Synchronization for a Multi-Port Frame Buffer on a Mesh-Connected Multicomputer

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

Patrick Hanrahan†
Stanford University

Abstract

Parallel rendering on multicomputers involves the parallelization of geometry processing, rasterization and composition. A known approach to support the back end of parallel rendering on multicomputers is to connect a multi-port frame buffer directly to the multicomputer routing network to take advantage of the aggregate bandwidth available on the network. However, a multi-port frame buffer design raises the question of how to synchronize the processors with the frame buffer in order to perform global control operations. The challenge is to provide a simple and efficient synchronization algorithm that requires minimal hardware support. This paper describes a software-based solution to the synchronization problem for a multi-port frame buffer on the Paragon mesh routing network. Simulations on the Paragon multicomputer show that our algorithm is indeed efficient.

Key Words and Phrases: Parallel rendering, mesh routing network, multicomputers, multi-port frame buffer, synchronization.

1 Introduction

In order to use multicomputers effectively for parallel three-dimensional graphics rendering, there must be mechanisms to rapidly composite and transfer images from processor nodes to a display device. On today's multicomputers, frame buffers are normally attached through I/O channels, and HiPPI frame buffers are commonly used [6, 7, 13, 17, 21]. The bandwidth available to the frame buffer thus is limited by the HiPPI channel itself, as well as by the I/O node which controls the HiPPI channel. For parallel rendering on existing multicomputers, this is a serious problem [8, 9, 11, 18, 23].

Connecting a frame buffer directly to the multicomputer's routing network can achieve better bandwidth. Scalability can be improved by connecting the frame buffer to the network using multiple ports.

An important issue for multi-port frame buffer designs is how to synchronize the frame buffer with the compute nodes. Synchronization is necessary for control operations such as starting a new frame. The synchronization overhead must be small compared to the time required to send a frame. Because there is no single channel that all data goes through, synchronization with multi-port frame buffers is quite different from the single-port case [4, 14] where the ordering of data through the port can always be known. The challenge is to provide a simple and efficient synchronization algorithm that requires minimal hardware support.

This paper describes a software-based solution to the synchronization problem for a multi-port frame buffer on the Paragon mesh routing network. We have simulated our algorithm on a 512-node Paragon multicomputer and showed that the overhead of our algorithm is small.

Related Work

Multi-port frame buffers are used in many graphics machines.

In the PixelFlow [19] machine, commands are issued by a parallel host computer which also generates frame synchronization commands.

In pipeline-structured graphics machines [1, 10, 12], control operations can be inserted serially into the graphics pipeline. These machines have special hardware support for synchronizing parallel pipeline stages.

In some parallel graphics systems, such as that of the NCUBE/ten [2], general-purpose processors are associated

*Department of Computer Science, Princeton NJ 88544. [bw,gws,doug,felten,li]@cs.princeton.edu

†127 Center for Integrated Systems, Stanford CA 94305. hanrahan@cs.stanford.edu

with each port of the distributed frame buffer. These processors can participate in ordinary global barrier operation, making frame synchronization relatively easy. Using general-purpose processors, however, adds cost and complexity to the frame buffer design.

When a multi-port frame buffer is connected to a SIMD machine like the CM-2, synchronization is easily handled by the global instruction sequencing hardware. In this case, the price is the lack of generality of SIMD designs.

Instead of distributed control, the frame buffer in the J-Machine uses a centralized video controller [16]. Thus synchronization between processing elements—message driven processors (MDPs)—and the frame buffer is handled by the video controller. The MDPs halt at a barrier synchronization until they receive a notice from the video controller.

A companion paper [20] reports our evaluations of design alternatives of multi-port frame buffers including 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 multi-port frame buffer architecture design alternatives are scalable. That paper however did not address the synchronization issue.

2 Design Overview

We have designed a multi-port frame buffer for the Intel Paragon routing backplane. The frame buffer can be used in either the Paragon multicomputer or the SHRIMP multicomputer [3], which uses the Paragon backplane to connect commodity PCs.

Our design goal is to provide a frame buffer capable of sustaining the data rate produced by parallel rendering. We also aim to put into the frame buffer some functions to help with parallel rendering.

