SAND2013-3092C

Design, Implementation, and Performance Evaluation of MPI 3.0 on Portals 4.0

Amin Hassani, Anthony Skjellum
University of Alabama at Birmingham
CH 115, 1300 University Blvd
Birmingham, AL, 35205
{ahassani,tony}@cis.uab.edu

Ron Brightwell, Brian W. Barrett
Sandia National Laboratories
PO Box 5800
Albuquerque, NM 87185-1319
{rbbrigh,bwbarre}@sandia.gov

ABSTRACT

The latest version of the Portals interconnect programming interface contains several improvements intended for MPI point-to-point and one-sided communication operations, including new functionality defined in MPI-3. This paper discusses the rationale for these improvements to Portals and describes how they can be used in an MPI implementation. We provide preliminary micro-benchmark performance results using a reference implementation of Portals over InfiniBand Verbs and the Open MPI implementation.

Keywords

MPI, Portals, One-Sided, Point-to-Point, Synchronization

1. INTRODUCTION

The MPI Standard [11] and accompanying implementations have evolved over the last twenty years to allow applications to continue to scale to machines with several hundred thousand network endpoints and beyond. The MPI Forum is continuing to develop the specification to address performance, scalability, and usability issues that will be required to allow applications to continue to run effectively on multi-petascale systems and eventually on exascale systems. As MPI evolves, so must the underlying transport layers upon which MPI implementations are layered.

Similar to MPI, the Portals [7, 8] data movement layer has been developed for nearly twenty years and has gone through several updates to address performance, scalability, and usability issues for implementing several upper-layer protocols, including MPI, for extreme-scale computing platforms. The previous generation of the Portals interface supported MPI and several other upper-layer protocols on the Cray SeaStar [6] network on the Cray XT series of machines, including Oak Ridge National Laboratory's Jaguar, which was the first petascale machine composed entirely of commodity x86 processors. The most recent version of Portals, version 4.0 [4], has been enhanced to support new capabilities available in MPI 3.0, such as non-blocking collective operations [14], as well as to provide optimizations for one-sided networking interfaces like OpenSHMEM [1]. In this paper, we describe the changes made to Portals to better support MPI peer communication operations as well as MPI one-sided operations.

The rest of paper is organized as follows: In section 2, we first introduce basics of portals 4.0 and its important functionalities. Then we continue with description of our design and implementation in Section 3 along with Section 3.1 on point-to-point operations and Section 3.2 on one-sided operations. In Section 4, we offer performance evaluation and at last we discuss our future work in Section 6.

2. PORTALS 4.0

Portals 4.0 is a modern network data transfer interface specification that is designed to be highly scalable and offer low overhead. The main goal of Portals implementations is to serve as optimized data transfer layers for upper layer protocols like MPI. The motivation for the enhancements to Portals between the previous version and the current version have been previously discussed [10, 5].

Portals' data transfer mechanism is one-sided. Usual one-sided data transfer systems require three values for addressing a target buffer: rank, buffer id and offset. Portals addressing needs rank, an index to *portal table* (PTE), *match bits*, and an optional offset. Each virtually initialized portal network interface has at least 64 indices in a portal table (PT), which is used as a protocol switch [9] for flexibility in higher level protocols. Match bits is a 64-bit integer that is used to match an incoming request with an already posted *match entry* (ME). Match entry structure has a buffer address, offset, an optional flag, and two 64-bit integers representing match bits and *ignore bits*. ignore bits serves as a mask to ignore certain bits of incoming match bits.

There are two types of addressing semantics in Portals: *Matching* and *Non-matching*. In matching semantics, each incoming request is matched to a pre-posted match entry based on its match bits while in non-matching semantics, an incoming request will be assigned to the first buffer available. Matching semantics are useful for MPI implementations while non-matching semantics are better aligned for implementations of systems such as OpenSHMEM.

