Performance Issues of a Distributed Frame Buffer on a Multicomputer

Bin Wei, Douglas W. Clark, Edward W. Felten, and Kai Li*
Princeton University

Gordon Stoll†
Stanford University

Abstract

A multiple-port, distributed frame buffer has been recently proposed to support parallel rendering on multicomputers. This paper describes an implementation of such a distributed frame buffer for the Intel Paragon routing network, and reports its performance results. We have conducted several experiments with the system we have developed. Our results indicate that placing a multiple-port, distributed frame buffer directly on the host internal routing network can provide high throughput to eliminate the bottleneck of merging a final image from multiple processors to a frame buffer. This architectural approach can also effectively support image composition for sort-last. The synchronization algorithm we have developed requires only one-way communication and minimizes receive overhead for message passing to the frame buffer.

CR Categories: B.4.3 [Input/Output]: Subsystems—Parallel I/O; I.3.1 [Computer Graphics]: Hardware Architecture—Parallel Processing; C.4 [Performance of Systems]: Design Studies.

Keywords: multiple-port distributed frame buffer, parallel rendering, multicomputers, synchronization.

1 Introduction

Many large computing problems require not only high computational power and huge memory resources to perform the computations, but also need interactive visualization capabilities in order to examine and validate their results. Multicomputers provide the computation power and memory resources needed for large-scale computing and simulations, but lack adequate support for real-time rendering. The traditional approach to this problem is to perform the computation on a multicomputer and store the resultant data sets on disks, which are then transferred to a high-performance graphics machine for retrieving and rendering. This approach requires very high I/O performance and it is difficult to get visual feedback interactively. Furthermore, it requires the graphics machine to have enough rendering power and memory space to render the huge resultant data sets. For very large problems such as the ASCI simulation problems [3], this approach is not viable.

* {bw,doug,felten,li}@cs.princeton.edu, 35 Olden Street, Princeton, NJ 08544

† gws@graphics.stanford.edu, Gates Hall 3B, Stanford, CA 94395

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
1998 Workshop on Graphics Hardware Lisbon Portugal
Copyright ACM 1998 1-58113-097-x/98/8...$5.00

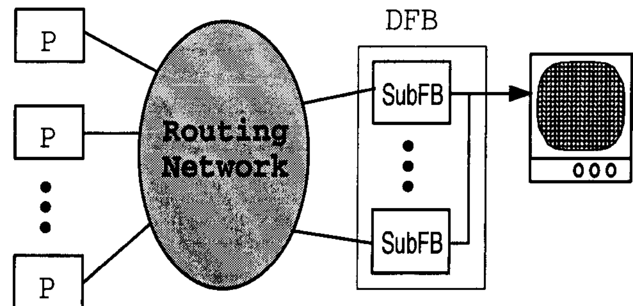

Figure 1: High-level architecture of a multiple-port Distributed Frame Buffer, composed of several Sub Frame Buffers that are directly connected to the routing network.

Another approach is to connect a frame buffer to a multicomputer via a high-bandwidth channel such as HiPPI [14], and perform both computation and parallel rendering on the same machine. Although this approach does not require a separate graphics machine and avoids storing resultant data sets to disks and then retrieving them, the HiPPI frame buffer connection forms a performance bottleneck in the parallel rendering pipeline. This is because a parallel renderer generates image pixels on multiple compute nodes and needs to merge these distributed pixels to the node that has a connection to the frame buffer in order to form a final image. Another problem is that the single channel frame buffer has a limited frame rate, especially for high-resolution displays.

A challenging question is what kind of architectural support for 3-D graphics on a multicomputer one should provide in order to perform interactive visualization. Our approach is to partition a frame buffer into several disjoint parts, or *subframes*, and connnect each subframe to the routing network directly. Thus an entire frame buffer has multiple ports on the network which can be accessed in parallel through the network routing, as shown in Figure 1. This approach has several advantages over the HiPPI frame buffer approach. It relieves the bottleneck of merging and transferring frames through a single channel. It takes advantage of high-bandwidth routing networks to provide scalable performance. It also allows the parallel renderer to perform hidden-surface removal (Z-buffering) at the frame buffer instead of using software sorting among compute nodes.

On the other hand, this approach also poses several design challenges and performance issues. In a previous paper we used traces from the photorealistic RenderMan rendering system [22] to evaluate the potential benefits of using a distributed frame buffer. In this

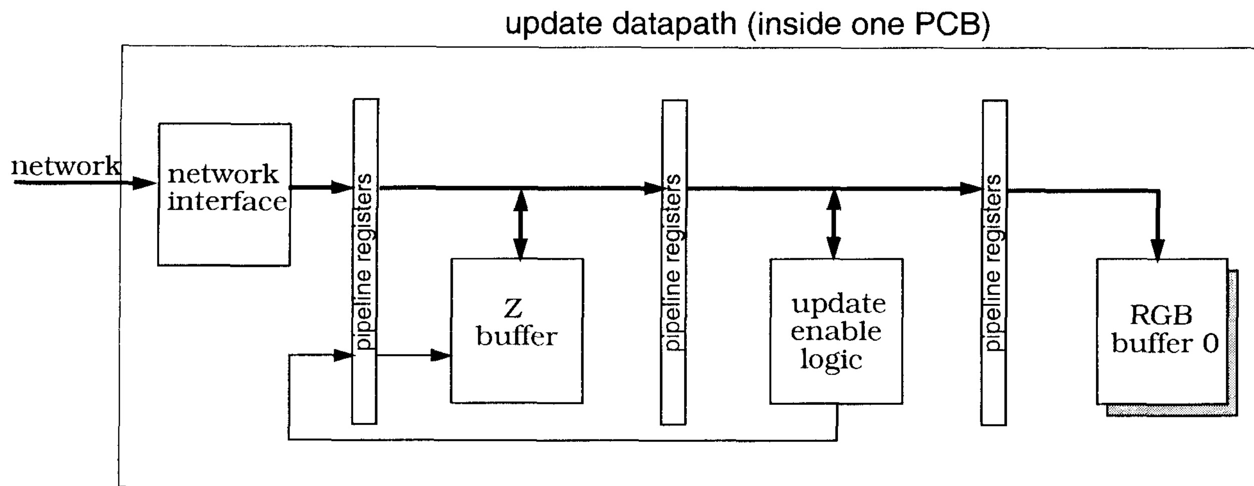

Figure 2: RAIN update datapath.

