Parallel Graphics and Interactivity with the Scaleable Graphics Engine

Kenneth A. Perrine, Donald R. Jones
William R. Wiley Environmental Molecular Sciences Laboratory
Pacific Northwest National Laboratory
PO Box 999
MS K1 - 96
Richland, WA 99352
kenneth.perrine@pnl.gov,
dr.jones@pnl.gov

Abstract

A parallel rendering environment is being developed to utilize the IBM Scaleable Graphics Engine (SGE), a hardware frame buffer for parallel computers. Goals of this software development effort include finding efficient ways of producing and displaying graphics generated on IBM SP nodes and of assisting programmers in adapting or creating scientific simulation applications to use the SGE. Four software development phases discussed utilize the SGE: tunneling, SMP rendering, development of an OpenGL API implementation which utilizes the SGE in parallel environments, and additions to the SGE-enabled OpenGL implementation that uses threads. The performance observed in software tests show that programmers would be able to utilize the SGE to output interactive graphics in a parallel environment.

Keywords: parallel hardware frame buffer OpenGL multithreaded SMP visualization framework

1. Introduction

Researchers at Pacific Northwest National Laboratory (PNNL), William R. Wiley Environmental Molecular Sciences Laboratory (EMSL) have a growing need to visualize distributed data generated by the 512-node IBM SP system at the Molecular Sciences Computing Facility (MSCF). Currently, researchers collect information generated by simulations performed on the SP and visualize the data on a graphics workstation. Transferring the data from the SP to a workstation is time-consuming and does not offer interactive capabilities, such as steering an ongoing simulation computation in real-time.

Alternatively, graphics data generated across many SP nodes can be consolidated onto one node and viewed across the network on an X terminal. This approach can be slow, both from the time required to consolidate the data and to transfer the images to the X terminal.

To address these problems, PNNL recently collaborated with the IBM Watson Research Center to produce the IBM Scaleable Graphics Engine (SGE) [6]. The SGE is a high-performance frame buffer for parallel computers. By generating graphics on SP nodes and eliminating the reliance on workstations, graphics can be more rapidly rendered. There is now an effort to write software to support the SGE and utilize its capabilities.

This paper describes PNNL's software development approach for utilizing the SGE in creating a parallel rendering environment. The first phase is an interactive simulation of a vibrating spring mesh to build graphical output from pre-rendered primitives. The second phase involves the use of threads on SMP nodes to decode a number of MPEG video streams and output the streams to the SGE in parallel. The third phase is a set of extensions for the Mesa and GLUT libraries to support OpenGL rendering in the parallel environment and accelerated rendering using automatically-assigned viewport clipping planes on all nodes. The fourth phase involves further modifying Mesa and GLUT to allow multiple graphics frames to be rendered simultaneously by the use of threads. Future work includes incorporating imaging compositing and geometry sorting to the rendering environment. In the descriptions of the programming projects, SP and SGE performance is addressed along with considerations for optimally utilizing the resources available, including SMP and the high-speed switch links.

First, a description of the SGE architecture [5] is given to provide a basic understanding of its capabilities.

SC2001 November 2001, Denver 1-58113-293-X/01/0011 $5.00.

2. SGE Architecture

The SGE is a hardware component that connects to the SP switch. It can also connect to a Linux cluster through Gigabit Ethernet interfaces. It receives pixel fragments directly from any number of nodes. Multiple high-speed links to the switch allow it to receive pixel data in parallel from multiple nodes. The SGE can support up to 16 links to the switch.

Disjoint pixel fragments are joined within the SGE frame buffer and displayed as a high-resolution, contiguous image. As an example, Figure 1 shows quarters of an image rendered on 4 nodes being displayed on the SGE as a contiguous image. The joining of pixel fragments from multiple switch links is done within the SGE through a router backplane that pipes incoming graphics data from the switch links to the multi-banked back-buffer memory. The router maintains multiple paths of pixel data from the input link interfaces to the memory banks, enabling the streams of incoming data to be stored in memory in parallel, rather than serially. The frame buffer can support up to 16 million pixels and can output to multiple, arbitrarily tiled display units through the display driver hardware. Both 24-bit and 16-bit color is supported, and stereo output is provided for use with stereo LCD glasses.

