MPIglut: Powerwall Programming Made Easier

Orion Sky Lawlor[*] Matthew Page[†] Jon Genetti[‡]
Department of Computer Science, University of Alaska Fairbanks

ABSTRACT

A powerwall is an array of separate screens that work together to provide a single unified display. Powerwalls are often driven by a small cluster, which requires parallel software to organize and synchronize the distributed rendering process. This paper describes MPIglut, our powerwall-friendly implementation of the popular sequential GLUT OpenGL 3D programming interface. MPIglut internally communicates using MPI to provide a single coherent display even across a distributed-memory parallel machine. Uniquely, MPIglut is source-code compatible with ordinary sequential GLUT code while providing high performance.

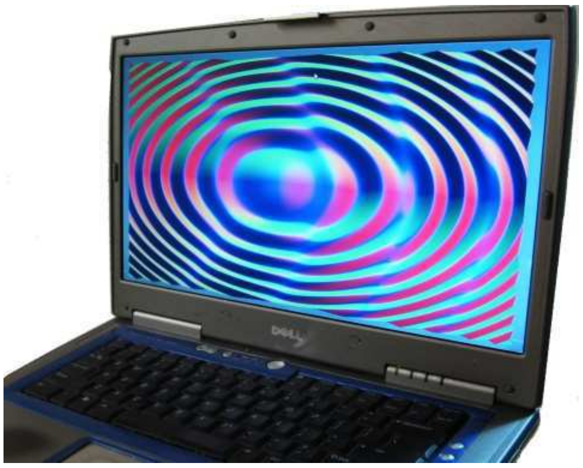
Keywords: Powerwall, large display, GLUT, MPI, OpenGL, API override.

1 INTRODUCTION

After decades of predictions, parallelism is finally arriving in mainstream computing. From instruction-level parallelism in CPUs, to pixel-level parallelism in GPUs, to today's multiple CPU/multiple GPU machines (for example, via multicore and SLI), parallelism at all levels is ubiquitous today.

However, despite its increasing importance, writing code for parallel machines is still difficult [Sut05]. One approach we have pursued recently [Law06] that preserves the millions of man-years invested in sequential software is to build "parallelizing libraries," reusable pieces of parallel code that enable existing sequential programs to operate correctly in parallel. Parallelizing libraries cleanly encapsulate much of the complexity of parallelization, leaving all application-domain complexity to the existing sequential program.

In this paper we describe our open-source parallelizing graphics library called MPIglut. MPIglut is designed to support the many existing sequential OpenGL 3D graphics applications that use the GLUT user interface. As shown in Figure 1, MPIglut allows these applications to operate correctly on a distributed-memory parallel cluster via a simple recompile. Our current primary use for MPIglut is for display walls, or powerwalls[1] [Woo94] [Sch00], where a single application drives a tiled array of physical display devices (such as monitors or projectors) as a large virtual display surface.


Sequential OpenGL/GLUT program running on a laptop.

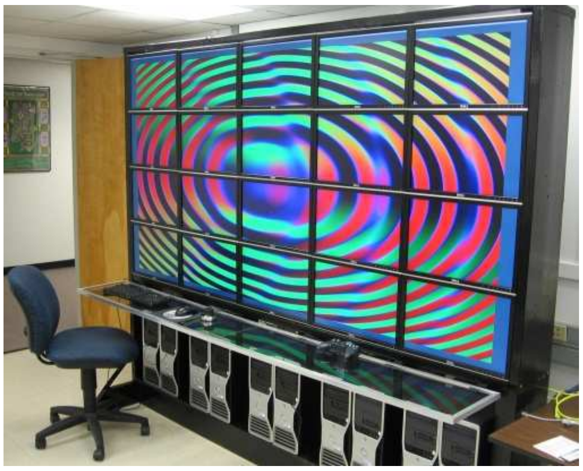

Same program in parallel on a powerwall with MPIglut.

Figure 1: MPIglut allows sequential OpenGL GLUT applications to run efficiently in parallel on powerwall-style tiled display clusters with distributed memory.

Journal of WSCG 2008, ISBN 1213-6964
WSCG'2008, F29, 2008
Plzen, Czech Republic.
Copyright UNION Agency – Science Press

[*]e-mail:olawlor@acm.org
[†]e-mail:ftmap2@uaf.edu
[‡]e-mail:genetti@cs.uaf.edu

[1] PowerWall (note capitalization) is a trademark of Fakespace Systems.

1.1 Prior Work

Many libraries already exist for adapting applications to a tiled parallel display—see Staadt et al's 2003 survey [Sta03]. Table 1 summarizes some of this prior work by the parallelism used in the geometry-generating application and the geometry-rendering display.

