White Paper

SGI Graphics Cluster™: The Cluster Architecture Challenges, the SGI™ Solution

Visualization Solutions Development Group

Abstract

Recent improvements in graphics performance of commodity systems have prompted the use of low-cost graphics clusters for historically high-end tasks. While not an entirely new concept, the viability of providing most of the requirements for immersive applications, such as visual simulation, at a low cost has made this approach quite appealing. This paper discusses issues and challenges specific to implementation of a graphics cluster in general, introduces the SGI Graphics Cluster product from SGI, and discusses some issues related to implementation of visual simulation run times for clustered environments.

1.0 The Cluster Architecture Challenges

Clusters of PCs or workstations can be used to solve many of the problems historically addressed by massively parallel systems. The recent increase in compute power capability and interconnect bandwidth, combined with supporting software, makes clustering of low-cost commodity components appealing.

Graphics clusters differ somewhat from compute clusters in their intent. A compute cluster will take a large problem and break it up into smaller components, distribute the problems to nodes on the cluster that solve each of their assigned tasks, then synchronize all the information at the back end. In a graphics cluster, the intent is to give multiple views of the same visual data set (sometimes referred to as viewing channels). Each node on the graphics cluster must have access to the entire data set, then independently determine how much of the data set is visible given its assigned viewing frustum, and render just that part.

An important difference between the functionality of a compute cluster and a graphics cluster is the real-time aspect. A graphics cluster performs interactive tasks in real time, while compute cluster tasks are usually offline and batch-oriented. The interactive or real-time rendering in typical graphics applications requires that a graphics cluster complete its entire task in a few milliseconds, making latency a challenging issue. A related obstacle is that the graphics cluster must also provide a coherent, seamless, and contiguous display from its isolated, distributed visual components.

Based on these issues, the challenge becomes one of synchronization across the cluster within tight latency limitations. Specifically, there are three levels of synchronization that need to take place:

- Video signal synchronization
- Dynamic data synchronization
- Frame completion synchronization

1.1 Video Signal Synchronization

All computer graphics subsystems contain a signal generator that drives the output to a display system (such as a monitor or a projector). This signal not only provides the visible image, but also directly controls the synchronization of the display device. This signal and its synchronization are not something that can be controlled via software on commodity graphics cards; they are a pure hardware function. A detailed explanation of this functionality is beyond the scope of this paper, but there are several key components that are pertinent to this discussion: refresh rate, blanking time, and pixel rate.

Refresh rate is the number of times a display or an image is updated (drawn) per second and is measured in Hertz (cycles per second). For example, in a raster display, like a CRT display, the image is typically drawn in lines from top left to bottom right, and this happens many times a second (typically 30, 60, or 72 Hz). This is fast enough that your brain believes it is seeing a solid image (although you may detect some flickering).

Blanking time is the time between when one frame is drawn and the next one begins.

Pixel rate refers to the rate at which each pixel on the screen is updated. This is a much faster rate than the refresh rate because for each image there are many pixels (frequently more than one million).

For those unfamiliar with real-time 3D graphics, the system draws one image in a hidden place (the back buffer or draw buffer) while displaying the previous image (which is said to be in the front or display buffer). These buffers are used to store pixels so that the system is not attempting to draw pixels as they are being displayed. In order to avoid this, pixels are buffered for up to one frame time. This technique is known as double-buffering.

SWAPBUFFERS is the term used for the moment when these two buffers are exchanged. Modern commodity graphics cards ensure that this occurs only during the blanking period on the display device because this is the only time that data is not being read out of the display (front) buffer.

The following is required to provide a seamless image:

- Each channel must render the same data set
- The pixel rates must be identical (otherwise the images will drift relative to each other)
- The displays must start new images at the same time (otherwise you will see different frames on different displays)

- The graphics systems must swap their buffers during the same blanking period (otherwise you will see one new frame before the others)

The nature of a graphics cluster is that each node in the cluster drives its own graphics output. While each graphics output is designed to run at the same refresh rate, the reality is that they will all be slightly different, so that they will have slightly different refresh and pixel rates, which causes drift of the images relative to each other. Naturally, without forced synchronization, they would also all start at different times, and even with the same pixel rates would be out of sync.

All of these actions are required before the last step—synchronizing the SWAPBUFFERS commands. If this is achieved, then there will be a truly seamless image across multiple viewing channels.