In matching semantics, each PTE refers to a *match list*, which is basically a list of match entries, and a list of *unexpected headers*, which saves the information about unexpected requests in a list. Match lists are broken into two sublists: the *priority list* and *overflow list*. Match entries can be appended to both lists depending on the usage. The

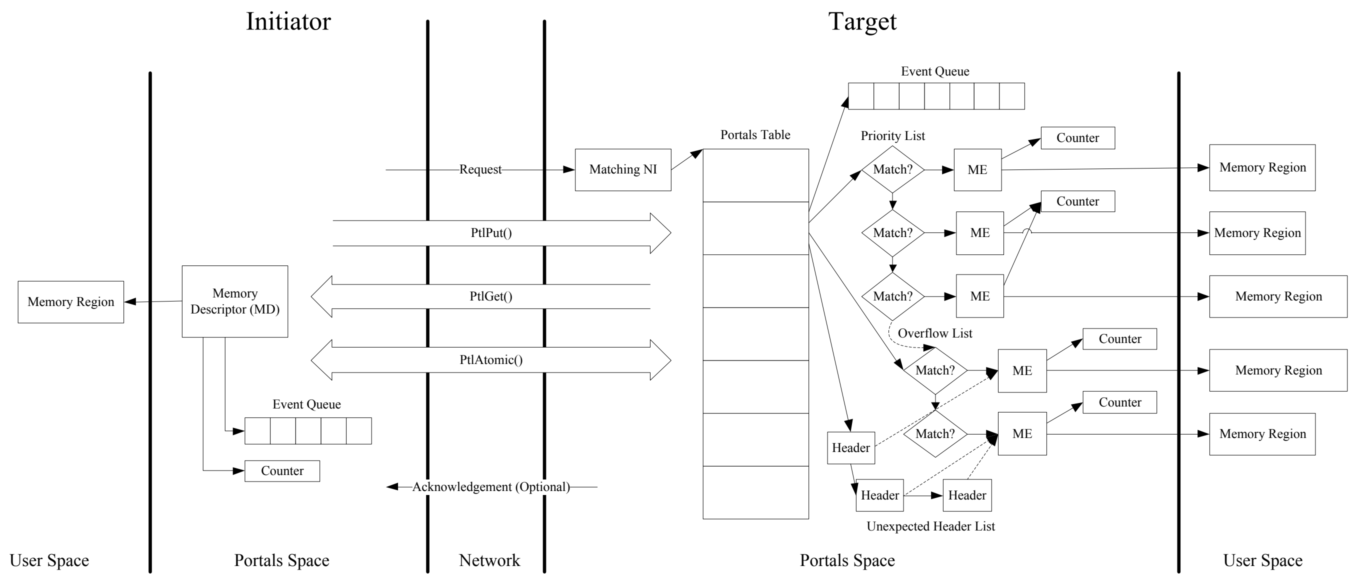

Figure 1: Portals 4.0 Addressing Semantics (Matching)

priority list is used for handling normal requests, while the overflow list is designed to help handling unexpected messages[1]. In a Portals communication operation, if a request cannot be matched to any match entries in priority list, the overflow list will be searched and, regardless of a match in overflow list, the request information will be saved in unexpected header list. On the other hand, when appending a match entry to a match list, first the unexpected header list will be searched for a match and if the match is found it will generate an appropriate event without appending match entry to the match list, otherwise the match entry will be appended to match list. While both priority and overflow lists can be searched and modified for a match entry, there is no direct access to the unexpected header list in Portals.

Exposing unexpected messages through the overflow list is one of the most visible changes made to Portals to better support MPI. Searching the unexpected message list and posting a receive must be done atomically, and the previous approach in Portals was inefficient. For an offloaded implementation, the previous approach required a round-trip operation between the host and network. The overflow list allows the network to perform the search-and-post operation and either identify a matching unexpected message or post the receive without further involvement from the host.

Data transfer operations use *memory descriptors* (MDs) on the initiator side. Each MD describes a region of memory and optional event queues and should be bound using `PtlMDBind()` before start of any communication operations. Binding is an important function that is necessary for underlying networks like InfiniBand [12] to pin the memory before any data transfer.

Portals uses an event-based mechanism to notify the upper layer any type of event. Each PTE includes an event queue which records all events. An event contains all the information about an operation and its consequences. Capturing all information on an event introduces high overhead which is not needed in every case. To compensate this problem another lightweight event type called *Counting Events* (CT) is introduced. Counting events are used to record just the number of successful or failed operations happened on a particular MD or ME. They are also used in triggered operation when an operation should not be started until certain number of events happen in an event queue [2]. Both types of event can report event type and its state as either success or failure. During an operation different events can happen on both initiator and target side.