Molnar et al [Mol94] provided a popular three-level taxonomy of approaches to parallel rendering: sort-first (send data before rasterization), sort-middle (send data during rasterization), and sort-last (send data after rasterization). Because rasterization is not the only noteworthy event in graphics programming, we find a slightly more fine-grained taxonomy useful:

1. *send-event*: The user interacts with the program via window system events. MPIglut and VR Juggler broadcast these events across the network, and are hence send-event systems. One advantage of this is events are normally far smaller than any other stage in the system.

2. *send-database*: The program responds to those window system events by traversing its scene database. Several parallel scene graph libraries, described below, are able to respond to changing viewpoints by sending the appropriate parts of the scene database across the network to their new displays.

3. *send-geometry*: The program generates renderable geometry for the scene by making calls to the graphics interface library. Chromium captures OpenGL calls at this level with its own libGL; DMX captures the GLX protocol stream generated by the stock X OpenGL library. The captured geometry is then potentially sent across the network to a different GPU for rendering, such as via Chromium tilesort. This is Molnar's "sort-first" level.

4. *send-groups*: During rasterization setup, many renderers decompose primitives into groups of pixels such as scanlines. This "Scan Line Interleave" approach was used with multiple 3dfx graphics cards, and is Molnar's "sort-middle" level.

5. *send-pixels*: After rasterization, rendered pixels must be delivered to the appropriate display and possibly composited together. The common approach is to divide the display surface into tiles and (possibly dynamically) assign a renderer to each tile. ATI's CrossFire, and IBM's scalable graphics engine [Pra05] network-attached-framebuffer work at this level to composite rendered pixels. Chromium's readback component also provides support for this, Molnar's "sort-last" compositing.

The Chromium [Hum02] system, formerly WireGL [Hum00], captures all OpenGL rendering calls sent to its special OpenGL library. The captured OpenGL calls can then be sent across the network to other processors for rendering in a flexible way, so Chromium can either distribute the calls coming from a single sequential application, or route the calls from pieces of a parallel application to the appropriate parallel or serial display. Because it uses binary call interception, Chromium is compatible with most OpenGL binaries. But because Chromium must intercept and forward all OpenGL calls, it cannot help but heavily intrude upon the rendering process. This makes the library difficult to extend to follow the evolving OpenGL standard, and also has performance implications. Finally, Chromium does not provide much assistance with application-level parallelization, although it does come with a GLUT-like library called CRUT, and provides unrendered geometry and rendered pixel communication.

Distributed Multihead X (DMX) [Mar] is an X Window System server that splits up incoming graphical user interface requests and forwards them to a list of "backend" X servers. DMX is often used on powerwalls to allow ordinary unmodified sequential X applications to run on the parallel tiled display. DMX also includes GLX Proxy, an implementation of X's native OpenGL network transmission protocol (GLX) which broadcasts each GLX request to all machines for rendering. Exactly like Chromium, GLX Proxy thus intrudes on every rendering operation, which can be slow and makes it difficult to keep up to date as OpenGL changes. DMX's GLX Proxy is purely broadcast-based, and does not do any of the intelligent geometry routing performed by Chromium's tilesort.

Like MPIglut, VR Juggler [Bie01] only handles event reception and OpenGL setup, leaving OpenGL rendering largely to the user. VR Juggler works in CAVE systems, supporting 3D head trackers and displays at arbitrary 3D orientations. A similar library specifically for SGI Performer hardware was pfCAVE [Pap97].

A number of libraries exist which provide a parallel scene graph interface. OpenSG [Rei02] (which is not related to OpenSceneGraph) provides a replicated scene graph that can be modified and rendered by multiple threads or the distributed machines of a cluster. To cite a few, Syzygy [Sch03], Aura [vdS02], OpenRM Scene Graph [Bet03], and Coin3D [Sys] are among the many feature-rich parallel scene graph libraries, which often target tiled displays. But the main barrier to adoption of all these libraries is that they are not, and cannot be, anything like classical immediate-mode OpenGL. This means existing 3D programs must be almost totally rewritten to take advantage of their features. MPIglut by contrast aims for source code compatibility. In the scene graphs' defense, MPIglut implicitly assumes the original program is capable of rendering any portion of the scene at any time, so even under MPIglut a parallel view-culling scene graph is still quite useful for large models.

	Single Display	Multiple Displays
Serial Application	Serial toolkits like Windows, X, GLUT, etc	DMX [Mar]
Parallel Application	ParaView, Tachyon MPI Raytracer [Sto98], etc	MPIglut, VR Juggler [Bie01], Aura [vdS02]

Table 1: Classification of prior work by primary use. Chromium [Hum02] can be used for all four cases.

