Griz: Experience with Remote Visualization over an Optical Grid

Luc RENAMBOT [a,*], Tom VAN DER SCHAAF [a], Henri E. BAL [a,b], Desmond GERMANS [b], Hans J.W. SPOELDER [b]

[a] *Division of Mathematics and Computer Science*

[b] *Division of Physics and Astronomy*
Faculty of Sciences, Vrije Universiteit
De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands

Abstract

This paper describes the experiments of remote rendering over an intercontinental optical network during the *iGrid2002* conference in Amsterdam from September 23 to September 26. A rendering cluster in Chicago was used to generate images which were displayed in real-time on a *4-tile* visualization setup in Amsterdam. On average, one gigabit per second ($1Gbps$) was consumed to enable remote visualization, at interactive frame-rate, with a $1600x1200$ pixels configuration.

Key words: remote visualization, parallel rendering, optical network, reliable UDP, interactive application

1 Introduction

The recent advances in optical networking are deeply changing the networking landscape, and in general the way we think about networked resources such as computing, storage, or remote instruments. "Lambda" networking with technologies such as WDM (wave division multiplexing) are on the verge of providing the "infinite bandwidth" dreams of the computer scientists. For instance, in a WDM system, each wavelength in an optic fiber can carry from 1 to 10 gigabit per second. An example of such an infrastructure is the StarLight/NetherLight optical network between Chicago and Amsterdam.

* Contact the corresponding author at `renambot@cs.vu.nl`

On the other hand, the current trend in hardware for parallel graphics is to use clusters of off-the-shelf PCs instead of high-end graphic supercomputers. This trend has emerged since the dramatic change in the price/performance ratio of today's PCs, in particular concerning their graphics capability. By applying similar techniques developed for cluster computing, graphics clusters can now drive large-scale high-resolution systems, overcoming the limited output resolution of standard devices such as monitors and video projectors. High resolution can be achieved by tightly coupling projectors (or TFT monitors), allowing for detailed scientific visualization with an increased pixel density.

This paper describes the experiments combining parallel rendering and usage of a very large bandwidth network to achieve remote visualization of interactive applications between Chicago (USA) and Amsterdam (The Netherlands) during the international *iGrid2002* conference in September 2002. A rendering cluster in Chicago was generating images which were displayed in real-time on a *4-tile* display setup in Amsterdam. On average, one gigabit per second (1 Gbps) was consumed for visual applications at interactive frame-rate (≈ 20 frames per second) for a 1600×1200 pixel frame-buffer.

The reasons to explore such configurations for remote visualization in a grid environment are:

- Rendering facilities are still scarce, compared to computing or storage resources, and need specific skills to be used,
- Advanced "consumer" graphics cards combined with fast processors are the basic blocks to provide large-scale graphic resources,
- Very large bandwidth between sites, provided by optical networks, opens a wide range of new possibilities to conduct research. It deeply changes the way we think about remote resources such as storage or visualization, and the nature of remote collaboration between scientists.

The considered scenarios for remote visualization in such a networking environment are:

- Remote rendering: using a remote cluster with a local display. The user wants to access a high-performance rendering facility not available locally. This facility can use a high-speed local network within an institution, or an optical backbone over a large distance, such as the one used during the *iGrid* conference. The target display can be a single monitor or a high-resolution tiled display.
- Support for collaborative visualization: two or more sites can visualize the same application or dataset to support a collaborative session.

In the following sections, we are describing the techniques and tools that enabled this demonstration. We will focus on the two main points, namely

pixel generation using parallel rendering, and network engineering to achieve high-speed data transfer over high-latency high-bandwidth network.

2 High-speed Networking

High-speed transfer of large datasets is an critical part of many data-intensive scientific applications. However, a major problem for the application developer is to achieve high throughput transfers over high bandwidth, long distance networks. This relates mostly to the design of the *TCP/IP* protocol. While performances are good over a large class of networks, high latency induced by long distances makes it very hard to get part of the available bandwidth. Algorithms of the *TCP/IP* protocol to detect packet loss and to deal with congestion are too sensitive and take too long to recover for gigabit networks with round-trip times (RTT) over 100ms (i.e., Chicago-Amsterdam RTT is around 100ms, Chicago-Japan RTT is round 150ms). For instance, the loss of a few packets can affect for a long time such a network. To illustrate these problems, Olivier Martin [6] reported at the Global Grid Forum in September 2001 that:

- The loss of a single packet will affect a 10Gbps stream with 200ms RTT for 5 hours. During that time the average throughput will be 7.5 Gbps. This loss is perceived as congestion by the aggressive TCP protocol which then decreases the emission rate. The high latency and the slow progression of the emission rate makes this loss very long to recover from.
- On a 2.5Gbps link with 100ms RTT, this translates to 38 minutes recovery time, during that time the average throughput will be 1.875Gbps.
- On a 2.5 Gbps link, a bit error rate of 10^{-9} translates to one packet loss every 250 milliseconds.

