Revisiting Rendezvous Protocols in the Context of RDMA-capable Host Channel Adapters and Many-Core Processors

Masamichi Takagi
Green Platform Research Lab., NEC Corp.
m-takagi@ab.jp.nec.com

Yuichi Nakamura
Green Platform Research Lab., NEC Corp.
yuichi@az.jp.nec.com

Atsushi Hori
RIKEN Advanced Institute for Computational Science
ahori@riken.jp

Balazs Gerofi
Dept. of Computer Science, Univ. of Tokyo
bgerofi@il.is.s.u-tokyo.ac.jp

Yutaka Ishikawa
Dept. of Computer Science, Univ. of Tokyo
ishikawa@is.s.u-tokyo.ac.jp

ABSTRACT

We revisit RDMA-based rendezvous protocols in MPI in the context of cluster computer with RDMA-capable HCA and many-core processors, and propose two improved protocols. The conventional sender-initiate rendezvous protocols cause costly processor-device communications via PCI bus on detecting completion of RDMA transfer. The conventional receiver-initiate rendezvous protocols need to send extra control messages when a value of the memory-slot to poll in the receive buffer has the same value as the send buffer. The first proposed protocol implements polling on a memory-slot in the receive buffer to eliminate the processor-device communications. The second proposed protocol randomizes the value of the memory-slot to poll to reduce extra control messages. We have evaluated the proposed protocols using micro-benchmarks and NAS Parallel Benchmarks. One of the proposed protocols has a benefit compared to the conventional protocols. And the second proposed protocol reduces the execution time by up to 11.14% compared to the first protocol.

Keywords

MPI, rendezvous protocol, RDMA, polling, HCA

1 Introduction

In HPC area, computers tend to employ cluster architecture in which a large number of commodity PC servers are connected via a network. This is because PC architecture has an advantage in cost-performance compared to the custom-built architecture. InfiniBand (IB) [3] is widely used as the network because it has a cost-performance advantage over other network technologies, and its RDMA capability lowers latency of large message transfer [5]. Furthermore, many-core processors such as the Intel Xeon Phi Co-processor have been adopted for the cluster computers because they have an advantage in computation throughput compared to processors with fewer cores.

One of the most important components in execution time in HPC programs is communication latency. There are three major approaches to reduce the communication latency. The first approach is to improve the communication hardware, e.g. increasing the throughput per wire. The second approach is to improve the software protocol to reduce the overhead, e.g. reducing protocol processing computation and control messages. The third approach is to refine the communication pattern in the programs to reduce the number of messages and their sizes. We focus on the second approach and we focus on improving the Message Passing Interface (MPI) [6] protocol because MPI is the de facto standard for writing parallel applications. In addition, we focus on improving rendezvous protocol because a large portion of communications is handled by it.

There are two major sources of latency overhead in rendezvous protocol. The first source is processor-device communications via PCI bus. That is, Host Channel Adapter (HCA) issues a PCI command on the PCI bus to write a value to a memory-slot. Such a communication occurs in a rendezvous protocol with two purposes. The first purpose is for the HCA to write contents of the MPI user buffer which came through the network to the receive buffer. The second purpose is for the HCA to notify the processor of completion of an RDMA transfer. And the cost of a processor-device communication amounts to hundreds of processor cycles. The second source is sending control messages. The latency caused by sending control messages is multiplied by tens in the cluster architecture. This is because the latency is multiplied by the number of cores sharing one HCA and performing simultaneous communications through it, and the number has increased to tens.

