Stand in for the signature page. I'm not sure if I provide this or the department has its own.

PARALLEL RENDERING AND DISPLAY OF TERRAIN DATA ON A POWERWALL

A

PROJECT REPORT

Presented to the Faculty

of the University of Alaska Fairbanks

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

BY

MATTHEW A. PAGE, B.S.

Fairbanks, Alaska

May 2007

Abstract

With recent advances in cheap common off the shelf personal computers with fast graphics cards, it has become easier for people to harness the power of large display systems. A powerwall is a generic term for a large tiled display of many common PC monitors or projector displays. Each of these displays works together to produce a cohesive image with extreme detail. MPIglut is a powerwall friendly implementation of the common OpenGL User Toolkit (GLUT) that is source code compatible with normal GLUT, which is used worldwide to help teach OpenGL programming. MPIglut allows a programmer to leverage current knowledge of GLUT to program a powerwall without requiring the learning of the intricacies of the Powerwall. For this project, SOAR, a serial glut terrain drawing program, was modified to work on a powerwall using MPIglut. The overall goal of MPIglut is to make powerwall programming easier.

Table of Contents

Page

Signature Page....i
Title Page....ii
Abstract....iii
Table of Contents....iv
List of Figures....v
List of Appendices....vi
Glossary....vii

1 Introduction....
1.1 Prior Workstation....
1.1.1 Chromium....
1.1.2 DMX....
1.1.3 VRJuggler....
1.1.4 Scene Graph Libraries....
1.2 Limitations of Prior Work....
1.3 Introducing MPIglut....
2 Implementation of MPIglut....
2.1 DMX....
2.2 MPIglut....
2.2.1 Parallel Programming with MPI....
2.2.2 Startup of an MPIglut program....
2.2.3 Dataflow with GLUT and MPIglut....
2.3 SOAR....
2.3.1 Multi-Threading....
2.3.2 Input....
2.3.3 Adaptive Quadtree....
3 Performance and Scalability....
3.1 Benchmarks....
4 Limitations of MPIglut....
5 Lessons Learned....
6 Conclusions and Future Work....
7 References....

List of Figures

1. The UAF CS Bioinformatics powerwall.....
2. GLUT and MPIglut user event handling.....
3. SOAR Input Files.....
4. SOAR Adaptive Quadtree.....
5. SOAR Sample Image.....
6. MPIglut SOAR Frames per Second.....
7. MPIglut SOAR Triangles per Second.....
8. MPIglut SOAR Bytes Transferred per Frame.....

Need more pictures... Any suggestions?

List of Appendices

Appendix A – MPIglut How To.....
Appendix B – System Stats.....
Appendix C – Software.....
Appendix D – System Layout.....
Appendix E – Performance Data.....
Appendix F – Code.....

Glossary

powerwall –	A generic term for a tiled display of high resolution screens.
MPI –	Message Passing Interface – A tool used for parallel programming
DMX –	Distributed Multihead X – DMX is used to make several screens act as one large screen
GLUT –	OpenGL User Toolkit – A cross platform user toolkit used to teach OpenGL
SOAR -	Stateless Adaptive One-pass Refinement – Terrain generator implementing techniques described in the paper, Terrain Simplification Simplified: A general framework for view-dependent out-of-core visualization.
MPIglut –	A cross between MPI and GLUT - MPIglut makes powerwall programming easier.

Any others?

The UAF CS BioInformatics powerwall

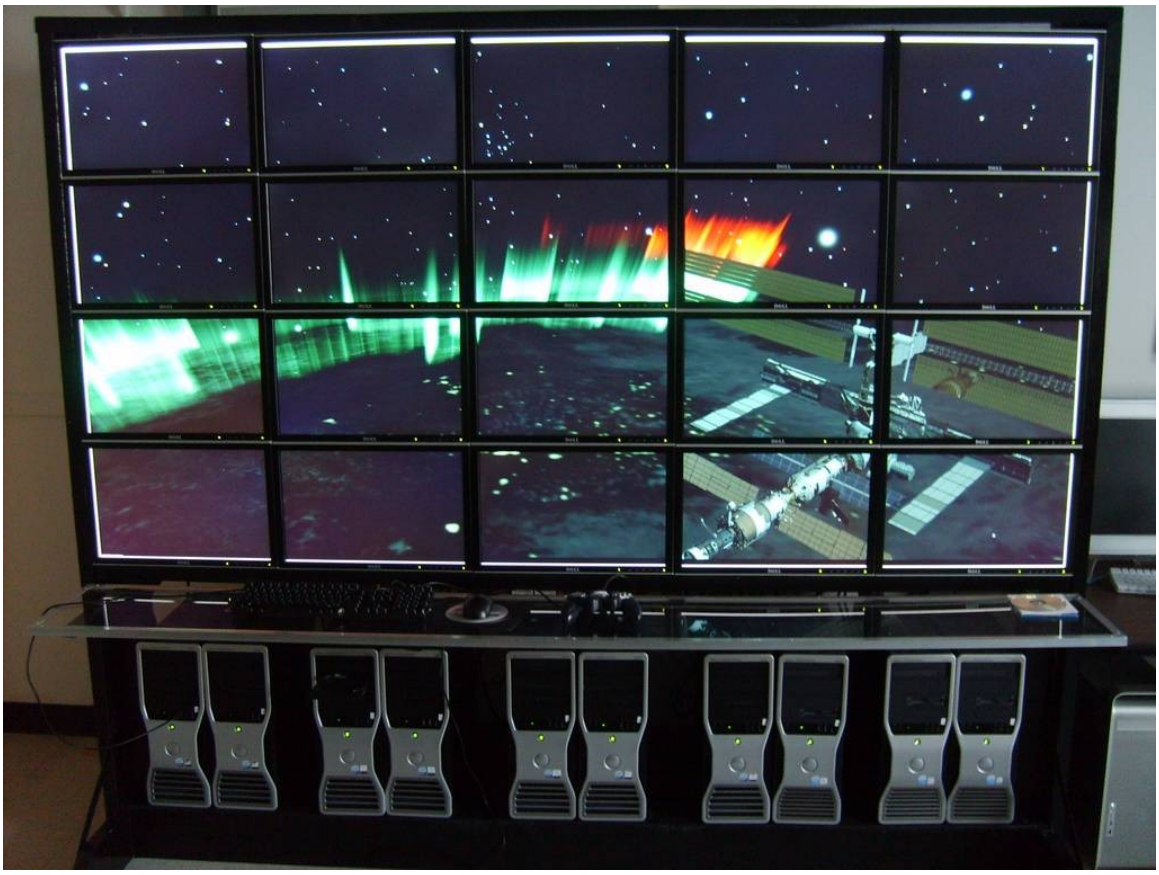


Figure 1: The UAF CS BioInformatics powerwall. 20 screens working together to display a high resolution image of a computer generated view of the Aurora Borealis as seen from space.