The problem of creating a seamless image is well-known and has solutions in both high-end graphics and TV broadcast markets. In these markets, there are two ways to address the synchronization issue.

Genlock is the most precise way of ensuring synchronization. Here the graphics system ensures pixel-level synchronization by using a PLL to lock onto the line rate to derive the pixel rate (or pixel clock). The lock is fine enough to allow phase adjustments for each pixel.

Framelock is a less-precise method and synchronizes once per frame at the end of the blanking period.

Genlock is technically more demanding than Framelock because it has a shorter period of time in which to gain synchronization. For reasons beyond the scope of this paper, it is generally accepted as the appropriate method of synchronization for high-end or specialist applications such as edge-blended displays, target projection, and video editing and capture. Framelock is often good enough for general low-end semi-immersive requirements.

1.2 Dynamic Data Synchronization

Real-time multichannel graphics applications have two kinds of dynamic data that will often change every frame: control information, which helps each node or channel determine what to draw (e.g., the direction of view for that node); and changing/dynamic data set information (e.g., database updates due to models that have moved or source data that has changed, such as textures for special effects). This information is often the result of control information (but may not be; for example, a computer-controlled object on a random path).

Dynamic data is generated by a stimulus in a real-time application. Examples of stimuli are:
- Keyboard and mouse events
- Joystick or flight-control devices
- Head trackers
- Gloves or body suits
- Playback of source data
- Frame scheduler

There is a response to these stimuli that, in turn, generates the dynamic data. This response can take the form of complex software, as in the case of a flight model or the creation of an iso-surface in engineering analysis, or be as simple as a transformation of coordinates, as in the case of a head tracker.

The response can itself be considered a stimulus for further generation of dynamic data—for example, a keyboard event may generate a new position for the viewer (first response) that in turn may cause the loading of new data that is now visible (second response).

1.2.1 Timing of the Dynamic Data Update

Many people assume that in an OpenGL graphics application the SWAPBUFFERS command is a blocking one, which is to say that the application is forced to wait until the graphics card (and hence the system) says it is ready to start drawing the next frame. In fact, there is a small buffer (FIFO) of graphics instructions between the application and the graphics system. This maximizes the throughput of the overall system, and the application will be blocked once the buffer is full. This will typically happen at some point soon after the issue of the SWAPBUFFERS command. This moment is unpredictable. To remove the unpredictability, many programmers put a block just after the SWAPBUFFERS command. This is a quiet time within the system and therefore an ideal time to synchronize dynamic data between channels.

1.2.2 Approaches to Data Updates

There are multiple approaches to synchronizing dynamic data:
- Distribute stimuli
- Calculate resulting data centrally and distribute
- Calculate end graphics data centrally and distribute

The programmer in a graphics cluster environment must decide whether it is more appropriate to transmit the stimulus, the resultant data, or the final graphics data.

Distribute Stimuli

Sometimes a simple stimulus such as a new position may generate the need to load or generate large new data sets. It may be more effective to send out stimuli

with time stamps and create the resulting data on each channel. This approach trusts that nodes will respond evenly, which may not be the case. It is simple to apply to existing software, altering only the environment in which each node receives its stimulus. However, the effects of losing data in transmission are severe, potentially leaving the channel permanently out of synchronization with the others. This is especially problematic if effects are cumulative.

Distribute the Calculated Data Results
A flight model involves a lot of calculation and results in little data, so transmitting the results is most appropriate. Here one node is responsible for input stimulus, response, and subsequent dynamic data generation. It then synchronizes its dynamic data with the other nodes that, in turn, cull and render based on the new information.

This approach is easier to manage because the variances in data availability on the channels are purely a function of the interconnection network. It also tends to be more tolerant of lost information.

Distribute End Graphics Data
This is the approach with the highest overhead; it puts all the work on the master channel, which takes input stimuli, generates responses, and performs culling and partitioning of data to be rendered. The channels then simply render what is handed to them.

This approach would make sense only in an interconnect topology that supports bandwidth scaling. It is, however, in many senses the most tolerant of lost data.

1.3 Frame Completion Synchronization

There remains one further task to complete a fully coherent image across all nodes. Since each channel is drawing a different view, each will have a different amount of work to do and will take a different amount of time to finish. It is both possible and common that one channel will finish well ahead of the others. If it were to call SWAPBUFFERS now, it is possible that it would actually update its image (in a blanking period) before the others all finish. It would then display an inconsistent image, which is frequently undesirable.

