

# Lawrence Berkeley National Laboratory

## Lawrence Berkeley National Laboratory

### **Title**

Remote and Distributed Visualization Architectures

### **Permalink**

<https://escholarship.org/uc/item/5c78613g>

### **Author**

Bethel, E. Wes

### **Publication Date**

2012-11-01

# Remote and Distributed Visualization Architectures

E. Wes Bethel

Computational Research Division  
Lawrence Berkeley National Laboratory,  
Berkeley, California, USA, 94720.

Mark Miller

Lawrence Livermore National Laboratory,  
Livermore, CA, USA, 94551.

November 2012

## **Acknowledgment**

This work was supported by the Director, Office of Science, Office and Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Some of the research in this work used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## **Legal Disclaimer**

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor The Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or The Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or The Regents of the University of California.

## Abstract

The term *remote and distributed visualization* (RDV) refers to a mapping of visualization pipeline components onto distributed resources. Historically, the development of RDV was motivated by the user's need to perform analysis on data that was too large to move to their local workstation or cluster, or that exceeded the processing capacity of their local resources. RDV concepts are central in high performance visualization, where a visualization application is often a parallel application consisting of a collection of processing components working together to solve a large problem. This report presents RDV architectures and case studies that illustrate how these architectures are used in practice.

## Preface

The material in this technical report is a chapter from the book entitled *High Performance Visualization—Enabling Extreme Scale Scientific Insight* [2], published by Taylor & Francis, and part of the CRC Computational Science series.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Visualization Performance Fundamentals and Networks</b>	<b>6</b>
<b>3</b>	<b>Send-Images Partitioning</b>	<b>7</b>
<b>4</b>	<b>Send-Data Partitioning</b>	<b>8</b>
<b>5</b>	<b>Send-Geometry Partitioning</b>	<b>8</b>
<b>6</b>	<b>Hybrid and Adaptive Approaches</b>	<b>9</b>
<b>7</b>	<b>Which Pipeline Partitioning Works the Best?</b>	<b>10</b>
<b>8</b>	<b>Case Study: Visapult</b>	<b>12</b>
8.1	Visapult Architecture: The Send-Geometry Partition . . . . .	12
8.2	Visapult Architecture: The Send-Data Partition . . . . .	13
<b>9</b>	<b>Case Study: Chromium Renderserver</b>	<b>14</b>
<b>10</b>	<b>Case Study: VisIt and Dynamic Pipeline Reconfiguration</b>	<b>17</b>
10.1	How VisIt Manages Pipeline Partitioning . . . . .	17
10.2	Send-Geometry Partitioning . . . . .	17
10.3	Send-Images Partitioning . . . . .	18
10.4	Automatic Pipeline Partitioning Selection . . . . .	18
<b>11</b>	<b>Conclusion</b>	<b>18</b>

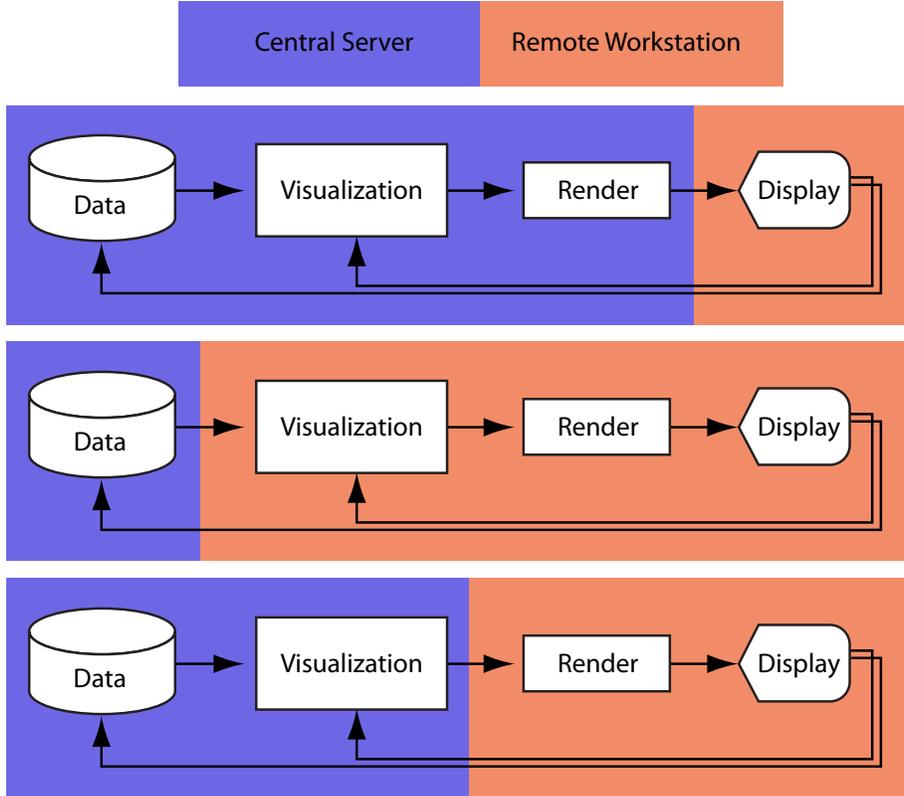


Figure 1: The three most common visualization pipeline partitionings in remote and distributed visualization. From top to bottom: send images, send data, and send geometry. Components in grey are located at the central computing facilities, and those in white are on the remote client. Image courtesy of E. Wes Bethel (LBNL).

# 1 Introduction

RDV concepts extend the idea of architectures that run on a single, large, parallel platform, without loss of generality, to configurations consisting of more than one computational platform. Historically, this type of architecture was motivated by the reality of a user needing to perform visualization on a data set that was either too large to move across the network or that exceeded the capacity of their local computational platform. The RDV solution to this problem was the idea that part of the visualization pipeline would be executed “close to” or colocated with the large data, and then a smaller subset of data or visualization results would be sent to the user’s local machine for additional processing and display.

From a high level, there are three fundamental types of bulk payload data that move between components of the visualization pipeline: “scientific data,” visualization results (geometry and renderable objects), and image data [6, 19]. In some instances and applications, the portion of the pipeline that moves data between components is further resolved to distinguish between raw, or unprocessed, and filtered data [11, 12, 25]. For simplicity, these three partitioning strategies are referred to as *send images*, *send data*, and *send geometry*, as shown in Figure 1. The first few sections of this report discuss each of these strategies, as well as *hybrid* variants that are not strictly one of these three, but a combination of aspects from more than one of these three partitionings. The question of which pipeline partitioning works best is the subject of Section 7.

In reality, each or any of these individual stages of the visualization pipeline may be comprised of distributed or parallel resources. Three different case studies will be presented that

explore alternative configurations of parallel components, comprised of high performance, remote and distributed visualization architectures. The Visapult architecture, which uses a hybrid partitioning combined with custom network protocols and that the Supercomputing Conference’s Network Bandwidth Challenge three years in a row, is the subject of Section 8. Chromium Renderserver, which consists of a distributed-memory parallel scalable back-end rendering infrastructure that can deliver imagery to a remote client through an industry-standard protocol, is the subject of Section 9. Dynamic pipeline configuration across distributed resources in the VisIt application is the subject of Section 10.