2 IMPLEMENTATION OF MPIGLUT

MPIglut implements a parallel version of the OpenGL Utilities Toolkit (GLUT) standard [Kil96]. GLUT is normally a sequential windowing and GUI event handling interface called by sequential programs. MPIglut parallelizes GLUT programs by running a separate copy of the user's sequential code on each of a set of MPIglut rendering processes called "backends". Each backend is responsible for rendering a small part of the overall display, although MPIglut provides the user's sequential code the appearance that it is rendering to the entire display.

MPIglut is built on top of a sequential GLUT implementation, which handles user input at the front end and the render system interfacing at the back end. We currently use a patched version of freeglut[Ols07] 2.4.0, since MPIglut requires one small modification to the underlying GLUT in order to work well with DMX (MPIglut forces its backend windows to be X children of the DMX backend window, which prevents window-stacking order and event routing problems). Also, MPIglut intercepts a few GLUT and OpenGL calls for special handling:

- MPIglut's glutInit on a backend calls MPI_Init, sets up MPIglut's internal state, and calls the underlying glutInit. On the frontend, glutInit spawns the appropriate number of backends (using mpirun) and forwards user events to those backends.
- MPIglut's glutCreateWindow (and other window manipulation calls, such as glutReshapeWindow) forwards the request to the frontend, which adjusts its window and correspondingly reorganizes the backends.
- MPIglut's glutMouseFunc (and all other user event handling functions) calls the user's callbacks based only on events broadcast from the frontend.
- MPIglut's glutGetModifiers returns the frontend's keyboard state as of the last event broadcast.
- MPIglut's glViewport command internally asks for an OpenGL viewport covering only our backend's screen region. This avoids the OpenGL implementation's GL_MAX_VIEWPORT_DIMS limit, which is often as low as 4096 pixels—less than half the display width of our 8400x4200 pixel powerwall!
- MPIglut's glLoadIdentity (and the other matrix load functions) pre-loads this backend's subwindow matrix, as described in Section 2.3.
- MPIglut's glutSwapBuffers synchronizes all displays (using a glFinish and MPI_Barrier). This avoids tearing and lag effects as slower or more heavily-loaded backends fall behind faster ones.

MPIglut's call interception scheme currently uses the preprocessor. For example, inside our MPIglut public header file, we intercept glLoadIdentity calls with the simple C/C++ preprocessor macro "#define glLoadIdentity mpiglLoadIdentity". For full binary compatibility, it would be straightforward to implement a shared-library technique such as LD_PRELOAD or even construct an entirely new replacement library, similar to Chromium [Hum02]. But for mere source-code compatibility the preprocessor is very small and simple.

2.1 Parallel Programming with MPI

Underneath, MPIglut uses the parallel Message Passing Interface (MPI) standard [MPI94] to synchronize and communicate GUI events between the backend processes. We currently use MPICH 1.2.7 [Gro96] as our MPI implementation, although any implementation of MPI should work. MPIglut programs are not required to make any MPI calls themselves, but are free to call MPI functions if needed, for example to accomplish some application-specific communication not provided by MPIglut.

Several of the best aspects of MPIglut are taken directly from MPI. Unlike with threaded multiprogramming, MPI and MPIglut run a completely separate copy of the main() program in each of the parallel backend processes. This avoids many of the race conditions common with threaded parallel programming, avoids slow and error-prone locking, and allows the entirely safe use of global or static variables by MPI and MPIglut programs.

One obvious major drawback of non-shared memory parallel programming is the potential for duplication of large shared data structures. However, if the larger shared structures are memory-mapped in from files, the OS kernel will safely point all local processes' pagetables at one copy of this common data, and so multiple processes can be made memory-use-competitive with multithreaded programming even on shared-memory hardware.

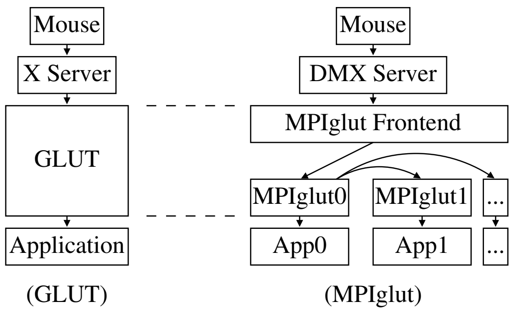


Figure 2: Sequential GLUT normally receives events from the X server and forwards them to the application's event handler callbacks. MPIglut receives events at the frontend and broadcasts them out to all the backends. Broadcast events are then delivered to the application's event handler callbacks collectively.