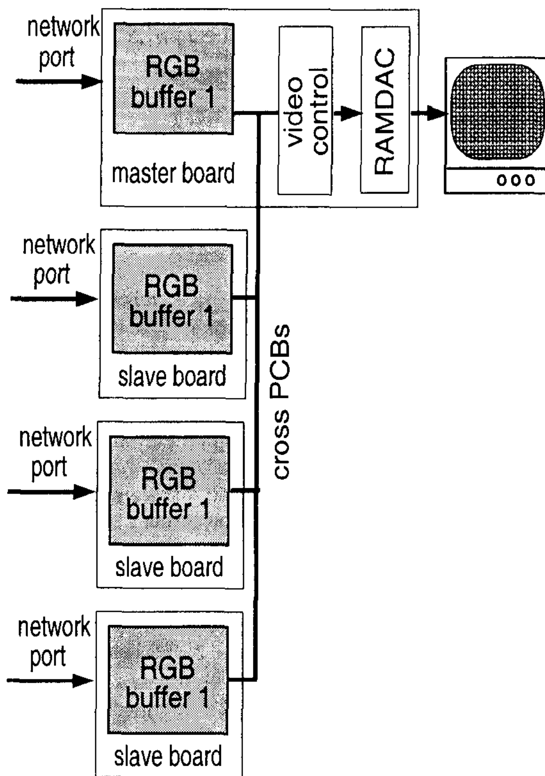

Figure 3: RAIN display datapath.

paper, we extend those discussions and study the real system.

In particular, we are interested in understanding the following performance issues:

- how to provide an efficient synchronization among multiple compute nodes and frame buffer ports; and
- how to improve parallel rendering with a distributed frame buffer.

We have designed and implemented a multiple-port, distributed frame buffer prototype for the Intel Paragon multicomputer. We conducted several experiments to evaluate the design and performance issues on a 64-node Paragon platform. In the rest of the paper, we first describe the system structure and the experimental environment. We then present performance analysis and results concerning each of the design and performance issues in turn.

2 RAIN: A Distributed Frame Buffer

We have implemented a four-port distributed frame buffer for the Paragon routing network. The prototype system is called RAIN, which stands for a **R**andomly **A**ddressable **I**mage Buffer on Multicomputer's **N**etworks.

2.1 Design Considerations

The purpose of designing a prototype system is to demonstrate the idea of a multiple-port frame buffer for general-purpose multicomputers and study performance issues. We considered several factors in designing the system:

- High bandwidth. The fundamental goal of a multiple-port frame buffer is to provide high bandwidth for frame buffer access. High sustained bandwidth is achieved when the frame buffer is able to process incoming data at the rate that the network can generate. This implies that the network interface of the frame buffer can handle input data as fast as the rate of data arrival and that the frame buffer can always have enough memory to hold the incoming data.
- Image composition. One of features we considered is to support hidden surface removal with Z-buffering. This enables an architecture for sort-last [16] on multicomputers. This approach takes advantage of the existing high-speed networks to provide scalable performance needed for image composition.
- Network oriented design. The frame buffer is connected to the routing network of a multicomputer. If the network is used by different systems, the frame buffer should be able to support all of them. In other words, the design is network oriented, rather than machine oriented. In our implementation, we use the Intel Paragon mesh-routing network. Because the Princeton SHRIMP [5] uses the same routing network to connect commodity PCs together, the frame buffer can also be used for SHRIMP.

In addtion, our design is based on simplicity. Since the hardware design is from the scratch, it is important to reduce the complexity of the system design and make implementation feasible with the constraints of time and resources. The goal is to investigate and validate the methodology of placing a multiple-port, distributed frame buffer directly on the routing network.

2.2 Features

RAIN has the following features:

- Multiple ports.
- 200MBytes/sec per port
- Double buffering
- Pixel update for finished frame transfer.
- Z-buffering
- Subpacket concatenations
- Buffer memory clearing.
- Subframe interleaving.
- True color display of 1280 by 1024 at refresh rates up to 76Hz.

The frame buffer is partitioned into four ports, though the architecture allows more than four ports. Each port has memory for a quarter of the screen, which consists of 1280 by 256 pixels. Each port is implemented with one printed circuit board and occupies one slot on the Paragon backplane. One of the ports, called the *master port*, has the video control to generate analog signals to drive a CRT. The remaining ports are *slave ports*. All ports have two buffers for holding images. One is a "display buffer" used for output to the screen. The other is an "update buffer" exported to compute nodes for storing the current rendered image. When a frame is finished, a synchronization is performed and a compute node tells the frame buffer to swap buffers. After swapping, the two buffers exchange their roles for a new frame.

Each port can accept pixel data, either RGB color or RGB color with depth Z, and write to its local memory at the speed of 200MBytes/sec, the maximum data rate provided by the network interface of the backplane. To reduce the communication overhead, multiple packets can be concatenated into one network packet, and the frame buffer will automatically decompose the multiple packets and put the pixels into the corresponding memory locations. There are 8 modes of Z comparisons to perform Z-buffering. The mode can be set by the application using the frame buffer.

RAIN can also clear its frame buffer memory to specified background values. The clearing is performed at all memory modules in parallel. The background values, both RGB color and depth Z values, can be set by a host processor on a per frame basis.

The mapping of the port memory to the screen can be interleaved in units of horizontal scanlines. The number of consecutive scanlines coming from one port memory can be 2^i for $i = 8, 7, , 1$.

2.3 Datapath

Because the frame buffer is used for both update and display at the same time, there are two datapaths in the RAIN system: update datapath and display datapath.

The update datapath consists of four pipeline stages: interface control, Z buffer, write enable logic, and RGB buffer, as shown in Figure 2. All four frame buffer boards use the identical update datapath. Only the master board has the video control and output circuitry assembled.

