Evaluating Multi-Port Frame Buffer Designs for a Mesh-Connected Multicomputer

Gordon Stoll, Bin Wei, Douglas Clark, Edward W. Felten, and Kai Li
Department of Computer Science
Princeton University

Patrick Hanrahan
Department of Computer Science
Stanford University

Abstract

Multicomputers can be effectively used for interactive graphics rendering only if there are mechanisms available to rapidly composite and transfer images to an external display device. One method for achieving the necessary bandwidth for this operation is to provide multiple high-bandwidth ports into a frame buffer. In this paper, we evaluate the design space of a multi-port frame buffer design for the Intel Paragon mesh routing network. We use an instrumented rendering system to capture the graphics operations needed for rendering a number of three-dimensional scenes; we then use those workloads as input to test programs running on the Paragon to estimate the performance of our hardware. Our experiments consider three major design questions: how many network ports the frame buffer needs, whether Z-Buffering should be done in hardware on the frame buffer or in software on the computing nodes, and whether the design alternatives are scalable.

1 Introduction

Multicomputers have become an important, cost-effective architecture for large scientific and database applications because they have hundreds or thousands of powerful microprocessors, high-bandwidth routing networks, and large amounts of main memory and secondary storage. On the other hand, multicomputers have not played an important role in interactive three-dimensional graphics systems though they seem to fit many graphics problems well. This paper studies design alternatives and tradeoffs in architectural and systems support for an object-parallel graphics system for multicomputers.

The object-parallel rendering process partitions the input to the rendering system on an object-by-object basis, assigning the input geometric objects to processors independently of their position in the final image. This is in contrast to image-parallel rendering, in which the objects in the input are partitioned and assigned to processors according to the image regions they cover. In an object-parallel system, once the pixels covered by each geometric primitive have been determined, all coverage information for a given pixel in the image must be considered together to determine its final color. Thus the final stage in object-parallel rendering can be viewed as a sorting of this pixel coverage information from object space to image space [15, 12, 11].

The basic architecture of an object-parallel graphics system consists of a front end and a back end. The front end is given geometric primitives to render (that is, to transform, shade, and rasterize), and produces streams of image position, color, and depth information. The back end combines these individual streams into coherent images by sorting the sample streams according to image position and using the depth information to achieve hidden-surface removal, a process called Z-Buffering, and sending the resulting images to a frame buffer for display. Although several existing multicomputers can support the front end well with their computing and communication power, there are questions about what architectural and system support should be provided for the back end.

This paper investigates architectural alternatives to support the back end of an object-parallel three-dimensional graphics system. We consider three major questions: how many network ports the frame buffer should receive input from, whether to perform Z-Buffering in hardware in the frame buffer or in software on the computing nodes, and whether the multiport frame buffer architecture design alternatives are scalable.

We evaluate these design alternatives using trace-driven simulations. We use an instrumented rendering system to capture the sequence of back-end operations needed to render a set of three-dimensional scenes. The traces allow us to determine the sequence of computation and communication that would be necessary to render a scene on a particular machine configuration. We then produce programs which run on the Paragon and do the same sequence of local computation and communication steps as the back end of an object-parallel renderer would do in rendering the scenes.

2 Parallel Rendering

Rendering Pipeline

Figure 1 shows the stages performed by a simple three-dimensional rendering system in order to create an image from geometric descriptions of objects in a scene.

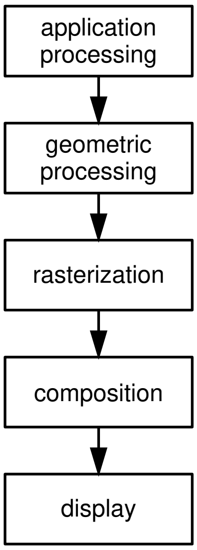

Figure 1: Stages in rendering a scene.

Application processing refers to the activities of whatever application program is creating the scene to be visualized. The application specifies a set of geometric primitives (e.g. triangles) which describe the objects in the scene, light sources to illuminate the scene, and viewing parameters such as the point of view, viewing direction, and field of view. All of these are described in separate convenient three-dimensional coordinate systems.

Geometric processing is the process of positioning the objects in a common three-dimensional "world" coordinate system, determining the interaction of light with the objects, and projecting the objects from the three-dimensional world coordinate system into the two-dimensional coordinate system of the screen.