Our initial prototype frame buffer connects to the routing backplane via four ports, though our design can be scaled to sixteen ports. Hardware Z–buffering is implemented in the frame buffer to support hidden surface removal.

The Paragon Mesh Routing Network

An Intel Paragon multicomputer consists of a two-dimensional array of processing nodes connected by a routing network. Each point on the routing mesh contains an iMRC chip [22]; the iMRCs collectively allow messages to be routed between any two points in the mesh. Each iMRC has four bidirectional ports for connecting to neighboring iMRCs, and one bidirectional port for connecting to a compute node.

The network uses wormhole routing, so it has very little buffering capability. The finite buffering property on each link is very important for our synchronization algorithm[1], as described in section 4. Message traffic moving from one router to another is pipelined along the wires so that speed becomes independent of distance for practical purposes. The network bandwidth is in excess of 200MB/s per link.

The network uses dimension-order routing: each message is first routed along the X-dimension, and then along the Y-dimension to the destination. This determinisic routing property is also very important for the synchronization algorithm described in section 4.

Frame Buffer Hardware

Figure 1 shows the block diagram of the 4-port frame buffer when the frame buffer ports are along the Y-dimension on the right side of the mesh. An alternative layout is along the X-dimension on the top of the mesh. The traffic patterns to the frame buffer are different for the two layouts because of dimension-order routing; this will also affect the synchronization overhead. We do not consider layouts in which frame-buffer ports are mixed in with the processor ports, because this would require significant rewriting of the Paragon's operating system.

To interface with the Paragon routing network, we use the Intel network interface chip (NIC) [22] that the Paragon compute nodes use. The frame buffer hardware is designed to accept data at the maximum network bandwidth.

The frame buffer ports implement disjoint subframes which collectively cover the entire display space. The mapping of subframes to the screen is flexible: it can be set up so that subframes are interleaved at a programmable number of horizontal scanlines. All subframes are double-buffered to allow one buffer to hold the currently-displayed frame while the next frame is arriving in the other buffer. Once the new frame has arrived, the roles of the two buffers are swapped to update the image. Each subframe is implemented on a separate printed circuit board which can be plugged into the Paragon routing backplane as if it were a Paragon compute node. One of the boards includes the output control logic and the true-color RAMDAC [5] which drives a standard 24-bit RGB display with a resolution of 1280×1024 pixels at a 72Hz refresh rate.

Compute nodes can send data to the frame buffer just as they do to other processors. The frame buffer can either accept pixel data of a finished frame, or handle Z-buffering for hidden surface removal. The unit of communication from processors to the frame buffer is a horizontal *span* made up of a contiguous sequence of pixels. For a machine which has effective support for small packet transfer such as SHRIMP [3], one span per packet is efficient. For machines with high sending overhead, small packets are concatenated before sent to the frame buffer. Thus, each message from a

[1] Significant buffering does exist at the entry and exit points of the network, but this does not affect the correctness of our synchronization algorithm since it cannot affect the order in which messages are transmitted or delivered.

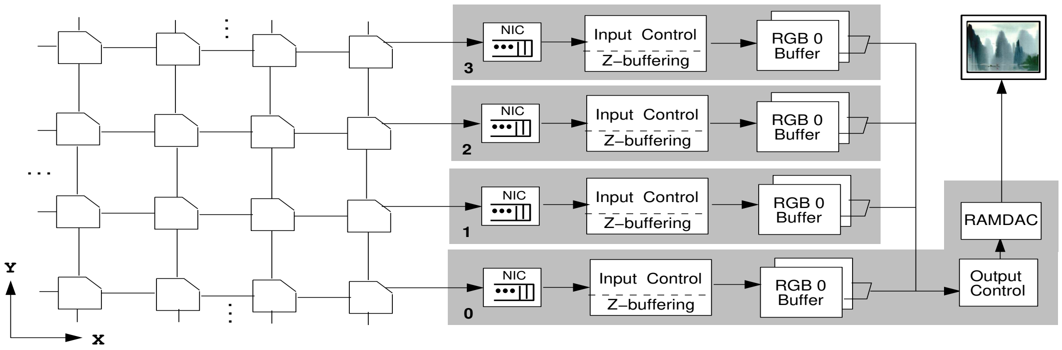

Figure 1: *Architecture of a 4-port frame buffer for the Paragon network.*

compute node to a frame buffer port contains one or more spans.