Concurrent read and write bandwidth of each memory bank is 45 megapixels per second. Each of the 8 memory cards contains 2 banks. The SGE is capable of updating 8 display driver cards with a peak performance of 720 megapixels per second. Display driver cards are available for analog outputs and digital outputs. With digital video output hardware, the SGE can drive the new IBM 204-DPI, 3840x2400-pixel "Bertha" LCD display.

The SGE software includes an X server (X11R6.4) and a subroutine library allowing graphics applications to communicate directly with the SGE. The use of this library to transfer graphics data to the SGE is called *tunneling*. X11 extensions coordinate with and support the high-bandwidth tunneling libraries.

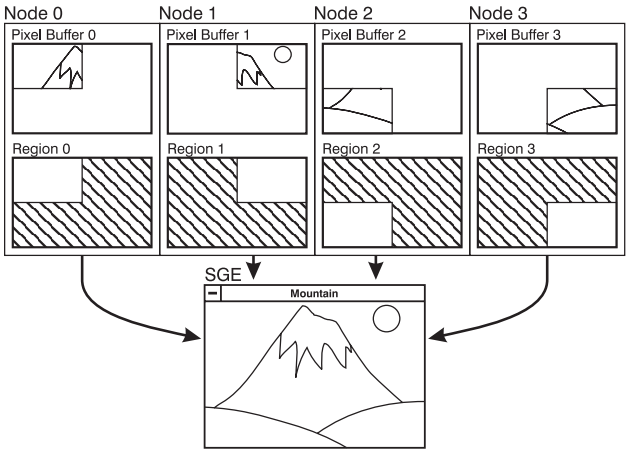


Figure 1. Separate buffers with their respective regions on compute nodes and the joined SGE image

Figure 2 shows the structure of a basic SGE parallel application. The application runs in parallel on multiple nodes and maintains a buffer for outgoing graphic data on each node. The application performs X11 window creation and event handling setup on one node and utilizes the SGE tunneling library to distribute events to the other nodes.

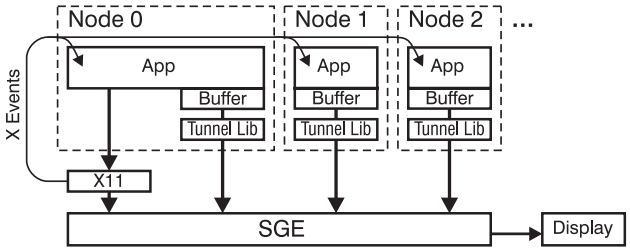


Figure 2. SGE application structure and data flow

X11 region structures are used on each node to define which pixel areas of the SGE output window are to be updated by each node. In addition to supporting evenly-divided regions as pictured in Figure 1, the SGE also supports arbitrarily-placed regions of any complexity, including multiple, disjoint regions from each node and nonrectangular regions. These regions may also be overlapped and sequenced to produce animation.

The SGE differs from PixelFlow [12][4], a graphics hardware device that is designed to work in conjunction with a parallel computer. Although the SGE contains parallel frame buffer memory and has the ability to perform image masking for arbitrarily-shaped regions, it does not contain hardware for geometry transformation, shading, anti-aliasing, depth-buffering, or the use of alpha pixels to perform compositing. When using the SGE for graphics output, all of these steps must be performed on the parallel computer's nodes. The SGE is also unlike the Lightning-2 [16], a passive external compositing device that interacts with digital video interfaces to combine multiple video outputs (in the form of DVI video signals) into a single video output. The SGE's hardware is targeted for framebuffer-related functionality at the pixel level, and provides flexibility for utilization with a variety of applications and rendering algorithms. For example, parallel software rendering algorithms for volume rendering, polygonal rendering, and large image manipulation can all utilize the SGE for parallel output. Rendering algorithms which produce blocks of rendered image data, such as some ray-tracing or bucketed polygonal rendering [2] algorithms can output graphics directly to the SGE whereas other algorithms may require a software compositing step in order to create ready-to-

display graphics data. The SGE also can be used in conjunction with hardware graphics accelerators installed in Linux clusters. The use of off-the-shelf graphics cards in Linux clusters is an increasing area of research—an example being Sandia National Laboratory's Ric cluster [17]. It is possible for such clusters to output regions of rendered images to the SGE using graphics accelerator framebuffer readbacks.

