Fast NIC-Based Barrier over Myrinet/GM[*]

Darius Buntinas   Dhabaleswar K. Panda   P. Sadayappan
Network-Based Computing Laboratory
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210
{buntinas, panda, saday}@cis.ohio-state.edu

Abstract

An efficient barrier implementation is desirable on parallel systems to obtain good parallel speedup and to support finer-grained computation. Some modern Network Interface Cards (NICs) have programmable processors which can be used to provide support for collective communications such as barrier. In this paper, we utilize such a programmable NIC to provide an efficient barrier synchronization operation. This paper describes the design, implementation and evaluation of a NIC-based barrier operation as an addition to Myricom's GM message passing system. Our NIC-based barrier implementation achieved a barrier latency of 102.14μs for 16 nodes which is a 1.78 factor of improvement over the host-based barrier using the same algorithm for LANai 4.3 NIC cards. Using LANai 7.2 cards, which has a faster processor, we achieved a 1.83 factor of improvement for eight nodes. Our NIC-based barrier operation promises scalable fine-grained parallel computation over clusters of workstations. To the best of our knowledge, this is the first NIC-level barrier implementation on a cluster with Myrinet/GM.

1. Introduction

Barrier synchronization is a common operation in parallel and distributed systems. An efficient implementation is important because while processors are waiting on a barrier, generally, no computation can be performed, which impacts parallel speedup. The efficiency of barrier operations also affects the granularity of a parallel computation. If the barrier latency is high, then the granularity must also be high. With a lower latency barrier operation finer-grained computation can be supported. So it is important to minimize the amount of time spent waiting on the barrier. Some modern *Network Interface Cards* (NICs) have programmable processors which can be used to provide support for collective communications, such as barrier. We utilize such a programmable NIC to provide an efficient barrier synchronization operation.

Earlier generation SMP systems and MPP systems, such as the Cray T3E and CM-5, had special hardware to perform barriers. Today, parallel systems are moving into clusters built from commodity workstations and networks, so it is difficult to provide barrier synchronization hardware. Dietz[5] had proposed providing hardware barrier over a separate network for clusters of workstations. Such approach requires two networks and may not be cost effective.

Most current clusters use software barriers based on *host-based* point-to-point communication. With *host-based* communication, each message is initiated by the host, passed to the NIC, then to the NIC on the receiving node and finally to the receiving host. The one way latency of such a host-based message may be as high as 30μs. Depending on the algorithm a software barrier would take $\log_2 N$ (e.g., a pairwise-exchange algorithm as used in MPICH[6]) to $2\log_2 N$ (e.g., a gather-and-broadcast algorithm as described in [9]) steps, where N is the number of participating processors. So a barrier across 16 processors would take 120 to 240μs per barrier. This provides high overhead for a barrier and does not lead to scalable or fine grained parallel implementations.

Networks for modern clusters use programmable NICs to improve the communication performance. In a barrier operation, often the reception of one message triggers the sending of another message. By not requiring the message to be transferred to the host, only to then have another message transferred back to the NIC again, we can improve the responsiveness of the barrier. This raises the challenge of whether these programmable processors can be used to support *firmware-level*, or *NIC-level*, barrier and increase the performance of barrier operations at the higher layers.

In this paper, we take on this challenge. This paper investigates the design issues of such an implementation, such as being able to handle multiple concurrent barriers with different processes which use the same NIC, being able to handle multiple consecutive barriers, assuring reliable, in-order delivery of the barrier messages, and initialization of barrier state at the NIC. The implementation of a NIC-based barrier as an addition to Myricom's GM message passing subsystem is also described. Our NIC-based barrier implementation achieved a barrier latency of 102.14μs for 16 processes which is a 1.78 factor of improvement over the host-based barrier for the same algorithm using LANai 4.3 cards. This factor of improvement is expected to increase with the size of the system and with the speed of the NIC processor. Using LANai 7.2 cards, which has a faster processor, we achieved a 1.83 factor of improvement for just eight processes. We expect that the factor of improvement will also increase if an additional programming layer, such as MPI, is added over GM because of the additional overhead the layer adds to each message sent or received. Another fea-

[*]This research is supported in part by an NSF Career Award MIP-9502294, NSF Grants CCR-9704512 and EIA-9986052, an Ameritech Faculty Fellow Award, and grants from the Ohio Board of Regents.

ture of our NIC-based barrier implementation is better utilization of the host processor. Because the barrier algorithm is performed at the NIC, the processor is free to perform computation while polling for the barrier to complete. This is known as a *fuzzy barrier*[7]. Our NIC-based barrier operation promises scalable fine-grained parallel computation over clusters of workstations.

Section 2 describes the basic idea of NIC-based barriers. We describe the design issues involved in implementing NIC-based barrier in Section 3. The implementation details are discussed in Section 4. The details of the barrier algorithms used are described in Section 5 followed by an evaluation of our implementation in Section 6. Related work is discussed in Section 7. Finally, we conclude and discuss our work in Section 8.