The frame buffer constantly drains packets from the network to prevent network congestion and deadlock. Depending on the packet type, frame buffer ports either perform Z-buffering, update image buffers, or write data to control registers. A special kind of message, called a *flush message*, is simply discarded by the frame buffer; flush messages are used to implement the synchronization algorithm described in section 4.

Because frame buffer ports are on the multicomputer's routing network, each packet is routed to the proper subframe by the mesh routers. There are no separate interconnections for routing among the frame buffer ports.

Since our algorithm never requires frame buffer ports to send messages, the frame buffer can only receive data from the network. This greatly simplifies the design of the frame buffer hardware.

Software Support for the Frame Buffer

We provide two low-level calls to support data transfer to the frame buffer: `MFB_DrawSpan` which queues pixel data for transmission to the frame buffer, and `MFB_FlushSpans` which flushes queued pixel data to the frame buffer.

```
int MFB_DrawSpan (
        int32 x, int32 y,
        int32 numPixels,
        int32 pixelType,
        int32 *spanBuf );

int MFB_FlushSpans ( void );
```

`MFB_DrawSpan` sends a contiguous horizontal span to the frame buffer. It supports the flexibility of the frame buffer to allow interleaving at scanlines among the subframes. It takes as arguments the screen location of the span, the number of pixels in the span, the type of pixel data in the span (either RGB pixels or RGB/Z fragments), and the location of the pixel data in memory.

The transfer of pixel data by `MFB_DrawSpan` is asynchronous and `MFB_FlushSpans` forces completion of pending pixel transfers. Specifically, when a call to `MFB_FlushSpans` returns, the pixel buffers from all preceding calls to `MFB_DrawSpan` are available for reuse. The implementation is thus free to buffer spans from multiple calls to `MFB_DrawSpan` until either `MFB_FlushSpans` is called, or the buffered data is too large.

These operations are supported by a single system call, `MFB_Send`, which injects packets into the routing network for transfer to the frame buffer.

The packet formats used with the frame buffer are based on the Paragon packet format, but can also be used on SHRIMP. The first 64-bit word contains header information, such as packet type, starting address and data length. The remaining 64-bit words, except the last in the packet, contain two valid 32-bit fields. The final 64-bit word can contain either 32 or 64 bits of data.

We also provide a call that implements global synchronization.

```
int MFB_Barrier ( MFB_Flags Flags );
```

`MFB_Barrier` synchronizes the compute nodes and the frame buffer in order to perform control operations with globally consistent semantics. The `Flags` contain the control information for setting control registers, swapping buffers, clearing the image buffer, and clearing the Z buffer.

All frame buffer operations which are requested by any node before calling `MFB_Barrier` are guaranteed to be performed by the frame buffer before any operations specified in the `MFB_Barrier` call itself and before any operations requested by any node after its corresponding call to `MFB_Barrier`.

`MFB_Barrier` is implemented using the algorithm described in section 4.

3 Synchronization Issues

All pixel data sent to the frame buffer is associated with some mode describing how the pixels are to be stored into the frame buffer. Changing the mode of the frame buffer requires a global synchronization between the compute nodes and the frame buffer. For example, swapping buffers in a double-buffered frame buffer requires a synchronization to guarantee that all of the completed frame is in the frame buffer before swapping. Another example is when the compute nodes need to send stencil data to the frame buffer. They need to synchronize with the frame buffer to prevent the stencil data from being overwritten incorrectly. Although only global control operations need synchronization and such operations occur at most a few times per frame, synchronization should be fast in order to sustain the bandwidth available to the frame buffer.

In the case of a single-port frame buffer, synchronization is simple: all data and commands sent to the frame buffer are serialized because they pass through a single network port. With a multi-port frame buffer, disjoint paths exist between multiple senders (compute nodes) and a given receiver (a frame buffer port), so using only the knowledge of when each packet is sent is insecure for synchronization—it is necessary to know when packets *arrive* at the frame buffer. The naive idea to do a barrier synchronization among the compute nodes is inadequate, since a packet sent after the barrier might still arrive at the frame buffer before some packet that is sent before the barrier.