However, the non-reliable UDP protocol does not suffer from these problems. Consequently, several projects proposed new protocols ensuring reliable communication based on UDP, without much focus on the congestion aspect. This last aspect raises the problem of fairness of these protocols since they use all the available bandwidth. In the extreme case, this could be considered as a "denial of service" (Dos) attack. However, programs making use of theses techniques are mainly deployed on research networks where this is not yet a problem.

These protocols or programs, such as *GridFTP*, *RBUDP*, *SABUL*, and *Tsunami*, are generally based on the coupling of a UDP channel for high-speed transfer and a TCP channel to control the data emission rate and signal the packet loss.

- GridFTP [1] is an enhanced version of the popular File Transfer Protocol, targeted for very large transfers between Grid sites around the world. The feature rich GridFTP includes capabilities such as automatic negotiation of window sizes between client and server, parallel data streams, security, and reliability support.
- RBUDP [3] (Reliable Blast UDP) is an aggressive bulk data transfer scheme, designed for extremely high bandwidth networks. The protocol contains two phases: a blast of data is sent to destination using a UDP channel at a specified rate, followed by reliability phase over TCP, where lost packets are sent again. RBUDP is part of the *QUANTA* toolkit, described later in this paper.
- SABUL [8] (Simple Available Bandwidth Utilization Library for High-Speed Wide Area Networks) is a C++ library for large data transfers over high-speed wide area networks. It is similar to the RBUDP protocol. However, SABUL maintains state information and controls data rate to minimize packet loss over the duration of the transfer.
- Tsunami [9], similar to *GridFTP*, is an experimental high speed network file transfer protocol designed to overcome some of the TCP problems for high-performance transfer over the considered networks. Again, data transport is carried out using a UDP channel, with limited congestion control. Missing packets are re-sent to ensure reliability.

For the networking aspects of this demonstration, we choose to use the *QUANTA* toolkit, successor of the *CAVERNSoft* project [5, 3]. *QUANTA*, or Quality of Service and Adaptive Networking Toolkit, provides an easy-to-use environment to the application programmer to achieve high-speed data transfer. It also gives the application various metrics to characterize and monitor each network connection. It becomes easier for the non-expert to understand the behavior of the application, and to optimize it for a specific network.

The *QUANTA* toolkit provides a *C++* interface containing:

- TCP, UDP, and multicast objects (client, server, reflectors),
- Socket parameter tuning, with defaults values for wide-area networks,
- Various high-speed protocols:
  - *Parallel TCP*, using transparently multiple sockets to achieve higher throughput by overcoming the window-size and latency shortcomings.
  - UDP with *Forward Error Correction*, where packet loss is resolved by encoding multiple times the same packets with checksums,
  - Reliable Blast UDP, the above described reliable high-speed data transfer protocol,
- Monitoring and logging of each connection, with metrics such as immediate and average bandwidth, latency, and jitter.
- A portable implementation for Linux, Windows, and IRIX operating systems.

3 Remote Rendering

Our remote rendering infrastructure consists of a parallel rendering infrastructure which generates the images and a remote visualization technique which brings the pixels to a remote location.

3.1 Parallel Rendering

The visualization in this work is based on the Aura API developed at the Vrije Universiteit in Amsterdam. Aura is designed as a portable 3D graphics scene graph layer [2]. Aura allows the user to apply modifications to any object attributes in the graph. For a complete description of Aura and some case study applications, we refer to [2].

In our parallel approach [10], the applications are executed only on the master. All graphics commands from the API are marshaled and sent to all slaves, instead of being executed locally. This approach has the advantage that all cluster-related issues can be hidden in the implementation of the interface. However, it requires graphic commands to be sent over the network, and it potentially needs a high bandwidth. High-speed networks such as Myrinet or Gigabit Ethernet solve this problem only partially.

We use a per-frame sort-first strategy to distribute the graphic updates [7], in a comparable approach to WireGL [4]. Whenever something changes in the scene graph, it is sent to the appropriate slaves (i.e. those that are affected by the change). The correct destination slave is computed using bounding boxes, defined by clipping planes for each sub-frustum. This strategy is simple to implement but can induce load imbalance. The protocol to keep the scene graph consistent on all slaves, distinguishes two types of operations:

Scene Graph operations The master sends separate update messages to create nodes, add nodes, and remove nodes from the scene. These types of operations are broadcast to every slave to maintain a consistent graph on each node.