1 Introduction

A powerwall is a very useful tool for visualizing large amounts of data without having to constantly zoom in to see detail and then zoom out to keep the big picture in mind. Traditionally, powerwalls have many screens connected to one powerful computer which has many graphics cards. With the reduction in price and increase in power of common computers, an effective powerwall can be created at a relatively low cost. While there is a lot of software out there that will allow programs to run on the entire powerwall screen, there aren't a lot of programs that will allow the programmer to harness the power of all of those computers.

Recently the big name computer processor manufacturers have been creating central processing units (CPU) with multiple cores on the same chip. This is like have two or more complete CPU's in the same computer that are available to work. The problem with these systems is that it's still hard to effectively use all of the available processing power. Even though programming for multiple CPU's (parallel programming) has been around almost from the advent of modern computing, it still isn't a 'solved problem'

Most programs are serial programs that are designed to only display information on one screen which has been calculated using one computer. A serial program can't handle large powerwalls, but parallel programs are tough to write. What is needed is an easy way to harness the power available in each of the separate computers in a powerwall to generate what is to be displayed on each display in the powerwall. MPIglut makes powerwall programming easier. (I need to reword this, but I can't think of a way right now.)

This project explores the question of whether a serial terrain generating program can be ported to run on a powerwall using MPIglut and maintain good scalability and performance.

1.1 Prior Work

There are many toolkits that allow one to write programs that will work on a cluster of computers. There are also a lot of toolkits that allow one to write programs that will take advantage of the multiple screens offered by a powerwall. Each of these has strengths and weaknesses.

1.1.1 Chromium

Chromium [CHROMIUM], formerly WireGL [WIREGL], is an OpenGL abstraction layer. It intercepts all OpenGL calls on a system by replacing the dynamic OpenGL libraries and instead of delivering the instructions to the local rendering system, it sends them over the network to receiver machines that actually do the display rendering. In doing this, Chromium uses a lot of bandwidth and quickly becomes network bound as the

powerwall display scales in the case of geometry-limited programs. Chromium works well for pixel-fillrate limited programs. Possibly Chromiums greatest advantage is that it does allow unmodified OpenGL applications to run due to the way that it intercepts OpenGL calls. It is possible to run Chromium applications even without access to the original source code for the program. Chromium, however, does not help a user with parallelization which means that Chromium applications are limited to one machine doing the processing.

1.1.2 DMX

Distributed Multihead X (DMX) [DMX] is an X Window system server which provides multihead support for multiple displays attached to different machines, each of which is running an X server. A typical X server only provides multi-head support for multiple displays attached to the same machine. DMX can also be used to allow a non-powerwall aware sequential application to run on the powerwall without modification. DMX includes GLX Proxy which allows DMX to also broadcast OpenGL calls to other machines in the powerwall for them to render. Unfortunately, this process is slow for DMX displays that span multiple machines. DMX is somewhat slower than Chromium, because Chromium is more intelligent about the information it sends. While DMX sends all OpenGL calls to every node in the powerwall, Chromium only sends what each node needs to display its part of the world.

1.1.3 VRJuggler

VRJuggler is a programming library that can be used to display 3D applications on a powerwall. [VRJUGGLER] Like MPIglut, VRJuggler only handles event reception leaving most OpenGL rendering code up to the user. This provides a low bandwidth solution to parallel rendering of OpenGL data. Unfortunately, VRJuggler applications need to be written against the input abstraction layer. Like MPIglut, a separate copy of the program must run on each node of the cluster. [OPENSGVRJUGGLER]

1.1.4 Scene Graph Libraries

There are many Scene Graph programming libraries out there that will help a user create a display for a powerwall. In general, these are very powerful and can be somewhat complicated to set up. OpenSG helps automate much of the process and even helps a user write cluster aware applications that will display on the powerwall. The problem with OpenSG and other scene graph libraries is that they sometimes require heavy code modification to work.

1.2 Limitations of prior Work

All of these methods take advantage of the high resolution of a powerwall, but each also has limitations that make it the non-ideal solution for general powerwall applications. Some run on one machine and feed data to each of the other machines in the cluster for display. This paradigm worked well when you had one big server that had multiple

display cards attached, but doesn't scale to larger powerwalls.

A CPU bound program will wither if it tries to calculate an entire powerwall worth of information on one CPU as happens with Chromium and DMX. Also, Chromium and DMX quickly become network bound as the number of displays increase. Their framerates drop well below acceptable even with moderately simple scenes. One CPU and graphics board can't keep up with the demands requested of it no matter how fast and powerful it is.

Except for Chromium, most of these systems all have another possibly major drawback. The program has to be written or rewritten to specifically take advantage of the power they offer. Again, except in the case of Chromium, it is not possible to take an unmodified program and just run it on the powerwall.

None of the systems described do anything about load balancing either. Ideally, if one were working with N CPUs and graphics cards, then any one of those N would handle $\frac{1}{N}$ of the work. Most of these solutions have just one CPU/Graphics card doing ALL of the work for every other machine.

1.3 Introducing MPIglut

MPIglut addresses most of these problems with minimal effort on the part of the programmer. It is network efficient as the data transmitted consists only of the minimum amount of data needed to reflect changes made by the user such as mouse movement and keyboard key presses. MPIglut also allows unmodified GLUT code to run directly on the powerwall without the programmer having to change any code. All that is required is the code be recompiled on the powerwall.

2 Implementation of MPIglut

2.1 DMX

MPIglut uses DMX to handle much of the administrative tasks that are necessary when you have a screen that is spread across multiple displays on multiple machines. DMX keeps virtual track of the location of each of the windows that are on the display as well as their height and width. It handles displaying their widgets and borders correctly. DMX also keeps track of where the mouse pointer is located and whether or not it can be seen. Finally, DMX also handles the keyboard and mouse input from the Master node and the dissemination of this information to all of the machines in the display cluster.

