Portals 3.0: Protocol Building Blocks for Low Overhead Communication

Ron Brightwell Rolf Riesen
Sandia National Laboratories *
Scalable Computing Systems Department
PO Box 5800, Albuquerque, NM, 87185-1110
{bright,rolf}@cs.sandia.gov

Bill Lawry A. B. Maccabe
University of New Mexico
Computer Science Department
FEC 313, Albuquerque, NM, 87131
{bill,maccabe}@cs.unm.edu

Abstract

This paper describes the evolution of the Portals message passing architecture and programming interface from its initial development on tightly-coupled massively parallel platforms to the current implementation running on a 1792-node commodity PC Linux cluster. Portals provides the basic building blocks needed for higher-level protocols to implement scalable, low-overhead communication. Portals has several unique characteristics that differentiate it from other high-performance system-area data movement layers. This paper discusses several of these features and illustrates how they can impact the scalability and performance of higher-level message passing protocols.

*K*eywords– message passing, network protocol, os bypass, workstation cluster, massively parallel

1. Introduction

The advent of cluster computing over the last several years 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 [2], have made low-level message passing protocols and programming interfaces a popular area of research [33, 34, 26, 22, 14, 9, 28]. Most of these research activities have been focused on delivering latency and bandwidth performance as close to the hardware limitations as possible.

To some extent, the research on clusters of PCs with gigabit networking hardware is addressing many of the same problems that proprietary distributed-memory message passing parallel machines of the early 1990's faced. Despite the differences in hardware architecture between custom-built parallel machines and today's PC cluster, many of the issues with respect to delivering message passing performance to parallel applications are similar.

Research into high-performance message passing protocols and interfaces was begun at Sandia National Laboratories in collaboration with the University of New Mexico nearly ten years ago. This research addressed the poor performance and scalability of applications running on massively parallel machines containing thousands of processors. This research culminated in the Cougar lightweight kernel, which was deployed on the 9000+ processor Intel ASCI/Red machine installed at Sandia in early 1997. Later that same year, we began the Computational Plant [4] (Cplant™) project, which was an evolution of our system software research from custom, vendor-supplied systems to Linux-based PC clusters.

A key component of this evolution is the Portals data movement layer. This paper discusses the development of Portals from the lightweight kernel research through the current implementation in use on our 1792-node Cplant™ cluster. We will show that Portals provides an interface for implementing many of the features required for low-latency, high-bandwidth, low-overhead, scalable message passing on massively parallel distributed-memory computing platforms.

The rest of this paper is organized as follows. We begin with an introduction to Portals as they were shaped by our lightweight kernel research in Section 2. Section 3 describes the challenges with implementing Portals on a PC running the Linux operating systems. We continue in Section 4 by discussing the current Portals API and semantics. We outline several benefits of this API as compared to other similar research projects in Section 5. We conclude in Section 6 with a summary of this paper and outline our plans for future work in Section 7.

2. Puma Portals

Portals were an outcome of early research into high-performance message passing in the Sandia/University of New Mexico Operating System(SUNMOS) [17] on the

*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.

nCUBE and Intel Paragon machines. Portals were initially designed and implemented in the successor to SUNMOS, called Puma, to address the need for zero-copy message passing, where incoming messages are delivered directly into an application's address space without intermediate buffering by the operating system [18]. The Puma operating system [31] implemented the second generation of Portals (now called Portals 2.0), which extended the functionality of the original design and provided the basic building blocks for various high-level message-passing layers.

Puma was designed to take full advantage of the hardware architecture of compute nodes on the Intel Paragon and ASCI/Red. Compute nodes are composed of two processors, memory, and a high-speed network interface integrated on the memory bus. The Puma kernel delivers messages directly from the network into an application's address space with no intermediate buffering. The network interface is able to perform DMAs directly between user-space and the network.

Portals in Puma are data structures in an application's address space that determine how the kernel should respond to message-passing events. Portals allow messages to be delivered directly to the application without any intervention by the application process. In particular, the application process need not be the currently scheduled process or perform any message selection operations, such as tag matching, to process incoming messages. We refer to this feature as *application bypass*, since the application is not involved in the data transfer once it has been set up. We will discuss the benefits of application bypass in greater detail in Section 5.1.

The fundamental characteristics and semantics of Portals can be attributed to their origin on these massively parallel distributed memory machines. In particular, Portals are connectionless and provide protected, reliable, in-order delivery. They were designed to support multiple communicating processes per node and communication between processes created from different executables. Portals were also designed to efficiently support multiple protocols within the same process. Since the only way to communicate with a process on a compute node is via Portals, they had to support not only application message passing, but also I/O protocols to a remote filesystem, and protocols between the components of the parallel runtime environment.

