Design and Implementation of MPI on Portals 3.0

Ron Brightwell[1], Arthur B. Maccabe[2], and Rolf Riesen[1]

[1] Scalable Computing Systems, Sandia National Laboratories, Albuquerque, NM 87185-1110, USA,
{bright,rolf}@cs.sandia.gov
[2] Computer Science Department, University of New Mexico, Albuquerque, NM 87131-1386
maccabe@cs.unm.edu.

Abstract. This paper describes an implementation of the Message Passing Interface (MPI) on the Portals 3.0 data movement layer. Portals 3.0 provides low-level building blocks that are flexible enough to support higher-level message passing layers such as MPI very efficiently. Portals 3.0 is also designed to allow for programmable network interface cards to offload message processing from the host processor. We will describe the basic building blocks in Portals 3.0, show how they can be put together to implement MPI, and describe the protcols of an MPI implementation. We will look at several key operations within an MPI implementation and describe the effects that a Portals 3.0 implementation has on scalability and performance.

1 Introduction

The advent of cluster computing has motivated much research into message passing APIs and protocols targeted for delivering low-latency high-bandwidth performance to parallel applications. Relatively inexpensive programmable network interface cards (NICs), like Myrinet [1], have made low-level message passing protocols and programming interfaces a popular area of research [2–8]. Most of these activities have been focused on delivering latency and bandwidth performance as close to the hardware limitations as possible.

The current Portals [9, 10] data movement interface (Portals 3.0) is an evolution of networking technology initially developed for large-scale distributed memory massively parallel systems. Portals began as a key component of our lightweight compute node operating systems [11, 12], and has evolved into a programming interface that can be implemented efficiently for different operating systems and networking hardware. In particular, Portals provides the necessary building blocks for higher-level protocols to be implemented on programmable or intelligent network interfaces without providing mechanisms that are specific to each higher-level protocol. This paper describes how these building blocks and their associated semantics can be combined to support the protocols needed for a scalable high-performance implementation of the Message Passing Interface (MPI) Standard [13].

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

The rest of this paper is organized as follows: In the next section, we give a brief overview of the Portals 3.0 API. In section 3, we present the initial implementation of MPI on Portals 3.0. Section 4 describes a problem that we encountered with this implementation and Section 5 describes a second implementation of MPI that uses a new semantic added to Portals to overcome this limitation. We briefly discuss the benefits of our implementation with respect to progress in Section 6 and conclude in Section 7 with summary of the key points of this paper.

2 Portals 3.0

The Portals 3.0 API is composed of elementary building blocks that can be combined to implement a wide variety of higher-level data movement layers such as MPI. We have tried to define these building blocks and their operations so that they are flexible enough to support other layers as well as MPI, but MPI was our main focus. The following sections describe Portals objects and their associated functions.

The Portals library provides a process with access to a virtual network interface. Each network interface has an associated Portal table that contains at least 64 entries. The table is simply indexed from 0 to $n-1$, and the entries in the table normally correspond to a specific high-level protocol. Portal indexes are like port numbers in Unix. They provide a protocol switch to separate messages intended for different protocols.

Data movement is based on one-sided operations. Other processes can use a Portal index to read (get) or write (put) the memory associated with the portal. Each data movement operation involves two processes, the **initiator** and the **target**. In a put operation, the initiator sends a put request containing the data to the target. The target translates the Portal addressing information using its local Portal structures. When the request has been processed, the target may send an acknowledgement. In a get operation, the initiator sends a get request to the target. The target translates the portal addressing information using its local Portal structures and sends a reply with the requested data.

Typically, one-sided operations use a triple to address remote memory: ⟨process id, buffer id, offset⟩. In addition, portal addresses include a set of match bits. Figure 1 presents a graphical representation of the structures used to translate Portal addresses. The process id is used to route the message to the target node. The buffer id is used as an index into the Portal table, this identifies a match list. The match bits (along with the id of the initiator) are used to select a match entry (ME) from the match list. The match entry identifies a list of memory descriptors. The first of these identifies a memory region and an optional event queue.

The match entries in a match list are searched in sequential order. If a match entry matches the request, the first memory descriptor associate with the match entry will be tested for acceptance of the message. If the match entry does not match the message or the memory descriptor rejects the message, the message continues to the next match entry in the list. If there are no further match entries, the messages is discarded.

