

Design and Implementation of MPI on Puma Portals *

Ron Brightwell
Lance Shuler
Massively Parallel Computing Research Laboratory
Sandia National Laboratories
Albuquerque, NM
{bright,shuler}@cs.sandia.gov

Abstract

As the successor to SUNMOS [8], the Puma operating system provides a flexible, lightweight, high performance message passing environment for massively parallel computers. Message passing in Puma is accomplished through the use of a new mechanism known as a portal. Puma is currently running on the Intel Paragon and is being developed for the Intel TeraFLOPS machine.

In this paper we discuss issues regarding the development of the Argonne National Laboratory/Mississippi State University implementation of the Message Passing Interface standard on top of portals. Included is a description of the design and implementation for both MPI point-to-point and collective communications, and MPI-2 one-sided communications.

1. Introduction

1.1. MPICH

MPICH is a portable implementation of the Message Passing Interface (MPI) [3] standard developed jointly by Argonne National Laboratory and Mississippi State University. MPICH contains an abstract device interface (ADI) upon which a high-level message passing application programmer interface such as MPI can be implemented. The ADI performs four main functions[6]: sending and receiving, data transfer, queueing, and device-dependent functions.

Porting MPICH to an architecture such as the Paragon involves the creation of new “device” that interacts with the ADI through a set of routines (see [5] for details) and handles. These handles are used to cache device specific data

*This work was performed at Sandia National Laboratories, operated for the U. S. Department of Energy under contract No. DE-AC04-94AL85000.

to pass information between the device independent and device dependent layers of MPICH.

1.2. Puma and Portals

Puma is an operating system designed to provide a flexible, lightweight, high performance message passing environment for massively parallel computing[11]. Message passing in Puma is accomplished through the use of portals, which are structures that inform the kernel how and where incoming messages should be deposited. Each application is allotted a finite number of portals in a portal table, and each entry in the portal table has an associated memory descriptor which describes how the memory is arranged. Messages destined for a particular portal table entry are deposited according to the type of memory descriptor attached to it. Additionally, matching lists may be attached to a portal table entry in order to provide further selection criteria for messages destined for a particular portal. Each match list entry contains 64 match bits and 64 ignore bits. The ignore bits can be used to mask off insignificant match bits. These matching lists in turn have memory descriptors associated with them. There are four basic types of memory descriptors.

The most basic is the single block memory descriptor, which describes a single contiguous block of memory. Messages destined for a portal with single block memory descriptor attached may be deposited anywhere within this single contiguous region.

A dynamic block memory descriptor describes a contiguous block of heap memory. The Puma kernel maintains a list of free memory blocks and a list of messages that have arrived. Messages destined for a portal with a dynamic block memory descriptor attached will be deposited in the first available space within this heap, and the message will be added to the end of an incoming message queue.

The independent block memory descriptor describes a table of possibly noncontiguous buffers. An independent

block contains a buffer descriptor table, each entry of which describes a contiguous block of memory. A message destined for a portal with an independent block memory descriptor attached will be deposited in the first available buffer in the buffer descriptor table.

Finally, the combined block memory descriptor describes a logically contiguous but possibly physically discontinuous block of memory. This descriptor is almost identical to an independent block descriptor, but rather than depositing a message into a single buffer, a message destined for this descriptor will keep filling successive buffers in the buffer descriptor table until reception is complete.

Each type of memory descriptor also has several configurable options regarding how to respond to incoming messages and how to progress through buffer lists. The following describes the options for each descriptor type:

- **Single block:**
 - Sender or receiver managed offset
 - Save message header only, save message body only or save both header and body
 - Read only or write only
 - Acknowledge sender
- **Dynamic block:**
 - Save message header only or save both header and body
 - Acknowledge sender
- **Independent block:**
 - Circular or linear buffer list
 - Save message header only or save both header and body
 - Acknowledge sender
 - Read only or write only
- **Combined block:**
 - Sender or receiver managed offset
 - Read only or write only
 - Acknowledge sender

Certain descriptors may also be overlaid so that the same memory region is accessed or manipulated by multiple match list entries or portals. For example, a match list entry may be overlaid onto another match list entry which has an independent block memory descriptor. Messages which are destined for either match list entry update the same independent block buffer table.

