2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Direct MPI Library for Intel Xeon Phi co-processors

Min Si*, Yutaka Ishikawa*†‡
*Department of Computer Science, University of Tokyo, Tokyo, Japan
Email: {msi@il.is.s, ishikawa@is.s}.u-tokyo.ac.jp
†Information Technology Center, University of Tokyo, Tokyo, Japan
‡RIKEN Advanced Institute for Computational Science, Japan
Masamichi Tatagi
Green Platform Research Laboratories, NEC Corporation, Kawasaki, Japan
Email:m-takagi@ab.jp.nec.com

***Abstract*—DCFA-MPI is an MPI library implementation for Intel Xeon Phi co-processor clusters, where a compute node consists of an Intel Xeon Phi co-processor card connected to the host via PCI Express with InfiniBand. DCFA-MPI enables direct data transfer between Intel Xeon Phi co-processors without assistance from the host. Since DCFA, a direct communication facility for many-core based accelerators, provides direct InfiniBand communication functionality with the same interface as that on the host processor for Xeon Phi co-processor user space programs, direct InfiniBand communication between Xeon Phi co-processors could easily be developed. Using DCFA, an MPI library able to perform direct inter-node communication between Xeon Phi co-processors, has been designed and implemented. The implementation is based on the Mellanox InfiniBand HCA and the pre-production version of the Intel Xeon Phi co-processor. DCFA-MPI delivers 3 times greater bandwidth than the 'Intel MPI on Xeon Phi co-processors' mode, and a from 2 to 12 times speed-up when compared to the 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode in communication with 2 MPI processes. It also shows from 2 to 4 times speed-up over the 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode in a five point stencil computation with an 8 processes * 56 threads parallelization by MPI + OpenMP.**

***Keywords*-MPI library; co-processor; Xeon phi; InfiniBand; direct communication; accelerator**

I. Introduction

Heterogeneous architecture clusters are currently in widespread use. A total of 62 supercomputers in the November 2012 Top500 List [1] are using accelerator/Intel Xeon Phi co-processor technology. There are two parallel execution models possible for use in these types of clusters: the host-assisted parallel execution model and the stand-alone parallel execution model. In the host-assisted parallel execution model, the computation is offloaded to accelerators by transferring computing data from host memory to accelerator memory. Communication between compute nodes is handled by the host CPU. In this execution model, data have to be moved between the accelerator and the host, and between the host and the remote host. These extra data movements result in communication overhead. GPGPU-based clusters follow this model because GPGPU cannot control communication devices. In the stand-alone parallel execution model, not only computation but also communication is executed in the accelerators. If this execution model were implemented, low overhead communication would be achieved. Although according to the PCI Express standard, the accelerator, a device in PCI Express, cannot configure devices and cannot receive interrupts from devices, a many- core based accelerator such as an Intel Xeon Phi co-processor can write commands to a communication device if the PCI Express device address is given by the host.

A direct communication facility for many-core based accelerators, called DCFA, has been introduced in our previous paper [2]. In DCFA, the host CPU configures and initializes the InfiniBand HCA, then the co-processor is able to transfer data directly to other co-processors using the HCA without host assists. However, this implementation has the following limitations: i) it was implemented only in co-processor kernel space, so the kernel has to be rebooted every time a program is executed; ii) the user has to write the host assist program, and it must be executed before kernel booting; iii) the components on the host and the co-processor haven't yet been defined clearly, so the host assist program has to be rewritten for every new communication program.

This paper designs an MPI library for Intel Xeon Phi co-processor clusters, called DCFA-MPI, to enable direct communication between Xeon Phi co-processors. As the basic communication library used by DCFA-MPI, DCFA is improved to function as a user space communication library providing the InfiniBand communication programming interface on Xeon Phi co-processors, thus allowing the communication programs to be executed multiple times without kernel rebooting. In the internal implementation of the functions requiring host assists, the corresponding assisting requests are sent to the host, the host module of DCFA, which implements the detailed assisting tasks, handles these requests. Therefore, the host-assisting implementation is hidden in DCFA, so users don't need to write host assist programs anymore. Moreover, since the interface of DCFA is uniform with the original host's InfiniBand Verbs library, a user can easily execute the host communication program on co-processors. Using the direct InfiniBand communication functions provided by DCFA, a DCFA P2P communication layer could be implemented in DCFA-MPI similar to the layers defined on the host. The MPI applications running on the host could be easily moved to co-processors, and benefit from direct InfiniBand communication.

After introducing the background of this paper in the following section, related work is introduced in Section III. In Section IV, the design and implementation of DCFA-MPI is presented. In Section V, the experimental environment, the evaluation experiments, and the results are presented. Finally, this paper will be concluded with a discussion of future work

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.179

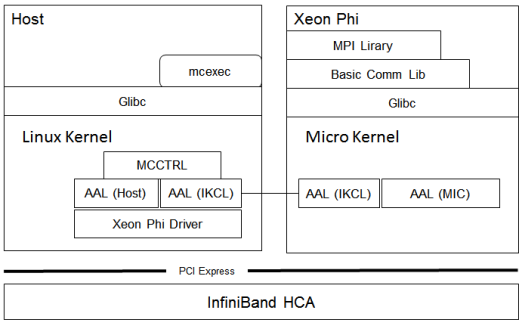


Figure 1. Xeon Phi co-processor Architectures

in Section VI.