We revisit rendezvous protocols of MPI in the context of reducing these kinds of latencies and propose two improved

EuroMPI '13, September 15 - 18 2013, Madrid, Spain

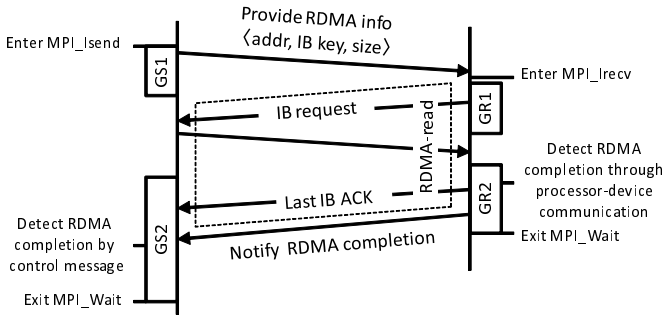


Figure 1: Timeline diagram of conventional sender-initiate rendezvous protocol.

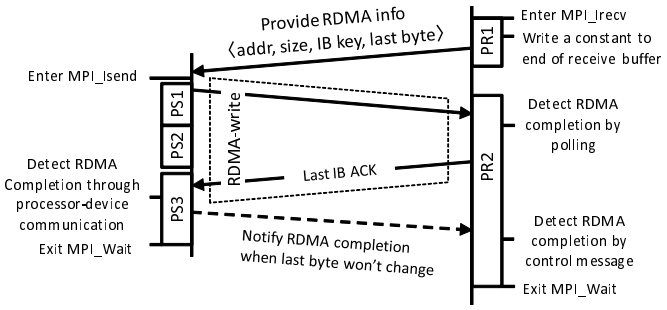


Figure 2: Timeline diagram of conventional receiver-initiate rendezvous protocol.

protocols. The contribution of this paper is two-fold.

- A new sender-initiate rendezvous protocol which eliminates one processor-device communication via PCI bus is proposed. The conventional sender-initiate rendezvous protocol involves a processor-device communication to detect completion of an RDMA transfer. Our proposed protocol exploits the fact that RDMA transfer changes the contents of the receive buffer and detects the completion by polling the change to eliminate the communication.

- A new receiver-initiate rendezvous protocol is proposed, which reduces the probability of sending an extra control message. The conventional receiver-initiate rendezvous protocol makes the receiver poll on a memory-slot in the receiver buffer to monitor change of the value to detect completion of an RDMA transfer. The sender needs to send an extra control message in the situation when the memory-slot has the same value as the corresponding memory-slot in the send buffer and its value won't be changed when RDMA transfer copies the contents of the send buffer to the receive buffer. The conventional protocol sets a constant value to the memory-slot before an RDMA transfer. However, the constant value might be the frequent value appearing in the corresponding memory-slot in the send buffer and hence resulting in causing the same-value situation frequently. Our protocol reduces the chance for the situation to happen by randomizing the value in the memory-slot.

The remainder of this paper is organized as follows. In Section 2, we discuss related work. In Section 3, we describe our protocol design. In Section 4, we explain the evaluation results. In Section 5, we conclude the paper.

2 Related Work

MPI utilizes a protocol called the rendezvous protocol when sending a large message. The typical message size is over several thousand bytes. Many studies proposed different protocols and they are categorized in three types. We discuss those conventional protocols following the types. We call the memory area from which a user program send a message **send buffer** and the memory area to which a user program receive a message **receive buffer**. We call an MPI process which tries to send a message **sender side**, which tries to receive a message **receiver side**.

2.1 Sender-Initiate Protocols

A protocol where the sender side first sends a control message to the receiver side is called **sender-initiate protocol**. We call the conventional sender-initiate protocol GETCQE. Its steps are as follows [4, 12]. **Fig. 1** shows a timeline diagram of the protocol. GRn are performed by the receiver side and GSn are performed by the sender side.

GS1 The sender side enters `MPI_Isend` and sends a control message to the receivers side requesting an RDMA read. The control message contains the information for RDMA (start address of the send buffer, its size, memory protection information).

GS2 The sender side enters `MPI_Wait` and waits for a control message of the RDMA read completion. And then it exits `MPI_Wait`.

GR1 The receiver side enters `MPI_Irecv` and checks arrival of the control message sent by the sender side. And then it initiates an RDMA read.