2. NIC-based barrier and performance benefits

2.1. Basic idea

The basic idea of the NIC-based barrier is to have the host tell the NIC to initiate a barrier operation and have the NIC notify the host when it has completed the barrier. Figure 1 shows block diagrams comparing host-based barrier to NIC-based barrier. The diagrams show barrier operations, where processes at nodes 0 and 1 exchange messages at the same time as processes at nodes 2 and 3 exchange messages, after which the processes at nodes 0 and 3 exchange messages at the same time as the processes at nodes 1 and 2 exchange messages. The diagram on the left in Figure 1 shows a host-based barrier. In the host-based barrier, in order for a message to be sent the host transfers the message to the NIC which transmits it on the network to the receiving NIC. The receiving NIC receives the message and transfers it to the host. Once the host receives the message, it can initiate sending a message to the next host.

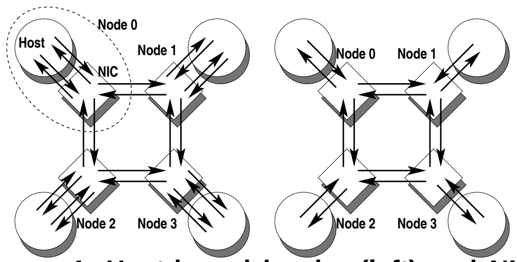

Figure 1. Host-based barrier (left) and NIC-based barrier (right)

By basing the barrier operation at the NIC, rather than at the host, the intermediate messages need not be transferred between the host and the NIC. The diagram on the right in Figure 1 shows a NIC-based barrier. In the NIC-based barrier model, the host sends a message to the NIC to initiate the barrier operation and waits for notification from the NIC that the barrier has completed. The barrier messages are then exchanged between NICs and need not be transferred to the host. As soon as a NIC receives a barrier message, the message to the next process can be sent directly.

2.2. Estimated performance improvement

In this section we estimate the performance improvement of using a NIC-based barrier over using a host-based barrier. This estimate is based on a pairwise-exchange algorithm similar to the one used in MPICH[6]. To perform a barrier with N processes using this algorithm, each process exchanges messages with $log_2 N$ other processes (This algorithm is described in more detail in Section 5.). Figure 2 compares the latency of a host-based barrier with a NIC-based barrier for eight processes. In these diagrams it takes three message exchanges per process to complete the barrier. Each timing diagram shows the breakdown of a barrier operation at a single node. For simplicity, we assume that each node has only one process and that all processes start the barrier at the same time, so the timing diagrams for all eight nodes would look the same. We also assume that the NICs have separate receive and transmit channels[1] to the network, so that one message can be received while another is being transmitted. In these diagrams, $Send$ corresponds to the time from when the host initiates the send until the NIC detects it. $SDMA$ is the time it takes for the NIC to transfer the data for the message from the host memory to the NIC transmit buffer. $Xmit$ corresponds to the time for the NIC to transmit the message on the network. We assume that the network is wormhole routed. Thus the time between when the transmit starts at the sender and when the receive starts at the receiver is small. This time is represented as $Network$ in the diagrams. $Recv$ represents the time for a message to be received by the NIC. The time to transfer a message from the NIC to the host is represented as $RDMA$. Finally, $HRecv$ corresponds to the time it takes the host to process the message once it has been transferred from the NIC.

Figure 2a shows the barrier latency for a host-based barrier. After the host transfers the message for the first destination to the NIC, the NIC starts transmitting it. The NIC will start receiving a message after a delay of $Network$ once the message has started being transmitted. Since we assume that the barriers started at the same time on all nodes, the NIC will receive a message sent to it after a delay of $Network$ after it has started transmitting its message. The diagram shows these transmit and receive events occurring concurrently. Once the message has been received, the NIC transfers the message to the host which processes it. The host then initiates sending a message to the second destination and the process is repeated. After the process is repeated again for the third destination, the barrier has completed.

Figure 2b shows the latency for a NIC-based barrier. Here, the host transfers a message to the NIC to initiate the barrier operation. The NIC starts transmitting the message to the first destination. As before, the NIC starts receiving a message from the corresponding node after a delay of $Network$. After the message has been received by the NIC, the NIC starts transmitting the message for the second destination. Again the message from the corresponding node is received while the second message is being transmitted and, similarly, for the third message. After the third message has been received, the NIC transfers a notification to the host. Once the host processes the notification, the barrier has completed.

From these diagrams, we can see that the latency for an eight node host-based barrier is $3 \times (Send + SDMA + Network + Recv + RDMA + HRecv)$, while the latency for a NIC-based barrier is only $Send + 3 \times (Network + Recv) + RDMA + HRecv$. More generally, for an N process system, the host-based barrier latency would be:

[1]Current Myrinet NICs support this feature.

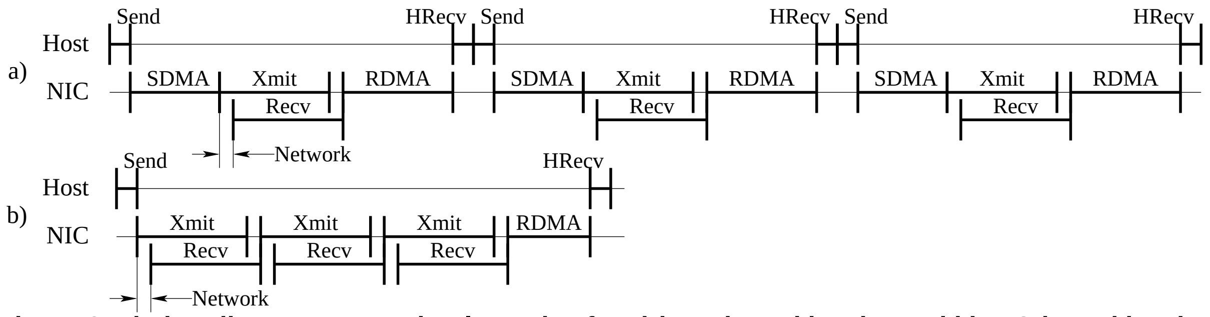

Figure 2. Timing diagram comparing latencies for a) host-based barrier and b) NIC-based barrier

$$T^{Host}_{Barrier} = \log_2 N \times \begin{array}{l}(Send + SDMA + Network \\ \quad + Recv + RDMA + HRecv)\end{array} \tag{1}$$