II. Background

The InfiniBand Architecture (IBA) [3] is an industry-standard architecture intended to define a low latency, high bandwidth System Area Network (SAN) for interconnecting multiple independent computing nodes and I/O nodes. There are two communication modes can be used in user-level software: the Send/Receive mode, in which both the sender and the receiver must explicitly post requests, and the RDMA mode, in which only the sender side is required to post requests, but the address information of the receive buffer must be known by the sender side before data transfer. All the memory locations containing data buffers must be registered as a memory region before HCA can access them.

Figure 1 depicts the operating system kernel for the target architecture. The Linux kernel runs on the host CPU while a micro kernel runs on Xeon Phi co-processors. An Accelerator Abstraction Layer (AAL) is designed to hide hardware-specific functions and provide kernel programming interfaces. The AAL resides on both host and Xeon Phi co-processors, AAL(IKCL) is the inter-kernel communication layer between the host and the Xeon Phi co-processors. The mcctrl module is responsible for delegating system calls in the Xeon Phi co-processor kernel. The mcexec command is the host command used to load and execute executable file on a Xeon Phi co-processor; it is also the delegation process in the host user space.

DCFA-MPI is implemented based on a light-weight host MPI implementation, called the Yet Another MPI Implementation (YAMPII) [4]. It was developed at the University of Tokyo starting in December of 2001. It has been used as the MPI core of GridMPI, developed by the NaReGI project in Japan.

III. Related Work

A. Intel SCIF

Intel published an Intel MIC Platform Software Stack (MPSS), which consists of an embedded Linux, a minimally modified GCC, and driver software [5] [6]. The Linux system running on the Xeon Phi co-processor is designed to be booted by a host processor, and is connected to the host via a host driver. In this Linux environment, C, C++ and Fortran programs can easily be executed as runs on the host, and the OpenMPI and Intel MPI Library are also provided for parallel programming. The MPSS contains a direct Symmetric Communication Interface (SCIF) [7] [8], which is designed to be used as the communication backbone between the host processors and the Xeon Phi co-processors in a heterogeneous computing environment. It provides a uniform API for communication between the host processor and the Xeon Phi co-processors, and also between Xeon Phi co-processors. An SCIF driver consists of a user library and a kernel driver on both the host and the Xeon Phi co-processor. The SCIF driver provides the communication between the host and the Xeon Phi co-processor; it can be called by the user library directly, and can also support the InfiniBand drivers, which provides inter-node communication. A pair of HCA Proxy modules is defined for offloading some InfiniBand internal implementation to the host IB Proxy Daemon above the SCIF driver, and a user space InfiniBand Verbs library is also defined on the Xeon Phi co-processor, therefore inter-node InfiniBand communication between Xeon Phi co-processors is available.

Compared with DCFA, SCIF is designed to provide the communication API for communication between the host processors and the Xeon Phi co-processors in a heterogeneous computing environment; not only can the kernel modules use it to communicate with any other processors or co-processors, but the user application can also use it to transfer data. Although DCFA also contains components to communicate between host processor and Xeon Phi co-processor, they are only designed for offloading some InfiniBand functions, so the implementation is light weight and is designed only to be used in the kernel space. Moreover, both the DCFA and SCIF are designed to offload some InfiniBand internal implementations to the host. In DCFA, the mcexec process is launched at the time the executable file is loaded on the Xeon Phi co-processor, then it waits to handle the offloaded requests, not only from the InfiniBand module, but also from system calls. In SCIF, the host IB Proxy Daemon is dedicated to handling InfiniBand requests.

B. Intel MPI Library for the Intel MIC Architecture

In the Intel MPI Library for the Intel MIC Architecture, the node can be either a host processor or a Xeon Phi co-processor, and the following three programming modes are supported [9].

- 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode
  MPI ranks are located only on host processors and computing is offloaded to Xeon Phi co-processors, communication is still performed between host processors.
- 'Intel MPI on Xeon Phi co-processors' mode
  MPI ranks are located only on Xeon Phi co-processors, all messages are transferred to/from Xeon Phi co-processors directly.
- 'Symmetric' mode
  MPI ranks are located on both host processors and Xeon Phi co-processors, messages can be transferred to/from any core.

'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode has to move data to Xeon Phi co-processors before computing, and move data out before host communication, 'Intel MPI on Xeon Phi co-processors' mode

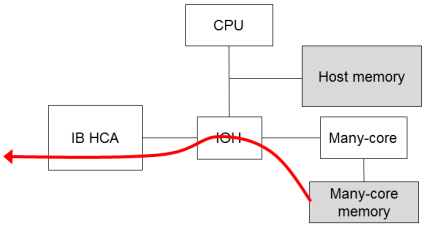


Figure 2. Data Transfer in DCFA

is similar to DCFA-MPI, in that both the computing and communication are performed on Xeon Phi co-processors directly. Both of them are compared with DCFA-MPI in Section V.

C. Intra-MIC MPI Communication using MVAPICH2

An early paper [10] introduces an early experience of intra-MIC communication using MVAPICH2. It enhances and tunes a shared memory based design in MVAPICH2 on the pre-production version of the Xeon Phi co-processors, and evaluated both P2P communication and collective communication. This implementation has not implemented inter-node communication yet, and runs entirely on the Linux environment provided by MPSS.

IV. Design and Implementation

A. DCFA

This paper introduces an MPI library for many-core based clusters, called DCFA-MPI to enable direct data transfer between Xeon Phi co-processors without the host assist. This library is based on DCFA [2], a direct communication facility for many-core based accelerators.

