

HP Scalable Visualization Array (HP SVA) Parallel Compositing Library

Facilitates Parallel Rendering



Introduction	2
When is Parallel Rendering Needed?	3
Features of the Parallel Compositing Library	4
Setting up the Environment	4
Context	5
System	6
Creating Frames	6
Framelets, Outputs, Results and Channels	6
Functions for Producing Frames	7
Performance	10
Sample Usage of the Library	11
Too Much Data for the Graphics Card	11
Not Enough Pixels	11
Too Much Processing & Not Enough Pixels	12
Frame Rate Too Low	13
Availability of the Library and Future Direction	14

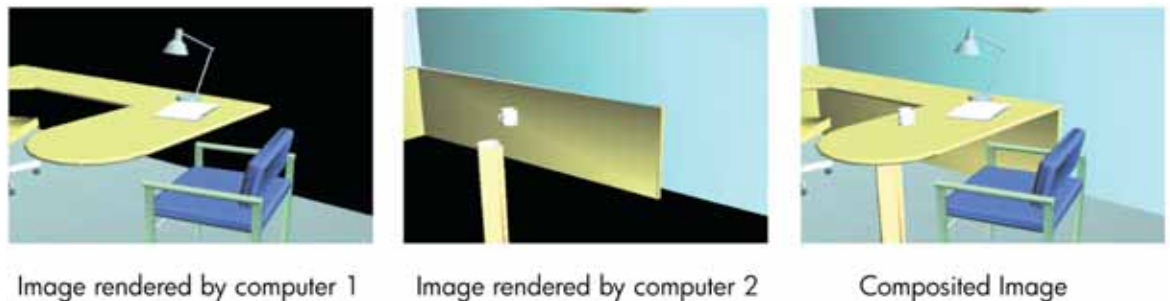
Introduction

As is true in most areas of computation, certain classes of problems overwhelm the resources of a single computer. In computer graphics, a common case occurs when a computer is unable to generate images at the needed frame rate with the necessary level of detail. When this happens, you have two options. You could wait for the next generation of computer hardware and software. Of course, that option isn't really viable since datasets are growing at an exponential rate. So the only practical solution is to investigate distributing the computation over multiple computers.

This white paper looks at distributing the generation of graphical images over a set of computers. The paper describes the Parallel Compositing Library defined and implemented by HP in collaboration with several partners. The intent of this Library is to provide a common set of functions that facilitate parallel rendering.

A literature search shows that researchers have been trying various techniques for distributing the rendering of images for several decades¹. Among the attempted techniques, image compositing is probably the most successful because it scales to higher levels of parallelism than most other techniques²³. Using image compositing, each computer in a set of computers renders a part of a final image. The rasterized image produced by each computer is then transmitted over a network to other computers. These computers combine (or composite) the partial images to produce the final image that is then displayed. Figure 1 illustrates this point.

Figure 1. Image compositing spreads image rendering over multiple computers.



Despite all the work accomplished in this area, little has been done to establish a common set of functions that an application can rely on to composite images. Applications that require parallel rendering have no equivalent to the MPI (message passing interface) used by computational applications.

¹ Ma, K-L, Painter, JS, Hansen, CD and Krogh, MF: Parallel Volume Rendering Using Binary-Swap Image Composition. IEEE GG&A Jul-94

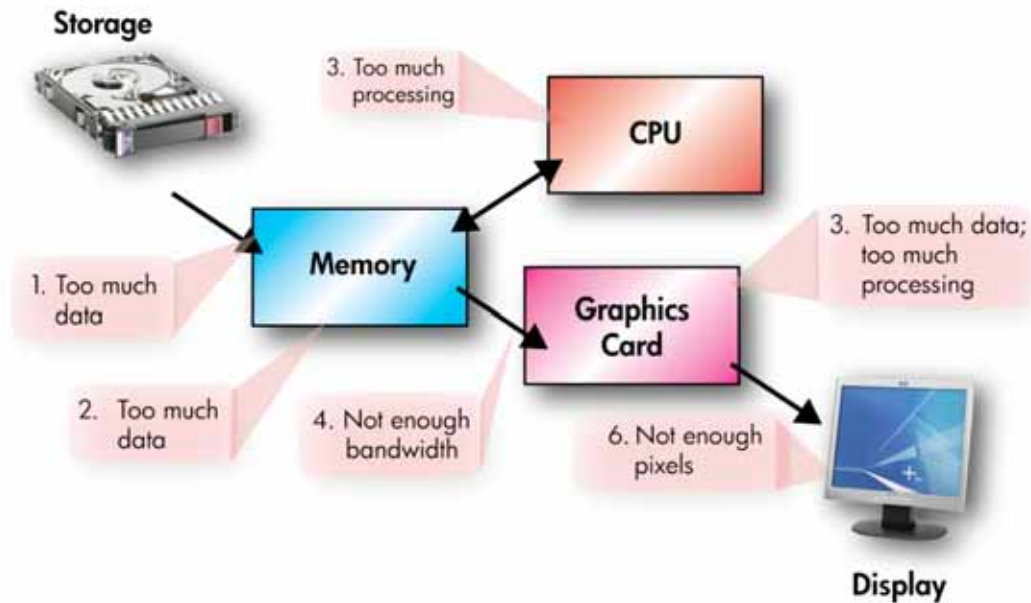
² SGI Marketing White Paper: The Power of Scalable Visualization (<http://www.sgi.com/pdfs/3620.pdf>)