One way to perform a global synchronization is to let the compute nodes send synchronization messages to all frame buffer ports and let each frame buffer port keep track of the synchronization messages in order to know when synchronization is done for that particular port. Global synchronization completes when all frame buffer ports pass the barrier.

This method is inefficient because it requires that every compute node send a packet to every frame buffer port on every global synchronization. In addition to causing a lot of network traffic, this requires frame buffer ports to count the synchronization messages; the counters must be set correctly before the synchronization, otherwise deadlock occurs or pixel data are mixed up.

The key point is to prevent any blocked *pre-sync* packet from remaining in the network when any synchronization related message enters the frame buffer. We use a synchronization algorithm which depends on the knowledge of synchronization between compute nodes together with the network sweeping technique [15].

4 The Synchronization Algorithm

In this section, we present a synchronization algorithm by considering the frame buffer hardware, the software and the interconnection network. The synchronization algorithm must satisfy the following requirements:

- The frame buffer sends no network packet back to the compute nodes. This requirement can greatly simplify the design of the multi-port frame buffer since there will be no arbitration logic needed to interface with the bidirectional NIC chip.
- The frame buffer cannot produce deadlocks due to user processes' mistakes. This requirement implies that the frame buffer should always be able to receive incoming packets independent of how user processes use it.

If the amount of internal message buffering in the network is M bytes per link traversed, and node A and node B are separated by N links, then the following property holds:

> *If node A sends a message of size $M \times N$ to node B, that message will arrive at the input FIFO of node B before any subsequent message sent by A arrives at the input FIFO of its destination.*

Our synchronization algorithm uses this property to flush pixel data out of the network with dummy "flushing" messages of the appropriate size sent from the compute nodes to the frame buffer. The algorithm is described in Figure 2. (Assume that frame buffer ports are configured on the right side of the mesh: locations with X maximal as in Figure 1.)

```
/* pre-sync data transfer */
synchronization of all compute nodes
if (node x-coord == 0) {
  send pixel_flush message to each frame buffer port
  if (node y-coord > 0) {
    send 'flush_complete' messages to node(0,0)
    wait for 'notice' message
  } else {
    wait for 'flush_complete' messages
    send 'sync' message to frame buffer port 0
    send 'sync_flush' message
    broadcast 'notice' message
  }
} else {
  wait for 'notice' message
}
/* post-sync data transfer */
```

Figure 2: *The Synchronization Algorithm.*

The algorithm maintains the invariant that a *sync* message is always received by the frame buffer after any *pre-sync* data and before any *post-sync* data. To maintain the ordering, *flushing* messages are used to prevent data from staying in the network.

The first step of the algorithm is a synchronization between the compute nodes. This step is needed to ensure that all compute nodes have finished sending *pre-sync* data[2]. The *pixel_flush* messages force all *pre-sync* data to enter the frame buffer ports because of the x-y deterministic routing strategy. The nodes sending the *pixel_flush* messages are chosen to be the minimum set to flush all links along the X-dimension of the mesh and all links along the Y-dimension at the frame buffer ports, i.e. all possible links on all routing paths to the frame buffer ports. Then the minimum set of nodes synchronize to one member of the set. At this time, all *pre-sync* data have been flushed into the frame buffer ports. Thus the *sync* message will arrive at the frame buffer port after all the *pre-sync* data. The use of the *sync_flush* message before broadcasting the *notice* message is to make sure the *sync* has entered the frame buffer port before any compute node gets the notification to proceed.

The above synchronization algorithm is different from the one described in Section 3 in the following aspects:

- The latter requires that all the compute nodes send synchronization messages to the frame buffer while the former only requires some boundary nodes send *flushing* messages.
- The latter requires that each frame buffer port keep track of synchronization messages and send back the acknowledgement. The former uses *flushing* messages to keep the ordering.
- If a buffer swap is the reason for a global synchronization, then in the latter case frame buffer will wait for the vertical retrace, swap buffers, and send back the acknowledgement. While in the former case, a wait of a frame time will be inserted before the *notice* message is sent out.

No deadlock can exist for the algorithm in Figure 2 as long as frame buffer ports always drain data from the network. The running time of the algorithm is determined by the synchronization and broadcast between compute nodes, and the pixel flushing. The complexity of the global operations between n compute nodes is $O(\log n)$ if the global operations are implemented using tree reduction. The complexity of the pixel flushing is on the order of the number of frame buffer ports, or $O(m)$. The total time is thus $O(\log n + m)$.

