Implementation and Performance of Portals 3.3 on the Cray XT3

Ron Brightwell Trammell Hudson* Kevin Pedretti Rolf Riesen Keith D. Underwood
Sandia National Laboratories†
PO Box 5800
Albuquerque, NM 87185-1110

Abstract

The Portals data movement interface was developed at Sandia National Laboratories in collaboration with the University of New Mexico over the last ten years. Portals is intended to provide the functionality necessary to scale a distributed memory parallel computing system to thousands of nodes. Previous versions of Portals ran on several large-scale machines, including a 1024-node nCUBE-2, a 1800-node Intel Paragon, and the 4500-node Intel ASCI Red machine. The latest version of Portals was initially developed for an 1800-node Linux/Myrinet cluster and has since been adopted by Cray as the lowest-level network programming interface for their XT3 platform. In this paper, we describe the implementation of Portals 3.3 on the Cray XT3 and present some initial performance results from several micro-benchmark tests. Despite some limitations, the current implementation of Portals is able to achieve a zero-length one-way latency of under six microseconds and a uni-directional bandwidth of more than 1.1 GB/s.

1. Introduction

Portals 3.3 [6] is an evolution of the user-level network programming interface developed in early generations of the lightweight kernel operating systems [11, 13] created by Sandia National Laboratories and the University of New Mexico for large-scale massively parallel distributed memory parallel computers. Early versions of Portals did not have functional programming interfaces, which severely hampered an implementation for intelligent or programmable networking hardware. In 1999, Sandia and UNM developed a functional programming interface for Portals that retained much of the desired functionality but removed the dependence on a lightweight kernel environment. This third generation implementation of Portals was specifically intended to support an 1800-node Linux cluster using the Myrinet [2] network fabric. In addition to supporting the Cplant [4] clusters, Portals was also adopted by Cluster File Systems, Inc. as the transport layer for their Lustre file system [7].

In 2002, Cray chose to use Portals 3.3 as the lowest-level programming interface for their custom designed SeaStar [1] network interface on the XT3 system. The SeaStar interconnect was designed specifically to support a large-scale distributed memory scientific computing platform. The network performance and scalability requirements for XT3 were ambitious when they were first proposed for the initial installation at Sandia, called Red Storm. The SeaStar network is required to deliver 1.5 GB/s of network bandwidth per direction into each compute node and 2.0 GB/s of link bandwidth per direction. This yields an aggregate of 3.0 GB/s into each node. The one-way MPI latency requirement between nearest neighbors is 2 μs and is 5 μs between the two furthest nodes.

In this paper, we describe the implementation of Portals 3.3 for the SeaStar network interface on the XT3 and provide an initial performance evaluation using low-level micro-benchmark tests. Despite the fact that the software environment is currently under active development, the initial performance results are promising. Current bandwidth performance is higher than what can be achieved using a single interface of any current commodity interconnect.

The rest of this paper is organized as follows. The next section provides an overview of the SeaStar network hardware. Section 3 discusses the software environment and the implementation of Portals. In Section 4 we describe the firmware on the SeaStar that implements Portals. A description of the test environment is presented in Section 5, while performance results are shown in Section 6. Relevant conclusions of this paper are presented in Section 7.

*Under contract to Sandia via OS Research, Inc.

†Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

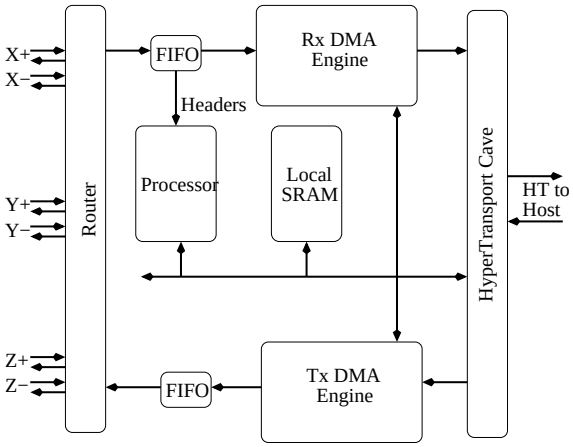

Figure 1. Basic SeaStar block diagram

2. Hardware

The Cray SeaStar ASIC [1] in the XT3 system was designed and manufactured by Cray, Inc. In a single chip, it provides all of the system's networking functions as well as all of the support functions necessary to provide reliability, availability, and serviceability (RAS) and boot services. The basic block diagram is shown in Figure 1. Independent send and receive DMA engines interact with a router that supports a 3D torus interconnect and a HyperTransport cave that provides the interface to the Opteron processor. An embedded PowerPC processor is also provided for protocol offload and programming the DMA engines.