And for the NIC-based barrier it would be:

$$T^{NIC}_{Barrier} = \begin{array}{l} Send + \log_2 N \times (Network + Recv) \\ + RDMA + HRecv \end{array} \tag{2}$$

The factor of improvement of the NIC-based barrier over the host-based barrier is given by:

$$\begin{aligned} \text{Factor of Improvement} &= \frac{T^{Host}_{Barrier}}{T^{NIC}_{Barrier}} \\ &= \frac{\log_2 N \times \begin{array}{l}(Send + SDMA + Network \\ \quad + Recv + RDMA + HRecv)\end{array}}{\begin{array}{l}Send + \log_2 N \times (Network + Recv) \\ + RDMA + HRecv\end{array}} \end{aligned} \tag{3}$$

From Equation 3 we can predict that as the host send overhead increases, say from the addition of another programming layer such as MPI, the factor of improvement will increase. The factor of improvement will also increase as the number of nodes increases and as the network performance increases.

3. Design issues

There are several major issues in designing a NIC-based barrier operation. One issue is how to handle unexpected barrier messages that are received by a node which hasn't initiated a barrier. Another issue is initializing barrier data structures at the NIC when a process opens an endpoint, and similarly cleaning up data structures after an endpoint is closed. Reliability and in-order delivery of barrier messages must also be addressed. The last issue is to handle multiple concurrent barriers at the same NIC.

In this section we first describe our system model and then identify these design issues and present some solutions. In the next section, we identify the solutions we have implemented.

System model: A system consists of a collection of *nodes*. Each node consists of one or more *programmable NICs* and one or more *host processors*. The nodes are connected, through the NICs, by a *communication network*. *Processes* run on a host processor and can communicate with each using the NICs by using an abstraction called a *communication endpoint*. A process can allocate one or more such endpoints. An endpoint is associated with a particular NIC at the node, so that messages sent or received by the process are handled by that NIC. Messages are sent from one endpoint to another. Similarly, a barrier operation is associated with endpoints. A barrier operation synchronizes the processes which are attached to the specified endpoints.

3.1. Handling unexpected barrier messages

If all processes start the barrier operation at the same time, then keeping track of which messages were received would be easy, because the barrier messages would be received in the same order as they are expected. In practice, however, processes may initiate barrier operations in an asynchronous manner. Thus, a NIC may receive barrier messages before the NIC is ready for them and possibly even before the host has initiated the barrier. To make matters worse, there may be multiple consecutive barriers with different subsets of processes, so the NIC may receive barrier messages from future barriers. In the worst case, a process might perform multiple consecutive two-process barriers, one with every other process in the system. Then, if that process is slower than the others and the other processes reach their barriers first, the associated NIC would receive $N - 1$ unexpected barrier messages, where N is the number of processes in the system. So the NIC must be prepared to receive a barrier message from any process on any node in any order at any time. However, once a process initiates a barrier operation and is waiting for it to complete, it will not initiate another one until that barrier completes. So the NIC can receive *at most* one unexpected message from every other process on every node.

One method of handling unexpected barrier messages is to accept and record the reception of every unexpected barrier message in a *unexpected barrier message record*. For instance, a flag could be allocated for every possible communication endpoint on every possible node. When a barrier message is received, the flag corresponding to the endpoint that sent the message would be set. Then, when the NIC is ready to receive a barrier message from a particular endpoint, the NIC would simply have to check the corresponding flag to see if that message has already been received. The flags are then reset after they are read to allow another unexpected message to be recorded from that same endpoint. Representing the flags as a bit array is the most space efficient representation and also setting, resetting or checking the flags would take constant time. Because GM allows only eight endpoints per NIC, this overhead is only one byte per connection.

3.2. Initialization and cleanup

Another problem is how to initialize the data structures recording these messages, and how to clean them up after a partially completed barrier is aborted. For example, let's say process A on node 0 initiates a barrier with process B

on node 1, and that process B dies before a barrier message is received. When the NIC at node 1 receives the message it will record it as an unexpected message, possibly destined to a process that hasn't started yet. Now, say, process A is killed, and two new processes A' and B' are started on nodes 0 and 1 respectively, and reuse the same endpoints as the previous processes. If process B' initiates a barrier, the NIC will see that it has received a message from node 0 from the same endpoint that process A' is using and will assume that it has received a barrier message from A' even though that message was sent by A. It is possible now for B' to complete the barrier before A' starts the barrier.