Rasterization takes each geometric primitive and generates from it a set of pixel-sized objects called *fragments* which specify the color of the object and its distance from the viewer at each pixel location on the screen covered by the projection of the object.

Composition takes all of the fragments that map onto a pixel and computes from them which color to paint the pixel. In the simple cases we consider this consists only of selecting the fragment which is closest to the viewer. This is done by "Z-Buffering": throughout the rendering process the depth of the closest fragment at each pixel is stored in a large buffer, and as each new fragment is computed its color and depth are written to the frame buffer and Z-Buffer if it is closer to the viewer than the previously stored fragment at the same pixel location.

Display consists of transferring the final colors for each pixel location to an output device, in our case a standard workstation monitor.

Bandwidth Requirements

As input is presumed to be generated by an application running on a general-purpose host processor, there must be a point in the processing sequence prior to display at which data is transferred from the host into an external device which implements the remainder of the sequence. We consider here the design of such a device for the case in which the host is a distributed-memory multicomputer.

The simplest solution is a frame buffer which handles only the display stage. All rendering operations up through composition are handled in software on the computational nodes of the host, and the color data for the finished image is then transferred to the device. This solution requires 3.75 MBytes of data to be transferred per frame for today's common workstation displays because there are 1280×1024 pixels and 3 bytes per pixel. A high-quality stereo display requires 11.25 MBytes of data to be transferred per frame because there are 1920×1024 pixels and 2 views per frame.

This type of solution is the only one in widespread use, with transfer generally performed via a HiPPI interface [8]. With HiPPI's physical bandwidth limitation of 100 MB/sec, the theoretical peak performance of a HiPPI frame buffer is 26.7 frames/sec for the workstation display or 8.7 frames/sec for the high-quality display. The workstation rate is close to the standard television frame rate of 30 frames/sec and is suitable for most interactive applications, while the high-quality rate is minimal or even unacceptable for interaction.

A more critical problem is that the HiPPI channel connects to only one computing node. Clearly this "solution" only changes the problem from "how do I get the data to the display" to "how do I get the data to the HiPPI node?" In other words, HiPPI doesn't make the data movement problem any easier; it simply allows us to place the frame buffer some physical distance away from the multicomputer.

Solving this bandwidth problem is the first requirement for a parallel graphics system: it must be able to move data from the computing nodes to the graphics system fast enough to allow interactive frame rates.

One strategy for reaching this goal is to build a *multi-ported* display device which attaches to multiple network ports so it can receive image data in parallel.

Other Back-End Requirements

Another concern is that although solving the bandwidth problem for the simple case of sending only completed frames to the external device might *enable* display at interactive rates from a multicomputer, it does nothing to alleviate the notoriously high demands of the rest of the rendering pipeline for processing, memory bandwidth, and communication.

Within the back-end, composition (Z-Buffering) is the most costly step. This requires sorting the generated fragments by image (screen) location, then discarding all but the "front" fragment in each pixel. Z-Buffering demands high network bandwidth to move the fragments, and high memory bandwidth to perform the fragment comparisons.

It is well established that custom hardware is very effective at satisfying the processing and memory bandwidth demands of rendering, particularly in the later stages of the process [7, 18, 6]. In the simple organization outlined above the general-purpose processing power and memory bandwidth of the computational nodes of the multicomputer are potentially squandered on the highly specialized operations of the latest stages of the rendering pipeline.

This suggests that it might be profitable to do Z-Buffering in the display hardware. In this design, the computed fragments are sent directly to the display; as the display consumes the fragments, it performs the Z-Buffering comparisons before placing the arriving data into frame buffer memory.

Note that this increases the input bandwidth demands substantially due to the inclusion of additional composition information with each pixel and due to the *depth complexity* of the scene, which is the average number of fragments which are composited to form each output pixel. For a moderate depth complexity of 3, the required data transfer per frame for the workstation display is 41.25 MBytes since there are 11 bytes per pixel (2 bytes each for the X and Y position of the fragment in the image, 3 bytes for RGB and 4 bytes for Z), or 123.75 Mbytes for the high-quality display. Although the software Z-Buffering cost is eliminated, the cost of moving data to the display is greatly increased.

3 Frame Buffer Design

Figure 2 shows the architecture of a multi-ported "distributed frame buffer" (DFB) which we are developing. The frame buffer connects directly to the Intel Paragon mesh routing network to provide high bandwidth for pixel updates.