The solution is to make all of the nodes wait for each other. This is achieved with a barrier that causes each node to block until all the others are ready. Since all visual frames end in SWAPBUFFERS, this is the natural place for a barrier.

This waiting for other nodes to finish rendering is sometimes referred to as a swap barrier synchronization. If there is a further hardware mechanism for coherent waiting before SWAPBUFFERS within the graphics card, then this augmented approach is called swap-ready.

2.0 The SGI Solution

SGI has been implementing graphics clusters for almost a decade, with early demos on Crimson™ machines in 1992, Silicon Graphics® O2® workstations in 1996, and Silicon Graphics® 320 workstations in 1998.

SGI™ Origin™ family and SGI™ Onyx® family systems are based on a very highly scalable, low-latency interconnect and a specialized concept (SGI™ NUMAflex™) that provides for CPU-level cache coherency (application-transparent data synchronization).

SGI drew on this extensive experience in parallelism, concurrency, graphics clusters, and data synchronization when architecting SGI Graphics Cluster.

2.1 Master–Channel Architecture

The primary goal of SGI Graphics Cluster is to enable a group of commodity-based machines to behave much like a single system for the user and programmer. The master–channel architecture facilitates this.

The master provides the traditional workstation environment to the user. One keyboard and one mouse are attached to the master, and all external input devices and external communication are on the master. Further, the user's primary desktop and GUI software is run on the master.

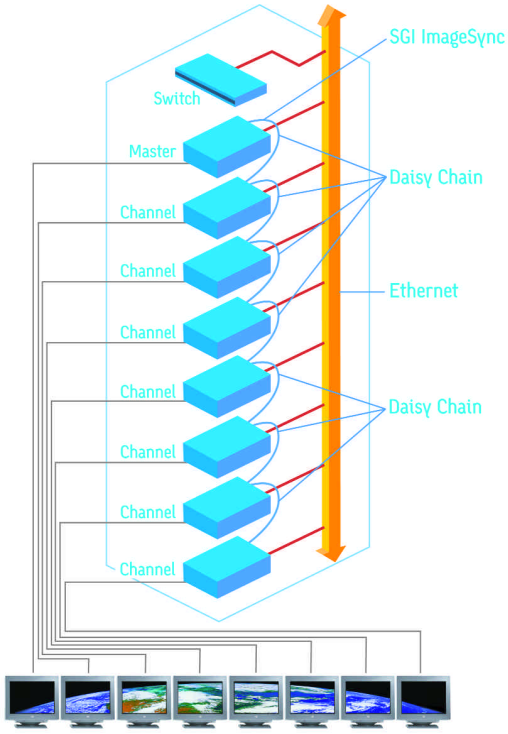
Each channel performs like a dedicated graphics pipe, not like a full system. The channel is more capable than a standard pipe in its capability to perform culling, drawing, and other channel-specific tasks.

The master and channels communicate through an internal network, which, like SGI NUMAlink™ in an SGI Onyx family system, is internal to the system. As a result, the master has two network interfaces—one to communicate with the channels and the other for the outside world. To the rest of the network and the outside world, the system appears as a single system, not a cluster.

2.2 Video Synchronization

In the past, there have been two primary methods to try to solve the video synchronization problem.

The first uses a type of software approach. Despite claims to the contrary, software synchronization (usually over a network of some kind) can offer little more than a swap barrier. Many of these attempts also miss the relevant barriers within SWAPBUFFERS to avoid latency variability issues inherent in the OpenGL FIFO approach.



The second is a hardware solution, which is the only workable approach if true video synchronization is to be achieved. Until now, it has been necessary for applications requiring video synchronization to use custom, proprietary graphics subsystems that implement this feature. Unfortunately, these subsystems are typically specialized and expensive and frequently lack the performance and capability of commodity graphics cards because of the length of time required to update them.

SGI ImageSync™ technology solves this problem by providing additional value-add to existing commodity graphics cards. SGI Graphics Cluster can then utilize whatever is the current performance leader in the commodity graphics space and also satisfy video synchronization requirements. This offers performance leadership with COTS affordability and flexibility. SGI ImageSync achieves this by providing an external clock to all of the graphics cards in the cluster using technology developed through our tight partnership with commodity-card providers. Through SGI's patented design, these cards guarantee that all graphics subsystems are synchronized to exactly the same pixel rate and that the output pixels are in complete lock-step.