Before we discuss possible solutions, we need to make certain assumptions about the state of the system when a process is started. Processes that will communicate may not all start at the same time. Because of this, it is possible that when a node sends a message to a remote endpoint the remote process to which the message was sent may not have started yet. This may be unavoidable, but is usually benign. In the worst case this message would have to be retransmitted. It is also possible that a different process is still using that endpoint. This has more serious implications. Messages may be sent between the nodes, each one thinking that the message has been sent or received by a different process. To avoid this possibility, it is sufficient to require that if a process p will communicate with a process q through a remote endpoint e, then when process p is started the endpoint e cannot be owned by a process other than q. Furthermore, no old messages can be in the communication channels between p and q. While this may seem like a rather strict requirement, this usually happens in practice. For instance when a parallel program is started on several machines, if a resource, such as an endpoint, is not available to one process, the whole program is aborted and restarted once the resource is available. A way around this requirement is to include a mechanism to distinguish messages of one parallel program from another.

One naive solution is to simply clear the unexpected barrier message record of all messages destined for a particular endpoint when that endpoint is opened. This may solve the problem mentioned above, but that does not allow barrier messages to be received for a process that hasn't started, or opened an endpoint. This may happen, if, for instance, the first action of a program is to do a barrier in order to make sure all its peers have started.

A better solution is to have the NIC reject any barrier messages for a closed endpoint. Then, the sender of the barrier message will resend the message, but only if the endpoint that initiated the barrier has not closed since the message was sent. Then, with the above requirement about the state of the system when a process starts, we know that once a process opens an endpoint, and starts accepting barrier messages, no old messages will be received. While this method may increase the latency of the barrier, this will only be the case if a participating endpoint has not been opened yet.

Another solution is to record received barrier messages for a closed port, but then reject those messages once the endpoint is opened. Then, the NICs which sent those messages will then resend them, but only if the endpoints which initiated the barriers have not closed since the original message was sent. This has the same drawbacks as the previous solution, except, it would require only one retransmission, rather than an unbounded number. Thus we adopt this approach in our implementation.

3.3. Reliability and in-order delivery

A lost barrier message could hang processes indefinitely. Therefore it is important to provide a mechanism to deliver barrier messages reliably. Related to this issue is the guarantee of the order in which the messages will be delivered. There are two design options with regard to the delivery order of barrier messages relative to non-barrier messages. If barrier messages are guaranteed to be delivered in-order with regard to non-barrier messages, then messages sent before a barrier is initiated by the sending process will be received before the barrier completes at the receiving process. Similarly messages sent after a barrier completes on the sending process, will not be delivered before the barrier completes on the receiving process. This will not be true if the relative order of barrier messages and non-barrier messages is not preserved. Instead, the order of messages will be maintained separately among barrier messages and among non-barrier messages.

One method of preserving the relative order between barrier messages and non-barrier messages is to use the same mechanism to provide in-order and reliable delivery for both types of messages. This is the approach we adopt in our implementation.

3.4. Multiple concurrent barriers

Because barriers using message passing do not depend on holding shared resources, independent barriers on separate nodes can occur concurrently. However, if a NIC can be used by more than one process, then the NIC-based barrier mechanism must be designed to allow multiple processes to initiate barrier operations concurrently. If two processes using the same NIC are participating in the same barrier, it may be possible to provide an optimization, where a barrier message need not actually be sent, but rather just have a flag set to indicate that it has been received. Our initial implementation allows multiple instances of barriers to exist concurrently on the same NIC. We intend to incorporate the above optimization in our final implementation.

4. Implementation

In this section we describe our implementation of a NIC-based barrier as an addition to Myricom's message passing system, GM[10], version 1.2.3. First, we describe GM, then identify our design choices and describe the implementation details of each.

4.1. Overview of GM

GM consists of a driver, a library and a Myrinet control program (MCP). The driver loads the MCP on to the NIC when it is loaded. During the execution of a program the driver is used mainly for opening *ports*, pinning and unpinning memory, and to put a process to sleep or to wake a process for blocking functions. A *port* is a data structure through which a process can communicate with the NIC while bypassing the operating system. A port also serves as a communication endpoint. Once a port is opened, the process can communicate with the NIC, bypassing the OS and avoiding system call overhead. In GM version 1.2.3, each NIC can support a maximum of eight ports, some of which are reserved.

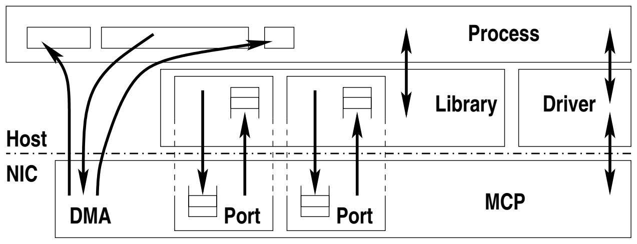

Figure 3. Block diagram showing the components of GM.

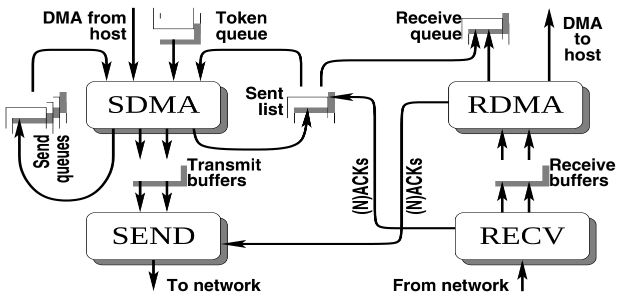

Figure 4. Block diagram of the components of the MCP.

