Remote High Performance Visualization of Big Data for Immersive Science

Faiz A. Abidi

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

Nicholas Fearing Polys, Chair

Joseph L. Gabbard

Christopher L. North

May 3, 2017

Blacksburg, Virginia

Keywords: Remote rendering, CAVE, HPC, ParaView, Big Data

Remote High Performance Visualization of Big Data for Immersive Science

Faiz A. Abidi

(ABSTRACT)

Remote visualization has emerged as a necessary tool in the analysis of big data. High-performance computing clusters can provide several benefits in scaling to larger data sizes, from parallel file systems to larger RAM profiles to parallel computation among many CPUs and GPUs. For scalable data visualization, remote visualization tools and infrastructure is critical where only pixels and interaction events are sent over the network instead of the data. In this paper, we present our pipeline using VirtualGL, TurboVNC, and ParaView to render over 40 million points using remote HPC clusters and project over 26 million pixels in a CAVE-style system. We benchmark the system by varying the video stream compression parameters supported by TurboVNC and establish some best practices for typical usage scenarios. This work will help research scientists and academicians in scaling their big data visualizations for real time interaction.

Remote High Performance Visualization of Big Data for Immersive Science

Faiz A. Abidi

(GENERAL AUDIENCE ABSTRACT)

With advancements made in the technology sector, there are now improved and more scientific ways to see the data. 10 years ago, nobody would have thought what a 3D movie is or how it would feel to watch a movie in 3D. Some may even have questioned if it is possible. But watching 3D cinema is typical now and we do not care much about what goes behind the scenes to make this experience possible. Similarly, is it possible to see and interact with 3D data in the same way Tony Stark does in the movie Iron Man? The answer is yes, it is possible with several tools available now and one of these tools is called ParaView, which is mostly used for scientific visualization of data like climate research, computational fluid dynamics, astronomy among other things. You can either visualize this data on a 2D screen or in a 3D environment where a user will feel a sense of immersion as if they are within the scene looking and interacting with the data. But where is this data actually drawn? And how much time does it take to draw if we are dealing with large datasets? Do we want to draw all this 3D data on a local machine or can we make use of powerful remote machines that do the drawing part and send the final image through a network to the client? In most cases, drawing on a remote machine is a better solution when dealing with big data and the biggest bottleneck is how fast can data be sent to and received from the remote machines. In this work, we seek to understand the best practices of drawing big data on remote machines using ParaView and visualizing it in a 3D projection room like a CAVE (see section 2.2 for details on what is a CAVE).

Acknowledgments

I would like to thank my adviser, Dr. Nicholas Polys, who was a constant support and encouragement throughout my stay at Virginia Tech. His guidance and suggestions were pivotal in making this research work possible. I also got to work with some wonderful people at the Visionarium lab at Virginia Tech including Ayat Mohammed, Srijith Rajamohan, and Lance Arsenault, who helped me with some of my work. Last but not the least, I would like to thank my parents for everything they have given to me. They are the most important people in my life.

Contents

List of Figures viii

List of Tables xi

1 Introduction 1

1.1 Motivation . . . 3

1.2 Research goal . . . 5

1.3 Document overview . . . 6

2 Review of Literature 8

2.1 Brief history . . . 8

2.2 What is a CAVE? . . . 10

2.3 ParaView . . . 12

2.3.1 ParaView Compression Parameters . . . 14

2.4 The X Window System . . . 17

2.5 Virtual Network Computing (VNC) . . . 18

2.5.1 TurboVNC 19
2.5.2 TurboVNC Compression Parameters 20
2.6 VirtualGL 23
2.6.1 VirtualGL compression parameters 26
2.7 Ganglia 27

3 Design and implementation **30**
3.1 Specifications of the hardware used 30
3.2 Experimental setup 32
3.3 Data used for running the experiments 36
3.4 Limitations of ParaView in a CAVE 36
3.4.1 Big data in CAVE 36
3.4.2 Stereo rendering in CAVE 38
3.5 Interacting with data and ParaView's timer log 38

4 Results **40**
4.1 Frame rates 41
4.1.1 Using 8 VTU files and 8 CPUs 43
4.1.2 Using 16 VTU files and 16 CPUs 45
4.2 Network usage 46
4.2.1 Using 8 VTU files and 8 CPUs 47

4.2.2 Using 16 VTU files and 16 CPUs 49
4.3 Memory Usage 51
4.3.1 Using 8 VTU files and 8 CPUs 51
4.3.2 Using 16 VTU files and 16 CPUs 54

5 Discussion **56**
5.1 Welch Two Sample t-tests 56
5.2 ANOVA Analysis 58
5.3 Major takeaways 59
5.4 Effect of network and stereo 60

6 Conclusion and future work **62**
6.1 Recommendations 63
6.2 Future work 64

Bibliography **66**

Appendices **75**

Appendix A Experimental results **76**

List of Figures

1.1 World oil consumption statistics by country in a tabular form [56] 2

1.2 World oil consumption statistics by country in a treemap form [56] 2

1.3 Remote rendering overview [25] 4

1.4 Network bandwidth is a bottleneck in remote client-server architecture [24] . 6

2.1 Airflow model being analyzed using InstantReality [27] in the The Hypercube CAVE at Virginia Tech 11

2.2 Different datasets that ParaView and VTK can handle. The upper-left dataset is a uniform rectilinear volume of an iron potential function. The upper-right image shows an isosurface of a nonuniform rectilinear structured grid. The lower-left image shows a curvilinear structured grid dataset of airflow around a blunt fin. The lower-right image shows an unstructured grid dataset from a blow-molding simulation [49]. 13

2.3 ParaView architecture overview [31] 14

2.4 Immersive ParaView architecture overview [29] 14

2.5 ParaView compression parameters [31] 15

2.6 Overview of how X server works [78] 18

2.7 VNC architecture [68] 19

2.8 Image quality factors [45] 21

2.9 The VGL transport with a remote 2D X server [67] 25

2.10 The X11 transport with an X proxy [67] 26

2.11 Ganglia monitoring the client machine running ParaView client 29

3.1 Overview of the experimental setup to run experiments 33

3.2 Flowdigram of the experimental setup 35

3.3 MPI error message with ParaView when visualizing big data in a CAVE . . 37

4.1 Average still render FPS with 8 VTU files and 8 CPUs 44

4.2 Average interactive render FPS with 8 VTU files and 8 CPUs 44

4.3 Average still render FPS with 16 VTU files and 16 CPUs 45

4.4 Average interactive render FPS with 16 VTU files and 16 CPUs 45

4.5 Maximum data received with 8 VTU files and 8 CPUs 47

4.6 Average data received with 8 VTU files and 8 CPUs 48

4.7 Average data sent with 8 VTU files and 8 CPUs 48

4.8 Maximum data received with 16 VTU files and 16 CPUs 49

4.9 Average data received with 16 VTU files and 16 CPUs 50

4.10 Average data sent with 16 VTU files and 16 CPUs 50

4.11 Maximum client memory used with 8 VTU files and 8 CPUs 52

4.12 Average client memory used with 8 VTU files and 8 CPUs 52

4.13 Maximum server memory used with 8 VTU files and 8 CPUs 53

4.14 Maximum client memory used with 16 VTU files and 16 CPUs 54

4.15 Average client memory used with 16 VTU files and 16 CPUs 55

4.16 Maximum server memory used with 16 VTU files and 16 CPUs 55

A.1 Combination of parameters used to run the experiments 77

A.2 Results of the experiments 78

List of Tables

3.1 Hardware configuration of the host on NewRiver 30

3.2 Hardware configuration of Hypercube 31

3.3 Software versions used in the experiment 31

3.4 Specifications of the Projectors 31

4.1 Main groups based on JPEG, quality, and JPEG chrominance sub-sampling 43

4.2 Different compression parameters used in the experiments 43

5.1 Groups created for Welch Two Sample t-tests 56

5.2 Tests performed on different variables 57

5.3 Significant results obtained from the t-tests 57

5.4 Groups created for ANOVA for the entire data 58

5.5 Significant results obtained from ANOVA for all the data 58

5.6 Sub-groups created for ANOVA 59

5.7 Significant results obtained from ANOVA of sub-groups 59

Chapter 1

Introduction

Visualization has been described in [13] as "the use of computer-supported, interactive visual representations of data to amplify cognition." If there are large quantities of data to be visualized, typically, humans tend to perform poorly in reading and analyzing the data when presented in a numerical form compared to when presented in a visual form. Consider the pictures shown in figures 1.1 and 1.2. Figure 1.1 shows the data in a tabular form where the countries are listed in decreasing order of oil consumption in 2011. The table is longer than shown in the figure and contains hundreds of countries. The same data is used to build a treemap [75] in figure 1.2. It is clear from the two figures that one can quickly see and understand the oil consumption pattern of the counties from the second figure compared to the first one. The second figure is also interactive in nature in the sense that when a mouse pointer is hovered over any box in the treemap, it shows more details about that country. This can be attributed to the Shneiderman's principle that states "Overview first, zoom and filter, then details-on-demand" [52].

As a matter of fact, in 2013, [50] quoted that 90% of the data in the world has been generated in the last two years and 2.5 billion GB of new data is generated on a daily basis.

Rank	World Oil Consumption	Oil Consumption	%Oil Consumption
1	United States	18840000	21.58
2	China	9790000	11.22
3	Japan	4464000	5.11
4	India	3292000	3.77
5	Russia	3196000	3.66
6	Saudi Arabia	2817000	3.23
7	Brazil	2594000	2.97
8	Germany	2400000	2.75
9	South Korea	2301000	2.64
10	Canada	2259000	2.59
11	Mexico	2133000	2.44
12	France	1792000	2.05

Figure 1.1: World oil consumption statistics by country in a tabular form [56]

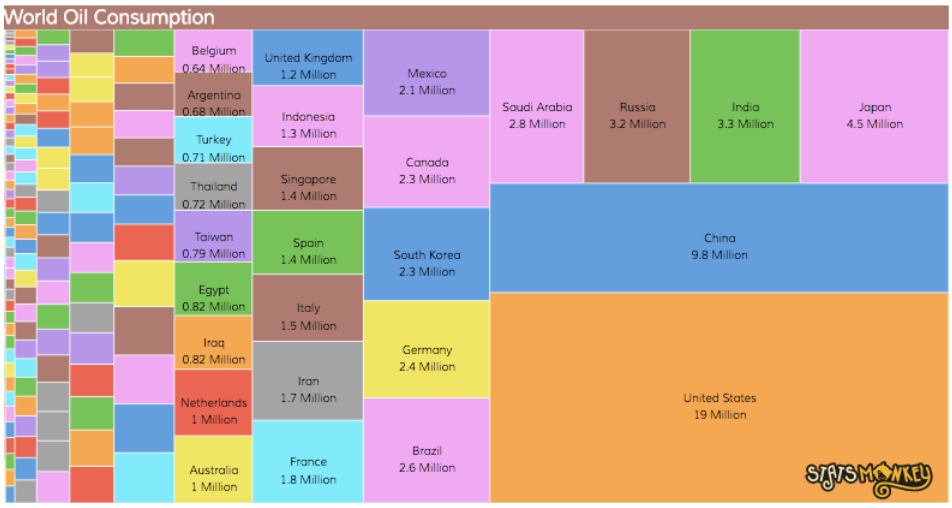


Figure 1.2: World oil consumption statistics by country in a treemap form [56]

With so much of data being generated, there is a greater need to come up with meaningful visualizations that help to better understand and analyze the data and see patterns, if any.

Visualizing data can have its own set of challenges. Sometimes the data contains noise that needs to be cleaned before visualizing it. Combining visual representations with textual labels can be a challenge too. Some other challenges can include finding related information, collaboration, viewing large data on a small screen, integrating data mining, achieving universal usability, and evaluation [14].

1.1 Motivation

One of the big challenges of big data visual analytics is rendering of the data. Rendering is a computationally expensive task that can take from a few hours to several days to render the data. Rendering can be thought of as the process to generate images from 2D or 3D models using powerful servers running some graphics algorithms. For example, think of a scene file that contains the model of a spherical ball. A user can define the geometry, colour, viewpoint, texture, lighting and other similar characteristics of the ball. All this metadata about the ball is then passed to a rendering algorithm that converts this information to a digital image. These images can be referred to as frames. A single scene file can generate multiple images or frames depending on the viewpoints.

To give an estimate of how computationally challenging rendering can be, when the animated movie "Cars 2" was released, Pixar stated that had to use 12,500 cores for rendering the entire movie, and each frame took an average of 11.5 hours to render [62]. Another example is that of the movie called "Big Hero 6" by Disney. Disney created a new renderer called the Hyperion renderer that basically allowed the creators to render with Global Illumination, which meant that the light sources would bounce around multiple times and behave much closer to actual light in the real world. Hyperion is run on a 55,000 core supercomputer that spans four geographically separate sites. This supercomputer produced 1.1 million core-hours each day totalling to 199 million core-hours that led to the creation of the movie [12].

Rendering can be challenging in terms of computation but the examples we discussed above needed pre-rendering and not dynamic rendering. Dynamic or on-demand rendering refers to rendering something in real time when there are no pre-defined frames and it is up to the user visualizing the data to change viewpoints as desired. Most of the video games are examples of dynamic rendering where a user playing the game can change viewpoints and

geometry every second and new frames have to be generated in real time and shown to the user. A combination of different frames shown every second gives a false perception of motion, and this is essentially what happens in movies or games. Frame rate can be referred to as the number of frames generated per second (see section 4.1 for details on what a good frame rate is). If the frame rate is not high enough, the user will be able to see/feel a lag and this adversely affects user experience. Thus, we need a way to render frames interactively for a good user experience and this can be done with sufficient amount of hardware resources that includes CPUs, GPUs, and memory.

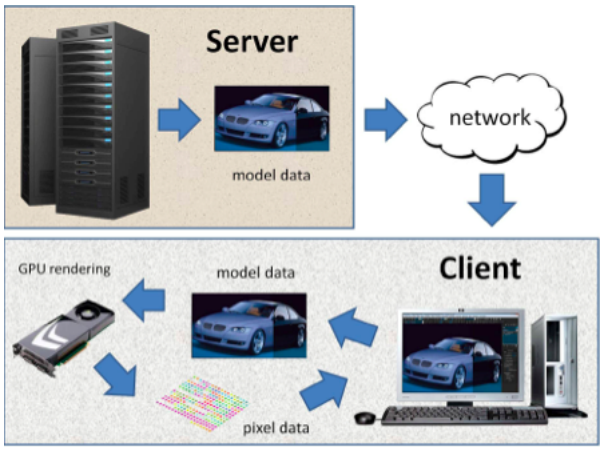


Figure 1.3: Remote rendering overview [25]

It is not typical to have a local access to a large pool of resources containing hundreds of CPUs and GPUs. Keeping that much computational resources locally will be very expensive and difficult to maintain. However, it is more feasible to have access to a large pool of remote cloud servers using services like AWS [9], Google Compute [23], Azure [34], etc. or a remote high performance computing cluster. Offloading the heavy computational tasks to these remote servers means that the client can be as thin as possible only running the application and getting the rendered images back from the remote servers. But this architecture is dependent on how efficiently can data be sent and received from the remote servers. In this architecture, the user experience will be dependent on the time taken to send the data to

the remote servers plus the time taken to render the data on the remote servers plus the time taken for the rendered data to be transferred back to the client. Figure 1.3 shows a high-level overview of how remote rendering works.

1.2 Research goal

Network is one of the biggest bottlenecks of remote rendering and thus, we want to minimize the data sent to the server (see figure 1.4). The most commonly used method to do this is to compress the data. Compression will make the data size smaller and the time taken to send this data from the client to the server and vice-a-versa will decrease as well. That said, compression has its own disadvantages too. Compression and decompression consumes CPU time meaning that some extra time will be added to the rendering process. Another disadvantage is that compression can lead to loss of data depending on what type of algorithms are being used (lossy versus lossless algorithms).

Apart from compression, there are other factors that can help minimize network bandwidth usage at the cost of reduced render quality like JPEG chrominance sub-sampling and the amount of compression level applied (see section 2.5.2 for details).

In this work, we are trying to render big 3D data (40+ million points) on the remote HPC clusters at Virginia Tech using a dedicated 10 GB research network called VT-RNET and visualizing them in a CAVE-style system (for details on CAVE, refer to section 2.2) that can support over 26 million pixels. From a high-level, our pipeline consists of ParaView in a CAVE mode (see section 2.3 for details), VirtualGL (see section 2.6 for details), and TurboVNC (see section 2.5.1 for details).

We are trying to find out what are the different compression parameters that we can vary

supported by TurboVNC that can help us reduce network traffic and at the same time provide interactive frame rates. The compression parameters discussed above can help reduce network bandwidth and the ultimate goal is to make sure that the users visualizing the big 3D data in a CAVE-style system have a good experience and do not feel a lag. However, getting interactive framerates is not the only goal since the user experience will not be good even at interactive frame rates if the render quality is bad.

Apart from the network traffic, we measure the effects of compression on CPU and memory consumed at the client and the server side. We describe our setup and the elements in our pipeline in detail in chapter 3.

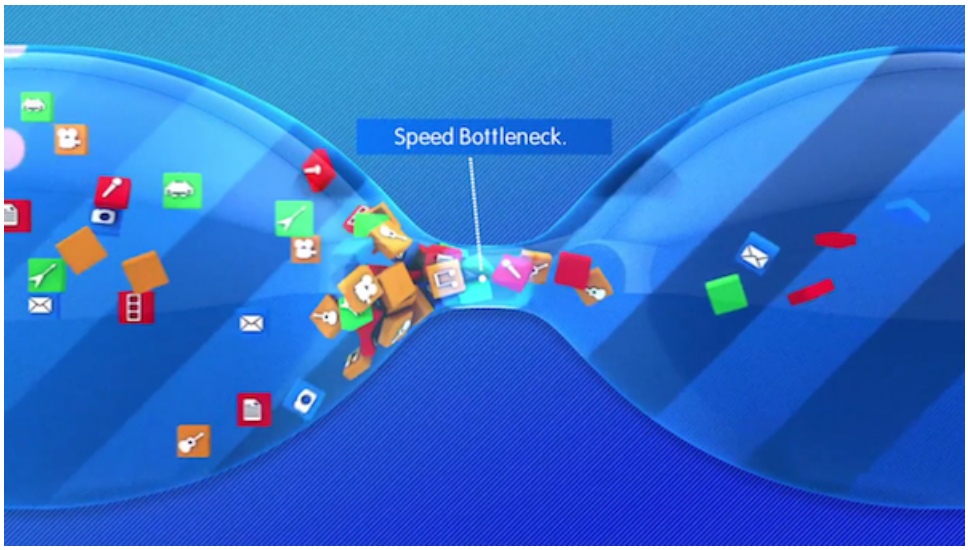


Figure 1.4: Network bandwidth is a bottleneck in remote client-server architecture [24]

1.3 Document overview

In chapter 2, we review some work done in the field of remote big data visualization (including the challenges) along with talking about the technologies used in our pipeline like ParaView, TurboVNC, VirtualGL, and Ganglia. In chapter 3, we layout our experimental pipeline

consisting of different elements and present some architecture diagrams. We also talk about some limitations of immersive ParaView in this chapter. Chapter 4 presents all the results we got and show it in the form of graphs along with t-tests and ANOVA evaluation. Chapter 5 discusses in detail about the results we got in chapter 4 and what meaningful results can be inferred from it. Finally, chapter 6 concludes this document and talks about future work.