For the layout in which frame buffer ports are on the top row of the mesh, the set chosen for sending *pixel_flush* messages consists of nodes with Y-coord=0 and the algorithm is similar to the one described in Figure 2.

[2] An alternative is that each compute node with x-coord>0 sends a message to the node with x-coord=0 of the same row which will then send pixel_flush messages.

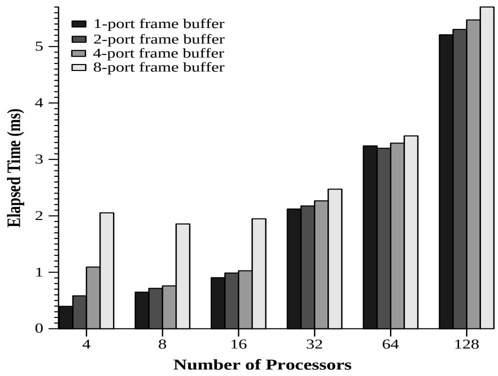

Figure 3: *Synchronization time for the frame buffer configured along the Y-dimension on the right side of the mesh.*

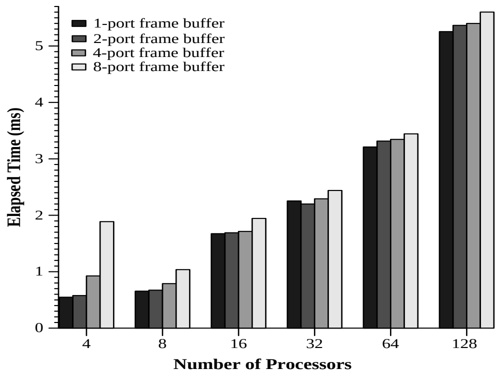

Figure 4: *Synchronization time for the frame buffer configured along the X-dimension on the top side of the mesh.*

Architectural Support

The synchronization algorithm requires only simple hardware support: each frame buffer port must always process arriving messages as long as the NIC input FIFO is not empty; each frame buffer port should recognize the flush message and drop it immediately.

5 Simulation

We simulated our design for two reasons: to measure the real cost of the synchronization, and to measure the overhead of synchronization when it is integrated into the process of finished frame data transfer to the frame buffer.

We used a 512-processor Paragon machine for the simulation. Frame buffer ports were simulated using compute nodes which simply receive data and discard it. The frame buffer was configured as 1-, 2-, 4-, and 8-ported. The number of compute nodes could be 4, 8, 16, 32, 64, or 128.

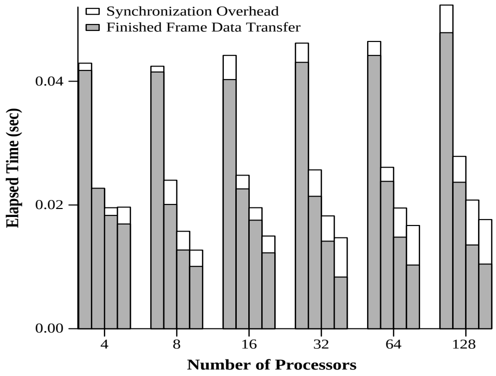

Figure 5: *Synchronization overhead for finished frame data transfer when the frame buffer is configured along Y-dimension. Each group of 4 bars corresponds to 1-, 2-, 4-, and 8-port frame buffer.*

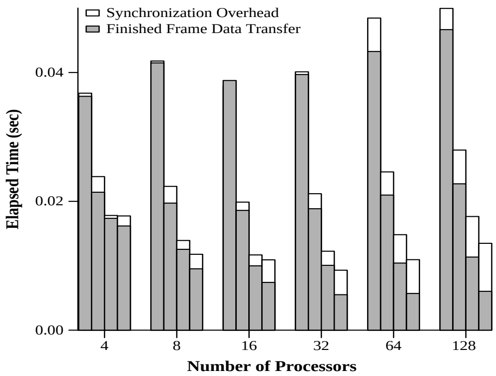

Figure 6: *Synchronization overhead for finished frame data transfer when the frame buffer is configured along X-dimension.*

The Paragon mesh is thus partitioned into two disjoint rectangles: one for compute nodes and the other for frame buffer ports.