Since Portals pre-dated the development of the MPI standard [20], multiple application-level message passing APIs were implemented on top of Portals, such as Intel's NX [27] interface and nCUBE's Vertex [23] interface. The MPI implementation for Portals in Puma [7] also utilized a high-performance collective communication library [1] implemented directly on Portals and contained a preliminary implementation of the MPI-2 [21] one-sided functions.

3. Portals in Linux

Despite our experience with the poor performance and scalability of full-featured UNIX kernels that motivated the research on lightweight kernels, the effort required to port and maintain these kernels on commodity PC hardware for the Cplant™ project was extensive. In using Linux on Cplant™, we hoped to leverage its portability and open source model. Linux allows us to have an operating system that runs well on the very latest commodity hardware, and the source code availability gives us the opportunity to manipulate the standard kernel to create an operating system that exhibits the important characteristics of our past lightweight kernels.

Our initial plan was to port Portals 2.0 to Linux, first as a Linux kernel module and then as a Myrinet Control Program (MCP) running on the Myrinet NIC. We hoped that the module implementation would allow for a rapid port of the parallel runtime environment and that the MCP implementation would eventually allow us to realize the performance benefits of application bypass that Portals in Puma provide.

Portals 2.0 in Linux was implemented via two kernel modules that work with a Sandia-developed MCP that runs on the LANai processor on the Myrinet interface card. The Portals 2.0 module is responsible for determining how incoming messages are processed. It reads the application process' memory and interprets the Portals data structures. The Portals module communicates information about message delivery to the RTS/CTS module, which is responsible for packetization and flow control. The RTS/CTS module communicates packet delivery information to the MCP, which is essentially a packet delivery device. Outgoing message data is copied into kernel memory, then copied into the Myrinet NIC. On the receive side, packets are copied from the Myrinet NIC into kernel memory, and then from kernel memory into the application's memory. All of these memory copies are overlapping, so we are able to achieve reasonable bandwidth due to packet pipelining. But since the module implementation was not our end goal, we put little effort into further optimizing this approach.

Soon after beginning the implementation of the Portals 2.0 kernel module, we discovered several problems with our approach. Most of these problems were a result of the lack of a functional API for Portals 2.0 and our limited knowledge of the internals of the Linux kernel. We discuss two of these problems below. For a more in-depth discussion of these limitations, see [6].

The lack of an API prevented us from moving Portals-related data structures out of user-space. Ideally, these data structures should be able to exist in user-space, kernel-space, or NIC-space – whichever provides the highest performance for the underlying network hardware.

The lack of an API also does not allow for pre-validation of user-space addresses. Because there is no API, the application process has no way to give the Puma kernel a destination address before a matching message arrives. Therefore, addresses of message destinations are validated when the message arrives and the appropriate user-space buffer is located. This strategy works well for Puma, which uses a physically contiguous memory management scheme where address validation is a simple bounds check and the translation from virtual to physical is simply an offset from the base physical address.

However, Linux only validates addresses for the currently running process, usually via a system call. Linux kernels beyond version 2.1 assume that a given user address is valid and perform the read or write operation. Should the address not be valid, the hardware will generate a page fault, and the kernel will gracefully recover. This method optimizes for the common case where the address is valid, and takes a significant performance hit when the address is not valid. This creates many problems for Portals 2.0, since address validation is not done from a system call for the currently running process.

It became clear that Portals 2.0 was designed to take advantage of the highly specialized features of the Puma kernel and that we would never be able to reach our performance goals with a Portals 2.0 MCP implementation for Linux. We decided to try to develop a functional API that would allow for the key message-passing data structures to exist in the most optimum address space while still providing the key message passing features of Portals 2.0.

In December of 1999, we released the first version of the Portals 3.0 message passing interface [5]. We implemented a reference implementation over TCP/IP, as well as an implementation that works with the existing RTS/CTS module for Myrinet. We have a port of MPICH version 1.2.0 over Portals 3.0, a port of MPI Software Technology's MPI/Pro® implementation of MPI over Portals 3.0, and the components of the Cplant™ parallel runtime system [3] have been ported to use Portals 3.0 as well. Portals 3.0 has been in production use on our large Cplant™ clusters since August of 2000. Our largest Cplant™ cluster is currently 1792 nodes, and has demonstrated 706 gigaFLOPS on the Linpack benchmark, placing it at number 30 on the November 2001 list of the Top 500 fastest supercomputers in the world [24].

We believe the Portals 3.0 API will allow us to achieve the functionality and relative performance for Linux and Myrinet that Portals 2.0 provided for Puma on ASCI/Red. A Portals 3.0 MCP implementation is currently in progress, and is achieving less than 20 μsec for a zero-length ping-pong latency test.

4. An Overview of the Portals API

Our primary goal in developing the Portals 3.0 API was to support an implementation on Cplant™. However, Portals 3.0 is an API that allows for different implementations on many different types of networking hardware. Some of this hardware is better suited to our performance and scalability goals than others. In this section, we provide an overview of the API, semantics, and characteristics that we believe are important for a high-performance, scalable message passing layer. See [5] for a more complete description of the API and semantics.