3. DESIGN AND IMPLEMENTATION

In this section, we describe our implementation of MPI 3.0 on Portals 4.0. We based our work on the Open MPI development trunk.[2] Both MPI point-to-point and one-sided data movement operations are implemented on top of Portals 4.0. For each types of protocols one or more PTEs are allocated. In the next section, we describe the implementation of each protocol in detail.

3.1 Point-to-Point Operations

Based on the approach in [9], our MPI point-to-point communication uses a two-level protocol for optimizing short and long message transfers. For long transfers we have two protocols from which to choose: *eager* and *rendezvous.* During initialization three PTEs are allocated: *recv, read* and *flow control.* The recv PTE is used for sending/receiving messages, while read PTE is used for rendezvous long messages and overflow PTE is used to handle message processing in case of overflow.

The match bits for point-to-point protocols are partitioned into four parts: Four bits are reserved to encode the type of protocol (long and short transfers, two bits are enough but four is chosen for simplicity and reserve for further modifications of protocol), 12 bits for communicator's context id, 16 bits for source rank, and 32 bits for message tag. Match

[1]A arrival message is unexpected when there is match-entry posted in match-list to accept the match bits of message

[2]The point-to-point work is available in Open MPI 1.7 and the one-sided work will be available in Open MPI 1.4.2.

bits allow for a reunification of the partitioned name space of MPI's multiplexing and demultiplexing of messages with the concepts of communicator 'context', message tag, source or destination ranks, and transport properties internal to the implementation without multiple stages of demultiplexing in software.

3.1.1 Short Messages

In our implementation, short messages are transferred using an eager protocol. The entire message buffer is transferred and if target is not ready, the message will be buffered in target. When `MPI_SEND()` is called in initiator, an associated MD is created, configured with the buffer information, target address, and match bits, and is bound through portals library. On the target side, on the call to `MPI_RECV()`, an ME is configured and attached to the priority list. A call to Portals' `PtlPut()` in initiator will transfer buffer to the target side.

In order to buffer unexpected messages, 16 MEs with 1 MB buffers will be attached to the overflow list of the recv PTE and its ignore bits are configured to accept any incoming short message. If a message is unexpected, Portals will save the message in unexpected buffer and its information in the unexpected header list. The `PTL_PUT` event in target shows the arrival of message in the unexpected buffer. When `MPI_RECV()` is called in the target, attachment request is called for appropriate ME to priority list, but portals first will search through unexpected header list and if it finds a match, target will be notified with its information in a `PTL_PUT_OVERFLOW` event and message have to be copied manually to destination buffer from unexpected buffer. Initiator process will be notified of completion of transfer in local side in a `PTL_SEND` event and if needed an additional `PTL_ACK` event shows whether transfer was succeed or failed (dropped in target).

3.1.2 Long Messages

We have two implementations for long message transfers: eager and rendezvous. In the default eager protocol, the message is eagerly sent to the receiver. If no ME is attached in target, the whole message will be dropped and its information will be saved in unexpected header list. After `MPI_RECV()` call, the message can be retrieved using `PtlGet()` operation in the target using a unique send identifier generated by the sender and sent in the original request as out-of-band data. In the rendezvous protocol, the first *eager_limit* bytes of the message are sent eagerly. If the message is unexpected, the first part of the message is delivered in the short message unexpected queue, otherwise it is delivered directly into user memory. A `PtlGet()` is issued either when the receive is posted (for the unexpected case) or during the next test/wait call (for the expected case). The eager protocol always provides asynchronous progress but wastes bandwidth for unexpected messages, while the rendezvous protocol only provides asynchronous progress for unexpected messages but does not waste bandwidth.

3.1.3 Overflow Control

In message passing systems exhaustion and overflow can happen when traffic toward a single process gets out of control. Resource exhaustion specially happens with two-sided semantics where unexpected messages happen frequently. In our implementation two situations can trigger overflow. Either number of unprocessed unexpected short messages go over its limits or many long messages are left at buffer that cause resource exhaustion. [3] implements a recovery-based overflow control mechanism in our implementation. Portals 4.0 provides a valuable interface for handling unexpected messages though overflow list and unexpected header list. In addition PTE can be disabled to stop receiving any incoming message in case of any resource exhaustion. Two methods were implemented, a static credit based, which increases allocated memory for unexpected short messages when number of processes increases. A second approach is a receiver-managed flow control which basically when overflow happens, PTE is disabled and all other processes are notified that a process entered into a flow control recovery session (triggered operations are used to notify all processes), All send operations are queued locally and waiting operations in receive queue are processed. All processes exit control flow recovery through a barrier and finally waiting messages are retransmitted to their destination.