GR2 The receiver side enters `MPI_Wait`. The HCA notifies the processor of completion of the RDMA read by a processor-device communication through a memory-slot called Completion Queue Entry (CQE) and the receiver side monitors the CQE to detect the completion. The receiver side sends a control message of RDMA read completion to the sender side. The receiver side exits `MPI_Wait`.

The protocol has an issue that it involves a costly processor-device communication in step GR2.

Small et al. [10] proposed an improvement over the sender-initiate protocol where the sender side exploits the knowledge of the time when the receiver side enters `MPI_Irecv` to reduce the time to exit `MPI_Wait`. The sender side copies the contents of the send buffer to read-only memory area when it knows that the receiver side will enter `MPI_Irecv` later. The sender side can exit `MPI_Wait` without waiting for completion of the following RDMA read and this method can reduce the time to exit `MPI_Wait` in this way. However, their method still involves the processor-device communication.

2.2 Receiver-Initiate Protocols

A protocol where the receiver side first sends a control message to the sender side is called **receiver-initiate protocol**. We call the conventional receiver-initiate protocol PUTNR. The steps are as follows [8, 2]. **Fig. 2** shows a timeline diagram of the protocol. PRn are performed by the receiver side and PSn are performed by the sender side.

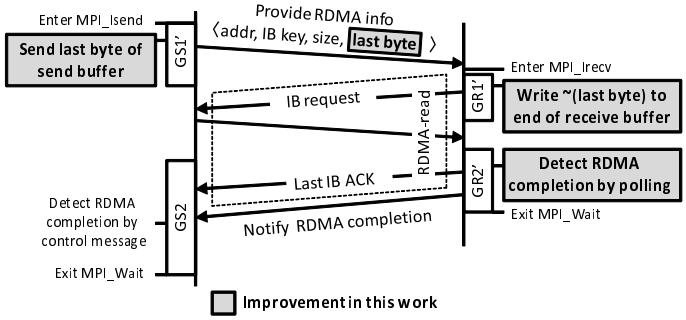


Figure 3: Timeline diagram of the GET protocol.

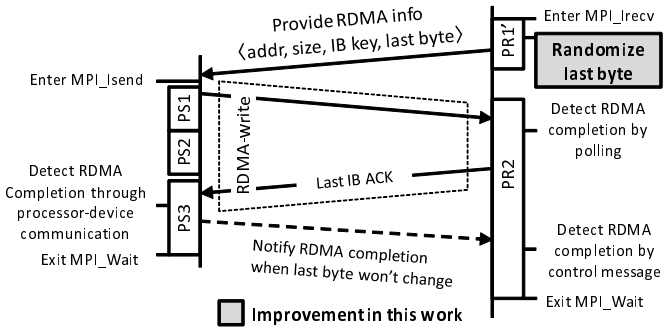


Figure 4: Timeline diagram of the PUT protocol.

PR1 The receiver side enters `MPI_Irecv`. The receiver side sets a constant value to the last byte-slot in the receive buffer. The receiver side sends a control message to the sender side denoting the transfer start and provides the start address of RDMA, its size, memory protection information and the value of the last byte in the receive buffer to the sender side through the control message. Note that a sender-initiate protocol must be used when `MPI_Irecv` with `MPI_ANY_SOURCE` is used because this step requires that `MPI_Irecv` be able to specify the sender side.

PR2 The receiver side waits either for change in the last byte of the receive buffer by polling or for arrival of a control message of transfer completion. The receiver side cancels the polling when the receiver side receives the control message. And then the receiver side exits `MPI_Wait`. Note that change in the receive buffer is observed to be occurring in-order in many processor-HCA combinations [5] and we exploit it for detecting the transfer completion.

PS1 The sender side enters `MPI_Isend` and checks arrival of the control message of transfer start sent by the receiver side. And then it initiates an RDMA write.