Chapter 2

Review of Literature

2.1 Brief history

Graphics rendering pipeline consists of two phases - geometry processing and b) rasterization. The authors in [35] have described three classes of parallel rendering algorithms; sort-first, sort-middle, and sort- last. In sort-first, the primitives or the objects in the scene are sorted early in the rendering pipeline - during geometry processing. The biggest advantage of sort-first is low communication requirements when the tessalation ratio and the degree of oversampling is high. In sort-middle algorithm, the primitives are sorted after the geometry-processing phase but before rasterization. In sort-last algorithm, the primitives are sorted after the rasterization but before the scene display. A significant advantage of the sort-last algorithm is that the renderers implement the complete rendering pipeline and are independent until pixel merging. In general, all the three rendering algorithms suffer from some common problems like load balancing, high processing and communication costs, and high pixelated traffic. ParaView [31] implements sort-last algorithm in its code.

Samanta et al. [47] discuss a hybrid of sort-first and sort-last parallel polygon rendering algorithm. The hybrid algorithm outperforms sort-first and sort-last algorithms in terms of efficiency. Overall, the hybrid algorithm achieved interactive frame rates with efficiencies of 55% to 70.5%. during simulation with 64 PCs. The hybrid algorithm provides a low cost solution to high performance rendering of 3D polygonal models.

Eilemann et al. [18] introduced a system called Equalizer, a toolkit for parallel rendering based on OpenGL. Equalizer also provides an API to develop graphics applications. Equalizer takes care of distributed execution, synchronization, and final image composting, while the application programmer identifies and encapsulates culling and rendering. This approach is minimally invasive since the proprietary rendering code is retained. Using Equalizer for rendering does not impose any restriction on how the application handles and accesses the data to be visualized. Equalizer does not facilitate parallel data access and distribution. Other parallel rendering algorithms like sort-first and sort-last can also be implemented using Equalizer. A disadvantage of Equalizer is that it does not solve the problem of load balancing, which is an important issue with parallel rendering.

Moreland et al. [37] used sort-last parallel rendering algorithm to render data to large tile displays. They mention that because the system does not replicate input data amongst the processors or transfer data between processors, it can handle very large data sets and support display resolutions with bigger rendering clusters. They also describe some optimization techniques that can be used to render and compose the images in an interactive application.

Ahrens et al. [7] talk about the challenges of visualizing huge datasets and propose that a simple solution to this problem is dividing the data into smaller pieces and streaming these smaller pieces through memory and processing them. The authors present a visualization architecture that uses advanced culling and prioritization features based on data and spatial location and also enable advanced rendering features such as view dependent ordering and

occlusion culling. The paper shows improvements for real-world visualization programs. [5] also talks about similar problems with respect to large-scale data visualization and presents an architectural approach based on mixed dataset topology parallel data streaming. This enables high code reuse and and consumes less memory to visualize big data sets.

Morland et al. [36] described a new set of parallel rendering components for the Visualization Toolkit (VTK) [49]. They introduced components of Chromium [26] and ICE-T (an implementation of [37]) into VTK and showed that VTK can be a viable framework for cluster-based interactive applications that require remote display.

2.2 What is a CAVE?

A CAVE automated virtual environment [17] is a virtual reality interface that consists of walls and each wall can be driven by one or more projectors. The first CAVE was developed at the University of Illinois, Chicago Electronic Visualization Laboratory in 1992 for scientific and engineering applications and overcome the limitations of head mounted displays (HMD). A CAVE allows multiple users to share the virtual environment to examine and interact with complex 3D models. A CAVE does not necessarily have a fixed number of walls. The CAVE at Virginia Tech in the Visionarium Lab - The Hypercube [61] - is a 4-walled CAVE driven by 2 projectors on each side of the wall. Figure 2.1 shows the CAVE at Virginia Tech.

Some of the advantages of a CAVE over HMDs include more pixels per screen, 170 degree field of view compared to 100 for HMDs, better sense of presence, wireless tracking, supports multiple users, light 3D glasses, less prone to simulator sickness, etc. One big disadvantage of a CAVE is its very high build cost while HMDs cost little.

CAVE2 [21] introduced LCD panels in stead of projectors and unlike the original CAVEs,

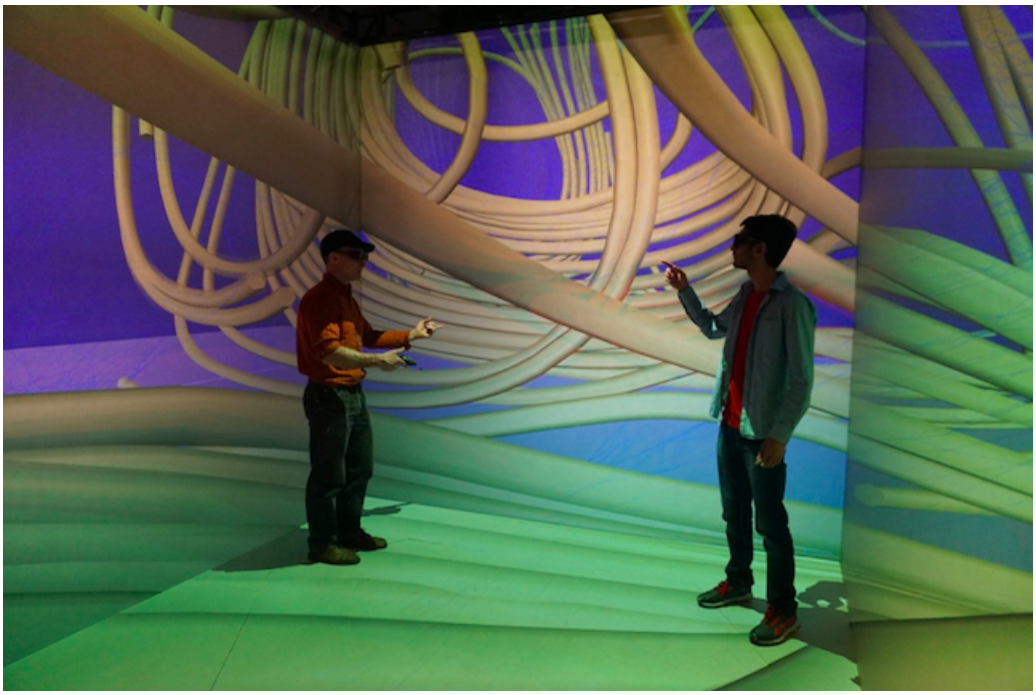


Figure 2.1: Airflow model being analyzed using InstantReality [27] in the The Hypercube CAVE at Virginia Tech

they can support multiple operating modes - 2D, 3D, or both. An advantage of CAVE2 is that compared to buying projectors, LCD panels are cheap and also easier to maintain. LCD panels also provide superior image quality and resolution. The first CAVE2 was built at the University of Illinois at Chicago and it used passive stereo along with an audio system.

A similar visualization facility called "Reality Deck" was developed at the Stony Brook University [42]. "Reality Deck" can support up to 1.5 giga pixels immersive tiled display capable of a 360 degree horizontal field of regard. It is a rectangular power wall configuration and the authors mention that they chose a rectangular shape over a square shape that is generally used by CAVE like systems because a rectangular shape adds more operational flexibility and allows the use of most of the available empty spaces in their lab. "Reality Deck" can offer more pixels per inch compared to a typical CAVE meaning that more data can be displayed. Like CAVE2, it also makes use of LCD displays.

Another immersive instrument called "The AlloSphere" [8] was developed at the University of California, Santa Barbara. It consists of a spherical space that can be used to visualize and explore huge quantities of data through multimodal interactive media. AlloSphere is capable of seamless stereo-optic 3D projection and does not distort the projected content due to room geometry. It has its own server farm for video and audio processing and a low-latency interconnection fabric to process data on multiple nodes in real time. It can accommodate several dozen people at a time.

2.3 ParaView

ParaView [6][31] is an open source toolkit that allows scientists to visualize large datasets. It supports the visualizations and rendering of big data by executing these programs in parallel on shared or distributed machines. It supports hardware-accelerated parallel rendering and also provides a graphical user interface for the creation and dynamic execution of visualization tasks.

ParaView is based on the visualization toolkit called VTK [49] that provides the data representations for a variety of grid types including structured, unstructured, polygonal, and image data (see figure 2.2). VTK also provides algorithms and visualizations to process the data including isosurfacing, cutting/clipping, glyphing, and streamlines. VTK is written using C++ but also offers scripting interfaces for Java, Python, and Tcl.

ParaView extended VTK to support streaming of all data types and parallel execution on shared and distributed-memory machines. Both data streaming and parallel computation depends on the ability to break a dataset into smaller pieces. Data streaming incrementally processes the smaller pieces one at a time and thus, a user can process a large dataset with computing resources that may not store the entire dataset in memory or on the disk.

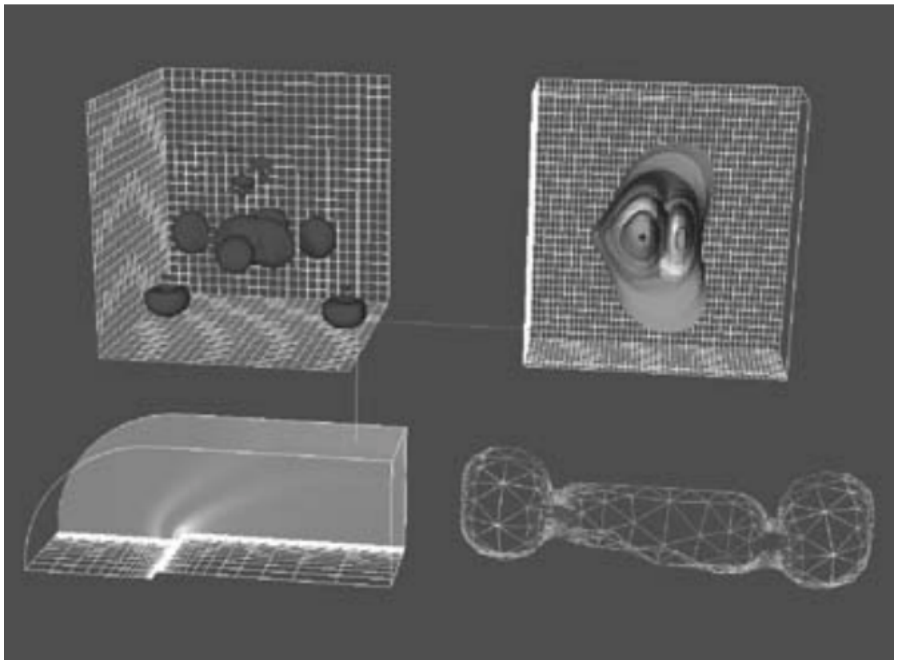


Figure 2.2: Different datasets that ParaView and VTK can handle. The upper-left dataset is a uniform rectilinear volume of an iron potential function. The upper-right image shows an isosurface of a nonuniform rectilinear structured grid. The lower-left image shows a curvilinear structured grid dataset of airflow around a blunt fin. The lower-right image shows an unstructured grid dataset from a blow-molding simulation [49].

ParaView can support parallelism using shared-memory processes or distributed-memory processes (using MPI [72]). Running ParaView in parallel makes the data read, data processing, and data render in a data-parallel fashion. ParaView's parallel architecture includes three types of processes; a client process to run the GUI and two types of server processes namely a data server and a render server. If data and render server are running on the same machine, they are collectively called a *pvserver* (figure 2.3 shows a high-level view of the ParaView architecture).

ParaView can also be used to drive immersive displays like a CAVE-styled environment [29] (see section 2.2 for details on what is a CAVE). ParaView does this by passing a *pvx* or an XML file at the server side that contains information about the displays, the coordinates of each display (bottom left corner, bottom right corner, and top right corner), and an eye

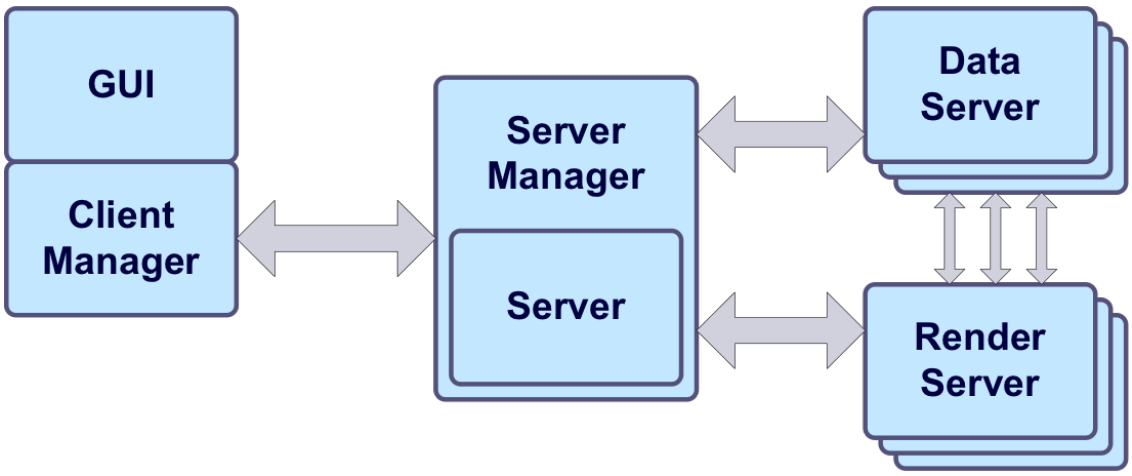


Figure 2.3: ParaView architecture overview [31]

separation value. To enable head and wand tracking, VRPN [59] or VRUI [64] can be used. See figure 2.4 for the architecture overview.

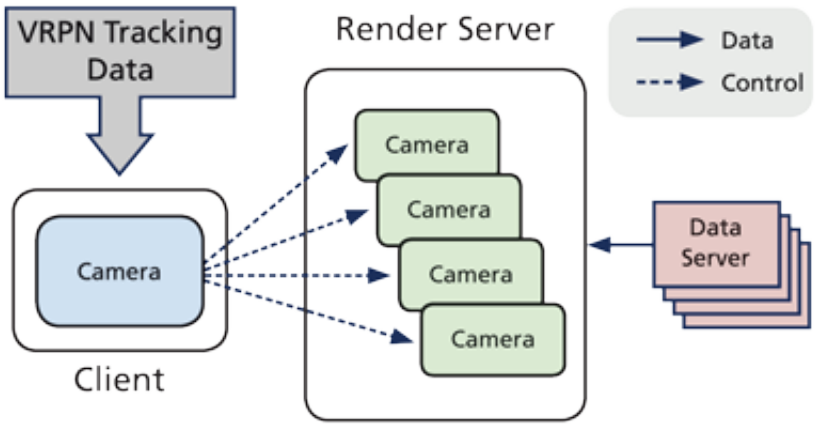


Figure 2.4: Immersive ParaView architecture overview [29]

2.3.1 ParaView Compression Parameters

ParaView supports several compression parameters when rendering data remotely [43]. The compression of data essentially helps in reducing network bandwidth but also means extra CPU computation in the compression and decompression steps. Figure 2.5 shows the various

options that can be set.

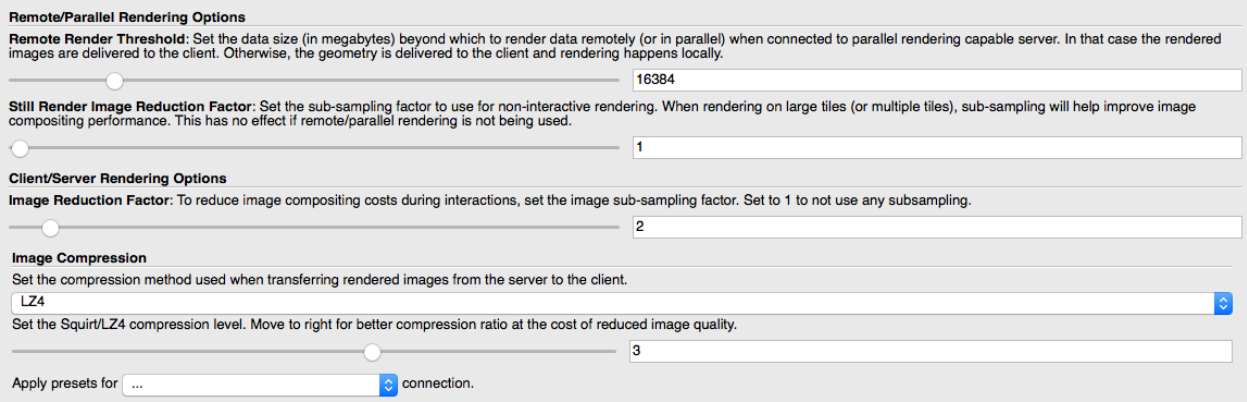


Figure 2.5: ParaView compression parameters [31]

1. *Remote render threshold*: This property defines the data size above which the data would be rendered remotely and below which the server will send all the geometry to the client and the client will do the rendering. For example, if we have a small data of 20 points, it does not make sense to send the data across a network to render it since the time it would take to send and receive the rendered data from the remote server will surpass the time taken to render the data on the local machine. But if the data is big, the overhead of sending and receiving the data will still be less than the time consumed to render the data locally.

2. *Still render image reduction factor*: This option is used to down-sample the rendered images before sending them from the server to the client for non-interactive rendering. But down-sampling also reduces the quality for the rendered images and ParaView suggests not to exceed a 3 pixel setting. If the network bandwidth is sufficient, turning off this option is recommended.

3. *Image reduction factor*: Unlike the still render image reduction factor, this option can be set to reduce image composting costs during interactions. A value of 1 would mean to not use any sub-sampling.

4. *Image compression*: ParaView supports 3 compression algorithms:

   (a) *LZ4 compression*: LZ4 [32] is a lossless data compression algorithm and belongs to the LZ77 family of byte-oriented compression schemes. It can provide compression speeds at 400 MB/sec per core and is scalable with multi-core CPUs. Diffrent levels of LZ4 compression can be set and a higher level means reduced image quality.

   (b) *Sequential Unified Image Run Transfer (SQUIRT) Compression*: SQUIRT [65] is a run-length-encoded (RLE) compression algorithm developed at Sandia National Lab. It is a fast algorithm but achieves relatively low compression ratios compared to other algorithms. It is most effective on fast networks since its strength is its speed. It is fast because in addition to being a RLE algorithm, it further improves its speed by performing operations on RGB triplets with single 32-bit integer operations [15]. In ParaView, we can also set the compression levels for SQUIRT and a higher level means better compression ratio at the cost of reduced image quality.