4.1. Scalability

The primary goal in the design of Portals is scalability. Portals are designed specifically for an implementation capable of supporting a parallel job running on the order of ten thousand nodes. Performance is critical only in terms of scalability. That is, the level of message passing performance is characterized by how far it allows an application to scale and not by how it performs in a two-node ping-pong benchmark.

Portals are designed to allow for scalability, but do not guarantee it. Portals cannot overcome the shortcomings of a poorly designed application program or overcome limitations in an underlying network transport layer. Applications that have inherent scalability limitations, either through design or implementation, will not be transformed by Portals into scalable applications. Scalability must be addressed at all levels. Portals are designed to not limit scalability.

To support scalability, the Portals interface maintains a minimal amount of state. Portals provide reliable, ordered delivery of messages between pairs of processes. They are connectionless: a process is not required to explicitly establish a point-to-point connection with another process in order to communicate. Moreover, all buffers used in the transmission of messages are maintained in user-space. The target process determines how to respond to incoming messages, and messages for which there are no buffers are discarded. That is, Portals are based on expected messages. Higher-level message passing layers that need support for unexpected messages, such as MPI, need to set aside a certain amount of space to receive unexpected messages. For many message passing systems, such as VIA [9], the amount of memory required for unexpected messages grows linearly with the number of connections. Portals allow for the amount of memory used for unexpected message buffers to be based on the needs and behavior of the application rather than based simply on the number of processes in a parallel job.

4.2. Communication Model

Portals combine the characteristics of both one-side and two-sided communication. They define a "matching put" operation and a "matching get" operation. The destination of a put (or send) is not an explicit address; instead, each message contains a set of match bits that allow the receiver to determine where incoming messages should be placed. This flexibility allows Portals to support both one-sided operations and traditional two-sided send/receive operations.

Portals allows the target to determine whether incoming messages are acceptable. A target process can choose to accept message operations from any specific process or can choose to ignore message operations from any specific process.

4.3. Data Movement

A Portal represents an opening in the address space of a process. Other processes can use a Portal to read (get) or write (put) the memory associated with the Portal. Every data movement operation involves two processes, the **initiator** and the **target**. The initiator is the process that initiates the data movement operation. The target is the process that responds to the operation by either accepting the data for a put operation, or replying with the data for a get operation.

In this discussion, activities attributed to a process may refer to activities that are actually performed by the process or *on behalf of the process*. The inclusiveness of our terminology is important in the context of *application bypass*. In particular, when we note that the target sends a reply in the case of a get operation, it is possible that reply will be generated by another component in the system, bypassing the application.

Figures 1 and 2 present graphical interpretations of the Portal data movement operations: put and get. In the case of a put operation, the initiator sends a put request message containing the data to the target. The target translates the Portal addressing information in the request using its local Portal structures. When the request has been processed, the target optionally sends an acknowledgment message.

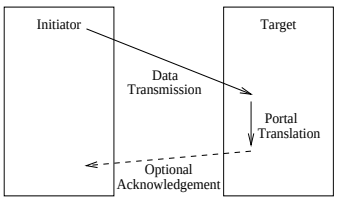

Figure 1. Portal Put (Send)

In the case of a get operation, the initiator sends a get request to the target. As with the put operation, the target translates the Portal addressing information in the request using its local Portal structures. Once it has translated the portal addressing information, the target sends a reply that includes the requested data.

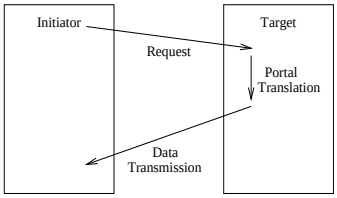

Figure 2. Portal Get

We should note that Portal address translations are only performed on nodes that respond to operations initiated by other nodes. Acknowledgments and replies to get operations bypass the Portals address translation structures.

4.4. Portal Addressing

One-sided data movement models (e.g., shmem [10], ST [32], MPI-2 [21]) typically use a triple to address memory on a remote node. This triple consists of a process id, memory buffer id, and offset. The process id identifies the target process, the memory buffer id specifies the region of memory to be used for the operation, and the offset specifies an offset within the memory buffer.

In addition to these standard address components, a portal address includes a set of match bits. This addressing model is appropriate for supporting one-sided operations as well as traditional two-sided message passing operations. Specifically, the Portals API provides the flexibility needed for an efficient implementation of the send/receive operations in MPI, which defines two-sided operations with one-sided completion semantics [1].

Figure 3 presents a graphical representation of the structures used by a target in the interpretation of a Portal address. The process id is used to route the message to the appropriate node and is not reflected in this diagram. The memory buffer id, called the **portal id**, is used as an index into the Portal table. Each element of the Portal table identifies a match list. Each element of the match list specifies two bit patterns: a set of "don't care" bits, and a set of "must match" bits. In addition to the two sets of match bits, each match list element has a list of memory descriptors. Each memory descriptor identifies a memory region and an optional event queue. The memory region specifies the memory to be used in the operation and the event queue is used to record information about these operations.