A match list is a linked list of match entries. When a message arrives for a portal with a match list attached, the

kernel traverses the list, comparing the group, rank, and match bits in the incoming message header to those in each match list entry. When a match is found, the kernel then attempts to deposit the message into the associated memory descriptor.

There are three types of failure the kernel can experience at each match list entry. The kernel can fail because the matching criteria are not met, because the memory descriptor has no available buffer, or because the available buffer is too small to hold the incoming message. Each entry can specify the next successive entry to which the kernel should proceed upon encountering any of these three failures.

Many of the options used by the different memory descriptors require information contained in the incoming message. The send side is responsible for providing this information depending on the type of send. For example, a memory descriptor that is configured to send an acknowledgment back to the sender needs to know to which portal the acknowledgment needs to return and what the match bits should be. Similarly, a sender managed memory descriptor needs to get the desired offset from the incoming message. The following values are required for the short message protocol: send buffer, number of bytes, destination offset, destination group, destination rank, destination portal, and destination match bits. In addition, the following values are needed for long and synchronous message protocols: return match bits, return portal, return length, return offset, and user data (twelve bytes). The configuration of the destination portal and memory descriptor will determine how and if some of these values are used. For example, a message that is destined for a portal with a memory descriptor attached, but no match list, with ignore the destination matchbits.

2. Point-to-Point Design and Implementation

A two-level protocol was decided upon at the outset to provide low latency for shorter messages and high bandwidth for larger ones. Figure 1 illustrates the portal used for receiving messages. The entry in the portal table points to a match list that contains an entry for each active receive that has been posted. The match bits for posted receives are used as follows:

- 32 bits for message tag
- 16 bits for local source rank
- 13 bits for context identifier
- 3 bits for message type

Each match list entry for a posted receive uses an independent block memory descriptor configured to save both

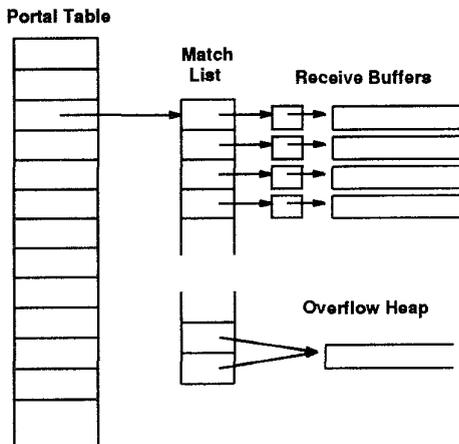


Figure 1. Receive portal.

the message header and the message body. Each independent block memory descriptor contains only one buffer. A single block memory descriptor cannot be used because a single block can only save incoming data and not header information. A posted receive needs header information in order to distinguish between different protocols and to obtain message tag and source values should these be wildcarded.

The final two entries in the match list are used to catch and queue unexpected messages for each protocol. The first catchall entry has a dynamic block memory descriptor configured to save both the message header and the message body. The second catchall entry also has a dynamic block memory descriptor, but is configured to save only the message header. The second catchall entry is overlaid on top of the first catchall entry so that both entries use the same heap list structure, insuring correct ordering for unexpected messages. The three bits for message type in the match bits are used to choose the catchall entry in which unexpected messages are buffered. For the short catchall entry, the matchbits are configured to ignore all bits except the first three, which must be zero. The long catchall entry is likewise configured to ignore all bits except the first three, the third of which must be set. Messages with the first message type bit set are ready send messages which must have a pre-posted receive. Consequently, these messages have no overflow buffer and will simply be discarded if there is no pre-posted receive. The second message type bit is used to distinguish between regular messages and reply messages which the receiver has requested. The match list entries for pre-posted receives are configured to ignore the ready send and long send message type bits.

Figure 2 illustrates the portals needed for sending messages. The first entry in the portal table contains a match list that is used for collecting acknowledgments from the receivers. Each acknowledgment is a message header with

the result (saved header, saved body, or saved header and body) of the message reception contained in the first byte of the twelve byte user data portion of the header. Each entry contains an independent block memory descriptor configured to save only the message header.

