

# Retained Mode Parallel Rendering for Scalable Tiled Displays

Tom van der Schaaf, Luc Renambot, Desmond Germans, Hans Spoelder, Henri Bal  
{tom,renambot,bal}@cs.vu.nl - {desmond,hs}@nat.vu.nl  
Faculty of Sciences - Vrije Universiteit  
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

## Abstract

*Most research in parallel rendering has focused on widely used immediate mode interfaces (mainly OpenGL) or on retained mode interfaces designed for parallel rendering only. In this paper, we describe parallel implementations of the Aura API which provides a multi-platform graphics interface using a retained mode paradigm. Using several test applications in the context of a tiled display infrastructure, we show that a retained approach performs much better than existing implementations.*

## 1. Introduction

The current trend in hardware for parallel graphics is to use clusters of off-the-shelf PCs instead of high-end super computers. This trend has emerged since the dramatic change in the price/performance ratio of today's PCs. Using large, high-resolution displays is another trend that is currently emerging. High resolution allows for detailed scientific visualization, and overcomes the limited screen resolution of standard monitors. Large displays also have applications in teaching environments, where multiple people (small groups or full classroom) are looking at a single large screen.

The goal of this project is to investigate the field of parallel rendering in the context of scalable tiled displays. The target system for this parallel renderer is a so-called tiled display, a high-resolution screen. It consists of several projectors that compose a single, seamless image onto the screen. For this project, a cluster of off-the-shelf PCs with 3D gaming video hardware performs the rendering. These PCs control the projectors, and calculate the image in parallel. This work is based on the Aura API developed at the Vrije Universiteit in Amsterdam. Aura is designed as a portable 3D graphics retained-mode layer for scientific visualization in virtual reality [4].

The contributions of this paper are:

- It presents a retained-mode API for distributed rendering of 3D graphics on a tiled display, with a portable implementation across a variety of VR platforms (several operating systems and hardware environments),
- Two different implementations of the API are presented, with different trade-offs in scalability and performance,
- It presents a preliminary performance study of network and graphics requirements of different benchmark applications.

The rest of the paper is organized as follows. Section 2 discusses briefly the work that has been done already in the field of rendering and parallel rendering. Next, Section 3 describes the Aura graphics interface and two parallel renderers that implement that interface. Using several benchmarks, we compare our implementations to WireGL in Section 4, showing that retained mode allows for better scalability for certain types of applications.

## 2. Related Work

Parallel rendering adds processors and graphics resources to achieve higher performances. In 1994, Molnar et al described a sorting classification for parallel rendering [10]. Three different types of sorting are proposed, based on where in the rendering pipeline sorting takes place. The three modes are sort-first, sort-middle, and sort-last. However, adding computing power and graphic resources is generally used in two ways. The first option is to distribute all 3D objects over all processors, and then each object is rendered by one processor. The alternative is to split the screen into a number of tiles and to let each processor render all objects that have to be displayed onto that tile. This approach usually causes some objects to be rendered by multiple processors, because those objects intersect multiple tiles.

Much work has been devoted to the field of parallel rendering. Probably the first step towards the parallel renderer was the creation of GLX [7] for X windows in 1992.

Although not designed for efficient parallel rendering, many of its design principles are used in more efficiency-oriented renderers. GLX was designed to enable remote OpenGL graphics. It allows an OpenGL program to run on a server and to display the results on a display physically connected to another computer, the client. Client and server are connected through a network and use the GLX protocol to send the OpenGL command streams. GLX also allows parallel rendering, but the emphasis lies with remote displays.

Based on GLX, the Stanford University Computer Graphics Lab has designed a better alternative called WireGL [1, 5, 6]. It replaces the standard OpenGL dynamically-linked library on the client by its own library. This new library, instead of executing the OpenGL commands, packs the commands in buffers and sends them to the appropriate rendering servers. There, the commands are executed normally (save for some frustum adjustments). The main difference between Aura and WireGL is that WireGL is an immediate-mode interface and Aura is a retained-mode one. The latter opens possibilities for improved performance and scalability on a cluster. The parallel renderers presented in this paper are compared to WireGL (note that the WireGL project continues under the name 'Chromium').

The NASA Institute for Computer Applications in Science and Engineering (ICASE) has developed a parallel renderer for distributed memory computers (such as Intel's Paragon) called PGL (Parallel Graphics Library)[2]. It is a software renderer which was designed for research purposes and does not have all the features that, for example, WireGL and Aura have. PGL is a retained mode renderer that uses multiple processors to render polygons and add them to the final image. The target display for PGL is a regular display, not the high-resolution type of display that Aura and WireGL have been designed for. The University of New York has designed another parallel renderer like PGL: the Parallel Mesa Library[9]. The target hardware is the same: distributed memory computers using message passing. The main difference with PGL is that the Parallel Mesa Library implements the OpenGL interface, to support existing programs.