Object Modifications Whenever the user-program modifies an Aura object, the master marks it as dirty. Before the rendering loop, it transmits an update containing the new data for each dirty object. Each type of object owns its own set of update types and state bits. For example, each vertex of a geometric object has a dirty bit and only dirty vertices are sent over the network. The same approach is used for normals, vertex colors, etc. The key idea is that a minimum of data is sent if an object is modified.

For the *iGrid2002* demonstration, this parallel rendering was implemented using standard MPI library over the *Gigabit Ethernet* network linking the rendering nodes. The rendering itself was managed by an OpenGL graphics card available on each node.

Pixel Generation Instead of connecting the output of the graphics card of each node to an display device (TFT panel, plasma TV, video projector) like in a regular tiled display, we read back the pixels at the end of the graphics pipeline (RGB values). This is a standard operation of the OpenGL API (i.e. *glReadPixels*).

Pixel reading is often considered as a slow process on most graphics cards. However, the situation is improving to the point where one can achieve interactive frame-rates while reading back the pixels generated by the graphic engine. We conduct experiments on two systems to validate our approach:

- During iGrid2002, on GeForceMX2 cards with P4 1.8GHz processor:

Resolution	Pixel Transfer	Frequency
640×480	18.5Mpixels/sec	60Hz
800×600	17.5Mpixels/sec	36Hz
1024×768	16.0Mpixels/sec	20Hz

  On such a system, interactive frame-rates are possible for a configuration above 800×600.

- On the *Vrije Universiteit* cluster, with GeForce4 cards and Athlon 1.3MHz processors:

Resolution	Pixel Transfer	Frequency
640×480	35.1Mpixels/sec	114Hz
800×600	38.5Mpixels/sec	80Hz
1024×768	40.5Mpixels/sec	51Hz

  On this more advanced system, even 1024×768 configuration can be used interactively.

Since we were using a very high-bandwidth and high-latency network during the *iGrid2002* demonstration, compression of the pixel data was not considered. However, early tests show that a simple *run length encoding* (RLE) or a *YUV* (raw video) conversion allows significant gains at interactive speed on a Pentium4 processor (without optimizations or multimedia instructions): it

is possible to compress 1024x768 images in RLE format at more than 60Hz, for a compression ratio varying between 2.3 and 25, and to compress the same images in YUV format at more than 80Hz for a compression ratio of 2.

3.2 Remote Visualization

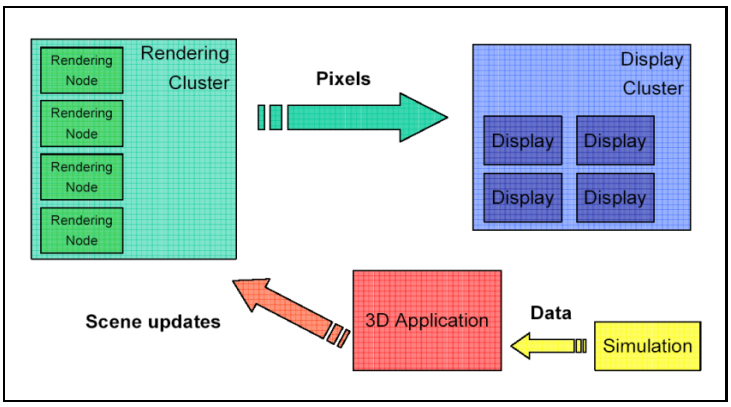


Fig. 1. Application Setup

The demonstration shown during *iGrid2002* consists of three components, as shown in Figure 1:

- The *application* generating scene-graph updates. This is a non-graphical process that can be run on a single host. The application program and the interaction events (mouse, keyboard) modify the scene graph. The updates are sent to the rendering cluster for pixel generation. The application can receive its data from an external simulation, in a grid interactive steering scenario.
- The *rendering* facility implements the parallel rendering system described in the previous section. It can consist of a single graphics workstation or rendering cluster. From the scene graph updates, the rendering generates images accordingly. The pixels are then streamed to the display infrastructure.
- The *display* infrastructure receives the pixels generated by the rendering facility, and schedules the images for a smooth animation. It can consist of a single display (monitor or projector) or a tiled display (video wall, TFT tiling).

The technologies used in this application are started as follows. First, a rendering cluster is set up with an empty scene to be rendered. Then an application using the Aura API is started on another machine and transmits scene graph updates to the rendering cluster. From the received scene each node of the cluster renders a part of the whole picture and send the pixels to a display facility. Finally, the generated images are visualized on the display machines.

