Toward Implementing Robust Support for Portals 4 Networks in MPICH

Ken Raffenetti, Antonio J. Peña, Pavan Balaji
Argonne National Laboratory
Mathematics and Computer Science Division
Argonne, IL USA
{kraffenetti, apenya, balaji}@anl.gov

***Abstract*—The Portals 4 network specification is a low-level API for high-performance networks developed by Sandia National Laboratories, Intel Corporation, and the University of New Mexico. Portals 4 is specifically designed to support both the MPI and PGAS programming models efficiently by providing building blocks upon which to implement their particular features. In this paper we discuss our ongoing efforts to add efficient and robust support for Portals 4 networks inside MPICH, and we describe how the API semantics influenced our design. In particular, we found the lack of reliability guarantees from the Portals 4 layer challenging to address. To tackle this situation, we implemented an intermediate layer—Rportals (reliable Portals), which modularizes the reliability functionality within our Portals network module for MPICH. In this paper we present the Rportals design and its performance impact.**

I. Introduction

The Message Passing Interface (MPI) is the most widely adopted parallel programming model on distributed-memory high-performance computing (HPC) systems. Used for over 20 years, MPI has successfully adapted to each new generation of supercomputer. MPI's ability to adapt to new architectures allows users to focus on application programming, without needing to know the low-level details of the underlying communication path, which may involve intra- and internode communications on top of a variety of hardware, including, for instance, coprocessor memory spaces. It is up to the MPI library to best handle communication, and MPI implementers to evaluate and test new research in that area.

In this paper we evaluate the Portals 4 network specification for use in MPICH [1], a high-performance and widely portable MPI implementation from Argonne National Laboratory. MPICH is the default MPI library in 9 of the top 10 supercomputers in the current TOP500 list [10].

Portals 4 is a recently emerged application programming interface (API) specifically designed for communication across high-performance networks under the MPI and partitioned global address space (PGAS) programming models. Influenced by the matching between the Portals 4 semantics and the MPICH network module (*netmod*) layer API, we have designed a robust MPICH netmod for Portals 4. In this paper we discuss the design and implementation details of our approach. In particular, we find challenging the implementation of reliable communication operations, which may fail inadvertently because of exhaustion of a variety of resources on either endpoint of the communication. Handling these events is explicitly left to the API user by the Portals 4 specification. In this paper we discuss the way we are addressing this situation and the performance implications of our solution, ultimately derived from the Portals 4 API design.

II. Portals 4

Portals 4 [3] is a low-level network API for high-performance networking developed by Sandia National Laboratories, Intel Corporation, and the University of New Mexico. Having gone through several iterations over the years, Portals 4 is designed to efficiently support an MPI implementation with performance and scalability in mind. Our goal when adding support for Portals 4 inside MPICH is to provide a correct, easy-to-understand implementation following the recommendations of the specification.

Portals 4 is designed to support both MPI and PGAS programming models on HPC systems via a connectionless, network-independent API. Communication is performed through "portals"—constructs representing "an opening in the address space of a process" according to the description in the Portals 4 documentation. Unlike similar constructs such as TCP sockets or Queue Pairs in InfiniBand verbs, a portal is not restricted to connect a single pair of endpoints. The Portals API provides semantics useful for both one-sided ("put", "get") and two-sided ("matching put", "matching get") communication models. These operations target list entries that determine where data is placed. Completion of operations is notified by events that can be read from one or more event queues. Asynchronous progress of communication operations is mandated by the Portals 4 specification. The reference implementation accomplishes this via an explicit progress thread on each process. Hardware implementations can provide more efficient progress capabilities.

Matching operations in Portals are intended for use by MPI two-sided communication. Operations over a matching interface include a set of match bits that can be used to encode MPI communication contexts, message tags, and any other necessary information. Unexpected messages in MPI are supported by an optional overflow list that can be used to build an unexpected message queue. These features provide the building blocks for an efficient two-sided MPI implementation. On the other hand, one-sided MPI operations are naturally implemented on top of the Portals 4 put/get primitives.