The SGE at PNNL is currently connected by 4 switch links to an IBM SP with 24 Winterhawk-2 nodes. Each node is equipped with 4 375MHz Power3 processors. The SGE is connected to the Panoram Technologies PV290 monitor, which is three 1280x1024-pixel flat-panel displays housed in a single unit. The three displays output the SGE frame buffer, which is configured for 3840x1024 pixels. One of the displays can be directed to a CRT video projector where stereo LCD glasses may be used to view stereo graphics.

The SGE will soon be configured with another SP that has 4 Nighthawk-2 nodes with 16 processors on each node. Although the SP "Colony" switch will be used in the SP, Gigabit Ethernet links will connect each node to the SGE.

3. Application Development

The following sections describe software development projects for the SGE. These projects all contribute to the final objective of having a set of tools and libraries—a parallel rendering environment—that developers and scientists can use in order to utilize the SGE.

So far, the SGE code has been designed at PNNL to perform software-only 2D and 3D rendering. Before development began, the decision to utilize software for rendering was made mainly because PNNL does not have hardware-accelerated rendering resources for the 4-processor Winterhawk-2 nodes or the 16-processor Nighthawk-2 nodes. The number of processors on each node of these SPs also encourages the use of multiple threads to generate geometry and to render images, an activity that may not be readily feasible with single-pipelined hardware graphics accelerators.

3.1 SGE Tunneling

Stretch is a sample application written by Peter Hochschild at IBM provided with the SGE. It is designed to test the operation of SP systems with the SGE. The application simulates a spring-coupled mesh of vibrating spheres and renders pixel-level images representing each step of the simulation. Figure 3 shows the steps each node performs as the demonstration application runs. In the application setup stage, shaded primitive graphics elements are pre-rendered and stored in buffers. These primitives include rods to represent the springs and colored spheres to represent the connections between springs. These primitives are later copied out of their buffers into a larger common buffer when a complete rendering representing the simulation is made.

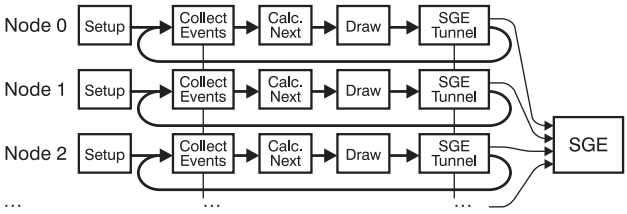


Figure 3. Steps performed by the Stretch parallel application

During this time, the application uses node 0 to create an X window for graphics output. It calls a synchronous SGE library function that places the X window into "tunneled" mode. This enables the window to receive graphics data from multiple nodes via the SGE tunneling library. Each node proceeds to the main calculation and rendering loop when the setup stage is complete. For each loop of the simulation, spring forces and sphere positions are calculated.

A representation of the simulation is pieced together in a large pixel buffer using the pre-rendered primitives. Scaleable rendering performance is achieved by assigning a proportionately-divided horizontal strip of the output window area to each node. For example, when the Stretch application is run on three nodes, each node renders one-third of the complete image.

The tunneling library is then called, which reads the pixel data from the large pixel buffer and transfers them to the SGE. The SGE displays the horizontal strips from all of the nodes as one contiguous image. The tunneling function call is synchronous across all of the nodes.

The user can interact with the simulation by moving the mouse cursor over the spheres. In the simulation, kinetic energy is added to the mesh of springs at spheres where mouse movement occurs. As the application runs, node 0 processes incoming X events (node 0 is the "owner" of the output window). Node 0 uses a synchronous SGE library call to distribute the events to all of the other nodes.

The SGE application achieves a frame rate of greater than 30 frames per second (rendering to an 800x800-pixel window), resulting in smooth animation. The simulation is also responsive to user interaction. Had this application been output to an X terminal, overhead would have been incurred by copying pixel data to a concentrator node (to join together the disjoint horizontal strips from all of the nodes) and then outputting the joined pixel data to an X terminal over the network.

3.2 SMP Rendering: Parallel MPEG Player

In Stretch, multiple processors on the SMP nodes had not been utilized in rendering graphics data. An experimental application was constructed to evaluate SGE performance where all processors on each node would be utilized in image creation. This application is called the SGE MPEG Player and consists of modifications to the Berkeley MPEG Player [1]. It simultaneously decodes many MPEG video streams, positions their outputs into locations within the final pixel buffer, and then uses the SGE to display all the streams. Figure 4a shows a diagram of simplified steps that the original Berkeley MPEG Player performs in decoding an MPEG stream (the frame read/decoding process actually involves further looping). Figure 4b shows the steps performed by the all threads on each node in the modified version, where each thread decodes a unique stream.