3.2 One-sided Operations

The Portals 4 implementation of one-sided supports true asynchronous operation for both active and passive target synchronization, with an eager protocol used for communication in both situations. Each window includes a communicator duplicated from the communicator used to create the window. This communicator is used both for collective operations during active target synchronization and as the source of a unique identifier for the window. Each window is represented in Portals by two match list entries, one exposing the user memory for the window and the other exposing control data, such as lock status and post and complete counts. Additionally, a window includes two memory descriptors that cover all memory of the local process. Both memory descriptors increase a counting event for acknowledgments and one also generates full events and is used for the new request-based operations. Window creation is always collective, and a barrier in particular is used to ensure that all memory descriptors are set up before any process exits the creation function.

The new MPI-3 dynamic windows pose a challenge for many network interfaces, since the dynamic nature of the exposed memory requires significant sharing of state throughout the window lifetime. Portals optionally allows a list entry or match list entry to span a process' virtual memory space. Current implementations provide this support, so the current one-sided implementation creates a single match list entry exposing the entire process space for dynamic windows. A scheme of match list entries per exposed segment will be used in the unexpected case where a Portals implementation does not provide this feature.

3.2.1 Communication

Portals imposes a number of data transfer sizes that are important to the MPI-3 one-sided implementation. Atomic operations are limited to an implementation-defined size, likely to be in the range of the network's packet size. Operations are only data ordered (they are always header ordered unless explicitly requested to be unordered) if they are smaller than another implementation-defined size. For `MPI_PUT()` and `MPI_GET()` operations, these sizes do not come into play, as they are always unordered. However, accumulate operations must be broken down at the source into multiple operations no larger than the maximum atomic size (or maximum ordered size if strict ordering is enabled).

Portals provides put and get interfaces similar to those provided by `MPI_PUT()` and `MPI_GET()`, so in the common case, these MPI calls are simple wrappers around the underlying Portals call. Non-contiguous user-defined datatypes are broken up into multiple puts of contiguous datatypes, although this feature is currently under development and only contiguous datatypes are currently supported. For each Portals put or get call, a 64-bit operation count is incremented, which allows the implementation to track the number of outstanding operations by comparing the value of the operation counter and the counting event associated with the window's memory descriptors.

Portals provides atomic and fetch-atomic operations capable of implementing the accumulate operations of MPI-3, including native support for most of the MPI-3 primitive datatypes and reduction operations. The exception is native support for `MPI_MINLOC` and `MPI_MAXLOC`. Presently, these operations are implemented using a lock/unlock strategy, although we are investigating using a thread to avoid network polling. Unlike the put and get operations, which only had to be concerned with user-defined datatypes, accumulate operations must also initiate operations that remain under the message size limits previously described. Like put and get operations, non-contiguous or long accumulate operations will be split into multiple smaller messages. This is safe from an atomicity standpoint, as MPI only guarantees atomicity on the primitive datatype level.

The request-based operations are implemented similarly, but operations are initiated from a memory descriptor that generates full events on completion. During the operation, a request is generated and a pointer to that request is passed through the *user_ptr* field of the Portals operation and corresponding event, allowing completion to be marked on the event. The operation counter and counting event are still manipulated like non-request operations, allowing the synchronization operations to only track one completion mechanism.

3.2.2 Active Target

Two types of active target synchronizations are defined in the MPI standard: Fence and generalized. `MPI_WIN_FENCE()` is a collective call which is called on both 'bookends' of communication (before and after a set of puts and/or gets constituting an *epoch*). After this call, all one-sided operations should be finished on both local side and remote sides, from the perspective of each process. Fence waits for the counting event to reach the same value as the operation counter, meaning that all operations have completed remotely (a Portals acknowledgment is only generated after remote completion), then enters a barrier call on the communicator associated with the window.

Generalized active target synchronization allows subsets of processes to synchronize, using MPI groups to describe the overlapping exposure and access epochs. An exposure epoch is started by a call to `MPI_WIN_POST()`, which atomically increments a counter on each process in the associated access epoch. The access epoch is started by a call to `MPI_WIN_START()`, which spins waiting for the atomic counter to reach the number of processes in the associated exposure epoch. Completion is essentially the inverse, with `MPI_WIN_COMPLETE` waiting for remote completion using the counting event and then incrementing an atomic counter at the process in the exposure epoch. `MPI_WIN_TEST()` and `MPI_WIN_WAIT()` indicate completion when the atomic counter reaches the size of the group.