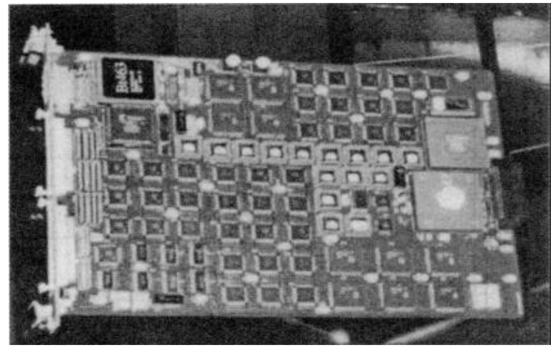

Figure 4: PCB of the master port. Network interface is at the right side. Video control is at the upper left corner.

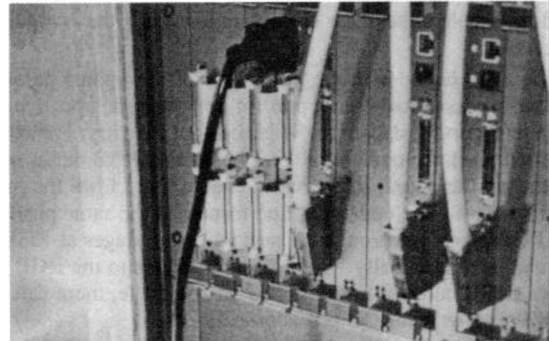

Figure 5: The four-port frame buffer (connected by two sets of ribbon cables) in a Paragon cabinet. The black cable is RGB video output.

The *interface control* is the logic to process incoming packets and transfer data to memory for updates accordingly. It consists of an Intel Network Interface Chip (NIC) and a Xilinx FPGA. The NIC Chip is a standard-cell ASIC that was designed to interface a router on the backplane to a 64-bit processor bus [24].

The Xilinx FPGA controls the access to the NIC for incoming packets from the network. It checks if there is a complete packet in the NIC or the amount of incoming data at the NIC reaches a threshold[1]. It then reads data out of the NIC, interprets header information, and transfers data to the following pipeline stages as needed. The internal logic in the FPGA is a synchronous state machine. Data coming out of the FPGA is 64 bits wide, containing two RGB values (24 + 24), one RGB and one Z values (24 + 32), or two Z values (32 + 32).

Because the incoming data from the network is 8-byte aligned and pixel can start at any memory location, multiplexing and word enabling logic is needed throughout the pipeline stages to route/combine data to the right memory modules.

The *Z buffer* memory is a place to hold the depth information for every pixel in the update buffer. It can perform a read and a write on different memory locations in one cycle. Memory modules are arranged so that there is no conflict for a read of the current Z location and a possible write of the previous Z location.

The *update enable logic* takes another pipeline stage to determine whether an update of Z buffer and RGB buffer needs to be performed for the current pixel. The conditional updates on the values of Z can be programmable in 8 options: less than, less-than-or equal, greater than, greater-than-or equal, equal, always, and never.

[1] NIC has a 2KB incoming FIFO

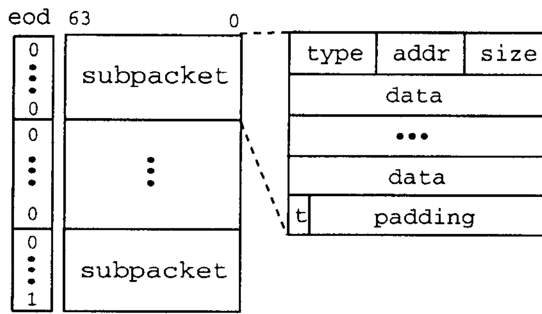

Figure 6: Packet format.

Two other signals, Z_write_enable and RGB_write_enable, can be used to disable either Z or color update even if the comparison is successful.

The *RGB buffer* is in the final stage of the update pipeline. It has two access ports: a write port for data from the update pipeline, and a read port for data to the video control circuitry. For an RGBZ unsorted pixel, the updates of the Z buffer and RGB buffer memory depend on the output of the write enable logic from the previous pipeline stage. The updates are performed at the same pipeline cycle. RGB pixels go through the same pipeline stages as RGBZ pixels, and unconditionally write the color values to the RGB update buffer (RGB buffer 0 in Figure 2). In one cycle, there can be one RGBZ pixel or two RGB pixels processed.

The display datapath consists of display memories, a video control and a RAMDAC, as shown in Figure 3. The video control part is to collect pixel data from the display buffers of all the ports. It also contains a video clock generator which provides the clock that all video control timing is derived from, and video timing circuitry which generates sync and blank signals for a monitor. A RAMDAC is used to handle digital to analog conversion.

Figure 4 is a picture of the printed circuit board of the master port. A case when all frame buffer boards are plugged into a Paragon cabinet is shown in Figure 5.

2.4 Packet Format

Because message passing in Paragon involves in significant software overhead, concatenating multiple packets into one network packet with one network send request can tolerate latency. Thus a generic network packet for the RAIN frame buffer packet consists of one or more subpackets, as shown in Figure 6.

Each subpacket has the same format: a 64-bit header, data of variable length, and a 64-bit padding for the tail. The header contains the packet type, the memory address of the pixel in the subpacket if applicable, and the packet size. The size information can be used to determine the boundary of each individual subpacket, so that the header of the succeeding subpacket can be identified. This packet format supports both Paragon and SHRIMP to communicate with the RAIN frame buffer.

Because the actual data can be any number of 32-bit words and a network packet is always 64-bit aligned, a 64-bit padding is used to provide the adjustment. If the t bit in the padding field is 1 (shown in Figure 6), the low 32 bits of the padding contain valid data; otherwise the padding contains no valid data and is ignored when receiving it at the frame buffer.

The 'eod' is a separate signal indicating the end of the packet is reached. Thus when eod is set, the incoming 64-bit data is the last 64 bits of the current network packet.

2.5 System Support and Graphics Library

We added a new system call to the OSF operating system for the Paragon multicomputer to provide a simple, protected communication mechanism for compute nodes to send data and commands to the distributed frame buffer. Unlike the NX message-passing mechanism, which uses a debit-credit algorithm to manage send and receive buffers [19], the simple communication mechanism for the distributed frame buffer manages no buffers. This is because the distributed frame buffer injects no messages into the routing network and consumes inputs at the data transfer rate of the routing network.