## 2 Visualization Performance Fundamentals and Networks

Since RDV is about distributing components of the visualization pipeline, the overall performance of a distributed system is influenced in part by the following factors:

- *Component performance*: for a given amount of input data, each of the different components in the pipeline will likely have different performance characteristics. Some components are more computationally expensive than others.
- *Inter-component communication*: moving data between components incurs a cost as well. The cost, or time required, is typically greater across wide-area than local-area networks.
- *Data characteristics and visualization algorithms*: some visualization algorithms, such as isocontouring, are sensitive to the characteristics of input data. For example, running an isocontouring algorithm on a “noisy” data set will result in a significantly greater amount of output than when run on a “smooth” data set of the same size. In extreme cases, the isocontouring algorithm can produce more output than input.
- *Data size*: the overall pipeline performance is influenced by the amount of data to be processed.

Of these factors, the report’s focus is on the cost of inter-component communication, since it can have a surprising impact on overall pipeline performance. Consider the following use scenario, where the visualization pipeline is split across two machines in some unknown partitioning. First, a client (user) at a remote machine requests a new image, due, perhaps to changing a visualization parameter, such as an isocontouring level, a new data set, or a change in viewpoint. The client transmits a message to a server requesting an update. The server performs some processing, then transmits results back to the client. This exchange requires, at least, two messages to transit the network. The time required for this two-way exchange is a function of both network latency and capacity.

Whereas network capacity, or bandwidth, refers to the amount of data that can move over a network, per unit of time, latency is the amount of time required to move a single byte between two points in a network. Latency is a combination of several kinds of delays incurred in the processing of network data. Latency values can range from less than a millisecond for local area networks up to tens or hundreds of milliseconds for cross-country network connections. To understand how latency affects performance, consider the following scenario: a client requests a new image, the server generates a new image, then sends the new image to the client. This communication pattern requires two stages of communication, one from the client to server, and the other from the server back to the client. If assuming an infinite bandwidth (network speed), then the maximum potential frame rate for a given amount of latency,  $L$ , measured in milliseconds is  $1000/2L$ . When  $L = 50$  ms, then the maximum possible frame rate is 10 frames per second, regardless of the capacity of the underlying network or the amount of data moving between server and client. The performance of some pipeline partitionings are more sensitive to latency and capacity than others.

### 3 Send-Images Partitioning

In a send-images partitioning, all processing needed to compute a final image is performed on a server, then the resulting image data is transmitted to a client. Over the years, there have been several different approaches to implement this partitioning strategy.

The Virtual Network Computing (VNC) [26] system uses a client–server model for providing remote visualization and display services to virtually any application using a client–server model of operation: the custom client viewer on the local machine intercepts events and sends them to the VNC server running at the remote location; the server detects updates to the screen generated by either the windowing system or the graphics/visualization application, then packages up those updated regions of pixels; and, finally, sends them to the client for display, using one of a different number of user-configurable compression and encoding strategies. In its original design and implementation, VNC didn’t support applications that used hardware-acceleration for rendering images. Paul et al. 2008 [24] describe a scalable rendering architecture, which is the subject of Section 9, that combines with VNC and overcomes this limitation for scalable rendering and image delivery.

OpenGL Vizserver [14] and VirtualGL [10] use a client–server model for the remote delivery of hardware-accelerated rendering of live-running applications. OpenGL Vizserver provides a remote image delivery of the contents of an application’s OpenGL rendering window. VirtualGL uses two different communication channels: one for encoding and transmission of the OpenGL’s window pixels, and another for Xlib traffic. In a sense, VirtualGL’s implementation falls partly into the hybrid category, since it sends both images and the “draw commands” formed by the Xlib command stream.

Some visualization applications, such as VisIt [18] and ParaView [16], support a send-images mode of operation, where a scalable visualization and rendering back end provides images to a remote client. In these approaches, a remote client component requests a new frame; the server-side component responds by executing the local part of the visualization pipeline to produce and transmit a new image. Interestingly, VisIt supports a dynamic pipeline configuration model, where it can select between send-images and send-geometry partitionings at runtime. (For a detailed discussion, see Section 10.)

Chen et al. 2006 [7] describe a remote visualization architecture and implementation, MBender, that allows for the exploration of precomputed visualization results in a way that gives the appearance of semi-constrained interactive visualization. Their approach is akin to that of QuickTime VR [8] (QTVR) “object movies,” which allow navigation through a 3D array of images. In contrast, QuickTime “panorama movies” consist of images taken or rendered from a fixed point where the view varies across azimuthal angles; object movie images typically represent viewpoints from different “latitude and longitude” positions, but with a common “look at” point, and with zoom-in and zoom-out options. Whereas QTVR supports navigation through up to three dimensions of images, Chen’s MBender architecture supports navigation through  $N$ -dimensional image arrays, which is better suited for scientific visualization. Each of the  $N$  dimensions represents a variation across a single visualization parameter: isocontouring level, slicing, temporal evolution, and so forth. In practice, this approach has a significant space requirement for the source images, which are requested via a standard web server by a custom Java-based client, and works best when  $N$  is relatively small.

The primary advantage of the send-images partitioning is that there is an upper bound on the amount of data that moves across the network. That upper bound is a function of image size,  $I_s$ , rather than the size of the data set,  $D_s$ , being visualized. Typically, when  $D_s \gg I_s$ , the send-images partitioning has favorable performance characteristics, when compared to the others.

Its primary disadvantage is related to its primary advantage: there is a minimum amount of data, per frame, that must move across the network. The combination of latency to produce the frame and the time required to move it over the network may be an impediment on interactive levels of performance. For example, if the user desires to achieve a 30 frame-per-second through-

put rate, and each frame is 4MB in size,<sup>1</sup> then, assuming zero latency, the network must provide a minimum of 120MB/s of bandwidth. Some systems, such as VNC, implement optimizations—compression and sending only the portion(s) of the screen that changes—to reduce the size of per-frame pixel payload.

The other disadvantage of send-images is the potential impact of network latency on interactivity. As discussed earlier in Section 2, network latency will impose an upper bound on absolute frame rate. This upper bound may be sufficiently high on local area networks to support interactive visualization when using the send-images approach, but may be too low on wide area networks. For example, achieving 10 frames per second is possible only on networks having less than 100 ms of round-trip latency: for  $1000/2L \geq 10$  fps, then  $L \leq 50$  ms.

## 4 Send-Data Partitioning

The send-data partitioning aims to move scientific data from server to client for visualization processing and rendering. The scientific data may be the “source data,” prior to any processing, or it may be source data that has undergone some sort of processing, such as noise-reduction filtering, or a computation of a derived field. In the former case, the visualization pipeline, which resides entirely on the user/client machine, will perform “remote reads” to obtain data. In the latter case, some portion of the visualization pipeline may reside on the server, but the data, moved between server and client, consists of the same type and size of data, as if the client were performing remote reads.

Optimizing this type of pipeline partitioning can take several different forms. One form is to optimize the use of distributed network data caches and replicas, so a request for data goes to a “nearby” rather than “distant” source [1, 30], as well as to leverage high performance protocols that are more suitable than a TCP for bulk data transfers [1]. Other optimizations leverage data subsetting, filtering, and progressive transmission from remote sources to reduce the amount of data payload crossing the network [11, 23]. Some systems, like the Distributed Parallel Storage System (DPSS), provide a scalable, high performance, distributed-parallel data storage system that can be optimized for data access patterns and the characteristics of the underlying network [31].

Another approach is to use alternative representations for the data being sent from server to client. Ma et al. 2002 [20] describe an approach used for the transmission of particle-based data sets in which either particles, or server-computed particle density fields, or both, are moved between server and client. The idea is that, in regions of high particle density, it may make more sense to render such regions using volume rendering, rather than using point-based primitives: in high density regions, point-based primitives would likely result in too much visual clutter. In their implementation, a user defines a transfer function for creating two derived data sets: one is a collection of particles that lie in “low density” regions, the other is a computed 3D density field that is intended to be smaller in size than the original particle field in “high density” regions. Then, the server component transmits both types of data sets to the client for rendering.