The DMA engines provide support for transferring data between the network and memory while providing support for the message packetization needed by the network. They also provide hardware support for an end-to-end 32 bit CRC check. This augments the extremely high reliability provided by a 16 bit CRC check (with retries) that is performed on each of the individual links.

The physical links in the 3D topology support up to 2.5 GB/s of *data payload* in each direction. This accounts for overhead in both the 64 byte packets used by the router and the reliability protocol on the individual links. The interface to the Opteron uses 800 MHz HyperTransport, which can provide a theoretical peak of 3.2 GB/s per direction with a peak payload rate of 2.8 GB/s after protocol overheads (and a practical rate somewhat lower than that). The table-based routers provide a fixed path between all nodes, resulting in in-order delivery of packets.

The PowerPC processor is designed to offload protocol processing from the host processor. It is a dual-issue 500 MHz PowerPC 440 processor with independent 32 KB instruction and data caches. It must program the DMA engines since transactions across the HyperTransport bus require too much time to allow the host processor to program these engines. On the receive side, the PowerPC is also responsible for recognizing the start of new messages and reading the new headers. Finally, the PowerPC must recognize DMA completion events. To hold local state and handle interactions with the host, the PowerPC has 384 KB of scratch memory. This memory is protected by ECC complete with scrubbing (to clear memory for switching from classified to unclassified use). In this context, a certain portion of the network management *must* be offloaded to the network interface, but there is an opportunity to offload the majority of network protocol processing as well.

3. Portals

The Portals [5] network programming interface was developed jointly by Sandia National Laboratories and the University of New Mexico. Portals began as an integral component of the SUNMOS [11] and Puma [13] lightweight compute node operating systems. In 1999, an operational programming interface was created for Portals so that it could be implemented for intelligent and/or programmable network interfaces outside the lightweight kernel environment [6]. Portals is based on the concept of elementary building blocks that can be combined to support a wide variety of upper-level network transport semantics.

Portals provides one-sided data movement operations, but unlike other one-sided programming interfaces, the target of a remote operation is not a virtual address. Instead, the ultimate destination of a message is determined at the receiving process by comparing contents of the incoming message header with the contents of Portals structures at the destination.

3.1. Reference Implementation

An initial reference implementation of Portals 3.3 was done by Sandia in 1999. This reference implementation was designed to be easily portable between different network and kernel architectures, with a mix of user-space, kernel-space, and network interface-space implementations as possible targets. As of this writing, there exist implementations of nearly all possible permutations of address spaces. It was not, however, designed to allow these address spaces to be mixed at runtime – the target was always a small number of applications per node all in the same address space.

The implementation of Portals for the SeaStar is based on this reference implementation developed by Sandia. The XT3 system presented a novel requirement in that it needed to support several different systems with the same code base to enable both user-level applications and kernel-level services for the two operating systems used on XT3 – the Catamount lightweight compute-node kernel [10] and Linux:

- Catamount compute nodes with “generic” applications and “generic” services
- Catamount compute nodes with “accelerated” applications and “generic” services
- Linux service nodes with many “generic” services and kernel-level Lustre service
- Linux compute nodes with single user-level application

The firmware on the SeaStar for each of these is exactly the same, so it must be generic enough to support moving data into both user-level data buffers, kernel buffers and delivering notifications to user-level event queues and to a single kernel-managed event queue. The design and implementation of the firmware is discussed in detail in [12]. We provide a summary in Section 4.

The reference implementation has a network abstraction layer (NAL) that allows all implementations to share the same Portals library code, with the NAL providing the user API to library to network communications paths. In practice, each of these NALs share a large amount of code for moving data between address spaces. For instance, all Linux NALs that run user-level applications with the Portals library in kernel-space will need to use the same address validation routines and routines to copy data between user- and kernel-space.

3.2. Cray Bridge

Since each of the different cases for the XT3 differs only in the communication path between the user-level API and the Portals library code, they can share all of the library to network interface methods of the NAL. Unfortunately, the existing reference implementation was not designed with this sort of sharing in mind. To abstract the separate communication paths, Cray designed a “bridge” layer that sits atop the NAL and overrides the methods for moving data to and from API and library-space, as well as the address validation and translation routines.