   (c) *ZLIB Compression*: ZLIB [3] is lossless open source data-compression library that ParaView provides as one of the 3 compression algorithms. ZLIB has a fixed memory footprint independent of the input data size. It can achieve compression ratios an order of magnitude higher than SQUIRT but speed an order of magnitude slower than SQUIRT. ParaView provides two parameters that can be set with the ZLIB compression. The first one is the compression level and higher the compression level, the slower it will be. The second parameter is the ZLIB colour sampling space width factor and a higher number means better compression ratio at the cost of reduced image quality. It is suggested that ZLIB should be used with low network bandwidth [51].

In addition to the above parameters, ParaView also provides some pre-set configurations for different network types ("consumer broadband/DSL", "Megabit Ethernet", "Gigabit Ethernet", "10 Gigabit Ethernet", or "shared memory/localhost"). Selecting one of them automatically sets the compression parameters to suitable values as suggested by ParaView. By default, Remote Render Threshold is set to 20, Still Render Image Reduction Factor is set to 1, Image Reduction Factor is set to 5, Image Compression is set to LZ4 and the compression level is set to 3.

All the compression parameters provided by ParaView can be altered using the GUI but we did not find a way to script them. For this reason, we did not use ParaView's compression parameters for our tests and instead used TurboVNC's compression parameters as discussed in section 2.5.2.

2.4 The X Window System

The X Window System [48] was developed at MIT and is a windowing system for graphics workstations. It follows a client server model in that a host machine runs an X server and the client machines request services from the server. The server is responsible for handling input and output devices and generating the graphical displays to be used by the clients. The X server allows windows to be moved, re-sized, and manipulated with a mouse.

The X.Org project [79] provides an open source implementation of the X Window System. The X.Org Foundation is the educational non-profit corporation that leads the development of this work. Xorg is also the most popular display server among Linux users. Figure 2.6 shows from a high level how X server works and interacts with the different X clients, some of which may be remote.

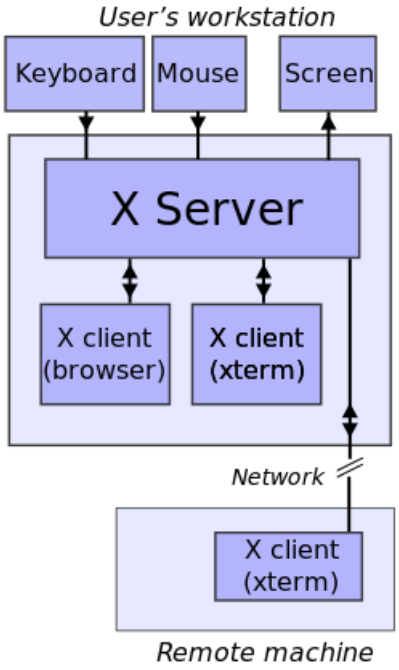


Figure 2.6: Overview of how X server works [78]

2.5 Virtual Network Computing (VNC)

The X Window System lets an application show a user interface on a remote machine. Oracle Research Laboratory (ORL) extended this functionality in their Teleporting System that allows the user interface of a running X application to be dynamically redirected to another display [46]. In virtual network computing (VNC) system, the server applications not only provide applications, data, and storage for a user's preference but also provide an entire desktop environment that can be accessed from any machine on the Internet using a software network computer. Members of ORL used VNC to access their personal Unix and PC desktops from any office in their Cambridge building and from around the world. VNC also allows a single desktop to be accessed from several places simultaneously.

The technology under VNC is a simple remote display protocol that is independent of the operating system or a windowing system. VNC works at the framebuffer level. A framebuffer

refers to the portion of video card memory that is reserved for holding the complete bit-mapped image that is sent to a display monitor. In some cases, the video chipset is integrated into the motherboard design and the framebuffer is stored in the main memory. A VNC server is the endpoint where changes to the framebuffer originate while a VNC viewer is the endpoint with which a user interacts. Figure 2.7 shows the VNC architecture overview.

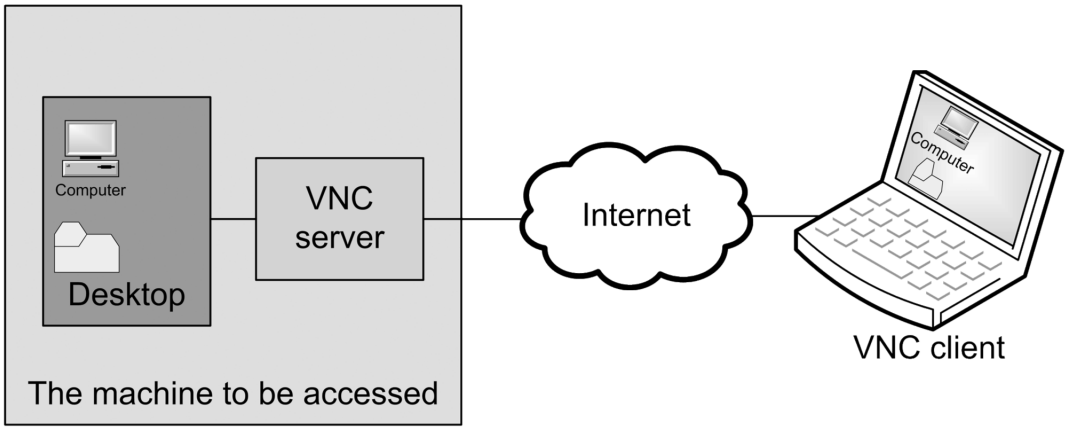


Figure 2.7: VNC architecture [68]

2.5.1 TurboVNC

TurboVNC is a derivative of VNC that has been specifically developed to provide high performance for 3D and video workloads [63]. It is an open source project that contains a modern X server code base (X.org 7.7). Like all other VNC implementations, it also uses the remote framebuffer (RFB) protocol to send framebuffer updates from the server to the viewer and each framebuffer update can contain multiple rectangles. TurboVNC splits these rectangles into multiple sub-rectangles and attempts to encode each sub-rectangle using the "subencoding type" that will provide the most efficient compression. The process by which TurboVNC does this is called "encoding method".

TurboVNC analyzes a rectangle to determine if any portion of it is solid and if so, that portion

is encoded as a bounding box and a fill colour ("Solid subencoding"). Sub-rectangles with only two colours are encoded as a 1-bit-per-pixel bitmap with a 2-colour palette ("Mono subencoding"). The sub-rectangles with a low number of unique colours are encoded as a colour palette and an 8-bit-per-pixel bitmap ("Indexed colour subencoding") and the sub-rectangles with a high number of unique colours are encoded using either JPEG or arrays of RGB pixels ("Raw subencoding"). ZLIB [3] can also optionally be used to compress the indexed colour, mono, and raw sub-rectangles.

TurboVNC uses libjpeg-turbo [1] which is a JPEG image codec that uses SIMD instructions to accelerate baseline JPEG compression and decompression on x86, x86-64, and PowerPC systems. TurboVNC also efficiently manages the smoothness detection routines that determine if a sub-rectangle is a good candidate for JPEG compression and thus, it reduces the CPU usage. Another advantage of using TurboVNC is that it eliminates buffer copies and maximizes network efficiency by splitting framebuffer updates into larger sub-rectangles and uses ZLIB compression levels that can be shown to have a measurable performance benefit.

2.5.2 TurboVNC Compression Parameters

TurboVNC supports difference compression parameters that can be set to achieve high image quality and high performance. There are 4 primary options that control how TurboVNC will compress the images [63].

1. JPEG compression: JPEG compression is explained well with an example in [33].

   (a) Take an image and divide it into 8-pixel by 8-pixel blocks. If the image can not be divided then we can add empty pixels around the edges and this process is called zero padding.

(b) Get image data for each of the 8-by-8 blocks such that you have values to represent colour at each pixel.

(c) Take the Discrete Cosine Transform (DCT) [4] of each block.

(d) Matrix multiply the block by a mask that will zero out certain values from the DCT matrix. Take the inverse DCT of each block to get the data for the compressed image. When all these blocks are combined, they make up the original image of the same size.

When the JPEG compression is enabled in TurboVNC, it uses it for sub-rectangles that have a high number of unique colours, and it uses the indexed colour subencoding for sub-rectangles that have a low number of unique colours. When JPEG compression is disabled, TurboVNC selects between indexed colour or raw subencoding depending on the size of the sub-rectangles and its colour count [63].

2. JPEG image quality: Image quality refers to the perceived image degradation. Figure 2.8 shows some of the different factors that determine the quality of an image.

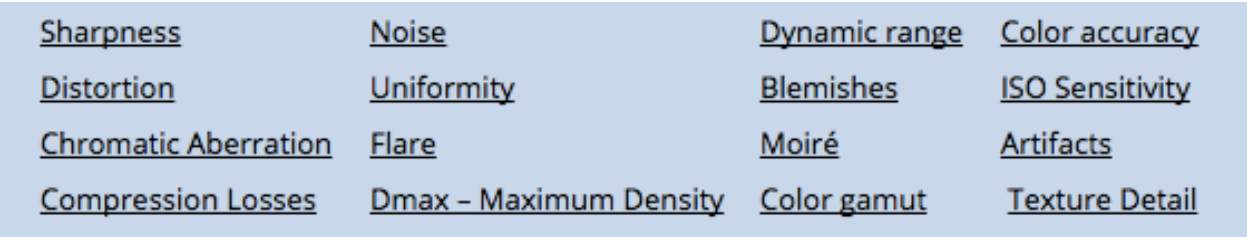


Figure 2.8: Image quality factors [45]

An image quality of 95 means that 95% of the original image quality is retained while 5% can be lost. In TurboVNC, a lower image quality produces grainier JPEG images but also uses less network bandwidth and computational time.

3. JPEG chrominance sub-sampling: Luminance (Y) refers to the brightness of light, Luma (Y') refers to luminance that has been applied with an encoding gamma, and

Chrominance (C) refers to the colour information in a signal [57]. CbCr refers to the blue-difference and red-difference chroma components.

In TurboVNC, when an image is compressed using JPEG, the RGB (Red, Green, Blue) pixels are converted into YCbCr colourspace. Chrominance sub-sampling is used to discard some of the chrominance components to save bandwidth. The authors of TurboVNC mention that this technique works since the human eye is more sensitive to changes in brightness than in colour [63]. A 2X sub-sampling will retain the chrominance components for every second pixel while a 4X sub-sampling will retain the chrominance sub-sampling for every fourth pixel. If "Grayscale" is used then that will remove all the chrominance from the image leaving only luminance.

The authors of TurboVNC also mention that it may be difficult to detect any difference between 1X, 2X, and 4X sub-sampling with photographic or other "smooth" image content.

4. Compression level: In TurboVNC, a given compression level specifies three things.

   (a) The amount of ZLIB [3] compression to be used with indexed colour, mono, and raw sub-rectangles. ZLIB performs a loss-less compression and finds the duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string in the form of a pair (distance, length) [20].

   (b) The "palette" threshold or the minimum number of colours that a sub-rectangle must have before it is encoded as JPEG or raw instead of indexed colour.

   (c) Whether interframe comparison should be used. Interframe comparison helps with certain applications that draw the same things time and again. This can lead to redundant framebuffer updates sent to the VNC viewer. The TurboVNC server can protect against this by maintaining a copy of the remote framebuffer for

each connected viewer comparing each new framebuffer update rectangle against the pixels in the framebuffer copy and discarding any redundant portions of the rectangle before they are sent to the viewer. Interframe comparison can lead to increased memory usage of the TurboVNC server by a factor of N, where N is the number of connected viewers. It is less common for 3D applications to generate duplicate frame buffer updates and hence, the benefits of interframe comparison is less for such applications. The authors of TurboVNC also mention that in real-world tests, interframe comparison rarely reduces the network usage for 3D applications by more than 5-10%. For these reasons, the interframe comparison is not enabled by default by TurboVNC and should not typically be used except on bandwidth-constrained networks and with applications that have shown to benefit from interframe comparison. In TurboVNC, interframe comparison can be enabled by either passing an argument of '-interframe' to 'vncserver' or by requesting a compression level of 5 or higher from the viewer [63].

2.6 VirtualGL

Typically, VNC [46] or other remote display environments do not support running OpenGL [53] applications and if they do, they force the OpenGL applications to use a slow, software-only renderer that detriments the performance. Indirect rendering uses the GLX [53] extension to the X Windows System (refer to section 2.4 for details) to encapsulate the OpenGL commands inside of the X11 protocol stream and ships them from an application to an X display. Indirect rendering supports 3D hardware acceleration but forces all the OpenGL commands to be sent over the network to be rendered on the client machine. This method has its own restrictions in that the network needs to be fast, data should not be big, and

the application should be tuned for the remote X windows environment.

VirtualGL [66][76] is an open source software that enables remote display software the ability to run OpenGL applications with full 3D hardware acceleration. It redirects the 3D rendering commands and the data of the OpenGL applications to a graphical processing unit (GPU) installed on the remote server and sends the rendered 3D images to the client. It also allows the GPUs on the remote server to be shared among multiple users. With VirtualGL, the clients do not need to have big processing power and the network does not have to be super fast since the users can visualize big data sets in real time on the remote server without needing to copy any data over the network.

When using an OpenGL application, all the 2D and 3D drawing commands and the data are sent to an X Windows server that may be remotely located. VirtualGL employs a technique called "split rendering" in that it forces the 3D commands and the data from the application to be processed on a GPU in the application server. It preloads a dynamic shared object into the OpenGL application at the run time and this dynamic shared object intercepts GLX, OpenGL, and X11 commands to perofrm split rendering. When an application attempts to use an X Window for the OpenGL rendering, VirtualGL intercepts the request and creates a 3D pixel buffer in the video memory on the application server and uses it for OpenGL rendering. Once the application swaps the OpenGL drawing buffers to indicate that it has finished rendering a frame, VirtualGL reads back the pixels from the 3D pixel buffer and sends them to the client.

VirtualGL is non-intrusive in nature and does not interfere with the delivery of 2D X11 commands to the server. It only forces the OpenGL commands to be delivered to a GPU that is attached to a different X server than the X server to which the 2D drawing commands are delivered.

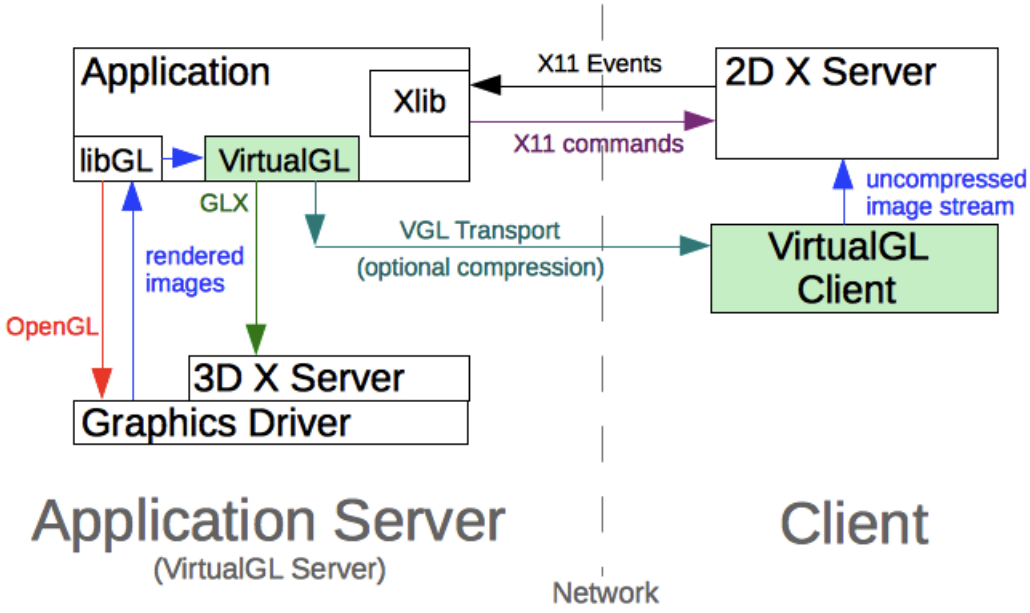


Figure 2.9: The VGL transport with a remote 2D X server [67]

VirtualGL supports two "image transports" to send the rendered 3D images to the client.

1. VGL transport: This is used when the 2D X server is located across the network from the application server and in this case, VirtualGL uses its own protocol on a dedicated TCP socket to send the rendered images to the client machine. The VirtualGL client application decodes the images and composites them into the appropriate X Windows. See figure 2.9 for the architecture.

2. X11 transport: This method draws the rendered 3D images into appropriate X Windows and is used along with an "X proxy" like VNC. The X proxies perform the X11 rendering to a virtual framebuffer in the main memory rather than to a real framebuffer. This enables the X proxy to send only images to the client machine rather than X Windows rendering commands.

   An important thing to note with this method is that when using the X11 transport method, VirtualGL does not perform any image compression but relies on the X proxy

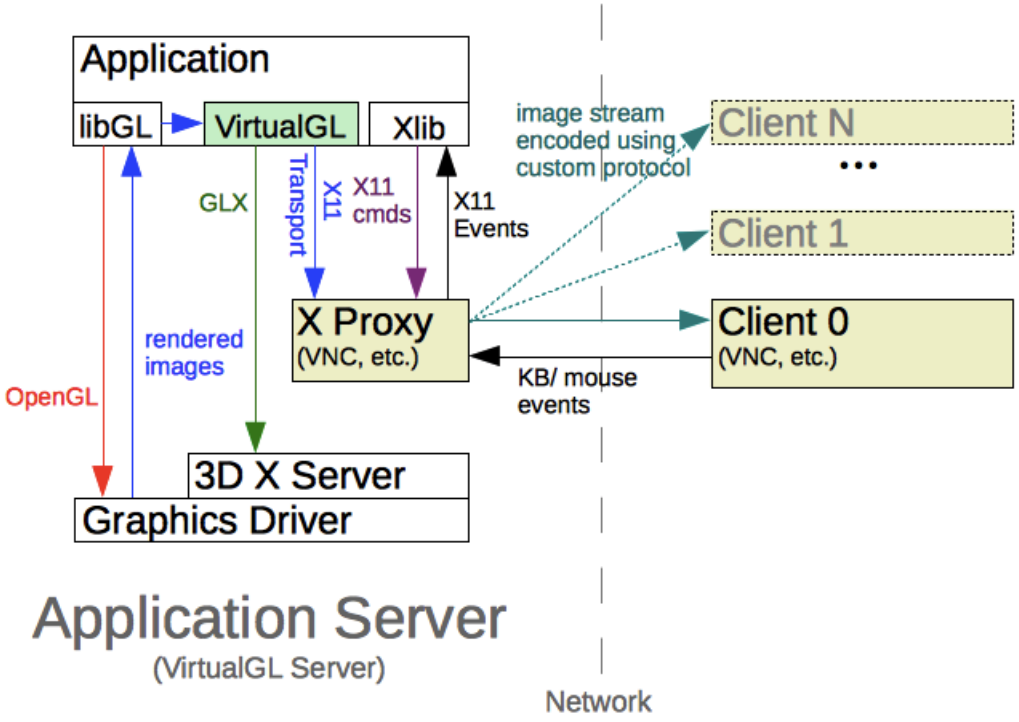


Figure 2.10: The X11 transport with an X proxy [67]

to encode and deliver the images to the client. We did not realize this initially when we started running our experiments and were trying to vary the compression parameters supported by VirtualGL. As expected, we did not observe any difference in the results that we got and this led us to explore the compression supported by TurboVNC instead (see section 2.5.1 for details). See figure 2.10 for the architecture.