With this simple message-passing mechanism, we designed and implemented a low-level library for graphics libraries or user programs to conveniently access the distributed frame buffer. The library consists of four types of calls:

- *Data Transfer:*

  Provide synchronous and asynchronous messaging for compute nodes to transfer pixel data (RGB or RGBZ) to a frame buffer port. We can concatenate multiple packets into a one larger packet and make one call to send all.

- *Control:*

  Set control registers for different control modes and submit requests for buffer swapping. A swapping request will automatically perform a synchronization among compute nodes and frame buffer ports.

- *Flushing:*

  Flush the network links along the paths that a packet went through. The frame buffer will ignore the flushing packets.

- *Miscellaneous:*

  Includes auxiliary functions such as initialization and termination.

Instead of developing a graphics rendering library for the frame buffer from scratch, we modified a portable Parallel Graphics Library (PGL) [7] by Tom Crockett at the Institute for Computer Applications in Science and Engineering at the NASA Langley Research Center. PGL is a parallel graphics library for distributed memory applications, available on Intel Paragon and IBM SP2. It includes basic functionality for describing, rendering, and displaying three-dimensional scenes.

We have done three main modifications to PGL:

- Tune several time-critical procedures to improve the rendering performance by taking advantage of the graphics pipeline and dual instruction mode of the Intel i860 processors. The tuning effort was mostly through assembly coding in order to use the special features on the i860 processors. We were able to improve the original rendering performance on a uniprocessor by a factor of between 2 and 5.

- Modify PGL to perform sort-middle with the RAIN frame buffer. The original PGL merges finished pixels to form a frame and then sends the frame data through stream sockets to a frame buffer on a workstation. We add a new display format to PGL and send finished pixels (RGB) to the RAIN frame buffer directly. The rasterization directly uses the graphics pipeline of i860 graphics pipeline for scan conversion and Z-buffering through hand-written assembly code.

- Modify PGL to perform sort-last with the RAIN frame buffer. When rasterizing horizontal spans, we store both RGB and Z values for the current span to the send buffer according

to which port the span falls in. Multiple spans are concatenated and sent to the corresponding frame buffer port when the buffer is filled up. Upon receiving RGBZ data, the RAIN frame buffer performs Z-buffering for hidden-surface removal automatically in the update datapath. The rasterization procedure is also modified with assembly code. The original PGL performs Z-buffering in software at each processors' local memory.

These modifications allow us to conduct various experiments to study performance issues.

2.6 Experimental Environment

The platform used in our experiments is a Paragon multicomputer with 64 compute nodes configured in a 16×4 2-D mesh. The four frame buffer ports are placed in a 2×2 mesh on the right side of the mesh of the compute nodes because of the available empty slots in the machine. Each port contributes 256 consecutive scanlines of the screen. The operating system is OSF version R1.4.3b with our modifications to access the distributed frame buffer.

We used four applications to drive the modified PGL in our experiments. The first is a triangle application from the PGL package. The application generates a list of randomly oriented triangles in a unit cube, and renders and displays them, as shown in Figure 7. By changing the number of triangles, we can change the complexity of the scene. We ran the program for two different scenes.

Another application captures a typical way of using PGL for visualization. The application generates a 2-D triangular grid with a special Z coordinate, forming spheres in 3-D, as shown in Figure 8. Selecting different numbers of grid cells in each direction will affect the number and the size of triangles. We ran the program with two different grid sizes.

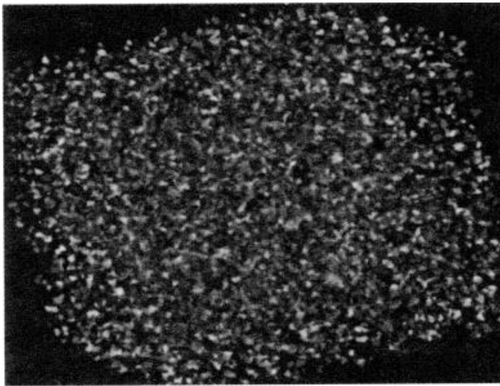

Figure 7: Random triangles.

Table 1 shows the number of triangles and the average size of triangles in pixels of the four scenes at display resolution 1280×1024.

Scenes	Tri-I	Tri-II	Grid-I	Grid-II
Triangles	8192	16384	5800	9800
Avg. Size	78.72	79.27	239.45	102.95

Table 1: Triangle information of four scenes.

In the following sections, we use these applications for discussing the synchronization issue and the contributions of the distributed frame buffer for parallel rendering.

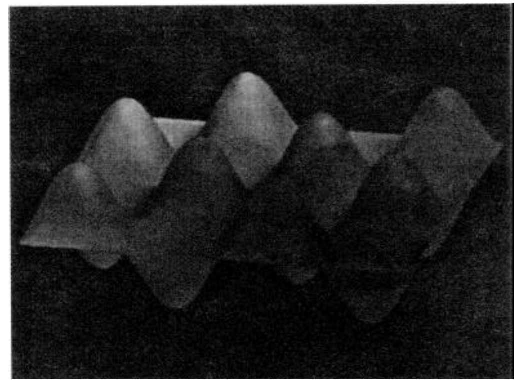

Figure 8: Grids.

3 Synchronization

An important design issue is the synchronization among compute nodes and the multiple-port, distributed frame buffer. Most frame buffer control commands require synchronization. For example, to swap buffers, one must make sure that all pixels to be displayed have arrived at the frame buffer memory. The synchronization algorithm [25] we have presented takes advantage of the property [8] of deterministic routing in the Paragon network. It is similar to the network flushing techniques used for synchronizing a snapshot of computations [15]. The implementation of the synchronization on RAIN is slightly different from [25] in how to handle the notifications among compute nodes. The only hardware support needed for synchronization is to have the distributed frame buffer recognize a special "flush" packet and drop it when it arrives.

The critical information for synchronization is that compute nodes as senders need to know that data they have sent to the frame buffer has all arrived at the destinations. A possible way to solve the synchronization problem is to let the hardware inform compute nodes about its state. This method can be efficient but very complex for two reasons. First, the frame buffer needs to know which compute nodes are communicating with the frame buffer, since it is common that an application uses only a fraction of the multicomputer. Second, the frame buffer needs to have reserved memory buffers on compute nodes if it sends them messages. If it uses the same messaging mechanism as the compute nodesm, it needs to understand the buffer management and flow control algorithms.