3.2.3 Passive Target

Passive target synchronization allows true one-sided communication, in which the target does not have to enter the MPI library for progress to be made. Passive target access epochs are bounded by calls to `MPI_WIN_LOCK()` and `MPI_WIN_UNLOCK()`, which provide exclusive or shared access to the remote window. MPI-3 adds `MPI_WIN_LOCK_ALL()` and `MPI_WIN_UNLOCK_ALL()` to allow global access epochs, and a set of flush routines to complete communication calls without completing the access epoch. Finally, `MPI_WIN_SYNC()` allows the user to synchronize the private and public copies of a window without ending the access epoch.

The one-sided implementation over Portals 4 utilizes a 64-bit atomic value at each process to represent the current lock status, with the lower 32 bits representing a shared lock counter and the upper 32 bits representing the exclusive lock counter. A process wishing to start a shared access epoch on a remote process atomically increments the 64 bit value. If the updated value is greater than 2^{32}, the target is in an exclusive lock and the counter is atomically decremented and the process must retry acquiring the lock. If the updated value is less than 2^{32}, the process has acquired the shared access. When the process wishes to end the epoch, the counter is atomically decremented. A process wishing to have an exclusive access epoch performs a 64-bit compare and swap operation on the lock value, requiring the entire value to be 0 and swapping in a 1 at the 32nd bit. If the compare and swap succeeds, the process has exclusive access. Otherwise, it must retry. A masked update setting just the 32nd bit to 0 releases the exclusive access.

The lock-all semantics are implemented by requesting a shared lock from all peers in the window, with linear complexity. It is assumed that most users of lock-all will provide the `MPI_MODE_NOCHECK` assert to lock-all, which eliminates the required linear operations. Likewise, the nocheck assert removes the need for the lock communication in the basic lock case.

The decision to use a polling lock is the result of two conflicting requirements: to avoid polling on long-latency events and the requirement to avoid state which scales with the number of processes in the job. An MCS lock [13] avoids remote polling, but requires a word of state on every process for every lock. A window can be seen as having p locks, one for each rank in the window, requiring p words of state at each rank be associated with each window. The polling implementation, on the other hand, requires only 1 word of state at each rank be associated with each window, greatly reducing memory footprint in return for added network operations.

As a trade-off for implementing very lightweight event tracking, the Portals one-sided implementation is unable to differentiate between local and remote completion or per-target completion. All flush variants therefore implement global remote completion semantics. Although this increases the cost of local or per-target remote completion, we believe the trade-off in lower synchronization cost and increased message rate is advantageous to most applications.

4. EXPERIMENTS

In this section, we describe our experimental results with the implementation on our machines with 24 cores Intel(R) Xeon(R) CPU E5-2620 2.00GHz with 64 GB of memory and OS kernel of 2.6.32-5-amd64 on Debian 6.0.6. Each machine has Mellanox MT27500 infiniband network card.

We ran our micro benchmarks to capture the latency for different types of communication and synchronization operations. In order to truly get the latency of each operation, synchronization latency should be separated from communication latency. To reach this goal we first calculated synchronization epoch's overhead without any communication in between and then we subtracted this timing value from a synchronization epoch with only one communication call in the epoch. In our experiments different synchronization methods didn't affect the latency of a data transfer operation. As shown in Figure 4 We did experiment for `MPI_PUT()` and `MPI_GET()` for one-sided and `MPI_SEND()/MPI_RECV()` for point-to-point operations. In all experiments we the lowest latency belonged to put operation while two-sided operations have higher latency.

Figure 3 shows our experiments with communication bandwidth for the same MPI methods. To calculate the bandwidth we called data transfer function several times in a synchronization epoch. Results shows we can get more bandwidth out of one-sided operations compared to point-to-point operations.