PS2 The sender side enqueues an IB command which sends a control message of transfer completion when the last byte in the receive buffer is the same as in the send buffer.

PS3 The sender side waits for completion of the RDMA write through processor-device communication via CQE. The sender side exits `MPI_Wait`.

The protocol makes the receiver set a constant value to the memory-slot residing at the end of the receive buffer. And then the receiver side polls on the memory-slot to monitor the change of value to detect completion of RDMA write. It relies on the fact that the value of the memory-slot is changed by the RDMA transfer. Therefore, the sender needs to send an extra control message in the situation the memory-slot has the same value as the corresponding memory-slot in the send buffer. The protocol has an issue that its effective latency is increased by up to the amount of latency overhead of sending one control message when the probability for the situation to happen is high. This would be the case when the constant happens to be the value frequently appears in the send buffer.

2.3 Switching Protocols Exploiting Timing Difference

Many studies [11, 10, 9] proposed rendezvous protocols which switch multiple protocols to reduce latency. Their methods select one protocol out of three based on the time difference between the time when the sender side enters `MPI_Isend` and the time when the receiver side enters `MPI_Irecv` to exploit the time difference to reduce latency. Protocols for the choice differ in which side (sender or receiver) initiates the transfer and whether to copy the contents of the send buffer to a read-only memory-area. However, each of the protocols still has the same issue as described above.

3 Protocol Design

We propose two improved protocols. The first one is an improved version of the GETCQE protocol and we call it the GET protocol. The second one is an improved version of the PUTNR protocol and we call it the PUT protocol. We explain them in the followings.

3.1 GET Protocol

We explain how to improve the GETCQE protocol to obtain the GET protocol. Step GR2 involves a costly processor-device communication via PCI bus. We replace it with a polling on the last byte of the receive buffer to eliminate the communication. We modify the related steps of the GETCQE protocol as in the followings to achieve it to obtain the GET protocol. **Fig. 3** shows a timeline diagram of the GET protocol.

GS1' The sender side enters `MPI_Isend` and sends a control message to the receiver side requesting an RDMA read. The control message contains the information for RDMA (start address of the send buffer, its size, memory protection information) and the value of the last byte in the send buffer.

GR1' The receiver side enters `MPI_Irecv` and checks arrival of the control message sent by the sender side. It writes the negative of the last byte in the control message to the last byte-slot of the receive buffer so that it can detect completion of the following RDMA read. And then it initiates an RDMA read.

GR2' The receiver side enters `MPI_Wait`. The receiver side polls on the last byte of the receive buffer to detect completion of the RDMA read. The receiver side sends a control message of the RDMA read completion to the sender side. The receiver side exits `MPI_Wait`.

Table 1: Parameters for evaluation environment

Component	Parameters
Node processor	Intel Xeon E5-2670, 2.601 GHz, 8-physical core, 16-logical core, 2-socket
HCA	Mellanox ConnectX-3, 6.79 GB/s
I/O bus	PCI Express 3.0, 8-lane, 7.88 GB/s

3.2 PUT Protocol

We explain how to improve the PUTNR protocol to obtain the PUT protocol. Step PS2 needs to send an extra control message when the last byte of the receive buffer has the same value as the send buffer. The probability would be high when using a constant value as the initial value of the last byte-slot in the receive buffer in step PR1 because the constant might be the value frequently appearing in the corresponding memory-slot in the send buffer. We randomize the initial value to reduce the probability. We modify the related step of the PUTNR protocol as in the followings to achieve it to obtain the PUT protocol. **Fig. 4** shows a timeline diagram of the PUT protocol.

PR1' The receiver side enters `MPI_Irecv`. And then it randomizes the last byte of the receive buffer. It sends a control message to the sender side denoting the transfer start and provides the start address of RDMA, its size, memory protection information and the value of the last byte in the receive buffer to the sender side through the control message.