When the player starts its setup procedure, it reads a text file that lists all of the MPEG streams to be decoded. Each node identifies the MPEG streams it is responsible for and determines where each MPEG stream should be positioned in the output window. Figure 5 shows an example of the placement of MPEG stream outputs for a set of 3 nodes, each node decoding 2 MPEG streams. A complex region mask is constructed, consisting of a set of rectangles. Each rectangle is positioned corresponding to where pixels from decoded MPEG streams are located in the pixel buffer. Threads are then generated; each thread opens its MPEG stream and performs the functions needed to decode the stream data into RGB pixels. The main thread waits until all decoding threads are finished decoding their frames and then initiates a tunneling operation. A double-buffering scheme is employed to allow decoding threads to begin decoding a new frame while tunneling takes place in the main thread.

An experiment was conducted which involved decoding 48 MPEG video streams to a 3840x960 screen

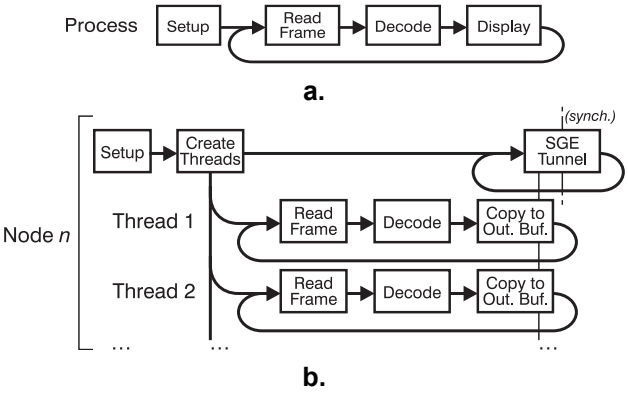


Figure 4. Application sequence of (a) the original Berkeley MPEG player, and (b) the parallel SGE MPEG player

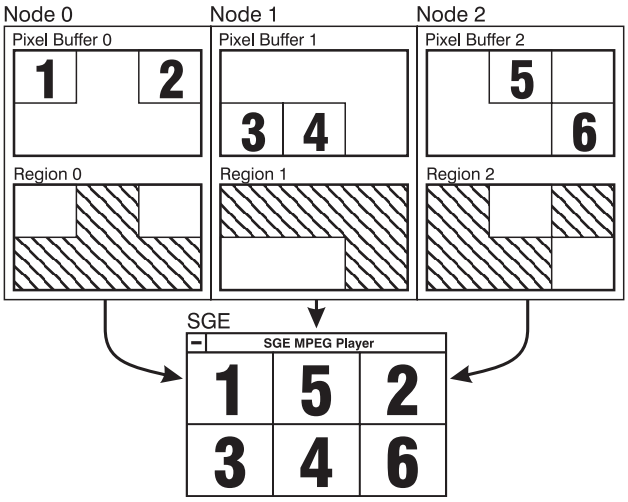


Figure 5. Placement and composition of six MPEG streams rendered by three nodes

area. Each MPEG video stream covered a 320x240 pixel area. Eight nodes were utilized, each decoding 6 MPEG streams. Each video stream was 320x240 pixels in size. The output frame rate averaged 31.0 frames per second (FPS), with a variance of +/– 2.4 FPS. On average, 114 million pixels per second were displayed.

The experiment was done with 4 switch links established between the SGE and the switch. An earlier experiment was performed with a similar setup, but with 2 switch links connecting the SGE to the SP. The maximum frame rate of that experiment was around 18 frames per second. From this result, it can be reasoned that the dominant limiting factor for the MPEG player performance while utilizing 2 links is the SP switch. This also demonstrates that the SGE scales favorably when the number of switch links are increased from 2 to 4. At this time, no other switch configurations have yet been tested at PNNL.

This experiment was intended to require a similar amount of processing to what could possibly occur in a simple multithreaded scientific simulation or large image manipulation application. These results show that multithreaded algorithms running in parallel can produce pixel data which covers a large screen area on the SGE, and that the screen area can be updated to display full motion. Such performance is needed for PNNL's future plans for utilizing IBM's new 204-DPI flat panel display.