The communication between these three software components (application, rendering, display) was carried out using the *QUANTA* toolkit, as described in Figure 2: between the application and the rendering cluster we used TCP connections to ensure the needed reliability. Large TCP windows were set to optimize these transfers. Between the rendering nodes and the the display setup, we used the *RBUDP* protocol to achieve the highest possible throughput for pixel data. The only parameter we controlled was the data emission rate, which was generally set around 500Mbps. This parameter was controlled by a configuration file, which value was set by experimentation before an actual run of the demo. In a ideal case, this value should be dynamic, like in the TCP protocol or the *SABUL* library, in order not to overload the routers or the destination machine. For optimal data transfer, the emission rate should be below the capacity of the whole link (physical links, routers, source and destination machines).

Even if the application can be run on any site, two places should be considered. First, the application can be run close to the display facility, in a scenario where the user uses the remote rendering cluster as graphics resource. In a second scenario, the application can be run close to the rendering facility for a collaborative session where two sites (the local rendering, and the remote display) share the views of the visualization. The second approach (application close to the rendering facility) was used during *iGrid2002* to overcome the large latency between the two sites (around 100ms). Remote interaction, using mouse and keyboard events, was still possible using a X11 remote display.

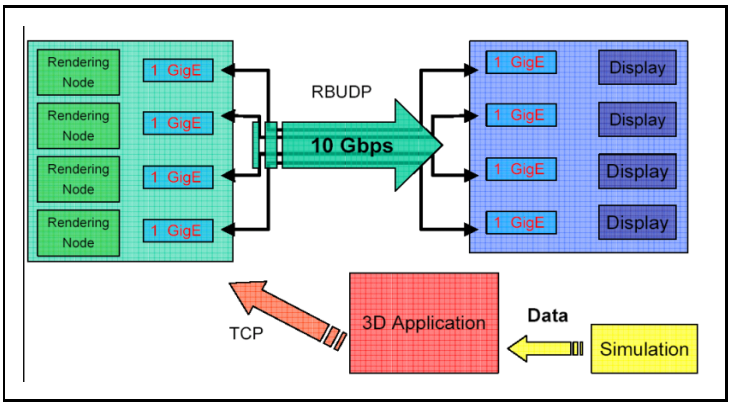


Fig. 2. Networking Setup

Multi-threaded communication To achieve high-throughput at the application level between the two end points, we had to carefully design the data path between the software components. We used a multi-threaded implementation to obtain the best network throughput without slowing down the rendering process, as shown in Figure 3.

Since the workload was small (frame size between 1 and 2 MByte) and the network latency was high (100ms), we decided to pack several frames into a single *RBUDP* message. The rendering thread pushes frames (pixel buffers) into a work queue, in a *producer-consumer* fashion. This allows the rendering process to work without delay, and allows the network thread to send a large amount of data at once. Experimental tests gave good results while combining 8 frames in a message (in general, between 5 and 10 frames). Small messages under-utilize the network and slow down the rendering process, while large messages can achieve the largest throughput possible. However, large messages introduce a lag in the system, where the display process receives the images, in a pipeline fashion, delayed by the size of the message (5 or 10 frames). This can be a drawback for interactive application. We noticed that around $1/3sec$ delay is bearable on such high-latency network, and offers the best compromise between throughput and latency.

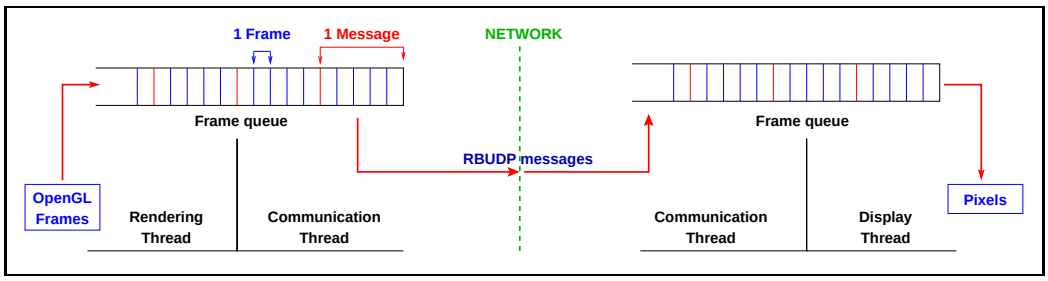


Fig. 3. Multi-threaded communication

Similarly, we implemented a network queue at the display side, managed with a *producer-consumer* scheme: a network thread reads the frame packets from the network and inserts them in a queue, while a display thread visualizes these images at a fixed frame rate. The network queue and the re-scheduling of the frames locally produce a fluid animation, and smooth out the jittery network transfers.