The main idea of the architecture of the frame buffer is to assign to each port of the frame buffer one n-th of the scanlines comprising the frame. The frame buffer uses the Intel Network Interface Chip (NIC), which connects to a processor port of an Intel Mesh Routing Chip, which connects to other Mesh Routing Chips to form a Paragon mesh routing network. The datapath of each port also implements optional Z-Buffering. The subframe memory is double buffered so that one frame can be displayed while the next frame is being drawn. The output control multiplexes the outputs of the subframes to a digital-to-analog converter (RAMDAC), which drives a standard workstation monitor.

Software vs. Hardware Z-Buffering

An important design issue is whether Z-Buffering should be done in hardware or software. The first three stages of the graphics pipeline (application processing, geometric processing, and rasterization) are the same in both cases, so we will focus on the last two stages, compositing and display.

If Z-Buffering is implemented in hardware, the compute processors can send all generated fragments directly to the frame buffer with no local compositing (a *sparse* pixel merge, in the terminology of [11]). Alternatively, compositing could be performed locally on each compute processor, and the resulting frame

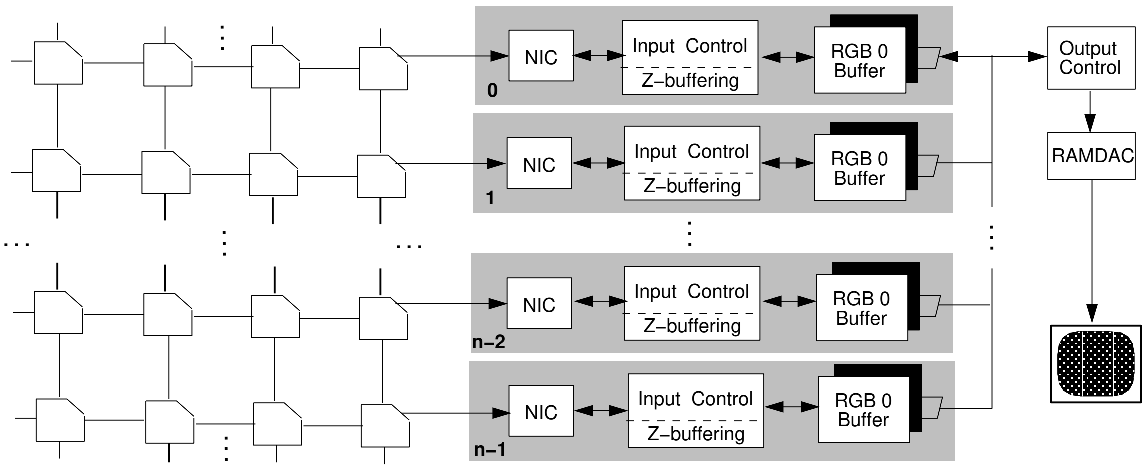

Figure 2: Architecture of an N-port frame buffer for the Paragon network.

sent to the frame buffer. This requires substantially more bandwidth [4, 11] and is not considered here. Given that a sparse merge is used, the data required for each fragment for composition must minimally include the screen coordinates (x, y: two bytes each), depth (z: four bytes), and color (r, g, b: one byte each).

If Z-Buffering is done in software on the compute nodes, three steps must be carried out:

- The fragments must be *redistributed*, so that all fragments that map to the same pixel are on the same processor.
- *Local Z-Buffering* is then done on each processor.
- The resulting finished frame is sent to the frame buffer (*display).*

In both the hardware and software based systems a synchronization operation must be performed among the host processors and DFB ports on completion of each frame in order to separate data from one frame and the next.

Hardware Z-Buffering requires less work from the compute processors, but requires sending more data to the frame buffer, which may create a bottleneck. It is not clear *a priori* which approach is better; our experiments will address this question.

4 Experimental Framework

We use an instrumented version of the Photorealistic RenderMan rendering system [17], as described in [4]. Output library calls have been added in the rasterization subroutines in order to capture every pixel fragment generated by the system. Each fragment record includes the screen coordinates (x, y), depth (z), and color (r, g, b) of the fragment, as well as an identifier of the input primitive from which the fragment was generated. Utilizing this data, we have performed high-level simulations of the composition and display stages of object-parallel rendering on a Paragon system with 64 MP-nodes, running version 1.3 of the Paragon operating system.

Application processing, geometric processing, and rasterization for a given primitive occur on a single host processor. Our simulations ignore the time spent in these front-end stages since it is independent of the design choices we are considering.