³ Reinhard, E and Hansen, C: A Comparison of Parallel Compositing Techniques on Shared Memory Architectures, Dept of Computer Science, University of Utah

When is Parallel Rendering Needed?

It is worth reviewing the problems distributed rendering is intended to solve. The Parallel Compositing Library can then be evaluated against this set of problems. Figure 2 enumerates a set of limits that an application running on a single computer can reach that can cause its frame rate to drop to an unacceptable level. The first three limits are shared with computational jobs that don't involve graphics.

Figure 2. Common bottlenecks that can cause frame rates to drop to unacceptable levels.



These bottlenecks include:

1. You cannot read data from persistent storage fast enough.
2. You need to cache more data in memory than the machine supports.
3. The CPU cannot process data fast enough.

These problems can be even more serious in graphics applications because a user expects a certain frame rate.

The next three problems are unique to graphics applications:

4. There is not enough bandwidth to the graphics card.
5. You need to cache more data on the graphics card than it can hold.
6. You need more resolution than the graphics card can deliver.

With these single computer limitations in mind, let's look at the features of the Parallel Compositing Library.

Features of the Parallel Compositing Library

This section discusses both the constructs exposed by the Library to simplify parallel rendering and also the flow of an application when using these constructs.

The Library is built around a few key constructs:

- A **frame** is one of a sequence of images that an application produces.
- A **framelet** is a rectangle area of pixel contributed by a host to a frame.
- An **output** is a rectangular area of the final frame produced.
- A **host** is a thread of execution that contributes framelets to a frame and/or retrieves outputs from a frame.

Using these constructs, we can state that the key purpose of the Library is to produce a sequence of frames. An application contributes pixels to a frame using one or more framelets created by its hosts. Next, the Library composites the framelets into a final image for the frame. Once the final image is produced, the application can again use its hosts to retrieve all or part of the composited frame.

Figure 3 illustrates this point graphically. Three framelets are contributed to the frame. These framelets could be contributed by from one to three hosts. Once the frame is completed, these hosts can retrieve any rectangular part of the frame as an output. The figure shows two such outputs being retrieved.

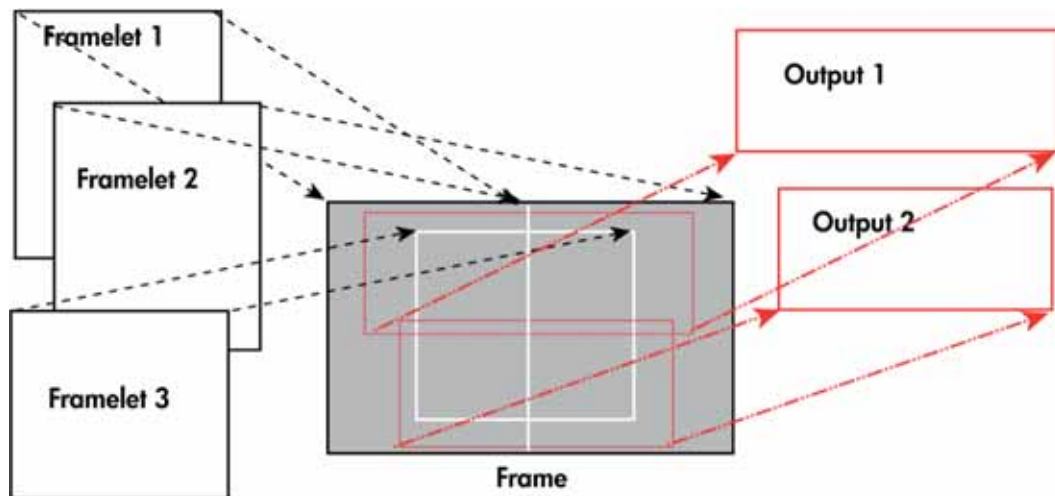


Figure 3. Hosts contributing and retrieving pixels from a frame.