DCFA is intended to implement direct data transfer for many-core architectures (Figure 2), especially the Intel MIC architecture. Because a many-core unit is a device of the PCI Express bus, according to the PCI Express specification, it is not capable of configuring and initializing the InfiniBand HCA. This means that the host has to assist memory transfer between many-core units, and thus extra communication overhead is incurred. In DCFA, the internal structures of the InfiniBand HCA are distributed to both the memory space of the host and that of the many-core unit. After the host CPU configures and initializes the HCA, it obtains the addresses of both the HCA and the internal structures assigned by the host. Using the information given by the host and the internal structures assigned in the many-core memory area, the many-core unit may transfer data directly between other many-core units using the HCA without host assists. The implementation of DCFA is based on the Mellanox InfiniBand HCA and Intel Knights Ferry. Evaluation results show that, for large data transfer, the latency of DCFA delivers the same performance as that of host to host data transfer.

B. DCFA-MPI

DCFA-MPI is an MPI library over the DCFA facility, implementing direct P2P communication between Intel Xeon Phi co-processors. Since DCFA has defined the same InfiniBand communication programming interface on the Xeon Phi co-processor user space as the one defined on the host, The InfiniBand P2P communication layer could be implemented on a Xeon Phi co-processor, MPI communication primitives executed on the Xeon Phi co-processor may then transfer data directly to other Xeon Phi co-processors by issuing commands to the HCA (Figure 3).

1) Components in DCFA-MPI: DCFA-MPI defines the DCFA as an architecture-specified P2P communication layer, Figure 3 shows its components in some detail.

- DCFA IB IF
  The DCFA InfiniBand Interface defines the same InfiniBand functions as those defined in the InfiniBand Verbs library. Since some functions, such as resource initialization and memory region registration, have to send requests into the host kernel, a command mechanism is designed for offloading these requests to a host delegation process and then to the host kernel. In the internal implementation of these functions, the parameters are submitted to the Xeon Phi co-processor kernel by calling the corresponding system call, thus the DCFA CMD Client module is able to handle these requests and transfer them to the host delegation process, and the result of each request handled will then be returned. The DCFA IB IF then creates its own InfiniBand structures and saves the host results. For corresponding functions for send / receive data, the internal implementation is almost the same as the one defined in the host's IB Verbs, the post send/receive commands are issued to HCA directly.
- DCFA CMD server / client
  The DCFA CMD Client is carrying out the communication required for transferring the Xeon Phi co-processor user space requests to the host side, and preparing parameters for host offloading. For example, the virtual address of a user buffer and its size are submitted to the kernel for memory region registration, but the host delegation process needs the physical address for memory mapping, thus the DCFA CMD Client transforms the buffer's virtual address to a physical address and sends it to the host. On the host side, the DCFA CMD Server is registered as an extension of the delegation process; it receives requests and executes the corresponding host InfiniBand functions, registers all the InfiniBand objects created for Xeon Phi co-processor in a hash table, and publishes a hash key for later reuse.
- DCFA-MPI P2P communication layer
  This layer implements the P2P communication for upper MPI layers. Since the DCFA IB IF provides all the InfiniBand functions on Xeon Phi co-processor user space, P2P communication using InfiniBand RDMA operations could easily be implemented. The communication protocol is described in Section IV-B3
- DCFA-MPI CMD server / client
  As with the DCFA CMD server / client modules, these two components are responsible for offloading MPI functions. Although MPI P2P communications and collective communications are already implemented above the DCFA-MPI P2P communication layer, some heavy functions, such as communication using user defined data types could be candidates for offloading to the host CPU. The design and implementation of these offloading

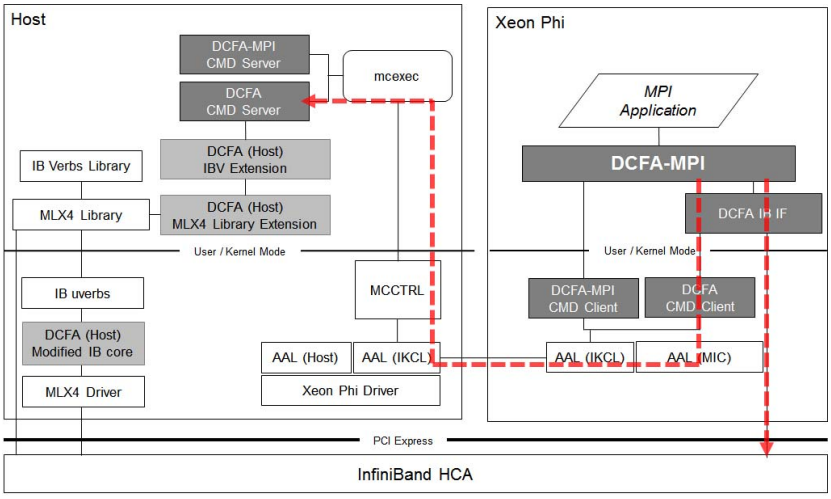


Figure 3. Implementation Components in DCFA-MPI

functions is planned in future work.

The DCFA (Host) IBV Extension, DCFA (Host) MLX4 Library Extension and DCFA (Host) Modified IB core components are designed so the host InfiniBand driver can access the Xeon Phi co-processor memory area [2].