We choose to have a simple distributed frame buffer design and implement synchronization in software. The frame buffer can only receive data. This one-way communication requirement makes the frame buffer interface simple and clean as we have seen in the Section 2.

Because routing on the Paragon network is wormhole and dimension-deterministic, packet buffering in the network is limited and the packet routing path from a sender to a receiver is predictable.

Consider a packet which is big enough that it can cover all the network links along the path from a sender to a receiver. If the tail of the packet is leaves the sender's outgoing FIFO, the header of the packet must have arrived at the receiver's incoming FIFO because the network has no place to hold the entire packet. Thus all data previously blocked along the path has been driven into the receiver.

Our synchronization algorithm uses this property to flush pixel data out of the network with dummy "flushing" messages of the appropriate size sent from compute nodes to the frame buffer. Our synchronization selects a proper subset of the compute nodes (*flush nodes*) such that the paths for these nodes to the frame buffer ports will cover all the possible links from a compute node to a frame

1. horizontal *flush request* notification;
2. network flush;
3. vertical *flush finish* notification;
4. synchronization message;
5. vertical *sync finish* notification;
6. horizontal *sync finish* notification.

Figure 9: The synchronization algorithm.

buffer port. Because of the X-Y routing policy, the flush nodes are the end nodes along the X dimension.

When compute nodes need synchronization with the frame buffer, they first notify the flush nodes. These flush nodes send flush packets to the frame buffer ports. After sending flush packets, the flush nodes can notify a particular node, the *coordinator*. When this coordinator receives the notifications from all the flush nodes, it knows that all valid data sent from all compute nodes prior to the synchronization must have arrived at the frame buffer. It then sends global control messages (e.g. swapping requests) to the frame buffer ports, followed by flush packets to make sure the synchronization messages will get into the destinations. After sending the synchronization messages with flush packets, the coordinator sends *notification messages* to inform all the rest of compute nodes that the synchronization is finished.

Thus this algorithm maintains the invariant that a synchronization message is always received by the frame buffer after any data sent prior to the synchronization and before any data sent after the synchronization. There is no deadlock as long as frame buffer ports can always receive data from the network, which is satisfied by the one-way communication interface of the frame buffer.

Figure 9 is the list of the steps for the synchronization. For the X-Y routing, the X axis is in the horizontal direction, the Y axis is in the vertical direction.

Step 1 starts the synchronization process by requesting a network flush. This is different from [25] in that we are not using a global two-way synchronization for notifications among compute nodes. Step 2 performs the network flush. Step 3 notifies the coordinator for synchronization. The coordinator sends synchronization messages in Step 4. This is control information concatenated with flush subpackets. Step 5 and Step 6 are used to inform all the compute nodes that the synchronization is over.

Both horizontal and vertical notifications are one-dimensional many-to-one short packets. For these short packets, merging in a binary tree fashion performs better than merging in a linear fashion.

The synchronization cost consists of two parts: notifications and flushing of network links. If the mesh of compute nodes is R rows and C columns, and the frame buffer has M ports, the time for synchronization is $O(\log C + \log R + M)$, or $O(\log N + M)$, where N is the number of the compute nodes.

Besides the synchronization operation, it is also important that when compute nodes start sending the pixel data of a new image to the frame buffer, they must know if the previous display buffer and update buffer have been swapped. Instead of waiting for a whole frame time (conservatively) to elapse during the synchronization, each node remembers the time when synchronization was performed. When it starts sending the first packet to the frame buffer, it checks to see if enough time has been elapsed after the swap request was issued. If the elapsed time is not enough, it will then wait. The deferred wait can start the rendering of a new frame at each compute node without waiting for the frame buffer to finish the swapping operation.

Table 2 shows the elapsed time spent on synchronization with four frame buffer ports in the four scenes.

# of nodes	8	16	32	64
Tri-I	482	606	746	865
Tri-II	481	645	815	939
Grid-I	455	613	760	893
Grid-II	504	611	819	893

Table 2: Synchronization time in μs of the four scenes.

As the number of compute nodes increases, the synchronization cost increases because of more notification and flush packets. As notifications dominate the total cost, the synchronization cost increases as the logarithm of the number of compute nodes.

The synchronization time is less than 1 ms when the number of compute nodes is 64. For 30 frames per second, this is 3.3% of the frame time.

4 Performance for Screen Subdivision

To understand the contribution of the distributed frame buffer to parallel rendering, there are two cases we need to consider for the distributed frame buffer: *screen subdivision* and *image composition*. In screen subdivision, a final image needs to be merged into a frame buffer. This approach happens when the rendering method is sort-first or sort-middle [16]. In image composition, each rasterization processor processes all its primitives. A step of compositing all these partial full-screen images is needed to form a final image at the frame buffer based on visibility. This approach happens when the rendering method is sort-last. Depending on whether it is *sort-last sparse* or *sort-last full*, each rasterization processor either outputs the rasterized primitives or its partially composited full screen for the composition at the frame buffer.

For comparison, consider the case when a HiPPI frame buffer is attached to the system through an I/O node in the performance analysis.

Table 3 lists the terms that we are going use in the following sections for the discussion of the performance issues of the distributed frame buffer for parallel rendering.

4.1 Image Merging through an I/O Node

Merging a final image through an I/O node is a many-to-one message passing. It has two communication properties:

- Data is sent in parallel from multiple senders, but received in serial at the one receiver.

Term	Definition
A	The resolution of the screen, in pixels.
N	The number of processors.
M	The number of frame buffer ports.
n	The number of primitives.
a	The average size, in pixels, of a primitive.
c	The size of a finished pixel in bytes.
z	The size of an unsorted pixel in bytes.
$B_{recv_i(S)}$	The bandwidth for receiving packets of size S.
$B_{send_e(S)}$	The bandwidth of the external link for size S.
$B_{dfb(S)}$	The bandwidth of sending to one port packets of size S.

Table 3: Terms used in performance evaluation

- Data from each processor has the same size.

Although the time spent for the communication is the maximum of the time spent on senders, receivers, and the network, because multiple senders send packets in parallel and there is only one receiver, the receiver becomes a bottleneck for the whole communication process.

Since the packet size is the same over the multiple senders, the total communication cost can be estimated if the effective bandwidth achieved at the receiver to handle the incoming packets of this particular size is known. Thus the total data divided by the bandwidth would be the time spent on the many-to-one message passing.