But this is not sufficient for synchronization. The pixels are now in lock-step, but they may be at some unknown offset to each other. It is therefore necessary to also provide a frame-level synchronization to ensure that the pixels are in alignment. SGI ImageSync technology does this with a synchronization scheme that, in the worst-case scenario, results in a 3 microsecond latency. This translates to less than one scan line, and in most cases the offset lies within just a few pixels. The result is a true video lock better than Framelock, with a precision approaching that of Genlock.

2.3 Swap-Ready Barrier

Many current graphics cluster implementations use a software swap-ready barrier that synchronizes over a network. At the point where the application would call for a buffer swap, all nodes must acknowledge readiness over the network, then, once acknowledged, call SWAPBUFFERS. This approach suffers from the latency of network acknowledgment and the nondeterministic period of time between when the application requests a SWAPBUFFERS and the SWAPBUFFERS actually takes place.

To compensate, stable applications must place an extra margin between the time that acknowledgment takes place and the next vertical retrace boundary. Unfortunately, this uses up precious frame time.

SGI ImageSync technology provides a hardware implementation of a swap-ready barrier and does so through the SGI standard GLX_SwapBarrier extension. When each channel issues a SWAPBUFFERS, the SGI ImageSync technology holds that channel until all channels have issued SWAPBUFFERS. At this point, acknowledgment has taken place and all channels' SWAPBUFFERS can continue, next waiting for vertical retrace. This implementation provides minimal latency for the swap barrier.

The software interface for this functionality is provided in the already-defined SGIX_swap_barrier extension.

2.4 Data Synchronization Software

Today in the visual simulation and training market, there are a number of applications on several different PC-IGs, and each application has its own approach as to what level of reliability the dynamic data is synchronized and its own scheme for passing the data between the nodes.

SGI Graphics Cluster provides a standard internal 100Base-T network (upgradable to Gigabit Ethernet) that allows any of these schemes to simply work should users wish to adopt such a scheme.

Should users wish to implement their own scheme, SGI provides—free of charge—SGI DataSync™, a simple, powerful API for data sharing and synchronization within a graphics cluster. Programming with DataSync, users create sessions that each of the masters and channels will join and pools of data that are shared and synchronized across the channels. The data pools support data that is primarily dynamic in nature—

such as eye-point updates, input devices, special effects, and additional entities. The pools also support modal messaging, such as “Delete node Y” or “Load file X.” Multiple channels can join multiple sessions and can produce, consume, or monitor the data being used and generated. This offers the user and programmer ultimate flexibility and support. In addition to SGI DataSync, SGI also bundles sample code for its most popular APIs (such as OpenGL Performer™), showing how to use the API in conjunction with SGI DataSync to maximize functionality across a graphics cluster.

3.0 Developing Software for SGI Graphics Cluster

The primary difference between developing software for an SSI (single system image) machine, such as an SGI Onyx family system, and a typical cluster lies in the treatment of memory.

On an SSI, the programming model is seemingly simple: allocate memory visible to all processes within an application and run. However, there are still challenges that are traditionally associated with shared memory between multiple threads or processes, requiring the use of methods to ensure data integrity. These are not trivial and are often solved nearly transparently for the programmer by packages such as OpenGL Performer. The application in question can be extended to support a clustered environment.

While the general case can be somewhat complex, the requirements for real-time graphics applications are simplified greatly with the following assumptions:

- Memory synchronization on a 16.667 msec boundary is all that is required. Data generated in one frame is used for rendering in the following frame. This is true in the tightly coupled multiprocessor/multidisplay SSI system. It is rare that dynamic data has a latency of less than 50 msec, which is easily achievable with clusters.

- Dynamic data needed to change a scene on a frame-by-frame basis is relatively small. The eye-point and all dynamic coordinate systems can be controlled at most with a 16-component floating-point array. Experience with visual simulation applications, for example, has shown that fewer than 1,500 bytes are required to control a rich, dynamic scene. The transmission time for 1,500 bytes over a 100Base-T Ethernet is submillisecond. In broadcast mode, data need be transmitted only once per frame to reach all channels.

The use of SGI DataSync ensures that the hardware network layer remains transparent to the application, since SGI DataSync will run over standard low-cost 100Base-T Ethernet, higher performance Gigabit Ethernet, or Myrinet.