Memory descriptors (MDs) have a number of options that can be combined to increase their utility. They can be configured to respond to put operations, get operations, or both. Each memory descriptor has a threshold value that determines how many op-

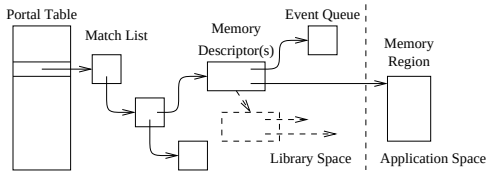

Fig. 1. Portal Addressing Structures

erations can occur before the descriptor becomes inactive. In addition, each memory descriptor has an offset value which can be managed locally or remotely. When it is managed locally, the offset is increased by the length of each message that is deposited into the memory descriptor. Consecutive messages will be placed in the user's memory one after the other. Memory descriptors can specify that an incoming message which is larger than the remaining space will be truncated or rejected.

Memory descriptors can be chained together to form a list. Moreover, each memory descriptors can be configured to be unlinked from the list when it is consumed (i.e., when its threshold becomes zero). When the last descriptor in the list is consumed and unlinked, the associated match entry also becomes inactive. Each match entry can be configured to unlink when it becomes inactive.

The decision to generate an acknowledgement in response to a put operation requires input from both the initiator and the target. The initiator must request an acknowledgement and the memory descriptor used in the operation must be configured to generate an acknowledgment.

Memory descriptors may have an associated event queue (EQ). Event queues are used to record operations that have occured on memory descriptors. Multiple memory descriptors can share a single event queue, but a memory descriptor may only have one event queue.

There are five types of events that represent the five operations that can occur on a memory descriptor. A `PTL_EVENT_GET` event is generated when a memory descriptor responds to a get request. A `PTL_EVENT_PUT` event is generated when a memory descriptor accepts a put operation. A `PTL_EVENT_SENT` event is generated when it is safe to manipulate the memory region used in a put operation. A `PTL_EVENT_REPLY` event is generated when the reply from a get operation is stored in a memory descriptor. A `PTL_EVENT_ACK` event is generated when a acknowledgement arrives from the target process. In addition to the type of event, each event records the state of the memory descriptor at the time the event occurred.

3 Initial MPI Implementation

In this section, we describe our MPI implementation for Portals 3.0. This implementation is a port of MPICH [14] version 1.2.0, which uses a two-level protocol based on message size to optimize latency for short messages and bandwidth for large messages. In addition to message size, the different protocols are used to meet the semantics of the different MPI send modes.

We use the match bits provided by Portals to encode the send protocol (3 bits), the communicator (13 bits), the local rank (within the communicator) of the sending process (16 bits) and the MPI-tag (32 bits). During `MPI_Init()`, we set up three Portal table entries. The *receive Portal* is used for receiving MPI messages. The *read Portal* is used to for unexpected messages in the long message protocol. The *ack Portal* is used for receiving acknowledgements for synchronous mode sends. We also allocate space for handling unexpected messages, the implementation of which we describe below. After these structures have been initialized, all of the processes in the job call a barrier operation to insure that all have been initialized.

3.1 Short Message Protocol

Like most MPI implementations, we use an eager protocol for short messages (standard and ready sends). The entire user buffer is sent immediately and "unexpected messages" (where there is no pre-posted receive) are buffered at the receiver.

For standard sends, we allocate an MD to describe the user buffer and an EQ associated with the MD. We configure the MD to respond to put operations and set the threshold to one. We create an address structure that describes the destination and fill in the match bits based on the protocol and MPI information. Using the Portal put function, we send the MD to the *receive Portal* of the destination process, requesting that no acknowledgement be sent. This send is complete when an event indicating that the message has been sent appears in the EQ.

For synchronous sends, we need to know when the message is matched with a receive. We start by allocating an MD and an EQ as described earlier. Next, we build an ME that uniquely matches this message. This ME is associated with the MD allocated in the previous step and attached to the local *ack Portal*. We configure the MD to respond to acknowledgements and put operations and set the threshold to two. When the Portal put operation is called, we request an acknowledgment, and include the match bits for the ME in 64 bits of out-of-band data, called the header data.

