1. Parallel ParaView

One of the main purposes of ParaView is to allow users to create visualizations of large data sets that reside on parallel systems without first collecting the data to a single machine. This chapter describes the concepts behind the parallelism in ParaView. Then three different modes of running ParaView in parallel are discussed: distributed stand-alone mode, client/server mode, and client/data server/render server mode. The remainder of the chapter describes ParaView's parallel rendering features (i.e., distributed rendering, offscreen rendering, and tiled displays).

1.1. Parallel Structure

ParaView has three main logical components: client, data server, and render server. The client is responsible for the user interface of the application. ParaView's general-purpose client was written to be flexible and can be customized using XML specifications. The client can also be replaced with a completely new GUI as required by application specifications.

The data server is primarily constructed from VTK readers, source, and filters. It is responsible for reading and processing data sets to create final geometric models needed for rendering. VTK partitioning, ghost levels, and synchronous parallel filters are responsible for handling data parallelism on the data server. Each data server process has an identical VTK pipeline, and each process is told which partition of the data it should load.

The render server is responsible for rendering the final geometry. Like the data server, the render server can run in parallel and creates identical visualization pipelines (only the rendering portion of the pipeline) on all of its processes. Having the ability to run the render server separately from the data server allows the optimal division of labor between computing

platforms. Most large computing clusters are primarily used for batch simulations and do not have hardware rendering resources. Since it is not desirable to move large data files to a separate visualization system, the data server can run on the same cluster that ran the original simulation. The render server can be run on a separate visualization cluster that has hardware rendering resources.

It is possible to run the render server with fewer processes than the data server, but never more. Visualization clusters typically have fewer nodes than batch simulation clusters, and processed geometry is usually significantly smaller than the original simulation dump. ParaView repartitions the geometric models on the data server before they are sent to the render server.

The client is a single-process program that connects to and communicates with both servers via the server manager. Since it is useful to control the ParaView visualization system from a desktop workstation, the client can be run on a separate machine from the servers. ParaView has the option of using parallel rendering on the render server or rendering directly on the client workstation. ParaView automatically chooses a rendering strategy to achieve the best rendering performance. Although small models may be collected on the client, ParaView's distributed rendering works well for models of all sizes.

ParaView can be run in many different configurations. In the simplest case the client, data server, and render server all run on the same process. In the most extreme case they are run as three separate programs: the client as a single-process program, the data server and render server as MPI multi-process programs. MPI is used to send messages between processes on a server, and socket connections are used to send messages between separate servers and between the servers and the client.

Running ParaView as a single-process application is simple. Simply run the `paraview` executable from the command line (or double-click the ParaView icon on Windows).

```
./paraview
```

The other configurations for running ParaView will be discussed in the remaining sections of this chapter.

1.2. Distributed Stand-Alone Mode

Although ParaView is designed from the ground up to be a parallel application, by default ParaView is built without parallel support. This is because there are so many different versions of MPI, the library ParaView uses for parallel communication. To use ParaView's parallel features, you must first compile ParaView with MPI support as described in chapter XX.

When ParaView is compiled with MPI, the `paraview` executable can be run in parallel, putting ParaView in its distributed stand-alone mode. In this case, the data server nodes and the render server nodes share the same processes. The client will execute on node 0 of the MPI group, which is also shared by node 0 of the data server and node 0 of the render server. The following will start ParaView in this mode with four processes. (The example is using the MPICH distribution of MPI; starting an executable with your MPI distribution may differ from the example shown.)

```
mpirun -np 4 ./paraview
```

A description of the four logical nodes of the above example follows.

- Node 0: data server node 0, render server node 0, client
- Node 1: data server node 1, render server node 1
- Node 2: data server node 2, render server node 2
- Node 3: data server node 3, render server node 3

1.3. Client / Server Mode

There are many reasons for ParaView to use multiple computers during a single ParaView session. One of the simplest examples is when the data is on a remote computer (or group of computers) and the user wants to visualize the data on a local desktop machine. Instead of copying the data to the local workstation, a ParaView server can be run on the remote computer(s), and a ParaView client can be run on the local workstation. The two programs communicate to create a single ParaView session. The server loads and processes the data, and the client creates and displays the graphical user interface that allows the user to interact with the data. In this mode both the data server and render server share the same processes, and the client is completely separate.

By default, the ParaView client actively connects to the ParaView server through a socket connection, so the server has to be started first. The server is a separate executable called `pvserver`. The client is started by running the `pvclient` executable and supplying the appropriate command-line arguments as shown below.

```
./pvserver

./pvclient --server-host=server_host
```

For the client, the command-line argument `--server-host` (or `-sh`) is used to specify where the server is running. (The default value of `server-host` is `localhost`.) If the `--server-port` option is used on the client command-line, it specifies which port will be used for the socket connection between the client and the server. (The default is `11111`.) For

obvious reasons, if the `--server-port` option is given to one executable, it must be given to both.