At the host level GM is connectionless, but provides reliability by maintaining reliable connections between NICs of different nodes. Flow control is used between the NIC and the host to avoid buffer overflows. To provide this reliability GM uses the concept of tokens. When a process opens a port, it has a certain number of *send tokens* and *receive tokens*. Each send token corresponds to a send event. For sending a message the process fills-in a send token describing the send event and queues it on the send queue. Once the NIC has completed the event, and has freed the resources corresponding to that event, the send token is returned to the process.

In order to receive a message, the process must allocate a buffer into which the message will be received and pass a receive token describing the buffer to the NIC. Once the NIC has DMAed the data into the buffer, the receive token is returned to the process. The process must poll to detect returned receive tokens. Messages may only be sent from and received into buffers which are pinned in memory. Memory is pinned using special functions supplied by GM.

Figure 3 is a block diagram of GM where a process has two ports through which send tokens and receive tokens are transferred to and from the MCP without going through the driver. The figure also shows DMA operations which transfer data directly to and from the process' memory.

The NIC has a data structure for each local port, which contains the send and receive queues. The NIC also has data structures each corresponding to a connection to one node in the system. The connection structure contains information about the state of the connection and the port from which to send next.

Figure 4 shows a block diagram of the MCP. The MCP consists of four state machines called SDMA, SEND, RECV and RDMA. The SDMA state machine polls for new send tokens and queues them on the queue for the appropriate connection. The SDMA state machine is also responsible for initiating a DMA to transfer data from the host memory to the transmit buffers in the NIC and to prepare the packet for transmission. Once the packet is ready to be transmitted, the send token is moved to the sent list. The SEND state machine is responsible for transmitting packets which were prepared by the SDMA state machine and any acknowledgment packets which may be pending. The RECV state machine receives incoming packets into receive buffers and handles acknowledgment and negative acknowledgment packets. When the RECV state machine receives an acknowledgment it removes the token associated with that send from the sent list and passes it back to the host. The RDMA state machine prepares acknowledgment and negative acknowledgment packets and DMAs the data to the host buffer corresponding to an appropriate receive token. The RDMA state machine also adds receive tokens in the receive queue to notify the process that the receive has completed.

4.2. Multiple concurrent barriers

In order for our implementation to support multiple concurrent barriers, we must allow multiple barriers to exist on the same NIC. Since each port may participate in a barrier independently, the NIC must keep the state of each barrier separate. We do this by putting the state information in the *send token* and keeping a pointer in the port data structure to this send token. This way, when a barrier packet is received, the RDMA state machine can access the state of the barrier by simply dereferencing the pointer. The token will store a list of the port ids and node ids with which barrier messages will be exchanged, as well as an index, *node index*, into this list to indicate which is the next node to receive from or to send to.

4.3. Handling unexpected barrier messages

To handle unexpected barrier messages, we used an unexpected barrier message record. Because there is already a data structure per connection, and each connection has at most eight ports, the record was implemented as a bit array for each connection. When an unexpected barrier message is received, the bit corresponding to the source port and connection is set. The NIC can then check for received messages by checking the appropriate bit. After a bit is checked, the bit is cleared.

4.4. Reliability and in-order delivery, and initialization and cleanup

The difficulty in providing reliability is that in GM, when a reliable packet is transmitted, the send token is added to the sent token list. Only once the packet is acknowledged is the send token de-queued and returned to the process. If a packet is negatively acknowledged, all packets sent after that packet must be resent. This is done by pushing the contents of the sent list back on the send queue.

In our current implementation, which uses unreliable barrier packets, once a barrier packet has been transmitted, it is de-queued then re-queued in the queue for the next destination. Now, since our barrier scheme uses only one send token, and the token can potentially be used to send to multiple destinations, if two or more barrier packets need to be retransmitted, the same token would have to be queued twice.

One solution is to have the barrier event use one token for every destination. Then the NIC will queue a send token separately for each packet sent. Another solution is to provide a separate retransmission mechanism just for barrier messages. Under this solution, the barrier messages will be acknowledged separately and will have separate sequence numbers. This will require separate acknowledgment packet types and structures to keep track of sequence numbers, as well as routines to resend the barrier messages. As discussed in Section 3.3, barrier messages and non-barrier messages will not necessarily be received in the same order that they were sent.

We have implemented some of the components necessary to provide reliability. We intend to complete the implementation soon. As described in Section 3.2, a barrier message retransmission mechanism is necessary for handling barrier messages which are sent to ports which are closed. Since we have not finished implementing this mechanism, when performing our benchmark programs, we must ensure that any barrier that is initiated completes normally (i.e., no participating port is closed during a barrier operation).

5. Barrier algorithm

In this section we describe two algorithms for performing barriers and how we implemented them on the NIC. The first is a gather-and-broadcast algorithm (GB) [9], and the second is a pairwise exchange algorithm (PE) that is used in MPICH [6].