In the field of high-end computer graphics, much research has been conducted on the Power Wall [11] and Infinity Wall [3] projects. However, the utilization of high-end graphics supercomputers is of a completely different nature and therefore beyond the scope of this paper.

A project similar to the one presented in this paper is the Distributed Graphics Database (DGD) by Ben Schaeffer of the University of Illinois [13]. DGD provides the user with a scene graph interface and hides all parallel rendering related issues underneath it. It defines function calls to add, change, and remove records from a remote database, which are low-level operations. Aura provides the user with a full graphics

API, which is much cleaner, easier to use and also exists as a non-parallel renderer (note that the DGD project has been renamed to 'Syzygy').

Another similar project exists at Princeton University [8]. It describes four different systems to do parallel rendering: "GL-DLL Replacement" implements the OpenGL API in a fashion comparable to WireGL, although not as optimized. System-level Synchronized Execution (SSE) and Application-level Synchronized Execution (ASE) are comparable to the Aura 'Multiple Copies' renderer described in Section 3. The fourth renderer is not really a renderer but a virtual display driver (VDD) and is used for 2D (desktop) rendering. Four different OpenGL programs are used as benchmarks to test the systems, but without a scalability study.

### 3. Parallel Rendering

In this section, we will first describe the setup of our system to form the background of all that follows. Next, we will describe the Aura API, followed by the description of the two parallel renderers that implement the Aura interface.

#### 3.1. Overview

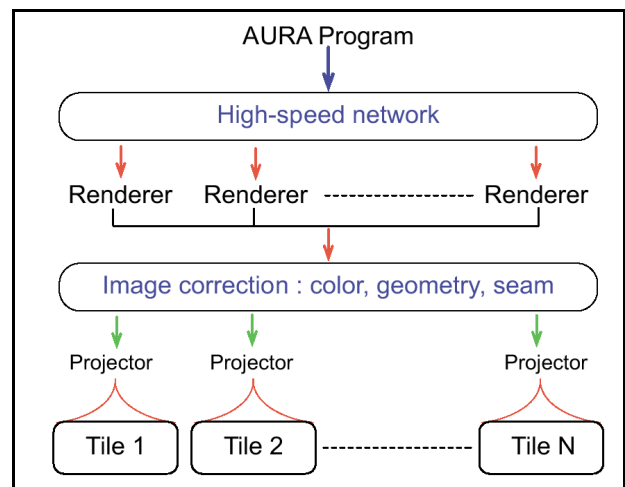


Figure 1. Parallel rendering for tiled display

Figure 1 presents an overview of how an application is executed on a tiled display infrastructure. The application (i.e. the program that uses the parallel renderer) runs on a single computer: the client. A fast network (e.g. Myrinet) connects the client to a cluster of computers: the servers or renderers. As shown by our experimental tests in Section 4, a regular network (switched 100Mbits Ethernet) is sufficient for a given class of application. The servers are connected to

a set of projectors; each server controls one projector. Every projector is responsible for 'drawing' a rectangular region of the screen (a projector-tile). Each server is responsible for assembling the final image of its projector and correcting the image for a seamless integration with its neighbors, and finally sending the image to the projector. The rendering of the image is done in parallel on all servers. The software layer correcting the images for a seamless projection is beyond the scope of this paper and consequently not addressed here.

The following subsections contain the main implementation issues that have to be considered when building a parallel renderer.

### 3.2. The Application Programming Interface (API)

The interface of the graphics library is a critical issue. Ideally the interface is easy to use and flexible, allowing the programmer to use every feature of the underlying hardware efficiently. When making a choice for an interface, two options are possible: a new interface, designed specifically for the purpose of a tiled display, or an existing interface. The latter has the advantage that existing applications, for that API, can run on the tiled display without any changes to the source code. In fact, when using dynamic libraries, no access to source code is even necessary. The first option requires rewriting of all applications, to be used for the new API. Writing a new interface is, of course, much work, but allows the interface to be optimized for the target environment.

Initial tests with WireGL (an implementation of the OpenGL API) revealed a scalability problem. In our opinion, the core of this problem lies with the immediate mode interface. It is not suitable for parallel rendering due to the bandwidth requirements of sending the data (vertex, normal, color, texture coordinates, matrices, etc) over the network. Retained mode interfaces on the other hand, due to the higher level of abstraction, can use information that is not available for an immediate mode renderer to minimize communication. Therefore, we decided to port our own graphics API called Aura to a distributed architecture.