2.2 Event Delivery

As shown in Figure 2, MPIglut receives user input events such as keystrokes and mouse motion using a single placeholder "frontend" process. This frontend process then sends the incoming events over a TCP socket to one backend process, where the events are broadcast via MPI to all the backends.

The semantics of some calls in MPI and MPIglut are "collective", meaning they must always happen in the same order on every backend process. In MPIglut, event reception and delivery is collective, so every backend is guaranteed to receive the same user input events in the exact same order. Collective calls usually allow the programmer of an MPIglut (or MPI) process to safely ignore the confusing unsynchronized execution common to parallel programming, and think of the processes as executing together in lock-step.

Applications must ensure they retain this collective property when they make GLUT windowing and overall rendering control calls such as glutSwapBuffers. Deterministic applications automatically remain collective. Applications that determine window state based on a nondeterministic function of their (identical) input data, (identical) command-line arguments, and (identical) user events would require additional synchronization to work properly under MPIglut. For example, additional code would be needed to synchronize applications based on a non-shared clock, or that already render data from the network. However, no OpenGL rendering commands (such as glDrawLists) are collective or intercepted by MPIglut, so all are safe to call in any order and all run at full speed.

2.3 MPIglut Coordinate Systems

MPIglut currently fetches both input events and screen geometry from the frontend's DMX window, although it would be trivial to have MPIglut fetch this information from some other program or a configuration file.

Most events in MPIglut are delivered collectively to the user code in *global coordinates*, the coordinates of the DMX display running across the collective virtual display screen. Because global coordinates are the same everywhere, user code never needs to translate coordinates due to MPIglut.

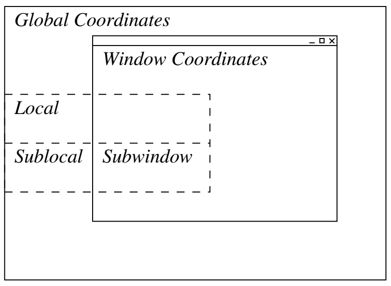


Figure 3: Coordinate systems used inside MPIglut.

But as shown in Figure 3, internal to MPIglut there are no fewer than five separate coordinate systems that must stay properly interrelated.

- *Global coordinates* are coordinates on the entire collective virtual screen. Global coordinates (0,0) are the top-left corner of the whole powerwall. These coordinates are used by DMX and the user code to specify window positions.
- *Local coordinates* are the coordinates of the local machine's directly-attached screen. Local coordinates (0,0) are the top-left corner of this MPI process's attached physical display. These are used internal to MPIglut backends to position windows on the local screen.
- *Sublocal coordinates* mark our backend process's portion of its own directly-attached screen. Sublocal coordinates (0,0) are the start of the portion of screen space this process is responsible for drawing. They are different from local coordinates because we may wish to have more processes than screens, for example on a multi-core machine.
- (Global) *Window coordinates* are measured on the frontend's virtual window. Window coordinates (0,0) are the top-left corner of the collective frontend window. All mouse events are reported by DMX and to the user code in these global window coordinates. OpenGL viewports are requested by the user code in window coordinates.
- *Subwindow coordinates* are the part of the window our local backend is responsible for drawing. Subwindow coordinates (0,0) are the topleft corner of where we actually must draw. OpenGL rendering actually happens in subwindow coordinates.

In the simplest case of one process drawing to a single fullscreen window, all five coordinate systems are identical! In any powerwall, global and local coordinates are different, but local and sublocal coordinates may still be identical. Sublocal coordinates are also useful for separating the images being delivered to two separate displays from a dual-output graphics card with a single contiguous framebuffer.

During rendering, the main task of MPIglut is simply to convert the global window coordinates used by the sequential user code (which knows nothing of the separate powerwall screens) into subwindow coordinates as used by the local graphics card to drive a portion of the display. For rendering, this coordinate shift should happen after the perspective divide, but before vertex clipping. In OpenGL, we simply need to fill the GL_PROJECTION matrix with the window-coordinates-to-subwindow-coordinates matrix—called the "subwindow matrix"—before any other matrix operations.

Because the MPIglut implementation of glLoadIdentity premultiplies the subwindow matrix into the projection matrix,[2] then any code that reads back this matrix (for example, via a glGetFloatv call) will instead receive the projection-to-subwindow matrix. This is a feature, not a bug! It means applications that construct clipping planes from the projection matrix will actually automatically cull away geometry they are not responsible for drawing locally. In other words, under MPIglut often well-written sequential OpenGL programs **will not** replicate every drawing call across the entire powerwall, but instead only load and draw the geometry visible on their own local piece of the overall display. The *soar* application we used for benchmarking generates only the geometry needed on each backend in this intelligent fashion, and a web search for "glGetFloatv culling" finds hundreds of similar applications.