[1] The Progress Rule in MPI mandates local completion semantics for the standard non-blocking two-sided message passing operations

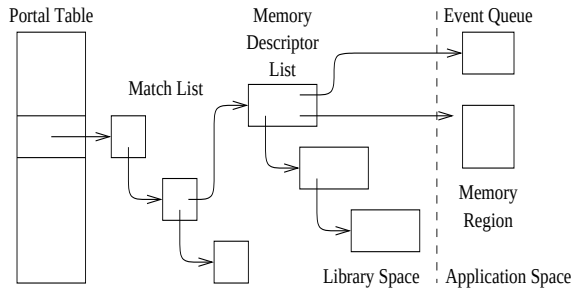

Figure 3. Portal Addressing Structures

Figure 4 illustrates the steps involved in translating a Portal address, starting from the first element in a match list. If the match criteria specified in the match list entry are met and the first entry in the memory descriptor list accepts the operation, the operation (put or get) is performed using the memory region specified in the memory descriptor. (Note, while the match list is searched for a matching entry, only the first element in the memory descriptor list is considered for the operation.) If the memory descriptor specifies that it is to be unlinked after a successful operation, it is unlinked from the list of memory descriptors. Next, if the memory descriptor is unlinked and this empties the memory descriptor list, the match entry will also be unlinked if its unlink flag has been set. Finally, if there is an event queue specified in the memory descriptor, the operation is logged in the event queue.

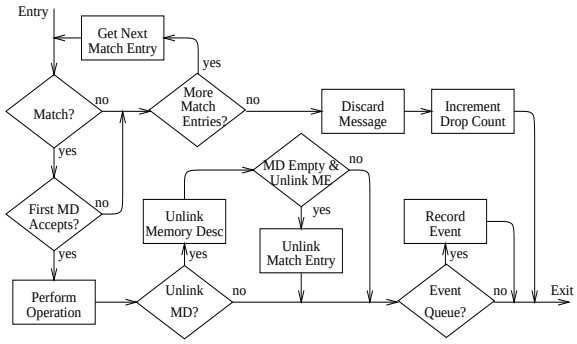

Figure 4. Portals Address Translation

If the match criteria specified in the match list entry are not met or the memory descriptor associated with the match list entry rejects the operation, the address translation continues with the next match list entry. If the end of the match list has been reached, the address translation is aborted and the incoming request is discarded.

4.5. Access Control

A process can control access to its Portals using an access control list. Each entry in the access control list specifies a process id and a Portal table index. The access control list is actually an array of entries. Each incoming request includes an index into the access control list (i.e., a "cookie" or hint). If the id of the process issuing the request doesn't match the id specified in the access control list entry or the Portal table index specified in the request doesn't match the Portal table index specified in the access control list entry, the request is rejected.

Process identifiers and Portal table indexes may include wildcard values to increase the flexibility of this mechanism. When the access control list is initialized, the entry with index zero enables access to all Portals for all processes in the same parallel application and the entry with index one enables access to all Portals for all system processes. The remaining entries are set to disable all other access.

Two aspects of this design merit further discussion. First, the model assumes that the information in a message header, the sender's id in particular, is trustworthy. In most contexts, we assume that the entity that constructs the header is trustworthy; however, using cryptographic techniques, we could easily devise a protocol that would ensure the authenticity of the sender.

Second, because the access check is performed by the receiver, it is possible that a malicious process will generate thousands of messages that will be denied by the receiver. This could saturate the network and/or the receiver, resulting in a *denial of service* attack. Moving the check to the sender using capabilities, would remove the potential for this form of attack. However, the solution introduces the complexities of capability management (exchange of capabilities, revocation, protections, etc).

4.6. The Semantics of Message Transmission

The Portals API uses four types of messages: put requests, acknowledgments, get requests, and replies. In this section, we describe the information passed on the wire for each type of message. We also describe how this information is used to process incoming messages.

Information	Description
operation	Indicates a put request
initiator	Local process id
target	Target process id
portal index	Target Portal table entry
cookie	Access control table entry
match bits	Matching criteria
offset	Offset within the target memory
memory desc	Local memory region for an ack
length	Length of the data
data	Payload

Table 1. Information Passed in a Put Request

4.7. Sending Messages

Table 1 summarizes the information that is transmitted for a put request. Most information that is transmitted is obtained directly from the put operation. Notice that the handle for the memory descriptor used in the put operation is transmitted even though this value cannot be interpreted by the target. A process can also signify that no acknowledgment is requested by using a special flag.