4 Evaluation

We have conducted three types of evaluations. The first evaluation was conducted to compare the GET and PUT protocols with conventional protocols by utilizing a micro-benchmark which performs communications with a typical pattern. The second evaluation was conducted to compare the GET protocol with the PUT protocol by utilizing two micro-benchmarks and programs from NAS Parallel Benchmarks [7]. The third evaluation was conducted to compare the GET and PUT protocols with MVAPICH [5] by utilizing the first micro-benchmark. We explain the evaluation methodology common to all the evaluations.

We implemented a plug-in module controlling IB HCA for MPICH 3.0.1 [1] and implemented the proposed protocols by modifying MPICH with it. MPICH is compiled with Intel C Compiler version 13.0.0 20120731 with the `configure` option of `-enable-fast=O2, nochkmsg, notiming, ndebug`. We used a cluster computer with the specifications listed in **Table 1**. The arrangement of processes is intended to emulate the architecture of future cluster computers where many processes on a compute node share one HCA. Sixteen MPI processes are run on sixteen logical cores (hyper-threading cores) on one CPU-socket for this purpose. Logical cores are used to increase the number of cores per processor to emulate the architecture.

4.1 Comparing GET and PUT with conventional protocols

4.1.1 Comparing GET with conventional protocols

We first compared the communication latency of the GET protocol with that of the GETCQE protocol. We utilize a micro-benchmark, we call it BOWTIE, for this purpose.

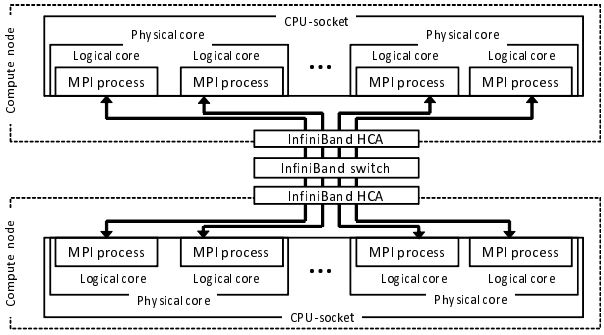


Figure 5: MPI process arrangement in the BOWTIE micro-benchmark.

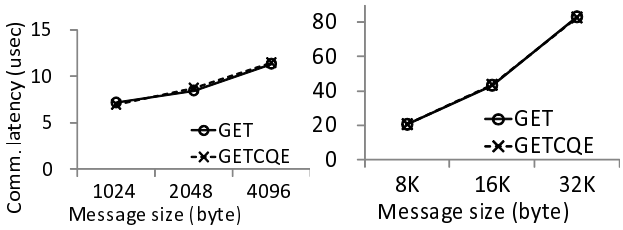


Figure 6: Latency of the BOWTIE micro-benchmark where 16 process-pairs are performing bidirectional communication using one pair of HCAs.

It utilzes 32 MPI processes. They utilize two compute nodes and the processes are arranged as shown in **Fig. 5**. BOWTIE makes each end of a process pair send messages toward the other end at the same time and waits for completion of the communication. The pseudo-code of BOWTIE is as follows.

```
/* nprocs: the number of processes */
/* each process with rank "r" performs this */
for(i = 0; i < NTRIAL; ++i) {
   MPI_Irecv from (nprocs >> 1) ^ r;
   MPI_Isend to   (nprocs >> 1) ^ r;
   MPI_Waitall;
}
```

The MPI library is modified so that all message transfers are performed by rendezvous protocol.

Fig. 6 shows the results. The X-axis shows the MPI message size specified by the user program. The GET protocol has almost the same latency compared to the GETCQE protocol even though it eliminates one processor-device communication from the GETCQE protocol. We explain the reason. Latency added to a rendezvous protocol by a processor-device communication is divided into two components and they are caused as follows. (1) HCA issues a PCI command and blocks other PCI commands and (2) processor issues a load instruction and causes a cache-miss. And (1) has a much larger effect than (2). The implementation of the GET protocol eliminates (2) but cannot eliminate (1) because of the restriction of the HCA utilized. Therefore, the improvement is limited.

