MPI on U-Net
Class Project CS268

Bernd Pfrommer
Department of Physics
501 Birge Hall
Tel. 642-2635

Abstract

A simple protocol is designed and implemented into the ADI layer of the MPICH message passing library to ensure reliability when running on top of unreliable network interfaces such as U-Net. Our protocol for short messages uses a reliable message stream and lazy acknowledgment to realize low latencies and a small protocol overhead. An implementation for two Pentium Pro 200 nodes connected with 100Mb/s Tulip Fast Ethernet shows that latencies of about 32 μs can be achieved at the ADI layer.

1 Introduction

Within a surprisingly short time, the MPI message passing library emerged to become the library of choice for many high-performance parallel applications. It offers paradigms which are particularly well suited and convenient for scientific computing, and achieves high performance by leaving the burden of buffering mostly to the application.

The MPI library comprises 130 functions, and is often not built directly on top of the network layer, but follows the strategy of the MPICH implementation [6]. There, a portable MPI layer is implemented on top of the Abstract Device Interface (ADI) layer, which contains a set of about 40 machine-dependent point-to-point communication primitives. The ADI layer in turn uses system-specific *reliable* message passing facilities to provide its service.

In this paper, we will address the issues that come up when the ADI layer is built directly on top of U-Net [3][4], which provides an *unreliable* service (see Figure 1). To the best of our knowledge, the present work is the first report of MPI/ADI on top of any unreliable service. We achieve reliability inside the ADI layer with an extremely simple sliding window protocol.

Eventually, a full implementation of ADI and a series of benchmarks on various platforms will be necessary to evaluate the quality of our protocol. Given that the ADI layer provides about 40 functions, this is not within the scope of a class project. As MPI is primarily deployed for high performance computing, we considered it necessary to implement at least part of the ADI layer to assess the overhead incurred. Two 200 MHz Pentium P6 platforms connected with Tulip Fast Ethernet served as the basis for our explorations.

2 Design Decisions

Our protocol and implementation is designed to perform well for a certain hardware configuration and operating mode which we think will be typical for high-performance computing. Applications insensitive to latency, or requiring little communication in the first place, already run well on clusters of workstations communicating via MPI on TCP/IP. We are aiming more at applications that need low latencies and high bandwidths. Those typically require parallel supercomputers with fast custom-made interconnects.

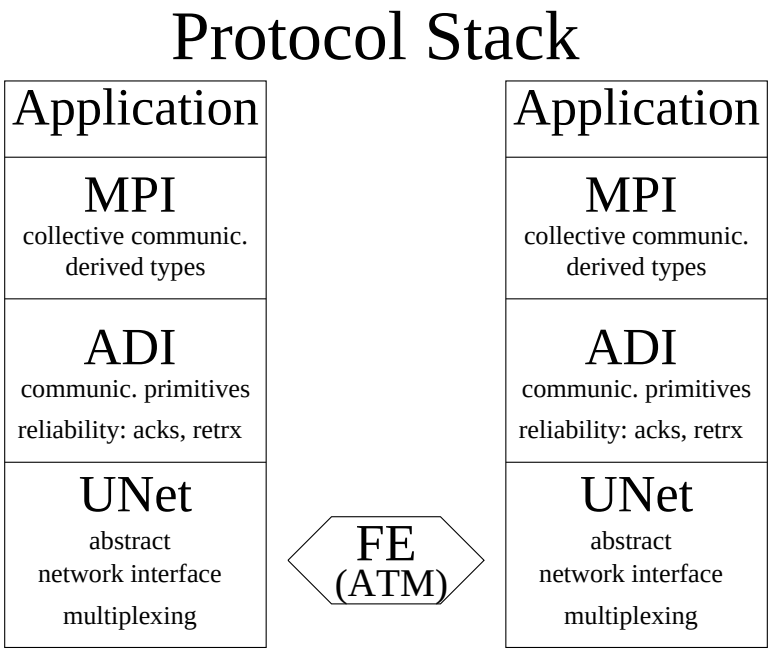

Figure 1: Protocol stack for our MPICH-based implementation of MPI on top of the unreliable U-Net service.

2.1 Operating Mode

Generally, such applications also synchronize often, and are very sensitive to scheduling effects. For this reason, we assume the individual nodes (or multiprocessors) to *be operated in dedicated mode*, and hence perform *busy waiting* (spinning) whenever synchronization is necessary. While most workstation clusters are non-dedicated, the majority of parallel supercomputers maintain their production nodes in dedicated batch mode.