3.3 Graphics API Development: SGE/Mesa

A goal at PNNL for SGE software development is to create tools for programmers allowing them to create and customize graphics and simulation applications to render in the parallel environment and output graphics to the SGE. It is also desirable to provide a way for users to

interact with the rendered graphics. In order to achieve that goal, it was logical to begin development by utilizing an existing graphics API which programmers were already familiar. The Mesa [9] and GLUT [14] libraries were utilized for this purpose because the OpenGL API is a commonly-used 3D graphics standard and GLUT is often used for simple, platform-independent window preparation and event handling. Mesa, an implementation of the OpenGL API, has a driver interface that allows programmers to supply code to support new graphics devices. A new driver has been developed at PNNL, called SGE/Mesa [15], to support the SGE and facilitate the handling of pixel buffers in parallel. GLUT was also altered to isolate window-creation calls to node 0, automatically distribute events caught on node 0 to the other nodes, swap buffers using SGE function calls, handle distributed color buffer reallocation on window resize events, and handle complex event-driven tasks such as generating and displaying pop-up menus in interactive parallel applications.

A guiding principle for adding parallel SGE support to Mesa was to require no changes to be made to existing OpenGL application code in order to let the code run in parallel and achieve improved rendering performance. The approach for doing this was to divide the output into a set of regions, each sized proportionately to the number of nodes running the OpenGL application. For example, a four-node execution would divide the output into quarters, as seen in Figure 1. The OpenGL viewports and frustums would then be set automatically according to the boundaries of the regions, resulting in each node having clipping planes to reduce rendering to only the respective region. This scheme works if all geometry to be rendered is provided on all nodes, or if geometry is prefiltered (or bucketed [2]) and provided on nodes maintaining the viewports in which it is to be displayed, similar to what WireGL [7][8] and its successor, Chromium [3], can do.

Advantages for programmers are gained by using Mesa and GLUT to drive the SGE, rather than directly programming for the SGE. First, GLUT hides the details of window creation, SGE window tunnel enabling, event distribution, and display updating for parallel SGE applications. Unlike the Stretch demo, the programmer does not need to know low-level SGE library details, such as how to configure windows for use with parallel tunneling. Second, Mesa does not require programmers to perform pixel buffer allocation and direct pixel manipulation in the SGE pixel format. For example, the Stretch demonstration application allocates an output buffer where direct pixel manipulation takes place in the form of pixel copying. Mesa, however, performs all of the OpenGL polygon-drawing functions into a pixel buffer it maintains and the contents of the pixel buffer are tunneled to the SGE when the GLUT swapping function call is made.

Applications which utilize GLUT and the SGE/Mesa library are structured so that execution flow proceeds as seen in Figure 6. While library functions handle much of the system-related functionality, the application utilizing GLUT and SGE/Mesa performs the next-step calculations (such as object positions or states) and does rendering with OpenGL function calls.

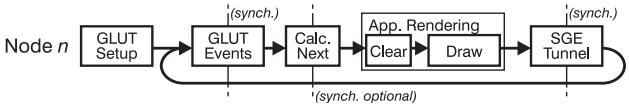


Figure 6. Application sequence of the SGE/Mesa library and the graphics application

Using SGE/Mesa and the modified GLUT library, simple OpenGL applications can achieve increased rendering performance when the code is run in parallel. Most applications in the Mesa distribution's "/demos" directory compile with the SGE/Mesa library and exhibit performance increases when run on more than one node. A preliminary experiment was conducted on two of these applications ("gears" and "gloss") and a customized version of the "gears" demo called "cgears". The "gears" demonstration handles 357 quadrilateral surfaces and 74 triangles, flat-shaded with z-buffering and backface culling. The "cgears" application draws 100 times the geometry of "gears" with backface culling disabled (making it so that all geometry is drawn, regardless of which side of the geometry is to be rendered) and z-buffering disabled. Effectively, 35,700 quadrilateral surfaces and 7,400 triangles are drawn per frame. The "gloss" demonstration draws a glossy teapot through software rendering with diffuse and specular texture mapping at around 5000 drawn polygons. All tests were run with graphics displayed in a 1280x1024-pixel window on the SGE. Table 1 lists the node count, average frame rate, and rendering latency for each test. The overhead incurred in tunneling the rendered images is not included in the latency measurement. For tests run on more than one node, the rendering latencies for the busiest and most idle nodes are listed as "maximum latency/minimum latency".