4.1.2 Comparing PUT with conventional protocols

We compare the communication latency of the PUT protocol with that of the PUTNR protocol under the worst-case scenario. The worst-case scenario is the case where the last

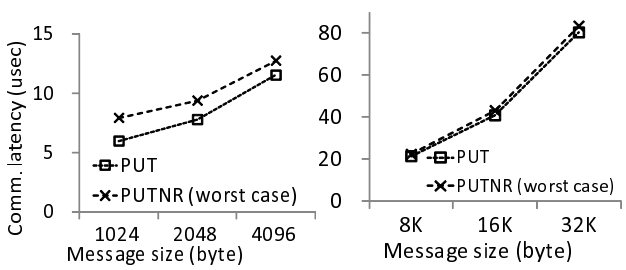


Figure 7: Latency of the BOWTIE micro-benchmark.

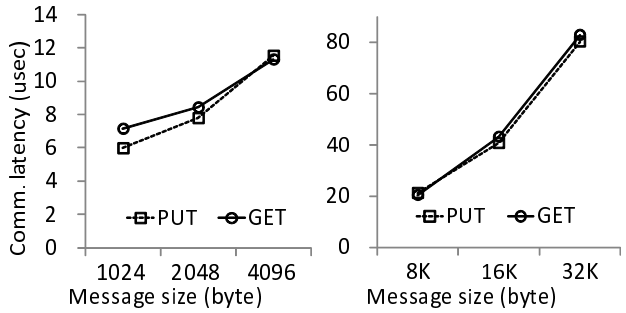


Figure 8: Latency of the BOWTIE micro-benchmark.

byte of the receive buffer always has the same value as the send buffer. We are trying to evaluate the maximum benefit of the PUT protocol over the PUTNR protocol in this way. We utilize BOWTIE for this purpose.

Fig. 7 shows the results. The PUT protocol reduces the latency by up to 24.60%. This is because the PUTNR protocol needs to send one extra control message when compared to the PUT protocol.

4.2 Comparing GET with PUT

We compare the GET protocol with the PUT protocol. We utilize BOWTIE micro-benchmark, `MPI_Alltoall` micro--benchmark and programs from NAS Parallel Benchmarks.

4.2.1 BOWTIE

First we compare the GET and PUT protocols by utilizing BOWTIE. **Fig. 8** shows the results. The PUT protocol reduces latency by up to 16.37% for messages of 1 KB, 2 KB, 16 KB and 32KB and has almost the same latency for messages of 4 KB and 8 KB compared to the GET protocol. This is because the PUT protocol sends fewer number of control messages than the GET protocol. Note that running many processes performing the same kind of IB message transfer at the same time accumulates latency overhead of individual processes, in this case overhead of sending an extra control message, when they are sharing a single resource with the least throughput, in this case the network wire. The accumulation occurs because sending an extra IB packet occupies the shared resource in an exclusive way to block IB message transfers of other processes.

4.2.2 MPI_Alltoall

We compare the GET and PUT protocols by utilizing a micro-benchmark which performs `MPI_Alltoall`. It utilizes 32 MPI processes. They utilize two compute nodes, and