### 3.3. Aura

The Aura API was designed by Desmond Germans [4] as a platform-independent graphics API. It runs on several platforms and operating systems, such as IRIX, Linux, and Windows. It offers the user a C++ scene graph interface and takes care of window management. This scene graph interface makes that Aura is a retained mode renderer. However, Aura allows the user to apply modifications to any object in the graph. For instance, the user can change the vertices of geometric objects in the graph at any time. Therefore,

Aura can also be used to write applications showing an immediate mode behavior. For a complete description of Aura functionalities and some case study applications, refer to [4].

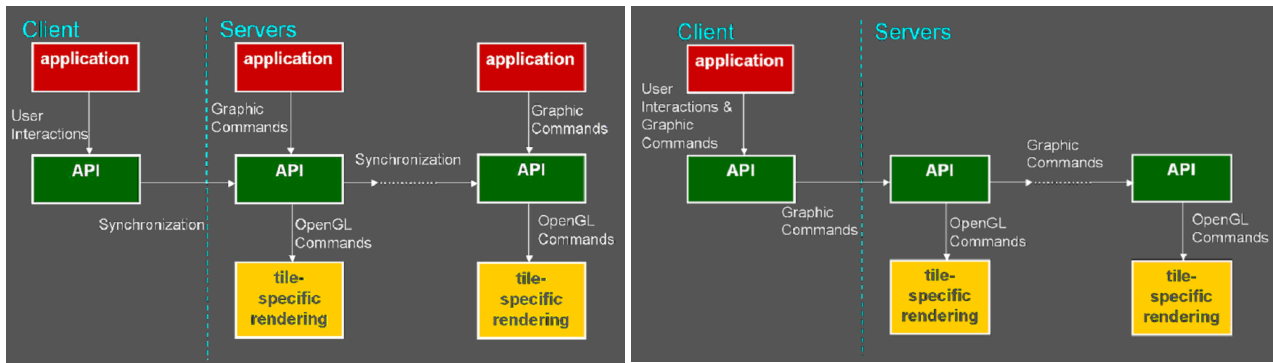
### 3.4. Parallel Aura

Two parallel versions of Aura have been designed:

- Aura 'Multiple Copies': an implementation of the replicated application strategy, described in the next subsection. It is expected to be the fastest on all tests, since little data is sent over the network. However, this approach exhibits several problems. Among them is the guaranty of synchronized executions of the different programs, and in particular system calls, random functions, and remote I/O.
- Aura 'Broadcast' was designed to solve this I/O and communication with external processes problem. Basically, it uses the same strategy as WireGL does, but for a retained mode interface. The consequence is that its performance for objects that are more or less static during execution is much better than for WireGL. For scenes with dynamic objects, performance should be similar to WireGL. In our current implementation, the entire scene graph is replicated on all servers and on the client. For sorting, we use the same approach as WireGL: per-frame sort-first. The client calculates by means of bounding-boxes which object should be sent to which server. Each server renders only those objects that fall into the tile of its associated projector. The latter means that the workload can be unbalanced if the scene is not spread equally over the screen. This could be solved by applying a hybrid sort-first/sort-last algorithm [12]. This would, however, increase bandwidth usage and would complicate matters. Therefore, our initial focus is on the simple sort-first approach. In the future, different approaches will be investigated.

#### 3.4.1 The Multiple Copies and Broadcast approaches

The two Aura renderers use different communication strategies for parallelization. The Multiple Copies approach replicates the application to all servers. Only frustum adjustments are made (see Figure 2(a)). These adjustments are necessary to let every server draw only a sub-frustum to form a correct viewing volume. To be able to interact with the application (keyboard, mouse, tracker, etc) and to perform some I/O operations, the client must broadcast a message that informs the other applications of the event. This technique assures that every server performs exactly the same actions. In addition, to prevent temporally misaligned images, the frames must be synchronized. The



(a) Multiple Copies

(b) Broadcast

**Figure 2. Multiple Copies approach versus Broadcast Approach**

second approach (Broadcast) runs the application only on the client. Then, all graphics commands from the API are broadcast to all servers, instead of being executed locally (see Figure 2(b)). Again, each server manages a sub-frustum. Synchronization between frames is also needed. Although this approach is called Broadcast, it is not mandatory to use a broadcast communication scheme. Some objects could be sent only to specific servers, which eventually draws it onto the screen. By clever use of bounding boxes, broadcast, and normal point-to-point communication, it is possible to reduce the required bandwidth.