Setting up the Environment

When using the Library to do parallel rendering, you start by setting up the environment. For the Parallel Compositing Library, this means establishing a set of hosts. These hosts can run on one or more computers. Having multiple hosts on a single computer would be more typical in a symmetric multi-processing machine that has multiple CPUs and potentially multiple graphics cards. The hosts can also run on different computers that are connected by a network. A combination of these is also possible as workstations (such as the HP xw8400 and xw9400) with multiple cores and multiple graphics cards increase in number.

These hosts need some means of coordinating their work. This coordination is provided by a context.

Context

Like threads, hosts largely run independently, rendezvousing when it is necessary to coordinate their work. One of the Library's primary functions is to support this coordination. The Library uses a construct known as a context to support the coordination.

A context is much like a distributed object found in object-oriented programming environments for two reasons. First, it has a set of methods that let the hosts do the following:

- Create and destroy a context
- Set and retrieve the properties of the frames the hosts produce
- Define the start of a new frame
- Contribute pixels to a frame
- Retrieve pixels from a frame once it is composited

Second, the context through its properties lets the hosts exchange information about the frames they produce. This permits one host to define properties, such as the size of the frame, and the remaining hosts to read the property so that all hosts have a common view of their work.

Context properties are of two types: properties that all hosts share and properties that are unique to the host.

- Examples of properties that all hosts share:
 - The size of the frame to be produced
 - The number of hosts contributing to the frame
 - The method used to combine the pixels
- Examples of properties that are unique to a host:
 - The name and rank of each host
 - The size and location of the framelets the host contributes
 - The size and location of the output the host wants to retrieve once the frame is produced

In order to set or retrieve host unique properties, you need to identify the host to which you are referring. The Library addresses this in two ways. First, each of the hosts has a unique identifier that consists of a `hostName` and `hostId`. These identifiers are supplied to the Library in the following format when the hosts create the context:

```
<hostName>:<hostId> (for example: mynode:0 or mynode:3)
```

The `hostName` is typically a TCP/IP name of the computer where the host runs. The integer `hostId` lets the application distinguish more than one host running on a given computer. The application is tasked with assigning each host with a unique host identifier.

When creating the context, the Library assigns to each host a unique integer known as its `hostIndex`. This parallels the idea of rank in MPI. If a context coordinates n hosts, the host indices are between $0..n-1$.

When setting or retrieving properties of a specific host, the application specifies the `hostIndex` of the host of interest. The context also supplies methods for mapping a host identifier to a `hostIndex`.

Using the methods and properties of the context, the hosts can coordinate their activity without even knowing whether a particular host is running on the same or a different machine. The transparency of the location of the host makes it possible to create an application that supports hosts on either the same or different machines with very little change in the application's code.

System

The Library is designed to be extensible. Like OpenGL, the Library will likely have new features added to it over time, such as new operators for compositing pixels or specific support for additional high-speed networks.

Given that the Library is extensible, an application often needs to know about the features of the Library supported by the current implementation. The Library provides this information via a set of system properties. Through these properties an application can determine:

- The version of the implementation
- The number of different networks supported
- The extensions implemented by this version
- The vendor of the implementation
- The path used to find Library modules
- Certain properties about the implementation, such as whether it supports overlapping the creation of frames

With the information from these properties, an application can determine which approach it will use to support a feature in the application. For example, it may be able to support a richer set of capabilities with a newer version of the Library.

Creating Frames

Once the environment is set up by creating a context, hosts can use the context to create a series of composited images. The Library refers to each composited image as a **frame**. Through the functions it provides, the Library does the following on a frame by frame basis:

- Accepts individual parts of the frame (framelets) produced by the hosts
- Composites the framelets to produce the final image
- Delivers parts of the final image (outputs) back to the application

To better understand how the Library processes frames, one needs to understand the set of frame constructs that are used by the Library to communicate with the application, and to understand the set of functions provided by the context to drive the process of producing frames. Both of these are discussed in the next section of this white paper.