III. Related Work

Previous work, such as OpenSHMEM [5] or GASNet [6], has discussed support for PGAS projects on top of Portals. In

this paper, however, we focus on the other target programming model of the Portals design: MPI.

The implementation of OpenMPI on top of the Portals 3.0 specification [7] leveraged an eager-based large-message protocol that enforces the initiator of the communication to wait for the remote endpoint to post the matching receive operation, in contrast with our target read-based approach. In addition, the lack of proper support for resource exhaustion recovery on the Portals 3 API forced the MPI runtime to abort the execution instead of attempting recovery.

After the release of the Portals 4.0 specification, flow control capabilities were incorporated into this OpenMPI-based implementation [4]. In addition to a credit-based approach, a receive-managed scheme conceptually similar to the one we leverage for MPICH was presented. Flow control recovery overhead, however, was not discussed on the paper, claiming that well-designed applications should not trigger this type of event. The current MPICH Nemesis design [8] oriented to two-sided network interfaces leads us to adopt a Portals 4 implementation based on a pool of receive buffers. Since this design is prone to trigger flow control events, we include an evaluation of flow control recovery overhead in our experimental results.

Software-based flow control schemes for different networking APIs have also been explored in the past, such as in the MVAPICH MPICH derivative for InfiniBand interconnects, analyzing different credit-based approaches [9]. Influenced by the connected point-to-point–oriented communication of its target communication API—a common factor on many low-level HPC communication APIs and other related work on this field—this work did not address multiendpoint connectionless environments.

In this paper we present the key design and implementation details of a Portals 4 network module specifically designed for MPICH.

IV. Design and Implementation

In this section we first introduce the initialization details of our new netmod. Next, we discuss our approach to support two-sided communications, followed by our approach for the one-sided primitives. We conclude this section by presenting our reliability layer to handle flow control events.

A. Initialization

Portals requires some initialization before it can be used for communication in an application. We combine this with MPI initialization in our implementation.

A memory descriptor covering the process's whole address space is bound to the interface. This allows Portals implementations that require memory registration to take any necessary steps up front. Since memory registration is time-consuming, this blanket method is more desirable than a more frequent, on-demand paging scheme.

As introduced in Section II, one or more "portals" must be allocated in order to communicate with other processes. In our design, we allocate three portals. The first is for basic data movement (send/receive) functionality, the second is for MPICH control messages and one-sided operations, and the third is for remote memory read ("get") operations to be used in large data transfers (see Section IV-B). These portals generate events that MPICH uses to interpret the state of communication. For MPICH to receive these events, we allocate and bind a separate event queue (EQ) to each portal. We allocate an additional EQ for events resulting from "origin"-side operations ("put," "get"). We discuss this design in Section IV-D.

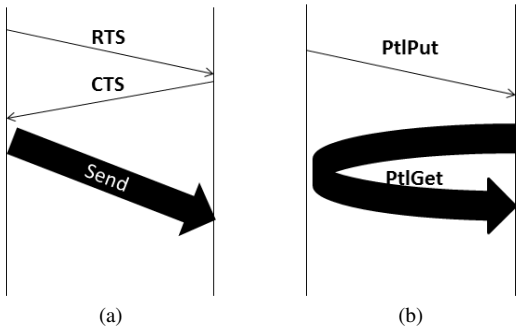


Fig. 1. Traditional (Left) and Get-based (Right) Rendezvous

Each process then retrieves its unique address information (node and process identifier) from Portals and publishes it via the Process Management Interface (PMI) [2] in MPICH for lookup by other connected processes.

B. Two-Sided Communication

Like most MPI implementations, MPICH utilizes two internal protocols for two-sided messages—eager and rendezvous. In the following we revise how we implement those MPI-layer protocols at the netmod layer.

The *eager* protocol is directly mapped to Portals 4 calls, taking advantage of the MPI-oriented semantics of the Portals 4 API: a single `PtlPut` operation is used to send data to the receiver regardless of whether the receiver is expecting the message. The target process uses a Portals matching list entry to receive the data, either from the priority list or from the unexpected list, depending on whether the message arrived before or after the receive operation was posted.