DMX is configured using a standard text file that describes the location of each of the nodes based on IP address or hostname and what the viewable area is for each of those displays. Each of these machines must have an X server already running in order for DMX to be able to 'take over' the display. On the master node, DMX creates a new X server which is usually :1. This X server spans all of the screens in the config file. When user events occur, such as mouse movement or key presses, DMX forwards them to all of the other machines in the DMX configuration.

MPIglut configures itself based on this information that it receives from DMX. It would be simple for MPIglut to get this information directly from the DMX config file or from a separate config file, but DMX also handles so many other details that it is advisable to keep it at this time.

2.2 MPIglut

The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system.

The two aims of GLUT are to allow the creation of portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. Getting started with OpenGL programming while using GLUT often takes only a few lines of code and requires no knowledge of operating system specific windowing programming interfaces.

The overall goal of MPIglut is to make powerwall programming easier. MPIglut is a library that can turn a serial GLUT program into a parallel powerwall-capable program with a simple recompile.

MPIglut implements the Graphics Library User Toolkit standard with a few modifications that allow it to run in parallel on each machine in the visualization cluster. MPIglut runs a complete copy of the user program on each one of the nodes. The allows the predistribution of all of the files and information necessary for each display. In fact, as far as each display knows, it is the only display on the wall. This might appear to be bad,

because you would think that each display is trying to render the entire dataset when in fact, this isn't the case. Each display knows which part of the overall global coordinate system falls into its display area and only calculates what is needed for its display.

The method that MPIglut uses is to 'override' the standard GLUT calls using simple preprocessor macros. For example, the macro to override the GLUT call glutInit() would be `#define glutInit mpiglutInit.` In this way, you can turn on and off MPIglut calls in the actual program if you should desire to do so. To add functionality to MPIglut, all that is needed is to create the function and then override the GLUT call is this manner.

2.2.1 Parallel Programming with MPI

To enable communication between backend processes, MPIglut uses the Message Passing Interface (MPI) standard. Specifically the current implementation uses MPICH version 1. [MPICH], any implementation of MPI should work. The user is not required to make any calls to directly to MPI themselves. All of this is handled by MPIglut transparently. The user, however, is not precluded from making MPI calls. In fact, it can be useful to ensure that certain things only happen on one specific node or for getting specific information about the current node for something like benchmarking.

2.2.2 Startup of an MPIglut program

On the prototype powerwall, all machines share data in the form of an NFS mounted drive. When an MPIglut program starts, all of the machines have access to the same executable and data, so the program doesn't have to go through any great lengths to get the data into memory. An MPIglut program starts as any other GLUT program. The user runs the program on the master node and execution continues as normal until the program reaches the GLUT call glutInit() which is what normally initializes the GL system. An MPIglut program is also an MPI program, and as such it starts how MPI programs start. In the case of MPIglut, glutInit() is overridden by the MPIglutInit() function. This function initializes the MPIglut backend or master server as well as starting the same executable on each of the other “worker” machines. Each of these machines runs the executable up until it reaches glutInit. Since these are slave “worker” nodes, they don't initialize MPI communications, but they do announce themselves to the MPIglut master process. When all cluster nodes have reported in, then they all proceed in unison through the rest of the initialization, including the regular glutInit routine. Each node is given its particular 'view' of the world during this initialization.

2.2.3 Dataflow with GLUT and MPIglut

As can be seen in Figure 1, Dataflow with GLUT and MPIglut are very similar. They both receive input from the user through the standard windowing system. The windowing system sends that information to the running program. This program deals with the information using user callback routines that determine what to do with a mouse click, a screen resize or any of the other possible user events. The difference is how this information is communicated.

In a serial GLUT program, all of this information stays on one machine and never leaves. MPIglut is different in that it has to communicate all of this information to every other node in the cluster. The DMX server forks the user input data as well as the standard window events into two streams at this point. One stream heads to each of the nodes in the cluster. This is how each node knows how to resize the screen or to display the mouse menu from the window manager. This is also how each screen knows where the mouse pointer is and what they have to display. DMX handles all of the window events including the display of the windows borders themselves and the widgets for each window. The other stream goes to the MPIglut frontend process to be passed to the MPIglut backend process. MPIglut accomplishes this by using sockets for communication between the MPIglut frontend and the MPIglut backend 'Boss' node. Using sockets here ensures that if it is necessary, the MPIglut frontend and backend processes don't have to be on the same machine. For speed sake, however, at this time they do run on the same machine. The backend MPIglut process takes the input stream from the window server and sends it to each of the backend MPIglut processes. Each one of these machines then follows the same pattern as serial GLUT to process the data. The correct user callback routines are called, the data is calculated and the screen refreshed with the newest information.

To ensure a smooth framerate across all machines, MPIglut inserts an MPIbarrier call in the glutSwapBuffers command. This assures us that all of the machines with display a new frame at the same time. It can be disorienting when one or two machines are delayed because they are more heavily burdened than other machines. Many of the calls used by MPIglut are 'collective' in this same manner. They must happen in the same order on every machine so that the machines stay in sync with each other. In particular, all user interface events, rendering and many GLUT calls are collective. [ORION] In the case of MPIglut, the programmer is safe to think of each process acting lock-step with each other. This simplifies programming because the programmer doesn't have to be concerned with dangerous race conditions common to asynchronous parallel programming.

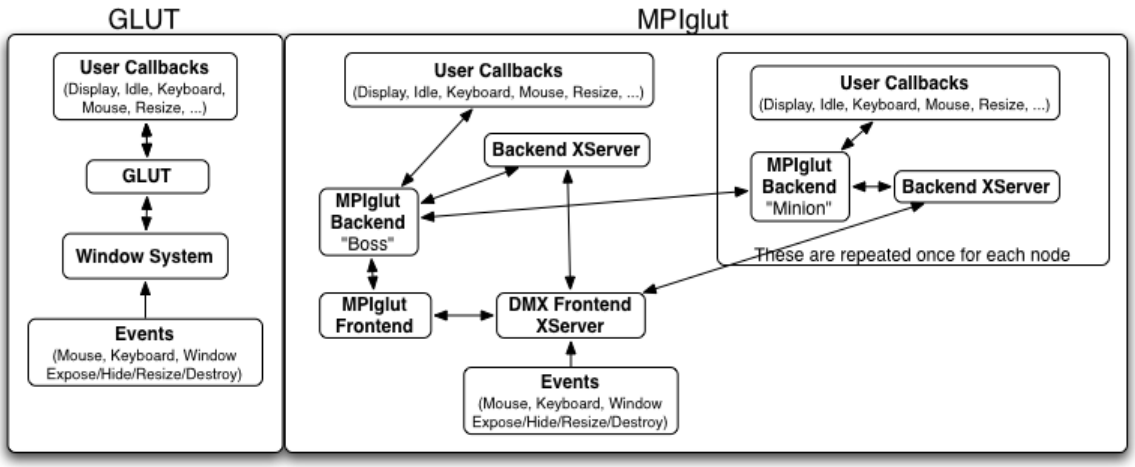


Figure 2: Sequential GLUT normally receives events from the X server and forwards them the the applications event handlers. MPIglut receives events at the frontend and sends them out to all of the backends. They are delivered to the applications event handlers collectively.