2.6.1 VirtualGL compression parameters

When using the VGL transport to send the rendered 3D images to the client, VirtualGL supports several compression parameters that can help in reducing the network bandwidth usage. We briefly mention them below:

1. Image Compression: Various image compression methods like JPEG, RGB or YUV can be set.

2. Chrominance sub-sampling: When using the JPEG compression, we can set the chrominance sub-sampling values to 1x, 2x, or 4x. For more details on chrominance sub-sampling, refere to section 2.5.2.

3. JPEG image quality: When using the JPEG compression mehod, the image quality can be set from 1 to 100.

4. Connection profile: Depending on the network, connection profile can be set to "Low Qual", "Medium Qual", or "High Qual". Selecting the connection profile automatically sets appropriate compression parameters.

5. Multi-threaded compression: This parameters defines how many CPUs to be used for compression.

2.7 Ganglia

Ganglia [54] is an open source scalable distributed monitoring tool for high performance computing systems, clusters, and networks. It uses XML for data representation, XDR for data transport, and RRDtool [73] for data storage and visualization. Running Ganglia on a server has very low per-node overhead and high concurrency. It can scale to support up to 2000 nodes in a cluster. It consists of two main daemons, one running on the server and the other running on the client.

1. Ganglia monitoring daemon (gmond): Gmond is installed on each node that needs to be monitored in the cluster. It monitors the changes in the host state, announces

relevant changes, listens to the state of all other ganglia nodes via a unicast or a multicast channel, and answers requests for an XML description of the cluster.

2. Ganglia meta daemon (gmetad): This daemon runs on the main server hosting Ganglia. It periodically polls a collection of child data sources and collects information in XML format, parses it, and shows it on the Ganglia PHP web front end. The web front end shows metrics like CPU, load, memory, network, etc.

For the purpose of our tests, we installed Ganglia gmetad on a machine in the lab and installed gmond on the client machine on which ParaView client ran. We set a polling frequency of 3 seconds and collected all our metrics for the client using using this monitoring tool. Figure 2.11 shows a screenshot of Ganglia running on one of our lab machines monitoring "cube.sv.vt.edu", which was the client machine running the ParaView client.

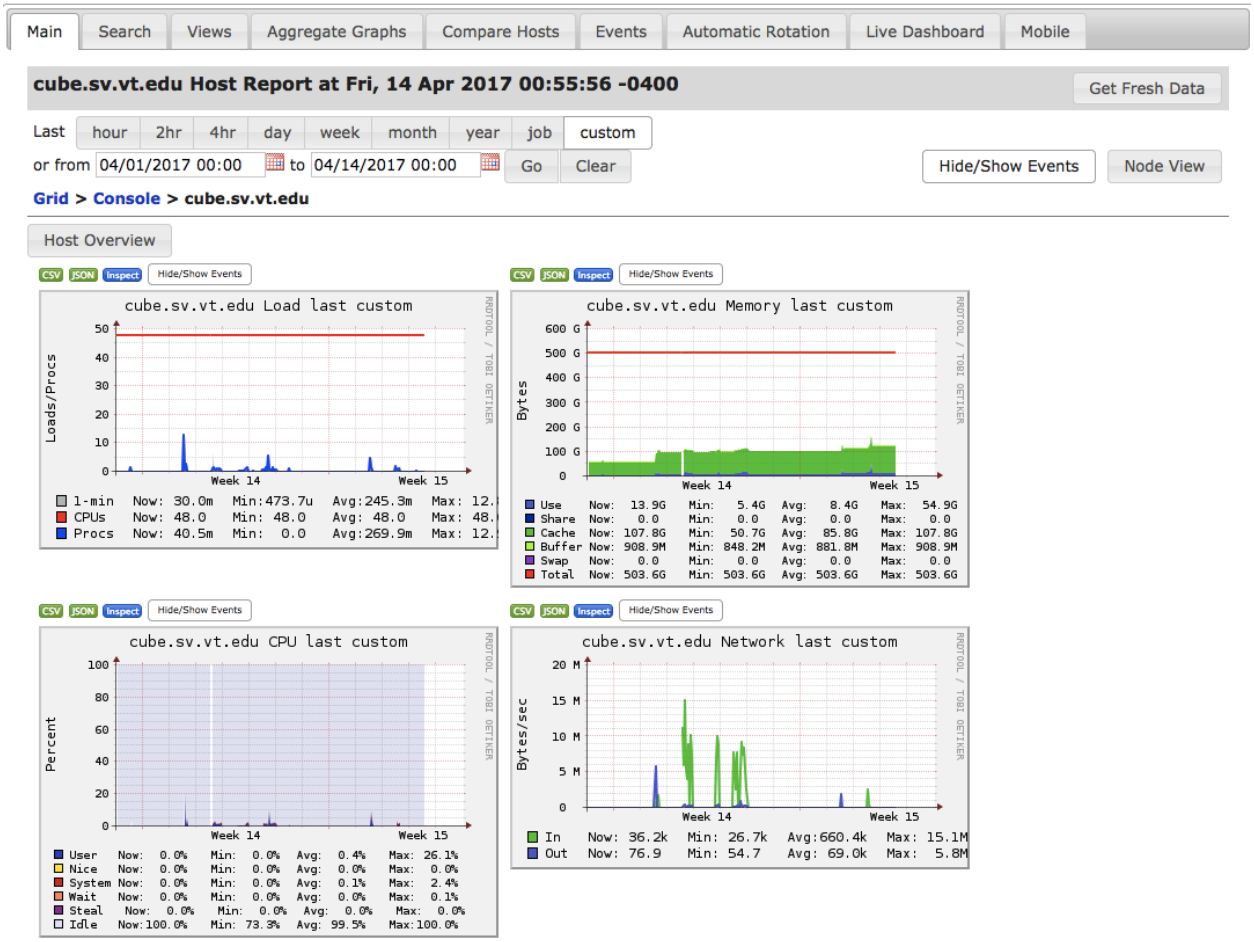


Figure 2.11: Ganglia monitoring the client machine running ParaView client

Chapter 3

Design and implementation

3.1 Specifications of the hardware used

We used the NewRiver HPC cluster at Virginia Tech for remote rendering. Table 3.1 shows the specifications of each host on NewRiver with a GPU. For running our experiments, we used 1 host on NewRiver. This host ran ParaView servers, VirtualGL, and VNC servers.

Table 3.1: Hardware configuration of the host on NewRiver

CPU Model	#CPUs	Memory	GPU
Intel(R) Xeon(R) CPU E5-2680 v3 2.50GHz	24	512 GB	Tesla K80

The specifications of the "Hypercube" machine that was used to run the ParaView client along with the VNC viewers is mentioned in table 3.2. The different software versions used in the experiments are mentioned in table 3.3. The specifications of the projectors used in the CAVE for these experiments are mentioned in table 3.4. To run our experiments, VT-RNET 10 GB network was used.

Table 3.2: Hardware configuration of Hypercube

CPU Model	#CPUs	Memory	GPU
Intel(R) Xeon(R) CPU E5-2680 v3 2.50GHz	48	512 GB	Quadro M6000

Table 3.3: Software versions used in the experiment

Name	Version
CentOS	v7.3.1611
ParaView	v5.2.0
VirtualGL	v2.5.1
TurboVNC	v2.1.1
Mpich	v3.1.4
Python	v2.7.10
Gmond	v3.7.2
Gmetad	v3.6.0

Table 3.4: Specifications of the Projectors

Model	Resolution	Frequency	Active Stereo	Contrast Ratio	Aspect Ratio
Barco F50	26,214,400 px	120 Hz	Enabled	5,300:1	16:10

3.2 Experimental setup

ParaView consists of a data server, render server, and a client. The data and the render server can be separate machines but if they are on the same host, they are referred to as a pvserver. In our experiments, we had the render and the data server running on the same machine acting as the pvserver.

We used NewRiver [16] for remote rendering and the Hypercube machine [60] for running the ParaView client. A vncserver was started on Hypercube and a user could use that vncsession to connect to Hypercube to run the experiments. Ganglia daemon running on Hypercube was set at a polling frequency of 3 seconds and used to collect metrics like memory and network usage.

Our experimental pipeline mainly consists of Paraview, TurboVNC, and VirtualGL. Figure 3.1 shows the architecture overview and the labels in the architecture have been described below. Figure 3.2 shows a flow diagram of the different components talking to each other.

1. In order to run ParaView in a CAVE mode, a pvx or an XML file containing all the DISPLAY numbers, geometry, and coordinates needs to be passed as an argument when starting the pvserver at the server side. In our case, we were using MPI [38] to start either 8 or 16 pvservers. Our pvx file can be found at [2] and depending on how many pvservers we started, we needed to have an equal number of DISPLAYs in the pvx file.

2. To enable remote rendering, an X server needs to be running on the remote server which is used by pvserver and VirtualGL. In the absence of an X server, ParaView will send back all the geometry to the client side for rendering and virtualgl will not be able to render anything on the server.

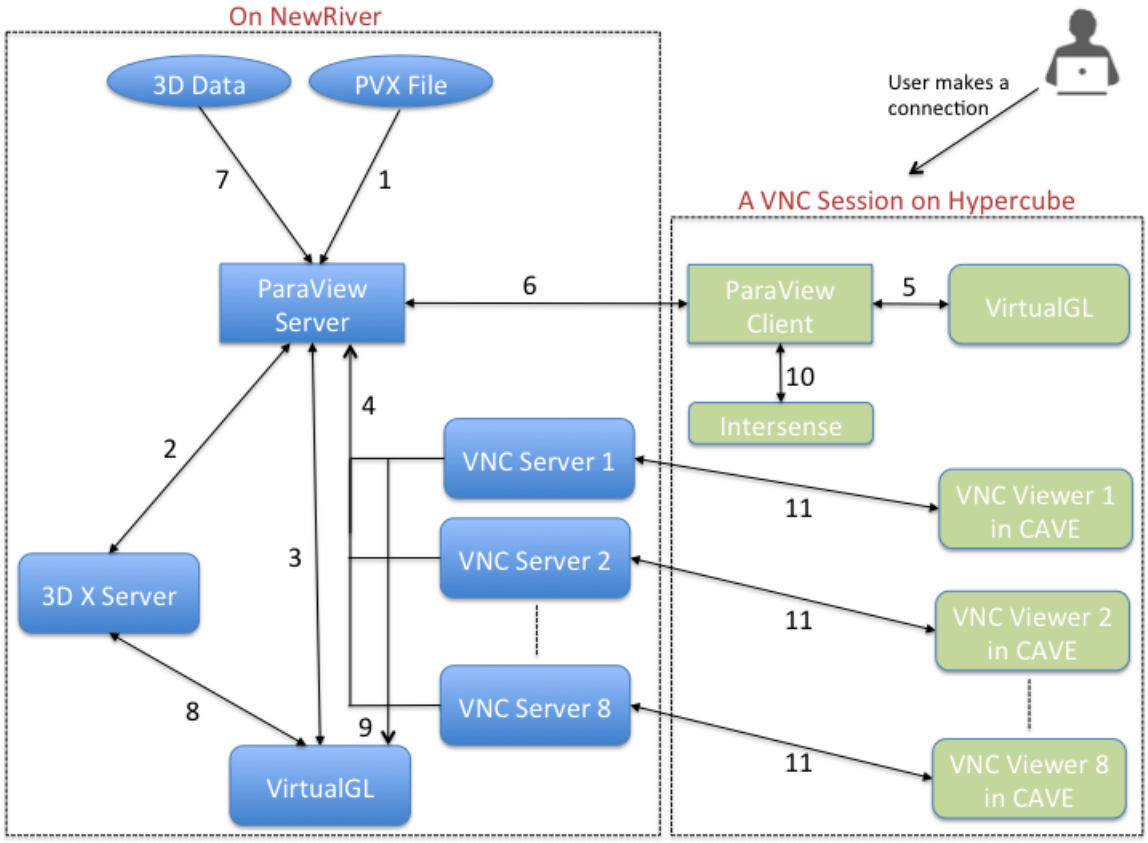


Figure 3.1: Overview of the experimental setup to run experiments

3. The pvservers need to be prefixed with "virtualgl" while starting on the remote machine for reasons stated in section 2.6.

4. Eight vncservers were started on the remote server starting from DISPLAY number 1 till 8. These are the display numbers set in the pvx file. If using 8 pvservers, there are 8 DISPLAYs and each DISPLAY's geometry is set as "Geometry='2560x1600". If using 16 pvservers, there will still be 8 DISPLAYs where each display will be split into 2 having a geometry of "1280x1600+0+0" and "1280x1600+1280+0".

5. Just like the pvservers, the ParaView client needs to be prefixed with "virtualgl" for reasons mentioned in section 2.6.

6. Since the pvservers were running on a host in the NewRiver cluster which is behind a firewall, we setup ssh tunneling [74] to make a connection between the client and the pvservers.

7. The data is stored on the NewRiver cluster and the client can browse and load it.

8. VirtualGL running on the server needs access to an X server on which it can render the intercepted OpenGL commands and geometry.

9. VirtualGL communicates with the vncservers using the X11 transport protocol (see section 2.4 for details on the X11 protocol).

10. Intersense is900 [28] can be used for tracking the user coordinates in a CAVE. It supports 6 degrees of freedom and integrates well with VRPN [59]. In our setup, the is900 was physically connected to the Hypercube and the VRPN server was configured to work with the ParaView client. Thus, when a user using a wand and a head tracker moved in the CAVE, the updated coordinates were sent to the server as X11 events and the server rendered the new images as per the tracker data received from the ParaView client.

11. Each side of the wall in the CAVE ran two instances of vncviewers, each connected to a vncserver running on the NewRiver host. The only role of the vncviewers was to display the pixels in the CAVE and provide an option to set any compression parameters.

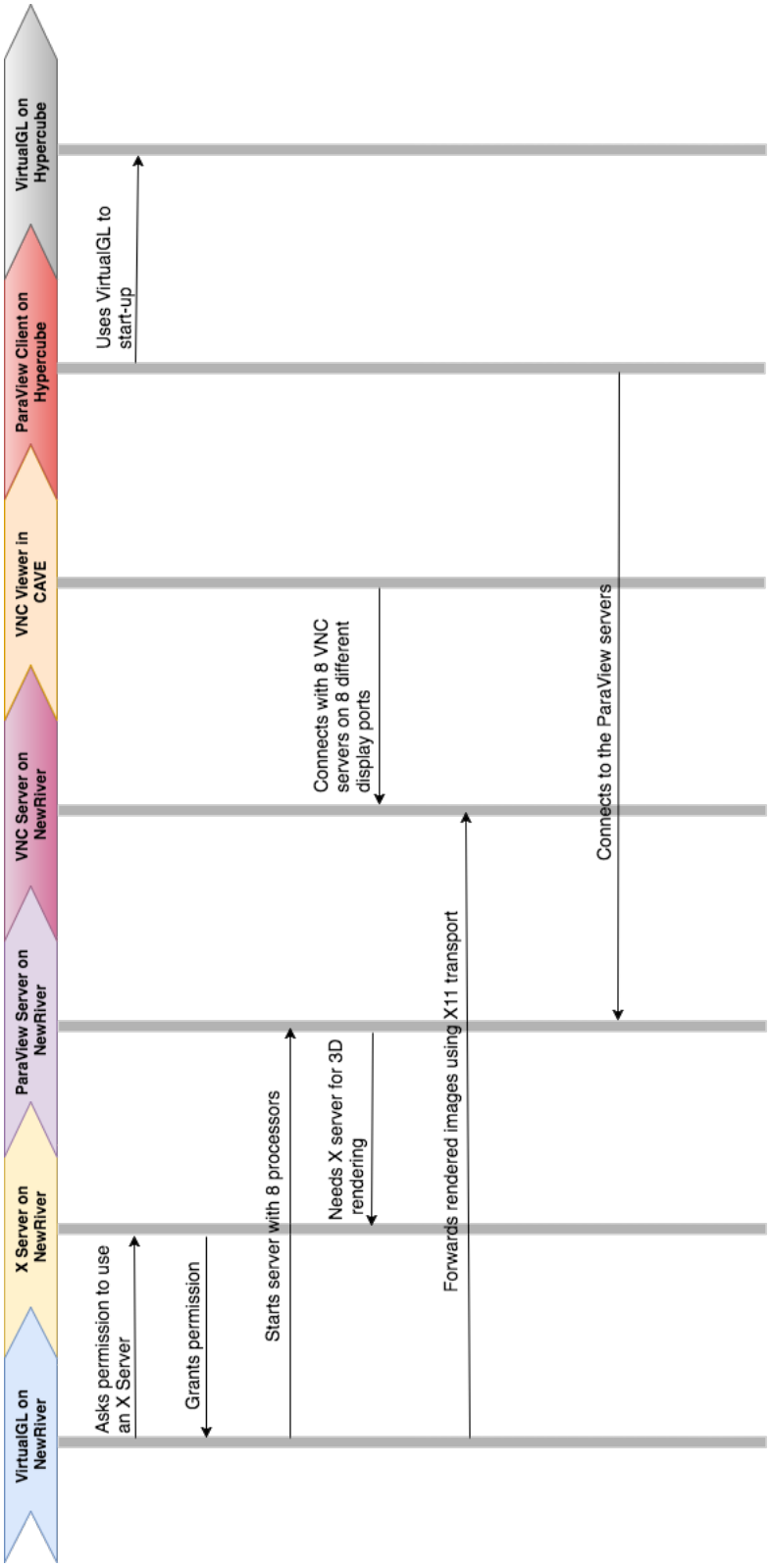


Figure 3.2: Flowdigram of the experimental setup

3.3 Data used for running the experiments

We used a point cloud dataset that has more than 5 billion points in it. It is a LiDAR [71] mass scan of the Goodwin hall at Virginia Tech that has motion sensors installed at different locations. The data collected from these sensors is in the form of x, y, and z coordinates and a 4th value for luminance.

The data was given to us in 5 different CSV files by Dr. Pablo A. Tarazaga who is an assistant professor in the Mechanical Engineering department at Virginia Tech. The combined size of the data was over 270 GB and it was run through some noise cleaning scripts. The cleaned CSV data was converted into unstructured XML VTK format in binary also called VTU [19] format using ParaView's D3 filter [31]. Converting the CSV files into VTU files was necessary to make sure that the data can be distributed in parallel when running ParaView with MPI i.e. load balancing.