In practice, the send-data approach may prove optimal when two conditions hold true: (1) the size of the source data is relatively small, and (2) interactivity is a priority. However, as the size of scientific data grows, it is increasingly impractical to move full-resolution source data to the user’s machine for processing, since the size of data may exceed the capacity of the user’s local machine, and moving large amounts of data over the network may be cost-prohibitive.

## 5 Send-Geometry Partitioning

In the send-geometry partitioning, the payload, moving between the server and client, is “drawable” content. For visualization, a class of visualization algorithms, often referred to as “mappers,” will transform scientific data, be it mesh-based or unstructured, and produce renderable

---

<sup>1</sup>1024<sup>2</sup> pixels, each of which consists of RGB $\alpha$  tuples, one byte per color component, and no compression.

geometry as an output [27]. In a send-geometry partitioning, the server component runs data I/O and visualization algorithms, producing renderable geometry, then transmits this payload to the client for rendering. One way to optimize this path is to send only those geometric primitives that lie within a view frustum. Such optimizations have proven useful in network-based walkthroughs of large and complex data [9]. Frustum culling, when combined with occlusion culling and level-of-detail selection at the geometric model level, can result in reduced transmission payload [17]. Both of those approaches require the server to have awareness of the client-side state, namely viewing parameters.

The send-geometry approach is included as part of several production applications, research applications, and libraries. The applications VisIt,<sup>2</sup> Ensight,<sup>3</sup> and ParaView<sup>4</sup> all implement a form of send-geometry (as well as a form of send-images). Unmodified OpenGL and X11-based applications may be configured, through an environment variable, to transmit Xlib [22] or GLX [15] protocol “draw commands,” respectively, from an application running on a server to a remote client.

The Chromium library [13] is a “drop-in” replacement for OpenGL that provides the ability to route an OpenGL command stream from an application to other machines for processing. A typical Chromium use scenario is for driving tiled displays, where each display tile is connected to a separate machine. Chromium will intercept the application’s OpenGL command stream, and, via a user-configurable “stream processing” infrastructure, will route OpenGL commands to the appropriate machine for subsequent rendering. The application emitting OpenGL calls may be either a serial or a parallel application. Chromium provides constructs for an application to perform draw-time synchronization operations like barriers. Chromium has several other user-configurable “stream processing units” that also provide a form of send-images partitioning. Chromium serves as the basis for Chromium Renderserver, which is a high performance system for remote and distributed visualization (see Section 9). Chromium includes several types of processing capabilities, important for parallel rendering operations—render-time synchronization and barriers—to support sort-first parallel, hardware-accelerated volume rendering, complete with level-of-detail-based model switching to accelerate rendering [3].

One disadvantage of the send-geometry approach is the potential size of the renderable geometry payload. In some circumstances, the size of this payload may exceed the size of the original data set, or it may be so large as to exceed the capacity of the client to hold in memory all at once for rendering. Streaming approaches are one mechanism for accommodating rendering data sets too large for a client’s memory, yet the relatively slower network connection may be a more significant barrier.

The primary advantage of the send-geometry approach is that, once the geometry content is resident in the client’s memory, the client may be capable of very high rendering frame rates. This approach may be the best when: (1) the geometry payload fits entirely within the client memory, and (2) interactive client-side rendering rates are a priority.

## 6 Hybrid and Adaptive Approaches

Hybrid pipeline partitioning approaches are those that do not fall strictly into one of the above categories. Adaptive approaches are those where the application will alter its partitioning strategy at runtime to achieve better performance in light of changing condition, such as slower network speeds or a shift in the relative amount of altering between data, geometry, image data, moved between server and client.

The Visapult system [4, 5, 28], described in more detail later in Section 8, employs an architecture that leverages the concept of *co-rendering*, which is best thought of as a hybrid approach. In this approach, the server performs partial rendering of the source data set, and then transmits these partial rendering results to the client. The data sent by the server is not strictly geometry

---

<sup>2</sup>VisIt visualization application website: <http://www.llnl.gov/visit>.

<sup>3</sup>Ensign visualization application website: <http://www.ceisoftware.com>.

<sup>4</sup>ParaView visualization application website: <http://www.paraview.org>.

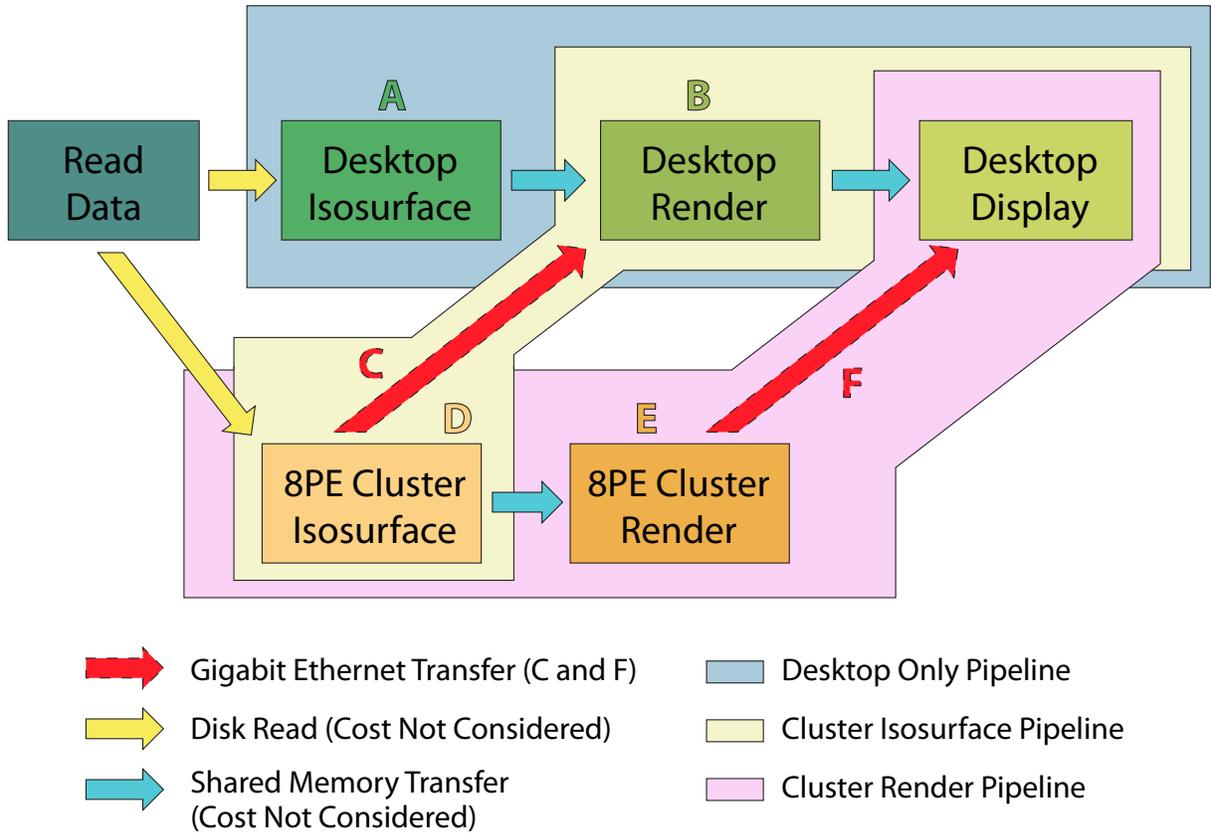


Figure 2: Three potential partitionings of a remote and distributed visualization pipeline performance experiment showing execution components and data flow paths. Colored arrows indicate each of the potential data flow paths. Image courtesy of E. Wes Bethel and John Shalf (LBNL).