It can be seen from the benchmark data that utilizing more than 8 nodes with the "gloss" demo and more than 4 nodes for the "gears" demo produces insignificant performance increases and even slightly decreases performance in some cases. It is estimated that this is partially caused by poorly distributed geometry across the fixed viewport regions. The minimum rendering latency values for the high node counts show that at least one node is mostly idle due to this poor load balancing; some of the corner subdivisions do not contain much geometry to be rendered.

Although further work can be done in finding ways to achieve proper load balancing and faster frame rates for increased polygon counts, the preliminary experiment shows promising results for SGE operation: increased performance is observed with multiple nodes performing rendering on different regions of one image.

Table 1: Frame rates (FPS) and max./minimum rendering latency for SGE/Mesa tests

	"gears"		"cgears"	
Nodes	**FPS**	**latency (max/min)**	**FPS**	**latency (max/min)**
1	14 Hz	0.045 sec	0.41Hz	2.4 sec
2	20	0.038/0.026	0.75	1.3/1.3
4	25	0.031/0.020	1.2	0.80/0.57
8	29	0.024/0.018	1.4	0.64/0.085
16	28	0.023/0.005	2.1	0.46/0.051

	"gloss"	
Nodes	**FPS**	**lat. (max/min)**
1	3.5 Hz	0.26 sec
2	6.3	0.14/0.14
4	8.8	0.10/0.090
8	7.5	0.10/0.040
16	8.6	0.10/0.040

3.4 SMP Rendering Threads with SGE/Mesa

The performance to be gained by utilizing multiple processors on each node in the SGE MPEG Player prompted additions to be made to SGE/Mesa to facilitate the use of multiple rendering threads. The utilization of multiple processors was determined to be the next step in development in order to increase the software-rendered polygon count and maintain interactivity.

For the next development phase, functionality was added to allow multiple frames and/or multiple regions to be rendered simultaneously on one or more nodes. It was expected that allowing multiple frames to be rendered simultaneously would increase the frame rate but keep the rendering latency the same as what was seen earlier with SGE/Mesa. Other multithreaded accelerations with geometry transformations, rendering, or compositing were not addressed in this stage of SGE/Mesa development. Figure 7 shows the typical application steps in rendering simultaneous frames on each node.

The timing diagram in Figure 8a shows the progression of execution steps for the main thread and two rendering threads each using their own pixel buffers. The main thread creates the rendering threads and waits for the first rendering thread to report that its pixel buffer is ready. The rendering thread's pixel buffer is ready for tunneling when the rendering thread calls the "Post" function. A frame sequence counter is utilized to associate the first rendering thread with the first, third, etc. frames and the second rendering thread with second, fourth, etc. frames. When the frame is ready, its pixel buffer is tunneled to the SGE. When tunneling occurs, the corresponding rendering thread is suspended since modifications cannot be made to the pixel buffers while tunneling is taking place. A double-buffering scheme can be employed to allow one buffer to be tunneled while another is being rendered, as seen in Figure 8b. This eliminates the need to suspend the rendering thread while its buffer is tunneled.

When multiple nodes are utilized, each rendering thread may be configured to render to a specific frame in a time-sequence. An example of this can be described by two nodes each running two rendering threads. Between the two threads on both nodes, a sequence of four frames can be simultaneously rendered. The two threads on the first node can render the first two frames of the sequence and the two threads on the second node may render the last two frames. In addition, multiple threads can be configured to render to a proportionately-sized region of a frame (the region being sized according to the number of threads across all nodes assigned to render to the same frame in the time-sequence). Each thread utilizes an automatically-set viewport and frustum to create clipping

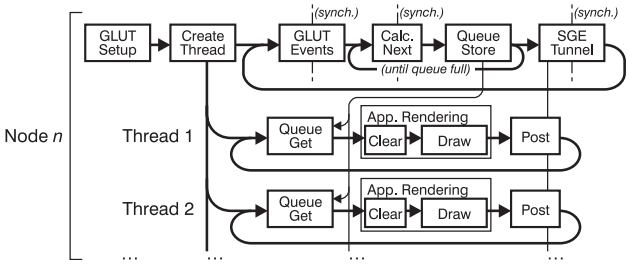
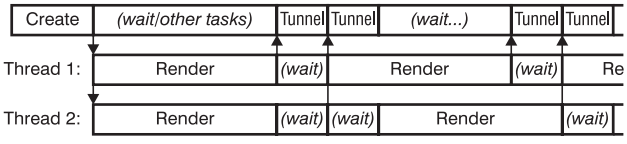
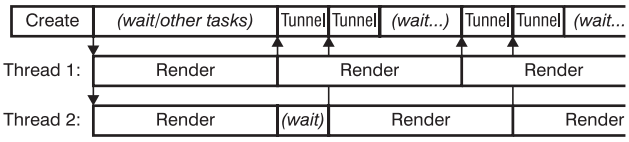