The primitives in a scene are assumed to be initially distributed uniformly across the host processors. This is accomplished in the simulation by a round-robin assignment of primitives to processors as used in [12]. The quantity of pixel-fragment data generated from the rasterization stage on a given processor is computed from the trace data according to this primitive distribution.

The next stage in the rendering pipeline is composition (Z-Buffering). How we simulate this depends on whether Z-Buffering is implemented in software on the Paragon or in hardware inside the DFB.

In hardware Z-Buffering, the screen is subdivided into scanlines which are assigned in round-robin fashion to the ports of the DFB. The pixel-fragment data generated from the rasterization stage is redistributed from the compute processors to the DFB ports for composition and display. The quantity of pixel-fragment data which must be transferred to the DFB from each compute processor is computed using the given primitive and scanline assignments. This transfer is simulated on a Paragon system using standard message passing with Paragon processors playing the roles of

scene	primitives	fragments	avg. depth
Capitol	6706	1100369	0.84
Centrosome	3132	2980672	2.27
Bike Shop	9624	2984172	2.28
Sphere	4609	3788533	2.89

Figure 3: Characteristics of the scenes we traced.

both the compute processors and the DFB ports of a given Paragon/DFB configuration. Fragment data is heavily buffered in order to avoid message-passing protocol overhead.

In software Z-Buffering the scanlines are assigned in round-robin fashion to the compute processors, and the following steps are performed:

- The *redistribution* stage is simulated, as above, using standard message passing on the Paragon; the calculation of the amount of data to be moved between each pair of compute processors is the same as that above, except for the assignment of scanlines to processors rather than ports.
- The *local Z-Buffering* stage is simulated locally on the compute processor in the straightforward fashion, with each compute processor compositing the fragments it receives to form a final image of its assigned scanlines.
- The *display* stage requires sending a fixed amount of final image data from each compute processor to a single port of the DFB. Again, this is simulated with standard Paragon message passing.

Per-frame synchronization is performed in both cases. In the graphs, the time for this operation is counted as display time in the software case. In all cases the time for synchronization was negligible.

The Paragon mesh configurations used for the simulations consisted of a square (2x2, 4x4, 8x8) processor mesh simulating the host nodes directly adjacent to a 1xN processor mesh simulating the DFB ports. Note that for configurations with few processors, the configuration of the mesh prevents utilization of the full bandwidth provided by the DFB. For instance, a four-processor configuration will generally not benefit from use of a four- or eight-ported DFB.

Traces

Our simulations were performed based on RenderMan traces from the following scenes:

- *Capitol* - An architectural model of the U.S. Capitol building, constructed in AutoCAD.
- *Centrosome* - A frame from a scientific visualization of a growing centrosome modeled by many small spheres.
- *Sphere* - A frame from an animation of the eversion of a sphere, from the animation *Outside In* [9].
- *Bike Shop* - An interior scene from a photorealistic animation produced using RenderMan [13].

More information about these scenes is shown in figure 3. All scenes were rendered at a typical workstation resolution of 1280 by 1024 pixels.

Experimental Issues

Our experiments address three main issues.

- *Hardware vs. software Z-Buffering*: Should Z-Buffering be performed in hardware in the frame buffer, or in software on the computing nodes?
- *Number of ports*: Do multi-ported frame buffers perform significantly better than single-ported ones? How many ports are enough?
- *Scalability*: How does performance change when the number of processors increases? Does this change the tradeoff between hardware and software Z-Buffering? Does it change the preferred number of ports?

5 Results

Figures 4, 5, 6, and 7 show the back-end time for one frame, for each of the four scenes. For each number of processors, the graphs show eight bars: the first group of four bars shows the time with software Z-Buffering and a 1-, 2-, 4-, and 8-ported DFB; the second group of four bars shows the time with hardware Z-Buffering and a 1-, 2-, 4-, and 8-ported DFB.

Figure 8 shows the imbalance in the amount of data to be sent in the redistribution phase with software Z-Buffering. The "imbalance" is the ratio of the maximum amount of data sent by any processor, to the average amount sent. If this value is close to one, then the load is well-balanced; if it is large then the load is poorly balanced.

Figure 9 shows the imbalance in the amount of data received in the redistribution phase with software Z-Buffering. Note that this is the same as the load imbalance in the local Z-Buffering computation, since the number of fragments a processor must Z-Buffer locally is exactly equal to the number of fragments it receives in the redistribution step.