The main advantage of the Multiple Copies approach lies on the performance aspect. Since there is hardly any communication necessary (only interaction and synchronization) it is not limited by the network performance. However, there is much redundancy; all servers perform the same actions. The scene graph is replicated on all servers. Multiple copies approach might look like the ideal solution, but it has two main drawbacks. The first is the fact that I/O can be a problem. If the application reads data from a file, a database, or a running simulation program, all servers require that information. Consequently all servers must read that file or connect to the simulation or database. It is clear these multiple accesses cause a scalability issue. A second problem is that this approach is not transparent. The application must use functions provided by the API for handling user interactions (mouse, keyboard, trackers) so that these events can be transferred to the servers. Also, for other functions that require synchronization, like system calls or random number generation, the programmer must use functions provided by the API instead of the standard ones. Lack of transparency, albeit very little, makes this approach unsuitable for existing APIs. Broadcast has the advantage that all cluster-related issues can be hidden in

the implementation of the interface. This complete transparency enables the implementation of an existing API. Furthermore, I/O is straightforward for the Broadcast approach. Only one application is running; only one application reads files or databases. To the outside world, the application now behaves like an ordinary application on a single node. The disadvantage is performance. Because this approach requires graphics commands to be sent over the network, it potentially requires a high bandwidth which becomes the limiting factor. High-speed networks such as Myrinet or Gigabit Ethernet solve partially this problem.

Besides these two general approaches, there is also a lot to be developed on a lower level. Especially the broadcast approach requires a carefully designed communication protocol that minimizes communication overhead. Several strategies can be used for this, such as:

- Fast compression and decompression of messages to reduce bandwidth,
- Collapsing multiple packets into one, to prevent latency problems,
- Preventing the sending of data to servers that do not need that data for displaying their part of the scene.
- Dynamic selection of function shipping and data shipping over the network.

However, these techniques are out of the scope of this study, and so not addressed.

### 3.4.2 Implementation

Both Aura Multiple Copies and Aura Broadcast use the standard message passing interface MPI for communication. OpenGL is used for the actual rendering (although

Direct3D is also supported on Windows systems). This means that it is even possible to run Aura applications using WireGL.

We applied several modifications to the sequential Aura implementation for Broadcast and Multiple Copies versions, while maintaining the same API:

**Aura Multiple Copies** The implementation is rather simple. All servers run a copy of the application. The clients synchronize before the frame swap to minimize image tearing (ie. frame lock). Also, whenever the user presses a key or moves the mouse, this interaction is recorded, broadcast, and then evaluated by each server. To compose the global picture, each server renders only a partial image using a reduced frustum.

**Aura Broadcast** As mentioned before, Aura Broadcast uses a similar strategy than WireGL. Whenever something changes in the scene graph, it is sent to the appropriate servers (i.e. those that are affected by the change). These changes can be divided into two groups:

- **Scene Graph operations:** There are separate messages sent to create nodes, to add nodes to the scene graph, and to remove nodes from the scene graph. These types of operations are broadcast to every server to maintain the graph consistent on every node.
- **Object Modifications:** Whenever the user-program modifies an Aura object, the client marks it as dirty. Before the rendering loop, it transmits a message containing the new data for each dirty object. This is also true for newly created objects, since objects are initially dirty (transmitting initial values is not part of the scene graph operations). Moreover, each type of object has its own set of message types and several state bits. This way not all the data of an object is transmitted when something changes. For instance, a dirty bit per vertex is maintained in vertex buffers and then, only dirty vertices are transmitted. The same approach is used for other parameters (normal, color, etc). After sending scene changes, the client synchronizes all servers before the frame swap. Note that the key idea is that nothing is sent if an object is not modified.

A dynamic buffer management is implemented over MPI to pack the data needed to be sent to remote servers. Basically, the system collapses multiple Aura messages into a single MPI Broadcast or MPI Send message per frame. This is done to prevent latency and MPI overheads. The Aura messages consist of a simple header optionally followed by a block of raw data (events or scene graph data).