Information	Description
operation	Indicates an acknowledgment
initiator	Acking process id
target	Target process id
portal index	Echoed
match bits	Echoed
offset	Echoed
memory desc	Echoed
requested length	Echoed
manipulated length	Obtained from the operation

Table 2. Information Passed in an Acknowledgment

Table 2 summarizes the information transmitted in an acknowledgment. Most of the information is simply echoed from the put request. Notice that the initiator and target are obtained directly from the put request, but are swapped in generating the acknowledgment. The only new piece of information in the acknowledgment is the manipulated length, which is determined as the put request is satisfied.

Information	Description
operation	Indicates a get operation
initiator	Local process id
target	Target process id
portal index	Target Portal table entry
cookie	Access control table entry
match bits	Matching criteria
offset	Offset within target memory
memory desc	Local memory region for reply
length	Length of requested data

Table 3. Information Passed in a Get Request

Table 3 summarizes the information that is transmitted for a get request. Like the information transmitted in a put request, most of the information transmitted in a get request is obtained directly from the get operation. Unlike put requests, get requests do not include the event queue handle. In this case, the reply is generated whenever the operation succeeds and the memory descriptor must not be unlinked until the reply is received. As such, there is no advantage to explicitly sending the event queue handle.

Table 4 summarizes the information transmitted in a reply. Like an acknowledgment, most of the information is simply echoed from the get request. The initiator and target are obtained directly from the get request, but are swapped in generating the acknowledgment. The only new information in the acknowledgment are the manipulated length and the data which are determined as the get request is satisfied.

Information	Description
operation	Indicates an acknowledgment
initiator	Replying process id
target	Target process id
portal index	Echoed
match bits	Echoed
offset	Echoed
memory desc	Echoed
requested length	Echoed
manipulated length	Length of requested data
data	Payload

Table 4. Information Passed in a Reply

4.8. Receiving Messages

When an incoming message arrives on a network interface, the runtime system first checks that the target process identified in the request is a valid process that has initialized the network interface (i.e., that the target process has a valid Portal table). If this test fails, the runtime system discards the message and increments the dropped message count for the interface. The remainder of the processing depends on the type of the incoming message. Put and get messages are subject to access control checks and translation (searching a match list), while acknowledgment and reply messages bypass the access control checks and the translation step.

Acknowledgment messages include a handle for the event queue where the event should be recorded. Upon receipt of an acknowledgment, the runtime system only needs to confirm that the event queue still exists. Should the event queue no longer exist, the message is simply discarded and the dropped message count for the interface is incremented. Otherwise, the runtime system builds an acknowledgment event from the information in the acknowledgment message and adds it to the event queue. Event queues are circular, which prevents indexing out of bounds. The higher level protocol needs to insure that there are enough event slots and the rate of event consumption is able to keep up with the rate of event production to avoid missing events.

Reception of reply messages is also relatively straightforward. Each reply message includes a handle for a memory descriptor. If this descriptor exists, it is used to receive the message. A reply message will be dropped if the memory descriptor identified in the request doesn't exist or if the event queue in the memory descriptor has no space and is not null. In either of these cases, the dropped message count for the interface is incremented. These are the only reasons for dropping reply messages. Every memory descriptor accepts and truncates incoming reply messages, eliminating the other potential reasons for rejecting a reply message.

The critical step in processing an incoming put or get request involves mapping the request to a memory descriptor. This step starts by using the Portal index in the incoming request to identify a list of match entries. This list of match entries is searched in order until a match entry is found whose match criteria matches the match bits in the incom-

ing request and whose **first** memory descriptor accepts the request.

Because acknowledge and reply messages are generated in response to requests made by the process receiving these messages, the checks performed by the runtime system for acknowledgments and replies are minimal. In contrast, put and get messages are generated by remote processes and the checks performed for these messages are more extensive. Incoming put or get messages may be rejected because: the Portal index supplied in the request is not valid; the cookie supplied in the request is not a valid access control entry; the access control entry identified by the cookie does not match the identifier of the requesting process; the access control entry identified by the access control entry does not match the Portal index supplied in the request; or, the match bits supplied in the request do not match any of the match entries with a memory descriptor that accepts the request. In all cases, if the message is rejected, the incoming message is discarded and the dropped message count for the interface is incremented.

A memory descriptor may reject an incoming request for any of the following reasons: the memory descriptor has not been enabled for the incoming operation; or, the length specified in the request is too long for the matching memory descriptor and the truncate option has not been enabled;

5. Benefits of Portals

We believe that the Portals API provides many benefits over other message passing interfaces designed for clusters as well as proprietary distributed memory parallel platforms. In this section we outline several of these benefits.

5.1. Zero Copy, OS-Bypass and Application Bypass

In traditional system architectures, network packets arrive at the network interface card, are passed through one or more protocol layers in the operating system, and eventually copied into the address space of the application. As network bandwidth began to approach memory copy rates, reduction of memory copies became a critical concern. This concern lead to the development of zero-copy message passing protocols in which message copies are eliminated or pipelined to avoid the loss of bandwidth.