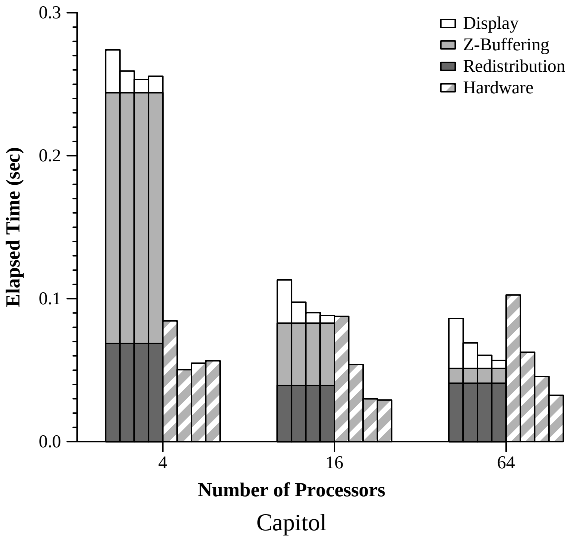

Figure 4: Frame rendering time, Capitol scene. Each group of eight bars corresponds to software Z-Buffering with 1, 2, 4, and 8 ports; and hardware Z-Buffering with 1, 2, 4, and 8 ports. The software Z-Buffering time is broken up into three components: redistribution of fragments among the compute nodes, local Z-Buffering, and sending finished pixels to the display.

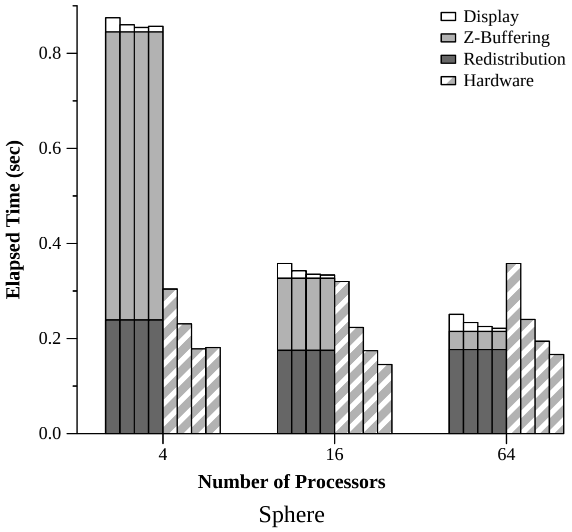

Figure 6: Frame rendering time, Sphere scene.

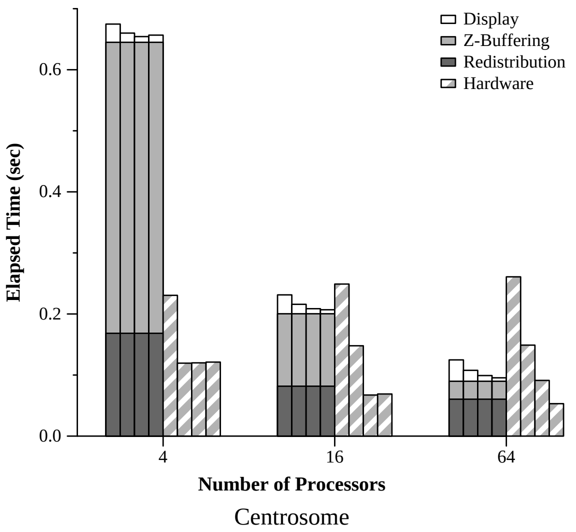

Figure 5: Frame rendering time, Centrosome scene.

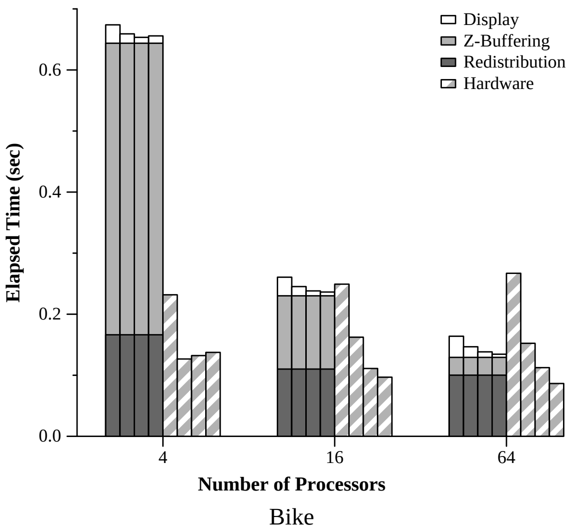

Figure 7: Frame rendering time, Bike Shop scene.