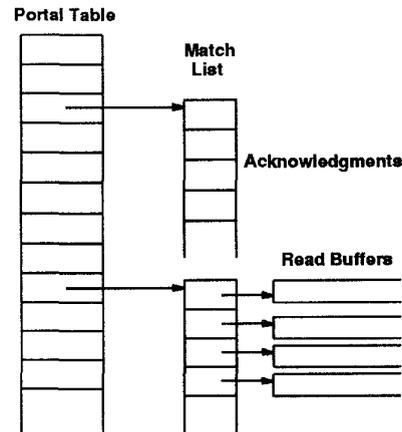


Figure 2. Send portals.

The second entry in the portal table contains a match list where each entry contains a single block memory descriptor configured to reply. A message destined for this type of read portal will cause the data in the appropriate buffer to be replied back to the original message's point of origin.

Because the match bits for both the read portal and the acknowledge portal must be unique for each send request, the first 32 match bits are set to the address of the device independent handle associated with the send request, while the second 32 bits are set to the address of the device dependent handle.

Figure 3 illustrates a short protocol send operation. Then sender sends both a message header and user data to the receive portal at the destination. The destination matchbits are set appropriately for the tag, the context identifier, rank within the communicator, and message type. The protocol type is also encoded in the user data portion of the message header. The short protocol send operation is complete once the kernel finishes delivering the message.

For the long send protocol (Figure 4), the sender uses an eager protocol where both the message header and the data are sent to the receiver. However, if there is no posted receive for a long message, only the header is saved, and the receiver must pull the message from the sender. After a message is sent, the sender waits for an acknowledgment. If a receive was pre-posted and the message was saved directly into the user buffer, the acknowledgment will indicate that both header and body were saved. If no receive was pre-posted, the message header will be saved in the dynamic heap and the acknowledgment will indicate that only

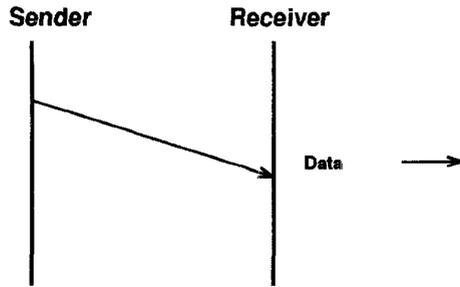


Figure 3. Short send protocol.

the header was saved. The sender must then wait for the appropriate number of bytes to be read from the single block portal.

To post a short protocol receive, a free receive match list entry is obtained and the necessary matching criteria is added. For short protocol receives, an independent block memory descriptor configured to save both header and body with no acknowledgment is attached to the entry. However, the entry is not activated until a search of the dynamic heap is performed. If there is an unexpected message stored in the dynamic heap that matches the receive that is being processed, the message is copied out of the heap and the space in the heap is freed. If the search of the heap is unsuccessful, the entry is activated. This operation must be atomic to insure that the kernel doesn't deposit a message in the dynamic heap between the time the heap is searched and the entry is activated. Message arrival on the entry is signalled by an update of the bytes written to the memory descriptor. Necessary header information is extracted when the message arrives.

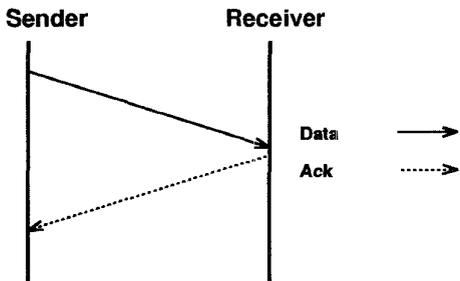


Figure 4. Long send protocol.

For the long protocol on the receive side, the match list entry is prepared in the same way as with the short protocol. However, the independent memory descriptor that is attached to the match list entry is configured to acknowledge the result back to the sender upon receipt of a message. Also, if a matching message header is found in the dynamic heap, the match bits are changed to accept a pulled

message, and a message is sent to the sender to pull the data across. Figure 5 illustrates the long protocol where the message must be pulled by the receiver.

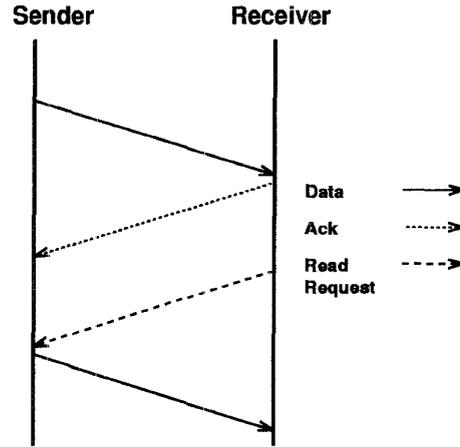


Figure 5. Long send read protocol.