Framelets, Outputs, Results and Channels

On a frame by frame basis, each host can declare the number of image contributions it makes to the frame being produced. Each of these contributions is known as a **framelet** and has the following properties:

- Height and width in pixels
- X and Y offset in the frame

A host could declare it will supply no framelets, which indicates that it will make no contributions to this frame. The host could declare it will supply one framelet that covers the entire frame. A host could also declare that it will supply two framelets: one for the top third of the frame and another for the bottom third of the frame. The number of framelets provided by a host and the position of each on the frame is entirely up to the host.

This flexibility in declaring framelets has a number of advantages. First, the frame may be much larger than an individual graphics card can draw. For example, the entire frame may be 12800x10240 pixels; but the individual framelets may be one-hundredth this size at 1280x1024 pixels. An application might also determine the bounding box around the objects drawn by a host. It can then dynamically set the size and location of the framelet it produces based on this bounding box.

On a frame by frame basis, each host can also declare the rectangular area of the final image that it wants to receive. This area is called an **output**. By setting properties in the context, each host can control if it wants any output or it can control the size and location of that output on the frame.

The two remaining constructs are related to a host's output:

- A result is a sub-rectangle of the output.
- A channel is a two-dimensional array of information associated with a result.

Using a result, the application can wait for a set of pixels to become available in the output. For example, if a host declares an output that covers the entire frame, the host could then wait for the top quarter of the output to be available, followed by the next quarter, and so on. Each quarter in this example would be a different result.

The HP `PC_FRAME_OUTPUT_HP` extension takes this a step further by allowing the host to wait for the next available result. In this case, after the Library assembles a reasonable result (perhaps a dozen scan lines), it returns a dynamically defined result to the host for processing. Once the host is done with this result, it can ask for the next available one. This permits the host to receive results without needing to predict the order in which the results become available. Using this technique a host can maximize the overlap in receiving output and processing it.

Ultimately, the host wants the pixel data in the final image. The actual pixel information available to the host varies based on what the operator used to composite the pixels and the information you request the Library to retain. This information can include:

- Color values for a pixel (RGB value)
- Depth value for a pixel
- Transparency value of a pixel (A in an RGBA value)

The Library refers to each of these as a **channel** of data associated with a result. By requesting channels of a result, the host can retrieve all or a part of the final image.

The current implementations of the Library provide support on two channels: depth and color (which can include transparency). Including the general concept of a channel allows the Library to support additional channels of data as graphic cards advance in capability.

Framelets, outputs, results, and channels are a rich set of constructs for collecting and delivering pixels data. This richness allows the Library to collect pixels from hosts and deliver pixels to hosts efficiently without exposing the internals of the Library.

Functions for Producing Frames

The previous section of the white paper described the constructs used by the Library to communicate between a host and the Library. This section illustrates the function calls a host makes to the Library causing it to composite images.

The first point to note is that the Library is defined as a set of functions that you need to call in the correct order. The Library does not use a callback mechanism in which the Library calls a set of application functions you register. As a result, it can be used within a scene graph that already has a specific workflow, or it can be used in an event-driven application where the application is driven by a series of user actions.

Figure 4 illustrates the basic sequence of calls each process will follow.

Figure 4. The basic sequence of calls each process will follow.

1. [examine system properties]⁴
2. for each host:
 - a. create a context
 - b. set/get context properties
 - c. for each frame:
 - i. [update context properties]
 - ii. begin the frame
 - iii. for each framelet
 1. render the pixels for the framelet
 2. add the framelet to frame
 - iv. end the frame
 - v. for each result
 1. retrieve the channels of the result
 - d. destroy the context

In an event-driven application, Step 2c is executed for each event that requires a redraw.

Each host executes Step 2a; however, one of the hosts, known as the **master**, calls a variant for the create context call that provides additional data to the Library. The call includes the following:

- A list of host identifiers for all hosts that create the context
- Any properties that the Library needs when the context is created, such as the network to use and the pixel format

Following step 2a, all the hosts are effectively equals, so any of them could coordinate the remaining hosts.

Step 2c(i) is the logical point to adjust context properties. The size of the frame could be adjusted based on a user having changed the size of a window.

Step 2c(ii) begins the new frame. At this point, each host needs to describe its framelets to the Library; this description includes the number of framelets, their width and height, and their location on the frame. Step 2c(ii) is also an implicit synchronization barrier; hosts that read context properties after Step 2c(ii) will block waiting for all other hosts to make it through the barrier. The synchronization barrier ensures all hosts see the same context properties from Step 2c(iii) thru 2c(v).

Once a host has described its framelets, it is free to render the pixels that it places in these framelets. It supplies these pixels to the Library in Step 2c(iii)2. There are two methods for doing this:

- `pcFrameAddFramelet` takes a pointer to a user buffer that holds the pixels.
- The `pcFrameAddGLFrameletExt` extension function reads the pixels from the current OpenGL context.