Figure 7. Structure of the SGE/Mesa library and the rendering application utilizing multiple rendering threads



a.



b.

Figure 8. Timing for SGE/Mesa with (a) single-buffering and (b) double-buffering

planes along the borders of the region. For example, four nodes could each employ two rendering threads. Each thread would render to a quarter of a frame, and one-quarter of the output area of the two-frame sequence would be maintained on each node. Since the SGE takes advantage of multiple connections to the switch, dividing the display area across multiple nodes offers optimal switch utilization when tunneling.

Application code utilizing the SGE/Mesa library must be modified from its original format in order to support multiple rendering threads. The reason for this is that special considerations must be made concerning how multiple rendering threads access global variables (such as positions and rotations of objects to draw) in a way that is thread-safe on SMP architectures. To work around this problem without relying upon mutexes in the user application code, a queue data structure was created for use with SGE/Mesa to allow the main thread to store variables, which could be read by the rendering threads in a thread-safe manner. The queuing approach also provides a way to distribute data to threads rendering to particular frames of a sequence. This is important for keeping data that change upon each iteration of the main thread, such as an increasing angle for a rotating gear, in the order that rendered frames would be displayed. The queue can hold data for all running threads. As seen in Figure 7, after the setup stage is complete, a loop causes data to be stored in the queue until it is filled, providing data for all running threads. The SGE/Mesa API reference for maintaining multiple rendering threads is available online at [15].

Table 2 shows benchmarks from multithreaded versions of the Mesa "gears" and "gloss" demonstrations and the "cgears" test application. The multithreaded versions are called "agears", "agloss", and "acgears" respectively. Each test is run at 1280x1024 resolution. Each 4-processor node is running 4 rendering threads, rendering 4 frames simultaneously. Note that the latencies are similar to those reported in Table 1, and that the frame rates for many of the tests are close to the previous rates multiplied by 4. This indicates that the multiple threads in this experiment are rendering at about the same rate as the processes in the earlier experiment. For the "agears" test application, it is estimated that the 61-Hz frame rate limitation may be caused by limited switch link bandwidth to the SGE and overhead in the tunneling process.

Although the frame rate is increased by utilizing multiple rendering threads, there are still limitations that keep SGE/Mesa from efficiently handling the rendering of large numbers of polygons. Most importantly, the load-balancing and geometry sorting limitations of the earlier single-process SGE/Mesa version are present in SGE/Mesa for multiple rendering threads.

Table 2: Frame rates of multithreaded SGE/Mesa applications

	"agears"		"acgears"	
Nodes	**FPS**	**latency (max/min)**	**FPS**	**latency (max/min)**
1	30 Hz	0.052 sec	1.6 Hz	2.5 sec
2	61	0.047/0.035	2.9	1.4/1.4
4	61	0.038/0.026	4.8	0.83/0.61
8	61	0.033/0.024	5.3	0.73/0.096
16	61	0.030/0.024	8.2	0.46/0.058

	"agloss"	
Nodes	**FPS**	**lat. (max/min)**
1	14 Hz	0.28 sec
2	23	0.17/0.16
4	31	0.11/0.11
8	32	0.11/0.051
16	33	0.11/0.051

4. Future Work

Future work will be done to utilize multiple threads in order to achieve an interactive frame rate, but to also decrease rendering latency.

The Chromium library [3] (the successor to WireGL [7][8]) will be evaluated to determine its compatibility with SGE/Mesa. Chromium allows geometry to be distributed from one or more nodes to a series of rendering nodes, each in charge of a region of the output rendered image. While Chromium and its predecessor can distribute geometry to multiple rendering nodes and drive multiple-monitor displays, the same geometry-sorting mechanism can be utilized with the viewport-division scheme employed by SGE/Mesa.

