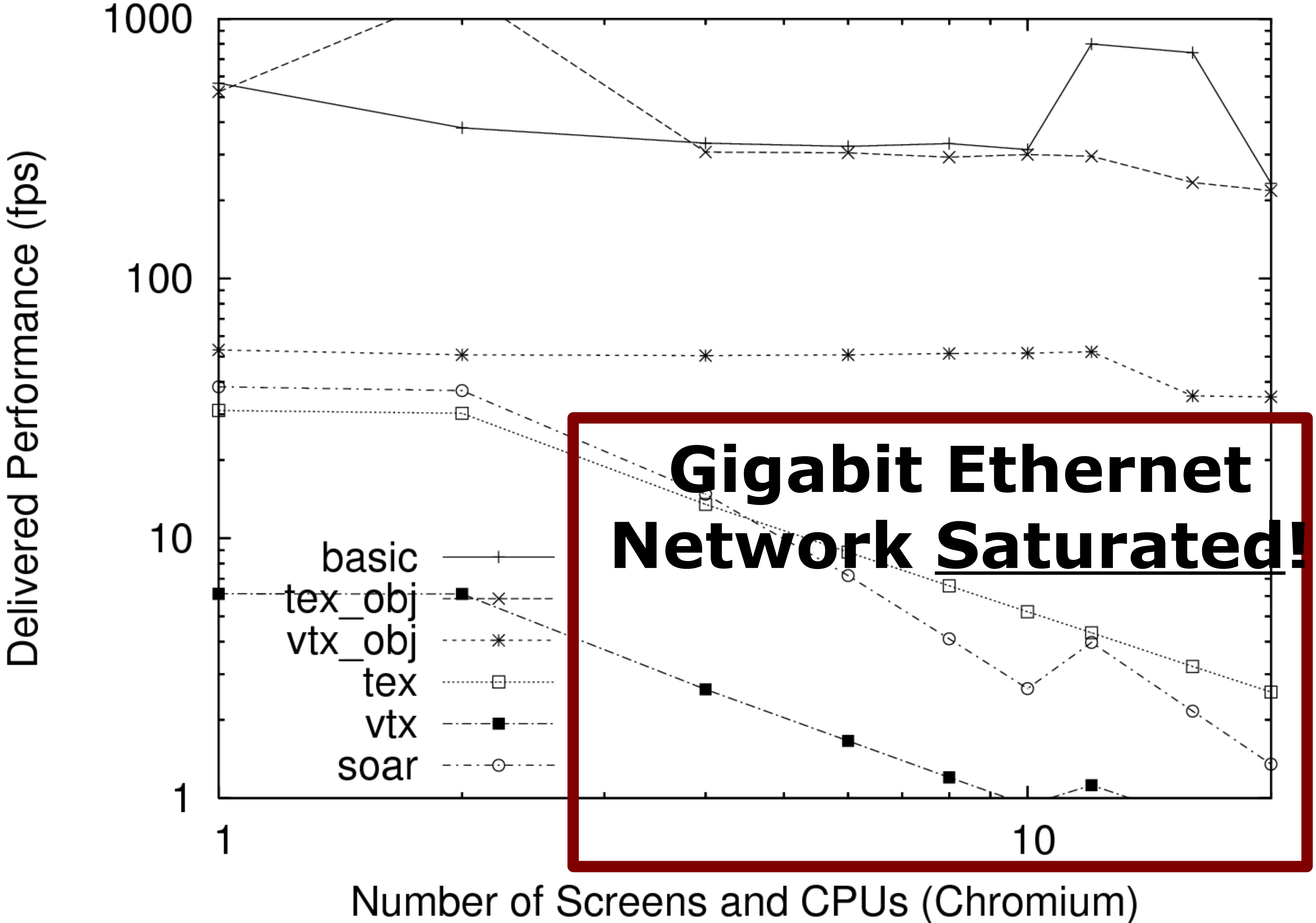
MPIglut Performance



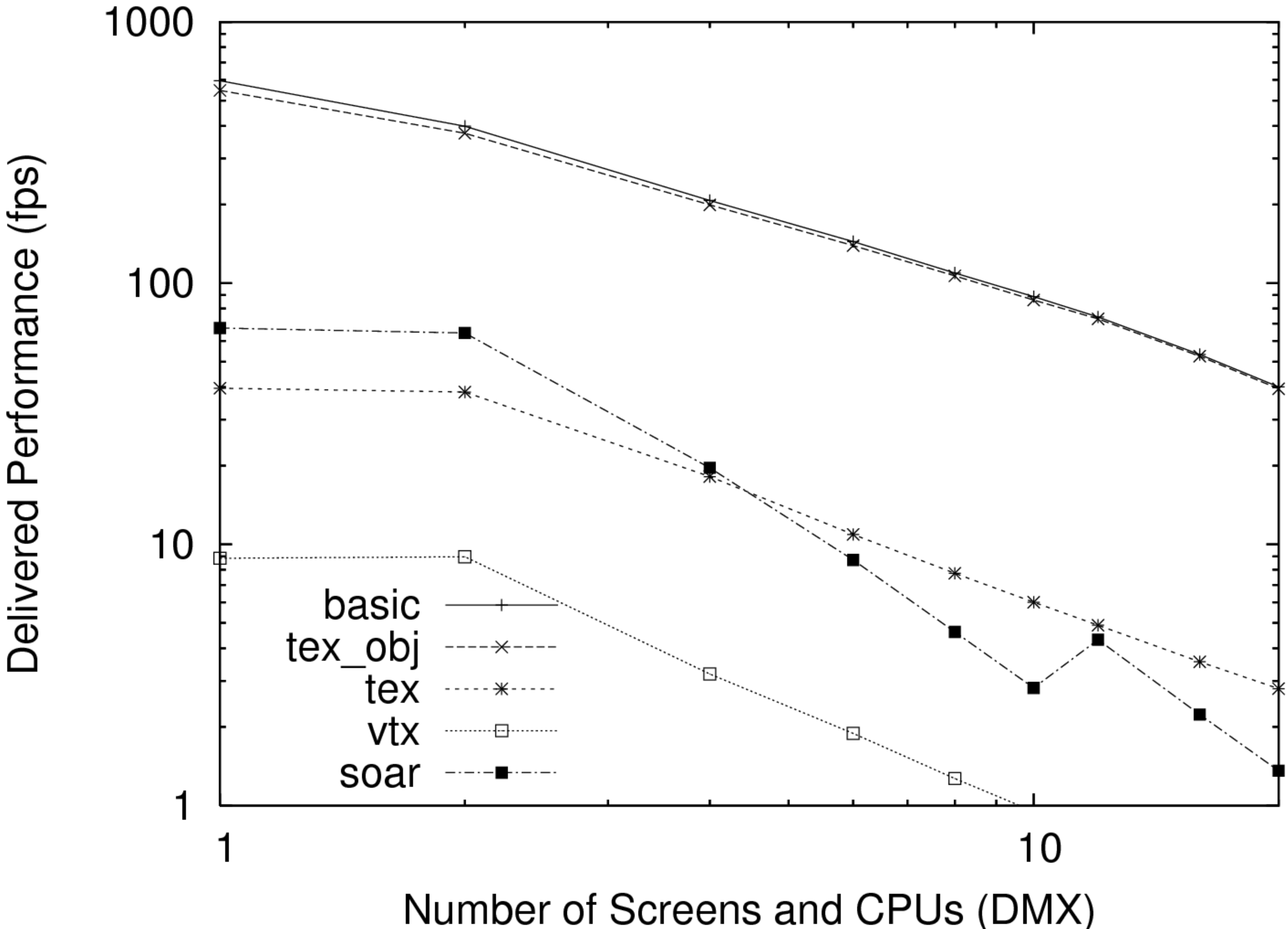
Chromium Tilesort Performance



Chromium Tilesort Performance



DMX Performance



MPIglut Conclusions

- **MPIglut: an easy route to high-performance parallel rendering**
- **<u>Hiding</u> parallelism inside a library is a broadly-applicable technique**
  - **THREADirectX? OpenMPQt?**
- **Still much work to do:**
  - **Multicore / multi-GPU support**
  - **Need better GPGPU support (tiles, ghost edges, load balancing)**
  - **<u>Need</u> load balancing (AMPIglut!)**

Load Balancing a Powerwall

- Problem:

- Solution: Move the rendering for load balance, but you've got to move the finished pixels back for display!

Future Work: Load Balancing

- AMPIglut: principle of persistence should still apply
- But need cheap way to ship back finished pixels every frame
- Exploring GPU JPEG compression
  - DCT + quantize: really easy
  - Huffman/entropy: really hard
  - Probably need a CPU/GPU split
    - 10000+ MB/s inside GPU
    - 1000+ MB/s on CPU
    - 100+ MB/s on network