The frame buffer ports can be put along the Y-dimension or along the X-dimension. Because of the deterministic routing, the two layouts may not give identical performance.

Figures 3 and 4 give the simulation results for both layouts. The elapsed time of synchronization was defined as the longest time that any node spent in the synchronization process.

As we can see from Figures 3 and 4, the two layouts behave similarly. When the number of the compute nodes is small, the elapsed time increases monotonically with the number of the frame buffer ports because the pixel flushing dominates. When the number of compute nodes increases, the time is almost independent of the number of frame buffer ports. We also notice that when the number of compute nodes is less than the number of frame buffer ports, the elapsed time is greatly increased with the number of frame buffer ports, because of the network contention of the *pixel_flush* messages on their way to frame buffer ports.

	1 port	2 ports	4 ports	8 ports
4 procs	2.73	-0.28	6.37	13.84
8 procs	2.19	16.32	19.20	20.69
16 procs	8.84	8.86	10.31	18.11
32 procs	6.73	16.64	22.44	43.23
64 procs	4.87	8.69	24.15	38.33
128 procs	8.53	15.01	34.91	40.79

Figure 7: *Percentage of time spent on synchronization for finished frames: Frame Buffer Ports on the Y-Dimension. The negative values occur because the synchronization overhead is smaller than the run-to-run variation in time for the finished frame data transfer.*

	1 port	2 ports	4 ports	8 ports
4 procs	1.30	10.17	2.47	8.77
8 procs	0.70	11.63	9.87	19.14
16 procs	-2.27	6.45	14.51	32.07
32 procs	1.05	10.99	17.88	40.82
64 procs	10.70	14.63	29.71	47.91
128 procs	6.59	18.76	35.68	55.27

Figure 8: *Percentage of time spent on synchronization for finished frames: Frame Buffer Ports on the X-Dimension.*

To measure the synchronization overhead, we start with a frame evenly distributed among the compute nodes. Each compute node sends its portion of the pixel data to the frame buffer and then performs a global synchronization for the next frame. This will give a pessimistic estimate of the fraction of time spent in synchronization, since we haven't taken into account any computation time required to generate the pixel data.

Figures 5 and 6 show the results for finished frame data transfer for the two layouts. The synchronization overhead generally increases when the number of compute nodes increases with the number of frame buffer ports fixed, because of the time needed for the global operations between compute nodes. When the frame buffer has more than 4 ports, the synchronization overhead is likely to increase with the number of frame buffer ports, because the time for data transfer decreases and the time for synchronization increases with the number of frame buffer ports. The X-dimension layout requires less synchronization time than the other layout for the same number of compute nodes because the synchronization algorithm starts pixel flushing in the X-dimension first and there is less conflict when the frame buffer layout is laid on along the X-dimension.

Figures 7 and 8 give the percentage figures for synchronization overhead corresponding to Figures 5 and 6. When the number of frame buffer ports and compute nodes is increased, less time is spent on the finished frame data transfer, and more is spent on synchronization overhead. Thus the worst cases happen when the number of frame buffer ports and compute nodes is large. This is not really a problem because both data transfer and synchronization are easily fast enough to display 72 frames per second.

Some of the numbers in Figures 7 and 8 are negative, which appears to indicate a negative cost for synchronization. This is accounted for by the run-to-run variance we observed; the negative values are less than zero by a statistically insignificant amount.

When we applied the synchronization algorithm to the process of redistribution and software Z-buffering for several scenes [20], the overhead was invisible in almost all configurations. We can deduce that the synchronization algorithm would not cause network congestion in real rendering applications.

Performance Summary

In summary, synchronization overhead generally increases as the number of processors increases, because of the cost of synchronizing the compute nodes with each other. Overhead also increases as the number of frame buffer ports increases, because more flushing messages are required. Even for the largest configuration we simulated—128 compute processors with an eight-port frame buffer—the synchronization overhead was low: when displaying 30 frames per second, synchronization overhead would account for 18% of the total time. Our most important performance result is that synchronization overhead for our algorithm is acceptable for a wide range of machine configurations.