If the computer running the server will be behind a firewall, it is useful to have the server connect to the client instead of the client connecting to the server. The command-line option `--reverse-connection` (or `-rc`) is used on both the client and server command lines. When the connection between the client and the server is reversed, the client executable (`pvclient`) must be started first, and the `--client-host` argument (if used) is specified on the server command line to indicate to the server how to connect to the client. The `--server-port` option has the same name and meaning for both the forward and reverse connections.

```
./pvclient --reverse-connection

./pvserver --reverse-connection --client-host=client_host
```

The server can also be run as an MPI program with multiple processes, but the client should always be run as a single process. Instructions for starting a program with MPI are implementation- and system-dependent, so contact your system administrator for information about starting an application with MPI.

When ParaView is run in client/server mode, all data processing occurs on the server. This includes generation of the polygonal representation of the full data set and decimated LOD models. However, rendering can occur on either the server or the client depending on which is most efficient. In many cases, the polygonal representation of the data set is much smaller than the original data set. (In an extreme case, a simple outline may be used to represent a very large structured mesh.) In these cases, it may be better to send the polygonal representation to the client for rendering. If the client workstation has high-performance rendering hardware, even large data sets can be interactively rendered on the client.

The second option is to have each node of the server render its geometry and send the resulting images to the client for display. Since there is a penalty per rendered frame for compositing images and sending the image across the network, it may not make sense to render on the server when the data set's geometry is very small. However, ParaView's image compositing and delivery is very fast and there are many options to ensure interactive rendering. It is therefore often better to render data remotely on the server even when the data is only moderately large.

1.4. Render Server

The render server allows you to have a separate group of machines (i.e., apart from the data server and the client) to perform rendering. This means that you can select specialized rendering machines to do the parallel rendering rather than relying on the data server

machines, which may have limited or no rendering capabilities. In ParaView, the number of machines (N) composing the render server must be no more than the number (M) composing the data server. There are two sets of connections that must be made for ParaView to run in render-server mode. The first connection set is between the client and the first node of each of the data and render servers. The second connection set is between the nodes of the render server and the first N nodes of the data server. The first connection set is initially established either from the servers to the client or vice versa. The second is started by either the data server or the render server. Once all of these connections are established, they are bi-directional. The diagram in Figure 1 depicts the connections established when ParaView is running in render server mode. Each double-ended arrow indicates a bi-directional connection from one machine to another. In all the diagrams in this section, the render server nodes are denoted by RS 0, RS 1, …, RS N. The data server nodes are similarly denoted by DS 0, DS 1, …, DS N, …, DS M.

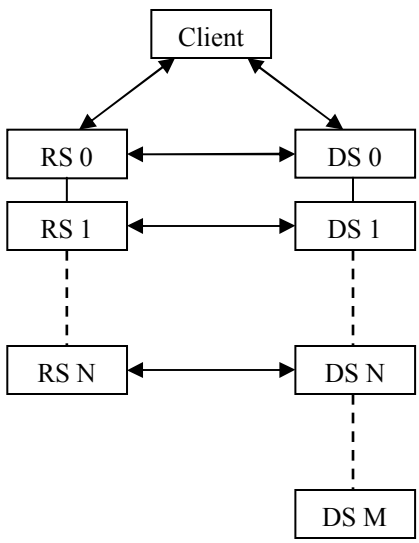


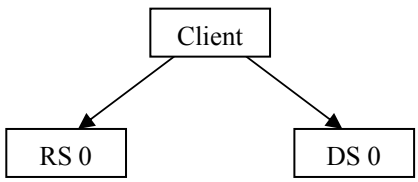
Figure 1. Connections required in render server mode

The establishment of connections between client and servers can either be forward (from client to servers) or reverse (from servers back to client). Likewise, the connections between render servers and data servers can be either from data server to render server or from render server to data server. The main reason for reversing the direction of any of the initial connections is that from behind a firewall a machine is able to initiate a connection to a machine outside the firewall, but not vice versa. If the data server was behind a firewall, the servers should initiate the connection with the client, and the data server nodes should connect to the render server nodes. If the render server is behind a firewall, still the servers should connect to the client, but now the render server nodes should initiate the connections with the nodes of the data server.

In the remaining diagrams in this section, each arrow indicates the direction in which the connection is initially established. Double-ended arrows indicate bi-directional connections that have already been established. In the example command lines, optional arguments are enclosed in `[]`'s. The rest of this section will be devoted to discussing the two connections required for running ParaView in render server mode.