2) The Process from MPI Library Installation to Application Execution:

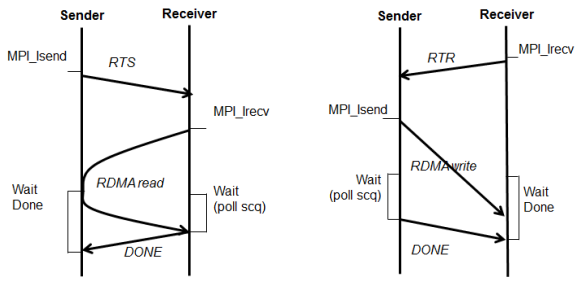
- Library Installation
  DCFA-MPI's installation process generates a static MPI library and user commands such as mpicc and mpirun. The static MPI library provides the MPI functions on the Xeon Phi co-processor, and the user application will be linked to this library in the compiling process using the mpicc command.
- Micro Kernel booting
  In the Xeon Phi co-processor kernel booting stage, some initialization tasks are performed. The host drivers for supporting the Xeon Phi co-processor system are inserted at this time.
- mpicc
  This command compiles the user source code to an executable MPI application. The mpicc compiles the user source code to an object file, which is then linked with the Xeon Phi co-processor static MPI library.
- mpirun
  As explained in Section II, the mcexec command executes the object file on the Xeon Phi co-processor. The mpirun command launches a process on each rank, and each process executes the object file using the mcexec command.

3) MPI Communication Protocol: MPI usually uses two internal protocols, Eager and Rendezvous, to implement the communication. In the Eager protocol, a message is sent to the receiver side regardless of the receiver's state. In the Rendezvous protocol, a handshake happens before a message is sent to the receiver side. The Eager protocol is usually used for small messages, data is copied to a preregistered global buffer, and then transferred to a global buffer on the receiver side, and then the receiver copies it to its receive buffer. For large data transfers, the overhead of extra data copies is expensive, thus the Rendezvous protocol is used. The handshake confirms the receive buffer on the receiver side is prepared, and then the sender can send data to the receive buffer directly.

A memory region registration operation on the Xeon Phi co-processor is much more expensive than that on the host because of the extra overhead of the offloading implementation which has been described in Section IV-B1. For reducing this operation, a buffer cache pool was designed for caching the most recently used memory regions. If the memory area of the user buffer exists in this cache pool, the memory region hit will be reused, otherwise a new memory region will be registered. However, the cache pool can only benefit applications which always reuse a few buffers. Since the data copy operation on the Xeon Phi co-processor spends less than 1 microsecond for 4Kbytes of data, DCFA-MPI uses a one-copy design for small messages. For large messages, not only the overhead of data copy but also the limited memory size of Xeon Phi co-processor should be considered, thus the zero-copy design was chosen.

InfiniBand provides two communication modes, the Send/Receive mode and the RDMA mode. In the zero-copy design for large messages, it's impossible to improve the performance of a sender first case using the Send/Receive mode. This is because, even if the sender sends first, it has to wait for the receiver to post a receive request with the prepared user receive buffer, and then it posts a send request with its user send buffer. Therefore, use of the RDMA communication mode was considered.

RDMA communication mode contains the RDMA write operation, which writes data from local scatter/gather elements (SGEs) to a remote buffer, and the RDMA read operation, which reads data from a remote buffer and writes to local SGEs. Current MPI implementations use several kinds of Rendezvous protocols using either RDMA write or RDMA read, to hide communication latency by overlapping computation with communication. DCFA-MPI uses another



(a) Sender First Rendezvous Protocol (b) Receiver First Rendezvous Protocol

Figure 4. Communication Protocols in DCFA-MPI

rendezvous protocol designed by the paper [11]. This protocol uses both RDMA write and RDMA read. The following four communication protocols are defined.

- Eager Protocol
  The sender decides to use the Eager protocol for small data transfers. It will copy the data to a global preregistered buffer, and then RDMA write an EAGER packet, which consists of an EAGER header SGE, the data SGE and a tail SGE, to a global preregistered RDMA buffer on the receiver side, and confirm its completion. Since it's ensured that the data payload of the receive buffer uses the same order as the SGEs defined in the sender request [3], the receiver will wait for the completion of data transfer by checking the tail of a received packet if the packet is EAGER type, then copy the data to the user receive buffer.
- Sender First Rendezvous Protocol
  The sender will send an RTS packet, including its buffer address and the memory region information, to the receiver, then wait for the receiver's DONE packet. When the receiver starts to receive, it must receive the RTS packet, then it starts an RDMA read by using the buffer information in the RTS packet. It will send a DONE packet to the sender when the read operation is finished (Figure 4(a)).
- Receiver First Rendezvous Protocol
  The receiver will send an RTR packet including the user receive buffer information, to the sender, then wait the DONE packet. The sender receives the RTR packet, and then it starts an RDMA write by using the buffer information included in the RTR packet. After the RDMA write is finished, the sender will send a DONE packet to the receiver (Figure 4(b)).
- Simultaneous Send/Receive Rendezvous Protocol
  In this situation, the sender will receive an RTR packet after it has sent an RTS; the receiver will also receive an RTS packet after it has sent an RTR. The sender will disregard the RTR and still wait for the receiver's RDMA read. The receiver will RDMA read by using the buffer data included in the RTS packet following the process of the Sender First Rendezvous Protocol(above).