processors	4	16	64	256
Capitol	1.41	1.89	3.71	9.80
Centrosome	1.02	1.28	1.91	3.57
Sphere	2.03	6.20	22.88	89.29
Bike	1.23	3.53	9.93	34.95

Figure 8: Send imbalance for the redistribution step with software Z-Buffering. Each entry shows the ratio of maximum bytes sent by any processor in the redistribution phase, to average bytes sent. In other words, this shows the load imbalance in sending.

processors	4	16	64	256
Capitol	1.01	1.03	1.17	2.22
Centrosome	1.00	1.00	1.01	1.17
Sphere	1.00	1.00	1.01	1.09
Bike	1.00	1.01	1.03	1.18

Figure 9: Receive imbalance for the redistribution step with software Z-Buffering. Each entry shows the ratio of maximum bytes received by any processor in the redistribution phase, to average bytes received. In other words, this shows the load imbalance in receiving.

Hardware vs. Software Z-Buffering

We can draw several conclusions from this data. First, we see that hardware Z-Buffering provides a large benefit when the number of processors is small. Software Z-Buffering requires many memory accesses; with only a few processors the aggregate memory bandwidth isn't high enough to make software Z-Buffering practical.

When the number of processors increases, aggregate memory bandwidth increases; software Z-Buffering isn't so slow anymore. The result is that the performance gap between hardware and software Z-Buffering shrinks as the number of processors grows. The two approaches have roughly equivalent performance with 64 processors.

Number of Ports

Figure 11 shows the benefit gained from increasing the number of frame buffer ports from one to eight, with 64 processors. With 64 processors, software Z-Buffering is usually better for a single-ported DFB, while hardware Z-Buffering is better for a DFB with four or more ports. The number of frame buffer ports makes a bigger difference to hardware Z-Buffering performance than to software Z-Buffering performance. With software Z-Buffering only a small fraction of the time is spent sending the finished frame to the DFB,

	1 port	2 ports	4 ports	8 ports
capitol	118	97	67	47
centrosome	126	110	90	77
bike shop	123	108	73	48
sphere	116	87	54	31

Figure 10: Effective bandwidth per frame buffer port, with 64 processors, measured in MBytes/sec. The peak bandwidth of a Paragon link is 200 MBytes/sec.

so making DFB input faster doesn't make much difference. With hardware Z-Buffering, on the other hand, the entire time is spent transferring data to the DFB, and the DFB input bandwidth is a primary bottleneck. For a scene with good load-balancing characteristics, such as "Centrosome", increasing the number of ports is very effective. For a scene such as "Sphere", with very poor load-balancing characteristics, going from one port to eight gives a speedup of only two. In this case the bottleneck is on the send side of redistribution, where individual compute processors are overloaded with data to transfer to the frame buffer. As can be seen in figure 10, the DFB ports are far from saturation in this case.

Scalability

The two approaches scale differently as the number of processors increases. For software Z-Buffering, the display stage scales well with the number of input ports, and is only sensitive to the number of processors when it is lower than the number of input ports. As the amount of data to be transferred is perfectly distributed on both the send and receive sides, this is as expected. This stage takes only a small fraction of the total time in nearly all cases. The local Z-Buffering stage depends only on the number of processors. As can be seen in figure 9, the interleaved scanline distribution provides very good load-balancing for this operation and as a result it scales well. With 64 processors, the local Z-Buffering time is quite small.

The redistribution time doesn't scale so well. Beyond 16 processors the redistribution doesn't get much faster; with 64 processors it is the dominant step. This is due to the imbalance in the amount of data that the processors must send in the redistribution phase. The redistribution time is limited by the sending bandwidth of the processor that has the most data to send, just as in the hardware case described above. Figure 8 shows this strong imbalance in the amount of data that processors have to send.

At first this imbalance may seem surprising. The scenes contain a very large number of primitives, and

	hardware Z-Buf	software Z-Buf
capitol	3.16	2.11
centrosome	5.05	1.39
bike shop	3.12	1.22
sphere	2.14	1.42

Figure 11: Speedup due to using eight frame buffer ports rather than one, with 64 processors.

the primitives are scattered across the processors in a round-robin fashion to break up underlying coherence in the input. In fact it has often been claimed in the object-parallel rendering literature that load-balancing falls out of the technique automatically [12, 4]. This claim is questioned in [3] on the basis of the potentially high variance in the primitive population, and our results tend to support this. The variance in the number of fragments generated per primitive is very high. Most primitives generate a small number of fragments, but a few primitives modeling such things as backdrops, walls, table tops, etc., generate a very large number of fragments.