The basic short and long protocols are extended to include an extra acknowledgment for synchronous messages. Synchronous acknowledgements are sent to the same portal as long send acknowledgments. For short synchronous messages, the return match bits are contained in the user data of the message header. In Figure 6 when a synchronous message is received, either by the posted receive or copied out of the heap, the receiver sends back a synchronous acknowledgment. In the long protocol, if the message is saved to a posted receive, the situation is similar to the short synchronous protocol. For long synchronous messages that are pulled, the extra acknowledgment is sent upon arrival of the pulled messages (Figure 7).

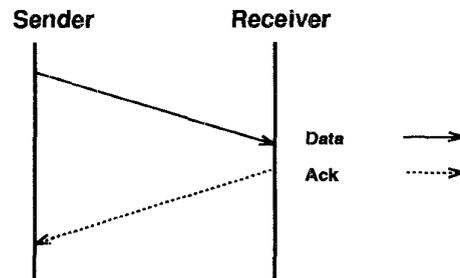


Figure 6. Short synchronous send protocol.

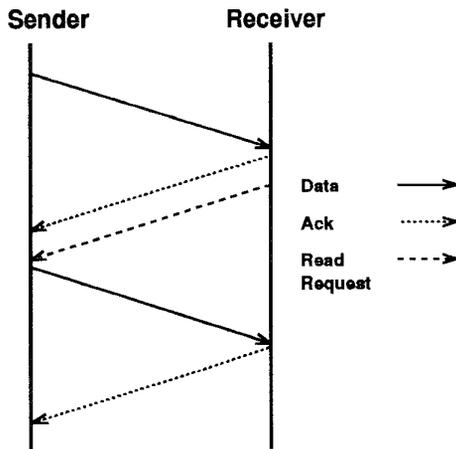


Figure 7. Long synchronous send read protocol.

3. Collective Communications Design and Implementation

In the Puma MPICH ADI, the collective communication operations are mapped to the native Puma collective communications which are built on top of Puma portals. This section discusses the implementation of the Puma collective communications on top of portals as it relates to MPI collective communications.

The native Puma collective communications are primarily interested in high performance collective communication with contiguous data over the entire range of vector lengths. They make use of hybrid techniques developed at the University of Texas [1, 10, 2] to achieve this full range of performance. The hybrid techniques use the physical multi-dimensional nature of an interconnect to maximize bandwidth and minimize message contention for long messages. For short messages, the hybrids make use of logical multi-dimensional mappings within each physical dimension (dimensional rings) to form new near-optimal short message algorithms. For medium length messages, the hybrids evaluate whether to use a short or long message algorithm in each logical/physical dimension to gain the best performance.

Since the implementation on top of portals concerns itself primarily with the short and long building block algorithms, this section will restrict discussions to the short/long building block implementations. Once these implementations are optimized, the advantages of the hybrids can be incorporated directly.

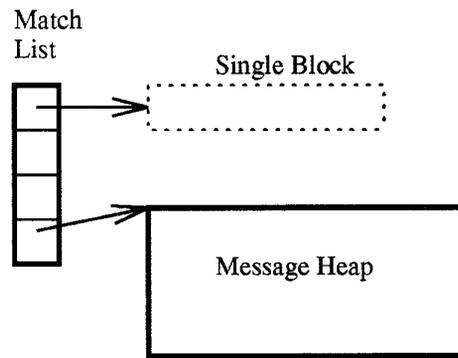


Figure 8. Portal structures for short message algorithms

3.1. Short Message Protocols

The best point-to-point short message algorithms embed a minimum spanning tree within the participating group of nodes in such a way as to enable the sending and receiving of contention free messages based on the structure of the tree. To support such a communication pattern, the Puma MPI collective communications use the structures as illustrated in Figure 8.