A sequence id is assigned to every MPI send / receive request in order to make sure the send / receive order is correct. The sequence id must be unique in each pair of MPI processes, and any pair of send / receive requests must hold the same sequence id. Therefore, every packet received from another MPI process can be matched to the correct local request. If an MPI_ANY_SOURCE receive request comes, all the sequences for receive requests will be locked, thus any later receive requests will also be locked because they cannot get a sequence id. The MPI_ANY_SOURCE request will get its sequence id when it first meets the matching packet from a sender, then all the sequences locked will be unlocked and later receive requests can also get their ids and continue their receive process.

Since correct request order is ensured by the sequence ids, the deadlock caused by Eager / Rendezvous mis-predictions can easily be solved.

- The Sender Eager and Receiver Rendezvous Protocols
  The sender sends an Eager packet and its data, then the receiver recognizes this mis-prediction when it receives this Eager packet. The sending data should be smaller than the receiving data, and so the receiver will copy the data and complete this request. In the receiver first case, the sender drops the RTR packet sent from the receiver, because, thanks to the sequence id, it's sure that this packet is only for the current send request but not for later ones.
- The Sender Rendezvous and Receiver Eager Protocols
  The sender sends an RTS packet and the receiver only waits for the packet from the sender, then the receiver recognizes this mis-prediction when it receives this RTS packet. The sending data should be larger than the receiving data so the receiver will issue an MPI error.

4) Offloading Design for Send Buffers: It was expected that the communication performance of DCFA-MPI would be similar to that of the host MPI library, because DCFA, which transfers data between Xeon Phi co-processors via InfiniBand, is able to deliver the same performance as that of host to host data transfer. We evaluated DCFA again on a newer experiment environment which uses the Intel Xeon CPU E5-2670 and the pre-production version of the Xeon Phi co-processor. This result is much different from the previous one [2], which was obtained using the Intel Xeon CPU X5680 and the Intel Knights Ferry. Xeon Phi co-processor to Xeon Phi co-processor InfiniBand data transfer is always slower than host to host, by more than 4 times.

The following experiments were performed to find the communication performance bottleneck of the current Xeon Phi co-processor. i) InfiniBand RDMA write from the buffer

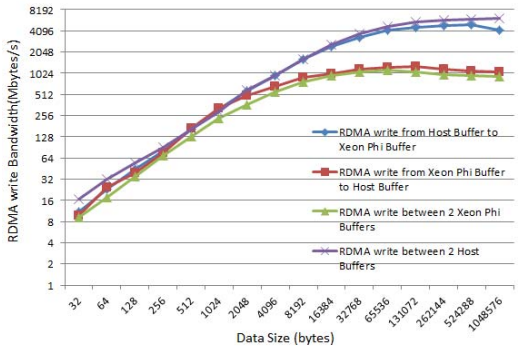


Figure 5. InfiniBand communication with different data transfer directions

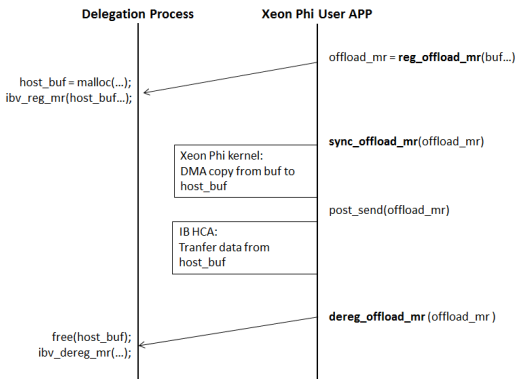


Figure 6. Offloading send buffer

allocated in the host memory (host buffer) to the buffer allocated in the remote Xeon Phi co-processor memory (Xeon Phi co-processor buffer); ii) InfiniBand RDMA write from a Xeon Phi co-processor buffer to a host buffer; iii) InfiniBand RDMA write from a Xeon Phi co-processor buffer to a remote Xeon Phi co-processor buffer ; iv) InfiniBand RDMA write from a host buffer to a remote host buffer. All of the experiments were conducted in Ping-Pong fashion, and the bandwidth is compared in Figure 5. The data transfer from a host buffer to a remote Xeon Phi co-processor buffer delivers the same bandwidth as host to host runs, however, the data transfer from a Xeon Phi co-processor buffer always gets much worse results whenever it is written into host memory or Xeon Phi co-processor memory. An InfiniBand RDMA write process consists of a DMA read from the send buffer to a local InfiniBand HCA, a data transfer from a local HCA to a remote HCA, and a DMA write from a remote HCA to a remote receive buffer, thus the performance bottleneck should be the DMA read from the Xeon Phi co-processor send buffer to the HCA.

For improving performance on the current experiment environment, this paper designs an offloading send buffer mode. Three functions are added to DCFA: reg_offload_mr, sync_offload_mr, and dereg_offload_mr. Figure 6 describes an example using these functions. The user application calls reg_offload_mr to register an offloading memory region, the corresponding host buffer is then allocated in the host delegation process and registered as an InfiniBand memory region, the information of this memory region is returned to the Xeon Phi co-processor side and used for later communication. Before starting the post send operation, data must be synchronized into the corresponding host buffer using the DMA engine, then the user application can issue a send request from the host buffer using its virtual address and the key of this memory region. Finally, function dereg_offload_mr destroys the offloading memory region on the Xeon Phi co-processor side, deregisters the memory region on the host side, and frees the host buffer. Using these functions, DCFA-MPI is easily able to register an offload memory region for a send buffer, and synchronize the latest data from a Xeon Phi co-processor to its host buffer before performing an InfiniBand send operation, thus the latest data can be transferred to a remote MPI process. This design benefits