This suggests that the scalability of the redistribution phase can be improved by detecting, geometrically subdividing, and redistributing large primitives. This technique has been utilized effectively in a hardware polygon mesh rendering system [5] to alleviate load imbalance in front-end geometric processing. In a system such as Photorealistic RenderMan which uses large complex primitives the detection and subdivision tasks are correspondingly more expensive, and the benefit potentially greater.

Overall Performance

For three of the four scenes, we can reach a display rate of ten frames per second, which is sufficient to make the eye see motion. For the Centrosome scene, which load balances relatively well, a frame rate of nearly 19 frames per second was achieved with 64 processors and 8 frame buffer ports. The bandwidth achieved in this case was 616 megabytes per second, or 38.5% of the available physical bandwidth. We believe that the frame rate for all the scenes can be improved to these levels or better by addressing the load balance problem as discussed above.

6 Related Work

Much research has been done on the use of multicomputers for rendering.

One method of doing this is to use existing graphics machines for post-processing and display. The graphics machines can range from workstations, to specialized graphics machines like PixelFlow, to supercomputers like the CRAY YMP. This approach is costly because of the requirement of moving graphics data between the MPP where it was generated and the graphics machine where it will be displayed.

Networked workstations are widely used for non-interactive rendering. Each workstation can be given a separate frame to compute, and the completed frames put in sequence at the end. This approach may be impractical, due to limitations on file system capacities and I/O rates, network bandwidth, and workstation memory and processing capabilities.

Another approach is to connect a frame buffer to an MPP using a HiPPI link. The standard interface and relative high bandwidth make HiPPI frame buffers available and attractive, and they have been used with many recent MPPs, such as the Intel Paragon, IBM SP2, and Meiko CS2. The maximum bandwidth of the HiPPI channel determines how fast the pixel data can be moved to the frame buffer. This approach does not scale well to higher resolution stereo displays.

In order to better utilize HiPPI's available bandwidth, pixel data has to be merged to the HiPPI interface I/O node and sent to the frame buffer in very large packets. In addition, most systems require that the image stream be serialized. All of these factors argue against using HiPPI for high end graphics applications.

Another way to integrate frame buffers with multiprocessors is to assign part of the processors' memory as the frame buffer memory. For example, in the shared memory Stellar GS1000 [1] machine, virtual frame buffers are allocated out of primary space. The frame buffers can use main memory bandwidth, so accesses can be fast. The problem is that memory bandwidth is already a scarce resource in a shared memory machine; in-memory graphics imposes an extra burden at the bottleneck of the shared memory machine.

The NCUBE/ten system [2] supported a 16-port distributed frame buffer, using a special "graphics node" to manage each piece of the frame buffer. Each graphics node was responsible for 32 pairs of widely-separated pixel columns and had eleven DMA channels, eight for connecting to the compute network, and three for connecting to other graphics nodes. The operating system running on each graphics node managed buffer management and synchronization. Experience with the NCUBE frame buffer showed that message buffering on the graphics nodes was the bottleneck which limited frame rate.

By contrast, in our system no buffer management is required since the hardware consumes incoming data directly from the network FIFO. Memory accesses are pipelined to match the network speed.

The AT&T Pixel Machine [14] distributes the frame buffer among the pixel node array, which connect to the processor node array via a VME bus and a specialized broadcast bus. The merit of this approach is that the design of pixel node array and the processor node array are orthogonal. The system is well-balanced in situations in which most of the processing is associated with pixels rather than with primitives as in our case. But the Pixel Machine is not scalable to greater numbers of processors and the limited amount of memory per processor restricts the complexity of the algorithms that can be run on it.

The Thinking Machines CM-2 [16] has a frame buffer that can be accessed in parallel. The CM-2 frame buffer uses 256 I/O lines connecting to 4K processors. All pixel data from all processors must be routed through these 4K processors to get into the frame buffer. Since the CM-2 is a SIMD machine, pixel data movement can't be overlapped with computation.

Our method for synchronization on control operations is similar in spirit to the network sweeping algorithm developed by Li, Naughton, and Plank [10].