2.2 Network

With respect to the networking hardware, we assume that eventually ATM networks will link the processing units with a very low error rate such that *lost packets are a rare event*. Assuming a bit error rate of 10^{-10} and a packet size of about 1kB, there will be a packet loss at most every 10^6 packets. For example, an additional overhead of say 1μs/packet due to a more sophisticated protocol will add up to a full second of overhead per dropped packet. For this reason, we think that on low error rate networks, minimal protocol overhead is an important design goal. While we make the common case go fast, we must also ensure that the rare case executes correctly. Assuming a correct MPI application, our protocol should handle all failure modes correctly[1].

[1] Aside from theoretical considerations, we simulated packet losses by dropping packets inside the receive function.

3 Startup Phase

This section reports our experience with implementing the MPI startup phase (MPI_Init()) under U-Net on Fast Ethernet. We encountered and overcame several difficulties which we feel are worthwhile mentioning.
Setting up a U-Net channel to a remote host requires knowledge of

1. the local U-Net port
2. the remote host's MAC address
3. the remote host's U-Net port

While the remote host's MAC address is a static parameters, and could be read from a configuration file at startup time, the U-Net port numbers are not known beforehand. Since port numbers are globally shared between all processes on a host, one cannot rely on a certain port number to be available. For this reason, we [2] had to add a port reservation function to the U-Net device. This allowed the following startup mechanism.

- The master queries its MAC address from the U-Net Tulip device, and reserves a set of ports, one for each channel to a slave.
- The master reads the names of the slaves from the host file, and starts up the slaves with rsh, passing its host name as an argument.
- When the slaves come up, they also learn about their MAC address and reserve a set of ports to communicate to all peers.
- The slaves set up a TCP connection to the master and tell it their MAC address and the reserved port numbers.
- Now the master has complete information about MAC addresses and port numbers at each slave, and it sends out the relevant information to the slaves.
- At that point, the master and all slaves know the MAC address and corresponding port numbers for their peers, and they set up the U-Net channels.
- Master and slaves synchronize via another TCP two-way handshake to make sure that all U-Net channels are set up completely before returning from MPID_Init(). This ensures that no communication is attempted before all U-Net channels are set up completely.

One could also design a faster startup procedure, where the slaves set up U-Net channels with the master in a star topology early on, and then use the U-Net channels to communicate the port numbers. We decided to use TCP because first the startup phase is not performance critical, and second we avoid dealing with reliability issues.

[2] The implementation of the reservation mechanism into the U-Net device was performed by Pat Bozeman

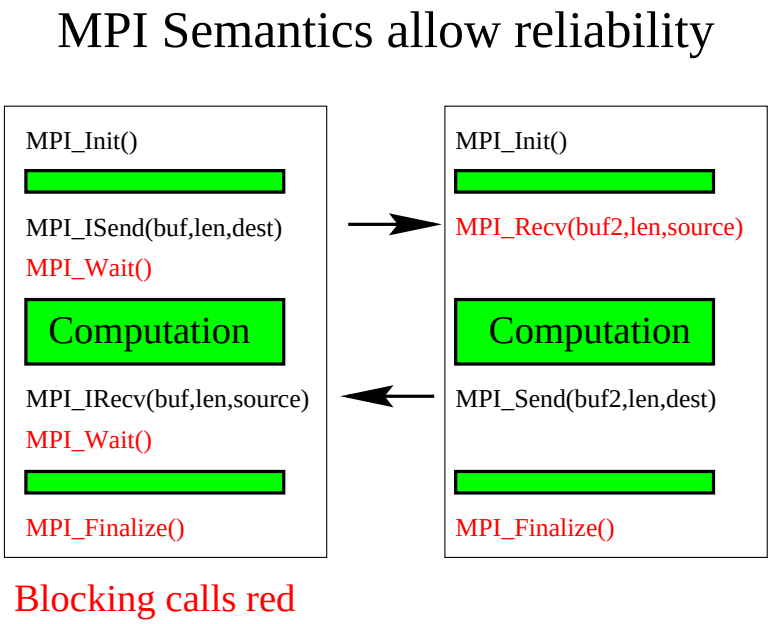

Figure 2: Schematic of an MPI example program (unimportant arguments have been ommitted for simplicity).