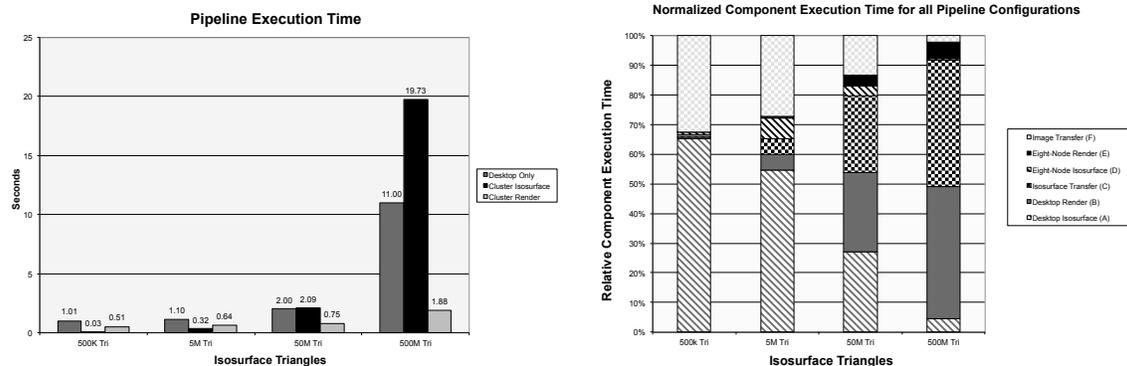
nor images: it is direct volume rendering (images) of subsets of the source data set. Once on the client, these partial renderings are further rendered in an interactive fashion to the user, who can interactively change the viewpoint, etc.

This approach, which hides latency from the user, strikes a trade-off between interactivity and data load, although the pipeline partitioning is static. If  $O(n^3)$  volume data is input to the server, then the server moves  $O(n^2)$  image-based data to the client. The trade-off is that the client has full interactive rendering capabilities of a reduced-sized version of the source data: it is able to render at full hardware-accelerated rates, but with known accuracy/fidelity limitations.

The VisIt visualization application uses an adaptive approach to decide when to use send-images vs. send-geometry. This adaptive approach is described in more detail later in Section 10.

## 7 Which Pipeline Partitioning Works the Best?

As part of an article examining how the Grid will affect the design and use of remote and distributed visualization systems, Shalf and Bethel 2003 [29] present a study that aims to determine which pipeline partitioning works the best. The main idea in the study is to set up three different partitionings of a common visualization algorithm and use various combinations of computational platforms to run different stages of the pipeline. They measure absolute end-to-end execution time for each pipeline partition as well as the time required for each individual component in each different pipeline configuration.



(a) Absolute runtime for each pipeline. *Desktop Only* performance is the sum of components A and B. *Cluster Isosurface* performance is the sum of components B, C and D. *Cluster Render* performance is the sum of components D, E and F. (b) Relative performance of execution components in the three pipelines. The desktop-only pipeline is the sum of  $A + B$ ; the cluster isosurface pipeline is the sum of  $B + C + D$ ; the cluster render pipeline is the sum of  $D + E + F$ .

Figure 3: The three potential partitionings have markedly different performance characteristics, depending on many factors, including some that are dependent upon the data set being visualized. Images courtesy of John Shalf and E. Wes Bethel (LBNL).

Figure 2 shows three potential ways to partition the visualization pipeline and shows the components of the visualization pipeline used to create an image of an isosurface. The links between the blocks indicate data flow. The *Desktop Only* pipeline places the isosurface and rendering components on the desktop machine. The *Cluster Isosurface* pipeline places a distributed-memory implementation of the isosurface component on the cluster, and transfers triangles over the network to a serial rendering component on the desktop. The *Cluster Render* pipeline performs isosurface extraction and rendering on the cluster, then transfers images to the desktop for display. The colors used in Figure 3a correspond to the groupings of components shown in Figure 2: dark gray for the Desktop Only pipeline, black for the Cluster Isosurface pipeline, and light gray for the Cluster Render pipeline.

For this study, the following conditions, optimizations, and simplifying assumptions are in effect:

- All geometry produced by the isosurface component are triangle strips. Each incremental triangle of the strip is represented with a single vertex, which consumes twenty-four bytes: six four-byte floats containing vertex and normal data.
- All graphics hardware is capable of rendering 50M triangles/second. An 8-node system, each node equipped with identical graphics hardware, can render 400M triangles/second.
- The image transfer assumes a 24-bit, high definition,  $1920 \times 1080$  pixels, CIF framebuffer.
- Interconnects use a 1 GB network with perfect performance.
- The performance model does not consider the cost of data reads, the cost of scatter-gather operations, or the cost of displaying cluster-rendered images on the desktop.

Whereas Figure 3a shows the absolute runtime of each pipeline for varying numbers of triangles, Figure 3b presents the relative runtime of all components for varying numbers of triangles. In Figure 3b, the absolute runtime of all components is summed, totaling 100%, and the size of each colored segment in each vertical bar shows the relative time consumed by that particular component. Each component’s execution time is normalized by the sum of all component execution times, so one may quickly determine which components, or network transfers, will dominate the execution time. Note, some components are used in more than one pipeline configuration.

The vertical bars in Figure 3b are color-coded by component: those in dark gray, cross-hatched and solid, are desktop-resident, and show the cost of computing isosurfaces on the desktop (A) and rendering isosurfaces on the desktop (B) in both the desktop-only and cluster-isosurface configurations. Those in a checkerboard pattern show the cost of network transfers of either image data (F), or isosurface data from the cluster to the desktop (C). Those in black, solid and cross-hatched, show the cost of computing isosurfaces (D) and rendering them (E) on the cluster. The components are arranged vertically so the reader can visually integrate groups of adjacent components into their respective pipeline partitionings.

In the 500K triangles case, the cost of desktop isosurface extraction dominates in the Desktop Only pipeline. In contrast, the Cluster Isosurface pipeline would perform very well—about six times faster. In the 500M triangles case, the Cluster Render pipeline is about five times faster than the Desktop Only pipeline, and about eight times faster than the Cluster Isosurface pipeline.

This study reveals that the best partitioning varies as a function of the performance metric. For example, the absolute frame rate might be the most important metric, where a user performs an interactive transformation of 3D geometry produced by the isocontouring stage. The partitioning needed to achieve a maximum frame rate will vary according to the rendering load and rendering capacity of pipeline components.

Surprisingly, the best partitioning can also be a function of a combination of the visualization technique and the underlying data set. The authors’ example uses isocontouring as the visualization technique and changes in the isocontouring level will produce more or less triangles. In turn, this varying triangle load will produce different performance characteristics of any given pipeline partitioning. The partitioning that “works best” for a small triangle count may not be the best for a large triangle count. In other words, the optimal pipeline partitioning can change as a function of a simple parameter change.

## 8 Case Study: Visapult

Visapult is a highly specialized, pipelined and parallel, remote and distributed, visualization system [5]. It won the ACM/IEEE Supercomputing Conference series High Performance Bandwidth Challenge three years in a row (2000–2002). Visapult, as an application, is composed of multiple software components that are executed in a pipelined-parallel fashion over wide-area networks. Its architecture is specially constructed to hide latency over networks and to achieve ultra-high performance over wide-area networks.