The merging of a final image to a frame buffer through an I/O node takes two steps: merging to an I/O node and transferring to a frame buffer. The two steps have to be serialized because of one memory system on the I/O node. As the total data is the product of the screen resolution and bytes per pixel, the time is:

$$T = \frac{A \times c}{B_{recv_i(S1)}} + \frac{A \times c}{B_{send_e(S2)}}$$

where $S1 = A \times c/N$ and $S2 = A \times c$.

Considering a screen of 1280×1024 with each pixel is represented in four bytes, the total data, $A \times c$, is 5MBytes. For a 64-node Intel Paragon, each processor has data of

$$\frac{A \times c}{N}$$

to send, so the packet size is

$$\frac{5M}{64} = 80K$$

The effective bandwidth at the receiver for this packet size is 75MBytes/sec for Paragon with the kernel R1.4.4a. All data will be then combined together and sent to a HiPPI frame buffer. The unblocked data rate is 96Mbytes/sec [14].

Thus the time is

$$T = \frac{5M}{75} + \frac{5M}{96} = 70 + 55 = 125ms$$

This overhead to merge and transfer a finished image to a frame buffer is over the time budget for achieving real time frame rate[2]. As the number of processors increases, the merging overhead is not decreasing and the percentage time spent display for each frame is increasing.

4.2 Image Merging with a DFB

For a multiple-port, distributed frame buffer on the network, each frame buffer port receives the portion of data corresponding to its region of the screen. We assume that there are more processors than the number of frame buffer ports so that receivers become bottlenecks of the communication process as in the arguments of the previous section.

If each processor can send data directly to the frame buffer, the time taken for merging a final image is:

$$T = \frac{A \times c/M}{B_{dfb(S1)}}$$

Considering the 64-node Paragon with a four-port distributed frame buffer. Each port receives a quarter of the final frame data which is $5M/4$. Each compute node has $5M/64 = 80K$ data to send to the frame buffer. One frame buffer port provides 188MBytes/sec effective bandwidth to receive data coming from all 64 nodes. The communication time is:

$$T = \frac{5M/4}{188} = 7.0ms$$

In this case, transferring a finished frame to the frame buffer is well below the time budget for the real time frame rate.

4.3 Sort-middle Rendering with RAIN

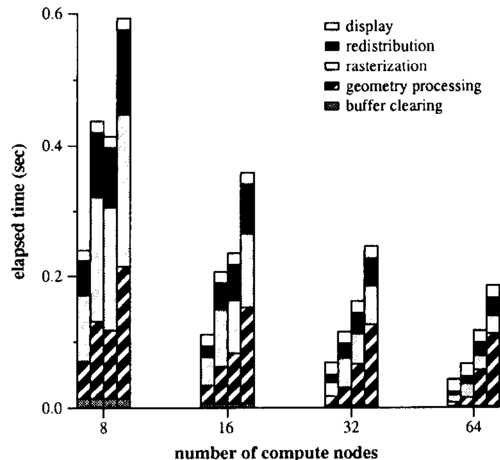

Figure 10: Sort-middle rendering with RAIN. The four columns in each group are four scenes (from left to right): Tri-I, Tri-II, Grid-I, and Grid-II.

We use a sort-middle rendering package to examine the behavior of the frame buffer for screen subdivision.

Most software-based parallel rendering on multicomputers uses the sort-middle rendering pipeline. This class of methods distributes geometry primitives to multiple nodes to perform geometry processing and then redistributes (or sorts) display primitives to certain nodes to perform rasterization according to the partitioning of the screen space.

The original PGL uses the sort-middle approach. Objects from the applications are distributed among compute nodes. Each compute node traverses the scene structure to transform, light, and clip locally-defined instances, generating primitives of horizontal spans to be rasterized. According to the locations of the spans on the screen, primitives are sent *(redistributed)* to the appropriate processor for rasterization.

In other words, each processor is assigned a portion of the screen. It receives incoming primitives, rasterizes them, and sends the finished subframe to its corresponding frame buffer port when the distributed frame buffer is used. For bundling redistribution messages, each node uses a 64KB buffer. In an N-node application, this buffer is further divided into N parts of size 64KB/N, where N-1 buffers are used for sending data to other nodes and one buffer is used for receiving data. For the N buffers, each is also partitioned into two units, so that when one unit is used for the network communication, the other one is still available to the processor to process data.

Figure 10 show the results of the sort-middle rendering with a four-port frame buffer. The total elapsed time consists of 5 parts: clearing processor's local frame buffer memory, geometry processing, rasterization, redistribution, and display for merging image to the frame buffer.

From Figure 10, we can see that the rendering performance improves as the number of processors increased. The time taken for

[2] By real time, we mean 30 frames per second.

image merging to the frame buffer stays stable for different numbers of the processors. This indicates the message passing time is dependent on the total amount of data and the available bandwidth of the four ports. In this experiment, the display time is about $17ms$, so the aggregate bandwidth of the frame buffer is 308MBytes/sec. Because the frame buffer is placed in a 2×2 mesh, the aggregate bandwidth achieved for four ports is restricted by the less than optimal bisection bandwidth.

The percentage of the time spent on display increased as the number of processors increased. For the scene of 8K triangles, the display time is 42% of total elapsed time.

5 Performance for Image Composition

In this section, we only consider *sort-last sparse* for image composition, in which there is no local Z-buffering at processors. Since sort-last full needs every processor to send a partially constructed whole frame to the frame buffer, it is only benefitial if the scene is very large and complex.

5.1 Image Compositing through an I/O Node

Let's assume that primitives are evenly distributed and processors have the same amount of data to send to the frame buffer for image composition. The total data depends on the scene size and bytes per unsorted pixel, which is,

$$D_{sl} = n \times a \times z$$

If there is no memory restriction and each processor sends its final results in one packet, the time taken for the image composition through an I/O node is:

$$T = \frac{D_{sl}}{B_{recv_i(S3)}} + \frac{D_{sl}}{B_{send_e(S4)}}$$

where $S3 = D_{sl}/N$ and $S4 = D_{sl}$.