## 4. Experiments

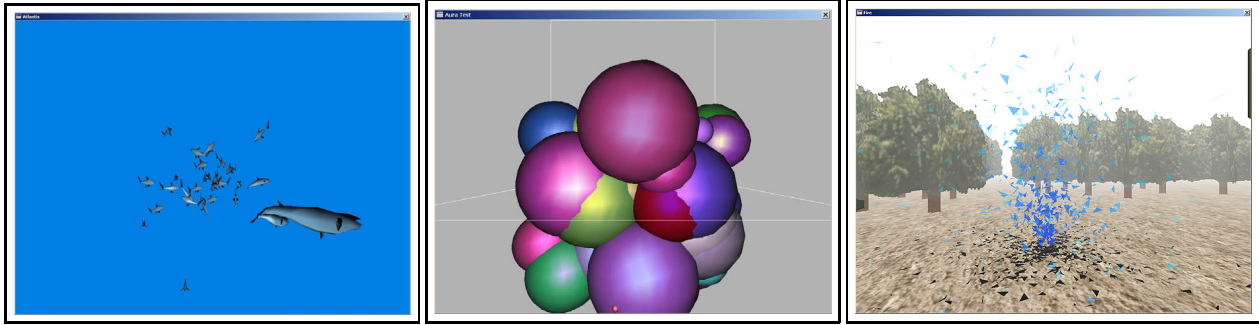
This section compares the parallel renderers Aura Multiple Copies and Aura Broadcast to WireGL. First, we describe the programs used for testing and the hardware setup. Then, we discuss the results obtained with and without actual rendering.

### 4.1. The Benchmark Applications

The comparisons between Aura and WireGL in this section are based on tests with three different application programs. All three programs are original OpenGL programs. For WireGL, these programs remained unmodified. For Aura, the test programs were rewritten, however the graphic complexity and the output are identical to the output of their respective OpenGL versions (i.e. all geometrical shapes are constructed using the same algorithm, and every Aura object corresponds with a single glBegin()/glEnd() loop from the OpenGL program). The three applications, as shown in Figure 3, are:

- **Atlantis:** It is a GLUT demo application and displays two whales and a dolphin swimming in circles in front of the camera. In the center of the screen, a number of sharks are swimming. Atlantis is an immediate-mode program because the objects are updated every frame. However, not all vertices of the sharks are updated every frame, only the tail is flapping.
- **Spheres:** Spheres is a purely retained mode application. It consists of a number of spheres of a specific complexity, that rotate as a whole in front of the camera. The spheres do not change at all (shape, color).
- **Fire:** The Fire program is also GLUT demo application. It displays a grass plane with a number of textured trees. In the center of this field, a 'fire' burns. The fire consists of a large number of triangles (and their shadows) that fly through the air. The triangles are updated each frame and thus are of an immediate mode nature.

Figure 4 shows, per application, the sequential execution times and their complexity for Aura and OpenGL. The throughput column gives an indication of the amount of data (vertex, color, normal, texture coordinates, etc) needed per frame. The use of OpenGL vertex buffers in the Aura implementation makes it significantly faster in some cases, by limiting the OpenGL overhead. Fire and Atlantis have been chosen for their immediate mode nature. They should prove that both Aura versions can handle immediate mode programs and that they do not perform much worse than WireGL. In addition, Fire serves as an example of a worst-case program for WireGL and Aura Broadcast. Due to

(a) *Atlantis* application(b) *Spheres* application(c) *Fire* application**Figure 3. Benchmark applications**

<i>Application</i>	Complexity # of items	Polygons per frame	Throughput Kbytes per frame	OpenGL sequential frame per second	Aura sequential frame per second
<b>Atlantis</b>	25	2949	69.1	391.3	310.0
<b>Fire</b>	2000	4076	73.8	37.6	104.4
<b>Spheres</b>	100	48000	1125.0	137.6	174.5

**Figure 4. Applications description**

the implementation of Fire (all triangles are rendered in a single `glBegin()/glEnd()` block, causing the entire triangle fountain to be resent to every server each frame), WireGL and Aura Broadcast need a high amounts of bandwidth. Spheres, with a high polygon count, serves as an example retained mode program. It is used to show that both Aura implementations perform much better than WireGL on this type of program. The results of the tests are presented in the next two subsections. Four different tests were performed per application:

- Run of the OpenGL implementation on WireGL,
- Run of the Aura implementation with Aura Multiple Copies,
- Run of the Aura implementation with Aura Broadcast,
- Run of the Aura implementation with Aura using WireGL.

The latter is added to prove that better results for Aura Broadcast are not just caused by faster rendering (if the Aura sequential is faster than the OpenGL sequential). It should also show that the performance of Aura Broadcast and Multiple Copies is significantly better than Aura on WireGL.

## 4.2. Setup

We run all the tests on our graphic cluster described in Figure 5. It is made of eight rendering nodes and a server, all constituted of the same components. The main parts are:

- Dual-CPU system, with two AMD Athlon MP 1.2Ghz processors,
- 512MB of main memory,
- NVIDIA GeForce3 graphics card (ASUS V8200),
- 100Mbits Ethernet network card,
- and a high-performance Myrinet network card (not used in this study).