Three bridges have been implemented:

- qkbridge for Catamount compute node applications
- ukbridge for Linux user-level applications
- kbridge for Linux kernel-level applications

Every NAL for Catamount, for instance, would share the same qkbridge code. Since the bulk of the API to library NAL is in the shared routines, this design allows very rapid development of new library to network NALs.

The ukbridge and kbridge are able to run simultaneously on a single node. Since both bridges use the same library to network interface communication paths, both kernel-level applications and user-level applications are able to cleanly share the network interface.

The latest Portals reference implementation now supports a similar interface abstraction based on the success of the bridge approach. It is hoped that this will allow more rapid development of new NALs. Thanks to the removal of much of the complexity in writing a new NAL, we hope that this ease of development will allow Portals to become more widely used on different platforms.

3.3. SeaStar NAL

There are two primary constraints to consider in an implementation of Portals for the SeaStar. The first is the limited amount of memory available in the SeaStar chip. Limiting the design to only the 384 KB of SRAM that could be provided internally on the SeaStar helps to improve reliability and reduce cost. Unfortunately, it also makes the offload of the entire Portals functionality somewhat challenging. The second constraint is the lack of any facilities to manage the memory maps for the small pages used by Linux.

In light of these constraints, the initial design of Portals for the SeaStar places relatively little functionality on the network interface. The network interface is primarily responsible for driving the DMA engines and copying new message headers to the host. When these new headers have been copied to the host, the host is interrupted to perform the Portals processing. In response, the host pushes down commands for depositing the new message. Similarly, on the transmit side, the host pushes commands to the network interface to initiate transfers. In both cases, the PowerPC is responsible for interpreting the commands and driving the DMA engines. When a message completes, the PowerPC must again interrupt the host to allow it to post the appropriate Portals event. All of this is handled by a tight loop that checks for work on the network interface and then check for work from the host.

The commands from the host to the network interface take different forms depending on the operating system. Under Linux, the host is responsible for pinning physical pages, finding appropriate virtual to physical mappings for each page, and pushing all of these mappings to the network interface. In contrast, Catamount maps virtually contiguous pages to physically contiguous pages. This means that a single command is sufficient to allow the network interface to feed all necessary commands to the DMA engine.

The SeaStar NAL, or SSNAL, implements all of the entry-points required by a Portals NAL, including functions for sending and receiving messages. Additionally, SSNAL provides an interrupt handler for processing asynchronous events from the SeaStar. In this way,

the platform-independent Portals library code can access the SSNAL through the common NAL interface and the SeaStar firmware can access platform-independent Portals functions (e.g., Portals matching semantics) through the interrupt handler. Our measurements indicate that a NULL-trap into the Catamount kernel requires approximately 75 ns of overhead—not a significant source of overhead. Interrupts, on the other hand, are very costly, requiring at least 2 μs of overhead each. Clearly, it will be necessary to eliminate all interrupts from the data path in order to meet the performance requirements of the XT3.

In the future, a new implementation of Portals will be created to supplement the existing implementation. Much of the Portals library functionality, including matching, will be offloaded to the SeaStar firmware. This will allow arriving messages to be immediately processed, rather than waiting for the host to determine what actions to take. This implementation, referred to as *accelerated mode*, will enable user-level Portals clients to post commands directly to the firmware, without performing any system calls. Asynchronous events, such as Portals completion events, will be processed by polling when the user-level library is entered. The existing implementation, or *generic mode*, will continue to be necessary and will run side-by-side with the accelerated implementation. Limited network interface resources allow only a small number of accelerated-mode clients per node. Additionally, Linux nodes will continue to use generic-mode for the foreseeable future because accelerated mode will not support non-contiguous message buffers.

4. Portals Firmware

This section describes the operation of the C-based SeaStar firmware that Sandia has developed. This firmware provides the low-level support needed to implement the Portals message passing API. It is based on the firmware originally provided for the system by Cray, Inc. The Cray version was written in assembly. We have found that a C version is much easier to modify and debug, but has no worse performance than the assembly version. The Cray team and our team are working together to combine the two code bases and move forward with a single version.

The C firmware currently consists of 3,434 source lines of C code and 253 source lines of assembly code, according to Wheeler's SLOCCount tool [8]. When compiled with GCC 4.0 using optimization level three (-O3), the resulting firmware image is 22 KB in size.