Consider a screen of 1280×1024 with 8 bytes per unsorted pixel. If a scene consists of 30K primitives and the size of each primitive is 100 pixels on average, in a 64-node Paragon, the time for sending unsorted pixels is:

$$T = \frac{30K \times 100 \times 8}{87} + \frac{30K \times 100 \times 8}{96}$$
$$= 282 + 256 = 538ms$$

We can see that sort-last approach is less practical for a multicomputer with a HiPPI frame buffer. The communication cost is too high to achieve a good frame rate.

5.2 Image Compositing with a DFB

Suppose data is evenly distributed among processors and each processor can send data directly to the frame buffer. The time for image composition with a multiple-port, distributed frame buffer becomes:

$$T = \frac{D_{sl}/M}{B_{dfb(S5)}}$$

where $S5 = D_{sl}/(N \times M)$.

Consider a four-port frame buffer on the 64-node Paragon and following the same assumption as the previous section, the time is:

$$T = \frac{30K \times 100 \times 8/4}{188} = 32.7ms$$

Since sort-last has more data to be sent to the frame buffer, the time for image composition can restrict the frame rate when a scene becomes complex.

5.3 Sort-last Rendering with RAIN

Our distributed frame buffer has the ability to perform hardware Z-buffering. This ability allows rendering programs to composite images for hidden-surface removal by sending unsorted pixels to the frame buffer. We modified PGL to enable sort-last rendering with RAIN.

To use hardware Z-buffering for sort-last, each node has, for each frame buffer port, two 8K send buffers so that while the communication processor is sending packets to the network, there is still a buffer to put spans in for the compute processor. The total buffer space on each node is 64KB. After performing geometry processing and getting horizontal spans rasterized locally, each node will put the color and Z values into the appropriate send buffer according to which port the span falls in. Multiple spans to the same port will be concatenated. When data in a send buffer reaches a threshold, it will initiate a send request and the data in the buffer will be sent to the frame buffer. When rasterization is finished for all objects, all send buffers which still contain spans will be flushed out to the frame buffer.

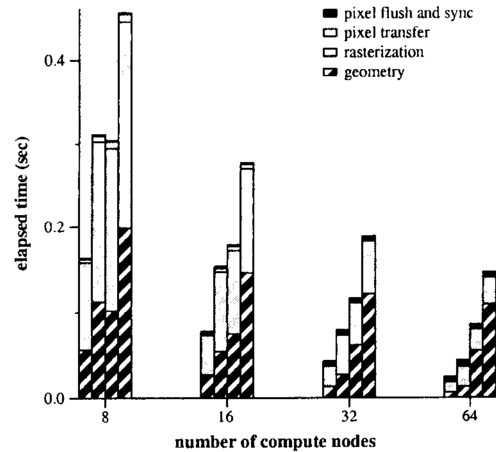

Figure 11: Sort-last rendering with RAIN. The four columns in each group are four scenes (from left to right): Tri-I, Tri-II, Grid-I, and Grid-II.

# of nodes	8	16	32	64
Tri-I	54 369.9	27 178.1	15 90.0	7 39.9
Tri-II	102 719.8	51 356.4	26 175.1	13 80.4
Grid-I	127 904.4	64 450.5	31 220.3	16 106.8
Grid-II	137 977.5	69 486.6	34 237.7	17 113.9

Table 4: Number of packets (the 1st row in each entry) and total packet size in KB (the 2nd row) of a node on average to transfer pixel data to the frame buffer for each of the four scenes in sort-last.

Figure 11 show the timing of the sort-last rendering. The total elapsed time consists of 4 parts: geometry processing, rasterization, transferring unsorted pixels to the frame buffer, and final pixel flush and synchronization. Since processors do not have to maintain a local frame buffer in sort-last sparse, there is no buffer clearing operation at the processor. Instead, each frame buffer port clears it

buffer memory upon swapping buffers. This operation is in parallel with any other processor's operation.

For sort-last rendering, the rendering performance also improves as the number of the processors increases. Because we use 8K buffer for each network packet, data to the frame buffer is diversified in the rendering process. When the number of processors increases, each processor has less data to send to the frame buffer. Since the frame buffer can always receive data at the maximum network rate, the total time for pixel transfer to the frame buffer is also reduced.

Table 4 shows the number of packets and the total packet size of one node per frame in sort-last rendering.

6 Related Work

Obtaining high bandwidth and low latency for frame buffer access is always one of the targets for graphics hardware, ranging from high-end graphics machines, graphics workstations, and graphics support for PCs.

PixelFlow [17, 11], a high-end graphics machine built with the knowledge and techniques of its predecessors Pixel-Planes [13, 10, 12], has a dedicated network for image composition to generate the final image to the frame buffer. PixelFlow also heavily uses enhanced memory chips which associates every sample in memory with some power of computation. These features provide significant bandwidth to the frame buffer for image update.

AT&T's Pixel Machine [20] has a pipeline of *pipe nodes* and an array of *pixel nodes*. The back-end pixel nodes are used for rasterization and a frame buffer is distributed among them. Pixel nodes may communicate with their four neighboring processors, allowing messages to be transferred across the array. Output from the pipeline of pipe nodes is broadcast to pixel nodes. This is an architecture for sort-middle rendering.

High-performance rendering engines on graphics workstations, such as RealityEngine [1] and its successor, InfiniteReality [18], use, for each raster memory board, tens of Image Engines which comprise portions of the frame buffer. For higher speed, multiple raster boards can be configured into the system. With so many image engines, tremendous aggregate bandwidth is available for frame buffer memory.

3D graphics on PCs [23, 6] uses *chunk* rendering techniques. In chunk rendering, a frame buffer is partitioned into multiple tiles, where each tile can be rendered individually to achieve high performance.

Because of the increasing gap between CPU and memory speed, combining logic with memory into a single chip has recently been drawn great attention. Examples are FBRAM [9] and Texram [21]. This on-chip connection significantly improves the access to frame buffer memory.

Using supercomputers for graphics rendering gives parallel applications a unique opportunity for interactive processing. High CPU speed and huge memory resources are also what graphics rendering needs. In the NCUBE/ten [4], 16 graphics nodes are used, among which a frame buffer is distributed. This approach provides a general mechanism to access a frame buffer in parallel. But the system performance suffers from the shortage of memory and message buffers on the graphics nodes and the limited interconnect for communication and synchronization among these graphics nodes. A different approach is to put the frame buffer to the main memory to achieve fast access such as the Stellar Supercomputer GS1000 [2]. This would slow down the general access to the main memory and may require additional memory copies to move pixel data from other areas to the frame buffer.