Connection1: Connecting the client and servers

The first connection that must be established is between the client and the first node of both the data and render servers. By default, the client initiates the connection to each server, as shown in Figure 2. In this case, both the data server and the render server must be running before the client is started.




```
./pvdataserver [--data-server-port=data_server_port]

./pvrenderserver [--render-server-port=render_server_port]

./pvclient --client-render-server --data-server-
host=data_server_host0 --render-server-host=render_server_host0
[--data-server-port=data_server_port] [--render-server-
port=render_server_port]
```

Figure 2. Starting ParaView in render-server mode using standard connections.

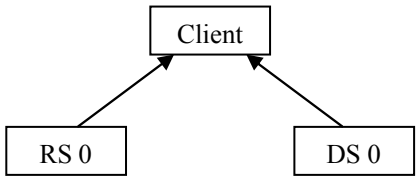
This is similar to running ParaView in client/server mode, but with the addition of a render server. The command lines for starting the client and the servers in this manner are also given in Figure 2. On the command line for the client, you will typically want to specify the data server host (node 0 of the data server) and the render server host (node 0 of the render server) because the default for both of these is `localhost`. Either of these can be specified as a machine name or as an IP address. You can also specify which ports to use in connecting the client to the render server and the data server, but it is best to use the default port numbers unless you must specify a specific port to open on a firewall. The data server port (for the connection between the client and the data server) is specified on the command line for the data server and the client using the `--data-server-port` option. (The default is `11111`.) The port specified on both the client and data server command lines must match, so if you use this option, specify the port on both command lines. To set up a similar connection for the render server, use the `--render-server-port` command-line option when starting the client and the render server. (This defaults to `22221`.) The port specified on both

the client and the render server command lines must also match, so if this option is used it must be specified both on the render server and client command lines. In the rest of the examples in this section, the `--data-server-port` and `--render-server-port` options will be omitted because they do not usually need to be specified.

In the above command lines, abbreviations can be used for certain command-line arguments. These abbreviations will be used in the rest of the example command lines in this section. (Some of the abbreviations listed here will be introduced later in this chapter. There are no abbreviations for `--port`, `--render-port`, and `--render-node-port`.)

- `--client-render-server` `-crs`
- `--data-server-host` `-dsh`
- `--render-server-host` `-rsh`
- `--reverse-connection` `-rc`
- `--connect-data-to-render` `-d2r`
- `--connect-render-to-data` `-r2d`

The connection between the client and the servers can also be initiated by the servers. In this case, the client must be started before both servers (similar to reversing the connection in client-server mode). The diagram indicating the initial connections is shown in Figure 3.




```
./pvclient -crs -rc

./pvrenderserver -ch=client -rc

./pvdataserver -ch=client -rc
```

Figure 3. Reversing the connections between the servers and the client.

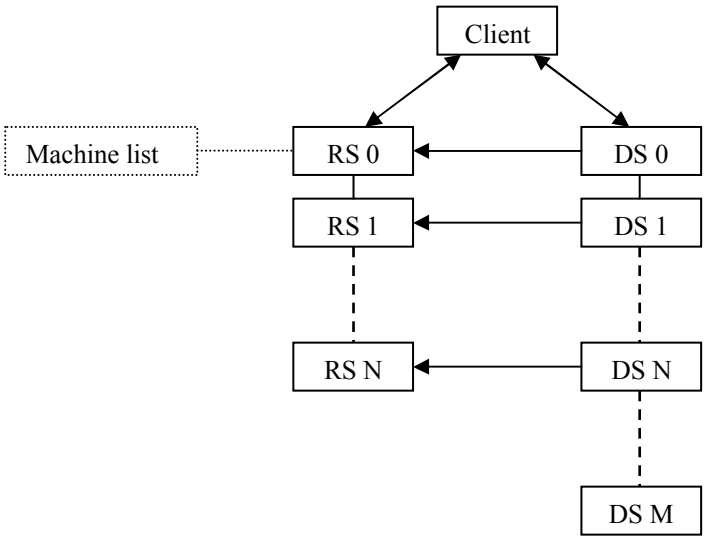
To do this, you must add `--reverse-connection` (or `-rc`) to the command lines for the data server, render server, and client. Also `--client-host` (or `-ch`) should appear on the data server and render server command lines; in both cases, the value of this command-line argument should be either the machine name or IP address of the client.

For the remainder of this chapter, `-rc` will be used instead of `--reverse-connection` when the connection between the client and the servers is to be reversed.

Connection 2: Connecting the render and data servers

Before the connection between the servers can be established, there must first be a connection from each server to the client, as described in the previous section. Once the connection to the client has been made, one server can access the necessary information to connect to the other server. It retrieves this information from the waiting server via the client. This information is then used to establish a connection between the nodes of the render server and the first N nodes of the data server.

In the default case, each node of the data server connects to a corresponding node of the render server, as shown in Figure 4. The information that the data server needs per node of the render server is a machine name and a port number. The machine names are specified in ParaView's XML configuration file that is passed to the each executable (pvdataserver, pvrenderserver, and pvclient) on the command line. The format of the XML configuration file is described in appendix XX. By default, the port number per machine is randomly generated. If you instead wish to specify a single port number to use in connecting to each node of the render server, include the `render-node-port` argument in the XML configuration file. ParaView does not allow you to specify a different port per machine.