4.1. General Architecture

Figure 2 shows a high-level view of the host interface to the C firmware. On the host, there are a number of processes that use the Portals API to send and receive messages. In Section 3.1 we explained that these processes are split into two groups, termed *generic* and *accelerated*. Generic processes forward all of their Portals API calls to the OS kernel, which multiplexes them to a single firmware mailbox. Accelerated processes, on the other hand, send some of their Portals API commands directly to a dedicated firmware mailbox. Such commands are said to be *offloaded* to the network interface. Some commands, such as those related to process initialization, cannot be offloaded and are always forwarded to the OS kernel.

The firmware processes commands that it receives in its mailboxes. Each mailbox contains a command and result FIFO. The host posts commands to the command FIFO by incrementing the tail index in the network interface's mailbox structure. If the command returns a result, the host busy-waits until the firmware posts the result to the result FIFO. The use of FIFOs allows the host to post multiple commands before waiting for a result in some cases. In particular, commands that do not return an immediate result (e.g., transmit message[1]) can be efficiently streamed to the firmware.

Each accelerated process and the generic Portals implementation in the kernel contain an Event Queue (EQ) for the firmware to post asynchronous events into. Examples of asynchronous events are "message transmit complete" and "message reception complete". Accelerated processes poll the EQ, if necessary, when the user-level Portals library is entered. The generic Portals implementation in the kernel is interrupt driven and only checks the EQ when the firmware has raised an interrupt. In order to reduce the number of interrupts, the Portals interrupt handler processes all of the new events in the generic EQ each time it is invoked. Individual events are small enough that they can be posted atomically by the firmware, allowing the host to simply read the next EQ slot to determine if a new event has arrived.

Limited network interface resources and OS limitations prevent all processes from operating in accelerated mode. Typically, there will be a small number of accelerated processes (one or two on each Catamount compute node) and the remaining processes will operate in generic mode. Supporting accelerated mode for Linux processes is particularly difficult because of memory paging—accelerated mode relies on message buffers being physically contiguous in memory. Catamount places application memory in physically contiguous regions so it is straightforward to support accelerated mode.

Currently, only generic mode has been fully implemented in the C firmware. It was implemented first because

[1] Transmit commands can be thought of as returning a result much later in the form of a "message transmit complete" event in the process' event queue, but this may be hundreds of microseconds after the command was posted, so it is not efficient to busy wait for completion.

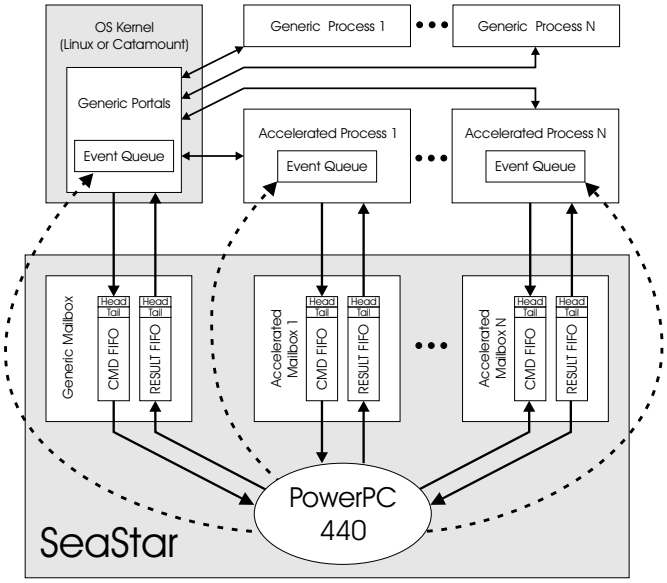

Figure 2. Firmware host interface

it was absolutely necessary for Linux nodes, where there are often many Portals clients, and it would function correctly for Catamount nodes until accelerated mode could be completed. Additionally, since generic mode Portals operates in the OS kernel and is interrupt driven, it allowed Cray to reuse an existing reference implementation of Portals provided by Sandia. Accelerated mode requires that the offloaded portions of Portals be reimplemented for the SeaStar. Much of the infrastructure for accelerated mode is already in place and we are actively working to complete it.