7 Conclusions

Using information from traces of the Photorealistic RenderMan rendering system, we have evaluated a variety of architectures for graphics systems to attach to multicomputers. Our experiments show the following results:

- For small numbers of processors, hardware Z-Buffering is much faster than software Z-Buffering. Software Z-Buffering benefits much more from increasing the number of processors, however, and at 64 processors the performance is about the same.
- Using a multi-ported frame buffer improves performance significantly, especially when hardware Z-Buffering is used.
- The primary barrier to scalability is load imbalance in the pre-Z-Buffering redistribution phase; solving this problem will require detecting, geometrically subdividing, and redistributing large primitives.

We are currently investigating this and other hardware and software issues surrounding the implementation of real-time rendering on multicomputers.

Acknowledgements

We would like to thank Paul Messina, Roy Williams, and Thanh Phung of the CalTech Concurrent Supercomputing Facilities, and Reagan Moore of the San Diego Supercomputer Center for providing Paragon access and exceptional technical support, and Tony Apodaca and Pixar for access to Photorealistic RenderMan and insight into its inner workings, Michael Cox for giving us the Photorealistic RenderMan tracing code. Most particularly, this paper could not have been written without the help of Rich Greco of the Intel Scalable Systems Division, who was tireless in tracking down Paragons and in helping us with after-hours simulation runs.

This project is sponsored in part by ARPA under contract DABT63-92-C-0053 and under grant N00014-91-J-4039, by NSF under grant MIP-9420653, and by Intel Scalable Systems Division. Felten is supported in part by an NSF National Young Investigator Award.

References

[1] Brian Apgar, Bret Bersack, and Abranham Mammen. A display system for the Stellar Graphics supercomputer model GS1000. *Computer Graphics*, 22(4):255–262, July 1988.

[2] Robert Benner. Parallel graphics algorithms on a 1024-processor hypercube. In *Proc. of the 4th Conference on Hypercubes, Concurrent Computers, and Applications*, pages 133–140, March 1989.

[3] Michael Cox. *Algorithms for Parallel Rendering.* PhD thesis, Princeton University, May 1995.

[4] Michael Cox and Pat Hanrahan. Depth complexity in object-parallel graphics architecture. In *Proc. Seventh Workshop on Graphics Hardware, Eurographics Technical Report Series*, pages 204–222, 1992.

[5] Michael Deering and Scott Nelson. Leo: A system for cost effective 3D shaded graphics. *Computer Graphics*, 27(4):101–108, August 1993.

[6] Michael Deering, Stephen Schlapp, and Michael Lavalle. FBRAM: A new form of memory optimized for 3D graphics. *Computer Graphics*, 28(3):167–174, July 1994.

[7] Henry Fuchs and John Poulton. Pixel Planes: A VLSI-oriented design for a raster graphics engine. *VLSI Design*, 2(3):20–28, 1981.

[8] Vineet Kumar. A host interface architecture for HIPPI. In *Proc. Scalable High Performance Computing Conference*, pages 142–149, May 1994.

[9] Silvio Levy, Delle Maxwell, and Tamara Munzner. *Outside In.* AK Peters, Ltd., Wellesley, MA, 1994. (Video).

[10] Kai Li, Jeffrey F. Naughton, and James S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *Intl. J. of Parallel Programming*, 20(3):159–180, June 1991.

[11] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Application*, 14(4):23–31, July 1994.

[12] Steven Molnar. *Image-Composition Architecture for Read-Time Image Generation.* PhD thesis, University of North Carolina at Chapel Hill, Oct 1991.

[13] Eben Ostby and Bill Reeves. A Night in the Bike Store. *Computer Graphics*, 21(4), July 1987. (Cover).

[14] Micheal Potmesil and Eric Hoffert. The Pixel Machine: A parallel image computer. *Computer Graphics*, 23(3):69–78, July 1989.

[15] I. E. Sutherland, R. F. Sproull, and R. A. Schumakeer. A characterization of ten hidden-surface algorithms. *Computer Surveys*, 6(1):1–55, March 1974.

[16] Connection Machine CM-200 Series Technical Summary. Technical report, Thinking Machines Corporation, 1991.

[17] S. Upstill. *The RenderMan Companion.* Addison-Wesley, 1989.

[18] Mary Whitton. Memory design for raster graphics displays. *IEEE Computer Graphics and Applications*, 4(3):48–65, March 1984.