The *rendezvous* protocol, on the other hand, is implemented following a read/get-based approach [11], as illustrated in Figure 1 along with the traditional implementation. This contrasts with the eager-based approach of previous work on the OpenMPI implementation [7].

C. One-Sided Communication

Our one-sided primitives make use of a separate portal that is shared with MPICH control messages. Our approach leverages a set of relatively small preposted receive buffers. These are large enough to hold eager-sized messages. When larger messages are to be exchanged, the sender places the appropriate information in the target buffer to enable the receiver side to initiate a "get" operation, as depicted in Figure 1b. These buffers are reposted for receiving during event handling after all associated operations are finished and the acknowledging events properly processed. In case an incoming

message is attempted while all these buffers are in use, a flow control event is triggered, and the Rportals layer is in charge of transparently and reliably restoring communication (see Section IV-D). By default our implementation deploys fifty 64 KB receive buffers.

This design is influenced by the two-sided orientation of the current MPICH RMA implementation. After the ongoing redesign effort toward purely one-sided approaches at all levels of the MPICH stack is finished, our Portals 4 implementation will be adapted accordingly.

D. Flow Control

The Portals specification defines reliable communication between processes, although with some limitations. If certain communication resources are exhausted, the affected portal enters into a "flow control" state in which messages may be dropped. Flow-control recovery must be done at the user level because Portals does not directly provide any recovery mechanisms.

Several scenarios may trigger flow control in Portals. An application may run out of space in the event queue for new entries or exhaust the number of allowed message headers in the overflow list, for example. This situation causes the portal to which the exhausted resource is bound to *disable* and negatively acknowledge (NACK) all incoming messages. In order to ensure all communication eventually completes, all potential senders must be notified of the disabled portal so they can resend any dropped messages. This requirement also forces senders to save a copy of the sent data until they receive the corresponding acknowledgment from the receiver side if reliable communication is required. The Portals specification recommends the use of an alternative portal to exchange recovery messages and globally quiesce the network in order to avoid interfering with the rest of the communications and potentially disabled portals. We adopt this recommendation in our design.

Our approach is to introduce an additional reliability layer, Rportals, that implements wrappers for operations that are affected by flow control. Rportals presents a clean, reliable API to the netmod, hiding the details of flow control recovery internally. To accomplish this, we add wrappers for communication operations (`PtlPut`, `PtlGet`) to add tracking overhead. We also wrap `PtlEQGet`—the event retrieval function—to allow Rportals to trap events specific to flow control and perform recovery as follows (see Figure 2): (1) send pause messages to any process that may target the disabled portal; (2) wait for all processes to acknowledge the disabled portal, ensuring that all dropped operations have been requeued; (3) process sufficient events to relieve pressure on the exhausted resource; and (4) send unpause messages.

Another preventative measure we adopt is to keep a separate EQ for locally issued events. At Portals initialization time, we query the maximum number of events a queue can hold before going into flow control state. While we cannot ensure that remote processes do not overflow an event queue, we can use a credit-based scheme for origin-side events to ensure that a process does not cause a flow control situation on itself. Before any origin operation is posted, we first check whether there are available credits, in which case the operation is issued and the credit counter decremented. If there are not sufficient credits, we process events in the queue, thus incrementing the credit counter to indicate there are resources available and new operations may proceed.