5.1. Algorithm descriptions

The GB algorithm constructs a fixed dimensional tree of the nodes participating in the barrier. The algorithm then proceeds in two phases: gather and broadcast. In the gather phase, each node, except the root, waits to receive a gather message from each child, then sends a gather message to its parent. The root waits for a gather message from all its children, then sends a broadcast message to each of them and exits the barrier. As each other node receives the broadcast message, it sends the broadcast to each child then also exits the barrier. We would expect that the dimension of the tree would impact the performance of the barrier. Thus, depending on the parameters of the communication subsystem and the size of the barrier one could use a different dimension tree to get the best performance.

The PE algorithm works recursively. The nodes are paired up and each node does a send followed by a receive with its partner. These nodes now form a group. Next, each group is paired with another group, and every node in one group performs a send followed by a receive with one node from the other group, then those groups are then merged. This pairing, exchanging messages and merging is repeated until only one group is left. The barrier is then finished. Each node will perform $log_2 N$ sends and receives, where N is the number of nodes performing barrier.

NIC processors are typically much slower than the host processors (e.g., Myrinet NIC processor speeds range from 33MHz to 132MHz while processor speeds for a typical host processor might range from 300MHz to 1GHz). For this reason it may be more efficient to have the host processor perform some parts of the algorithm.

An issue here is the construction of the tree for the GB algorithm. One alternative is to pass the list of participating nodes to the NIC and have the NIC construct the tree. However, the tree construction is a relatively computationally intensive task which can easily be computed at the host. The host at a particular node needs to inform the NIC only of the children and parent of the node, rather than all the nodes in the barrier. This also reduces the amount of data that has to be transferred to the NIC. Similarly, for the PE algorithm, the task of determining the pairings can be done either at the NIC, or at the host. Again, this can be done much quicker at the host and also the whole list of nodes need not be transferred to the NIC.

5.2. Algorithm implementation

Both algorithms were implemented on the NIC. We will first describe the changes to the GM API, then describe the implementation details at the NIC.

We added two new functions to the GM API to support NIC-based barriers: `gm_provide_barrier_buffer()` and `gm_barrier_send_with_callback()`. Before initiating a barrier the host calls `gm_provide_barrier_buffer()` to provide the NIC with a receive token. To perform a barrier, the host must compute the barrier tree (for the GB algorithm), or the list of processes with which to exchange messages with (for the PE algorithm). Then, the process then calls `gm_barrier_send_with_callback()`. For the GB algorithm, the process specifies, in the function call, the parent node id and port id, and the node and port ids of each of the children. For the PE algorithm, the process specifies the list of nodes and port numbers with which to exchange messages. Next, the host polls `gm_receive()` until it receives a `GM_BARRIER_COMPLETED_EVENT`. The reception of this event indicates the completion of the barrier. Because we separate the barrier initiation from the polling of the barrier completion, a *fuzzy barrier* [7] can be performed, where some bounded computation can be done while polling for the barrier completion.

In the GB algorithm, the `gm_barrier_send_with_callback()` function creates a send token with the node list and passes it to the token queue on the NIC. There is a separate packet type for each phase. When the SDMA state machine receives the barrier send token from the process, it first sets the send token pointer in the port structure to this send token, then it checks if it received a barrier *gather* packet from each of its children. If so, it clears the bits for the received packets, and queues the send token for the parent node. If it has not received a gather packet from each child, then it must wait until all have been received.

When a barrier gather packet is received, the packet is recorded, then, if the send token pointer in the port data structure is non-zero, the RDMA state machine checks to see if gather packets have been received from all the children, and, if so, the send token is prepared to send a barrier *gather* packet with the parent's port id and is queued in the send queue for the parent's node id.

When the root node receives gather messages from each child, or when a child receives a barrier *broadcast* packet, the RDMA state machine sends a receive token to the host indicating that the barrier has completed, and sets the send token pointer in the port data structure to zero. Then the send token is prepared to send a barrier *broadcast* packet to the first child, and is enqueued on the connection to the node of the first child. Once the SDMA state machine has prepared the packet to be transmitted, the send token is updated to be sent to the next child, and it is re-queued. This continues until a broadcast packet has been sent to each child. Then the send token is returned to the port.

In the PE algorithm, the `gm_barrier_send_with_callback()` function creates a send token with the node list and passes it to the token queue on the NIC. When the SDMA state machine receives the barrier send token from the host, it sets the *node index* to point to the first node in the node list, sets the destination node id and port id of the send token to this first node and port, then sets the *send token pointer* in the port data structure to point to the send token to indicate that a barrier has been initiated. Then, after the SDMA state machine prepares the packet to be sent, it checks to see if a barrier packet has been received from that same destination. If it has, it 1) clears the bit for that message, 2) increments the index in the send token to point to the next destination, 3) writes the next destination's port number in the send token, 4) removes the send token from the current queue and 5) queues the send token in the queue for the connection for the next destination. If the expected barrier packet was not yet received, then the send token is simply removed from the queue.

When a barrier packet is received, the RDMA state machine checks if the port that the message is addressed to has received a barrier send token from the host by checking if the pointer to the send token is non-zero. If so, and if this is the expected barrier message, then the send token is updated and enqueued for the next destination. In all other cases, the reception of the message is simply recorded.

Once the packet to the last destination has been sent and the corresponding packet has been received, the barrier is complete. The NIC DMAs a receive token to the host, returns the send token, and sets the send token pointer in the port data structure to zero.