Algorithms Figure 4 gives an overview of the several programs involved in the remote rendering system: the application processing events and sending scene updates to the rendering master which controls the rendering nodes, the nodes send pixel buffers to the display node for visualization.

```
// Application
Setup(application);
while ( ! done )
{
   ApplicationStep();
   EventProcessing();
   PackSceneUpdates();
   SendUpdates();

}
```

```
// Master
Setup(cluster);
while ( ! done )
{

   ReceiveUpdates();
   Synchronize()
   BroadcastUpdates();

}
```

```
// Rendering
Setup(graphics);
while ( ! done )
{
   Draw();
   ReadPixels();
   SendPixels();
   Synchronize();
   ReceiveUpdates();
   UpdateScene();
}
```

```
// Display
Setup(display);
while ( ! done )
{

   ReceivePixels();
   ConvertPixels();
   WriteFramebuffer();
   SwapBuffers();
}
```

Fig. 4. Algorithms: application, master, rendering, and display

4 Experiments

During the *iGrid2002*, the *Griz* demonstration was run successfully numerous times. In the current section, we show the setup we used and some statistics collected during the last day of the conference.

4.1 Setup

The hardware architecture, as sketched in Figure 2, was mostly provided by the *Electronic Visualization Laboratory* from the University of Illinois at Chicago. For the rendering, we used a 5-node graphics cluster set up at the *StarLight* facility in Chicago. Each node was using a dual-Pentium4 system with a GeForceMX graphics cards (consumer cards), and equipped with a gigabit interface linked to the 10Gigabit backbone to Amsterdam.

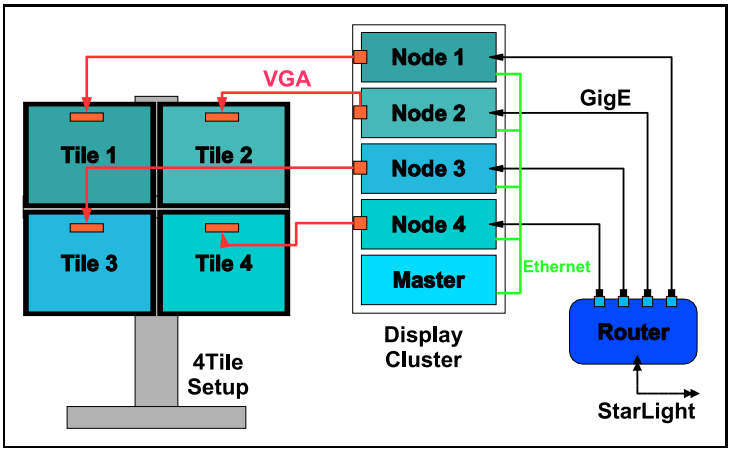
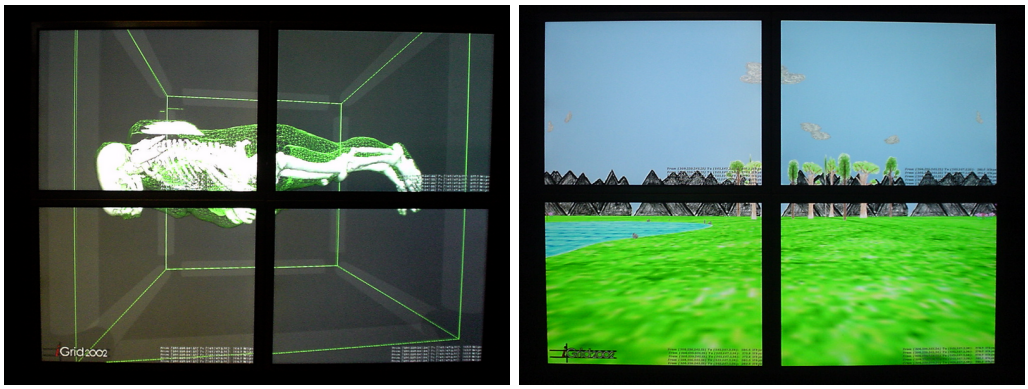


Fig. 5. Display Infrastructure

In Amsterdam, we used a tiled display made of four TFT 18" panels arranged in 2x2 configuration for an overall resolution of 2560x2048 pixels, as shown in Figure 5. This display, located on the demonstration site in Amsterdam, is itself driven by a cluster similar to the one used for rendering at Chicago. Moreover, it is also equipped with gigabit interfaces onto the *StarLight* network. Little graphics resources are used on this cluster. The received images are just drawn into the frame-buffer with some additional information (frame rate, network utilization, etc.). The portable *SDL* graphics library was used for its pixel blitting, scaling, and overlays capabilities.