The ability for the SP nodes to effectively handle multiple rendering threads calls for experimentation with partial-geometry compositing functionality. Work will be done to allow graphic fragments generated with multiple threads and nodes to be composited together in sort-last fashion [11]. Compositing will be carried out using a binary-swap approach, similar to what is described in [10]. It is anticipated that the compositing capability will improve load balancing and rendering speed by allowing overlapping fragments of an image to be simultaneously rendered and joined together as a complete frame.

At a future time when PNNL tests the SGE with a Linux cluster, the possibility of utilizing hardware accelerators with the SGE will be evaluated.

5. Conclusion

The SGE's ability to accept pixel data from multiple nodes simultaneously makes it a viable tool for use with

further software development in the parallel environment. With functionality available in libraries for rapidly transferring pixel data from nodes to the SGE, programmers writing applications for IBM SPs and Linux clusters will be able to support direct output of graphics and be able to interact with data. It will be possible to use graphical interaction to steer scientific simulations. These new capabilities for PNNL will ultimately enhance quality of research and versatility of PNNL's computing resources.

When the SGE rendering environment is operational at PNNL, it is planned that the SGE will be used with IBM DX-MPI, (a parallel version of IBM Open Visualization Data Explorer [13]), molecular visualization, volume rendering, and large image manipulation.

6. Acknowledgement

The authors would like to thank Peter Hochschild and Richard Swetz from IBM, T. J. Watson Research Center, who are the principal developers for the SGE. The authors also wish to thank Tom Jackman for being the technical liaison between PNNL and IBM.

This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory at the Pacific Northwest National Laboratory. The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. Pacific Northwest is operated by Battelle for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

7. References

[1] "Berkeley MPEG Player", http://bmrc.berkeley.edu/frame/research/mpeg/

[2] M. Chen, G. Stoll, H. Igehy, K. Proudfoot, and P. Hanrahan, "Models of the Impact of Overlap in Bucket Rendering", *Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*.

[3] "The Chromium Project", http://sourceforge.net/projects/chromium/

[4] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England and L. Westover, "PixelFlow: The Realization", *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, August 1997, pp. 57-68.

[5] P. Hochschild, "SGE Tunnel Programming Notes", http://mscf.emsl.pnl.gov/capabs/mscf/visualization/sge/Sgeprog.pdf

[6] P. Hochschild, R. Swetz, "Scaleable Graphics Engine: High-Performance SP Graphics", http://mscf.emsl.pnl.gov/capabs/mscf/visualization/sge/Sge.pdf

[7] G. Humphreys, I. Buck, M. Eldridge, and P. Hanrahan, "Distributed Rendering for Scalable Displays", *Proceedings of Supercomputing 2000*, October 2000.

[8] G. Humphreys and P. Hanrahan, "A Distributed Graphics System for Large Tiled Displays", *Proceedings of IEEE Visualization Conference* 1999, pp. 215–223.

[9] "The Mesa 3D Graphics Library", http://www.mesa3d.org/

[10] T. Mitra, and T. Chiueh, *Implementation and Evaluation of the Parallel Mesa Library*, Technical report, State University of New York at Stony Brook, 1998.

[11] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A Sorting Classification of Parallel Rendering", *IEEE Computer Graphics and Applications*, Vol. 14, No. 4, July 1994, pp. 23-31.

[12] S. Molnar, J. Eyles and J. Poulton, "PixelFlow: High-speed Rendering Using Image Composition", *Computer Graphics (SIGGRAPH 92)*, July 1992, pp. 231-240.

[13] "Open Visualization Data Explorer", http://www.research.ibm.com/dx/

[14] "OpenGL Utility Toolkit", http://reality.sgi.com/mjk/glut3/

[15] K. Perrine, "SGE/Mesa Reference Guide", http://mscf.emsl.pnl.gov/capabs/mscf/visualization/sge/SGEMesa_Ref10.PDF

[16] G. Stoll, M. Eldridge, D. Patterson, A. Webb, S. Berman, R. Levy, C. Caywood, M. Taveira, S. Hunt, and P. Hanrahan, "Lightning-2: A High-Performance Display Subsystem for PC Clusters", *Computer Graphics (SIGGRAPH 2001)*.

[17] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland, "Scalable Rendering on PC Clusters", *IEEE Computer Graphics and Applications*, Vol. 21, No. 4, July/August 2001, pp. 62-70.