2.3 SOAR

As a sample application and in keeping with the goal of having a working terrain generator for the powerwall the Stateless Adaptive One-pass Refinement (SOAR) code was used to test MPIglut. SOAR uses an adaptive quadtree design which improves performance by culling out data that either can't be seen, or is to small to make a difference.

2.3.1 Multi-threading

This code is currently single threaded, but it could be made multi threaded for a possible boost in performance. The two most obvious tasks for a multi threaded application would be for one thread to handle updating the terrain and for the other to update the display. These two activities could happen in parallel without to much fear of one causing problems for the other. For serial GLUT or with a powerwall that has one machine for one screen, multi threading could help greatly, but because of the way MPIglut creates a process for each screen in the powerwall, we wouldn't see much of a benefit because we have two screens attached to each machine.

2.3.2 Input

SOAR takes two PNG files as input. One of them is the actual height data and the other is the texture that will be painted onto the generated terrain. It is not essential, but these two files should both be powers of two in height and width and should be the same size.

The terrain file is not required. The terrain data will be converted from PNG format into the format that SOAR uses internally to represent the terrain. This is the quad tree. The quadtree breaks the image recursively into quarters. You can specify the final dimensions of the quadtree when it is created. For the purposes of this project, the 4096x4096 pixel PNG image was broken into a 10,000 x10,000 quadtree format. You can also specify the vertical exaggeration. For this example, 100 literally made a mountain out of a mole hill, 10 produced a terrain that was super flat, but 50 produced something that looked realistic. Figure ###### shows small versions of the terrain input files that were used.

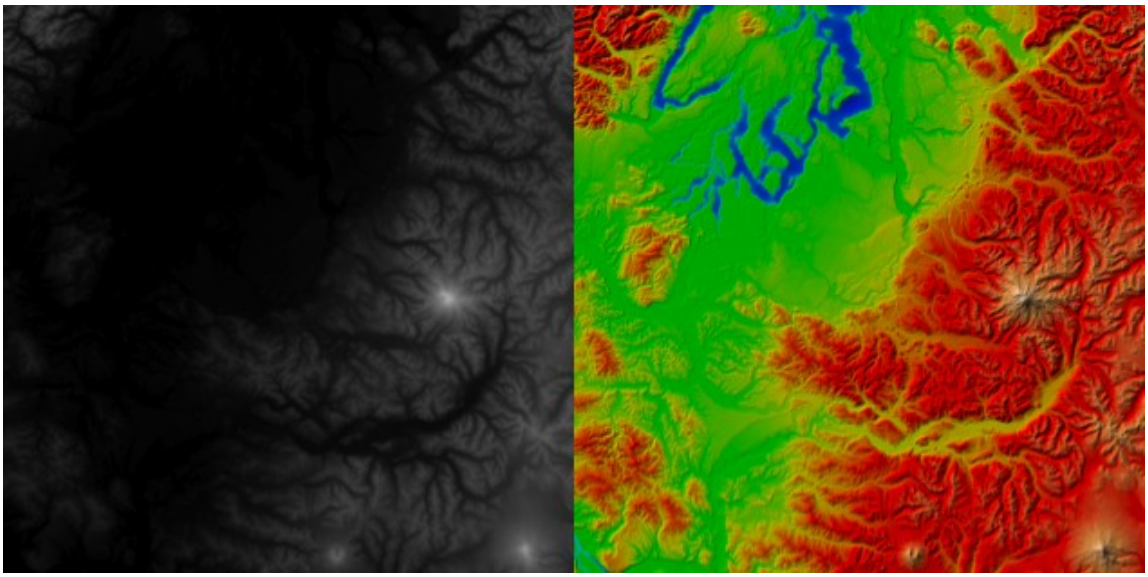


Figure 3: On the left is the height file. Higher altitudes correspond to brighter whites. On the left is the texture file.

2.3.3 Adaptive Quadtree

The goal of an adaptive quad-tree is to allow branches of the terrain to be culled based on whether or not they can be seen. This is a little more complex than just whether or not they are behind the viewer. When the tree is built, it calculated how much difference there is between the actual value of a point and where the lines that would have ran to that point would be if it were not included. If this calculated difference exceeds a number that the user can define when the program runs, then that part of the tree is not shown. This number is compared to a calculation that takes into account how far a person is from a particular part of the terrain and what is between the user and the terrain. In this way, data that is hidden behind a mountain, for example, will not be displayed. Neither will data that makes a small difference on the screen because the viewer is so far away. Using this technique, the displayed triangles can be cut by up to 90% without losing terrain detail in the viewing window. Figure #### shows what the quadtree looks like for a section of the SOAR terrain data. You can see that the flat area at the bottom of the picture doesn't have many triangles, but the data in the middle has a much more extreme

change in height, so it includes many triangles. Todays video cards are optimized to display triangles, which is why the image is broken into triangles. Also, using triangles helps prevent gaps or holes from forming when the terrain is displayed.