All the nodes run the Linux operating system, with the standard MPI library for communication and the NVIDIA-provided driver for hardware-accelerated OpenGL. Each node is connected to a video projector to compose a display of around 5.5meters by 2meters, in a 4x2-tile configuration. The choice of a dual-cpu system will allow to run the rendering process concurrently to a networking process or a simulation program (for instance an isosurface extraction) which will provide graphical data to the rendering process. The use of two network interfaces (Ethernet and Myrinet) will enable the study of network requirements of different

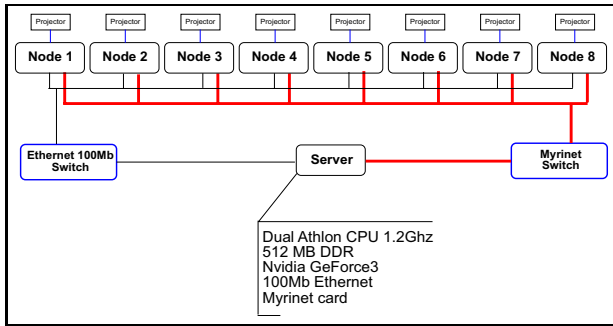


Figure 5. Architecture of the cluster

parallelization strategies. All these points will be addressed in future publications.

### 4.3. Communication Analysis

Before executing the full tests, we run the applications for 1002 frames without rendering. Using these executions, we can analyze precisely the network requirements and scalability of each parallelization without the graphics implementation differences. This is made possible using an “empty” OpenGL dynamic library. All the OpenGL calls are performed, but without any rendering action. Results are shown in Figure 6 for the three rendering implementations (“Aura MC” is Multiple Copies version and “Aura BC” is Broadcast version) and the three benchmarks.

First, we can notice that Multiple Copies, as expected, shows very stable performances since very little communication occurs. The slight decrease in performances is due to the barrier at the end of each frame. Aura Broadcast exhibits good performances for the Spheres benchmark since the scene is mostly static, but also for Atlantis where part of the scene is dynamic. The Fire benchmark, a fully dynamic one, shows the worst performances for Aura Broadcast. However, these results are always better than the WireGL ones, which requires an increasing bandwidth as the number of processors scales. Comparing the results between Aura Broadcast and WireGL for Atlantis are quite instructive: both start at around 50fps (frame per second) for one and two processors, but WireGL quickly decreases to 31fps for eight processors while Aura Broadcast performance is quite stable (47fps for eight processors). Rapidly, WireGL hits the maximum throughput of the Ethernet network, as also shown in Figures 7, 8, and 9 for the executions with graphic rendering.

### 4.4. Rendering Analysis

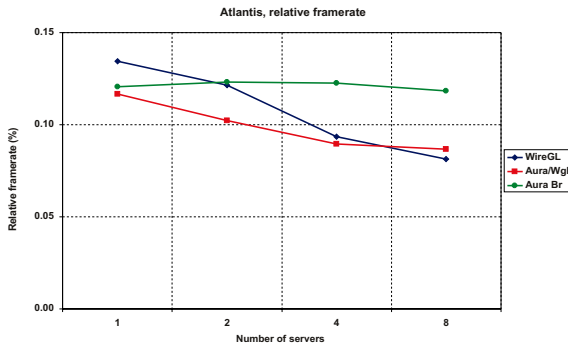
We executed the three previously described applications on our cluster using only the Ethernet network. They were

Fire			
Processors	Aura MC	Aura BC	WireGL
1	249.4	41.8	29.2
2	241.7	25.4	15.1
4	233.2	14.4	9.9
8	221.5	7.7	5.0
Atlantis			
Processors	Aura MC	Aura BC	WireGL
1	504.0	50.5	50.5
2	500.3	49.6	47.5
4	449.0	49.1	36.5
8	405.4	47.4	31.8
Spheres			
Processors	Aura MC	Aura BC	WireGL
1	459.4	222.2	7.8
2	477.8	217.6	6.6
4	451.6	198.7	5.1
8	412.4	139.1	4.1

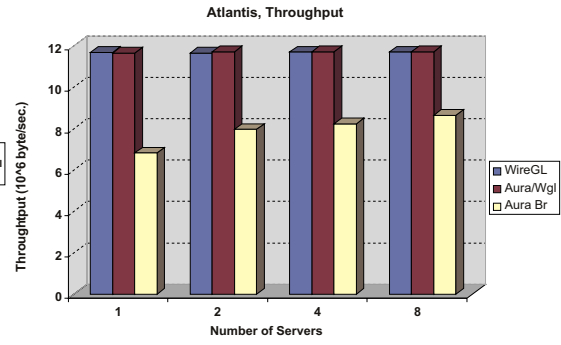
Figure 6. Parallel runs, without rendering, in frame per second