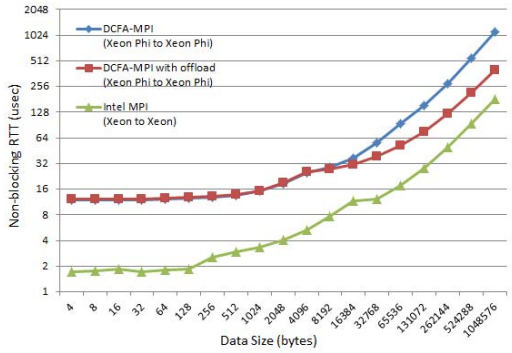


Figure 7. Evaluation of DCFA-MPI with offloading send buffer design using non-blocking inter-node MPI communication

large message communication because of the overhead of data synchronization. The message size at the beginning of offloading should be tuned in a different server environment. In our environment, an offloading send buffer starting from 8Kbytes shows the best performance. Figure 7 shows the RTT of non-blocking inter-node communication using MPI_Isend and MPI_Irecv. The offloading design improves the performance of large messages and is getting closer and closer to the host performance. It is only 2 times slower than the host at 1Mbytes. Figure 8 also shows that the DCFA-MPI with offloading send buffer design improves the inter-node communication bandwidth to 2.8Gbytes/sec.

Table I
SERVER ARCHITECTURE USED IN THE EXPERIMENTS

CPU	Intel Xeon CPU E5-2670 0 @ 2.60GHz x 16
InfiniBand HCA	Mellanox Technologies MT27500 Family [ConnectX-3]
Card	Pre-production of Intel Xeon Phi x 1
Operating System	Red Hat Enterprise Linux Server 6.2
Intel MPSS	2.1.4982-15
Intel MPI Library	4.1.0.027
Intel C++ Compiler	Composer XE 2013.0.079
InfiniBand driver for Intel MPI	OFED-1.5.4.1
InfiniBand driver for DCFA-MPI	MLNX_OFED_LINUX 1.5.3-3.1.0-rhel6.2-x86_64

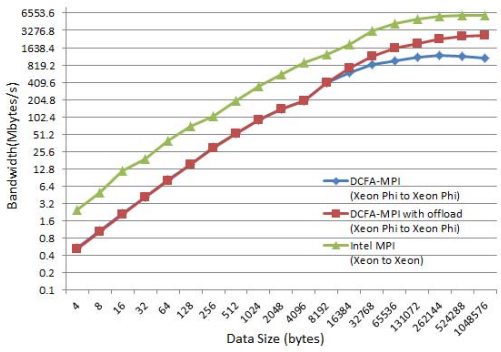


Figure 8. Evaluation of DCFA-MPI with offloading send buffer design using MPI inter-node communication bandwidth

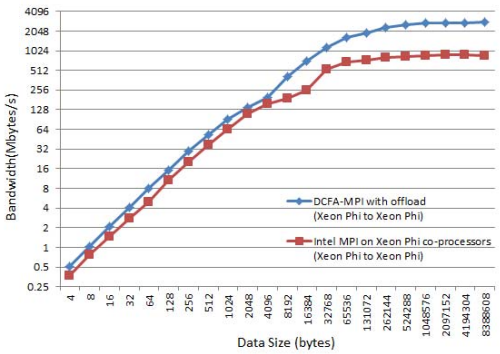


Figure 9. Inter-node communication bandwidth comparison between DCFA-MPI and 'Intel MPI on Xeon Phi co-processors' mode

V. Evaluation

We performed all experiments on an 8 node cluster with the configuration given in Table I. The InfiniBand driver for Intel MPI is different from the one used for DCFA-MPI, because Intel MPSS with OFED only supports the OFED-1.5.4.1 InfiniBand driver, but DCFA is implemented based on the MLNX_OFED_LINUX 1.5.3-* driver. The DCFA-MPI evaluation is based on a comparison with 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode and 'Intel MPI on Xeon Phi co-processors' mode. Three experiments were performed.

The first experiment compares the inter-node communication bandwidth of DCFA-MPI and 'Intel MPI on Xeon Phi co-processors' mode because both of them perform direct MPI communication between Xeon Phi co-processors. The bandwidth result is calculated using the round trip latency of MPI blocking communication. 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode is not included because its MPI communication is only between hosts. Two MPI processes are launched on different nodes. As shown in Figure 9, DCFA-MPI always outperforms 'Intel MPI on Xeon Phi co-processors' mode, and delivers a 3 times speed-up after the 1Mbytes message size. For 4bytes round trip blocking communication, the 'Intel MPI on Xeon Phi co-processors' mode spends 28 microseconds while the DCFA-MPI only spends 15 microseconds. For large messages, because of the bottleneck discussed in Section IV-B4, 'Intel MPI on Xeon Phi co-processors' mode cannot get bandwidth greater than 1Gbytes/s, but DCFA-MPI benefits from the offloading send buffer design, so bandwidth can grow up to 2.8Gbytes/s.