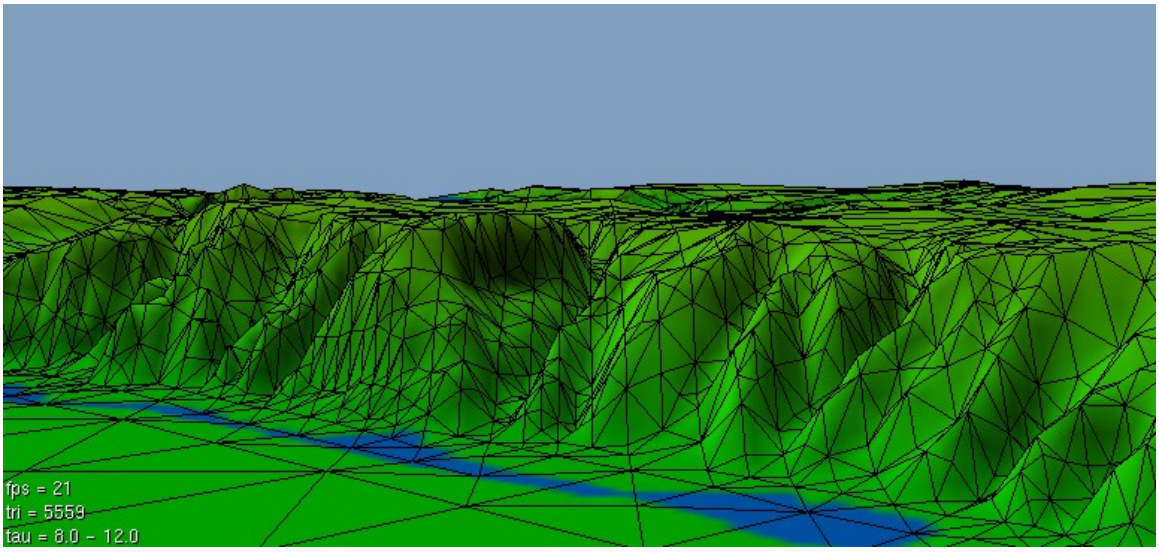


Figure 4: SOAR Terrain broken into Quad trees. Many triangles on the steep parts and few on the flat.

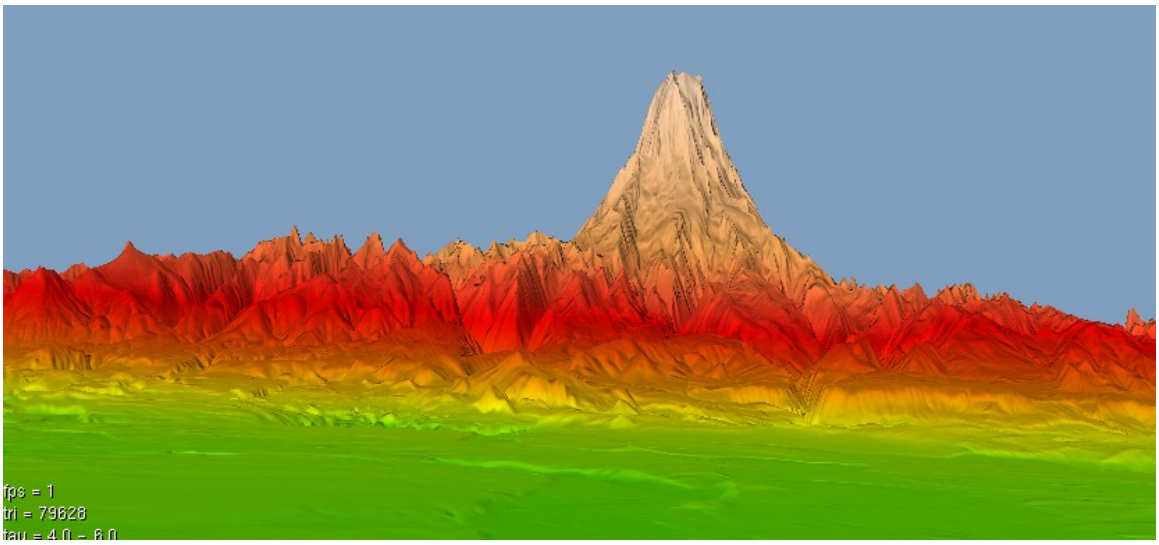


Figure 5: A sample image from SOAR depicting a mountain in profile.

3 Performance and Scalability

3.1 Benchmarks

The benchmark data was gathered by modifying the base SOAR code to collect and store CPU information, network data and memory data gathered from the operating system. Also several variables were cleared and used as a starting point for the statistics such as total frames displayed, start time and total triangles displayed. To ensure that the benchmark results would be the most accurate possible, SOAR was also set up to run a simple course or 'camera path' around the terrain when the program started. This allowed the statistics to be gathered in an automated and consistent fashion.

For comparative purposes, benchmark data was gathered for the same program running with MPIglut, DMX with GLX/Proxy and Chromium. The data collection was started at frame 200 to bypass the startup jitters that were noticed when the program first started running. The program ran through 500 frames before it gathered final statistics, wrote the data to a file and exited. For each run of the trial program, DMX was reset to run at a different size to get information on how well the three solutions scaled to larger and larger powerwalls. It was discovered in the beginning that the method of gathering statistics at the beginning of the run and then gathering them again at the end of the run was not sufficient for the network transfer numbers. The internal structure of the network information gathering program uses a 32 bit integer to store the amount received and sent over the network. For Chromium and DMX, the data transfer overflowed the 32 but integer pretty quickly. To remedy the problem, data for the network was gathered every 5 seconds and stored in a global variable. The effectively made it possible to get accurate network transfer data. The graphs for frames per second, triangles per second and Bytes per frame are contained below in figures ####, #### and ###.