The Visapult system uses a multistage pipeline partitioning. In an end-to-end view of the system, from bytes on disk or in simulation memory, to pixels on screen, there are two separate pipeline stages. One is a send-geometry partitioning, the other is a send-data partitioning.

### 8.1 Visapult Architecture: The Send-Geometry Partition

The original Visapult architecture, described by Bethel et al. 2000 [5], shown in Figure 4, consists of two primary components. One component is a parallel *back end*, responsible for loading scientific data and performing, among other activities, “partial” volume rendering of the data set. The volume rendering consists of applying a user-defined transfer function to the scalar data, producing an  $RGB\alpha$  volume, then performing axis-aligned compositing of these volume subsets, producing semi-transparent textures. In a typical configuration, each data block in the domain decomposition will result in six volume rendered textures: one texture for each of the six principal axis viewing directions. As a result, if the Visapult back end loads  $O(n^3)$  data, it produces  $O(n^2)$  output in the form of textures.

Then, the back end transmits these “partially rendered” volume subsets to the *viewer*, shown in Figure 4 as **Partially Rendered Payload**, where the subsets are stored as textures in a high performance scene graph system in the viewer. The viewer, via the scene graph system, renders these semi-transparent textures on top of proxy geometry in the correct back-to-front order at

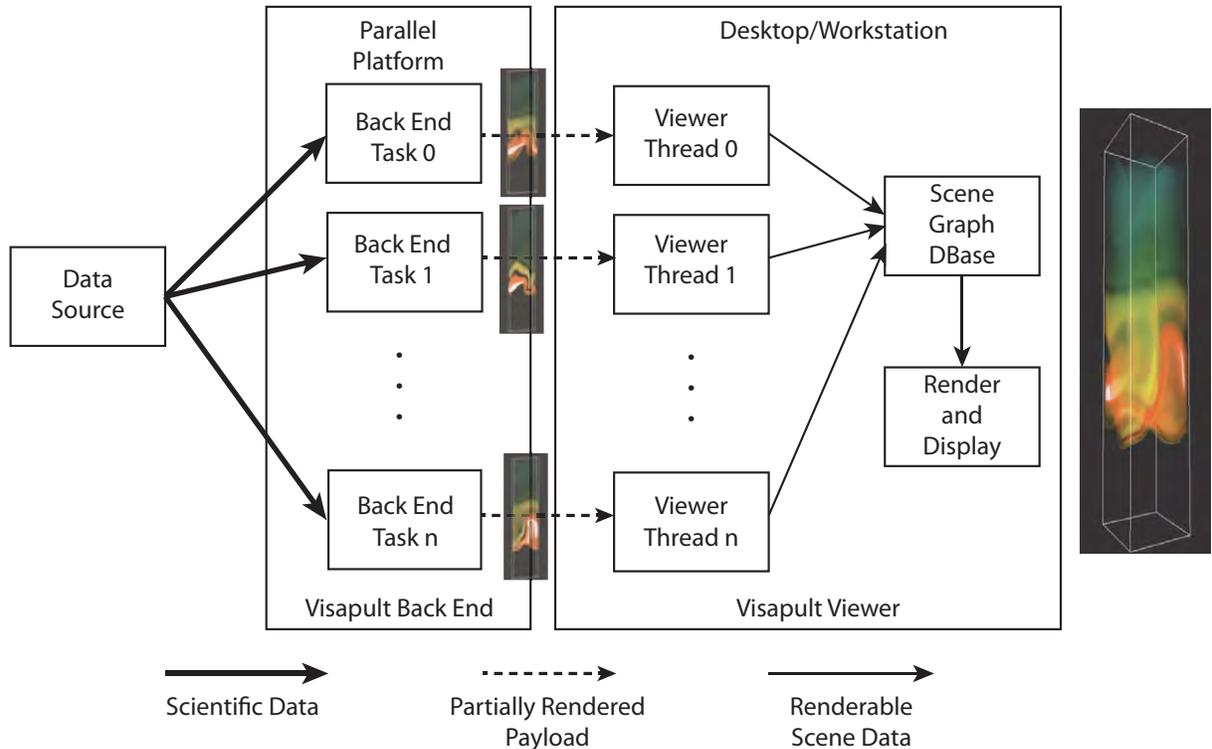


Figure 4: Visapult’s remote and distributed visualization architecture. Image courtesy of E. Wes Bethel (LBNL).

interactive rates via hardware-acceleration. Inside the viewer, the scene graph system switches between each of the six source textures for each data block depending upon camera orientation to present the viewer with the best fidelity rendering.

This particular “co-rendering” idea—where the server performs partial rendering and the viewer finishes the rendering—was not new to Visapult. Visapult’s architecture was targeted at creating a high performance, remote and distributed visualization implementation of an idea called image-based rendering assisted volume rendering described by Mueller et al. 1999 [21].

## 8.2 Visapult Architecture: The Send-Data Partition

Like all visualization applications, Visapult needs a source of data from which to create images. Whereas modern, production-quality visualization applications provide robust support for loading a number of well-defined file formats, Visapult’s data source for all SC Bandwidth Challenge runs were remotely located network data caches as opposed to files.

In the SC 2000 implementation, source data consisted of output from a combustion modeling simulation. The data was stored on a Distributed Parallel Storage System (DPSS), which can be thought of as a high-speed, parallel remote block-oriented data cache [31]. In this implementation, the Visapult back end invoked DPSS routines, similar in concept to POSIX `fread` calls that, in turn, loaded blocks of raw scientific data in parallel, from a remote source and relied on the underlying infrastructure, the DPSS client library, to efficiently move data over the network.

In an effort to make an even better use of the underlying network, Shalf and Bethel, 2003 [28], extended Visapult to make use of a UDP-based “connectionless” protocol. They connected to

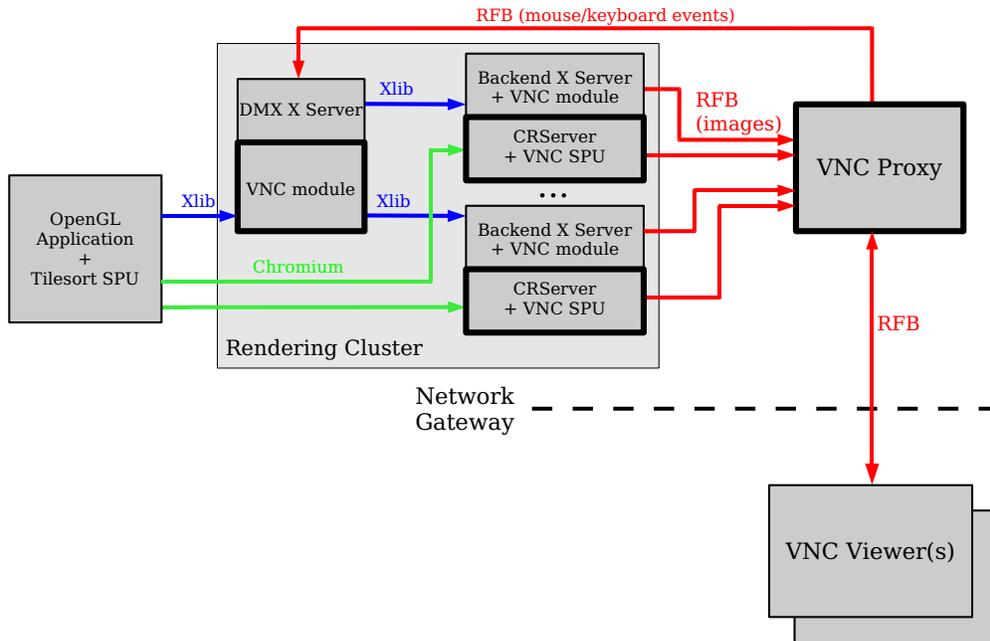


Figure 5: CRRS system components for a two-tile DMX display wall configuration. Lines indicate primary direction of data flow. System components outlined with a thick line are new elements from this work to implement CRRS; other components outlined with a thin line existed in one form or another prior to the CRRS work. Image source: Paul et al. 2008 [24].

a freely-running simulation, which provided a data source. This change resulted in the ability for the Visapult back end to achieve unprecedented levels of network utilization, close to 100% of the theoretical line rate, for sustained periods of time.