6. Experimental results

We evaluated our implementation on a cluster of 16 dual 300MHz Pentium II machines, each with 128MB of RAM, running RedHat 6.0 with kernel version 2.2.5. The machines are connected by a Myrinet LAN network with LANai 4.3 cards, with 33MHz NIC processors, via a 16 port switch. Eight of these nodes also have LANai 7.2 cards, with 66MHz NIC processors, connected to a Myrinet LAN network via an 8 port switch.

We tested the latency of our NIC-based barrier implementation and compared it to a host-based barrier implementation on GM. We compared the performance for both the GB and PE algorithms. To test the barrier latency, we ran 100,000 barriers consecutively and took the average latency. Tests were performed for 2, 4 and 8 nodes using LANai 4.3 and the LANai 7.2 NICs, and for 16 nodes using LANai 4.3 NICs.

The performance of the GB algorithm on a given system for a given size depends on the dimension of the gather and broadcast tree. In order to find the optimal dimension for the tree, we ran the test for every dimension from 1 to $N-1$, where N is the number of nodes participating in the barrier. The latencies reported in the graphs are the minimum latencies over all dimensions.

Figure 5(a) shows the barrier latencies of NIC-based and host-based barriers for each algorithm using the LANai 4.3 cards. Notice that the NIC-based PE barrier performed better than all other barriers, with a 16-node barrier latency of 102.14μs. Also, the NIC-based GB barrier performed better than either host-based barrier except for the two node barrier. The NIC-based GB barrier performed worse for the two node barrier than the host-based GB barrier because of the overhead of processing the barrier algorithm at the NIC. The 16 node barrier latency of the NIC-based GB barrier is 152.27μs. The host-based PE barrier performed better than the host-based GB barrier.

Figure 5(b) shows the factor of improvement of the NIC-based barrier over the host-based barrier for both algorithms using the LANai 4.3 cards. For a barrier with 16 nodes, the NIC-based PE barrier gave a 1.78 factor of improvement over the host-based PE barrier, while the NIC-based GB barrier gave a 1.46 factor of improvement over the host-based GB barrier.

One possible reason why the factor of improvement for the GB algorithm is not as large as that for the PE algorithm is that in the broadcast phase of the host-based barrier, the messages sent by the host are pipelined through the NIC, i.e., after the host transfers a send event to the NIC, it is free to transfer the next send event while the NIC is processing the first one. So part of the overall send time of one message is overlapped with that of the subsequent message. There is no such overlapping in the PE algorithm because the host must wait to receive a message before sending the next one.

We also ran similar tests using the LANai 7.2 cards. Because we only have eight of these cards, we show the results for up to only eight nodes. Figure 5(c) shows the barrier latencies for NIC-based and host-based barriers for each algorithm using these cards. Notice that the faster NIC processor improved the performance of all implementations. With the faster NICs the NIC-based barrier using the PE algorithm performed a barrier in 49.25μs compared to 90.24μs for the host-based PE barrier for eight nodes.

Figure 5(d) shows the factor of improvement of the NIC-based barrier over the host-based barrier for both algorithms using the LANai 7.2 cards. This shows a 1.83 factor of improvement of the NIC-based barrier over the host-based barrier using the PE algorithm for eight-nodes. This is a greater factor of improvement than we saw for the LANai 4.3 cards for eight nodes which was 1.66.

A more extensive evaluation of our NIC-assisted barrier implementation using MPICH over GM is presented in [4] an [3].

7. Related work

NIC-level support for collective communication has been studied previously. Bhoedjang[1], Verstoep[13] and Buntinas[2] have implemented NIC-supported multicast/broadcast. Kesavan[8] and Sivaram[11] have evaluated different aspects of NIC-supported multicasting. These papers discussed adding functionality to the programmable NIC to support broadcasting/multicasting. However, we are not aware of any work done on providing NIC-based support for barrier synchronization.

Sivaram[12] proposed enhancements to a network switch architecture to support reliable barriers. Dietz[5] has implemented hardware barrier support for workstation clusters through the parallel ports of the workstations of the cluster. However, this scheme requires a separate network.

8. Conclusions and future work

We proposed using a programmable NIC to support a barrier synchronization operation. We then identified issues in implementing the barrier synchronization operation on the NIC, and described our implementation over GM 1.2.3 using two barrier algorithms.

We evaluated our implementation and compared them to host-based barrier operations. When using the pairwise exchange barrier algorithm for both the NIC-based and host-based barriers, we found the NIC-based barrier gave a 1.78