6 Conclusion

We have shown that global synchronization of compute nodes with a multi-ported frame buffer can be effectively handled by software with minimal hardware support. The only requirements on the hardware are that it keep receiving network messages, and that it discard flush packets.

Our synchronization algorithm is very efficient; the overhead of a synchronization is quite small compared to the frame updates. Our algorithm is also safe: there is no way an errant compute node can deadlock the frame buffer.

Although our particular synchronization algorithm is dependent on the network topology, our idea can be applied to many other networks, because deterministic wormhole routing is widely used on multicomputers.

7 Acknowledgements

This work was performed by using the Intel Paragon at Caltech. We would like to thank Paul Messina, Roy Williams, and Thanh Phung of the CalTech Concurrent Supercomputing Facilities.

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 Research Council and Intel Scalable Systems Division. Felten is supported in part by an NSF National Young Investigator Award.

References

[1] Kurt Akeley. RealityEngine Graphics. In *Proceedings of SIGGRAPH '93.* (Anaheim, California, August 1-6, 1993), pp. 109-116.

[2] Robert Benner. Parallel Graphics Algorithms on a 1024-Processor Hypercube. In *The Fourth Conference on Hypercubes, Concurrent Computers, and Applications.* (Monterey, California, March 6-8, 1989), pp. 133-140.

[3] Matthias Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathan Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputers. In *Proceedings of the 21st Annual International Symposium on Computer Architecture.* (Chicago, Illinois, April 18-21, 1994), pp. 142-153.

[4] Avaika Networks Corporation. *HiPPI Frame Buffer.* Data Sheet, 1994.

[5] Brooktree Corporation. *RAMDACs–Workstation Graphics.* Section 5., Graphics and Imaging Product Databook, 1993.

[6] Intel Corporation. *Paragon XP/S Product Overview.* Intel Corporation, 1991.

[7] Thinking Machines Corporation. *CM5 Technical Summary.* Thinking Machines Corporation, 1993.

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

[9] Thomas Crockett. Design Considerations for Parallel Graphics Libraries. In *Proceedings of Intel Supercomputer Users Group.* (San Diego, California, June 26-29, 1994), pp. 3-14.

[10] Michael Deering and Scott Nelson. Leo: A System for Cost Effective 3D Shaded Graphics. In *Proceedings of SIGGRAPH '93.* (Anaheim, California, August 1-6, 1993), pp. 101-108.

[11] David Ellsworth. A New Algorithm for Interactive Graphics on Multicomputers. *Computer Graphics and Applications*, July 1994, pp. 33-40.

[12] Chandlee Harrell and Farhad Fouladi. Graphics Rendering Architecture for a High Performance Desktop Workstation. In *Proceedings of SIGGRAPH '93.* (Anaheim, California, August 1-6, 1993), pp. 93-100.

[13] CRAY Research Inc. *T3D System Architecture Overview.* 1993.

[14] Vineet Kumar. A Host Interface Architecture for HiPPI. In *Scalable High Performance Computing Conference.* (Knoxville, Tennessee, May 23-25, 1994), pp. 142-149.

[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] Eric Lawrence McDonald. A Video Controller and Distributed Frame Buffer for the J-Mchine. Master's thesis, MIT, 1995.

[17] Meiko. *CS-2 Product Description*. Meiko Computing Surface, 1993.

[18] S. Molnar and H. Fuchs. *Advanced Raster Graphics Architecture*. Chapter 18, Computer Graphics and Practice, Second Edition. Addison-Wesley Publishing Company, Inc., 1990.

[19] Steven Monlnar, J. Eyles, and J. Poulton. Pixelflow: High-Speed Rendering Using Image Composition. In *Proceedings of SIGGRAPH '92.* (Chicago, Illinois, July 26-31, 1992), pp. 231-240.

[20] 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.

[21] C. B. Stunkel, D. G. Shea, B. Abali, M. M. Denneau, P. H. Hochschild, D. J. Joseph, B. J. Nathanson, M. Tsao, and P. R. Varker. Architecture and Implementation of Vulcan. In *Proceedings of the 8th International Parallel Processing Symposium.* (Cancún, Mexico, April 26-29, 1994), pp. 268-274.

[22] 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.

[23] Scott Whitman. *Multiprocessor Methods for Computer Graphics Rendering*. AK Peters, Wellesley, 1992.