A typical zero-copy protocol has the NIC generate an interrupt for the CPU when a message arrives from the network. The interrupt handler then controls the transfer of the incoming message into the address space of the appropriate application. The interrupt latency, the time from the initiation of an interrupt until the interrupt handler is running, is fairly significant. To avoid this cost, some modern NICs have processors that can be programmed to implement part of a message passing protocol. Given a properly designed protocol, it is possible to program the NIC to control the transfer of incoming messages, without needing to interrupt the CPU. Because this strategy does not need to involve the OS on every message transfer, it is frequently called *OS-bypass*. Scheduled Transfer (ST) [32], Virtual Interface Architecture (VIA) [9], FM [16], GM [22], and Portals are examples of APIs and/or protocols that support OS-bypass.

However, many protocols that support OS-bypass still require that the application actively participate in the protocol for data to be transferred. This is especially true in the case of active message architectures, such as AM [34] and FM [26]. The fundamental concept of active messages is to integrate computation and communication. Conversely, the fundamental concept of Portals is to decouple the host processor from the network and allow data to flow with virtually no application processing.

Portals are aimed at significantly reducing receive overhead, which has been shown to have a greater impact on application performance [35, 19] than latency and bandwidth. Most studies that analyze receive overhead also do not account for the necessary protocol processing that higher-level protocols such as MPI must do as well.

5.2. MPI Progress

The limitations of OS-bypass with respect to overlap of computation and communication are most evident in implementations of higher-level protocols, such as MPI. Since MPI is typically the main message passing interface that OS-bypass protocols are targeting, it is interesting to analyze the effectiveness of OS-bypass protocols in supporting overlap for implementations of MPI.

MPI has asynchronous send and receive calls that allow high quality implementations the opportunity to overlap computation and communication. MPI also defines rules for how asynchronous communication operations make progress. The Standard states: "A communication is *enabled* once a send and a matching receive have been posted by two processes. The progress rule requires that once a communication is enabled, then either the send or the receive will proceed to completion. ... In particular, if the matching send is nonblocking, then the receive completes even if no complete-send call is made on the sender side. ... Similarly, a call ... that completes a send eventually returns if a matching receive has been started, even if no complete-receive call is made on the receiving side."

Every OS-bypass MPI implementation described in current literature [15, 29, 25, 11] requires application processing to move data. These implementations typically use a two-level protocol, where short messages are sent eagerly and long messages are sent using a rendezvous protocol. The short eager messages are buffered at the receiver and copied by the application into the appropriate receive buffer

after context and tag matching occur. In the rendezvous protocol, the sender sends a request to the receiver. This request is recognized by the application, the context and tag matching occur, and when the appropriate receive buffer is found, a message is sent back to sender indicating the exact location in memory where the data can be delivered. However, because the application must be involved in these transfers, the opportunity for significant overlap is lost.

The semantics of Portals 3.0 support the necessary progress engine for an MPI implementation without the need for explicit application intervention. Portals 3.0 provides the necessary building blocks for protocols to be implemented on NICs in a way that is not specific to MPI and is general enough to support several other higher-level data movement interfaces.

5.3. Demonstration of Application Bypass

In order to demonstrate the benefits of application-bypass, we conducted an experiment using a simple MPI program running on two nodes. Table 5 outlines the basic experiment. Both nodes iterate over this outline although only one node performs "work."

```
pre-post several non-blocking receives;
barrier;
post a batch of sends;
work (fixed loop iterations);
get time A;
wait for the batch of messages;
get Time B;
repeat;
```

Figure 5. Testing For Application Bypass

In our experiment, a batch consists of ten equal sized messages and timings were averaged by repeating the experiment several times. Also, the work, or fixed number of loop iterations, establishes a "work interval" during which parallel or concurrent message handling may proceed if allowed by the MPI implementation. We varied the work interval and timed how much of the message handling remained to be done after the work interval.

We ran this application bypass test on a 500 MHz Pentium III with a LANai 7.2 Myrinet NIC running GM 1.4 and MPICH/GM 1.2..7. We also ran the test on a Cplant™ cluster running the Portals 3.0 and RTS/CTS kernel modules with our Portals 3.0 port of MPICH 1.2.0.

Figure 6 presents the duration of waiting for messages as a function of work interval for MPICH/GM and MPICH/Portals 3.0 for 50 KB messages. MPICH/GM does not make any progress on message passing until we either wait for the messages or make other calls to the MPI library. In related testing not shown here, we introduced three

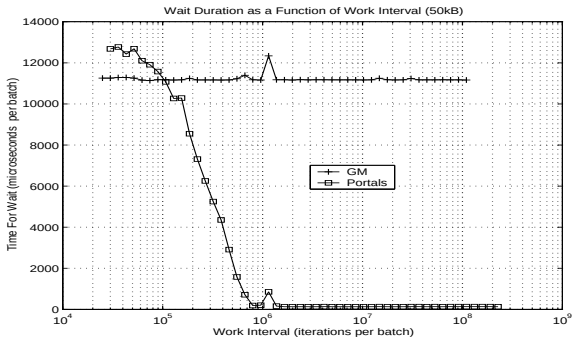

Figure 6. Progress in MPICH/GM and MPICH/Portals For 50KB Messages

calls to MPI_Test during the work interval and MPICH/GM could then make significant progress before the call to wait.