executed for 1002 frames. As in a speedup formula for a parallel program, the relative frame rate for each application on the cluster is compared to the fastest sequential result for that application as described in Figure 4. A relative frame rate (or speedup) of one means that the program achieves the same frame rate in parallel than the fastest sequential implementation. Figures 7 through 9 show on the x-axis the number of servers and on the y-axis the speed-up (relative to the fastest sequential). For Fire and Atlantis the results for Multiple Copies are left out for readability. The relative frame rate for Multiple Copies are around 0.7 to 0.8 on for all configurations for these applications. The bandwidth graphs show the data cumulated throughput required by the client (the total number of bytes going over the network, in  $10^6$  byte/second) for both WireGL configurations and Aura Broadcast. All the values for Aura Multiple Copies are almost zero and thus left out.

As expected, Aura Broadcast and Multiple Copies perform better than WireGL on the retained mode Spheres application. Throughput of Broadcast is minimal and the throughput of Multiple Copies is close to zero. WireGL consumes up all bandwidth by resending all the polygons every frame. Both Aura implementations achieve (minor) speedups due to the fact that each processor can clip a lot of objects before rendering, effectively reducing the amount of work. The dip for eight processors is caused by the fact that, for such a configuration, the work load is not balanced.

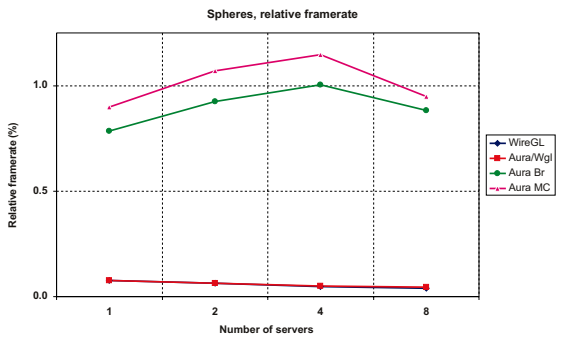


(a) Speedup

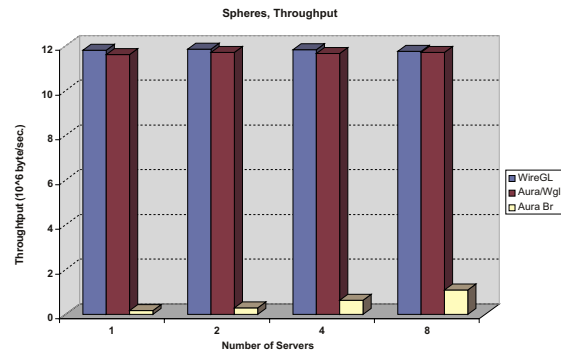


(b) Bandwidth

Figure 7. Atlantis results

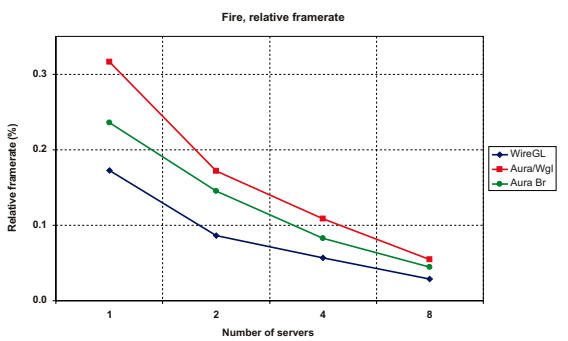


(a) Speedup

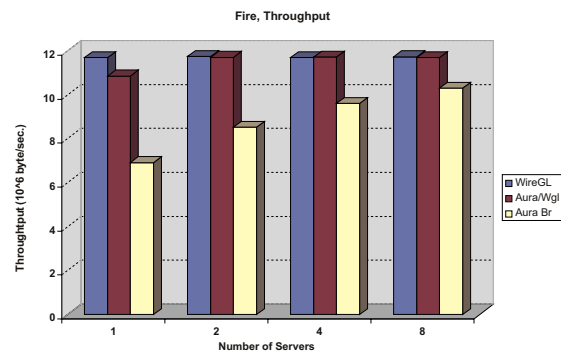


(b) Bandwidth

Figure 8. Spheres results



(a) Speedup



(b) Bandwidth

Figure 9. Fire results

Aura (like WireGL) does not apply any load balancing. For Atlantis, WireGL performs best initially (if we ignore Aura Multiple Copies). However, again Aura Broadcast can save bandwidth by using a strategy that prevents unmodified vertices to be sent over the network every frame. Therefore, its scalability is much better, resulting in a better performance for eight servers. From the bandwidth graph, it is clear that Aura Broadcast does not use the entire available bandwidth due to inefficiencies in the message passing layer. Certain optimizations in the communication scheme can increase this throughput, which will increase performance of Aura Broadcast on both Fire and Atlantis applications. Also, the performance of Broadcast on the spheres application can be improved by some optimizations. These changes will be applied in the near future. WireGL is using all network bandwidth (100Mbit/sec Ethernet network, so around the 12 Mbyte/sec, limit clearly visible in the graphs) and so is network bound. It will be difficult to optimize WireGL on such network, and will require an gigabit network (such as Myrinet) to perform well.