3 PERFORMANCE RESULTS

We benchmarked MPIglut's performance against both Chromium[3] and DMX[4] on our 20-screen powerwall, shown in Figure 4, which consists of ten nodes[5] connected with switched gigabit ethernet. The aggregate resolution of the 5x4 array of 20 screens is 8400x4200 pixels, not counting the 150-pixel gap between screens, which once accounted for increase the overall display dimensions to 9000x4650 pixels.

[2] The premultiplication of course only happens in GL_PROJECTION mode.

[3] Chromium 1.8, using DMX tilesort client and crserver render SPUs.

[4] Xorg DMX 7.1.1 version of DMX, running with glxProxy.

[5] Software: 32-bit Linux 2.6.15, nVidia 87.62 drivers, gcc 4.04, and MPICH 1.2.7. Hardware: dual-core Intel Core2 Duo 6300 CPU, 2GB RAM, and one nVidia QuadroFX 3450 or 1450 PCI Express graphics card connected to two 1680x1050 DVI LCD monitors.

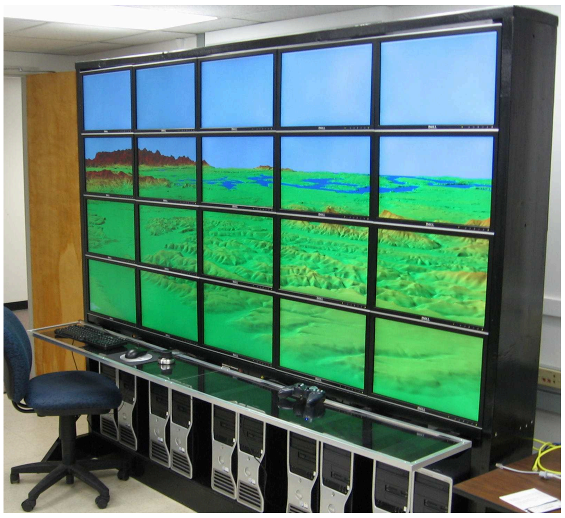


Figure 4: The UAF CS Bioinformatics powerwall, running the *soar* terrain renderer used for benchmarking.

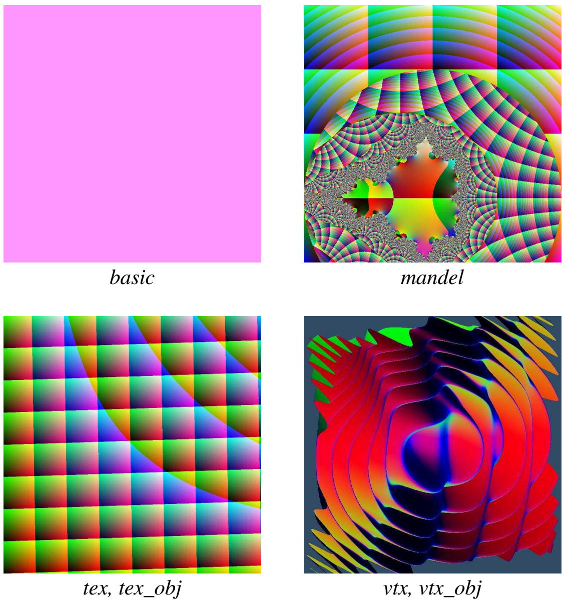


Figure 5: Output of six of our seven benchmark programs (the bottom row images represent two benchmarks each). The *soar* benchmark is shown in Figure 4.

To show different performance aspects, we present results from seven small GLUT programs as shown in Figures 4 and 5, and described in detail below. Each of these programs began as an ordinary serial GLUT program, but ran without problems in parallel using MPIglut. Figure 6 and Table 2 show framerates for each program.