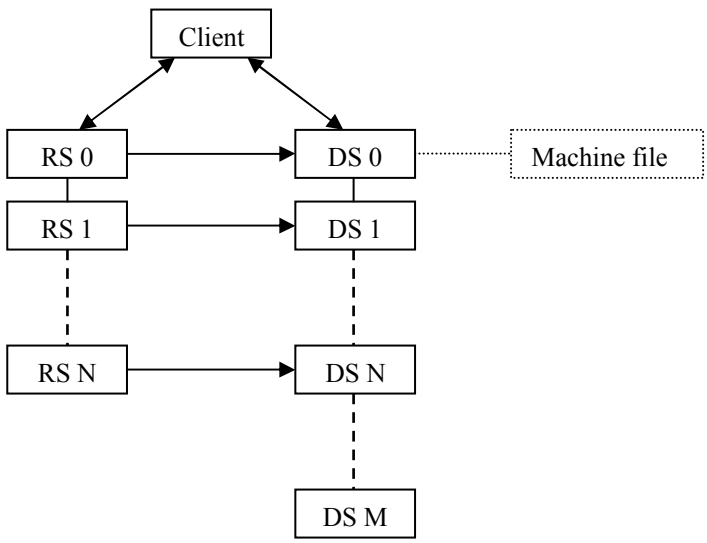



```
./pvdataserver config.pvx

./pvrenderserver config.pvx [--render-node-
port=render_node_port]

./pvclient config.pvx -crs -dsh=data_server_host0 -
rsh=render_server_host0
```

Figure 4. Initializing the connection from the data server to the render server.

If you wish to explicitly indicate that the data server will be connecting to the render server, add the `--connect-data-to-render` (or `-d2r`) option to the *client* command line. However, since this is the default behavior, using this command-line argument is not necessary. The command lines shown in Figure 4 indicate a standard connection between the client and the servers.

The direction of the initial connection between the nodes of the two servers may also be reversed (i.e., the render server nodes connect to those of the data server) as shown in Figure 5. Typically when this connection is reversed, the direction of the connection between the client and the servers is also reversed (e.g., if the render server is behind a firewall). This means that the render server will retrieve the machine names and ports from the data server via the client. This information will then be used to establish a connection from the nodes of the render server to the first N nodes of the data server. The XML configuration file should list the machine names of the first N nodes of the data server; if used, the `render-node-port` option should be specified there as well.




```
./pvclient config.pvx -crs -rc -r2d

./pvdataserver config.pvx -rc -ch=client

./pvrenderserver config.pvx -rc -ch=client
```

Figure 5. Reversing the connection between the servers and client, and connect the render server to the data server.

The command-line option for changing the direction of the connection between the servers is `--connect-render-to-data` (or `-r2d`); it is specified on the *client* command line. In the command lines shown in Figure 5, the direction of connection is reversed both between the client and the servers and between the nodes of the data and render servers.

1.5. Parallel Rendering / Compositing

When ParaView is run in parallel, either in stand-alone mode or in client/server (or client/data server/render server) mode, the final image in the display area has contributions from multiple processes. If the data is small enough, the geometry can be collected to one process for rendering. Usually it is more efficient to leave the geometry distributed and employ ParaView's parallel rendering in which images from each rendering process are collected and composited to form a single image for displaying in the display area of the user interface. In parallel, there are additional controls on the **General** tab of the **3D View Properties** property sheet for specifying how rendering should occur. In client/server mode, the **LOD Parameters** portion of the **General** tab appears as shown in Figure 6.

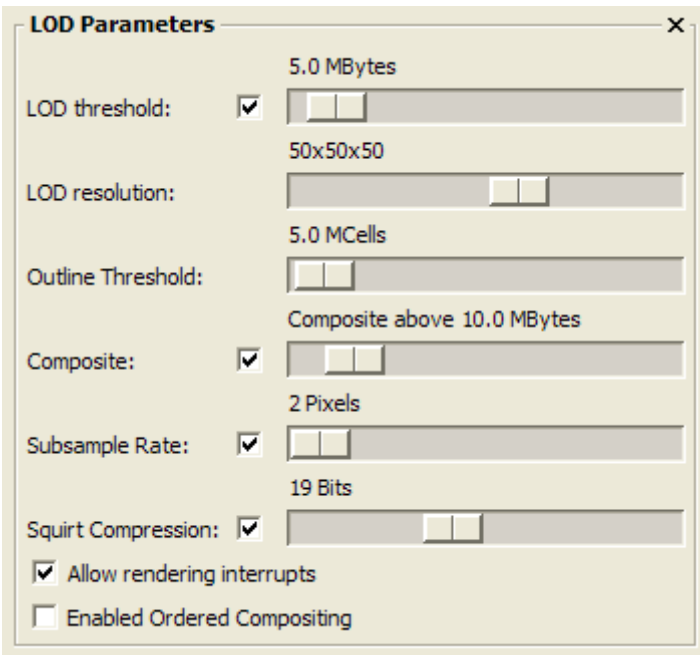


Figure 6. Parallel LOD parameters

LOD threshold, **LOD resolution**, and **Outline Threshold**: These options control the geometric levels of detail. They have the same meaning in parallel as they do in serial rendering mode. A detailed description of their meanings can be found in chapter XX.