Consider the collective operation `MPI_Bcast()` for short messages. Before the application enters `main()`, the match list, and message heap are setup and initialized. The single block memory buffer is not present. When a child enters `MPI_Bcast()`, it checks whether the message has arrived in the message heap. If it hasn't, then the child sets up the single block buffer for the message that will be arriving. Upon entering `MPI_Bcast()`, the root node immediately begins sending to its children within the minimum spanning tree. It is possible that a parent may be sending before its children have reached `MPI_Bcast()`. If this is the case, then the broadcast message is placed in the child's message heap as soon as it arrives.

This implementation has advantages in that it both avoids unnecessary memory copies when a child enters `MPI_Bcast()` before the parent, and does not hold up the parent if a child is not ready. For longer messages, it may be desirable to avoid memory copies completely, in which case it is worthwhile to add an additional handshake between parent and child to make sure the child is ready before the parent sends.

In this description, `MPI_Bcast()` was used as an example collective operation. Other MPI operations such as `MPI_Scatter()`, `MPI_Gather()`, `MPI_Allgather()`, etc., all have minimum spanning tree algorithms [1] that would make use of the same portal structures for short messages.

Features that have not been implemented which are read-

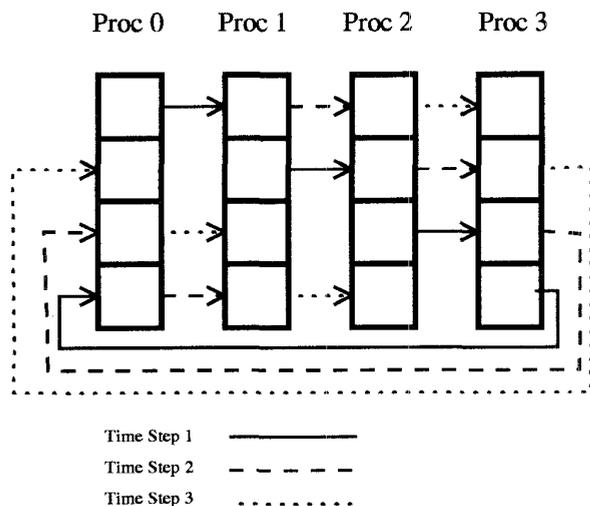


Figure 9. Bucket algorithm implementation for MPI_Allgather()

ily supported by this design include:

- multiple simultaneous collective operations
- asynchronous collective operations (via handler or co-processor)

3.2. Long Message Protocols

For long messages¹, versions of the “bucket” algorithm or “ring” algorithm have proven to be the most efficient, since the amount of data traversing the network is less than the amount transferred using a minimum spanning tree algorithm. Figure 9 illustrates the steps in a bucket algorithm for the MPI_Allgather() operation. At each time step, every process sends a piece of the message to one neighbor and receives a piece from another. Bucket algorithms are also natural for MPI_Reduce(), MPI_Reduce_scatter(), and MPI_Allreduce().

It is clear from Figure 9 that bucket algorithms are very lock-step in nature. As a result, it makes sense that a sending neighbor would synchronize with its receiving neighbor and then stream the pieces of the message into the waiting buffer. Figure 10 illustrates the portal structures necessary for supporting this mechanism. Each process sets up the receive buffer locally in the single block portal and sends a message to the sending neighbor announcing that it is ready to receive. In the mean time, it will wait for a

¹Long messages refer to messages with lengths larger than say 10Kbytes depending on the bandwidth and latency measurements for a given architecture and the number of processes participating in the operation.

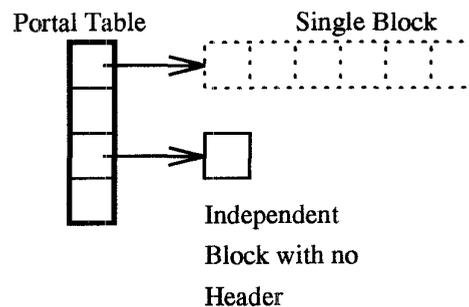


Figure 10. Portal structures for long message (bucket/ring) algorithms

ready-to-receive message from its receiving neighbor to arrive in its independent block portal². The ready-to-receive message will tell the process that a buffer is available at the receive neighbor and streaming data can begin. The process can watch the message counter on its single block portal to make sure it does not get ahead of the sending neighbor.

This long message portals implementation makes use of the lock-step characteristic of bucket algorithms to avoid memory copies which are costly for long messages. It accomplishes this by synchronizing with its neighbors and by following up with streaming data into the appropriate receive buffer. Also, this implementation cuts down on additional costs by being able to eliminate the need for an additional indirection through a match list.