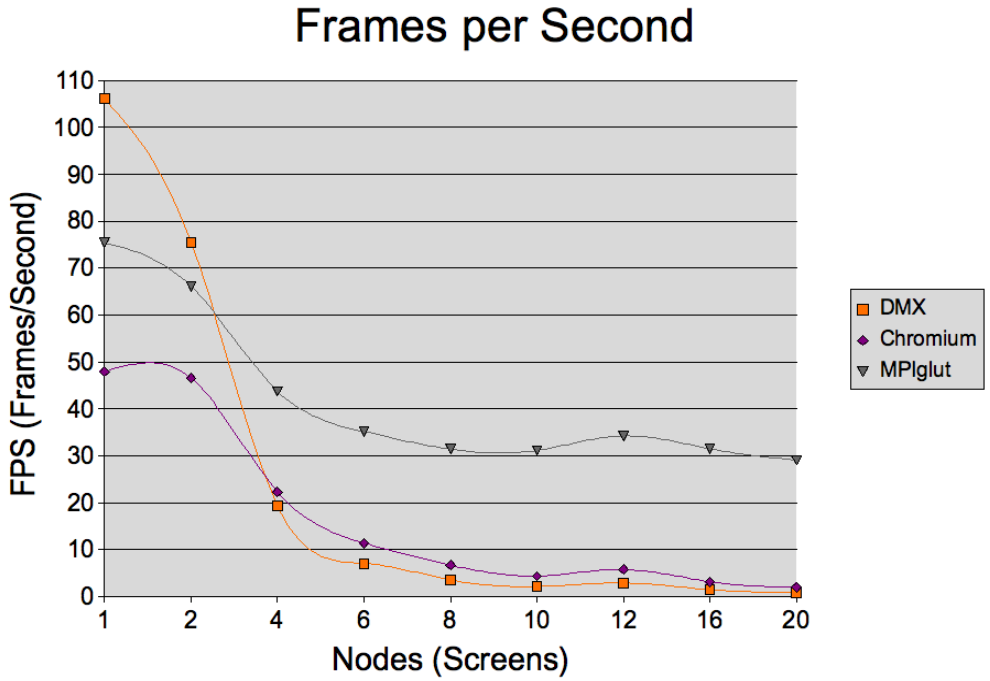


Figure 6: All do very well up to two screens. Chromium and DMX fall of rapidly while MPIglut levels off at about 30 frames per second.

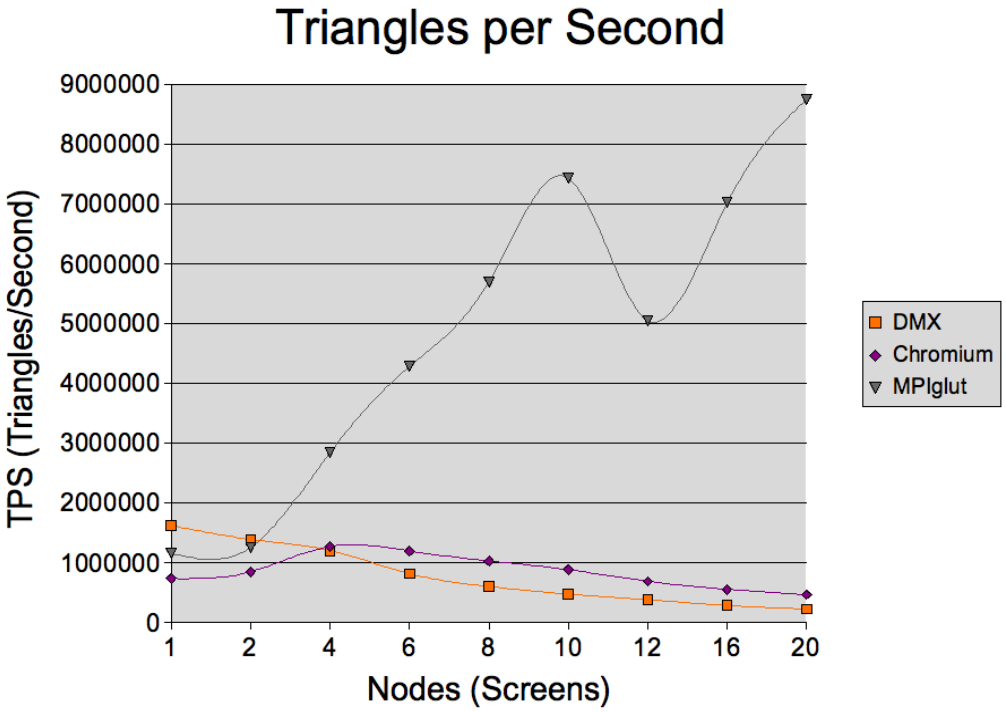


Figure 7: Chromium and DMX start low and end even lower. MPIglut grows continuously except for the aspect ratio change induced dip at 12 screens.

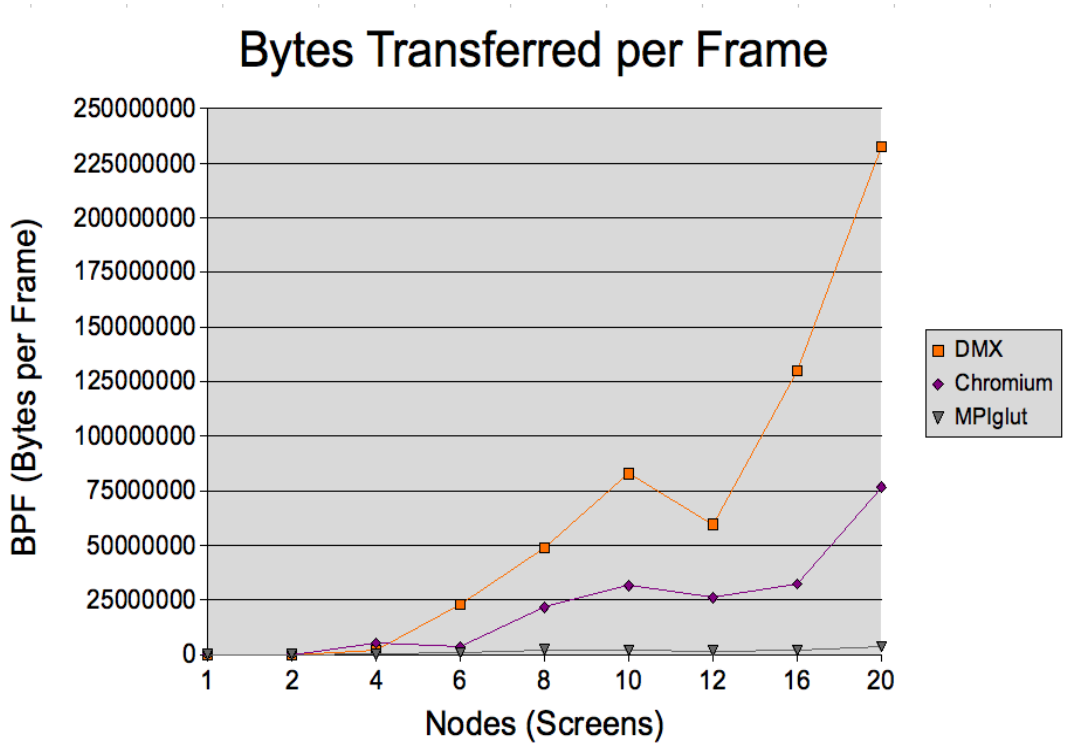


Figure 8: Bytes of Network Transfer per Frame of Output – Network transfer grows very quickly for Chromium and DMX. MPIglut grows much more slowly which means it will scale to much larger display sizes and still be very responsive.