The TCP-based approach can be thought of as a process of “load a timestep’s worth of data, then render it.” Additionally, because the underlying protocol is TCP-based, there was no data loss between remote source and the Visapult back end.

Going to the UDP-based model required rethinking both the network protocol and the Visapult back end architecture. In the TCP approach, the Visapult back end “requests” data, which is a “pull” model. In the UDP approach, though, the data source streams out data packets as quickly as possible, and the Visapult back end must receive and process these data packets as quickly as possible. This approach is a “push” model. There is no notion of timestep or frame boundary in this push model; the Visapult back end has no way of knowing when all the data packets, for a particular timestep, are in memory. After all, some of the packets may be lost as UDP does not guarantee packet delivery. See Bethel and Shalf, 2005, [4] for more design change details and see Shalf and Bethel, 2003, [28] for the UDP packet payload design.

## 9 Case Study: Chromium Renderserver

Paul et al. 2008 [24] describe Chromium Renderserver (CRRS), which is a software infrastructure that provides the ability for one or more users to run and view image output from unmodified, interactive OpenGL and X11 applications on a remote, parallel computational platform, equipped with graphics hardware-accelerators, via industry-standard Layer 7 network protocols and client viewers.

Like Visapult, CRRS has a multi-stage pipeline partitioning that uses both send-geometry and send-images approaches. Figure 5 shows a high-level architectural diagram of a two-node CRRS system. A fully operational CRRS system consists of six different software components,

which are built around VNC's remote framebuffer (RFB) protocol. The RFB protocol is a Layer 7 network protocol, where image data and various types of control commands are encoded and transmitted over a TCP connection, between producer and consumer processing components. The motivation for using RFB is because it is well understood, and there exist (VNC) viewers for nearly all the current platforms. One of the CRRS design goals is to allow an unmodified VNC viewer application to be used as the display client in a CRRS application.

The CRRS general components are:

- The application is any graphics or visualization program that uses OpenGL and/or Xlib for rendering. Applications need no modifications to run on CRRS, but they must link with the Chromium *faker* library rather than the normal OpenGL library. Figure 6 shows visual output from an interactive molecular docking application being run under the CRRS system, to produce a high-resolution image output on a  $3 \times 2$  tiled display, with the image being gathered, encoded, and sent to another machine for remote display and interaction.
- The VNC viewer, which displays the rendering results from the application.
- The Chromium VNC Stream Processing Unit (SPU), which obtains and encodes the image pixels produced by the OpenGL command, stream from the application and sends the pixels to a remote viewer.
- Distributed Multihead X (DMX), which is an X-server that provides the ability for an Xlib application to run on a distributed-memory parallel cluster.
- The VNC Proxy is a specialized VNC Server that takes encoded image input from VNC Servers and VNC SPUs, running on each of the parallel rendering nodes and transmits the encoded image data to the remote client(s). The VNC Proxy solves the problem of synchronizing the rendering results of the asynchronous Xlib and OpenGL command streams.<sup>5</sup>
- The VNC Server X Extension is present on the X server at each parallel rendering node. This component harvests, encodes and transmits the portion of the framebuffer modified by the Xlib command stream.

In a CRRS system, the send-geometry partitioning exists between the potentially parallel visualization application—which loads scientific data, performs visualization processing, and emits OpenGL draw commands—and a parallel machine, where those draw commands are routed by the Chromium Tilesort Stream Processing Unit [13]. In this part of the system, the visualization or graphics application will *push* draw commands over the wire to the rendering nodes.

There are actually two stages of send-images partitioning in the CRRS pipeline. The first exists between each of the nodes where parallel rendering occurs and the VNC Proxy component. The second exists between the remote viewer (a VNC viewer) and the centrally located VNC Proxy component. Whereas the send-geometry partitioning operates under a *push model*, the send-images partitioning operates under a *pull model*. In this mode of operation, the remote VNC Viewer will request an image update from the VNC Proxy. In turn, the VNC Proxy requests RFB-based image updates from each of the rendering nodes, which, in effect, is running a VNC server in the form of the VNC Chromium Stream Processing Unit.

The authors of this study examined the end-to-end performance of several different types of optimizations, listed below. Collectively, all these optimizations, when enabled, produce a 20–25% performance improvement on all network types—local area network and three types of wide area networks. The optimizations fall into two broad classes: those that reduce end-to-end system latency and those that reduce the amount of payload moving between systems.

- *RFB caching* improves performance by maintaining a cache of RFB Update messages that contain encoded image data. The VNC Proxy responds to RFB Update Requests by

---

<sup>5</sup>See <http://vncproxy.sourceforge.net> for more information about the VNC Proxy and CRRS setup and operation.

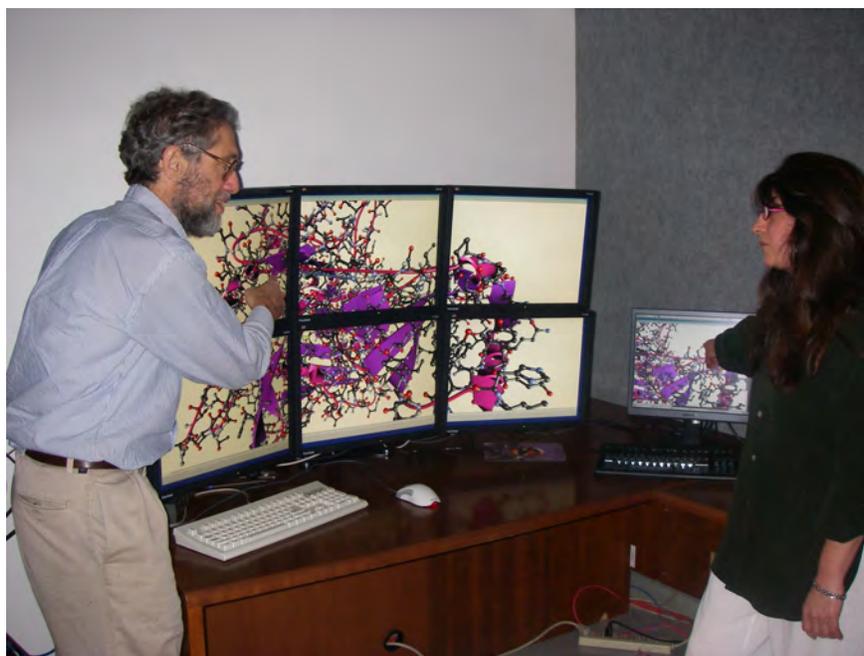


Figure 6: An unmodified molecular docking application, run in parallel, on a distributed-memory system using CRRS. Here, the cluster is configured for a  $3 \times 2$  tiled display setup. The monitor for the “remote user machine” in this image is the one on the right. While this example has a “remote user” and a “central facility” connected via a local area network, the model typically used is one where the remote user is connected to the central facility via a low-bandwidth, high-latency link. Here, the image shows the complete  $4800 \times 2400$  full-resolution image from the application running on the parallel, tiled system, appearing in a VNC viewer window on the right. Through the VNC viewer, the user interacts with the application to change the 3D view and orientation, as well as various menus on the application to select rendering modes. As is typically the case, this application uses Xlib-based calls for GUI elements (menus, etc.) and OpenGL for high performance model rendering. Image source: Paul et al. 2008 [24].

sending cached responses, rather than encoding, possibly multiple times, the contents of its internal VFB. This optimization helps to reduce end-to-end latency.