4.2. Data Structures

The firmware manages a number of data structures needed to transmit and receive messages. Figure 3 depicts an abstract view of the most important of these structures. First, there is one network interface Control block that contains global information concerning the entire firmware. Next, the accelerated host process and the generic Portals implementation in the OS kernel each have a dedicated process structure and mailbox structure allocated to them. These structures are one-to-one mapped and are split due to caching requirements—the mailbox must be un-cached so that coherency is maintained with the host while the data in the process structure is accessed only by the firmware so it can be stored in cached memory. Each process structure has a pool of pendings, split into upper and lower portions, that are used to track in-progress message transmissions and receptions. Finally, each node that the firmware is sending a message to or receiving a message from has a source structure allocated to it. There is one pool of source structures for the entire firmware (i.e., all processes on each node).

Each pending is split into lower and upper portions, which are one-to-one mapped. The lower pending structure is located in cached SeaStar local SRAM and contains all of the information needed to progress and complete the message it represents. The upper pending structure is located in host memory and contains all of the information needed by the host regarding the message. In normal operation, the firmware never reads data from the upper pending structure because doing so requires a high latency round-trip across the HyperTransport link. The firmware does write information that is needed by the host into the upper pending. The upper pending structures are stored in cached host memory and are automatically kept coherent with respect to firmware writes by the Opteron's memory controller.

Every firmware-level process has two pools of pending structures, one managed by the firmware and the other managed by the host. The firmware managed pool is used for message receptions. When a new message arrives, the firmware allocates a pending from the target process' `RX pending free list`. The host managed pool is used for message transmissions. To prepare to send a message, a host process (i.e., an accelerated process or the generic Portals implementation in the kernel) allocates a pending structure from a free list that it maintains.

There is no dynamic allocation of any data structures by the firmware. All structures are pre-allocated at initialization time and inserted into free lists or slab caches, from which they may be rapidly allocated for use. While this introduces compile time constraints on the number of outstanding messages, in practice sizing these constants has not

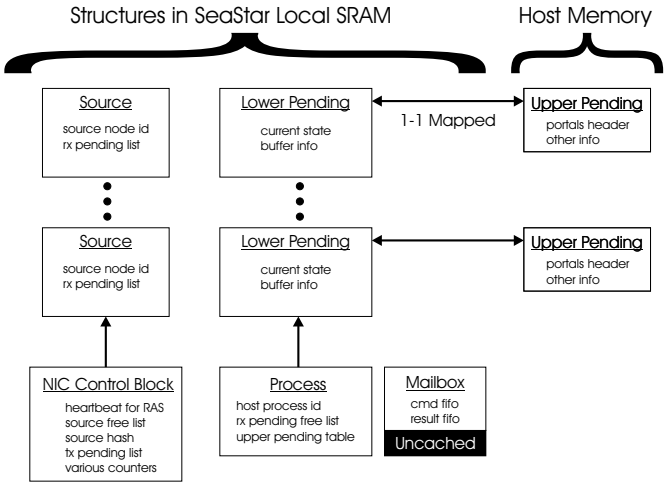

Figure 3. Firmware Data Structures

been too difficult. Resource exhaustion is addressed more fully at the end of Section 4.3.

The two primary consumers of SeaStar local SRAM are the source structures and the lower pending structures. The memory occupied by these structures can be calculated by

$$M = (S * S_{size}) + \sum_{i=1}^{N} (P_i * P_{size})$$

where S is the number of sources, S_{size} is the size of each source structure, N is the number of firmware-level processes, P_i is the number of pendings associated with process i, P_{size} is the size of each pending structure, and M is the SRAM occupied. For the current firmware, there are 1,024 global source structures and 1,274 pending structures allocated to the generic process (N is currently 1). These structures are small enough that several more similarly sized pending pools can be supported for additional firmware-level processes.

4.3. Firmware Processing

When idle, the firmware executes in a tight polling loop waiting for events. When an event occurs, the corresponding event handler is dispatched. The firmware is single threaded so handlers execute until they return, at which point a new event can be processed.

In order to transmit a message, the host must setup the message header and then send a transmit command to the firmware. First, the host allocates a pending structure from the pool that it manages (i.e., the transmit pool). The host then stores the Portals header in the upper portion of the pending structure. Next, the host sends a transmit command to the firmware, including the message's pending ID, target node ID, payload address in main memory, and the number of bytes to transmit. If the message buffer is not physically contiguous, the host must pre-compute the commands for the TX DMA engine and pass them to the firmware. DMA commands for physically contiguous messages are generated by the firmware.