The second experiment uses a communication-only application to compare DCFA-MPI and 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode. Two MPI processes are launched on different nodes. Since DCFA-MPI is running entirely on Xeon Phi co-processors, the computing data always stays in Xeon Phi co-processor memory, and only inter-node MPI communication is required. However, 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode is running mainly on the host and offloads computing to the Xeon Phi co-processors, thus data have to be transferred into Xeon Phi co-processor memory before computing, and transferred out after computing is finished, then inter-node MPI communication is performed on the host. Table II lists its communication data size. Non-blocking communication using MPI_Isend and MPI_Irecv is used for inter-node communication. As shown in Figure 10, DCFA-MPI is 12 times faster than 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode when the data transferred is less than 128bytes, because 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode has to spend a fixed amount of time to initialize offloading and perform data transfer between the host and the Xeon Phi co-processor, even though this application has been optimized based on the following policies:

- eliminate offload Initialization from the communication loop;
- keep the buffer persistent and only transfer necessary data;
- align the buffer on a 4Kbytes page boundary and make sure data is a multiple of 4Kbytes to get fastest data transfer over PCI express;
- overlap offloading data transfer and MPI communication using the double buffer method;

With the increase of transferred data, the 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode is gradually getting closer to DCFA-MPI because the offloading overhead is becoming smaller than the latency of inter-node MPI communication, but DCFA-MPI also maintains better performance than 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode, 2 times faster when the data transferred is larger than 512Kbytes.

The third experiment compares DCFA-MPI, 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode and 'Intel MPI on Xeon Phi co-processors' mode using a five point stencil computation, and parallelizes

Table II
Communication data size of the communication only application

Data size	X bytes
Offloading Data	Copy In X bytes + Copy Out X bytes
MPI Communication Data	Send X bytes + Receive X bytes

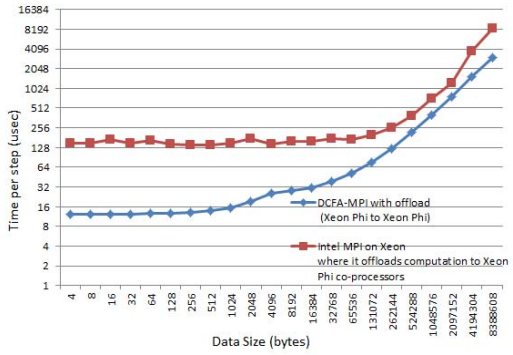


Figure 10. Comparison between DCFA-MPI and 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode using communication only application

computing using both MPI and OpenMP [12]. Computing data are separated into MPI processes on different nodes, and then OpenMP threads parallelize local computing. Because demand paging hasn't been implemented on our Xeon Phi co-processor kernel yet, the memory consumption of the test application is strictly limited. Our experiment uses a 1282*1282 problem size, and table III describes the data size used for computing and communication. Since the inter-node data exchange only happens at adjacent boundaries, the MPI communication data size is 10Kbytes, and the data required to be transferred in / out at every offloading computing stage is also 10Kbytes, because only this area is exchanged by host MPI, all the other areas can persistently be kept on the Xeon Phi co-processors. Figure 11 shows the average processing time with 100 iterations. When the number of MPI processes is 1, both the MPI communication data and offloading data are 0bytes because MPI doesn't exchange any data. DCFA-MPI and 'Intel MPI on Xeon Phi co-processors' mode deliver the same performance, but 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode still spends double the time compared with DCFA-MPI and 'Intel MPI on Xeon Phi co-processors' mode, this is because of the fixed overhead of preparing computation offloading. With the increase in the number of MPI processes, the computing data size on every MPI process becomes smaller, but the offloading data and MPI communication data are not changed, thus the gap between DCFA-MPI and 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode becomes larger. With the increase in the number of OpenMP threads, the gap between DCFA-MPI and 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode becomes larger, because OpenMP decreases the total computing time. DCFA-MPI delivers in the worst case, a 2 times speed-up with 1 MPI process and without OpenMP, and in the best case, a 4 times speed-up with 8 MPI processes and 56 OpenMP threads. The results of DCFA-MPI and 'Intel MPI on Xeon Phi co-processors' mode do not show a big difference, because in both cases the computing is on the Xeon Phi co-processors and the only communication is the MPI communication between Xeon Phi co-processors. Figure 12 compares their speed-up over the serial program. In the cases using over 1 MPI process or

Table III
COMMUNICATION DATA SIZE OF THE FIVE POINT STENCIL COMPUTATION APPLICATION

Problem Size (Number of Points)	1282 * 1282
Computing Data	12Mbytes
Offloading Data	Copy In 10Kbytes + Copy Out 10Kbytes
MPI Communication Data	Send 10Kbytes + Receive 10Kbytes

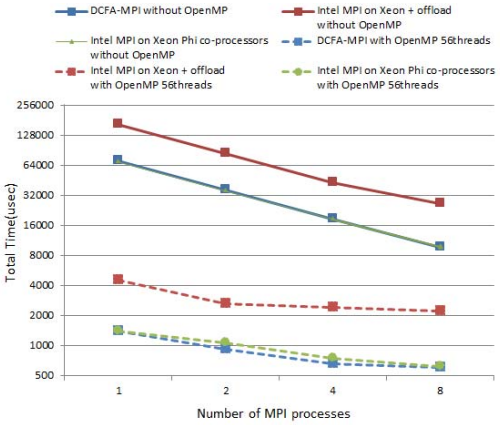


Figure 11. Processing time of five point stencil computation with different number of MPI processes using DCFA-MPI, 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode (Intel MPI on Xeon + offload) and 'Intel MPI on Xeon Phi co-processors' mode

over 4 OpenMP threads, 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode shows poorer results than DCFA-MPI and 'Intel MPI on Xeon Phi co-processors' mode, because the time of offloading computing data occupies a greater proportion when the computing data is reduced by increasing MPI processes and the computing time is reduced by increasing OpenMP threads. With 8 MPI processes and 56 OpenMP threads, DCFA-MPI delivers a 117 times speed-up, 'Intel MPI on Xeon Phi co-processors' mode delivers a 113 times speed-up, and 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode only delivers 74 times speed-up.