3.1 The Transparent Approach

It is possible, in a limited way, to take existing applications written for SSI architectures and transparently convert them to run in a clustered environment. This approach requires the use of overloaded APIs or plug-ins that implement the dynamic data synchronization without changing anything in the existing application.

SGI has created an example of this in an OpenGL Performer loader bundled with SGI Graphics Cluster. This example adds a node to the OpenGL Performer scene graph that automatically synchronizes scene graph data. This data is loaded at run time and specifies details of which nodes etc. are to be synchronized.

SGI is modifying its most popular APIs to run in this fashion, helping to ease the burden of transitioning from an SSI environment to a cluster environment.

3.2 The Light-Touch Approach

Many users will want a light-touch approach to programming in a cluster environment that will need some coding changes and recompilation as well as additional linking of the application to SGI DataSync routines needed to implement the network synchronization. This approach requires that the application be designed originally with all dynamic data in close proximity or that it be small enough to be copied when modified. The dynamic data is then allocated in a special memory pool that can be transmitted or received at frame synchronization time, depending on whether the application runs in master mode or slave mode.

3.3 The Rigorous Approach

Using the transparent approach or the light-touch approach has many drawbacks. The programmer will soon realize that there are a lot of potential inefficiencies with the other approaches and a lot of dead code in the application because it runs in master mode or slave mode.

It is recommended that new applications be designed from the outset to run on a cluster. This requires moving from a monolithic programming model to a modular model. Some modules will be designed to run on the master and others on the slave. The run-time environment can then be configured to load those modules that are relevant to the node’s role.

It is important to note that applications designed to run on a cluster will run on SSI systems with minimal configuration changes. The opposite is not true.

One approach is to develop modules with common interfaces, so that they can sit on the master or channels, but to give them different behaviors. For example, on the master the module might send data at regular intervals (perhaps at frame boundaries, driven by the frame scheduler). The channel's module might be event-driven, simply updating data when a data-received event occurs. The master might also respond to feedback events from the channel—for example, feedback on the graphics load occurring on the channel. To complete the circle, the frame scheduler running on the master may be driven by a vertical refresh interrupt originating from the SGI ImageSync hardware or even the swap-ready signal.

4.0 Summary

Three challenges must be overcome to implement a high-quality graphics cluster:

- Video synchronization (pixel rate, pixel alignment)
- Dynamic data synchronization
- Frame completion synchronization

SGI solves these in SGI Graphics Cluster with SGI ImageSync technology, which provides precision video synchronization as well as a swap-ready barrier. SGI also provides SGI DataSync to facilitate the implementation of run times for a clustered environment.

5.0 References

[1] Beowulf. *www.beowulf.org*.

[2] MPI. *www-unix.mcs.anl.gov/mpi/*.

[3] Bob Kuehne. An Architectural Comparison of Single-System-Image and Clusters of Workstations for Interactive Graphics Applications. Technical Report, SGI, 2001.

[4] Don Burns. Multichannel Synchronization with Loosely Coupled, Low-Cost IGs. Technical Report, SGI, 2000.

[5] DGD. *www.isl.uiuc.edu/ClusteredVR/paper/DGDoverview.htm*.

6.0 Definition of Terms

NUMA — Non-uniform memory access

SGI NUMAlink — Patented interconnect for the SGI Onyx family and SGI Origin family

COTS — Commercial off-the-shelf

CRT — Cathode ray tube

FIFO — First in first out

SSI — Single system image

API — Application programmer's interface

PLL — Phase loop lock

PC-IG — PC-based image generator, used in visualization to create a semi-immersive environment

Corporate Office
1600 Amphitheatre Pkwy.
Mountain View, CA 94043
(650) 960-1980
www.sgi.com

North America 1(800) 800-7441
Latin America (52) 5267-1387
Europe (44) 118.925.75.00
Japan (81) 3.5488.1811
Asia Pacific (65) 771.0290

© 2001 Silicon Graphics, Inc. All rights reserved. Specifications subject to change without notice. Silicon Graphics, O2, Onyx, and OpenGL are registered trademarks, and SGI, Crimson, SGI Graphics Cluster, NUMAlink, Origin, NUMAflex, OpenGL Performer, SGI ImageSync, SGI DataSync, and the SGI logo are trademarks, of Silicon Graphics, Inc. All other trademarks mentioned herein are the property of their respective owners.

3088 (7/01) J12814