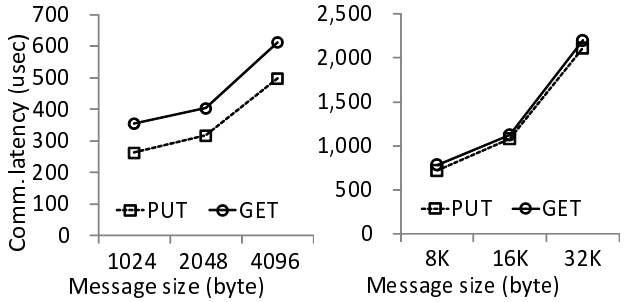
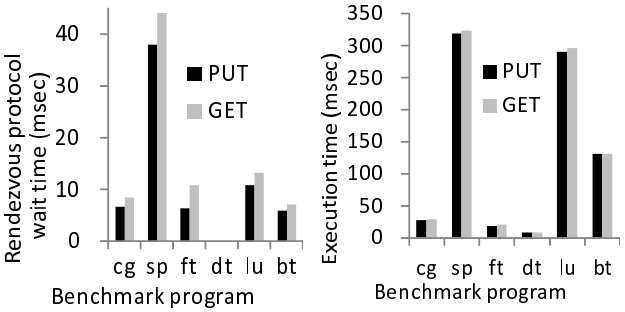


Figure 9: Latency of micro-benchmark where 32 processes are performing `MPI_Alltoall`.



(A) (B)

Figure 10: (A) Rendezvous protocol wait time and (B) execution time of programs from NAS Parallel Benchmarks.

sixteen processes utilize sixteen logical cores on each of the nodes. The following pseudo-code shows the steps of `MPI_Alltoall`.

```
/* nprocs: the number of processes */
/* each process with rank "r" performs this */
/* showing only the case nprocs % 4 == 0 */
for(j = 0; j < nprocs; j += 4) {
      for(k = 0; k < 4; ++k) {
         MPI_Isend to   (r + j + k) % nprocs;
         MPI_Irecv from (r - j - k) % nprocs;
      }
      MPI_Waitall;
}
```

The MPI library is modified so that all message transfers are performed by rendezvous protocol. **Fig. 9** shows the results. The PUT protocol reduces latency by up to 26.13% compared to the GET protocol. This is because the PUT protocol sends fewer number of control messages than the GET protocol.

4.2.3 NAS Parallel Benchmarks

We compare the GET protocol with the PUT protocol by comparing execution times of programs from NAS Parallel Benchmarks 3.3.1. We choose six programs, `cg`, `sp`, `ft` `dt`, `lu` and `bt`. Class `W` is used as the input data size. All programs are compiled with Intel C/Fortran Compiler version 13.0.0 20120731 with the option of `-O3 -xavx`. 64 MPI processes are utilized in all programs. 16 MPI processes run on sixteen logical cores and 4 compute nodes were utilized. The MPI library switches from the eager protocol to rendezvous protocol when a message size is larger than 4047

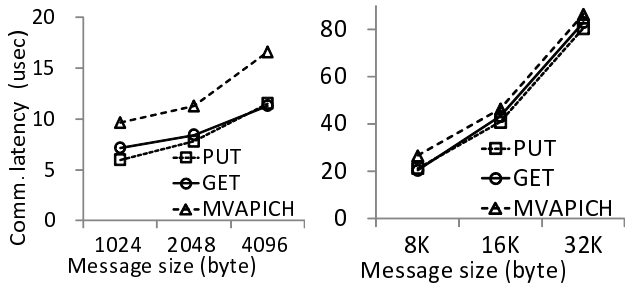


Figure 11: Latency of BOWTIE micro-benchmark.

bytes. We measure performance in two metrics. The first one is execution time of the program and the second one is time spent to wait for completion of rendezvous protocol sending messages across compute nodes. Note that it is the same as latency of rendezvous protocol minus the time of overlapped computation. We call this rendezvous protocol wait time.

Fig. 10 shows the rendezvous protocol wait time. The X-axis shows the program name and the Y-axis shows the time. The PUT protocol reduces rendezvous protocol wait time by up to 41.72% compared to the GET protocol. This is because the PUT protocol reduces one control message transfer compared to the GET protocol. `cg`, `sp`, `ft`, `lu` and `bt` exhibits relatively large reduction because they send many messages large enough to utilize rendezvous protocol and small enough to increase ratio of overhead of sending a control message to the total protocol latency.