MPI one-sided synchronization overhead is important specially if we deal with several processes and race over a few processes' buffer can lead to high overhead. All synchronization method are benchmarked with up to 8 processes as shown in Figure 2. `MPI_FLUSH()` had constant and low overhead over all tests and the reason is that it just tries to complete all local tasks. `MPI_LOCK()` with `MPI_LOCK_SHARED` option and generalized active target synchronization for have similar overheads and reason is that in the implementation both try to synchronize with remote node using Portals' atomic operations but there is no race condition involved. `MPI_WIN_FENCE()` overhead is high because it is a collective operations on all processes in window's group. Racing to acquire the lock of a single target in `MPI_LOCK()` with `MPI_LOCK_EXCLUSIVE` option results in relatively high overhead when number of processes increases in contrast to other synchronization operations.

Figure 5 shows the one way latency of `MPI_LOCK()` when the number of calls to `MPI_PUT()` inside a synchronization epoch changes. In this experiment the overhead effect of synchronization is omitted.

5. SUMMARY

In this paper, we have demonstrated the ability to perform MPI 3.0 one-sided operations using the Portals 4.0 interface, including the mapping of both MPI 3.0 one-sided and point-to-point operations to Portals 4.0. The implementation is throughly described and different trade offs for different protocols are discussed. MPI's two-sided operations' implementation for both short and long transfers are described and we also briefly mentioned the overflow recovery mechanism implemented. MPI's one-sided operations are discussed with both Active and Passive synchronizations. We provided experimental results on a modern InfiniBand interconnect, and obtained good results for one-sided operations.

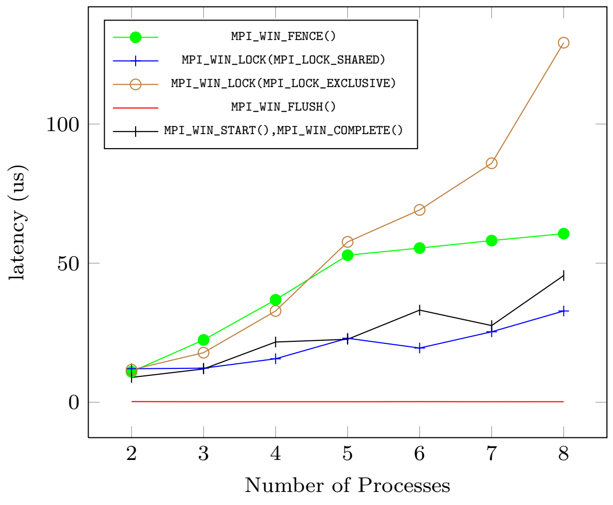

Figure 2: Synchronization latency for different numbers of processes

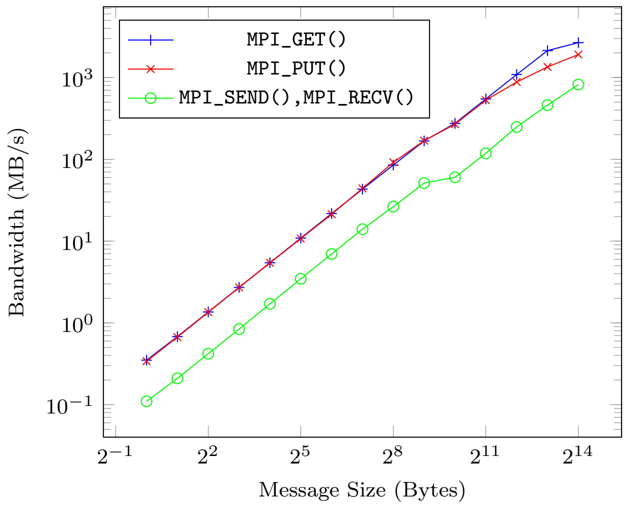

Figure 3: Bandwidth

6. FUTURE WORK

There plans to move one-sided communication over non-matching addressing structure of portals and give each window its own PTE. We trying to support non-contiguous data types in near future. Improvements on performance and provide support for different types of upper protocols are our next goals.

7. ACKNOWLEDGMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

8. REFERENCES

[1] B. Barrett, R. Brightwell, K. S. Hemmert, K. Pedretti, K. Wheeler, and K. Underwood. Enhanced support for

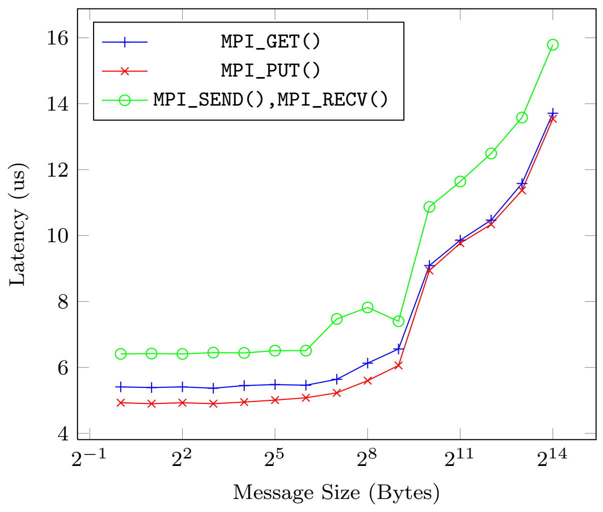

Figure 4: Latency

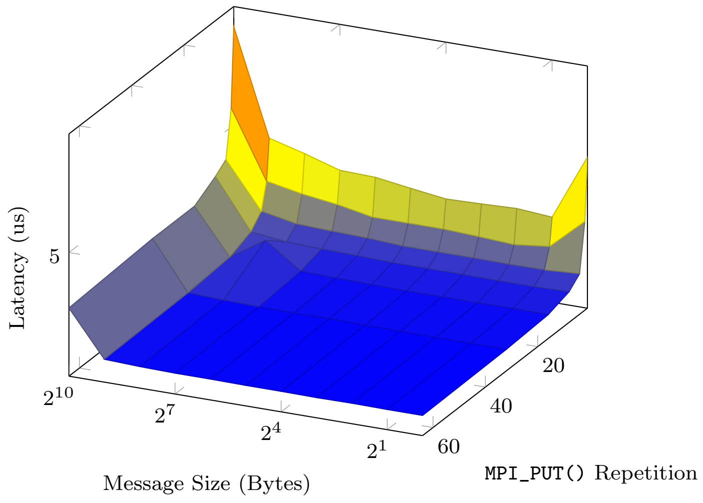

Figure 5: Impact of message transfer repetitions between lock synchronization epoch and message length to data transfer latency of `MPI_PUT()`.

openshmem communication in portals. In *IEEE Symposium on High-Performance Interconnects*, pages 61–69, Santa Clara, California, 2011.

[2] B. Barrett, R. Brightwell, K. S. Hemmert, K. Wheeler, and K. Underwood. Using triggered operations to offload rendezvous messages. In *European MPI Users; Group Conference*, pages 120–129, Santorini, Greece, 2011.

[3] B. Barrett, R. Brightwell, and K. Underwood. A low impact flow control implementation for offload communication interfaces. In *in Proceedings of the European MPI Users; Group Meeting*, 2012.

[4] B. W. Barrett, R. Brightwell, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. B. Maccabe, and T. Hudson. The portals 4.0 network programming interface. Technical Report SAND2012-10087, Sandia National Laboratories, Nov. 2012.

[5] R. Brightwell, B. Barrett, K. S. Hemmert, and K. D. Underwood. Challenges for high-performance networking for exascale computing (invited paper). In *International Conference on Computer Communication Networks*, Zurich, Switzerland, 2010.

[6] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, May/June 2006.

[7] R. Brightwell, A. B. Maccabe, R. Riesen, and T. Hudson. The portals 3.0 message passing interface revision 1.1. *Sandia National Laboratories*, 1999.

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

[9] 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*, 17(1):7–19, 2003.

[10] R. Brightwell and K. Underwood. Network programming interfaces for high-performance computing. In A. Gavrilovska, editor, *Attaining High-Performance Communication: A Vertical Approach*, pages 149–168. CRC Press, 2009.

[11] M. P. I. Forum. MPI: A message passing interface standard, 1994.

[12] T. Hamada and N. Nakasato. InfiniBand trade association, InfiniBand architecture specification, volume 1, release 1.0; http://www.infinibandta.com. In *in International Conference on Field Programmable Logic and Applications, 2005*, pages 366–373.

[13] T. Johnson and K. Harathi. A simple correctness proof of the mcs contention-free lock. *Information Processing Letters*, 48(5):215 – 220, 1993.

[14] K. Underwood, J. Coffman, R. Larsen, K. S. Hemmert, B. Barrett, R. Brightwell, and M. Levenhagen. Enabling flexible collective communication offload with triggered operations. In *IEEE Symposium on High-Performance Interconnects*, pages 35–42, Santa Clara, California, 2011.