When the firmware receives the transmit command, it looks up the lower pending structure using the pending ID that the host pushed down and initializes it. If there is no source structure for the destination node, a new one is allocated and initialized. The lower pending structure is then enqueued at the tail of the `TX pending list` in the control block. All transmits, regardless of destination or process type, are serialized through a single TX FIFO.

Once the pending reaches the head of the list, the firmware programs the TX DMA engine to transmit the message. The header is first DMA'ed out of the upper pending, followed by the payload DMA'ed directly from main memory.[2] If the message does not fit into the TX FIFO, the transmit state machine will yield and return to the main loop until there is more room in the FIFO. Finally, when the message has been completely sent, the firmware unlinks the lower pending from the `TX pending list` and posts a completion event to the host process' event queue. This completes the transmit from the firmware's perspective. The host posts the Portals completion event to the application and then returns the pending to its free list.

The RX DMA engine notifies the firmware when a new message arrives from the network. In response, the firmware inspects the message header to determine the source node ID and the target host process ID. The source node ID is used to retrieve the corresponding source struc-

[2] This is often referred to as zero-copy.

ture from a hash table of active sources. If no source structure is found, the firmware allocates a new one from its free list and inserts it into the hash table. The host process ID in the message header is used to look up the target firmware-level process. Once identified, the firmware allocates a pending from the target process' `RX pending free list`.

At this point, the future actions of the firmware depend on whether the target process is a generic process or an accelerated process. For generic processes, the firmware writes the entire Portals header into the upper pending, posts an event to the generic Portals event queue on the host, and then raises an interrupt. When the host receives the interrupt, it reads the event from the event queue and uses it to lookup the upper pending structure containing the Portals header. The header is then used to perform Portals matching on the host. Once the target memory descriptor has been identified, the host sends a receive command to the firmware, including the message's pending ID, the payload address in main memory, and the number of bytes to receive (and implicitly the number of bytes to discard). Like the transmit case, message buffers that are not physically contiguous have to have their receive DMA engine commands pre-computed by the host. The firmware uses the target buffer information in the receive command to setup the lower pending structure.

For accelerated processes, Portals matching is performed by the firmware. Therefore, there is no need for the firmware to raise an interrupt to ask the host where to put an incoming message. Once the target memory descriptor has been matched, the lower pending structure can be setup immediately. Like the generic case, the firmware writes the entire Portals header into the upper pending structure. This information is needed by the user-level Portals library when the firmware posts the message reception complete event.

Once the lower pending structure has been setup, the firmware links it to the tail of the target source structure's `RX pending list`. When the pending reaches the head of this list, the firmware programs the receive DMA engine to deposit the message directly into the target buffer in host memory. Once complete, the firmware posts a completion event to the host process' event queue. The host then uses the information stored in the upper pending to post the Portals completion event. Finally, the host sends a release pending command to the firmware to indicate that it is done with the upper pending structure and that the firmware can return the pending to the appropriate free list. This completes the receive from the firmware's perspective.

Unlike message transmission, there can be multiple receives in progress simultaneously—one from each source node. The packets of multiple incoming message streams arrive interleaved from the network. Normally, the RX DMA engine can transparently handle de-multiplexing the interleaved packets to the correct target buffers, based on the commands programmed by the firmware. In exceptional cases, there may be too many incoming messages for the RX DMA engine to handle. This is treated as a resource exhaustion case, described below.

There are a number of network interface-level resources that can be exhausted. For example, there may not be an unused pending structure available to handle a new message. Similarly, there may be too many sources trying to send to a node simultaneously. When this occurs, the firmware should become involved to resolve the situation.

The C firmware currently assumes that resource exhaustion does not occur. This has been sufficient to run several of Sandia's applications on approximately 7,700 nodes of Red Storm, which at the time of this writing is the maximum sized system partition that we have had access to. We have carefully monitored firmware resource usage and have never observed anything approaching dangerous levels. However, we expect that production-level use will occasionally trigger resource exhaustion. We are currently working on a simple go-back-n protocol to resolve resource exhaustion gracefully. The current approach is to panic the node, which results in application failure.

5. Test Environment

5.1. Platform

The platform used for our experiments is the 10,368-processor Red Storm machine at Sandia. This machine is a slightly specialized version of the commercial XT3 product. It differs from the XT3 in that the network is not a torous in all three directions. In order to support easily switching portions of the machine between classified and unclassified use, special switching cabinets were created for Red Storm. This capability and the limitation of cable lengths only allow the network to be torous in the z-direction. Each node in Red Storm has a 2.0 GHz AMD Opteron with 4 GB of main memory.