4 Reliability Without Context Switch

It is important to observe that the MPI semantics allow for reliable message transfer without relying on a background process or interrupt mechanism. This avoids the performance penalties associated with context switches.

The schematic of an example MPI program is shown in Figure 2. The difficulty with achieving reliability is that the flow of control might not be inside the MPI library routines when a check for reliable delivery is due. For instance, a send call might return immediately, the application goes off to do computation (shown as green boxes in Figure 2), but the packet gets lost and needs to be retransmitted. We observe that *always* the flow of control returns to the MPI layer when it is critical, for instance when there is any kind of blocking operation (e.g. wait, or blocking send or receive). While blocking inside the MPI layer, we make sure that communication progresses to prevent deadlock. If the application does not call any blocking routines, we can still rely on the fact that a correct MPI program *must* call MPI_Finalize() before it exits. At that point, we deal with packet loss, retransmit unacknowledged packets, and acknowledge received packets. If the probability of packet loss is very small, or the application is synchronizing frequently anyway, deferring the reliability enforcement to MPI_Finalize() will hardly degrade performance.

5 Protocol

To achieve reliability without incurring large latencies, our protocol is connection-oriented. Much like in the TCP protocol, a reliable stream is set up between each of the nodes. The cost of connection establishment and release is paid only once in the startup (MPID_Init()) and tear-down

(MPID_End()) phase.

5.1 Description of the Protocol

What follows describes the use of the reliable stream to implement the MPI short messages protocol[3]. When a short message is sent by the application, it is placed into the reliable message stream. A sequence number gets assigned to it, and a queue descriptor is pushed into the U-Net transmission FIFO. The U-Net device immediately transmits the message. On the receiving side, the U-Net device picks the message up and places it into the U-Net receive buffer. When the application on the receiver calls a receiving subroutine, the ADI layer checks for messages in the receive buffer. If it detects an out-of-order or duplicate (retransmitted) packet, it immediately sends out a duplicate acknowledgment[4]. If the received packet is in order, it is not acknowledged, but the receive call returns immediately. The acknowledgments are all piggy backed onto messages, unless they are triggered by retransmissions or out-of-order packets. This lazy acknowledgment strategy will work well for small loss rates, and for applications that exchange a comparable number of messages in a rather synchronous fashion. It will not work well for a producer-consumer type of application. In this case, one could configure MPI with an environment variable to follow an eager acknowledgment protocol.

For simplicity, only a single retransmission timer is maintained for the packet at the left sender window. Upon retransmission, the timer is backed off exponentially, clamped by a constant[5]. Incoming acknowledgments reset the timer.

A correct MPI program must call MPI_Finalize() before a group member process exits. As mentioned before, our protocol relies on this. Before a process leaves the group, it makes sure that it has received acknowledgments for all messages it has sent, and sends out acknowledgments for all packets it has received. If it has not received acknowledgments for sent-out packets after retransmitting a fixed number of times, the process exits with a warning message, and it is up to the user to make sure the application finished correctly. Assuming that message (and acknowledgment) losses rarely happen, we feel this two-way handshake release protocol is adequate to handle the three-army problem[2]

In summary, we suggest to use a reliable message stream with a "go-back-n"[2] window-based protocol. The send window is only limited by the size of the U-Net transmission queue, whereas at the receiver, a window size of one at the ADI level discards out-of order packets. At the U-Net level however, the receive window size is given by the size of the U-Net recieve queue, such that packets can arrive in a burst without being dropped.

5.2 Short Versus Long Messages

The above protocol is appropriate for short messages, and it is assumed that there is enough buffer space on the receiver's side to store the incoming data until the application posts the matching receive. For longer messages, a rendezvous protocol is necessary where the sender first sends a message "request to send", waits until the receiver sends "o.k. to send", and then transfers the data. During the data transfer, both sides are in the communication library routines, and one can perform congestion and flow control. We defer the handling of long messages to future work.

[3] In our test implementation, the MPI payload is 1482 bytes

[4] We assume that the network does not reorder messages, so a gap in the sequence numbers indicates a packet loss

[5] The best clamping constant should be determined by benchmarking. We suggest 0.1 sec as a reasonable value.