We ran two types of experiments: one which had 8 vtu files and the other with 16 vtu files. Both types of experiments were run through the same nine combinations of compression as shown in table 4.2. We discuss our results in chapter 4

3.4 Limitations of ParaView in a CAVE

3.4.1 Big data in CAVE

Our initial goal was to visualize all the five billion points in the CAVE and test the different compression parameters. But after setting up our entire pipeline, when we started loading and testing the data in the CAVE, we encountered an issue with ParaView and the MPI implementation in that we saw an error message like the one shown in figure 3.3.

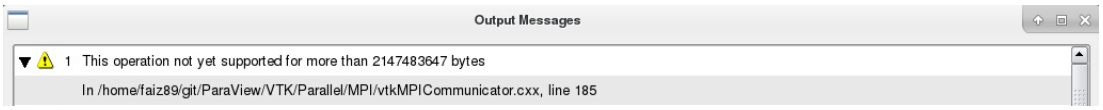


Figure 3.3: MPI error message with ParaView when visualizing big data in a CAVE

This error message is not encountered when we tried to load smaller datasets like 10 million points. More research into this issue showed that there are two MPI routines that are used for sending and receiving data used in ParaView - MPI_Send [40] and MPI_Recv [39]. The return type for both of these routines is an integer meaning that the return value can not be greater than 2^31 - 1, which is equal to 2,147,483,647 bytes. This essentially meant that whenever MPI will try to send or receive data greater than 2,147,483,647 bytes, it will bail out.

ParaView in a CAVE mode duplicates data on each node and there is no composting. Previously, "vtkCaveRenderManager" class in the ParaView source code did this but now the part of code that does this has been shifted to "vtkCaveSynchronizedRenderers.h" and "vtkCaveSynchronizedRenderers.cxx" [30]. This duplication of data does not happen in a non-CAVE mode and hence, bigger datasets can be visualized. The reason given by the ParaView developers for the duplication of data in the CAVE mode is to increase its efficiency.

There are at least two solutions that can potentially be applied in the ParaView source code to fix this issue. If an application wants to send, say, 4 x 2,147,483,647 bytes to a peer, instead of sending one big single 8 GB message (4 x 2,147,483,647 = 8 GB), the application can send 4 messages each of 2 GB in size. Another workaround is to use "MPI_TYPE_CONTIGUOUS" to create a datatype comprised of multiple elements, and then send multiple of these creating a multiplicative effect [55].

3.4.2 Stereo rendering in CAVE

We use Barco F50 projectors in the CAVE at Virginia Tech (see table 3.4 for details on the projectors). With these projectors, the active stereo happens at the projector level. It sends two separate video channels, 1 for the left eye and one for the right eye.

ParaView in a CAVE mode does not have an option to define a left eye and a right eye. It assumes that there is just one channel coming from the projector that has information for the both eyes. The DISPLAYs set in the ParaView's pvx file are not meant to be different for different eyes and defines a single eye separation value for the whole configuration.

This problem can be solved if we enabled the stereo option in the Nvidia configuration file, in which case ParaView would have given us a left eye on one display and the right eye on the other and this would have been fed to the projector which would again turn it into active stereo. But we kept the stereo option disabled in the X configuration file since if stereo is enabled it interferes with the Nvidia driver-level blending, and this is a known issue with Nvidia drivers for Linux (as of this writing, Nvidia has put it on their to-do list to fix it).

The other option to solve this problem as suggested by the ParaView community is to somehow define a left eye and a right eye in ParaView source code, which would lead to passive stereo. This is potentially future work.

3.5 Interacting with data and ParaView's timer log

To measure the frame rates, we needed a standard way to interact with the data and observe the time taken to render it. ParaView provides a timer log that monitors the time taken to render the data during different stages. We were interested in observing the *still render time* and the *interactive render time*. Still render measures the time from when the render

is initiated to the time when the render completes. Interactive render occurs when a user moves and interacts with the 3D data using the ParaView GUI. Still renders happen after the interaction stops to give a full-resolution view of the data.

We needed the interactions to remain consistent for all the experiments run and for this reason we decided to script the mouse movements on the client machine using [44]. We added 1000 scripted mouse movements on the client side interacting with the data shown on the ParaView GUI. We used the values of still and interactive renders obtained from the timer log and averaged them to obtain the frame rates.

Chapter 4

Results

This chapter discusses the different results we got by varying the compression parameters supported by TurboVNC. We measured the maximum, minimum, and average network and memory consumed at the client side, the maximum memory consumed at the server side, and the average still and interactive frame rates. Note that we did not measure percentage CPU on the server side and the reason for this is that when running the pvserver [11] with MPI [72], we get 100% CPU utilization even when the pvserver sits idle. The reason for this is how the MPI layer is implemented in the two most common implementations: OpemMPI [38] and MPICH [41]. In both of these implementations, when a process is waiting for a message, the process actually sits in a busy wait loop. The argument for this intentional implementation is that once a process goes idle, it may be scheduled off that core by the operating system scheduler and later scheduled back to a different core when it needs to process something. This can have detrimental effects on the memory access since each core can have its own cache hierarchy [43].

4.1 Frame rates

The experience of a user in a CAVE-like [70] environment is dependent on how fast can interactive frames be rendered and shown to a user. Typically, the frame rates for motion picture cameras is set to 24 frames per second (FPS) [58]. This is because 24 FPS is the slowest frame rate that can still support audio when played from a 35 mm reel. But alternate frame rates exists now like 48 FPS and this was used to film the movie The Hobbit [22]. Higher FPS like 90 and 100 are used in GoPro Hero cameras and an even higher frame rate of 300 FPS is what TV channel BBC uses for some of its sports broadcasts [10]. For the purpose of our benchmarking in this work, a higher frame rate is better.

Table 4.2 shows the different compression parameters that we varied to run our experiments. Details on each parameter can be found in section 2.5.2. Certain cases that we define in our table in 2.5.2 have been given specific names by TurboVNC that we describe below as mentioned in the documentation.

1. "Tight + Perceptually Lossless JPEG": This encoding method requires a great deal of Tight + Perceptually Lossless JPEG. CASE 6 in table 4.2 shows the settings for this encoding method.

2. "Tight + Medium-Quality JPEG": For sub-rectangles that have a high number of unique colors, this encoding method produces some minor, but generally not very noticeable, image compression artifacts. All else being equal, this encoding method typically uses about twice the network bandwidth of the "Tight + Low-Quality JPEG" encoding method and about half the bandwidth of the "Tight + Perceptually Lossless JPEG" encoding method, making it appropriate for medium-speed networks such as 10 Megabit Ethernet. Interframe comparison is enabled with this encoding method (Compression Level 6 = Compression Level 1 + interframe comparison). CASE 8 in

table 4.2 shows the settings for this encoding method.

3. "Tight + Low-Quality JPEG": For sub-rectangles that have a high number of unique colors, this encoding method produces very noticeable image compression artifacts. However, it performs optimally on low-bandwidth connections. If image quality is more critical than performance then use one of the other encoding methods or take advantage of the Lossless Refresh feature. In addition to reducing the JPEG quality to a "minimum usable" level, this encoding method also enables interframe comparison and Compression Level 2 (Compression Level 7 = Compression Level 2 + interframe comparison). CASE 9 in table 4.2 shows the settings for this encoding method.

4. "Lossless Tight": This encoding method uses indexed color subencoding for sub-rectangles that have a low number of unique colors, but it otherwise does not perform any image compression at all. Lossless Tight is thus suitable only for gigabit and faster networks. This encoding method uses significantly less CPU time than any of the JPEG-based encoding methods. CASE 1 in table 4.2 shows the settings for this encoding method.

5. "Lossless Tight + ZLIB": This encoding method uses indexed color subencoding for sub-rectangles that have a low number of unique colors and raw subencoding for sub-rectangles that have a high number of unique colors. It compresses all sub-rectangles using ZLIB with ZLIB compression level 1. For certain types of low-color workloads (CAD applications, in particular), this encoding method may use less network bandwidth than the "Tight + Perceptually Lossless JPEG" encoding method, but it also uses significantly more CPU time than any of the JPEG-based encoding methods. Interframe comparison is enabled with this encoding method. CASE 4 in table 4.2 shows the settings for this encoding method.

We created four main groups as shown in table 4.1 based on if JEPG compression is enabled and the quality and the JPEG chrominance sub-sampling set. Each main group is shown in the scatter plots using a specific colour. The varying component in the main groups is the amount of compression level set. Table 4.2 shows all the 9 cases in detail.

Table 4.1: Main groups based on JPEG, quality, and JPEG chrominance sub-sampling

Main Group	Encoding	JPEG	Quality	JPEG Subsampling	Colour
1	Tight	0	95	1x	Blue
2	Tight	1	95	1x	Red
3	Tight	1	80	2x	Yellow
4	Tight	1	30	4x	Green

Table 4.2: Different compression parameters used in the experiments

#	Encoding	JPEG	Quality	JPEG Subsampling	Compression Level
CASE 1	Tight	0	95	1x	0
CASE 2	Tight	0	95	1x	1
CASE 3	Tight	0	95	1x	5
CASE 4	Tight	0	95	1x	6
CASE 5	Tight	1	95	1x	0
CASE 6	Tight	1	95	1x	1
CASE 7	Tight	1	95	1x	5
CASE 8	Tight	1	80	2x	6
CASE 9	Tight	1	30	4x	7

4.1.1 Using 8 VTU files and 8 CPUs

Figure 4.1 shows the still render frame rate with a mean of 6.536, standard deviation of 0.118, and a standard error of 0.039. Figure 4.2 shows the interactive render frame rates with a mean of 52.073, standard deviation of 4.393, and a standard error of 1.464. These results were obtained for the 9 different cases shown in table 4.2.

We can see from the graphs that for still render FPS, CASE 1 (6.727 FPS) performed the

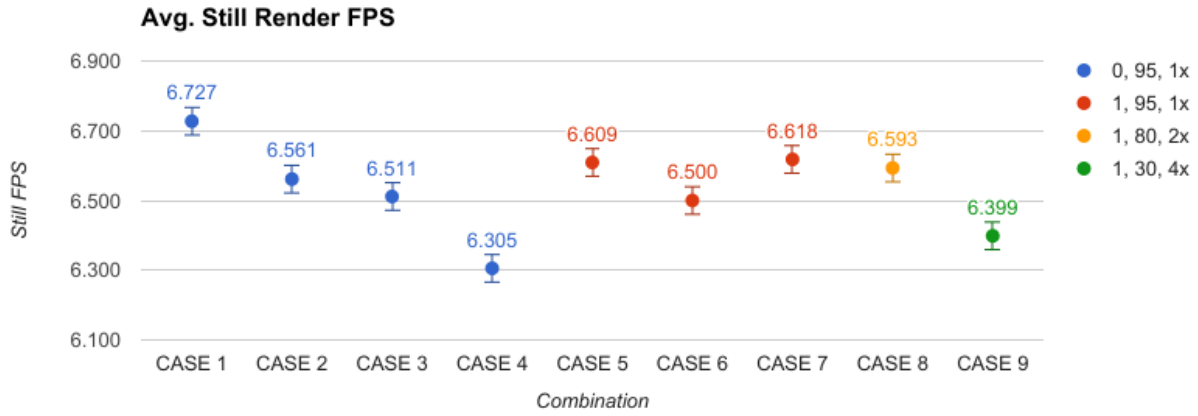


Figure 4.1: Average still render FPS with 8 VTU files and 8 CPUs

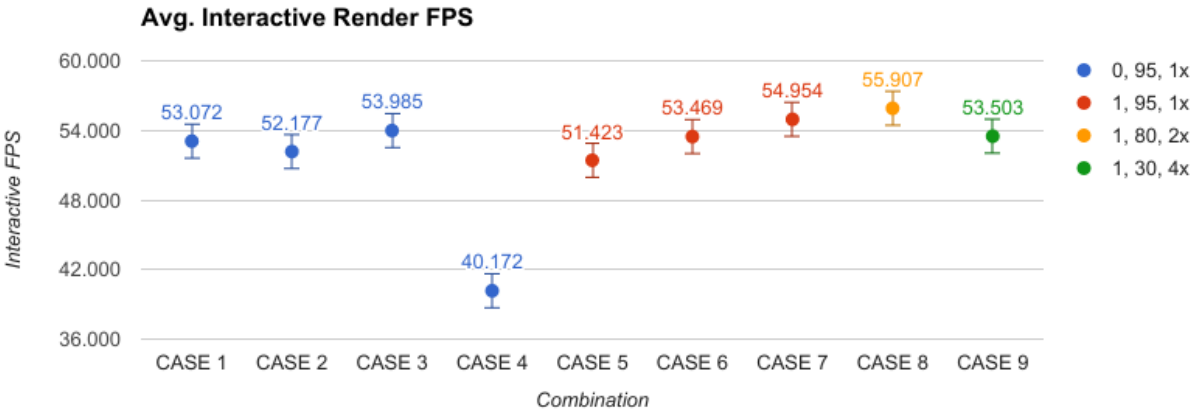


Figure 4.2: Average interactive render FPS with 8 VTU files and 8 CPUs

best while CASE 4 (6.305 FPS) performed the worst. For interactive render FPS, CASE 8 (55.907 FPS) preformed the best while CASE 4 (40.172 FPS) performed the worst. Overall, there is not much difference between the different still and interactive render FPS obtained for the different cases except for CASE 4.

4.1.2 Using 16 VTU files and 16 CPUs

Figure 4.3 shows the still render frame rate with a mean of 0.783, standard deviation of 0.004, and a standard error of 0.001. Figure 4.4 shows the interactive render frame rates with a mean of 28.037, standard deviation of 5.255, and a standard error of 1.751. These results were obtained for the 9 different cases shown in table 4.2.

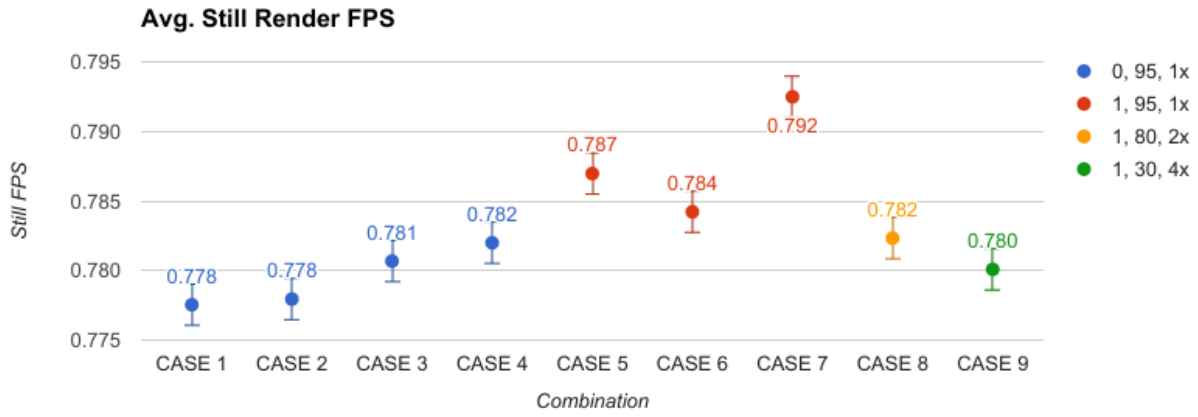


Figure 4.3: Average still render FPS with 16 VTU files and 16 CPUs

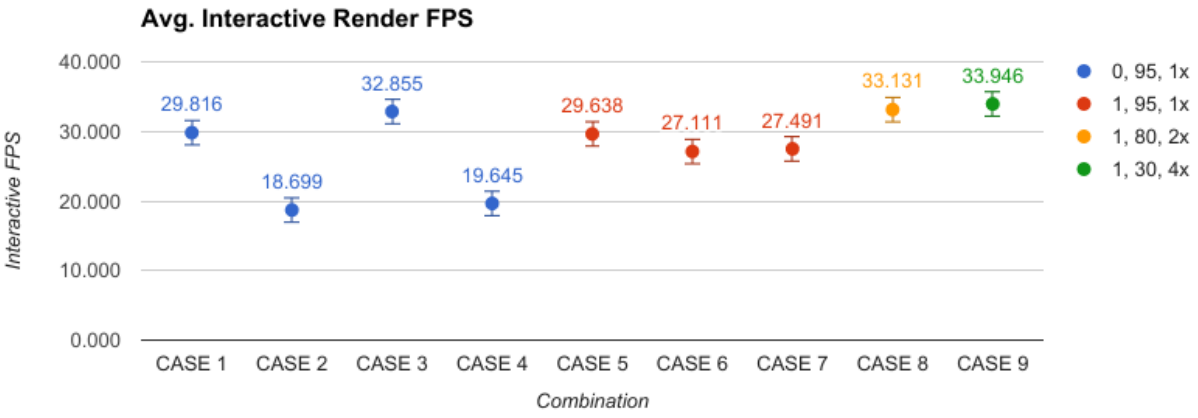


Figure 4.4: Average interactive render FPS with 16 VTU files and 16 CPUs

We can see from the graphs that for still render FPS, CASE 7 (0.792 FPS) performed a little better than the others while CASE 1 and CASE 2 performed the worst (0.778 FPS each).

For interactive render FPS, CASE 9 (33.946 FPS) preformed the best while CASE 2 (18.699 FPS) performed the worst. Overall, there is not much difference between the different still and interactive render FPS obtained for the different cases except for CASE 2 and CASE 4.

4.2 Network usage

When rendering big data on remote servers, one of the big challenges is to send and receive the data over a network. If the client does not receive the rendered data from the server within an appropriate amount of time, the user will notice a lag. This lag can significantly affect the user performance in a 3D immersive environment where the user is interacting with the data in real time.

We can compress the data like we discussed in section 2.5.2 to minimize sending and receiving of bytes over a network and in this section, we show how the 9 different cases discussed in table 4.2 affect the network performance. We used a 10 Gigibit Ethernet for these tests and this bandwidth meant that we could get maximum download speeds of around 250MB/s and maximum upload speeds of around 256MB/s.

For the purpose of this study, the results using less bandwidth are preferred. We measured the maximum and average bytes received and the average bytes sent at the client side using an open source software called Ganglia as discussed in section 2.7. Note that we did not measure the network traffic on the server side because a) the network traffic received by the client should almost be equal to the network traffic sent by the server and thus, we can safely assume that measuring one side of the network traffic (either the client or the server) should be sufficient and b) since we were using the Newriver [16] cluster for remote rendering, we did not have administrative permissions to install Ganglia on it to measure the network like we did for the client.

4.2.1 Using 8 VTU files and 8 CPUs

Figure 4.5 shows the maximum data received at the client side, figure 4.6 shows the average data received at the client side with a mean of 15.388, standard deviation of 6.723, and a standard error of 2.241, and figure 4.7 shows the average data sent by the client to the server with a mean of 0.587, standard deviation of 0.017, and a standard error of 0.005.