Completion of a short synchronous mode send can happen in one of two ways. If the matching receive has been posted, an acknowledgment is generated by the remote memory descriptor as illustrated in left side of Figure 2. If the matching receive is not posted, the message is buffered until the matching receive is posted. At this point, the receiver will send an explicit acknowledgment message using the Portal put operation, as illustrated on the right side of Figure 2. A short synchronous mode send completes when the `PTL_EVENT_SENT` event has been recorded and either the `PTL_EVENT_ACK` event or the `PTL_EVENT_PUT` event has been recorded.

3.2 Long Message Protocol

Unlike most MPI implementations, we also use an eager protocol for long messages. That is, we send long messages assuming that the receiver has already posted a matching receive. Because the message could be discarded, we also make it possible for the receiver to get the message when it finally posts a matching receive.

We start by inserting an ME that uniquely describes this send on the *read Portal*. We create an MD that describes the user buffer. This MD is configured to respond to put

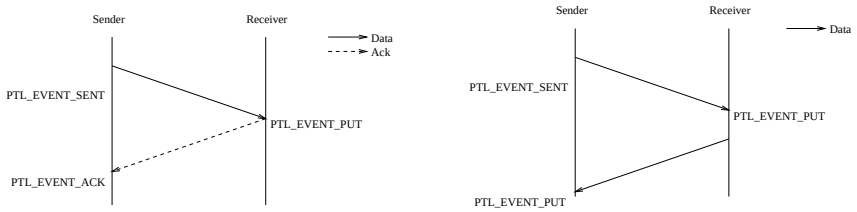

Fig. 2. Short Synchronous Send Protocol: Expected and Unexpected

operations, get operations, and ack operations. Since all of three of these may occur, we set the MD's threshold to three. We then attach the MD to the ME. We set the protocol bits in the match bits for a long send and fill in the other match bits appropriately. We call the Portal put function, requesting an acknowledgment and we include the match bits of the ME in the header data.

As with the short synchronous mode sends, the long send protocol can complete in one of two ways. First, if the message is expected at the receiver, an acknowledgment is returned. In this case, the event queue will contain a `PTL_EVENT_SENT` event and a `PTL_EVENT_ACK` event. After these two events have been recorded, the send is complete. This is illustrated on the left side of Figure 3.

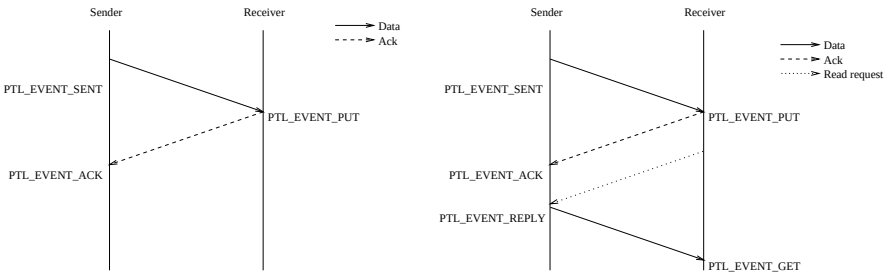

Fig. 3. Long Standard Mode Send: Expected, Unexpected

If the message was not expected, an acknowledgment is returned to the sender indicating that the receiver accepted zero bytes. The sender must then wait for the receiver to request the data from the sender's read Portal. In this case, three events mark the completion of the send: a `PTL_EVENT_SENT` event, a `PTL_EVENT_ACK` event, and a `PTL_EVENT_GET` event. This is illustrated on the right side of Figure 3.

Since the completion of this send protocol is dependent on a matching user buffer being posted at the receiver, this protocol is the same for standard mode and synchronous mode sends. Moreover, the ready mode send for long messages is identical to a short standard mode send. Since the MPI semantics guarantee that a matching buffer is posted at the receiver, the sender need not wait for an acknowledgment or set up an entry on the read Portal.

3.3 Posting Receives

Posting a receive involves two lists: the posted receive list and the unexpected message list. The unexpected message list holds messages for which no matching receive has been posted. Before a receive can be added to the posted receive list, we must search the unexpected list. Figure 4 illustrates the match list structure we use to represent these two list. This list starts with entries for the posted receives (no entries are shown), followed by a *mark* entry that separates the two lists, followed by entries for unexpected messages.