The second method is more convenient in that the Library efficiently reads the pixels it needs from the graphics card for you; but you need to use OpenGL. The first method is not tied to any graphic library; but you need to do the pixel readback.

The separation of Steps 2c(ii) and 2c(iii)2 may also seem unnecessary. It may seem cleaner to just put these steps together. However, despite the fact that the Library hides moving pixels over a network of computers to do the compositing, these steps still need to take place. The information provided in Step 2c(ii) is precisely what is needed to determine the routing of the images to do the compositing.

⁴ [] indicates a step is optional

Given that it takes some time to collect this information from the hosts, figure out the routing, and notify the hosts of the role they must play, splitting the calls makes it possible to overlap these calculations with the rendering being done by the host.

Step 2c(iv) marks the end of submitting framelets to the Library. It is not a synchronization point among the hosts; however, it does allow the Library to catch errors when a host does not submit the framelets described in Step 2c(ii). Based on analysis done by the Library, it may be able to start compositing framelets before Step 2c(iv); however, once the last host has passed this point, all framelets are available for the Library to composite.

Step 2c(v) begins the retrieval of the composited image. As previously discussed, each host can have a maximum of one output. There are two general approaches to retrieving the pixel information contained in the output. The first is to request a series of results using `pcFrameResultChannel`. If you want to retrieve the color channel of a 1280x1024 output in eight parts, your code might look like this:

```
for (i=0; i<8; i++) {
    pcFrameResultChannel (
        context=ctx, frame=frameNo,
        xoffset=0, yoffset=i*128, width=1280, height=128,
        channel=PC_COLOR_CHANNEL, channel=&channelPtr )
    processFrame(channel=channelPtr);
}
```

The call to `pcFrameResultChannel` is a blocking call that waits until the result is available. There is another function, `pcFrameResultQuery`, that is non-blocking, and thus lets you determine if a result is available before making a blocking call. However, to overlap processing of a part of the output while the Library is still collecting other parts, you need to know which part of the output will arrive first.

The second approach works well, if you want to overlap processing of the output with delivery of pixels to the host. Now your code looks like this:

```
while (pcFrameWaitOutputHP(
    context=ctx, frame=frameNo, result=&resultPtr)) {
    pcResultGetChannelHP (
        result=resultPtr,
        channel=PC_COLOR_CHANNEL, channel=&channelPtr )
    processFrame(channel=channelPtr);
}
```

The Library now returns to your application dynamically sized results as they become available. You don't need to predict what result will be delivered first.

A few more points are worth noting about the Library's functions. First, the Library delivers pixels in memory buffers to the host. There are no functions that draw these buffers to a graphics card's frame buffer for display. This task is left to the host. However, the information in the channel structure makes it possible to do this using a single call to a routine such as `glDrawPixels`.

Finally, a host may be both a source of framelets (Step 2c(iii)) and a destination for results (Step 2c(iv)). In other words, a host may both render and display. It is the choice of the application whether this is an efficient use of a host.