VI. CONCLUDING REMARKS

This paper has implemented an MPI library based on the DCFA, called DCFA-MPI, to provide direct Xeon Phi co-processor to Xeon Phi co-processor inter-node MPI communication. Since DCFA provides the same InfiniBand communication interface on the Xeon Phi co-processor user space, the InfiniBand P2P communication layer of DCFA-MPI can be implemented on a Xeon Phi co-processor. Therefore, floating-point computing and P2P communication are to be performed directly on the Xeon Phi co-processor.

However, in our current evaluation machines, the DMA read from the Xeon Phi co-processor to the InfiniBand HCA has limitations in bandwidth, and as a result, the InfiniBand communication from the Xeon Phi co-processor is slower than host to host communication, by more than 4 times. This paper designs an offloading send buffer mode for improving performance in such cases. The evaluation using non-blocking MPI communication shows that the Xeon Phi co-processor to Xeon Phi co-processor communication using

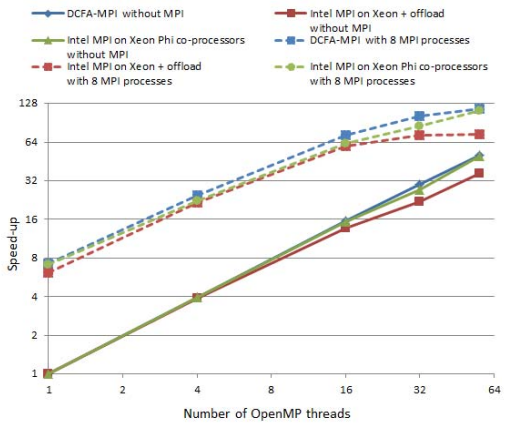


Figure 12. Speed-up of five point stencil computation with different number of OpenMP threads using DCFA-MPI, 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode (Intel MPI on Xeon + offload) and 'Intel MPI on Xeon Phi co-processors' mode when comparing to the serial program

DCFA-MPI is only 2 times slower than host to host for large messages.

The DCFA-MPI(MPI on Intel Xeon Phi co-processor) has also been compared with the 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode and 'Intel MPI on Xeon Phi co-processors' mode. It delivers 3 times greater bandwidth than 'Intel MPI on Xeon Phi co-processors' mode, and from 2 to 12 times speed-up when compared to 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode in communication with 2 MPI processes. It also shows from 2 to 4 times speed-up over 'Intel MPI on Xeon where it offloads computation to Xeon Phi co-processors' mode in a five point stencil computation with an 8 processes * 56 threads parallelization by MPI + OpenMP.

For future research, some heavy functions, such as collective communication and communication using user defined data types are planned to be offloaded to the host CPU. Because relative to multi-cores, the Xeon Phi co-processor, an example of a many-core architecture, has small memory caches per core and limited memory bandwidth per core, the footprint in the cache during execution of both the application and the communication library should be minimized.

ACKNOWLEDGMENTS

This work is partially supported by the "feasibility study on Future HPC R&D in Japan" project and the CREST project in Japan.

REFERENCES

[1] "Top500 supercomputing sites," http://www.top500.org.

[2] M. Si and Y. Ishikawa, "Design of direct communication facility for many-core based accelerators," in *IPDPS2012 CASS workshop*, 2012, p. 918.

[3] *InfiniBand™ Architecture Specification, Volume 1, Release 1.2.1*, http://members.InfiniBandta.org/kwspub/spec/vol1r1_2.zip, InfiniBand Trade Association Std., 2007.

[4] Y. Ishikawa, "Yampii, yet another mpi implementation," *IPSJ SIG Technical Report*, vol. 2004, no. 81, pp. 115–120, 2004-07-30. [Online]. Available: http://ci.nii.ac.jp/naid/110002914121/

[5] James Reinders, "Knights corner: Open source software stack," http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack/.

[6] ——, "Knights corner micro-architecture support," http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-micro-architecture-support/.

[7] "Intel(r) many integrated core (mic) platform software stack (mpss) knights corner pci* express co-processor card software development platform," http://registrationcenter.intel.com/irc_nas/2618/readme.txt.

[8] "Intel manycore platform software stack (mpss)," http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

[9] S. McMillan, "Programming models for intel xeon processors and intel mic architecture," http://www.tacc.utexas.edu/documents/13601/d9d58515-5c0a-429d-8a3f-85014e9e4dab.

[10] S. Potluri, K. Tomko, D. Bureddy, and D. K. Panda, "Intra-mic mpi communication using mvapich2: Early experience," *TACC-Intel Highly-Parallel Computing Symposium*, 2012.

[11] R. M. J. and A. Ahmad, "Improving communication progress and overlap in mpi rendezvous protocol over rdma-enabled interconnects," in *Proceedings of the 2008 22nd International Symposium on High Performance Computing Systems and Applications*, ser. HPCS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 95–101. [Online]. Available: http://dx.doi.org/10.1109/HPCS.2008.10

[12] "Openmp," http://openmp.org/wp/.