5.3 Sender Versus Receiver Initiated Retransmission

In MPI, every send has to have a matching receive, which opens the possibility of receiver-initiated retransmission. In other words, when the receiver is expecting a message and has not received after a certain time, it sends a request for retransmission. While this is in principle possible, we decided for sender-based retransmission. The reason is that MPI allows the use of wild cards, so the receiver might not know from whom to expect the data, and would have to broadcast its retransmission request to all members in the group. They in turn might retransmit packets which have arrived correctly, but had not been acknowledged yet.

5.4 Comparison With TCP

At a first glance, there are many similarities between TCP and our protocol, e.g. sliding window, sequence numbers, sender-based retransmission, and acknowledgments. However, there are three differences worthwhile noting:

- Unlike in TCP/IP, there is no guarantee that the protocol at the receiver's side is serviced by the CPU when the sender sends. This makes the use of retransmission timers difficult. An expired timer is not a good indicator of packet loss – the application on the other side could be off doing computations, for instance because of load imbalance.
- For small messages, a similar issue exists on the sender side. Performance reasons (and for the non-blocking case the MPI semantics) mandate a release of control by the ADI layer, which in turn makes it impossible to build a TCP-style data transfer with a round trip estimator.
- Whereas TCP provides a reliable byte stream, we operate on a per message basis. This requires a smaller sequence number space and results in a smaller header.

6 Test Implementation

To verify correct execution, and to get an idea of the amount of overhead introduced by the protocol, we implemented and tested it on two Pentium Pro 200 MHz under Free BSD 2.2 connected by 100 Mb/s Tulip Fast Ethernet. We [6] ported the available Linux U-Net to Free BSD, and implemented our protocol inside the ADI-2 layer of the freely available MPICH version of MPI.

6.1 Details of the Implementation

Of the ADI layer, we implemented those functions required to run a ping pong benchmark for timing: MPID_Init(), MPID_End(), MPID_CH_Eagerb_send_short() and MPID_CH_Check_incoming(). The latter two belong to the MPICH channel abstraction, and are called by ADI layer functions on receive and send, respectively. Communication progresses only while the thread of control is with the ADI layer. For simplicity, we maintain only a single retransmission timer for the left edge of the send window.

Since short sends are non-blocking, they are performed as follows:

1. preparing the 16-byte MPI header, of which three words are reserved for the MPI layer's tag, length and mode field, and one word carries the 15 bit sequence and acknowledgment numbers along with a SEQ and ACK bit indicating the validity of those fields.

[6] the porting was done by Pat Bozeman

2. moving the header and user data as a packet into the transmission buffer area inside the memory segment shared with the U-Net device.

3. Pushing a descriptor into the U-Net transmission FIFO containing the destination, length, and a pointer to the packet.

4. Trapping into the Kernel to transfer the packet.

5. If there are no older unacknowledged packets, reset the timer.

6. Increase the right send window edge by one message.

When the ADI layer is invoked to receive a message in blocking mode, the following happens:

1. The U-Net device is queried in a spin lock until a message comes in. While no message is coming in, we check if the oldest message sent off has become due for retransmission. If overdue, we retransmit this message, resets the timer, and go back spinning for an incoming message.

2. If the incoming message is a duplicate or out-of-order packet, an acknowledgment with the left receive window (duplicate ack) is sent out immediately.

3. In case the incoming message is in sequence, advance the left edge of the receive window, and remember the sequence number such that the acknowledgment can be piggy backed on the next send[7].

4. If the incoming message has a valid ACK field, reset the retransmission timer and advance the left edge of the send window according to the acknowledgment number.

All of this happens inside the MPID_CH_Check_incoming(), which is basically the ADI progress engine.

Finally, MPID_End() finishes by retransmitting until all packets are acknowledged, or a timer goes off. If it has not received acknowledgments for all packets by then, it issues a warning message and exits. It is then the responsibility of the user to ensure that the application has completed correctly.

On the sender side, there is only a single memory copy from the user's send buffer into the U-Net transmission buffer. From there, the packet gets DMAed directly into the network. On the receive side, we were able to collapse the ADI layer to perform only a single memory copy from the U-Net receive buffer space into the user's receive buffer. This was accomplished by leaving the packet inside the U-Net receive buffer until the MPI header is inspected, and the destination of the data is known. The Tulip Device does not have this flexibility – the incoming packet gets first placed into kernel buffers, and, after the U-Net channel is known, moved to the appropriate U-Net receive buffers. In summary, we have two memory copies on the receive side, and one memory copy for the sender.