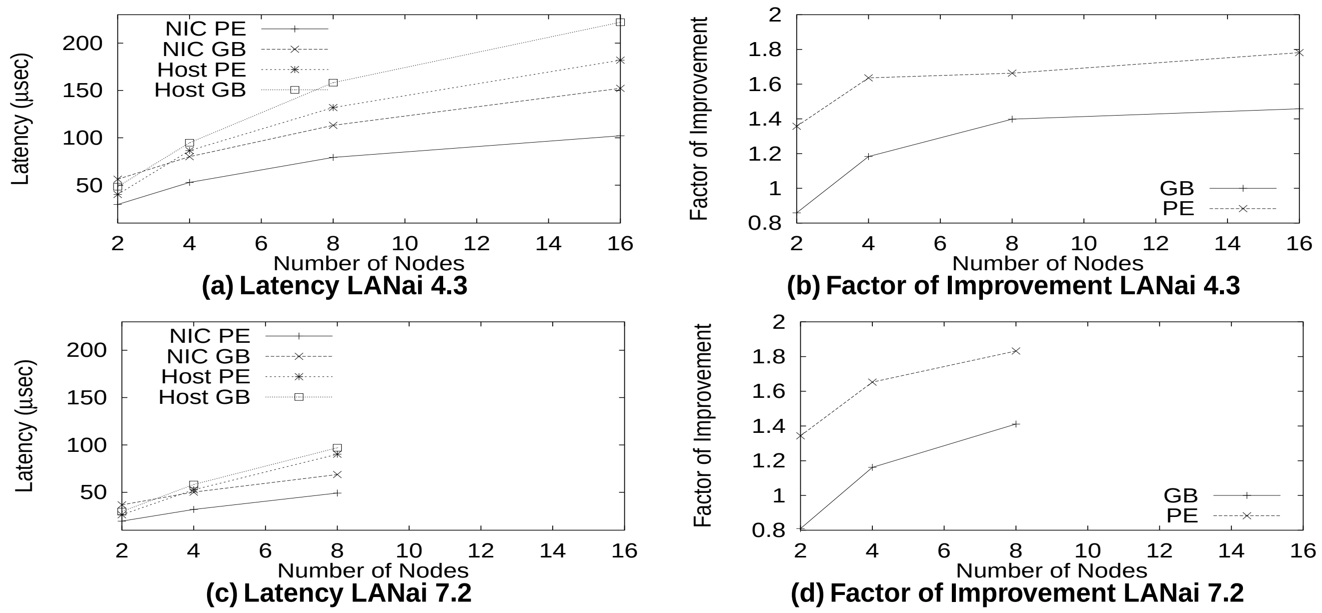

Figure 5. Comparison of NIC-based barrier and host-based barrier for two algorithms (PE and GB) using the LANai 7.2 and LANai 4.3 NICs

factor of improvement. Using the gather and broadcast algorithm in both the NIC-based and host-based, the NIC-based barrier gave us a 1.46 factor of improvement. Because the barrier performance has been poor for clusters in the past, the granularity of parallel computation had to be coarse. Now, with lower latency barrier operations, finer grained computation is feasible.

We intend to study the effects of our NIC-based barrier operation on higher communication layers, such as MPI or Get/Put, and also at the application level. We expect that our NIC-based barrier would show an even greater improvement over host-based barrier with these layers because of the additional latency to individual messages which is added by them.

On a more general level, we intend to investigate whether other collective communication operations, such as reductions or all-to-all broadcast could benefit from similar NIC-level implementations.

Additional information: Additional papers related to this research can be obtained from the following Web pages: Network-Based Computing Laboratory (http://nowlab.cis.ohio-state.edu) and Parallel Architecture and Communication Group (http://www.cis.ohio-state.edu/∼panda/pac.html). If you are interested in using this software, please contact Dr. D. K. Panda at panda@cis.ohio-state.edu.

References

[1] R. A. F. Bhoedjang, T. Ruhl, and H. E. Bal. Efficient Multicast on Myrinet Using Link-Level Flow Control. In *Proceedings of the 27th Int'l Conf. on Parallel Processing (ICPP '98)*, pages 381–390, August 1998.

[2] D. Buntinas, D. K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-Assisted Multidestination Messages. In *Proceedings of Int'l Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC)*, 2000.

[3] D. Buntinas, D. K. Panda, and P. Sadayappan. Fast NIC-based barrier over Myrinet/GM. Technical Report OSU-CISRC-10/00-TR22, The Ohio State University, October 2000.

[4] D. Buntinas, D. K. Panda, and P. Sadayappan. Performance benefits of NIC-based barrier on Myrinet/GM. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS '01*, April 2001.

[5] W. E. Cohen, H. G. Dietz, and J. B. Sponaugle. Dynamic Barrier Architecture for Multi-mode Fine-grain Parallelism using Conventional Processors. In *Int'l Conf. on Parallel Processing*, Aug 1994.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[7] R. Gupta. The Fuzzy Barrier: A Mechanism for the High Speed Synchronization of Processors. In *Proceedings of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, 1989.

[8] R. Kesavan and D. K. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proceedings of Int'l Conf. on Parallel Processing*, pages 370–377, Aug 1997.

[9] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers. Technical Report MSU-CPS-94-35, Michigan State University, 1994.

[10] Myricom. Myricom GM myrinet software and documentation. http://www.myri.com/scs/GM/doc/gm_toc.html, 2000.

[11] R. Sivaram, R. Kesavan, D. K. Panda, and C. B. Stunkel. Where to Provide Support for Efficient Multicasting in Irregular Networks: Network Interface or Switch? In *Proceedings of the 27th Int'l Conf. on Parallel Processing (ICPP '98)*, pages 452–459, August 1998.

[12] R. Sivaram, C. B. Stunkel, and D. K. Panda. A Reliable Hardware Barrier Synchronization Scheme. In *Proceedings of the 11th IEEE Int'l Parallel Processing Symposium*, pages 274–280, April 1997.

[13] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proceedings of the Int'l Conf. on Parallel Processing*, pages III:156–165, Aug 1996.