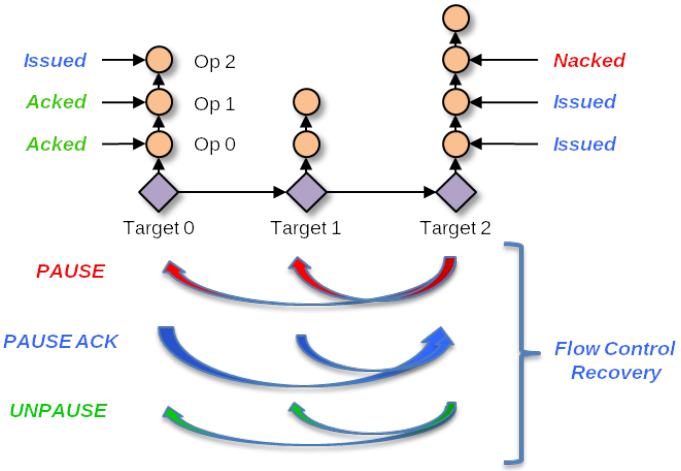


Fig. 2. Flow Control Recovery

V. Evaluation

In this section we analyze our evaluation results for the new MPICH Portals 4 netmod. For the evaluation we use two nodes in the Breadboard cluster at Argonne National Laboratory. Nodes consist of two Intel Xeon E5620 (four core, 2.4 GHz) processors, and 24 GB of RAM. They are connected by Mellanox MT26428 ConnectX HCAs. On the software side, we use the most recent version available at the development repositories of both MPICH and the Portals 4 reference implementation, dating from January 27, 2015, and January 28, 2015, respectively. Our latency and bandwidth evaluations are performed with the Intel MPI Benchmarks (version 4.0) package. We include results for the TCP netmod, our new Portals 4 netmod, and the MXM[1] netmod.

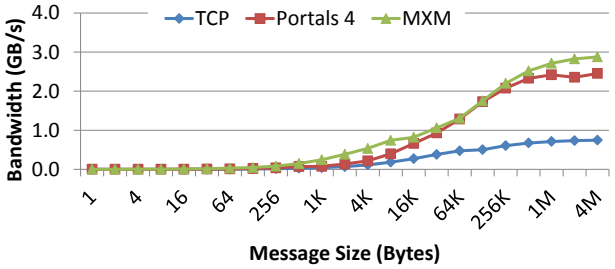
a) Initialization: We have measured the initialization time (i.e., that of the `MPI_Init` function call) in up to four nodes in our testbed, leveraging one process per core (eight processes per node). The initialization time incurred by our Portals netmod is close to that of the MXM netmod: 15% higher when the processes are within the same node, and 2–3% for the rest of the cases.

b) Latency: Our latency tests for 1-byte data payloads are summarized in Table I for both two-sided roundtrip ("Ping-pong") and one-sided "put" operations. We include results for the Portals 4 netmod with and without Rportals enabled ("Portals 4" and "Rportals," respectively). Our results show large gains with respect to the TCP netmod. A comparison with the MXM netmod, however, reveals room for improvement. In addition, the Rportals layer poses a nonnegligible overhead. We are currently addressing these overheads.

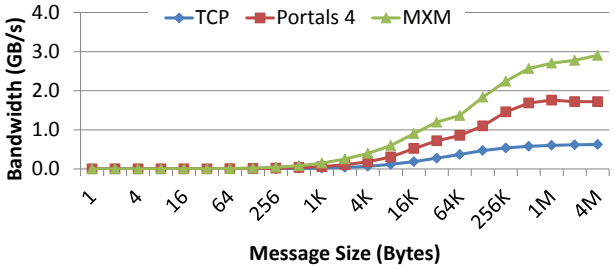
[1]MXM is a proprietary low-level communication API from Mellanox Technologies designed to easily interact with their networking hardware (if compared with the traditional InfiniBand Verbs API) while providing an efficient use of their networking infrastructure.

TABLE I. 1-BYTE LATENCY (µS)

	TCP	Portals 4	Rportals	MXM
Pingpong	18.28	4.90	6.87	1.79
Put	36.10	8.79	13.05	6.64



(a) Two-Sided Bandwidth



(b) One-Sided Bandwidth

Fig. 3. Experimental Evaluation Results

c) Bandwidth: Figure 3a reports the bandwidth attained in two-sided communications, whereas Figure 3b shows that of one-sided "put" operations. The Portals 4 implementation consistently outperforms the TCP netmod by up to 3.5x in the 2-sided experiments and up to 3.2x in the one-sided experiments. Compared with the MXM netmod, however, our new netmod yields up to 24% lower bandwidth for large data transfers starting at 16 KB, being within 2% for 64–128 KB. The bandwidth loss, however, is larger for 8 KB and less. In the case of one-sided ("put") operations, our bandwidth loss is consistently above 50% for large data transfers starting at 16 KB, and higher for small data payloads. We attribute these bandwidth differences to two main factors: (1) the MXM netmod is a highly tuned network module contributed by Mellanox Technologies that employs an API specifically designed for the underlying networking hardware, whereas the reference implementation of Portals 4 is a generic library implemented on top of InfiniBand verbs; and (2) the Rportals layer introduces nonnegligible overhead. We are currently investigating ways to address these limiting factors.