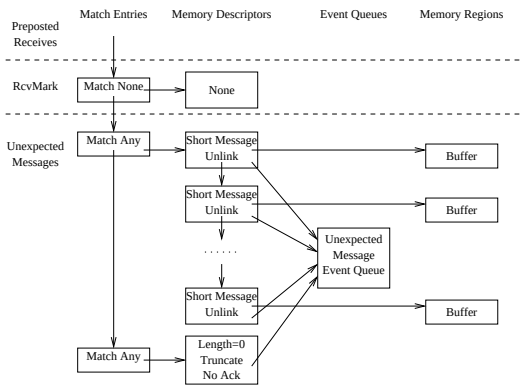

Fig. 4. Message Reception in the Initial Implementation

When a process calls an MPI receive, we create a memory descriptor that describes the user buffer. The MD is configured to accept put operations, generate acknowledgements, unlink when used, and is given an initial threshold of zero (making the MD inactive). The communicator id, source rank within the communicator, and message tag are translated into match bits. We use these match bits along with the MD to build an ME which is inserted just in front of the *mark* entry in the match list.

Next, we search the unexpected messages for a match. If no match is found we activate the MD by setting its threshold to one (this test and update is atomic with respect to the arrival of additional unexpected messages). If we find a match, we unlink the ME from the match list and take the appropriate action based on the protocol bit in the message. If the message is a long protocol message, we activate the MD and perform a Portal get operation, which reads the send buffer from the sender. If the message is a short protocol message, we simply copy the contents of the unexpected buffer into the user's receive buffer. If the header data in the event is nonzero, this indicates that the send is synchronous. In this case, we send a zero-length message to the ack Portal at the sender using the Portal put operation.

4 Limitations

When the MPI library is initialized, we create two match entries, one for unexpected short protocol messages and another for unexpected long protocol messages. The match entry for short messages will match any message that has the short message bit set. There are 1024 memory descriptors attached to this match entry. Each MD provides an 8 KB memory buffer, has a threshold of one, is configured to accept put operations, and does not automatically generate an acknowledgment. The match entry for long unexpected messages has an MD of zero bytes with an infinite threshold that is configured to accept and truncate put operations and automatically generate an acknowledgment. All of the MD's for unexpected messages are attached to the same event queue to preserve message ordering. This event queue is created with 2048 entries.

Despite being able to support 1024 outstanding unexpected short messages at any time, this limitation proved to be too restrictive in practice, especially as applications scaled beyond 700 processes. Our application developers typically think of limits in terms of the messages sizes rather then message counts. In our implementation, an unexpected message of 0 bytes would consume an 8 KB memory descriptor. To an application developer, this message shouldn't consume any buffer space at all. Moreover, in a NIC-based implementation, 1024 MD's consumes a significant amount of a limited resource, NIC memory, leaving fewer MD's for posted receives.

5 Second Implementation

To address these problems, an additional threshold semantic was added to MDs. An MD could be created with a maximum offset, or high-water mark. Once this offset was exceeded, the MD would become inactive. Figure 5 illustrates our new strategy for handling short protocol unexpected messages. We now create three match entries for unexpected short messages, all with identical selection criteria. Each ME has an MD attached to it that describes a 2 MB buffer. As unexpected messages come into the MD, they are deposited one after the other until a message causes the maximum offset to be exceeded. When this happens, the MD is unlinked, and the next unexpected short message will fall into the next short unexpected MD. Once all of the unexpected messages have been copied out of the unlinked MD, the ME and MD can be inserted at the end of the short unexpected ME's.

This new strategy allows for the number of unexpected messages to be dependent on space rather than count. It also reduces the number of MD's for short unexpected messages to three, significantly reducing the amount of NIC resources required by MPI. The handling of posted receives and unexpected long protocol messages did not change. This semantic change to Portals eliminated the need for lists of memory descriptors, so the API was changed to allow only a single memory descriptor per match entry.