It is worth noting that by switching out the single block memory descriptor and replacing it with a combined block memory descriptor, this design would support MPI non-contiguous datatypes.

3.3. Implementation Issues

In the management of both the short and long message building blocks, issues with race conditions and dropped messages due to overlapping collective operations arise and must be dealt with. Since all portal structures are in user space, race conditions can occur between the kernel and the process when both access the same structures at the same time. Cooperation between the kernel and the libraries can ensure that the race conditions do not occur.

Using the structures above, it is possible for back to back “fan-in” minimum spanning tree operations to overlap and lose messages. This is because in “fan-in” algorithms, for instance in MPI_Gather(), the leaf nodes can send their first contribution, enter the second MPI_Gather() and send their

²If there are no overlapping bucket collective algorithms, then one could use the faster zero length single block memory descriptor instead of an independent block memory descriptor which saves a message header.

second contribution before the parent is ready for the second gather operation. One could either use separate portals, sequence numbers, or some other form of matching criteria to ensure that overlapping collective operations are handled properly.

4. One Sided Communications

A proposal for one sided communications is currently under consideration by the MPI-2 Forum. One sided communications is an extension to the communications mechanisms of MPI allowing for remote memory access (RMA) where the transfer of data from the memory of one process to the memory of another process occurs with only the explicit involvement of one of these processes [9]. This proposal hopes to provide an interface for taking advantage of the opportunities for high performance RMA on those systems that have dedicated RMA hardware, such as the Cray T3D [4], systems with communications coprocessors, such as the Intel Paragon, and on shared memory multiprocessor systems. The current proposal contains functions for initialization, remote memory reads and writes, atomic memory updates, remote synchronization, and message handlers. The initialization and RMA access functions provide the basis for doing one sided communications.

The design of Puma provides for doing efficient RMA operations. Because portals allow for writing into and reading out of (using reply portals) the memory of a remote process without the process' explicit involvement, Puma has the capability to do RMA communications easily and efficiently. Cache coherency is maintained for all incoming messages.

The current proposal includes two functions for initialization. The first is *MPI_RMA_init()* which exposes a window of memory to RMA communications and returns a communicator enabled for performing both RMA operations and normal MPI communications. The second is a routine for allocating special memory which is provided for those platforms where a different type of memory must be used for RMA. Puma provides the capability to write into any memory in an application's address space, so the RMA allocation function is equivalent to the standard *malloc()* function.

The current proposal has four functions for RMA access: *MPI_Put()*, *MPI_Get()*, *MPI_Iput()*, and *MPI_Iget()*. When used with the communicator returned by *MPI_RMA_init()*, the put functions perform a remote write of the data supplied at the origin process into the exposed window at the target process. The get functions perform a remote memory read of the exposed window at the target process, depositing the data into a supplied buffer on the origin process. The non-blocking versions return a request handle that may be used with any of the normal MPI wait or test

functions. These functions also contain an offset argument so that reads and writes can be initiated at an offset from the base of the window.

4.1. Design and Implementation

Figure 11 illustrates the portals used for RMA operations. Two portal table entries are used for RMA, one for puts and one for gets. In the *MPI_RMA_init()* function, the next available match list entry for the put portal is obtained. The first 32 match bits in this entry are set to the send context identifier in the RMA communicator. The second 32 match bits are set to a special tag value. A sender-managed single block memory descriptor referencing the RMA window is attached to the entry. Similarly, the next available match list entry for the get portal is obtained, and the match bits are set to the receive context on the RMA communicator and a special tag value. A sender-managed single block reply memory descriptor referencing the same RMA window is attached to the get match list entry.

The put functions send a message to the designated put portal on the target process with the destination match bits set to the send context of the communicator and the correct byte offset calculated from the offset arguments to the function. The put operation completes as soon as the message is sent. A blocking put requires no further action, while a non-blocking put must build a request handle that is immediately marked completed. Therefore, non-blocking puts have a degradation in performance over blocking puts. Put operations maintain pairwise ordering.