Composite: This slider determines how large the data set must be in order for parallel rendering with image compositing and delivery to be used (as opposed to collecting the geometry). The value of this slider is measured in megabytes. The size of the entire data set must consume more than the specified amount of memory for compositing of images to occur. If the check box beside the **Composite** slider is unmarked, then compositing will not happen; the geometry will always be collected. This is only a reasonable option when you can be sure the data set you are using is very small. In general, it is safer to move the slider to the right than to uncheck the box.

When compositing is turned on, ParaView uses **IceT** to perform image compositing. IceT is a parallel rendering library that takes multiple images containing partial geometry and combines them into a single image. IceT employs several image compositing algorithms, all of which are designed to work well on a distributed memory machine. Examples of two such image compositing algorithms are demonstrated in Figure 7 and Figure 8. IceT will automatically choose a compositing algorithm based on the current workload and computing resources.

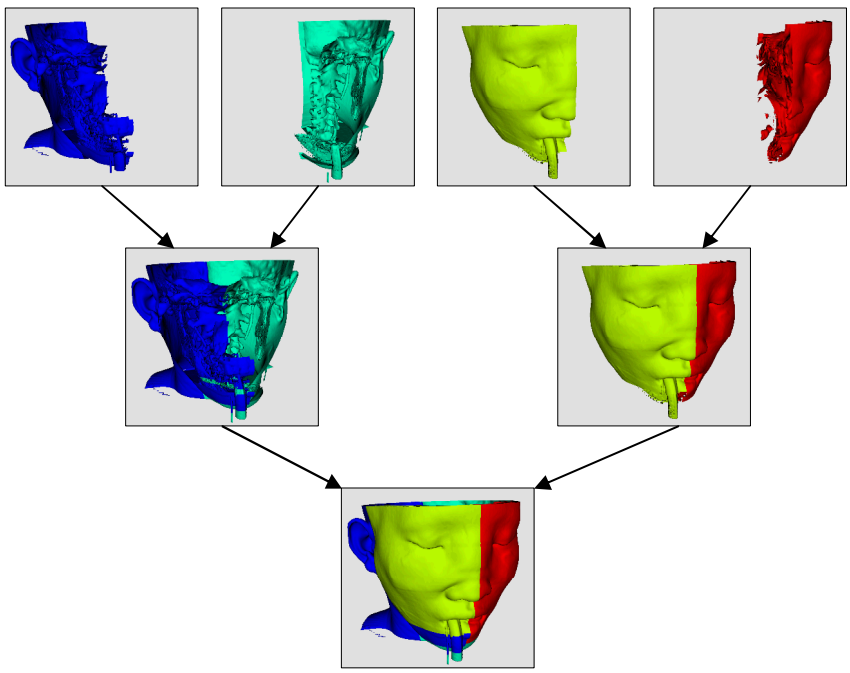


Figure 7. Tree compositing on four processes.