Parallel programmers will notice that powerwall rendering is naturally a "scaled problem"—because we add screens, CPUs, and GPUs at the same rate, with zero communication or synchronization cost our framerate would remain constant regardless of the machine's

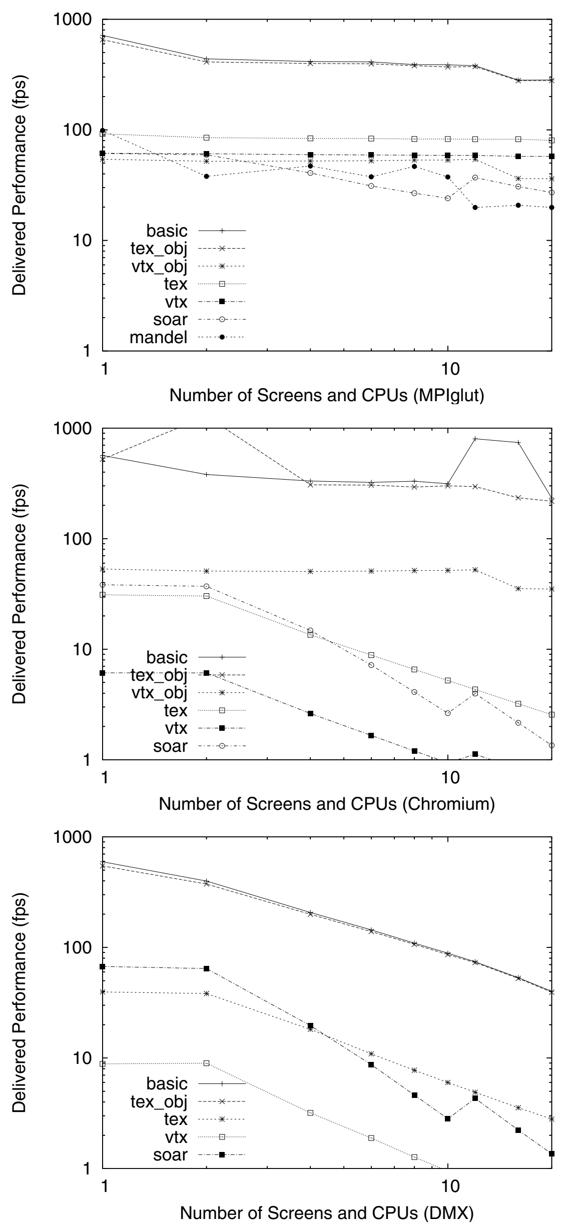


Figure 6: Framerate as a function of machine size for all our benchmarks, running under MPIglut, Chromium, and DMX. Machine sizes: 1, 2, 4, 6, 8, 10, 12, 16, and 20 screens and CPUs. Framerates below 10fps are unusable (log-log scale).

size (or "scale"). Hence a communication solution that "scales" will have near-constant framerates as a function of machine size. Communication costs show up as a fall-off in framerate as the machine scales up.

- *basic* draws one fullscreen quad of a fixed color per frame. This was intended as a baseline to test frame synchronization cost. Both MPIglut and Chromium sustain hundreds of frames per second out to the full 20 CPUs, but DMX scales poorly even for this simple program, ending up just below 40fps.

	MPIglut	Chromium	DMX
basic	282.5	232.6	40.1
tex_obj	279.3	217.9	39.4
vtx_obj	36.1	35.0	fail
tex	80.3	2.6	2.8
vtx	57.5	0.7	0.4
soar	27.1	1.4	1.4
mandel	19.9	fail	fail

Table 2: Framerates (frames/second) of our seven benchmark GLUT programs running under MPIglut, Chromium, and DMX on 20 screens and CPUs.

- *tex_obj* draws one fullscreen quad using a 1024x1024 texture loaded from an OpenGL texture object. All three systems were able to locally cache the texture, so the performance of this test was similar to the *basic* test.

- *vtx_obj* draws a 2-million triangle mesh from an OpenGL vertex buffer object (loaded with a usage of GL_STATIC_DRAW_ARB). Again, MPIglut and Chromium were able to locally cache the mesh object, and hence maintained good performance. DMX does not support the 2003 ARB_vertex_buffer _object OpenGL extension, and so could not execute this program.

- *tex* draws one fullscreen textured quad exactly like *tex_obj*, but reloads the 1024x1024 texture's data from the CPU every frame using glTexSubImage2D. This is intended to mirror a high-definition movie player using software decoding, or other live external data display. MPIglut uses the parallel CPUs to load all the textures in parallel, and hence scales perfectly. Chromium and DMX must broadcast the updated texture over the network every frame, and scale terribly as expected.

- *vtx* draws a 320-thousand triangle mesh using an OpenGL vertex array rendered with glDrawElements. Unlike *vtx_obj*, vertex arrays cannot be stored in the GPU, and must be copied from the CPU every frame. As with *tex*, under MPIglut each node uses its local copy of the data and hence the vertex upload scales well, while Chromium and DMX must send all the vertex data via the network every frame and hence do not scale.