6 Progress

Unlike most MPI implementations described in current literature [15–18], the semantics of Portals 3.0 support the necessary progress engine for an MPI implementation

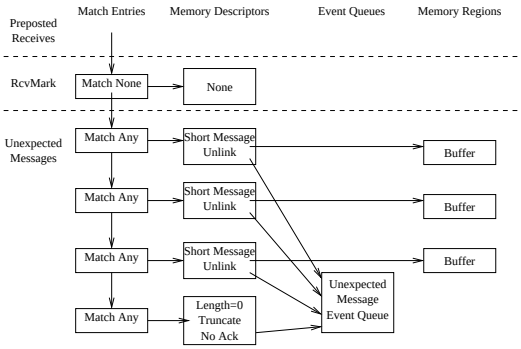

Fig. 5. Message Reception in Second Implementation

without the need for explicit application intervention. Portals' flexible message selection semantics and the use of eager protocols allow communication operations to make progress independent of making frequent MPI library calls. While progress of Portals messages is dependent on the underlying transport layer, NIC-based implementations are able to support progress for Portals, and hence MPI, without any other mechanism, such as a user-level thread.

7 Summary

This paper has described an implementation of MPI on the Portals 3.0 data movement layer. We have illustrated how Portals 3.0 provides the necessary building blocks and associated semantics to implement MPI very efficiently. In particular, these building blocks can be implemented on intelligent network interfaces to provide the necessary protocol processing for MPI without being specific to MPI. The implementation described in this paper has been in production use on a 1792-node Alpha/Myrinet Linux cluster at Sandia National Laboratories for more than two years.

References

1. Boden, N., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., Su, W.: Myrinet-A Gigabit-Per-Second Local-Area Network. IEEE Micro **15** (1995) 29–36
2. von Eicken, T., Basu, A., Buch, V., Vogels, W.: U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In: Proceedings of the Fifteenth SOSP. (1995) 40–53
3. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active Messages: A Mechanism for Integrated Communications and Computation. In: Proceedings of the 19th International Symposium on Computer Architecture. (1992)
4. Pakin, S., Karamcheti, V., Chien, A.A.: Fast Messages(FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors. IEEE Concurrency **40** (1997) 4–18
5. Myricom, Inc.: The GM Message Passing System. Technical report, Myricom, Inc. (1997)

6. Ishikawa, Y., Tezuka, H., Hori, A.: PM: A High-Performance Communication Library for Multi-user Parallel Envrionments. Technical Report Technical Report TR-96015, RWCP (1996)
7. Compaq, Microsoft, and Intel: Virtual Interface Architecture Specification Version 1.0. Technical report, Compaq, Microsoft, and Intel (1997)
8. Prylli, L.: BIP Messages User Manual for BIP 0.94. Technical report, LHPC (1998)
9. Brightwell, R.B., Hudson, T.B., Maccabe, A.B., Riesen, R.E.: The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratories (1999)
10. Brightwell, R., Lawry, W., Maccabe, A.B., Riesen, R.: Portals 3.0: Protocol Building Blocks for Low Overhead Communication. In: Proceedings of the 2002 Workshop on Communication Architecture for Clusters. (2002)
11. Maccabe, A.B., McCurley, K.S., Riesen, R.E., Wheat, S.R.: SUNMOS for the Intel Paragon: A brief user's guide. In: Proceedings of the Intel Supercomputer Users' Group. 1994 Annual North America Users' Conference. (1994) 245–251
12. Shuler, P.L., Jong, C., Riesen, R.E., van Dresser, D., Maccabe, A.B., Fisk, L.A., Stallcup, T.M.: The Puma operating system for massively parallel computers. In: Proceedings of the 1995 Intel Supercomputer User's Group Conference, Intel Supercomputer User's Group (1995)
13. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. The International Journal of Supercomputer Applications and High Performance Computing **8** (1994)
14. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing **22** (1996) 789–828
15. Lauria, M., Chien, A.A.: MPI-FM: High Performance MPI on Workstation Clusters. Journal of Parallel and Distributed Computing **40** (1997) 4–18
16. Prylli, L., Tourancheau, B., Westrelin, R.: The Design for a High Performance MPI Implementation on the Myrinet Network. In: Proceedings of the 6th European PVM/MPI Users' Group. (1999) 223–230
17. O'Carroll, F., Hori, A., Tezuka, H., Ishikawa, Y.: The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In: ACM SIGARCH ICS'98. (1998) 243–250
18. Dimitrov, R., Skjellum, A.: An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In: Proceedings of the Third MPI Developers' and Users' Conference. (1999) 15–24