Fig. 10 shows the execution time. The PUT protocol reduces the execution time by up to 11.14%. `cg` and `ft` exhibit relatively large reduction. This is because rendezvous wait time occupies a large portion of execution time in them and the reduction of rendezvous wait time is relatively large in them.

4.3 Comparing with MVAPICH

We compared the latency of the GET protocol and the PUT protocol with MVAPICH2-1.9b [5], which is used for many HPC cluster computers to see the practicality of the proposed protocols. We use BOWTIE for this purpose. The default rendezvous protocol of MVAPICH is as follows. (1) the sender side first sends a control message to the receiver side when the sender side enters `MPI_Isend`, (2) the receiver side responds to the control message and sends the second control message to the sender side, (3) the sender side responds to the second control message and initiates an RDMA write and sends the third control message of RDMA completion to the receiver, (4) the receiver side responds to the third control message and exits `MPI_Wait`, (5) the sender side confirms completion of RDMA write through processor-device communication via CQE and exits `MPI_Wait`.

Fig. 11 shows the results. The PUT and GET protocols reduces the latencies by up to 37.84% and 25.67% over MVAPICH. This is mainly because they reduce one or two control message transfers.

5 Conclusions

We revisited RDMA-based rendezvous protocols in MPI in the context of cluster computers with RDMA-capable HCA with many-core processors, and proposed two improved protocols. The conventional sender-initiate rendezvous protocols involve costly processor-device communications via PCI bus on detecting completion of RDMA transfer. The conventional receiver-initiate rendezvous protocols need to send extra control messages when the last byte of the receive buffer has the same value as the send buffer. These add latency to execution time of HPC applications. To eliminate processor-device communications, the first proposed protocol, GET, implements polling on a memory-slot in the receive buffer. In addition, to reduce extra control messages the second proposed protocol, PUT, randomizes the value of a memory-slot to poll in the receive buffer. We evaluated the benefit of the proposed protocols over the conventional methods by utilizing micro-benchmarks and have confirmed effectiveness of the GET protocol. And then we compared the GET protocol with the PUT protocol by utilizing programs from NAS Parallel Benchmarks. The results show that the PUT protocol reduces the execution times by up to 11.14%.

6 Acknowledgments

This research is partly funded by the National Project of MEXT called Feasibility Study on Advanced and Efficient Latency Core Architecture.

7 References

[1] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[2] T. Hoefler and T. Schneider. Optimization Principles for Collective Neighborhood Communications. *In Proc. of SC'12*, Article No. 98, 2012.

[3] InfiniBand Trade Association Std. InfiniBand™ Architecture Specification, Vol. 1, Rel. 1.2.1, 2007.

[4] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. *In Proc. of IPDPS'04*, 2004.

[5] J. Liu, J. Wu, and D. K. Panda. High Peformance RDMA-Based MPI Implementation over InfiniBand. *In Proc. of ICS'03*, pages 295–304, 2003.

[6] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 2012. http://www.mpi-forum.org/docs/mpi-3.0/.

[7] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[8] S. Pakin. Receiver-initiated Message Passing over RDMA Networks. *In Proc. of IPDPS'08*, pages 1–12, 2008.

[9] M. J. Rashti and A. Afsahi. Improving Communication Progress and Overlap in MPI Rendezvous Protocol over RDMA-enabled Interconnects. *In Proc. of HPCS'08*, pages 95–101, 2008.

[10] M. Small, Z. Gu, and X. Yuan. Near-optimal Rendezvous Protocols for RDMA-enabled Clusters. *In Proc. of ICPP'10*, pages 644–652, 2010.

[11] M. Small and X. Yuan. Maximizing MPI Point-to-Point Communication Performance on RDMA-enabled Clusters with Customized Protocols. *In Proc. of ICS'09*, pages 306–315, 2009.

[12] 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 Proc. of PPoPP'06*, pages 32–39, 2006.