As you can see from Figure ####, all three methods had a strong showing when there were only two screens. This is because there is no network transfer with only two screens because they share the same machine. DMX makes a particularly good showing here as it has very little overhead. Both DMX and Chromium fall quickly to very low frames per second scores. Each one of them has to calculate the entire powerwalls worth of data on one node and then send out everything each screen needs to display. Chromium is slightly more efficient than DMX in this regard, but it isn't enough to give it a strong showing. Again, this is because Chromium is smarter about sending the OpenGL calls only to the machines that need them, instead of sending the same stream to all machines in the powerwall.

MPIglut also loses performance as the size of the display scales up, but it levels off at about 30 frames per second and holds pretty steady there. The numbers for triangles per second follow exactly what you would expect them to do. With an even number of frames per second, but ever increasing screen size, there should be more triangles for MPIglut and less for Chromium and DMX. MPIglut follows a steady increase that is linear. The only anomaly is the huge dip in triangles per second at 12 screens. This is also reflected in a bump in frames per second for all programs. This is due to the way the

benchmarks work. It was mentioned that the size of the DMX virtual screen changed with each iteration of the benchmark. At 12 screens the aspect ratio of the window changed drastically from being a short but wide display to being more or less square. This changed what geometry was 'viewable', which led to a huge drop in triangles necessary to display a single screen. If the display wall were 1000 screens by 1000 screens, then the difference between one layout and another wouldn't be as drastic, because we could always keep near the same aspect ratio. As it is, this anomaly is easily explainable, so it's not a worry.

Overall, MPIglut performed very well. There is still room for improvements and optimizations, but for a first round, it is exceptional. The prototype powerwall was only 20 screens, so it is difficult to estimate how far how far the apparent linear speedup will continue. Logic says that it can't continue forever, but since the network will be the first bottleneck, the speedup should continue until the network connection is saturated. The screen count should be able to double before framerates start to dip below 30 frames per second due to traffic. This depends on how well MPI broadcasts scale to a larger number of screens.

4 Limitations of MPIglut

Currently MPIglut is not cross platform. It only works for Xwindows running DMX. Currently, DMX is not available for OSX, so DMX is limited to Linux. Also the changes required for GLUT don't compile under OSX. This problem is easier to fix than the lack of DMX for OSX.

MPIglut as yet doesn't support all GLUT calls. For example, glutFullScreen doesn't change to fullscreen correctly and GLUT calls that work with the framebuffers (screen grabbers, and mouse picking, for example) don't work.

MPIglut needs to perform some sort of load balancing. In the SOAR benchmarks, many times entire screens were covered with blue sky. These screens took next to nothing to compute, so these nodes should have been taking some of the burden off of the screens that were full of detailed geometry. This would be easy to implement for the two screens that share a single CPU/Graphics card. It would be harder, and possible less useful, to implement this across all machines.

All of these problems can be overcome with continued development of MPIglut.

5 Lessons Learned

The first lesson learned was to not allow system updates midstream. There was a problem with the nVidia drivers and DMX which surfaced after a complete system update occurred. Of course this happened right before an important deadline, which made the problem that much more annoying. Luckily everything was fixed pretty quickly. Or so we thought... At about the same time as the complete system update, autoupdates were turned on. It is not known whether the autoupdates were turned on my a person or when the system update occurred. This made the operating system we were using reinstall the newer, but broken drivers every time the system restarted.

Lesson number two was not to give root access to people that don't understand lesson one. You are just asking for trouble.

Lesson number three was to watch out for bogus benchmark data. It can indicate a larger problem with the system. Say, for example, the problems caused when someone updates the system when it was working just fine. (See lesson number one.)

6 Conclusions and Future Work

The answer to the question that was asked in the beginning of this paper is unequivocally yes, it is possible to port a serial GLUT application to the powerwall and have good scalability and performance. With MPIglut, the SOAR terrain generator runs in parallel on all ten machines and twenty screens. Each screen displays data lockstep with every other machine. The terrain displays with a near constant framerate on nearly all sizes of the powerwall. Based on the performance data, MPIglut should scale well beyond what would be a practical size for a powerwall.

For this project, MPIglut was shown to work well with one particular terrain rendering program. In fact, this program was an ideal example to use with MPIglut in that it performed object-space clipping to reduce the amount of work done by any particular node. In testing, MPIglut also performed well with a variety of other GLUT programs. It should perform well for any program that allows the calculations necessary to draw any particular screen to be performed on the machine that is attached to that screen. In other words, MPIglut will lose performance if each of the machines has to calculate what to display on the entire powerwall.

With that said, there is still a lot of room for MPIglut to improve and grow. It could be extended to work with a parallelism in a single machine to 'stripe' the display and have a separate MPIglut process control each of the stripes.

It could also be extended to allow greater load balancing. During the benchmarks, it was clear that some machines were bored silly with very little to do and others were working their guts out. It would be easy to 'shift' which part of a screen each process handled so that at least on one dual screen node, the processors are balanced in CPU usage. The same idea could be extended to multiple machines to allow load balancing for the entire wall.

MPIglut could also be extended to allow the framebuffer routines to work as expected. Currently, if you had GLUT code that would take a screen shot, each node would take a shot of just what it could see. A method to allow the entire screen to be captured is possible with a little bit of work.

VRJuggler is good at displaying data on non-planar, 3D surfaces. MPIglut could also be made to do this with a change in the simple 2D coordinate representation to a more 3D representation. This may not be all that easy, but it would allow MPIglut to be used in Cave environments and in the inside of domes.

Finally, MPIglut could be made to be more cross platform. It is specific to Linux right now, which, for the time being, is okay because most powerwalls run Linux. OSX would be the next logical porting choice. Windows would be an entirely different animal to tame.

This has been a fun and useful project. (If this ends up on a page by itself in the final version, I'm going to cut it.)

7 References

List of references here. To be filled in and once we have the numbers, search and replace the placeholder text to point to the correct place. Would also be nice to find where I missed references and point to them also. Wouldn't want to plagiarize.

GLUT
MPIglut
SOAR
Chromium
DMX
VRJuggler
OpenSG
OpenSceneGraph

Appendix A - MPIglut How-To

Haven't written this yet. Any requests? Things I should definitely throw in? Directions I should head?