In contrast, the Portals implementation has a greater degree of application bypass and makes progress independently of the application making library calls. Given a large enough work interval, Portals can virtually complete message handling whereas MPICH/GM makes very little progress. We ran the same experiment on MPICH/Portals with the work interval having three test calls. The resulting graph of Portals data is essentially the same curve as shown in Figure 6.

This issue is not only specific to implementations of MPI on clusters using Myrinet [30], but have also been analyzed on proprietary parallel platforms as well [13]. The fundamental problem is the protocols underlying the MPI implementation that require the MPI library to be directly involved in the message selection activities. There are several ways to accomplish this, as suggested most recently in [30]: a separate thread can handle MPI progress, interrupts can be used to run MPI handler routines, or MPI-aware firmware can be placed on the NIC. However, the most popular approach is to ignore the progress rule and simply require the application to make frequent calls to the MPI library. This method allows the MPI library to continually check for outstanding communication operations and attempt to complete them.

Using a separate thread to facilitate the overlap of computation and communication has several disadvantages. The implementation of MPI must be designed to take advantage of using a separate thread for progress. Unfortunately, the most popular implementation of MPI, MPICH [12], was not. Care must be taken to reduce the amount of interference that a separate communication thread creates. Since nearly all message passing systems do not allow threads to be scheduled in response to message events, some overhead is incurred on the host processor.

Using interrupts to run MPI handler routines also has many drawbacks. Interrupts and context switching can be very expensive. In fact, it is this cost that motivated the OS Bypass approach in the first place.

Requiring the application to make frequent MPI calls in order to guarantee progress directly conflicts with the progress rule. In fact, this is the very situation that the progress rule intended to avoid. Some implementations explicitly state this limitation [8], but most do not. Along with the illegality of this approach, it also has several implementation drawbacks. Primarily, it forces non-portability of applications. Applications that are MPI compliant must be sprinkled with superfluous MPI calls in order for communications to make progress. The most effective places to insert calls are highly dependent on the underlying implementation. More importantly, some operations are atomic and MPI calls cannot be inserted at all. For example, posting a receive just before calling a BLAS library routine or calling an I/O operation. The BLAS library or I/O routines would need to be compiled to occasionally make unnecessary MPI calls. These are the very types of operations where overlap is most desirable and would be most effective.

Finally, implementing MPI-aware firmware on the NIC alleviates many of the problems with the other approaches. The NIC is able to make progress on MPI message operations independently from the application process, thus providing the opportunity to efficiently overlap computation and communication. However, this method is specific to MPI constructs and semantics. Ideally, the opportunity for overlap should be exploitable by all high-level message passing interfaces. The Portals API allows for NIC firmware to implement MPI semantics without being specific to MPI, so that other higher-level protocols can also reap the benefits of application bypass and reduced overhead.

The particular implementation of Portals 3.0 that we used for the above experiment is interrupt-driven, so it has the same drawbacks that an interrupt-driven implementation of MPI would have. However, the NIC-based implementation of Portals will address these limitations and still provide the desired benefits of application bypass.

6. Summary

This paper has described the evolution of the Portals message passing architecture and programming interface from its initial development on tightly-coupled massively parallel platforms to the current implementation running on a 1792-node commodity Linux cluster. The current generation Portals API provides the basic building blocks necessary for higher-level protocols to implement scalable, high-performance communication. The availability of programmable NICs with significant processing power makes it possible to implement Portals in such a way as to significantly reduce receive overhead, even for higher-level message passing layers such as MPI.

7. Future Work

As mentioned previously, work on the Portals MCP for Myrinet is currently under way. We expect to complete this work in early 2002 and be able to demonstrate the full performance advantages of the programming interface. We are also working on ports to other networking hardware, specifically Quadrics and programmable gigabit Ethernet cards. We are also considering porting Portals 3.0 into the Cougar lightweight kernel to validate our approach on a massively parallel tightly coupled platform.

We have also considered adding a few features to Portals since the initial release of the API. We would like to extend the API to support gather/scatter operations more efficiently, and we have had requests from filesystem implementors to extend the functionality of memory descriptor processing to more easily accommodate an in-kernel implementation.

8. Acknowledgments

The authors would like to acknowledge Tramm Hudson, Michael Levenhagen, Dena Vigil, and Riley Wilson for their contributions to this research.

References

[1] M. Barnett, S. Gupta, D. Payne, P. 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.