- *soar* is Peter Lindstrom et al's SOAR v1.11 terrain renderer [Lin02] using a flight path through the 4096x4096 Puget Sound terrain model, which is read from a .geo file on disk. This renderer is CPU-intensive and generally geometry-rate limited, generating and drawing approximately 50,000 polygons per screen per frame. Under MPIglut SOAR scaled fairly well, running at over 27fps even on the entire machine. But because both Chromium and DMX use a single sequential program to generate all geometry on node 0, they both quickly became network bound, and gave terrible performance on the full machine–under 1.5fps!

- *mandel* interactively renders the famous Mandelbrot set fractal using an OpenGL GLSL fragment program, using the recently added hardware pixel shader loop and branch support. Because rendering pixels in different regions of the Mandelbrot set requires dramatically differing numbers of iterations, this program's parallel speed under MPIglut varies substantially due to load imbalance between the different backends, but is still acceptable. Chromium and DMX do not yet support programmable shaders, and hence neither one could execute this program.

In general we have found that MPIglut scales well for the applications and machines we have tested, providing usable framerates even for difficult applications on the full machine. Similarly, Chromium scales well for some applications, specifically those where the geometry and texture data is either simple or locally cached. But Chromium and DMX both become network-limited for other applications, since they must often send geometry and texture data across the network. We observed Chromium and DMX both saturate gigabit ethernet, often sending over 100 MB/s of geometry and texture data over the network from node 0, sometimes even in a machine configuration with only two nodes!

We measured per-frame network overhead with the trivial *basic* benchmark. On 20 screens, MPIglut ran this program at approximately 300fps (3.28ms/frame), and each machine sent a few kilobytes of data across the network per frame (0.79MB/s maximum total network usage). 82% of each frame time was spent waiting for the GPU to render pixels; 8% (about 300 microseconds per frame) was spent in the MPIglut MPI_Barrier software framesync; and another 8% in MPIglut's event broadcast and delivery. The remaining time, less than 2%, was spent by the CPU actually issuing OpenGL commands. MPIglut's total overhead on 20 screens is thus about half a millisecond per frame, which at a more reasonable framerate amortizes out to a few percent communication overhead (for example, at 30fps, MPIglut takes about 1.5% of the runtime). Chromium had similarly low per-frame overhead, although we occasionally got anomalously high performance in the >200fps region, which may be caused by dropped frames. DMX on 20 screens appears to become network latency limited to 40fps (25ms/frame), despite the low network data rate (under 250KB/s) and CPU and GPU utilization (both under 8% utilized).

We have not evaluated the performance of MPIglut compared to the many quality parallel scene-graph libraries such as VR Juggler [Bie01], though assuming those libraries also use only a small fraction of their time communicating then we expect our overall performance would be comparable. But the reason we have not done this comparison is telling–porting a GLUT application to a non-GLUT library would mean rewriting all the event handling and rendering setup code, which for many real applications is rather painful.

4 CONCLUSIONS & FUTURE WORK

We have presented MPIglut, a minimally invasive library to help sequential GLUT programs run on parallel powerwalls. We have surveyed the architecture of MPIglut, and compared its performance to similar existing libraries. The implementation of MPIglut is small, consisting of one C/C++ header and one two thousand line C implementation file, small enough to be statically linked. MPIglut is still being developed, and we plan to try several promising improvements.

Although currently designed for powerwalls, MPIglut could be used with a single display to more easily take advantage of multi-CPU or multi-GPU parallelism. A single display could be divided into dozens of small strips or tiles, with each region of the screen rendered by a separate local MPIglut MPI process.

When developing complicated applications, MPIglut would be a natural place to add load balancing support, to ensure that each node shares in both application and rendering work. Within each shared-memory screen, static load balance could easily be improved by "overdecomposition": creating many more MPI ranks than physical CPUs, and allowing the OS to schedule the tiles as needed. With standard MPI it is difficult to implement more dynamic forms of load balancing, but a migratable MPI like AMPI [Hua03] could help.

MPIglut could be extended to perform edge blending and color balance correction inside glutSwapBuffers at the end of each frame, which MPIglut already intercepts to provide frame synchronization. One could even resample the finished framebuffer to compensate for geometric nonlinearities in the screen, such as a curved display wall. MPIglut could also be made to work on entirely non-planar displays such as projector domes, although this would likely not be compatible with normal OpenGL projection matrices which assume a flat 2D display.

At the moment, MPIglut does not intercept framebuffer readback routines such as glReadPixels or glCopyTexSubImage2D, so these currently read back only local pixels. For some uses of these functions, such as rendering small or screen-local environment or reflection maps, this provides correct answers. But for other uses of these routines, such as taking screenshots or picking, this gives an incomplete set of pixels. The best solution would probably be to provide optional collective versions of these routines, like mpiglReadPixels.