The get function begins by posting a receive for the reply message. Posting a receive is done exactly as if the receive were being posted for a normal MPI message, with a few exceptions. The matchbits for this receive are set to the receive context of the communicator and the special tag value, in order to avoid mixing RMA communications with regular MPI communications. Instead of attaching an independent block memory descriptor to the match entry, a single block descriptor can be used. A message is then sent to the designated get portal on the target process with the destination match bits set to the receive context of the communicator and the correct byte offset calculated from the offset arguments to the function. And in addition, the requested return portal is the local receive portal and the return matchbits are set appropriately. The blocking version of get is implemented by calling the non-blocking version and waiting for the request to complete.

5. Future Work

A concerted effort is being made to increase the performance of Puma to be at least that of its predecessor. MPI has yet to be tested under the various coprocessor modes

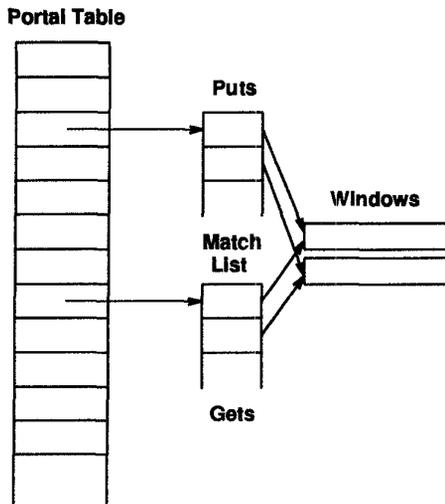


Figure 11. Put and Get Portals.

being developed, and some base functionality still needs to be implemented at the operating system level.

The combined block memory descriptor was designed to be used for operations with non-contiguous datatypes. However, the combined block has yet to be implemented. Consequently, non-contiguous datatypes are packed and unpacked into contiguous buffers. For one sided communications, each block in the datatype generates a separate message so that the offset can be used properly. Combined block memory descriptors will greatly reduce this cost.

For the ASCI/DOE TeraFLOPS machine, hybrid techniques will be incorporated into MPI collective operations in order to take advantage of the topology of the machine. In addition, once combined blocks are implemented, the collective operations will be modified to support non-contiguous datatypes.

Effort is nearly completed on a new ADI for MPICH [7]. The goal of this next generation ADI is to achieve lower latencies and remove as much overhead as possible, especially when handling messages with contiguous datatypes. Providing better support for multi-protocol devices and heterogeneous systems are additional goals. Work has already begun on moving this implementation to the new ADI.

6. Acknowledgments

Gratitude is expressed to Arthur B. Maccabe for his contribution to the design of MPI point-to-point communications and to Lance Mumma and Tramm Hudson for help in the design and implementation of the native Puma collective communications.

References

- [1] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Building a high-performance collective communication library. In *Supercomputing '94 Proceedings*, pages 107–116, November 1994.
- [2] M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library (intercom). In *Scalable High-Performance Computing Conference*, pages 357–364, May 1994.
- [3] L. Clark, I. Glendinning, and R. Hempel. The MPI message passing interface standard. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, 1994.
- [4] Cray Research, Inc. *Cray T3D System Architecture Overview, HR-04033*, March 1994.
- [5] W. Gropp and E. Lusk. *MPICH ADI Implementation Reference Manual*. Mathematics and Computer Science Division, Argonne National Laboratory, October 1994.
- [6] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.
- [7] W. Gropp and R. Lusk. MPICH working note: The second-generation ADI for the MPICH implementation of MPI. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory, February 1996.
- [8] A. B. Maccabe, K. S. McCurley, R. Riesen, and S. R. Wheat. SUNMOS for the Intel Paragon: A brief user's guide. In *Proceedings of the Intel Supercomputer Users' Group, 1994 Annual North America Users' Conference.*, pages 245–251, June 1994.
- [9] Message Passing Interface II Forum. *One Sided Communications*, April 1996. <http://www.cs.wisc.edu/lederman/mmpi2/mmpi2-report.ps.Z>.
- [10] P. Mitra, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Tutorial: Collective communication: Theory and practice (fast collective communication libraries, please). In *Intel Supercomputing '94 Proceedings*, June 1995. ISUG web page: <http://www.cs.sandia.gov/ISUG>.
- [11] L. Shuler, C. Jong, R. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. The Puma operating system for massively parallel computers. In *Proceeding of the 1995 Intel Supercomputer User's Group Conference*. Intel Supercomputer User's Group, 1995.