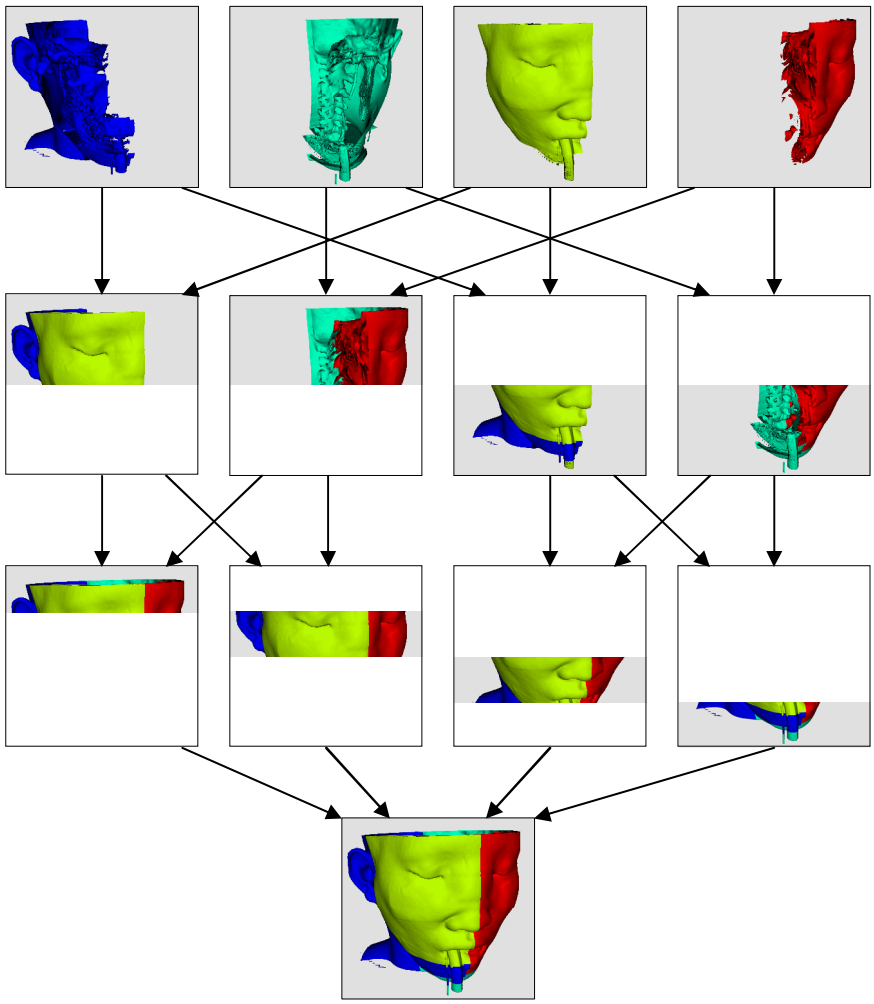


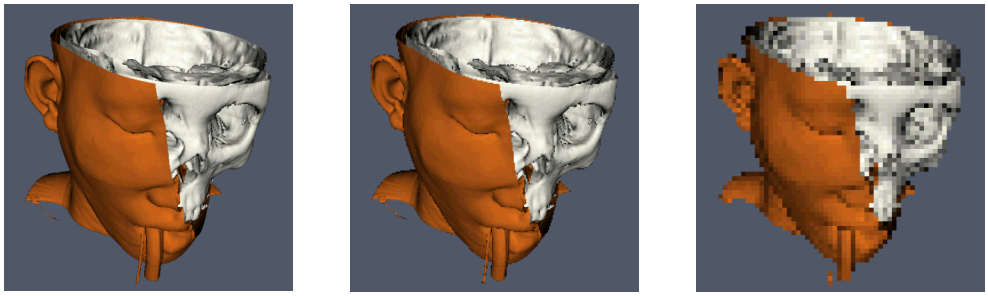
Figure 8. Binary swap on four processes.

Enable Ordered Compositing: By default, depth information is used to composite images together. As part of its normal operation, graphics hardware keeps a **depth buffer** containing the relative depth of each pixel from the camera. For compositing, this depth buffer is retrieved and used to choose which pixel is closest to the camera.

Choosing the closest pixel color is fine when the original geometry is opaque, but when the original geometry comprises transparent polygons or volumes, this compositing operation produces incorrect results. For proper compositing of translucent geometry, the colors must be blended in front to back order. When the **Enable Ordered Compositing** flag is on, IceT will composite the images in front-to-back order.

In general, a collection of polygons or polyhedra has no true front-to-back order. To ensure a proper visibility order, ParaView will redistribute the data when **Enable Ordered Compositing** is on. The distribution remains fixed during interaction, but may need to be recomputed whenever a change to one of ParaView's filters occurs. So although ordered compositing works equally well with opaque and translucent geometry, redistribution, a potentially lengthy operation, must occur whenever the geometry changes.

Subsample Rate: The time it takes to composite and deliver images is directly proportional to the size of the images. The overhead of parallel rendering can be reduced by simply reducing the size of the images. ParaView has the ability to subsample images before they are composited and inflate them after they have been composited. The **Subsample Rate** slider specifies how much images are subsampled. This is measured in pixels, and the subsampling is the same in both the horizontal and vertical directions. Thus a subsample rate of 2 will result in an image that is ¼ the size of the original image. The image is scaled to full size before it is displayed on the user interface, so the higher the subsample rate, the more obviously pixilated the image will be during interaction as demonstrated in Figure 9. When the user is not interacting with the data, no subsampling will be used. If you want subsampling to always be off, unmark the check box beside the **Subsample Rate** slider.