I don't intend to make a tutorial on how to create a powerwall. Just how to download, 'install' and use MPIglut. I will note what other software has to be installed for it to work correctly. (DMX and MPI) Anything else?

Appendix B – System Stats

7 - Dell Precision 390 Workstations
- Intel Core2 Duo E6300 CPUs
- nVidia QuadroFX 3450 dual-DVI PCI Express graphics cards
- 2 Gigabytes RAM

3 - Dell Precision 380 Workstations
- Pentium D CPU
- nVidia QuadroFX 1400 graphics cards.
- 2 Gigabytes RAM

1 - Dell PowerConnect 2724 24-port gigabit ethernet switch.

20 - Dell UltraSharp 2007FPW 20.1-inch DVI Flat Panel LCD
- 1680x1050 resolution

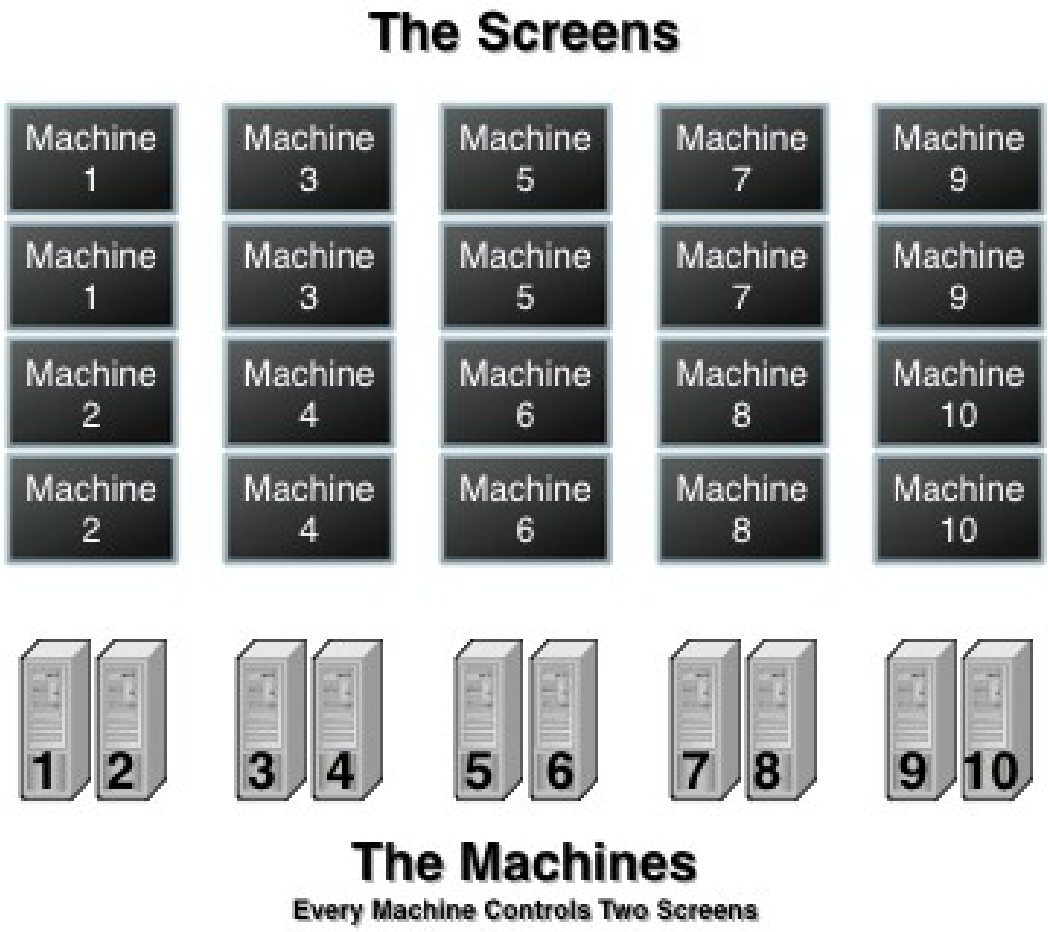
Overall Dimensions
- 9000x4650 pixel resolution including gaps between monitors
- 8400x4200 pixels displayed.
- 35.28 million displayed pixels.
- 93 inches wide
- 52 inches high
- 106 inch diagonal

- Including the frame, it's exactly eight feet wide and just over six feet high.
- Approximately power consumption: 7.2 kilowatts.
- Approximate gross weight: 1/2 ton.

Appendix C – Software

Ubuntu 6.06 and Ubuntu 6.10 (See lessons learned.)
XOrg 7.0.0
DMX version 7.0.1
SOAR Version 1.11
MPIglut Version 0.80
nVidia Driver Version 8762

Appendix D – System Layout



Appendix E – Performance Data

FPS	DMX	Chromium	MPIglut	
	1	106.16	47.89	75.41
	2	75.53	46.51	66.14
	4	19.31	22.26	43.74
	6	7.03	11.37	35.12
	8	3.5	6.63	31.38
	10	2.11	4.33	31.12
	12	2.86	5.71	34.21
	16	1.42	3.09	31.45
	20	0.85	1.94	29.05

TPS	DMX	Chromium	MPIglut	
	1	1623557.11	732466.86	1153386.43
	2	1387359.82	843626.23	1254470.11
	4	1200152.3	1271461.75	2846829.22
	6	815111.28	1194115.42	4279904.72
	8	599827.77	1027811.74	5697303.75
	10	473598.29	881338.43	7429781.82
	12	379857.11	686819.18	5049514.06
	16	282674.69	554942.99	7028001.76
	20	224389.45	464352.95	8744474.97

BPF	DMX	Chromium	MPIglut	
	1	0	0	0
	2	0	12.8	28.45
	4	1843030.2	5452740.8	8998.44
	6	22920156	3564114.8	82931.67
	8	48933008.8	21608713.6	210775.68
	10	82861311	31525220	99502.75
	12	59526524.8	26036499.2	411231.06
	16	129788768.8	32448382.2	682952.55
	20	232600074.6	76619499.2	1806134.18

Appendix F – Code

Where to get SOAR
Where to get MPIglut (maybe include since it's short)
Where to get DMX
Modified main.c for SOAR
Code for benchmarks
Bash scripts

Will attach ALL code to the back of this, but won't paste it into this program, unless otherwise directed.