- *Bounding box tracking.* Here, only the portions of the scene rendered by OpenGL that have changed are rendered and transmitted to the remote client. This optimization helps to reduce the amount of RFB image payload.
- *Double buffering.* By maintaining a double-buffered VFB in the VNC SPU, two operations can occur simultaneously—application rendering and image encoding/transmission. This optimization helps to reduce end-to-end latency.
- *Frame synchronization.* While not strictly an optimization, this feature is needed to synchronize parallel rendering streams and to prevent frame dropping (spoiling), when images are rendered more quickly than they can be delivered to the remote client.

Another interesting element of this study was the examination of image compression algorithm combinations and their end-to-end performance, when used over various types of networks. The VNC system supports several types of image encoding/compression methods, the use of which is a user-selectable parameter. Some forms of compression are lossless; they produce a relatively lower levels of compression, but they execute relatively quicker. Other forms of compression are lossy; they produce relatively greater degrees of compression and execute rel-

atively slower. The study identifies a trade-off in which lossless compression algorithms are better suited, from a performance perspective, on local area networks (LAN), which have a lower latency and higher bandwidth. In this setting, the runtime of the compression algorithm became a bottleneck. In contrast, on wide area networks (WAN), the extra runtime required for performing lossy compression was outweighed by the performance gain, resulting from sending less image payload data over a slower, higher latency network.

The two-stage CRRS design, which integrates send-geometry and send-images, combined with using the industry-standard RFB protocol, resulted in a system that was highly flexible, scalable, and that provided the ability to support a potentially large number of visualization and graphics applications accessible to remote users via a common, industry-standard viewer.

## 10 Case Study: VisIt and Dynamic Pipeline Reconfiguration

The VisIt parallel visualization application supports both send-images and send-geometry pipeline partitionings, as well as a dynamic, runtime change from one partitioning to another, depending on several factors. This section describes how each of these partitionings work, along with the methodology VisIt uses for deciding to change from one pipeline partitioning to another.

### 10.1 How VisIt Manages Pipeline Partitioning

VisIt uses a construct, known as an `avtPlot` object, to manage pipeline execution and different forms of pipeline partitioning. An `avtPlot` object exists, in part, in both VisIt's server and client components. The two parts manage pipeline partitioning and also optimize the amount of computation used to update a visual output (plot), as inputs change. For example, some inputs impact only how geometry is rendered. Other inputs impact the geometry itself. The `avtPlot` object differentiates between these update operations and ensures that the appropriate parts of the pipeline are executed, only when inputs effecting them change.

An `avtPlot` object knows which parts of a pipeline require execution, as inputs change. This knowledge, then, affects how VisIt manages pipeline partitioning for any given plot. In a send-geometry partitioning, an input change that affects only the rendering step does not require any interaction with the server because all the computation can and will be handled on the client. On the other hand, when VisIt uses a send-images partitioning to produce and deliver the very same plot, then the same change in inputs will, indeed, involve interaction with the server. Nonetheless, even in this circumstance, the server will execute only those portions of the pipeline necessary to re-render the plot.

### 10.2 Send-Geometry Partitioning

The send-geometry mode of pipeline partitioning is typically most appropriate when visualizing small input data sets, where the results of visualization algorithms, such as renderable geometry, can fit entirely in the client's memory. In VisIt, the definition of "small" is determined by a user-specifiable parameter, indicating a maximum size threshold for the amount of geometry data produced by the server and sent to the client. This is known as the Scalable Rendering Threshold (SRT).

In this partitioning, the client can vary the visualization of a plot in a variety of ways, without requiring any further exchanges of data with the server. This includes variations in viewing parameters, such as pans, zooms and rotations, changes in transfer function, transparency, glyph types used for glyphed plots, and so forth. In addition, when visualizing a time-series and sufficient memory is available, the client can cache geometry for each timestep. This cache thereby enables the user to quickly animate the series, as well as vary the visualization in these ways, without the need for interactions with the server.

If, during the course of a run, the amount of geometry produced by visualization algorithms increases, due perhaps to processing larger or more complex data sets, it may exceed the client's ability to store and process in its entirety. Either the client will have to accept and render geometry in a streaming fashion, or the server will have to take on the responsibility for rendering. VisIt employs the latter approach. This is VisIt's send-images mode of operation.

### 10.3 Send-Images Partitioning

The send-images mode of pipeline partitioning is known as the *scalable rendering* mode in VisIt. It is typically used for large-scale data where the amount of geometry to be displayed exceeds the SRT. In the send-images partitioning, VisIt's server does all the work to compute the geometry to be visualized and then renders the final image. After, it sends this image to the client for a final display to the user. In parallel, each processor renders a portion of the geometry to the relevant pixels into a local z-buffered image. Results from different processors' local image are z-buffer-composited using a sort-last approach via a user-defined MPI reduction operator, optimized for compositing potentially large images (e.g.,  $16K^2$  pixels). The server then sends the final image, with or without the associated z-buffer, to the client for display to the user. The z-buffer will be included if the client window into which the results will be displayed is also displaying results from other server processes. Use of the z-buffer in this fashion is what enables VisIt to properly handle multiple servers, each potentially employing a different pipeline partitioning but displaying to the user, via the same integrated visual output.

### 10.4 Automatic Pipeline Partitioning Selection

VisIt offers the user the ability to decide: to always or never use a send-images partitioning for execution; or to use an automatic mechanism for switching between send-geometry and send-images.

For the automatic selection mode, VisIt uses heuristics to decide which pipeline partitioning is best for a given situation. The heuristic relies on having an estimate of the geometry's volume data that would overwhelm the client either because it will require too much memory to store or because it will take too long to render. This estimate is governed by the SRT user-settable threshold. It is a count of polygons of which VisIt estimates the client can reasonably accommodate. VisIt does not measure this value explicitly, but instead relies on the user's judgment regarding the performance characteristics of the machine where the client is run, and then sets this threshold accordingly. By default, the SRT is set at two million polygons.

For each window, in the client employing a send-geometry partitioning, the server keeps track of the total polygon count of all plots being displayed there. When the count exceeds the SRT, VisIt automatically switches all the plots in that window to a send-images partitioning. When and if the polygon count drops below the SRT, VisIt will then automatically switch all plots to a send-geometry partitioning. There is some hysteresis included in this process to prevent oscillations in partitioning when the polygon count hovers right around the SRT.

For a parallel server, changes in pipeline partition are complicated somewhat, by the fact that the SRT can be exceeded when results from only some of the processors have been accumulated. Therefore, VisIt may have to do a reversal midway through a presumed send-geometry pipeline execution and restart it as a send-images partitioning.