We used several resolutions on this setup, ranging from 640×480 to 1024×768 pixels per node. However, the images were up-scaled to fill each screen (native resolution of 1280×1024). This trick was hardly noticeable since we used high-quality settings during the rendering phase, with techniques such as "full-scene anti-aliasing" and "anisotropic texture filtering".



(a) "Skeleton" application (b) "Crayoland" application

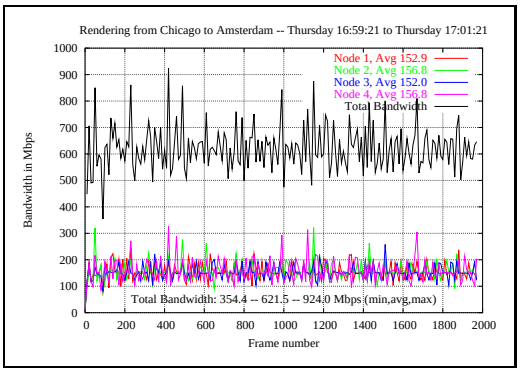
Fig. 6. Images of the 4Tile setup

We demonstrated various applications during the event, such as a *Powerpoint*-like presentation tool, a 3D model viewer (Figure 6(a)) for the visible human dataset, or a interactive 3D world ("EVL Crayoland", Figure 6(b)). Figure 6 shows two photographs of the visualization setup with the four TFT panels, during the *iGrid2002* event.

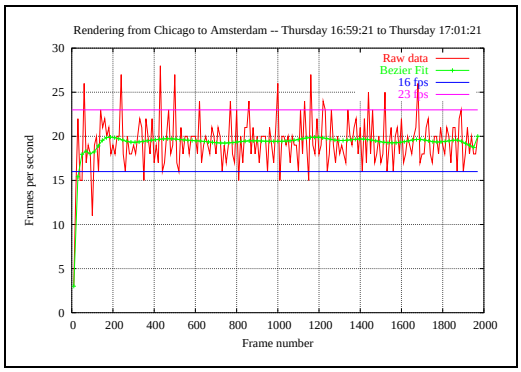
4.2 Results

Figure 7 shows the throughput and the visualization frame rate obtained for a 640×480 configuration per tile (1280×960 overall resolution), during the approximate 2 minutes of a demonstration run. As noted in a previous section, the maximum pixel read back performance on the *iGrid2002* hardware is around 60Hz (or frame per second), excluding any rendering. We managed to achieve an average frame rate between 16fps and 23fps, for an average overall throughput of 621Mbps. This performance is sufficient to enable interactive application and collaboration. While each of the four rendering nodes consume around 150Mbps, the overall system manage to peak at a 924Mbps. However, we can notice a significant jitter in the throughput, which directly influences the effective frame rate, as described in "zoom-in" Figure 8 where bandwidth usage is depicted on the left axis and frame rate on the right axis. This jitter is probably due to some buffering and contention on the various routers (10 Gigabit router of the rendering cluster in Chicago, and the 10 Gigabit router on the demonstration site). Also, other demonstrations were scheduled at the same time. More details on the low levels of the networking setup are discussed in various papers in the proceedings.

Figures 9 and 10 report similar results for a 800×600 pixel configuration. Notably, the frame rate remains similar (between 16fps and 23 fps), but the