We present performance results using two different implementations of MPI for the XT3. The first is a port of MPICH 1.2.6 for Portals 3.3 developed by Sandia. The second is the Cray supported version of MPICH2 [9]. A detailed description of these implementation can be found in [3].

5.2. Benchmarks

In order to measure and compare the performance of Portals and MPI, we use the NetPIPE [14] benchmark. We developed a Portals-level module for NetPIPE version 3.6.2. This module creates a memory descriptor for receiving messages on a Portal with a single match entry attached. The

memory descriptor is created once for each round of messages that are exchanged, so the setup overhead for creating and attaching a memory descriptor to a Portal table entry is not included in the measurement. Rather than choosing a fixed message size interval and fixed number of iterations for each test, NetPIPE varies the message size interval and number of iterations of each test to cover a disparate set of features, such as buffer alignment. NetPIPE also provides a performance test for streaming messages as well as the traditional ping-pong message pattern. The Portals module that was developed for NetPIPE allows for testing put operations and get operations for both uni-directional and bi-directional tests and for uni-directional streaming tests for gets and puts. In the next Section, we compare the results from the Portals module with the existing MPI module in NetPIPE.

6. Results

Figure 4 shows latency performance for Portals put and get and the two implementations of MPI. One-byte latency is 5.39 μs, 6.60 μs, 7.97 μs, and 8.40 μs respectively. A significant amount of the current latency is due to interrupt processing by the host processor. At 12 bytes we see the results of a small message optimization currently in the firmware. Because 12 bytes of user data will fit in the 64 byte header packet, these 12 bytes can be copied to the host along with the header. This allows the new message and message completion notification to be delivered simultaneously and saves an interrupt. For longer messages, the combination of the independent progress semantics of Portals with the processing of the message headers on the host requires that two interrupts be used — one to have the header processed and one to post a completion notification to the application. In the fully offloaded implementation, both interrupts will be eliminated as the network interface will process headers and will write completion notifications directly into process space.

Figure 5 shows uni-directional bandwidth performance. The bandwidth for put tops out at 1108.76 MB/s for an 8 MB message. The bandwidth curves are fairly steep, with half the bandwidth for a unidirectional put being achieved at a message of around 7 KB. The MPI bandwidth is only slightly less, with both MPI implementations achieving the same performance.

Figure 6 shows the streaming bandwidth performance. As expected, the graph is steeper for this curve than the ping-pong bandwidth results. Half bandwidth for this benchmark is achieved at around a message size of 5 KB. Again, the MPI implementations have similar performance. We can also see that the streaming test has a much greater impact on the performance of the get operation, which is a blocking operation (for this benchmark) that cannot be