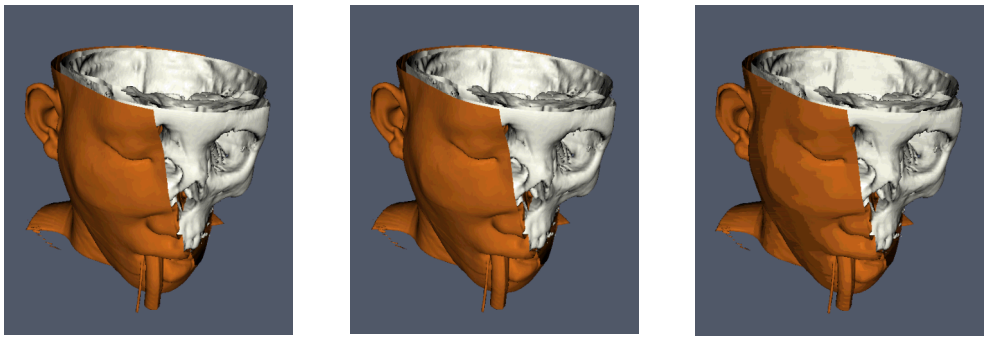
No Subsampling | Subsample Rate: 2 pixels | Subsample Rate: 8 pixels

Figure 9. The effect of subsampling on image quality.

Squirt Compression: When ParaView is run in client/server mode, ParaView uses image compression to optimize the image transfer. The compression is an encoding algorithm optimized for images called SQUIRT (developed at Sandia National Laboratories).

SQUIRT uses simple **run length encoding** for its compression. A run length image encoder will find sequences of pixels that are all the same color and encode them as a single run length (the count of pixels repeated) and the color value. ParaView represents colors as 24-bit values, but SQUIRT will optionally apply a bit mask to the colors before comparing them. Although information is lost when this mask is applied, the sizes of the run lengths are increased and the compression gets better. The bit masks used by SQUIRT are carefully chosen to match the color sensitivity of the human visual system. A 19-bit mask employed by

SQUIRT greatly improves compression with little or no noticeable image artifacts. Reducing the number of bits further can improve compression even more, but can lead to more noticeable color banding artifacts as shown in Figure 10.



24-bit mask 19-bit mask 10-bit mask

Figure 10. Artifacts caused by SQUIRT compression. As the bit mask used gets smaller, the artifacts become noticeable.

The Squirt Compression slider determines the bit mask used during interactive rendering (i.e., rendering that occurs while the user is changing the camera position or otherwise interacting with the data). During still rendering (when the user is not interacting with the data), lossless compression is always used. The check box to the left of the **Squirt Compression** slider toggles whether the SQUIRT compression algorithm is used at all.

1.6. Offscreen Rendering

When running ParaView in a parallel mode, it may be helpful for the rendering on the remote processes to be done offscreen. For example, other windows may be displayed on the node(s) where you are rendering; if these windows cover part of the rendering window, they may be captured as part of the display results from that node. A similar situation could occur if more than one process is on a given machine, and the processes share a display. Also in some cases the remote rendering nodes are not directly connected to a display.

In order for offscreen rendering to work in ParaView, you must use the `--use-offscreen-rendering` command-line option on the client, or set the `PV_OFFSCREEN` environment variable to `1`. If you have a Unix-based cluster and you do not have an xhost available, you must also compile ParaView with Mesa support (for software rendering) and with the OSMESA library.

1.7. Tiled Display

If you have a 2D grid of display devices on which you wish to show visualization results, you should run ParaView in tiled display mode. Because the ParaView application window should appear on a separate monitor from the tiled display, you must be running in either client/server mode or client/data server/render server mode to use ParaView's tiled display capabilities. The tiled display command-line argument may be included in the command line for the client, server, or data server. To put ParaView in tiled display mode, you must specify the x- and y-dimensions of the tiled display using `--tile-dimensions-x` (or `-tdx`) and `--tile-dimensions-y` (or `-tdy`), respectively. The x- and y-dimensions default to 0. If you set only one of them to a positive value on the command line, the other will be set to `1`. The example below will create a 3 x 2 tiled display.

```
./pvserver -tdx=3 -tdy=2

./pvclient
```

Tiled displays may be used similarly in client/data server/render server mode, as shown below.

```
./pvdataserver -tdx=3 -tdy=2

./pvrenderserver

./pvclient -crs
```

In tiled display mode, there must be at least as many server or render server nodes as tiles. The IceT library, which ParaView uses for its image compositing, has custom compositing algorithms that work on tile displays. Although compositing images for large tiled displays is a compute intensive process, IceT reduces the overall amount of work by employing custom compositing strategies and removing empty tiles from the computation as demonstrated in Figure 11. If the number of nodes is greater than the number of tiles, then the image compositing work will be divided amongst all the processes in the render server. In general, rendering to a tile display will perform significantly better if there are many more nodes in the cluster than tiles in the display it drives. It also greatly helps if the image data is **spatially decomposed**. Spatially decomposed data is broken into pieces that are contained in small regions of space, and are therefore rendered to smaller areas of the screen. Spatially decomposed data will allow IceT to reduce the amount of image compositing work required. Running the **D3** filter will make the geometry spatially decomposed. See section XX.X for more information on running the **D3** filter.