(a) Bandwidth statistics: 354 Mbps minimum – 621 Mbps average – 924 Mbps peak



(b) Frame rate: between 16fps and 23fps, on average

Fig. 7. A "640 × 480 configuration" run

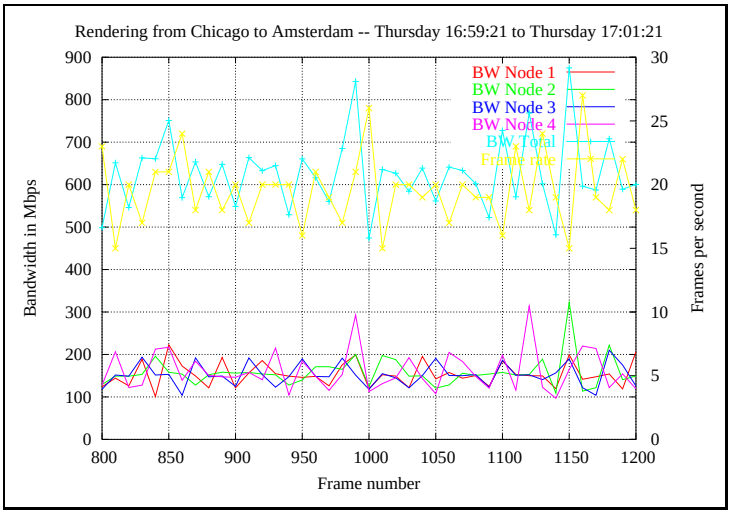
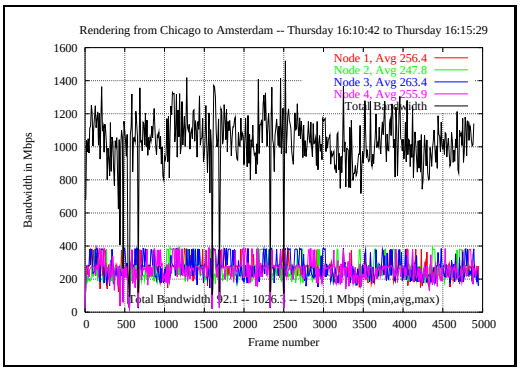


Fig. 8. Zoom on a "640 × 480 configuration" run

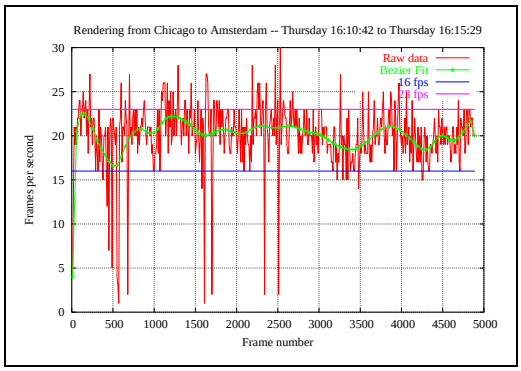
larger frame buffers consume more bandwidth with an average of 1.0Gbps to a maximum of 1.5Gbps during the 5 minutes of the run. Each of the four rendering nodes is able to push around 250Mbps. This specific run transfered approximatively 300Gbit (37.5GB) of data between Chicago and Amsterdam.

While the frame rate matches the obtained throughput (800*600*RGB with 8 frames per message at 20fps is roughly 250Mbps), the throughput is lower than expected (500Mbps per node). The overhead includes the rendering itself (application dependent) and the pixel read back from the graphics cards (constant). The whole system seems bound to the round trip time of 100ms observed during the event (1000ms / 0.5 RTT = 20fps). This behavior needs further investigation to achieve higher performance. Some severe performance loss can be noticed on Figure 9 and on the zoom Figure 10. This can be explained by some network contention that the buffering mechanism can not

hide, or by some load imbalance on the rendering cluster. The latter can be produced when a rendering node has to render the whole scene (for instance the whole skeleton in the Visible Human application) while the other nodes are idle. This view-dependent behavior should be solved by a more efficient parallel rendering technique.



(a) Bandwidth statistics: 92 Mbps minimum – 1026 Mbps average – 1520.1 Mbps peak



(b) Frame rate: between 16fps and 23fps, on average

Fig. 9. A "800 × 600 configuration" run

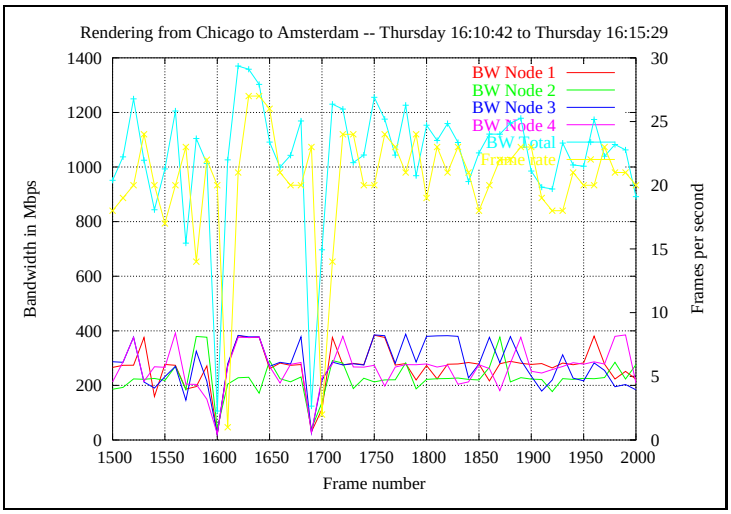


Fig. 10. Zoom on a "800 × 600 configuration" run

In Figure 11, it is possible to pin-point these specifics runs of the *Griz* demonstration on the network statistics collected during the *iGrid2002* event. This figure reports the incoming traffic (solid curve) and the outgoing traffic (line curve) for the Thursday 26^{th} of September. Our runs, between 16.00 and 17.00, clearly generates an additional traffic, up to 1.5Gbps.