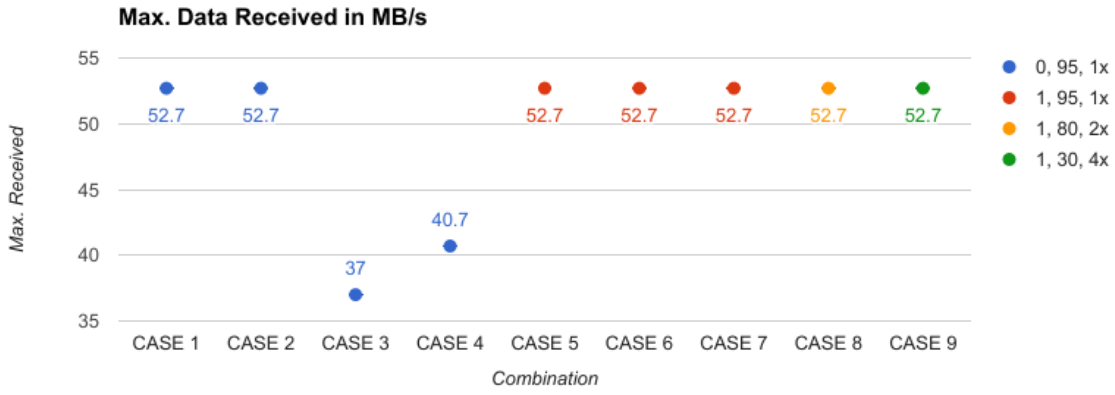


Figure 4.5: Maximum data received with 8 VTU files and 8 CPUs

Figure 4.5 shows the maximum data received by the client from the server during any time in the rendering process. This can happen just once or maybe more, but we are interested in knowing how much maximum data was received by the client during the entire process. All the test cases except for CASE 3 (37 MB/sec) and CASE 4 (40.7 MB/sec) show the same metric (52.7 MB/sec).

Figure 4.6 shows the average data received by the client from the server during the rendering process. The maximum data load may not occur frequently and thus, it is important to average the traffic pattern during the entire rendering pipeline. We can see from the graphs that CASE 1 and CASE 3 received the maximum amount of data (24.4 MB/sec each) while the least average data was received by CASE 9 (6 MB/sec).

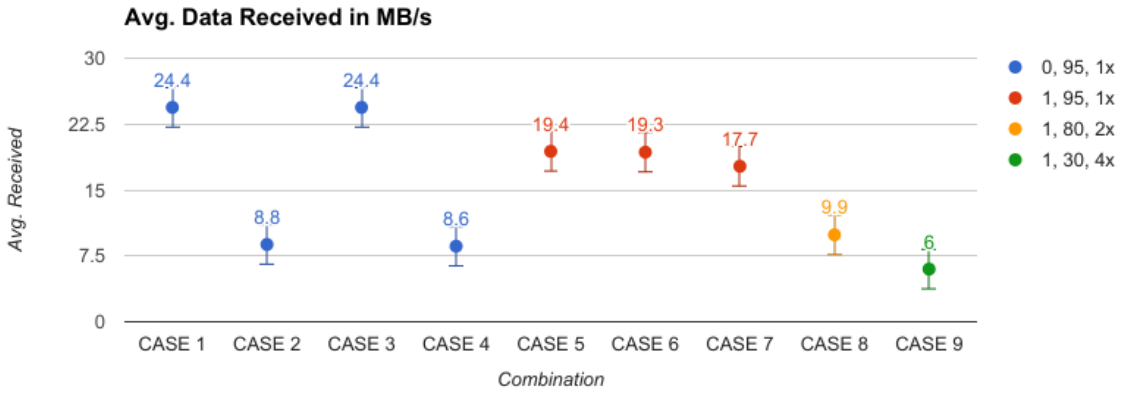


Figure 4.6: Average data received with 8 VTU files and 8 CPUs

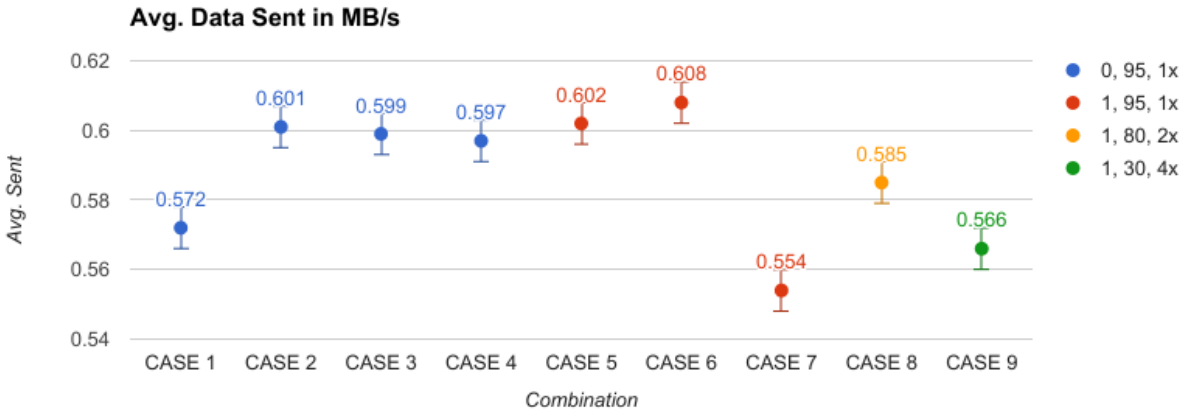


Figure 4.7: Average data sent with 8 VTU files and 8 CPUs

Although very little traffic was sent from the client to the server, we were still interested to know how much was the average during the rendering process. Figure 4.7 shows all the values and though all the cases show similar metrics, CASE 7 (0.554 MB/sec) sent the least data to the server.

4.2.2 Using 16 VTU files and 16 CPUs

Figure 4.8 shows the maximum data received at the client side, figure 4.9 shows the average data received at the client side with a mean of 7.322, standard deviation of 3.333, and a standard error of 1.111, and figure 4.10 shows the average data sent by the client to the server with a mean of 0.307, standard deviation of 0.023, and a standard error of 0.007.

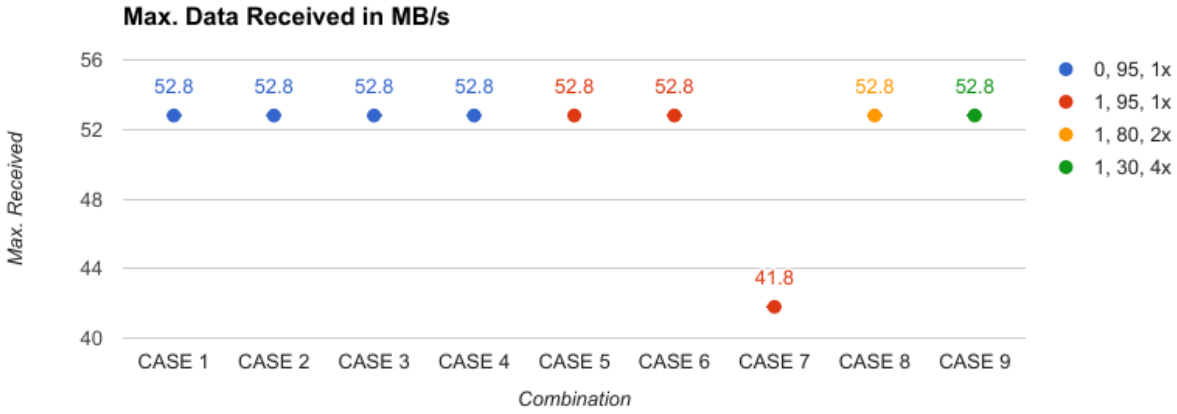


Figure 4.8: Maximum data received with 16 VTU files and 16 CPUs

For the maximum data received at any point during the rendering process, all the test cases showed the same metric (52.8 MB/sec) except for CASE 7 (41.8 MB/sec). Figure 4.8 shows all the results.

For the average data received during the entire rendering process, we see that CASE 9 (2.8 MB/sec) averaged the minimum followed by CASE 4 (4 MB/sec). CASE 1 averaged the maximum (12.4 MB/sec) followed by CASE 3 (11.4 MB/sec). Figure 4.9 shows all the results.

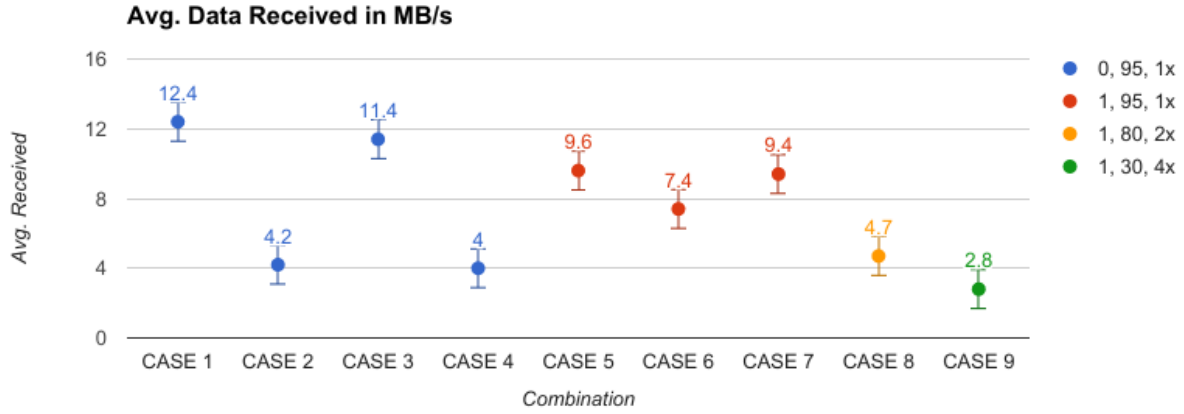


Figure 4.9: Average data received with 16 VTU files and 16 CPUs

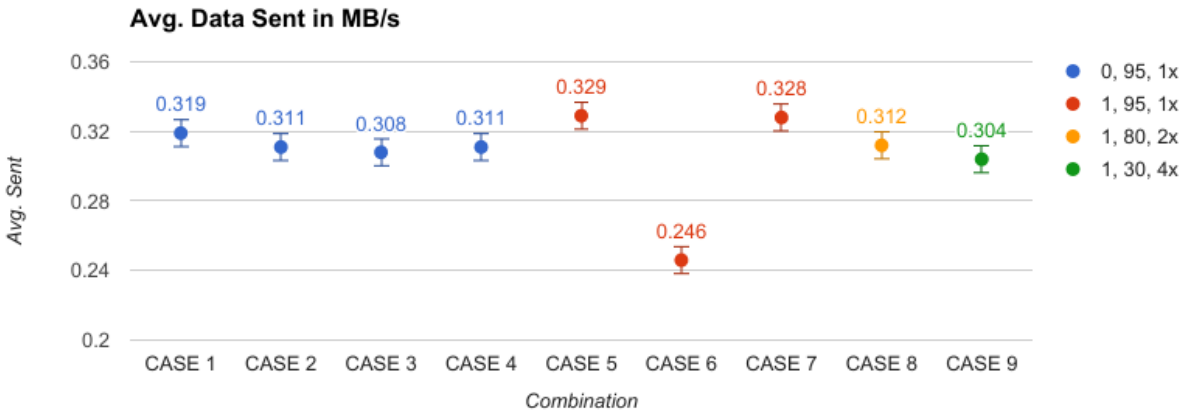


Figure 4.10: Average data sent with 16 VTU files and 16 CPUs

Although the data sent from the client machine to the server seemed insignificant, we still measured what was the average data sent. As we can see in figure 4.10, CASE 6 (0.246 MB/sec) sent the minimum data while CASE 5 was highest (0.329 MB/sec).

4.3 Memory Usage

Rendering big data sets on local machines requires the local machines to have sufficient amount of memory that gets used in the process otherwise the rendering may result in a failure or may take a long time to complete. However, if one has remote access to more powerful servers with more memory, CPUs, and GPUs, it makes sense to send the data to the remote servers that perform the rendering and the local machine gets back the rendered data.

In our experiments, the local machine was running 8 vncviewers (see section 2.5.1 for details) along with VirtualGL (see section 2.6 for details on it) and the ParaView client (see section 2.3 for details). The remote server was running 8 vncservers, virtualgl, and 8 or 16 pvservers (see section 2.3 for more details).

We used Ganglia (see section 2.7 for details) to measure the maximum and minimum memory footprint on the client side. We only measured peak memory usage on the server side using native operating system tools since we did not have administrative access to install Ganglia on the server, which would let us measure the average memory consumption during the rendering process.

4.3.1 Using 8 VTU files and 8 CPUs

Figure 4.11 shows the maximum memory consumed at the client side, figure 4.12 shows the average memory consumed at the client side with a mean of 15.877, standard deviation of 0.332, and a standard error of 0.110, and figure 4.13 shows the maximum memory consumed at the server side.

Figure 4.11 shows that CASE 9 (23 GB) consumed maximum memory followed by CASE 7

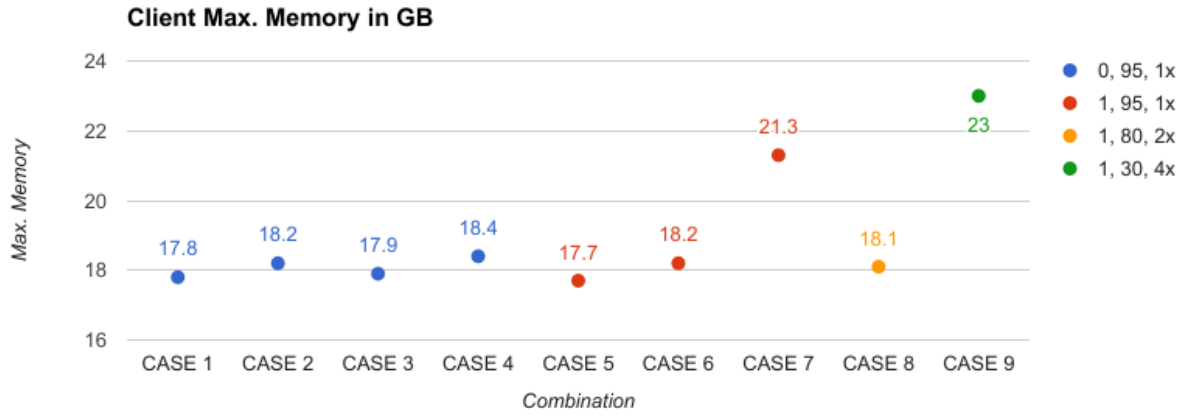


Figure 4.11: Maximum client memory used with 8 VTU files and 8 CPUs

(21.3 GB). Relatively, CASE 7 (17.7 GB) consumed the least memory. Figure 4.12 shows that on an average, CASE 2 and CASE 7 consumed the maximum memory (16.3 GB each) closely followed by CASE 9 (16.2 GB) and CASE 4 (16.1 GB). CASE 3 (15.4 GB) consumed the least average memory. On the server side, there was not much difference in the peak memory usage for any test case. Figure 4.13 shows that, relatively, CASE 4 (50.84) consumed the maximum memory while CASE 1 (50.65 GB) consumed the minimum.

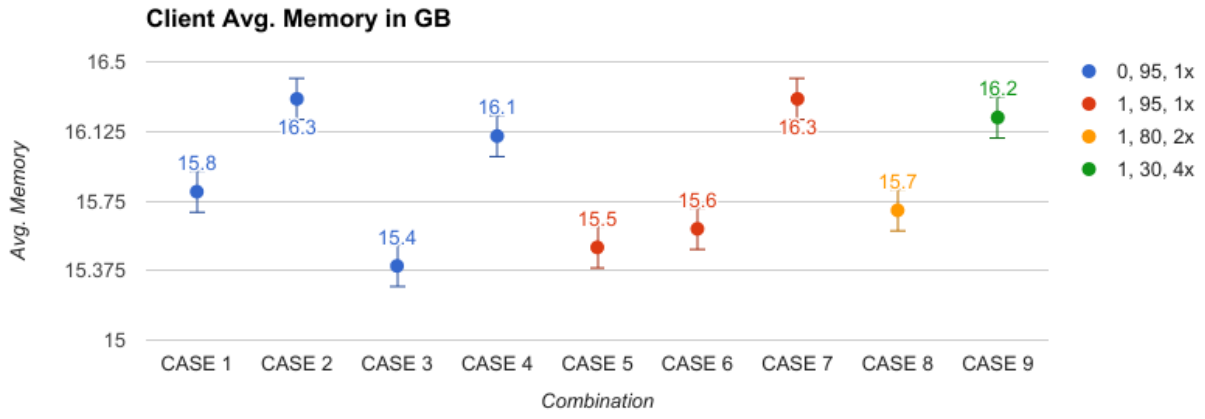


Figure 4.12: Average client memory used with 8 VTU files and 8 CPUs

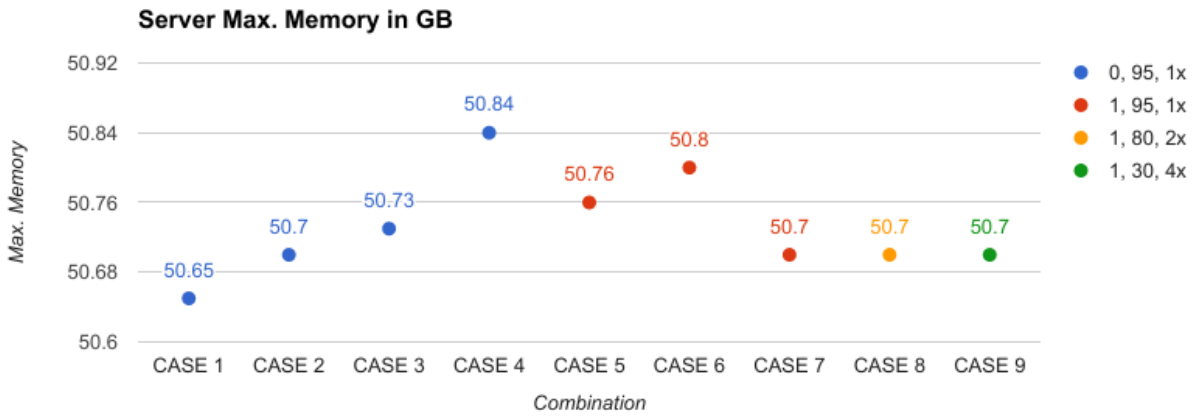


Figure 4.13: Maximum server memory used with 8 VTU files and 8 CPUs

4.3.2 Using 16 VTU files and 16 CPUs

Figure 4.14 shows the maximum memory consumed at the client side, figure 4.15 shows the average memory consumed at the client side with a mean of 17.077, standard deviation of 0.456, and a standard error of 0.152, and figure 4.16 shows the maximum memory consumed at the server side.