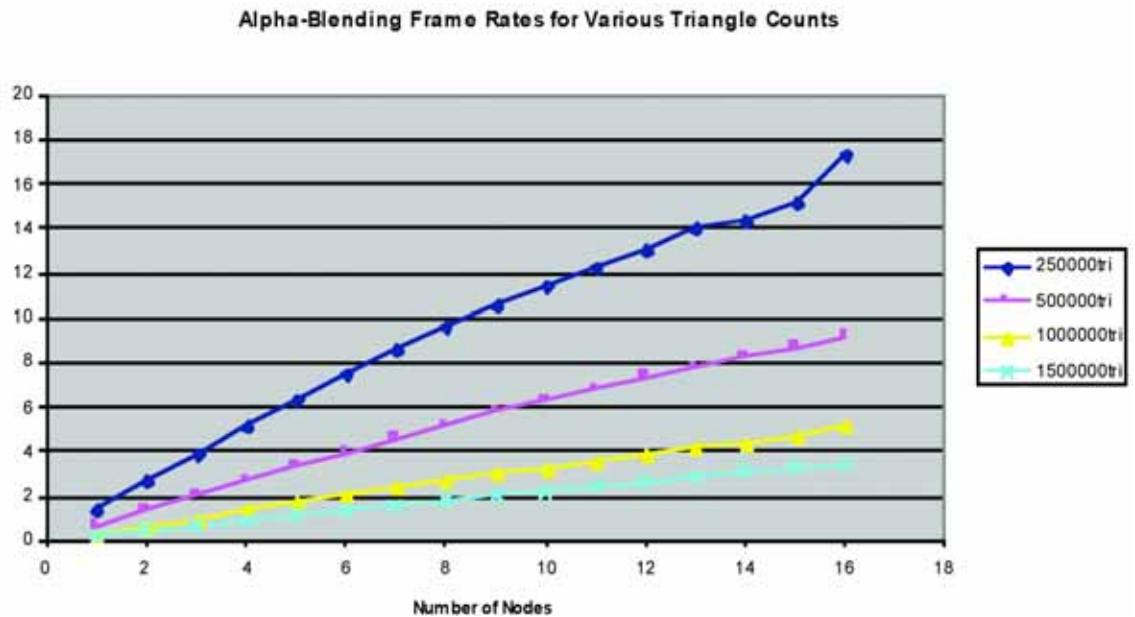
Performance

A good number of the limits discussed previously were related to a single computer not being able to produce images at an acceptable frame rate. When using the Library to share the rendering load across multiple graphics cards or CPUs, you now need to bear the cost of reading back images, compositing these intermediate images, and drawing the composite image to the final display.

A typical environment the Library must support consists of a set of workstations that share a private network. Private in this case means that the network is largely dedicated to file traffic and message passing among these workstations. The corporate web server or email system is not using half the available bandwidth. InfiniBand (IB) is an ideal fabric to connect the workstations. IB can transmit a 1024x768 image in a few milliseconds, including the depth channel between two nodes.

Figure 4 shows the results of running 1-16 HP DL140G3 1U servers. Each server contains an NVIDIA FX1500 card and is connected to the other servers using a Single Data Rate (SDR) IB switch. The test consists of drawing a large number of large triangles. The time it takes to draw the triangles (fill-rate) dominates the frame rate (fill-limited). For various loads (determined by the number of triangles drawn), the frame rate improves linearly by distributing the rendering over a range of 2-16 graphics cards.

Figure 5. Performance (in FPS) obtained by distributing rendering.



Sample Usage of the Library

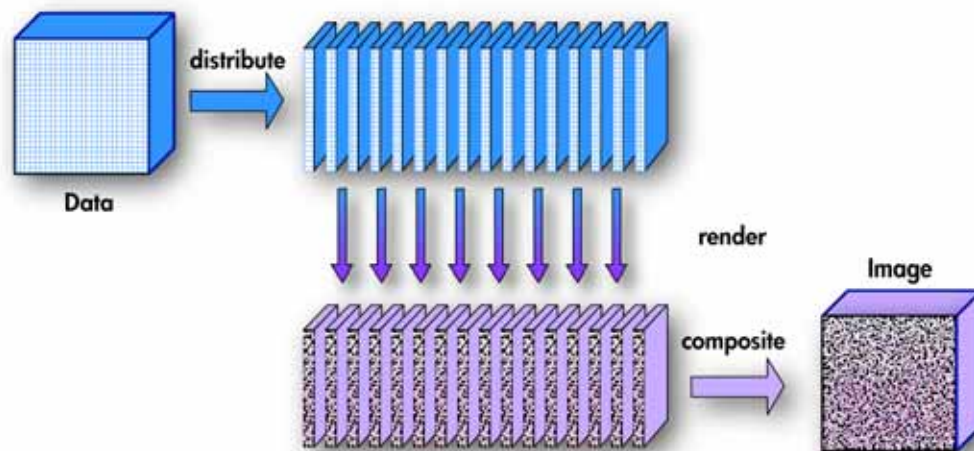
If you go back to the scenarios that exhausted the capabilities of a single computer, we can now design solutions using the Library that address many of these. We'll look briefly at the following scenarios:

1. Displaying a 2K x 2K x 2K volume medical model on a single tiled display at 20 fps⁵
2. Displaying a 3D rendering of a vehicle on 3 x 2 multi-tiled display at 15 fps
3. Displaying a very complex molecule on 2 X 2 display at 10 fps
4. Rendering a very complex scene

Too Much Data for the Graphics Card

Scenario #1 requires rendering of a 2K x 2K x 2K volume of medical data. For each point in the volume, the model holds density information that you want to color, based on these density values. To color, pan, and zoom around such a model in real-time, the model must be resident in the memory of the graphics card. You have high-end graphics cards with enough texture memory to support a 1K x 1K x .5K volume, but our model is 16 times this size.