Our approach is to use minimal hardware support and take advantage of the high computation power, huge memory, and fast communication network of existing multicomputers. Exploiting general-purpose multicomputers for computer graphics also provides a powerful and flexible platform for diverse rendering methods.

7 Conclusions

This paper reports experimental results from a four-port distributed frame buffer we have built on a 64-node Intel Paragon multicomputer with a modified parallel rendering library.

Our results show that a multiple-port distributed frame buffer has several attractive properties. Connecting the distributed frame buffer directly to the host routing network offers high throughput. This high throughput is achieved not merely through the aggregation of multiple ports but also because of the negligible receive overhead of frame buffer ports. The negligible receive overhead is achieved through one-way message passing which is enabled by the synchronization algorithm we have developed. The synchronization algorithm provides a simple interface for frame buffer design and is also efficient.

The multiple-port frame buffer can relieve the bottleneck for merging final images from multiple compute nodes to a (HiPPI) frame buffer through an I/O channel for sort-middle rendering. It also provides an architecture for sort-last rendering and achieves good performance.

Our experiments have several limitations. First, we have only experimented with the scan-line based rendering. Tiling has not been examined for sort-middle rendering. Subpacket concatenations can be used to support tile-based rendering with the RAIN frame buffer for efficiency. Second, we have not addressed load balancing issues and investigated different interleaving factors of the frame buffer for unbalanced scenes. Third, in the current system, there is no oversampling support at the distributed frame buffer. When antialiasing is taken into consideration, it must be handled by compute nodes.

Acknowledgments

We would like to thank Patrick Hanrahan for his initiation of the project, Tom Crockett of ICASE for providing the PGL source code and suggestions for modification, Cezary Dubnicki and Yuqun Chen for the help of implementing the OS support for the RAIN frame buffer, and Paul Messina, Heidi Lorenz-Wirzba, Chip Chapman and Sharon Brunett at the Caltech Center for Advanced Computing Research for their support for the use of Paragon machines. Thomas Funkhouser read the draft of this paper carefully and provided many good suggestions.

This project is sponsored in part by the Scalable I/O project under DARPA grant DABT63-94-C-0049, by DARPA grant DABT63-92-C-0053 and DAAH04-96-1-0212, and by Intel Corporation. Ed Felten is supported by NSF National Young Investigator Award.

References

[1] Kurt Akeley. RealityEngine Graphics. *SIGGRAPH'93, Computer Graphics*, pages 109–116, 1993.

[2] 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.

[3] ASCI. Appliations Overview. *http://www.llnl.gov/asci/applications*.

[4] 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.

[5] Matthias Augustin Blumrich. *Network Interface for Protected User-Level Communication.* PhD thesis, Princeton University, June 1996.

[6] Michael Cox and Narendra Bhandari. Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC. In *Proc. 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pages 25–34, August 1997.

[7] Thomas Crockett. Design Considerations for Parallel Graphics Libaraies. In *Proc. Intel Supercomputer Users Group North American Annual Conference*, pages 3–14, 1994.

[8] William Dally and Charles Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Trans. on Computers*, C-36(5):547–553, May 1987.

[9] 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.

[10] J. Eyles, J Austin, H. Fuchs, T. Greer, and J. Poulton. Pixel-Planes 4: A Summary. In *Advances in Computer Graphics Hardware II*, pages 138–208. Springer-Verlag, 1988.

[11] J. Eyles, S. Molnar, J. Poulton, T. Greer, A.Lastra, N. England, and L. Westover. PixelFlow: The Realization. In *Proc. 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997.

[12] H. Fuchs, J. Poulton, H. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories. *Computer Graphics*, 23(3):79–88, July 1989.

[13] Henry Fuchs and John Poulton. Pixel Planes: A VLSI-Oriented Design for a Raster Graphics Engine. *VLSI Design*, 2(3):20–28, 1981.

[14] Vineet Kumar. A Host Interface Architecture for HIPPI. In *Proc. Scalable High Performance Computing Conference*, pages 142–149, May 1994.

[15] 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.

[16] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics and Applications*, 14(4):23–31, July 1994.

[17] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-Spped Rendering Using Image Composition. *SIGGRAPH'92, Computer Graphics*, pages 231–240, July 1992.

[18] J.S. Montrym, D.R. Baum, D.L.Dignam, and C.J.Migdal. InfiniteReality: A Real-Time Graphics System. *SIGGRAPH'97, Computer Graphics*, pages 293–303, August 1997.

[19] Paul Pierce and Greg Rengier. The Paragon Implementation of the NX Message Passing Interface. In *Proc. Scalable High Performance Computing Conference*, pages 184–190, May 1994.

[20] Micheal Potmesil and Eric Hoffert. The Pixel Machine: A Parallel Image Computer. *Computer Graphics*, 23(3):69–78, July 1989.

[21] A. Schilling, G. Knittel, and W. Strasser. Texram: A Smart Memory for Texturing. *Computer Graphics and Applications*, 15(1):32–41, January 1989.

[22] Gordon Stoll, Bin Wei, Douglas Clark, Edward Felten, Kai Li, and Patrick Hanrahan. Evaluating Multi-Port Frame Buffer Designs for a Mesh-Connected Multicomputer. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture.* (Santa Margherita Ligure, Italy, June 22-24, 1995), pp. 96-105.

[23] Jay Torborg and James T. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. *SIGGRAPH'96, Computer Graphics*, pages 353–364, August 1996.

[24] Roger Traylor and Dave Dunning. Routing Chip Set for Intel Paragon Parallel Supercomputer. In *Proceedings of Hot Chips '92 Symposium.* (Standford, California, August 9-11, 1992), [slide] 7.1.1-7.1.3.

[25] Bin Wei, Gordon Stoll, Douglas Clark, Edward Felten, Kai Li, and Patrick hanrahan. Synchronization for a Multi-Port Frame Buffer on a Mesh-Connected Multicomputer. *Parallel Rendering Symposium*, pages 81–88, October 1995.