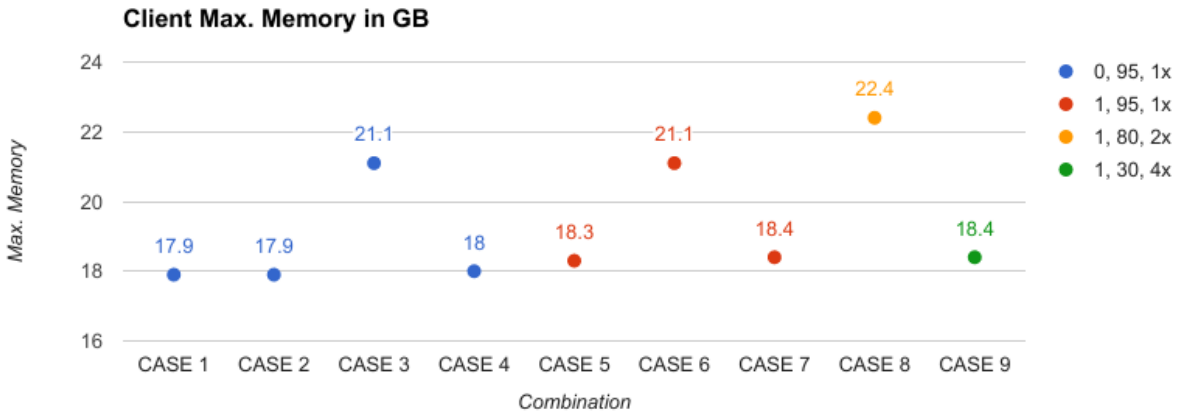


Figure 4.14: Maximum client memory used with 16 VTU files and 16 CPUs

Figure 4.14 shows that CASE 8 (22.4 GB) consumed the maximum memory followed by CASE 3 and CASE 5 (21.1 GB each). CASE 1 and CASE 2 consumed the least memory (17.9 GB each). Figure 4.15 shows that on an average, CASE 3 (18.1 GB) consumed the maximum memory while CASE 6 (16.3 GB) consumed the least. Figure 4.16 shows that all the test cases consumed almost the same peak memory at the server side. Relatively, CASE 2 and CASE 7 consumed the maximum memory (108.5 GB each) while CASE 5 (108.39 GB) consumed the least.

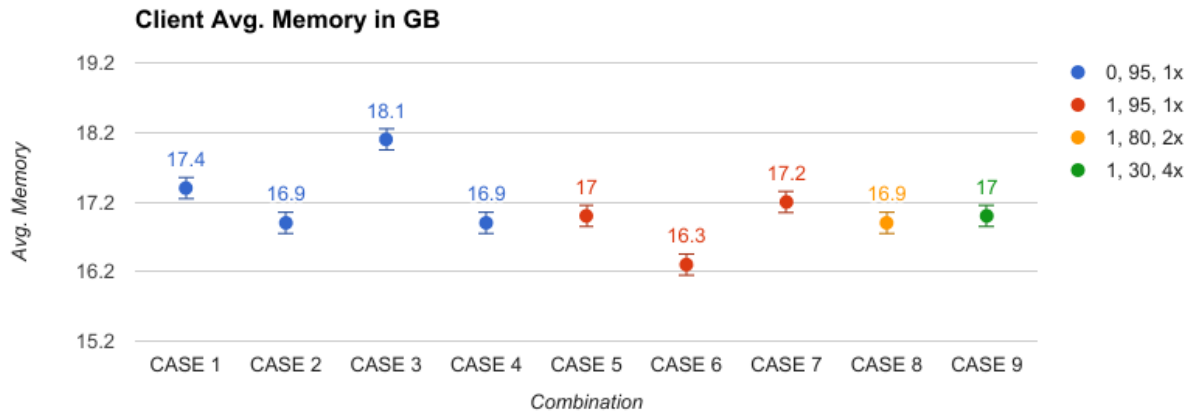


Figure 4.15: Average client memory used with 16 VTU files and 16 CPUs

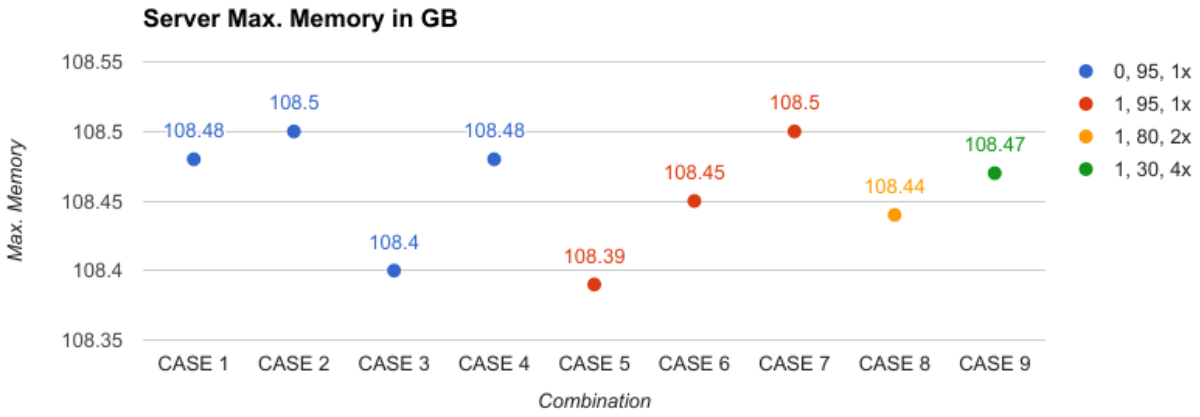


Figure 4.16: Maximum server memory used with 16 VTU files and 16 CPUs

Chapter 5

Discussion

5.1 Welch Two Sample t-tests

We had several combination of variables that we varied and got different results. In order to understand which of these results are significant and dependent on the variables we tested, we ran 8 Welch Two Sample t-tests [77]. We created two groups as shown in table 5.1. The 8 t-tests were run for the variables shown in table 5.2.

Table 5.1: Groups created for Welch Two Sample t-tests

Group	# Processors	# VTU files
I	8	8
II	16	16

Only 4 out of the 8 p-values that we got were significant as shown in table 5.3. This means that the other 4 variables (Max. data received, Avg. memory on client, Max. memory on client, and Max. memory on server) were not affected by changing the number of processors and VTU files.

Table 5.2: Tests performed on different variables

Variables
Avg. data received
Max. data received
Avg. data sent
Avg. memory on client
Max memory on client
Max. memory on server
Avg. still render frame rate
Avg. interactive render frame rate

Table 5.3: Significant results obtained from the t-tests

Variable name	p-value
Avg. data received	0.005267
Avg. data sent	1.96E-14
Avg. still render frame rate	4.14E-15
Avg. interactive render frame rate	2.03E-08

5.2 ANOVA Analysis

To further analyze the effects of the different compression parameters on the variables, we created sub-groups as shown in table 5.4 and ran 8 ANOVAs [69] for each of the variables shown in table 5.2. The results we got are shown in table 5.5. All the results except for Max. memory on the client and Max. data received were affected by the number of processors and VTU files. We can see from the results that the number of processors and the VTU files were the main factors affecting the variables and there was no variable that was affected by if JPEG was enabled or disabled.

Table 5.4: Groups created for ANOVA for the entire data

Group	# Processors and VTU files	JPEG
I	8	Disabled (0)
II	8	Enabled (1)
III	16	Disabled (0)
IV	16	Enabled (1)

Table 5.5: Significant results obtained from ANOVA for all the data

Variable name	Significant factor	p-value
Avg. data received	Processors and VTU files	0.01209
Avg. data sent	Processors and VTU files	3.25E-13
Avg. memory on client	Processors and VTU files	2.68E-05
Max. memory on server	Processors and VTU files	<2e-16
Avg. still render FPS	Processors and VTU files	<2e-16
Avg. interactive render FPS	Processors and VTU files	5.169e-08

Within each of the two main groups (8 and 16 processors and VTU files), we created sub-groups and used 3 predictors: JPEG, Quality, and Chrominance (see table 5.6). ANOVA did not show significant results for any variable except for one as shown in table 5.7.

Table 5.6: Sub-groups created for ANOVA

Group	JPEG Compression	Quality	Chrominance
I	Disabled (0)	95	1
II	Enabled (1)	95	1
III	Enabled (1)	80	2
IV	Enabled (1)	30	4

Table 5.7: Significant results obtained from ANOVA of sub-groups

Variable name	Significant factor	p-value
Avg. still render FPS	JPEG	0.04305 for 16 processors and 16 VTU files

5.3 Major takeaways

1. The still render frame rate for 8 CPUs and 8 VTU files was significantly better than those for 16 CPUs and 16 VTU files by a factor greater than 8. This is also shown by the t-tests.

2. The interactive render frame rate for 8 CPUs and 8 VTU files was significantly better than those for 16 CPUs and 16 VTU files by a factor greater than 1.6.

3. Based on the ANOVA analysis, the most significant factor deciding the frame rates was number of CPUs and VTU files and if JPEG compression was enabled.

4. Quality and JPEG chrominance sub-sampling did not have any significant impact on the dependent variables.

5. The average data received by the client for 8 CPUs and 8 VTU files was significantly more than those for 16 CPUs and 16 VTU files by a factor almost equal to 2.

6. The peak data received by the client for 8 CPUs and 8 VTU files was almost the same as for 16 CPUs and 16 VTU files.

7. The average memory consumed by the client for 8 CPUs and 8 VTU files was slightly less than that consumed for 16 CPUs and 16 VTU files by a factor almost equal to 1.11.

8. The peak memory consumed by the client for 8 CPUs and 8 VTU files was almost equal to that consumed for 16 CPUs and 16 VTU files.

9. The peak memory consumed by the server for 8 CPUs and 8 VTU files was significantly less than that consumed for 16 CPUs and 16 VTU files by a factor almost equal to 2.

5.4 Effect of network and stereo

All the results that we got used the VT-RNET 10 GB network, which meant that we could get upload speeds of up to 256 MB/sec and download speeds of up to 250 MB/sec. In our experiments, the peak data transfer speed achieved was 52.7 MB/sec, which means that if we were using the 1 GB network that we previously had in the lab to run our experiments, our results would have been affected significantly. The 1 GB network only supports a download and upload speed of approximately 5 MB/sec and if we were to use this to run the same experiments, we predict that the still and interactive render frame rates would have decreased significantly affecting the user experience in the CAVE.

All our experiments were run in a monoscopic fashion and the main difference between monoscopic and stereoscopic images is the addition of depth information in stereoscopic

images, which gives a false perception of viewing the images in 3D. The CAVE at Virginia Tech has 8 projectors (2 projectors per wall) capable of doing active stereo but we could not make use of this capability because of limitations of ParaView as discussed in section 3.4. However, if we were to overcome this limitation and re-did our experiments in stereoscopic mode, we think that the performance will not be as good as that achieved in the monoscopic mode. In monoscopic mode we had 8 displays each using 1 processor but we had to use 16 processors for our second case where each display was split into 2 having a geometry of "1280x1600+0+0" and "1280x1600+1280+0". The number of MPI processes and DISPLAYs set in the ParaView XML file or the pvx file should match for a successful connection between the client and the server. As we saw in our results, the frame rate achieved with 8 processors was almost double the frame rate achieved with 16 processors and this is mainly because of how ParaView duplicates the data on each node (see section 3.4.1 for more details). So, if we were to do stereoscopic data visualization in the CAVE that would mean defining 2x number of DISPLAYs in the pvx file (one for the left eye and one for the right eye) and requesting 2x number of processors for rendering, and because of how ParaView duplicates data on each node, adding more processors should adversely affect performance.

Chapter 6

Conclusion and future work

In this work, we visualized a point cloud data consisting of more than 40 million points in a CAVE-style system. Our pipeline consisted of ParaView, VirtualGL, and TurboVNC. We varied 6 parameters including number of processors, number of VTU files, JPEG compression, Quality, JPEG chrominance sub-sampling, and compression level. We measured the data flowing to and from the client to the servers, the memory foot print on the client and the server, and the frame rates received at the client side. We used NewRiver HPC machines at Virginia Tech for remote rendering.

Our results show that adding more CPUs and dividing the data into more number of VTU files does not help in speeding the rendering process and visualizing the data in a CAVE-style system. The frame rates achieved with 8 processors and 8 VTU files is significantly better than the frame rates achieved with 16 processors and 16 VTU files. The average data sent and received over the network is also significantly more for 8 CPUs and 8 VTU files than 16 CPUs and 16 VTU files. This shows that adding parallelism beyond a certain extent actually has an adverse effect on the frame rates when visualizing data using ParaView in a CAVE-style system.

6.1 Recommendations

In this work we observed that there was no significant effect of varying the compression parameters like Quality and JPEG chrominance sub-sampling. Enabling or disabling JPEG compression had a significant impact in only one case while measuring average still render FPS using 16 VTU files and 16 processors. The biggest impact that we observed was while varying the number of VTU files and the number of processors. We should also note here that we used a 10G network for our tests, which could easily accommodate the maximum data flowing between the client and the server. Based on our experimental results, below are some recommendations that can be followed while dealing with big data and remotely visualizing it in a CAVE-styled setup using ParaView.

1. There is no need to degrade the quality of the image or apply any JPEG chrominance sub-sampling since it does not have any significant impact on still and interactive frame rates, network, or memory.

2. Enabling or disabling JPEG compression will not have a significant impact in all cases except one (see table 5.7 for details). Hence, it is okay to not vary this compression parameter.

3. Since ParaView duplicates data on each node (see section 3.4 for details), minimum number of processors should be used that is needed to make the CAVE work. For example, in our case, we had 8 projectors and each projector was responsible for 1 display. Hence, the optimum number of processors to be used for getting the highest frame rates was 8.

4. Tiled rendering using ParaView in a CAVE-styled setup has an adverse impact on performance because ParaView duplicates data on each node and hence, tiled rendering

is not recommended. This is evident from our results using 16 VTU files and 16 processors.

5. If the data to be visualized is in a format that can not be read in parallel (like CSV) then it should be converted into a parallel file format like VTU otherwise all the rendering will occur on a single processor.

6.2 Future work

The work we did in this thesis leaves room for some future. It would be interesting to see how other combinations perform like 8 CPUs and 16 VTU files, 16 CPUs and 8 VTU files, etc. Currently, there is a limitation to how much data can be visualized using ParaView in the CAVE but if some more work is done on enhancing ParaView's capability to support bigger data sets by incorporating the changes suggested in section 3.4, it would be interesting to see how ParaView performs and how it affects the network and frame rates.

We used a 10G network for all our tests, which could easily accommodate the maximum data flowing between the client and the server. This is could be one of the reasons why compression parameters like quality and JPEG chrominance sub-sampling did not have any significant impact on the frame rates achieved. It would be interesting to see how the same experiments behave while using a 1G network since a 1G network will not be able to accommodate the maximum data flowing between the client and the server and compression parameters like quality and JPEG chrominance sub-sampling may have a more significant role to play then.

Another possibility of future work is adding active stereo support to ParaView in addition to the mono mode that it currently supports. Like we discussed in section 5.4, we think that using stereo will adversely affect frame rates compared to the frame rates achieved in

monoscopic mode. This is because stereoscopic mode needs twice the number of displays and processors compared to the monoscopic mode and ParaView will copy all the data on each processor (see section 3.4.1 for details) affecting the total render time. However, this also brings the question of what frame rates are acceptable for a good user experience? So, even if the frame rates drop to 20 FPS in stereoscopic mode compared to getting 40 FPS in monoscopic mode, which one is better? This is another interesting question where more work can be done.

Bibliography

[1] Jpeg image codec - libjpeg-turbo. http://www.libjpeg-turbo.org/, 2017. [Online; accessed 11-April-2017].

[2] Faiz Abidi. Github repository. https://github.com/faizabidi/Thesis, 2017. [Online; accessed 16-April-2017].

[3] Mark Adler. A massively spiffy yet delicately unobtrusive compression library. http://www.zlib.net, 2017. [Online; accessed 10-April-2017].

[4] Nasir Ahmed, T_ Natarajan, and Kamisetty R Rao. Discrete cosine transform. *IEEE transactions on Computers*, 100(1):90–93, 1974.

[5] James Ahrens, Kristi Brislawn, Ken Martin, Berk Geveci, C Charles Law, and Michael Papka. Large-scale data visualization using parallel data streaming. *IEEE Computer graphics and Applications*, 21(4):34–41, 2001.

[6] James Ahrens, Berk Geveci, Charles Law, CD Hansen, and CR Johnson. 36-paraview: An end-user tool for large-data visualization. *The Visualization Handbook*, page 717, 2005.

[7] James P Ahrens, Nehal Desai, Patrick S McCormick, Ken Martin, and Jonathan Woodring. A modular extensible visualization system architecture for culled priori-

tized data streaming. In *Electronic Imaging 2007*, pages 64950I–64950I. International Society for Optics and Photonics, 2007.

[8] Xavier Amatriain, JoAnn Kuchera-Morin, Tobias Hollerer, and Stephen Travis Pope. The allosphere: Immersive multimedia for scientific discovery and artistic exploration. *IEEE MultiMedia*, 16(2):0064–75, 2009.

[9] Amazon. Cloud computing with amazon web services. https://aws.amazon.com/what-is-aws, 2006. [Online; accessed 15-April-2017].

[10] MG Armstrong, DJ Flynn, ME Hammond, SJE Jolly, and RA Salmon. High frame-rate television. *SMPTE motion imaging journal*, 118(7):54–59, 2009.

[11] Utkarsh Ayachit. The paraview guide: a parallel visualization application. 2015.

[12] Tony Bradley. The amazing technology behind disney's big hero 6. https://www.forbes.com/sites/tonybradley/2014/11/30/the-amazing-technology-behind-disneys-big-hero-6/#196d61bf3543, 2014. [Online; accessed 15-April-2017].

[13] Stuart K Card, Jock D Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[14] Doris L Carver. Designing the user interface, strategies for effective human-computer interaction. *Journal of the Association for Information Science and Technology*, 39(1): 22–22, 1988.

[15] Andy Cedilnik, Berk Geveci, Kenneth Moreland, James P Ahrens, and Jean M Favre. Remote large data visualization in the paraview framework. In *EGPGV*, pages 163–170, 2006.

[16] Advanced Research Computing. Newriver cluster for hpc. https://secure.hosting.vt.edu/www.arc.vt.edu/computing/newriver/, 2017. [Online; accessed 11-April-2017].

[17] Carolina Cruz-Neira, Daniel J Sandin, Thomas A DeFanti, Robert V Kenyon, and John C Hart. The cave: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–73, 1992.

[18] Stefan Eilemann, Maxim Makhinya, and Renato Pajarola. Equalizer: A scalable parallel rendering framework. *IEEE transactions on visualization and computer graphics*, 15(3): 436–452, 2009.

[19] Elmerfem. Vtu file format and paraview for visualization. http://www.elmerfem.org/blog/uncategorized/vtu-file-format-and-paraview-for-visualization, 2016. [Online; accessed 17-April-2017].

[20] Klaus Engel, Ove Sommer, and Thomas Ertl. A framework for interactive hardware accelerated remote 3d-visualization. In *Data visualization 2000*, pages 167–177. Springer, 2000.

[21] Alessandro Febretti, Arthur Nishimoto, Terrance Thigpen, Jonas Talandis, Lance Long, JD Pirtle, Tom Peterka, Alan Verlo, Maxine Brown, Dana Plepys, et al. Cave2: a hybrid reality environment for immersive simulation and information analysis. In *Is&t/spie electronic imaging*, pages 864903–864903. International Society for Optics and Photonics, 2013.