Figure 6. An example of sixteen rendering servers and hosts.



If you use the Library, you can drive the rendering using 16 servers and use the alpha-blending operator to composite the result, as illustrated in Figure 6. The context for this solution has 16 rendering hosts and one display host. The volume rendered on each render host is 2K x 2K x 128 pixels. After a host renders its slab, it reads it back from the graphics card, including the resulting alpha value for each rendered pixel. This information is given to the Library as a framelet. The Library then composites the framelets and delivers the final image to the display host running on the 17th server, which in turn draws the pixels to a display using its graphics card.

Not Enough Pixels

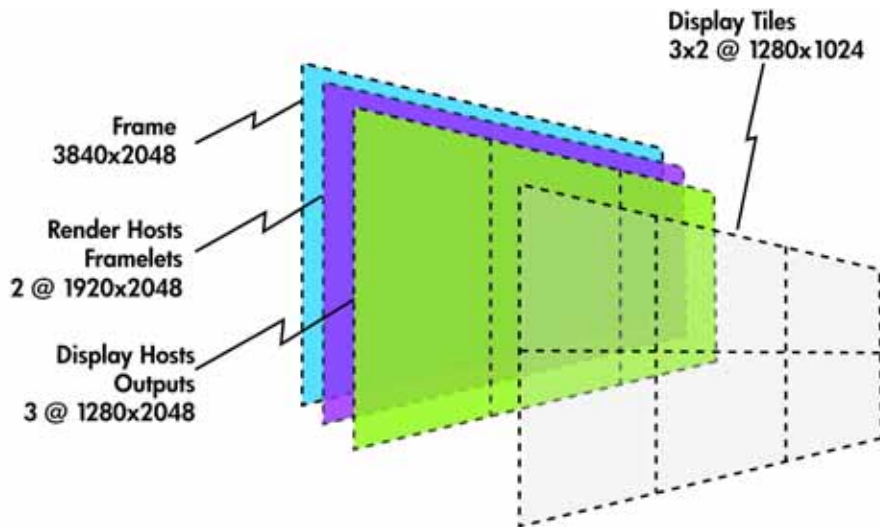
Scenario #2 is a case of not enough pixels. A graphics card has a fixed upper limit in terms of the resolution that it supports and the number of display tiles it can drive. This scenario has exceeded

⁵ Frames per second

both of these limits. For the sake of illustration, let's say you use three computers (Figure 7), each with one graphics card to drive the six tiles (two tiles per graphics card). Each tile has a resolution of 1280x1024 and thus, the size of the frame buffer⁶ on each graphics card is 1280x2048 (green plane). You will use a Library host output to drive each of the three 1280x2048 frame buffers.

Because the actual model is not that complex and a 1920x2048 window on the model can be rendered in 25ms, you set up two hosts to render the image: one for the left half and one for the right half (purple plane). Your application sets up one window and frustum for the left half of the desired image and another window and frustum for the right half. By declaring the corresponding framelets (purple) and outputs (green), the Library delivers the correct pixels from the framelets to the correct output.

Figure 7. Two hosts can be used to provide higher resolution.



Too Much Processing & Not Enough Pixels

The third scenario draws a complex image on a multi-tiled (2x2) display. The scenario can be solved in a number of different ways, depending on the exact bottleneck.

There are a number of methods of driving the display. If you don't have a workstation (such as the HP xw9400 with two NVIDIA fx4500 cards each with 2 outputs) that can drive four display tiles, you could use two servers, each with two display tiles (such as the HP ProLiant DL140g3 server with an NVIDIA FX3500 card). Alternatively, you could use four servers or workstations, each with a 1-tile display. You can vary the number of Library display hosts to fit the hardware configuration you choose.

To deal with rendering the image, you need to understand the bottleneck in a bit more detail:

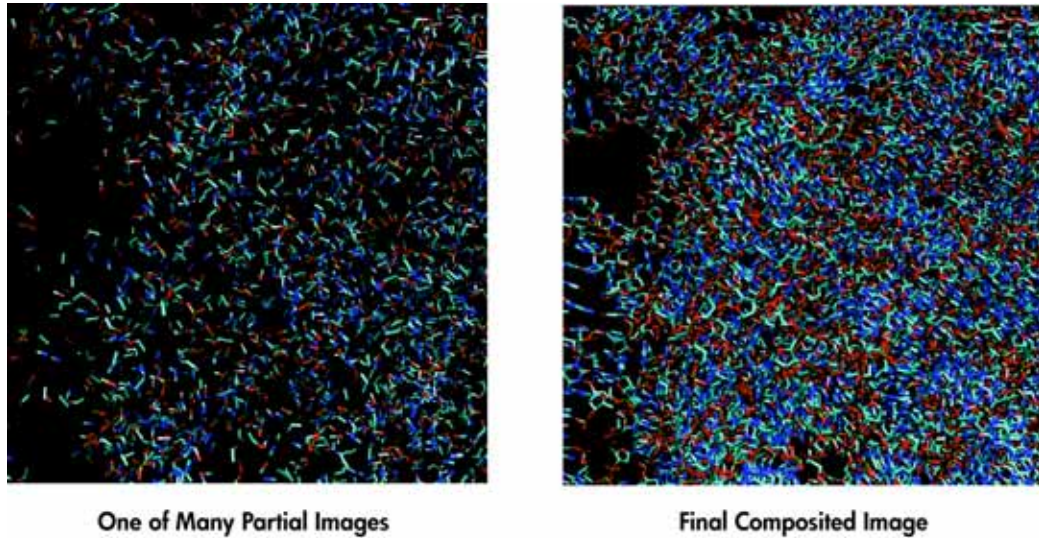
- Which takes too long, extracting or calculating what to draw? (1, 2 or 3 in Figure 2)
- Does the graphics card become bogged down trying to render the image? (4 or 5 in Figure 2)

Both problems can be fixed by distributing the work over multiple machines. The first case is a computational problem that might be solved with MPI. If MPI is the solution, the rank 0 node could draw the results of the computation using its graphics card, requiring only one rendering host. In that case the Parallel Compositing Library would not be needed.

⁶ Frame buffer is being used here to describe the buffer on the card where the image is assembled

If the problem is that too many triangles need to be drawn (vertex or fill limited), you could then ask multiple hosts to each draw a subset of the model and use depth compositing to put the partial results back together (Figure 8). For example, drawing a complex molecule or protein is often quite straightforward. You ask each of the hosts running on different machines to draw a subset of the atoms and bonds as illustrated below.

Figure 8. Multiple hosts can be used to draw a highly complex image.



Frame Rate Too Low

The final scenario is a common sub-case of the third scenario. You have a large, very complex model and you want high frame rates. The complex model is an aggregate of many smaller detailed models such as a car where the engine, wheels, seats, frame, exhaust system, and cooling system are all described separately. Since the data for drawing the subsystem is already separated and is unlikely to span the entire final image, you would like to take advantage of this and not have every framelet cover the entire frame. Reducing the size of the framelets can greatly reduce the time it takes to transfer the image to another host, where it will be composited.

In such a scenario a set of parts is assigned to each of the hosts. The host makes an initial pass over the parts determining the area of the window (called a bounding box) that encloses each subsystem. Each bounding box becomes a framelet for the host. If on average, the area covered by the bounding boxes is 30% of the total frame; transferring the frames will require only 30% as much bandwidth as transferring the complete frame. If the frame rate is limited by the compositing, a 30% reduction in the network bandwidth could dramatically boost the frame rate.

Availability of the Library and Future Direction

The Library is currently available as a part of HP's Scalable Visualization Array (SVA). HP is interested in opening this implementation to the broader community. The specification for the Library is located on HP Competency & Collaboration Network website:

http://www.hp.com/techservers/hpccn/sci_vis/frmPCR_implementation.html

HP documentation of the Library (the HP Parallel Compositing Reference Guide) can be found under the SVA portion of the HP High Performance Computing web page:

<http://www.docs.hp.com/en/highperfcomp.html>

HP believes that a compositing library that makes parallel rendering much easier is essential for advancing the adoption of graphics clusters. HP also believes that having an implementation that can be readily used by ISVs and researchers on a variety of platforms enhances the adoption of the library. To this end, HP looks forward to working with ISVs and researchers to make a version widely available.

Table of Figures

Figure 1. Image compositing spreads image rasterizing over multiple computers.	2
Figure 2. Common bottlenecks that can cause frame rates to drop to unacceptable levels.	3
Figure 3. The hosts can retrieve any rectangular part of the frame as an output.	4
Figure 4. The basic sequence of calls each process will follow.	8
Figure 5. Performance (in FPS) obtained by distributing rendering.	10
Figure 6. An example of sixteen rendering servers and hosts.	11
Figure 7. Two hosts can be used to provide higher resolution.....	12
Figure 8. Multiple hosts can be used to draw a highly complex image.	13

© 2007 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

Itanium is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

4AA1-2935ENW, June 2007