[7] To make consumer/producer type of applications work better, we could count the number of acknowledgments accumulated for piggy backing, and send a pure acknowledgment packet out if it exceeds a certain threshold.

6.2 Performance Measurements

To assess the performance of our implementation, we ran a ping pong benchmark between two hosts, and measured the round trip time for a four byte long message (twenty bytes long including the MPI header). We find latencies of about 32 μs. Timing the raw U-Net ping pong benchmark revealed that the ADI layer is responsible for about 11 of the 32 μs. A more detailed timing of our ADI code was severely hampered by the lack of accurate timers. Nevertheless, it appears that the overhead added by the reliability protocol is only about 3-4 μs. Another 7-8 μs are spent for searching the MPI posted and unexpected queue, for message handling, and for overhead incurred by the multiple device support.

Our first naive implementations showed a latency of 60 μs, which under great efforts we curbed to 32 μs. There might still be considerable room for improvement by optimizing the queue searches and hand-coding the most time critical routines in assembly language.

7 Related Work

A large number of groups is pursuing high performance computing on commodity hardware. We could only find few reports of performance numbers for MPI on such systems.

- The Fast Messages Project [5] achieves MPI latencies of 19 μs and a peak bandwidth of 17.3 MB/s on two SPARCstations 20/71 connected by a Myrinet Network.
- The NOW group [1] at Berkeley reports MPI latencies of 17 μs between UltraSparc workstations connected with Myrinet Switches.
- Efforts by V. Gupta to build MPI on top of the unreliable AAL5 have announced ont the WWW (http://web.cps.msu.edu/ guptavi1/proj.html), but nothing has been published yet.
- S. Nog and D. Kotz present measurements of MPICH on TCP/IP with Fast Ethernet (http://www.cs.dartmouth.edu/reports/abstracts/TR95-273). They find latencies of 900 μs for MPICH on Fast Ethernet between Pentiums P/100.

8 Future Work

We have reported first efforts towards a portable MPI library on top of an unreliable network interface. As a next step, we intend to include the rendezvous protocol to be able to send large messages, and get accurate bandwidth measurements. For this transfer mode, we certainly have to provide flow control. Once this phase is completed, it is not much more work to get the complete ADI and then MPI up and running. Only then will we be in a position to evaluate the protocol in a real environment, and test its performance and correctness for a wider range of cases.

Lastly we would like to mention that Intel, Compaq, and Microsoft have joined forces to create the Virtual Interface Architecture (VIA) standard, see for instance

http://www.pentium.com/pressroom/archive/releases/SP041697.HTM .

While we must remain confidential about the details, we can say that VIA is similar enough to U-Net to require only small modifications to our existing U-Net implementation.

9 Acknowledgments

This work was performed in collaboration with Pat Bozeman (porting of the Free BSD U-Net Device Driver) , William Saphir (expert on MPI semantics, discussion of the protocol, help with implementing the TCP part of the startup phase) and Bill Johnston (group leader).

References

[1] *The Berkeley Networks of Workstations (NOW) Project*, T.E. Anderson, D.E. Culler, and D.A. Patterson, Digest of Papers. COMPCON '95., San Francisco, CA, USA, 5-9 March 1995, IEEE Comput. Soc. Press, 1995. p. 322-6.

[2] *Computer Networks*, Andrew S. Tanenbaum, third edition, Prentice Hall (1996).

[3] *Low-Latency Communication over Fast Ethernet*, M. Welsh, A. Basu, T. von Eicken, Proceedings of Euro-Par '96, Lyon, France, August 27-29, 1996.

[4] *U-Net: A User-Level Network Interface for Parallel and Distributed Computing*, A. Basu, V. Buch, W. Vogels, T. von Eicken, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP), Copper Mountain, Colorado, December 3-6, 1995.

[5] *MPI-FM: High performance MPI on workstation clusters* M. Lauria, A. Chien, Journal of Parallel and Distributed Computing, 10 Jan. 1997, vol.40, (no.1):4-18.

[6] *A high-performance, portable implementation of the MPI message passing interface standard*, W. Gropp, E. Lusk, N. Doss, and A. Skjellum, Parallel Computing, Sept. 1996, vol.22, (no.6):789-828.