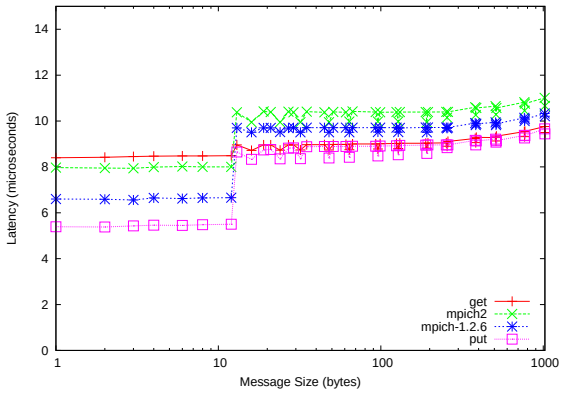

Figure 4. Latency performance

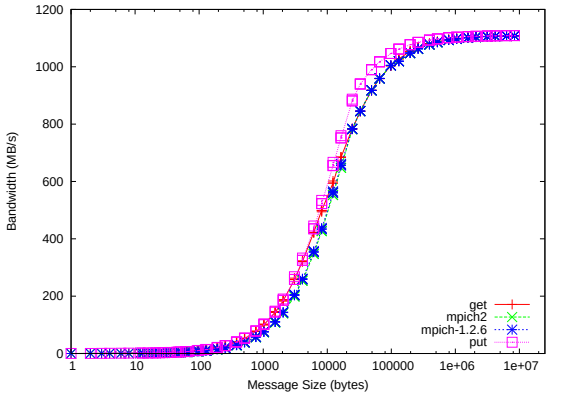

Figure 5. Uni-directional bandwidth performance

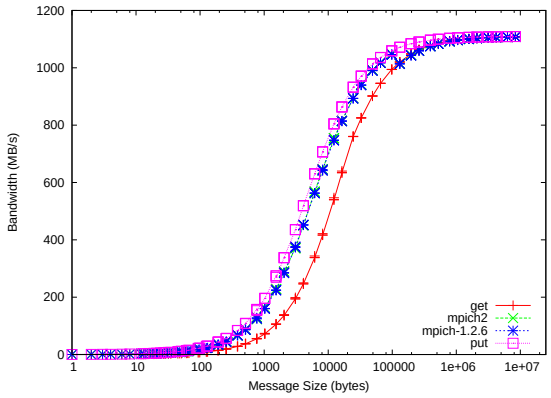

Figure 6. Streaming bandwidth performance

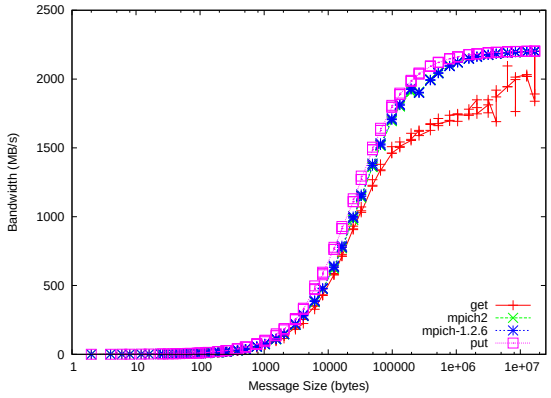

Figure 7. Bi-directional bandwidth performance

pipelined.

Figure 7 shows bi-directional bandwidth performance. The performance of the put tops out at 2203.19 MB/s for an 8 MB message, while both MPI implementations achieve only slightly less. This test shows that the SeaStar is able to sustain its unidirectional bandwidth performance when sending as well as receiving. For all of the bandwidth tests, we expect a dramatic decrease in the point at which half bandwidth is achieved as processing is offloaded from the host and the costly interrupt latency is eliminated.

7. Summary

This paper has described the implementation and performance of Portals 3.3 for the Cray SeaStar, the custom interconnect developed by Cray for their XT3 platform. The current implementation of Portals for the SeaStar does a limited amount of processing on the network interface and relies on the host to perform many of the message processing duties. This initial implementation has demonstrated respectable performance, with a zero-length NetPIPE latency of 5.39 μs and a peak bandwidth of over 1.1 GB/s. The software stack for both the generic mode (using the host CPU) and the accelerated mode (using the network interface CPU) are currently under active development. We expect both latency and bandwidth performance to increase for each mode over the next several months.

8. Acknowledgments

The authors gratefully acknowledge the work of the members of the Scalable Computing Systems and Scalable Systems Integration departments at Sandia, especially Jim Laros and Sue Kelly.

References

[1] R. Alverson. Red Storm. In *Invited Talk, Hot Chips 15*, August 2003.

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

[3] R. Brightwell. A comparison of three MPI implementations for Red Storm. In *Proceedings of the 12th European PVM/MPI Users' Group Meeting*, September 2005.

[4] R. 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. 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. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

[7] Lustre: A scalable, high-performance file system. Cluster File Systems Inc. white paper, version 1.0, November 2002.

[8] David A. Wheeler. *SLOCCount*. Available from `http://www.dwheeler.com/sloccount`.

[9] W. Gropp. MPICH2: A new start for MPI implementations. In D. Kranzlmuller, P. Kacsuk, J. Dongarra, and J. Volkert, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 9th European PVM/MPI Users' Group Meeting, Linz, Austria*, volume 2474 of *Lecture Notes in Computer Science*. Springer-Verlag, September/October 2002.

[10] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Cray User Group*, Albuquerque, NM, May 2005.

[11] 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.

[12] K. T. Pedretti and T. Hudson. Developing custom fi rmware for the Red Storm SeaStar network interface. In *Cray User Group Annual Technical Conference*, May 2005.

[13] 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.

[14] Q. O. Snell, A. Mikler, and J. L. Gustafson. NetPIPE: A network protocol independent performance evaluator. In *Proceedings of the IASTED International Conference on Intelligent Information Management and Systems*, June 1996.