[2] N. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet-a gigabit-per-second local-area network. *IEEE Micro*, 15(1):29–36, February 1995.

[3] R. B. Brightwell and L. A. Fisk. Scalable Parallel Application Launch on Cplant™. In *Proceedings of the SC2001 Conference on High Performance Networking and Computing*, November 2001.

[4] R. B. Brightwell, L. A. Fisk, D. S. Greenberg, T. B. Hudson, M. J. Levenhagen, A. B. Maccabe, and R. E. Riesen. Massively Parallel Computing Using Commodity Components. *Parallel Computing*, 26(2-3):243–266, February 2000.

[5] R. B. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 Message Passing Interface. Technical Report SAND99-2959, Sandia National Laboratories, December 1999.

[6] R. B. Brightwell and S. J. Plimpton. Scalability and Performance of Two Large Linux Clusters. *Journal of Parallel and Distributed Computing - Special Issue on Cluster and*

Network-based Computing, 61(11):1546–1569, November 2001.

[7] R. B. Brightwell and P. L. Shuler. Design and implementation of MPI on Puma portals. In *Proceedings of the Second MPI Developer's Conference*, pages 18–25, July 1996.

[8] F. Chaussumier, F. Desprez, and L. Prylli. Asynchronous Communications in MPI – the BIP/Myrinet Approach. In J. Dongarra, E. Luque, and T. Margalef., editors, *Proceedings of the EuroPVM/MPI'99 conference*, number 1697 in Lecture Notes in Computer Science, pages 485–492, Barcelona, Spain, September 1999. Springer Verlag.

[9] Compaq, Microsoft, and Intel. Virtual Interface Architecture Specification Version 1.0. Technical report, Compaq, Microsoft, and Intel, December 1997.

[10] Cray Research, Inc. *SHMEM Technical Note for C, SG-2516 2.3*, October 1994.

[11] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. In *Proceedings of the Third MPI Developers' and Users' Conference*, pages 15–24, March 1999.

[12] 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, September 1996.

[13] J. B. W. III and S. W. Bova. Where's the Overlap? An Analysis of Popular MPI Implementations. In *Proceedings of the Third MPI Developers' and Users' Conference*, pages 1–6, March 1999.

[14] Y. Ishikawa, H. Tezuka, and A. Hori. PM: A High-Performance Communication Library for Multi-user Parallel Envrionments. Technical Report Technical Report TR-96015, RWCP, 1996.

[15] M. Lauria and A. A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[16] M. Lauria, S. Pakin, and A. A. Chien. Efficient Layering for High Speed Communication: Fast Messages 2.x. In *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 1998.

[17] A. B. Maccabe, K. S. McCurley, R. E. 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.

[18] A. B. Maccabe and S. R. Wheat. Message passing in PUMA. Technical report SAND93-0935, Sandia National Laboratories, 1993.

[19] R. P. Martin, A. M. Vahdata, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 85–97, June 1997.

[20] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8, 1994.

[21] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[22] Myricom, Inc. The GM Message Passing System. Technical report, Myricom, Inc., 1997.

[23] nCUBE Corporation. nCUBE-2 Programmers' Guide, 1990.

[24] Netlib. *Top 500 Supercomputers*, 1998. http://www.top500.org.

[25] F. O'Carroll, A. Hori, H. Tezuka, and Y. Ishikawa. The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network. In *ACM SIGARCH ICS'98*, pages 243–250, July 1998.

[26] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages(FM): Efficient, Portable Communication for Workstation Clusters and Massively Parallel Processors. *IEEE Concurrency*, 40(1):4–18, January 1997.

[27] P. Pierce. The NX message passing interface. *Parallel Computing*, 1993.

[28] L. Prylli and B. Tourancheau. BIP: A New Protocol Designed for High Performance Networking on Myrinet . *Parallel and Distributed Processing, IPPS/SPDP'98*, 1388:472–485, April 1998.

[29] L. Prylli, B. Tourancheau, and R. Westrelin. The Design for a High Performance MPI Implementation on the Myrinet Network. In *Proceedings of the 6th European PVM/MPI Users' Group*, pages 223–230, September 1999.

[30] L. Prylli and R. Westrelin. Current Issues in Available Implementations on Myrinet. In *Proceedings of the First Myrinet Users Group Conference*, pages 149–155, September 2000.

[31] P. L. Shuler, C. Jong, R. E. Riesen, D. van Dresser, A. B. Maccabe, L. A. Fisk, and T. M. Stallcup. 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.

[32] Task Group of Technical Committee T11. Information Technology - Scheduled Transfer Protocol - Working Draft 2.0. Technical report, Accredited Standards Committee NCITS, July 1998.

[33] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth SOSP*, pages 40–53, December 1995.

[34] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communications and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

[35] F. Wong, R. Martin, R. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proceedings of the SC99 Conference on High Performance Networking and Computing*, November 1999.