This support would enable GPGPU applications to be used more easily under MPIglut.

Finally, the idea behind MPIglut is by no means limited to either MPI or GLUT. The source-compatible divide-up-the-screen parallelizing library approach could equally easily be applied to arbitrary graphics toolkits including Microsoft's DirectX or portable GUI libraries such as GTK or Qt, as well as arbitrary communication schemes including threads and bare sockets. We feel parallelizing libraries offer a simple path towards high performance with the increasingly prevalent multi-core and multi-GPU machines.

Readers may download [MPI07] and try MPIglut!

REFERENCES

[Bet03] E. Wes Bethel, Greg Humphreys, Brian Paul, and J. Dean Brederson. Sort-first, distributed memory parallel visualization and rendering. In *Proceedings of IEEE Symposium on Parallel and Large Data Visualization and Graphics*, 2003.

[Bie01] Allen Bierbaum, Christopher Just, Patrick Hartling, K. Meinert, A. Baker, and Carolina Cruz-Neira. VR Juggler: A virtual platform for virtual reality application development. In *IEEE Virtual Reality*, pages 89–96, 2001.

[Gro96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. Mpich: A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[Hua03] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, pages 306–322, College Station, Texas, October 2003.

[Hum00] Greg Humphreys, Ian Buck, Matthew Eldridge, and Pat Hanrahan. Distributed rendering for scalable displays. In *IEEE Supercomputing*, pages 60–60, 2000.

[Hum02] G. Humphreys, M. Houston, Y. Ng, R. Frank, S. Ahern, P. Kirchner, and J. Klosowski. Chromium: A stream processing framework for interactive graphics on clusters. In *SIGGRAPH Proceedings*, pages 693–702, 2002.

[Kil96] Mark J. Kilgard. The opengl utility toolkit (glut) programming interface: Api version 3, 1996. http://www.opengl.org/documentation/specs/glut/.

[Law06] Orion Sky Lawlor, Sayantan Chakravorty, Terry L. Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant V. Kale. ParFUM: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering With Computers*, 22(3):215–235, 2006.

[Lin02] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified. *IEEE Viz. and Graphics*, 8(3):239–254, 2002.

[Mar] Kevin E. Martin, David H. Dawes, and Rickard E. Faith. Distributed Multihead X (DMX). http://dmx.sourceforge.net/.

[Mol94] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. In *IEEE Computer Graphics and Applications*, volume 14-4, pages 23–32, July 1994.

[MPI94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[MPI07] MPIglut authors. MPIglut Project Page, 2007. http://www.cs.uaf.edu/sw/mpiglut/.

[Ols07] Pawel W. Olszta, Andreas Umbach, and Steve Baker. freeglut, 2007. http://freeglut.sourceforge.net/.

[Pap97] Dave Pape. pfCAVE CAVE/Performer Library (CAVELib 2.6), 1997. http://www.evl.uic.edu/pape/CAVE/prog/.

[Pra05] Prabhat and Samuel G. Fulcomer. Experiences in driving a cave with IBM Scalable Graphics Engine-3 (SGE-3) prototypes. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 231–234, New York, NY, USA, 2005. ACM Press.

[Rei02] D. Reiners, G. Voss, and J. Behr. OpenSG: Basic concepts. In *First OpenSG Symposium*, 2002.

[Sch00] Daniel R. Schikore, Richard A. Fischer, Randall Frank, Ross Gaunt, John Hobson, and Brad Whitlock. High-resolution multiprojector display walls. *IEEE Comput. Graph. Appl.*, 20(4):38–44, 2000.

[Sch03] B. Schaeffer and C. Goudeseune. Syzygy: Native PC Cluster VR. In *IEEE Virtual Reality*, 2003.

[Sta03] O. Staadt, J. Walker, C. Nuber, and B. Hamann. A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering. In *Proceedings of the Workshop on Virtual Environments*, 2003.

[Sto98] John Stone. An efficient library for parallel ray tracing and animation. Master's thesis, Dept. of Computer Science, University of Missouri Rolla, 1998. http://jedi.ks.uiuc.edu/˜johns/.

[Sut05] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[Sys] Systems In Motion. Coin3d library. http://www.coin3d.org/.

[vdS02] T. van der Schaaf, L. Renambot, D. Germans, H. Spoelder, and H. Bal. Retained mode parallel rendering for scalable tiled displays. In *Immersive Projection Technologies Symposium*, 2002.

[Woo94] Paul Woodward and U. Minnesota PowerWall Team. Powerwall, 1994. http://www.lcse.umn.edu/research/powerwall/.