## 5. Conclusion

This paper focuses on rendering for large tiled displays using an off-the-shelf cluster of PCs. The Stanford University parallel renderer WireGL is the most used renderer in the field. It has the advantage of being compatible with all existing OpenGL programs. However, its scalability is poor on a regular network. The two Aura renderers presented in this paper try to solve this scalability problem, by using retained mode graphics instead of immediate mode graphics if possible. Aura Multiple Copies uses simple replication of the application and synchronization to achieve the best performance. Aura Broadcast broadcasts scene data to co-operating PCs to achieve complete transparency. Tests regarding the communication requirements of WireGL and both Aura versions show that Aura Multiple Copies has by far the fastest communication and the best scalability. Aura Broadcast, a more flexible and transparent renderer than Aura Multiple Copies, performs almost as good on real retained mode benchmark tests. On the immediate mode oriented tests, it performs as good as WireGL. On the tests for the partially retained mode application (Atlantis), it scales a lot better than WireGL. Some optimizations to Aura Broadcast can improve its performance, whereas the implementation of WireGL seems to have reached its limits.

It has never been the purpose of this paper to show that Aura is 'better' than WireGL. It tries to show that there are alternatives. For standard OpenGL programs, WireGL is the way to go. However, if a new 3D-visualization program is designed to run on a parallel system, WireGL might not be the obvious choice. If performance is an issue, Aura (or any other immediate mode renderer) might be a better

choice. Aura also has a simpler, more natural interface. The choice between Aura Broadcast and Aura Multiple Copies also requires some thought. If the program is to run only on a cluster and does not read large files or databases from a remote location, Aura Multiple Copies is the best option. Otherwise, Aura Broadcast should be used.

## References

- [1] I. Buck, G. Humphreys, and P. Hanrahan. Tracking graphics state for networked rendering. In S. N. Spencer, editor, *Proceedings of the 2000 Eurographics Workshop on Graphics Hardware*, pages 87–96. ACM Press, Aug. 21–22 2000.
- [2] T. W. Crockett and T. Orloff. A parallel rendering algorithm for MIMD architectures. *ICASE Report 91-3*, June 1991.
- [3] M. Czernuszenko, D. Pape, D. Sandin, T. DeFanti, G. L. Dawe, and M. D. Brown. The ImmersaDesk and infinity wall projection-based virtual reality displays. *Computer Graphics*, 31(2), May 1997.
- [4] D. Germans, H. J. Spoelder, L. Renambot, and H. E. Bal. VIRPI: A High-Level Toolkit for Interactive Scientific Visualization in Virtual Reality. In *5th Immersive Projection Technology Workshop (IPT98)*, May 2001.
- [5] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan. Distributed rendering for scalable displays. In *SC2000: High Performance Networking and Computing*. Dallas, TX, USA. ACM Press and IEEE Computer Society Press, 2000.
- [6] G. Humphreys and P. Hanrahan. A distributed graphics system for large tiled displays. In D. Ebert, M. Gross, and B. Hamann, editors, *Proceedings of the 1999 IEEE Conference on Visualization*, pages 215–224, Oct. 25–29 1999.
- [7] P. Karlton. Integrating the GL into the X environment: A high performance rendering extension working with and not against X. *The X Resource*, 1(1):27–32, Jan. 1992.
- [8] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Early experiences and challenges in building and using A scalable display wall system. *IEEE Computer Graphics and Applications*, 20(4):29–37, July/Aug. 2000.
- [9] T. Mitra and T. cker Chiueh. Implementation and Evaluation of the Parallel Mesa Library. Technical report, State University of New York at Stony Brook, 1998.
- [10] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.
- [11] U. of Minnesota. Power Wall. <http://www.lcse.umn.edu/research/powerwall/powerwall.html>.
- [12] R. Samanta, T. Funkhouser, K. Li, and J. P. Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of PCs. In *Proceedings of the 2000 SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 97–108. ACM Press, 2000.
- [13] B. Schaeffer. A Software System for inexpensive VR via Graphics Clusters. <http://www.isl.uiuc.edu/ClusteredVR/paper/DGDOverview.htm>, 2000.