d) Flow Control: Figure 4 shows the impact of flow control recovery on runtime using the `One_put_all` RMA benchmark over two nodes and 16 total processes. The number of times flow control was triggered varied from run to run in our experiments, from zero to 100+. Recovery overhead can be observed in the increase in runtime: 30% on the low end, and over 300% on the high end. Avoiding recovery will be important to application performance, and methods to do so will be part of our ongoing work.

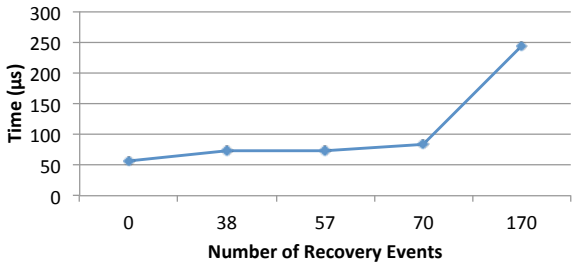


Fig. 4. Flow Control Recovery Overhead

VI. CONCLUSIONS

We have discussed the key design and implementation details of a Portals 4 network module for the widely used MPICH MPI implementation. These include initialization, two-sided and one-sided communication, and flow control. We discuss a set of evaluation results covering the major pieces of our netmod. Our target get-based rendezvous implementation yields high bandwidth for large-data payloads on two-sided operations. Nevertheless, room for improvement exists. For example, work is needed—and under investigation—on the latency for small messages. Moreover, we are working to reduce the nonnegligible overhead that the current implementation of our reliability layer introduces.

ACKNOWLEDGMENT

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC02-06CH11357.

REFERENCES

[1] Argonne National Laboratory, "MPICH — high-performance portable MPI," http://www.mpich.org, 2015.

[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Krishna, E. Lusk, and R. Thakur, "PMI: A scalable parallel process-management interface for extreme-scale systems," in *Recent Advances in the Message Passing Interface*. Springer, 2010, pp. 31–41.

[3] B. W. Barrett, R. Brightwell, R. E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson, "The portals 4.0.2 network programming interface," Sandia National Laboratories, Tech. Rep., Oct. 2014.

[4] B. W. Barrett, R. Brightwell, and K. D. Underwood, "A low impact flow control implementation for offload communication interfaces," in *Proceedings of the 19th European conference on Recent Advances in the Message Passing Interface*. Springer-Verlag, 2012, pp. 27–36.

[5] B. Barrett, R. Brigthwell, K. Hemmert, K. Pedretti, K. Wheeler, and K. Underwood, "Enhanced support for OpenSHMEM communication in Portals," in *IEEE 19th Annual Symposium on High Performance Interconnects (HOTI)*, Aug. 2011, pp. 61–69.

[6] D. Bonachea, P. Hargrove, M. Welcome, and K. Yelick, "Porting GASNet to Portals: Partitioned global address space (PGAS) language support for the Cray XT," in *Cray User Group Conference*, May 2009.

[7] R. Brightwell, R. Riesen, and A. B. Maccabe, "Design, implementation, and performance of MPI on Portals 3.0," *International Journal of High Performance Computing Applications*, vol. 17, no. 1, pp. 7–19, 2003.

[8] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem," in *International Symposium on Cluster Computing and the Grid*, 2006.

[9] J. Liu and D. K. Panda, "Implementing efficient and scalable flow control schemes in MPI over InfiniBand," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[10] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP500 supercomputing sites," http://www.top500.org/lists/2014/11, Nov. 2014.

[11] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006, pp. 32–39.