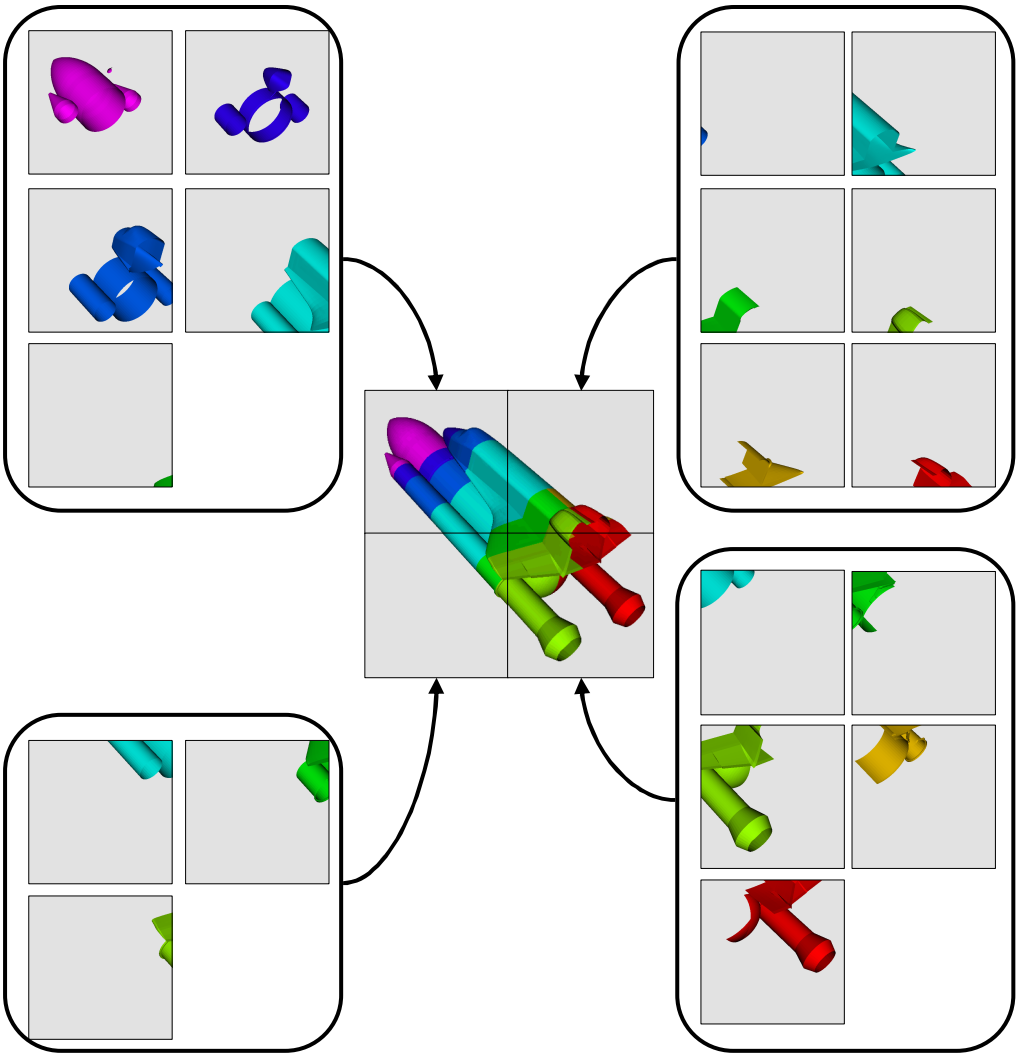


Figure 11. Compositing images for 8 processes on a 4 tile display.

Unlike other parallel rendering modes, composited images are not delivered to the client. Instead, image compositing is reserved for generating images on the tile display, and the desktop is responsible for rendering its own images. To render images locally, a decimated version of the geometry is transferred to the desktop. However, when the data is very large, even a decimated version of the geometry can overwhelm the desktop. In this case, ParaView will replace the geometry on the desktop with a bounding box.

To control behavior for downloading the geometry to the desktop client, tile display mode adds a **Client Collect** option to the **LOD Parameters** (on the **General** tab of the **3D View**

Properties property sheet) as shown in Figure 12. The slider determines when geometry is collected on the client. When the geometry is less than the threshold, a decimated model is collected on the client. Otherwise, the geometry is replaced with a bounding box on the client (but the geometry is still visible on the tile display). If the check box is unchecked, then geometry will always be collected on the client. This can be a dangerous option if you have a large cluster.

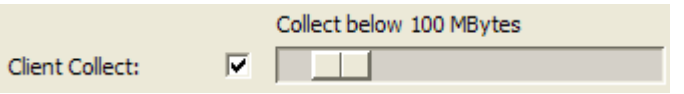


Figure 12. Options for collecting geometry on the client in tile display mode.

1.8. Acknowledgements

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.