[22] Carolyn Giardina. Showeast 2012: Major exhibitors sign for high frame-rate 'hobbit' despite format challenges. http://www.hollywoodreporter.com/news/showeast-2012-major-exhibitors-sign-387289, 2017. [Online; accessed 10-April-2017].

[23] Google. Google compute engine documentation. https://cloud.google.com/compute/docs, 2008. [Online; accessed 15-April-2017].

[24] Michell Consulting Group. Is your company's internet connection a bottleneck on productivity: Part 2. https://www.michellgroup.com/tag/bottleneck/. [Online; accessed 16-April-2017].

[25] Alex Herrera. Is cloud-based cad ready for prime time? part 2. http://www.cadalyst.com/hardware/workstations/cloud-based-cad-ready-prime-time-part-2-19748, 2014. [Online; accessed 15-April-2017].

[26] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D Kirchner, and James T Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM transactions on graphics (TOG)*, 21(3):693–702, 2002.

[27] InstantReality. The instantreality-framework. http://www.instantreality.org/story/what-is-it, 2017. [Online; accessed 14-April-2017].

[28] Intersense. Is-900 system. http://www.intersense.com/pages/20/14, 2017. [Online; accessed 16-April-2017].

[29] Nikhil Shetty Katie Sharkey, Aashish Chaudhary and Bill Sherman. Paraview in immersive environments. https://blog.kitware.com/paraview-in-immersive-environments, 2012. [Online; accessed 14-April-2017].

[30] Kitware. Paraview's github repository. https://github.com/Kitware/ParaView, 2017. [Online; accessed 17-April-2017].

[31] Kitware. Paraview - parallel visualization application. http://www.paraview.org/, 2017. [Online; accessed 12-April-2017].

[32] LZ4. Lz4 compression. http://lz4.github.io/lz4, 2017. [Online; accessed 13-April-2017].

[33] Matt Marcus. Jpeg image compression. https://math.dartmouth.edu/~m56s14/proj/Marcus_proj.pdf, 2014. [Online; accessed 10-April-2017].

[34] Microsoft. What is azure? https://azure.microsoft.com/en-us/overview/what-is-azure, 2010. [Online; accessed 15-April-2017].

[35] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE computer graphics and applications*, 14(4):23–32, 1994.

[36] Kenneth Moreland and David Thompson. From cluster to wall with vtk. In *Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003. IEEE Symposium on*, pages 25–31. IEEE, 2003.

[37] Kenneth Moreland, Brian Wylie, and Constantine Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 85–92. IEEE Press, 2001.

[38] Open MPI. Open mpi: Open source high performance computing. https://www.open-mpi.org, 2017. [Online; accessed 10-April-2017].

[39] Mpich. Mpi receive routine. https://www.mpich.org/static/docs/v3.2/www3/MPI_Recv.html, 2017. [Online; accessed 17-April-2017].

[40] Mpich. Mpi send routine. http://www.mpich.org/static/docs/v3.1/www3/MPI_Send.html, 2017. [Online; accessed 17-April-2017].

[41] Mpich. Mpich. https://www.mpich.org, 2017. [Online; accessed 10-April-2017].

[42] Charilaos Papadopoulos, Kaloian Petkov, Arie E Kaufman, and Klaus Mueller. The reality deck–an immersive gigapixel display. *IEEE computer graphics and applications*, 35(1):33–45, 2015.

[43] ParaView. Setting up a ParaView Server. http://www.paraview.org/Wiki/ParaView, 2017. [Online; accessed 10-April-2017].

[44] Python. pynput 1.3 - monitor and control user input devices. https://pypi.python.org/pypi/pynput, 2017. [Online; accessed 22-April-2017].

[45] Digital Image quality testing. Image quality factors. http://www.imatest.com/solutions/iqfactors-2/, 2017. [Online; accessed 24-April-2017].

[46] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R Wood, and Andy Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[47] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 97–108. ACM, 2000.

[48] Robert W Scheifler and Jim Gettys. The x window system. *ACM Transactions on Graphics (TOG)*, 5(2):79–109, 1986.

[49] William J Schroeder and Kenneth M Martin. The visualization toolkit-30. 1996.

[50] ScienceDaily. Big data, for better or worse: 90% of world's data generated over last two years. https://www.sciencedaily.com/releases/2013/05/130522085217.htm, 2013. [Online; accessed 15-April-2017].

[51] Bill Sherman. Paraview & vtk on bigred-ii. https://rt.uits.iu.edu/ci/training/files/paraview.pdf, 2013. [Online; accessed 13-April-2017].

[52] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343. IEEE, 1996.

[53] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009.

[54] Sourceforge. What is ganglia? http://ganglia.sourceforge.net, 2016. [Online; accessed 14-April-2017].

[55] Jeff Squyres. Can i mpi_send (and mpi_recv) with a count larger than 2 billion? http://blogs.cisco.com/performance/can-i-mpi_send-and-mpi_recv-with-a-count-larger-than-2-billion, 2014. [Online; accessed 17-April-2017].

[56] StatsMonkey. World oil consumption statistics by country. https://www.statsmonkey.com/table/1356-world-oil-consumption-statistics-by-country.php, 2011. [Online; accessed 15-April-2017].

[57] Sareesh Sudhakaran. Understanding luminance and chrominance. http://wolfcrow.com/blog/understanding-luminance-and-chrominance/, 2016. [Online; accessed 10-April-2017].

[58] Andrew Tarantola. Why frame rate matters. http://gizmodo.com/why-frame-rate-matters-1675153198, 2017. [Online; accessed 10-April-2017].

[59] Russell M Taylor II, Thomas C Hudson, Adam Seeger, Hans Weber, Jeffrey Juliano, and Aron T Helser. Vrpn: a device-independent, network-transparent vr peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61. ACM, 2001.

[60] Virginia Tech. Hypercube. cube.sv.vt.edu, 2017. [Online; accessed 16-April-2017].

[61] Virginia Tech. Visionarium lab at virginia tech. http://vis.arc.vt.edu, 2017. [Online; accessed 24-April-2017].

[62] Daniel Terdiman. New technology revs up pixar's 'cars 2'. https://www.cnet.com/news/new-technology-revs-up-pixars-cars-2, 2011. [Online; accessed 15-April-2017].

[63] TurboVNC. A brief introduction to turbovnc. http://www.turbovnc.org/About/Introduction, 2017. [Online; accessed 10-April-2017].

[64] UCDavis. Vrui vr toolkit. http://idav.ucdavis.edu/~okreylos/ResDev/Vrui, 2017. [Online; accessed 14-April-2017].

[65] R Vickery, Joel Martin, J Fowler, R Moorehead, Yogi Dandass, T Atkinson, A Cedilnik, P Adams, and Jerry Clarke. Web-based high performance remote visualization. In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 364–369. IEEE, 2007.

[66] VirtualGL. A brief introduction to virtualgl. http://www.virtualgl.org/About/Introduction, 2017. [Online; accessed 12-April-2017].

[67] VirtualGL. User's guide for virtualgl 2.5.1. https://cdn.rawgit.com/VirtualGL/virtualgl/2.5.1/doc/index.html, 2017. [Online; accessed 14-April-2017].

[68] Muhammad Wannous and Hiroshi Nakano. Nvlab, a networking virtual web-based laboratory that implements virtualization and virtual network computing technologies. *IEEE Transactions on Learning Technologies*, 3(2):129–138, 2010.

[69] Wikipedia. Analysis of variance. https://en.wikipedia.org/w/index.php?title=Analysis_of_variance&oldid=775807211, 2017. [Online; accessed 20-April-2017].

[70] Wikipedia. Cave automatic virtual environment. https://en.wikipedia.org/w/index.php?title=Cave_automatic_virtual_environment&oldid=774185890, 2017. [Online; accessed 10-April-2017].

[71] Wikipedia. Lidar. https://en.wikipedia.org/w/index.php?title=Lidar&oldid=775297216, 2017. [Online; accessed 17-April-2017].

[72] Wikipedia. Message Passing Interface. https://en.wikipedia.org/w/index.php?title=Message_Passing_Interface&oldid=774632477, 2017. [Online; accessed 10-April-2017].

[73] Wikipedia. Rrdtool. https://en.wikipedia.org/w/index.php?title=RRDtool&oldid=768555171, 2017. [Online; accessed 14-April-2017].

[74] Wikipedia. Tunneling protocol. https://en.wikipedia.org/w/index.php?title=Tunneling_protocol&oldid=772954558, 2017. [Online; accessed 16-April-2017].

[75] Wikipedia. Treemapping. https://en.wikipedia.org/w/index.php?title=Treemapping&oldid=773824737, 2017. [Online; accessed 15-April-2017].

[76] Wikipedia. Virtualgl. https://en.wikipedia.org/w/index.php?title=VirtualGL&oldid=768309042, 2017. [Online; accessed 14-April-2017].

[77] Wikipedia. Welch's t-test. https://en.wikipedia.org/w/index.php?title=Welch%27s_t-test&oldid=760289070, 2017. [Online; accessed 20-April-2017].

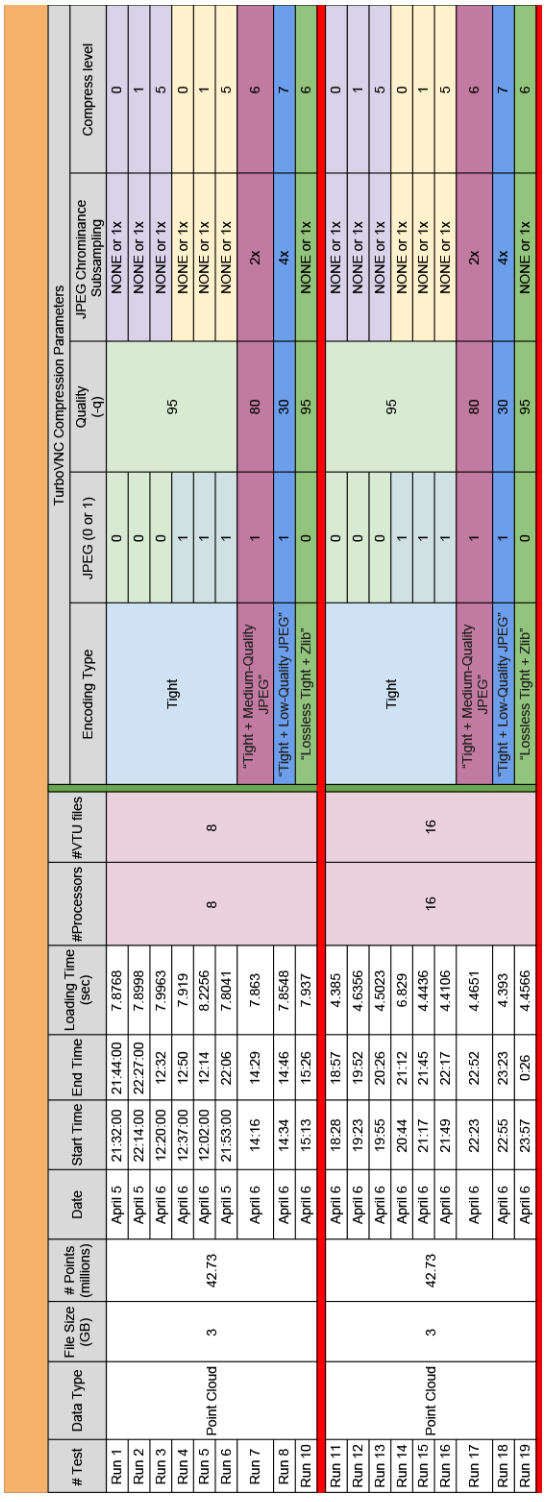
[78] Wikipedia. X window system. https://en.wikipedia.org/w/index.php?title=X_Window_System&oldid=769190233, 2017. [Online; accessed 11-April-2017].

[79] X.org. X.org foundation. https://www.x.org/wiki/, 2016. [Online; accessed 11-April-2017].

Appendices

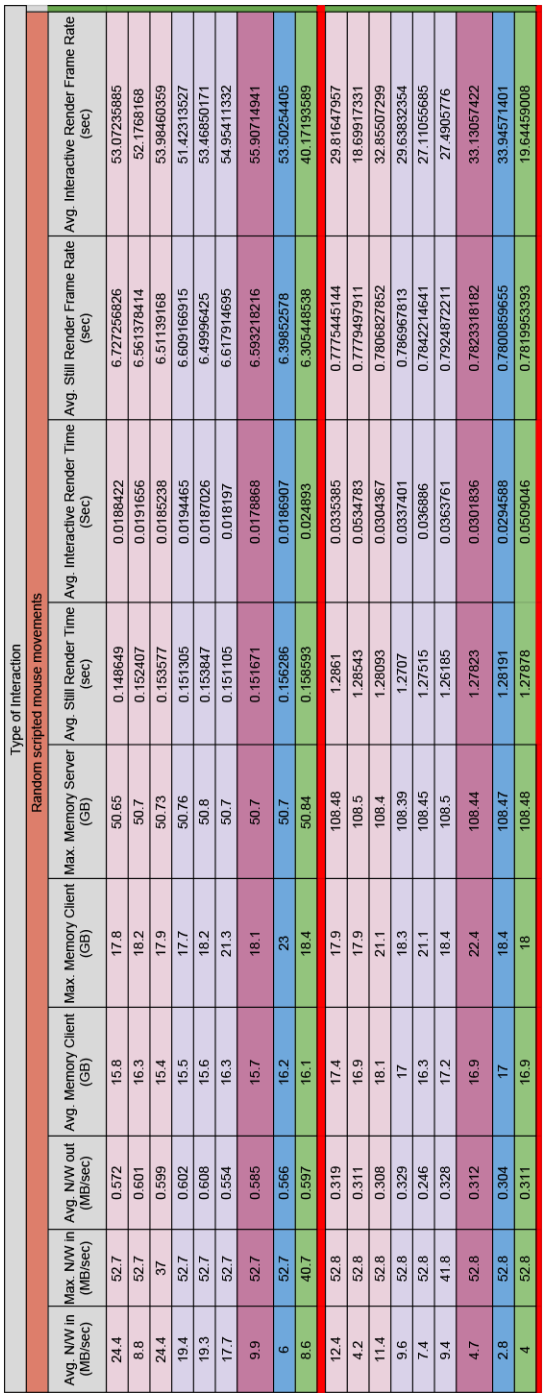
Appendix A

Experimental results



# Test	Data Type	File Size (GB)	# Points (millions)	Date	Start Time	End Time	Loading Time (sec)	#Processors	#VTU files	TurboVNC Compression Parameters				
										Encoding Type	JPEG (0 or 1)	Quality (-q)	JPEG Chrominance Subsampling	Compress level
Run 1	Point Cloud	3	42.73	April 5	21:32:00	21:44:00	7.8768	8	8	Tight	0	95	NONE or 1x	0
Run 2				April 5	22:14:00	22:27:00	7.8998				0		NONE or 1x	1
Run 3				April 6	12:20:00	12:32	7.9963				0		NONE or 1x	5
Run 4				April 6	12:37:00	12:50	7.919				1		NONE or 1x	0
Run 5				April 6	12:02:00	12:14	8.2256				1		NONE or 1x	1
Run 6				April 5	21:53:00	22:06	7.8041				1		NONE or 1x	5
Run 7				April 6	14:16	14:29	7.863			"Tight + Medium-Quality JPEG"	1	80	2x	6
Run 8				April 6	14:34	14:46	7.8548			"Tight + Low-Quality JPEG"	1	30	4x	7
Run 10				April 6	15:13	15:26	7.937			"Lossless Tight + Zlib"	0	95	NONE or 1x	6
Run 11	Point Cloud	3	42.73	April 6	18:28	18:57	4.385	16	16	Tight	0	95	NONE or 1x	0
Run 12				April 6	19:23	19:52	4.6356				0		NONE or 1x	1
Run 13				April 6	19:55	20:26	4.5023				0		NONE or 1x	5
Run 14				April 6	20:44	21:12	6.829				1		NONE or 1x	0
Run 15				April 6	21:17	21:45	4.4436				1		NONE or 1x	1
Run 16				April 6	21:49	22:17	4.4106				1		NONE or 1x	5
Run 17				April 6	22:23	22:52	4.4651			"Tight + Medium-Quality JPEG"	1	80	2x	6
Run 18				April 6	22:55	23:23	4.393			"Tight + Low-Quality JPEG"	1	30	4x	7
Run 19				April 6	23:57	0:26	4.4566			"Lossless Tight + Zlib"	0	95	NONE or 1x	6

Figure A.1: Combination of parameters used to run the experiments



Type of Interaction												
Random scripted mouse movements												
Avg. N/W in (MB/sec)	Max. N/W in (MB/sec)	Avg. N/W out (MB/sec)	Avg. Memory Client (GB)	Max. Memory Client (GB)	Max. Memory Server (GB)	Avg. Still Render Time (sec)	Avg. Interactive Render Time (Sec)	Avg. Still Render Frame Rate (sec)	Avg. Interactive Render Frame Rate (sec)
24.4	52.7	0.572	15.8	17.8	50.65	0.148649	0.0188422	6.727256826	53.07235885
8.8	52.7	0.601	16.3	18.2	50.7	0.152407	0.0191656	6.561378414	52.1768168
24.4	37	0.599	15.4	17.9	50.73	0.153577	0.0185238	6.51139168	53.98460359
19.4	52.7	0.602	15.5	17.7	50.76	0.151305	0.0194465	6.609166915	51.42313527
19.3	52.7	0.608	15.6	18.2	50.8	0.153847	0.0187026	6.49996425	53.46850171
17.7	52.7	0.554	16.3	21.3	50.7	0.151105	0.018197	6.617914695	54.95411332
9.9	52.7	0.585	15.7	18.1	50.7	0.151671	0.0178868	6.593218216	55.90714941
6	52.7	0.566	16.2	23	50.7	0.156286	0.0186907	6.398652578	53.50254405
8.6	40.7	0.597	16.1	18.4	50.84	0.158593	0.024893	6.305448538	40.17193589
12.4	52.8	0.319	17.4	17.9	108.48	1.2861	0.0335385	0.7775445144	29.81647957
4.2	52.8	0.311	16.9	17.9	108.5	1.28543	0.0534783	0.7779497911	18.69917331
11.4	52.8	0.308	18.1	21.1	108.4	1.28093	0.0304367	0.7806827852	32.85507299
9.6	52.8	0.329	17	18.3	108.39	1.2707	0.0337401	0.786967813	29.63832354
7.4	52.8	0.246	16.3	21.1	108.45	1.27515	0.036686	0.7842214641	27.11055685
9.4	41.8	0.328	17.2	18.4	108.5	1.26185	0.0363761	0.7924872211	27.4905776
4.7	52.8	0.312	16.9	22.4	108.44	1.27823	0.0301836	0.7823318182	33.13057422
2.8	52.8	0.304	17	18.4	108.47	1.28191	0.0294588	0.7800859655	33.94571401
4	52.8	0.311	16.9	18	108.48	1.27878	0.0509046	0.7819953393	19.64459008

Figure A.2: Results of the experiments