## 11 Conclusion

RDV architectures are those in which the components of the visualization processing pipeline are distributed across multiple computational platforms. The differences between different RDV architectures tend to center around how the visualization pipeline is decomposed across distributed platforms. Some architectures move images from one machine to another, others move renderable geometry between machines, while others move raw scientific data between machines. It is an open question which form of partitioning works best. There is no one single solution that

is optimal across all combinations of use scenario, specific data set and visualization/processing techniques, and computational and networking infrastructure.

There are a number of different RDV implementations, ranging from single applications, like VisIt, that have some form of RDV processing capability, to more general purpose solutions, like the Chromium Renderserver, which provide general purpose RDV capabilities along with scalable rendering, for many different applications. Interestingly, the notion of *in situ* processing includes a form of distributed visualization processing as one potential implementation path.

## References

- [1] Micah Beck, Terry Moore, and James S. Plank. An End-to-end Approach to Globally Scalable Network Storage. In *SIGCOMM '02: Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 339–346, New York, NY, USA, 2002. ACM Press.
- [2] E. Wes Bethel, Hank Childs, and Charles Hansen, editors. *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, Boca Raton, FL, USA, November 2012. <http://www.crcpress.com/product/isbn/9781439875728>.
- [3] E. Wes Bethel, Greg Humphreys, Brian Paul, and J. Dean Brederson. Sort-First, Distributed Memory Parallel Visualization and Rendering. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 41–50, Seattle, WA, USA, October 2003.
- [4] E. Wes Bethel and John Shalf. Consuming Network Bandwidth with Visapult. In Chuck Hansen and Chris Johnson, editors, *The Visualization Handbook*, pages 569–589. Elsevier, 2005. LBNL-52171.
- [5] E. Wes Bethel, Brian Tierney, Jason Lee, Dan Gunter, and Stephen Lau. Using High-Speed WANs and Network Data Caches to Enable Remote and Distributed Visualization. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, Dallas, Texas, USA, 2000. LBNL-45365.
- [6] Ian Bowman, John Shalf, Kwan-Liu Ma, and E. Wes Bethel. Performance Modeling for 3D Visualization in a Heterogeneous Computing Environment. Technical report, Lawrence Berkeley National Laboratory, Berkeley, CA, USA, 94720, 2004. LBNL-56977.
- [7] Jerry Chen, Ilmi Yoon, and E. Wes Bethel. Interactive, Internet Delivery of Visualization via Structured, Prerendered Multiresolution Imagery. *IEEE Transactions in Visualization and Computer Graphics*, 14(2):302–312, 2008. LBNL-62252.
- [8] Shenchang Eric Chen. QuickTime VR—An Image-Based Approach to Virtual Environment Navigation. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 29–38, Los Angeles, CA, USA, 1995.
- [9] Daniel Cohen-Or and Eyal Zadicario. Visibility Streaming for Network-based Walk-throughs. In *Proceedings of Graphics Interface*, pages 1–7, 1998.
- [10] D. R. Commander. VirtualGL. <http://www.virtualgl.org>, 2011.
- [11] H. Hege, A. Hutanu, R. Kähler, A. Merzky, T. Radke, E. Seidel, and B. Ullmer. Progressive Retrieval and Hierarchical Visualization of Large Remote Data. In *Proceedings of the Workshop on Adaptive Grid Middleware*, pages 60–72, September 2003.
- [12] Hans-Christian Hege, André Merzky, and Stefan Zachow. Distributed Visualizaton with OpenGL VizServer: Practical Experiences. ZIB Preprint 00-31, 2001.
- [13] 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. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 693–702, San Antonio, TX, USA, 2002.
- [14] Ken Jones and Jenn McGee. OpenGL Vizserver User's Guide Version 3.5. Technical report, Silicon Graphics Inc., Mountain View, CA, USA, 2005. Document Number 007-4245-014.
- [15] Mark J. Kilgard. *OpenGL programming for the X Window System*. OpenGL Series. Addison-Wesley Developers Press, 1996.
- [16] Kitware, Inc. and Jim Ahrens. ParaView: Parallel Visualization Application. <http://www.paraview.org/>.

- [17] Dieter Kranzlmüller, Gerhard Kurka, Paul Heinzlreiter, and Jens Volkert. Optimizations in the Grid Visualization Kernel. In *IEEE Parallel and Distributed Processing Symposium (CDROM)*, pages 129–135, 2002.
- [18] Lawrence Livermore National Laboratory. VisIt: Visualize It Parallel Visualization Application. <http://www.llnl.gov/visit/>.
- [19] Eric J. Luke and Charles D. Hansen. Semotus Visum: A Flexible Remote Visualization Framework. In *VIS '02: Proceedings of the Conference on Visualization '02*, pages 61–68, Boston, MA, USA, 2002.
- [20] Kwan-Liu Ma, Greg Schussman, Brett Wilson, Kwok Ko, Ji Qiang, and Robert Ryne. Advanced Visualization Technology for Terascale Particle Accelerator Simulations. In *Proceedings of Supercomputing 2002 Conference*, pages 19–30, Baltimore, MD, USA, November 2002.
- [21] Klaus Mueller, Naeem Shareef, Jian Huang, and Roger Crawfis. IBR Assisted Volume Rendering. In *Proceedings of IEEE Visualization, Late Breaking Hot Topics*, pages 5–8, October 1999.
- [22] A. Nye. *Xlib Programming Manual: For Version 11 of the X Window System*. Definitive guides to the X Window System. O'Reilly & Associates, 1992.
- [23] Valerio Pascucci and Randall J. Frank. Global Static Indexing for Real-time Exploration of Very Large Regular Grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (CDROM)*, Denver, CO, USA, 2001.
- [24] Brian Paul, Sean Ahern, E. Wes Bethel, Eric Brugger, Rich Cook, Jamison Daniel, Ken Lewis, Jens Owen, and Dale Southard. Chromium Renderserver: Scalable and Open Remote Rendering Infrastructure. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):627–639, May/June 2008. LBNL-63693.
- [25] Steffen Prohaska, Andrei Hutanu, Ralf Kahler, and Hans-Christian Hege. Interactive Exploration of Large Remote Micro-CT Scans. In *VIS '04: Proceedings of the Conference on Visualization '04*, pages 345–352, Austin, TX, USA, 2004.
- [26] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
- [27] Will Schroeder, Kenneth M. Martin, and William E. Lorensen. *The Visualization Toolkit (2nd ed.): An Object-Oriented Approach to 3D Graphics*. Prentice-Hall, Upper Saddle River, NJ, USA, 1998.
- [28] John Shalf and E. Wes Bethel. Cactus and Visapult: A Case Study of Ultra-High Performance Distributed Visualization Using Connectionless Protocols. *IEEE Computer Graphics and Applications*, 23(2):51–59, March/April 2003. LBNL-51564.
- [29] John Shalf and E. Wes Bethel. How the Grid Will Affect the Architecture of Future Visualization Systems. *IEEE Computer Graphics and Applications*, 23(2):6–9, May/June 2003.
- [30] Mohammad Shorfuzzaman, Peter Graham, and Rasit Eskicioglu. Adaptive Popularity-Driven Replica Placement in Hierarchical Data Grids. *Journal of Supercomputing*, 51(3):374–392, 2010.
- [31] Brian Tierney, Jason Lee, Brian Crowley, Mason Holding, Jeremy Hylton, and Fred L. Drake. A Network-Aware Distributed Storage Cache for Data Intensive Environments. In *Proceedings of IEEE High Performance Distributed Computing Conference (HPDC-8)*, pages 185–193, Redondo Beach, CA, USA, 1999.