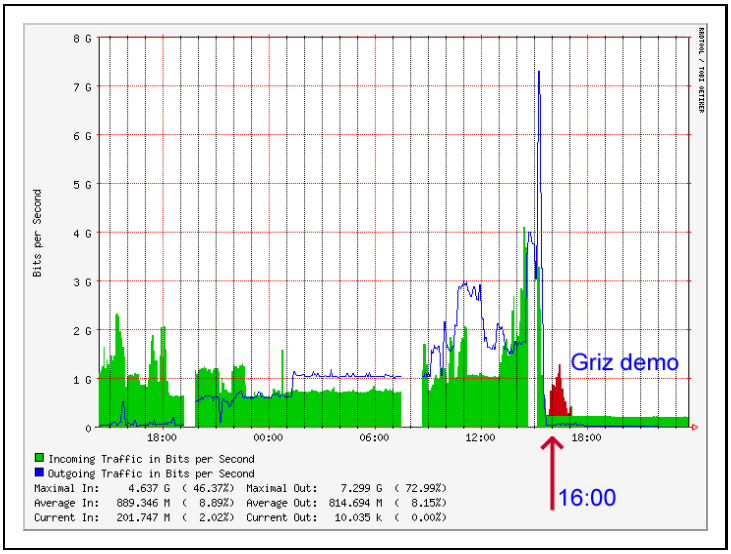


Fig. 11. Griz demonstration network usage on the statistics graph

5 Conclusion

We demonstrated in this event and publication that remote rendering over intercontinental optical networks was possible at interactive frame rate. Using a 4-node rendering cluster in Chicago and a 4-node display setup in Amsterdam, one gigabit per second ($1Gbps$) was used on average to enable remote visualization with a $1600x1200$ pixels configuration. Various visualization application were shown.

Future enhancements include various compression techniques to cope with the bandwidth requirements on more limited networks. One can think of algorithms such as simple *run-length encoding* or more complex *motion JPEG*. This will lead to an approach similar to video delivery to one or more users, and could be integrated into the AccessGrid environment. The dynamic adaptation to match the requirements of various display environments (PDA, laptop, or large tiled display) will be also considered. Finally, an important issue to be investigated is the latency of the overall system to support collaborative work.

6 Acknowledgments

The authors would like to thank the *Electronic Visualization Laboratory* from the University of Illinois at Chicago for the *QUANTA* toolkit, and for the use of the rendering cluster in Chicago and the 4-tile display during the event.

References

[1] Bill Allcock, Joe Bester, John Bresnahan, Ann L. Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Data management and transfer in high-performance computational grid environments. *Parallel Computing*, 28(5):749–771, May 2002.

[2] Desmond Germans, Hans J.W. Spoelder, Luc Renambot, and Henri E. Bal. VIRPI: A High-Level Toolkit for Interactive Scientific Visualization in Virtual Reality. In *5th Immersive Projection Technology Workshop*, May 2001.

[3] E. He, J. Leigh, O. Yu, and T. DeFanti. Reliable Blast UDP : Predictable High Performance Bulk Data Transfer. In *IEEE Cluster Computing 2002, Chicago, Illinois*, September 2002.

[4] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett, and Pat Hanrahan. WireGL: A scalable graphics system for clusters. In *SIGGRAPH 2001, Computer Graphics Proceedings*, Annual Conference Series, pages 129–140. ACM Press / ACM SIGGRAPH, 2001.

[5] J. Leigh, O. Yu, D. Schonfeld, and R. Ansari et al. Adaptive Networking for Tele-Immersion. In *Immersive Projection Technology/Eurographics Virtual Environments Workshop (IPT/EGVE), Stuttgart, Germany*, May 2001.

[6] Olivier Martin. The EU DataTAG Project. http://datatag.web.cern.ch/datatag/presentations/GGF3-Frascati.ppt, 2001. Global Grid Forum (GGF3) Frascati, Italy.

[7] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14(4):23–32, July 1994.

[8] SABUL. Simple Available Bandwidth Utilization Library. http://www.dataspaceweb.net/sabul.htm, 2002.

[9] Tsunami. http://www.anml.iu.edu/anmlresearch.html, 2002.

[10] Tom van der Schaaf, Luc Renambot, Desmond Germans, Hans Spoelder, and Henri Bal. Retained Mode Parallel Rendering for Scalable Tiled Displays. In *Proc. 6th annual Immersive Projection Technology (IPT) Symposium, Orlando Florida*, March 2002.