Developing an OpenSHMEM model over a Switchless PCIe Non-Transparent Bridge Interface

Seung-Ho Lim
Div. of Computer and Electronic Systems Engineering, Hankuk University of Foreign Studies, Yongin, Korea
lim.seungho@gmail.com

Ki-Woong Park
Dept. of Computer and Information Security, Sejong University Seoul, Korea
woongbak@sejong.ac.kr

Kwangho Cha
Center for Supercomputer Development Korea Institute of Science and Technology Information, Daejeon, Korea
khocha@kisti.re.kr

***Abstract*—OpenSHMEM is an emerging parallel programming model in distributed systems that supports partitioned global address space (PGAS). It was originally developed for Cray systems, and it was subsequently adopted by numerous vendors of the InfiniBand and Ethernet network interfaces, since most high-performance computing (HPC) systems deploy switch-based interconnect networks using InfiniBand and Ethernet technologies. Recently, Peripheral Component Interconnect Express (PCIe) has become one of the most promising candidates for deploying cost-effective popular HPC systems because of its low cost and powerful features, as well as the Non-Transparent Bridge(NTB) technology (interconnect interface for PCIe). However, there is little work done on implementing the OpenSHMEM library for PCIe systems. Herein, we introduce a prototype of a switchless interconnect network with PCIe NTB. In our switchless interconnect network system, the computing nodes are interconnected via the PCIe NTB interface. Based on the PCIe NTB interconnect network, we implemented the OpenSHMEM programming interface to support the PGAS mechanism for PCIe NTB interconnect networks. Our design and implementation of an OpenSHMEM programming model via PCIe NTB shows that is feasible and possible to have a PCIe-based, cost-effective, high performance interconnect network in a high-performance programming model.**

***Keywords*-PCIe, Non-Transparent Bridge, Interconnect Network, OpenSHMEM, PGAS**

I. Introduction

High performance computing (HPC) systems are widely used to run numerous high-performance applications such as in scientific computing, big data and cloud computing-based applications. In HPC systems, hundreds of computing nodes are connected via a high-performance interconnect network to get high throughput and low latency. Most HPC systems deploy the interconnection network with InfiniBand, Ethernet, and Fiber channel interfaces [1]. Although traditional interconnect networks can provide high bandwidth, their cost and power dissipation are augmenting with the increasing volume of HPC.

Among those interconnect technologies, PCI Express [2] is one of the most promising candidates for deploying cost-effective popular HPC systems due to its low cost and powerful features [4]. The PCI Express technology was originally developed to support connectivity between host processors and peripheral I/O devices [2], [4] with the so-called bridge specification. There are two bridging concepts in PCIe specification. On one hand, PCIe transparent bridge connects the processor with I/O devices and the process manages the I/O devices within its own address space. On the other hand, PCI Non-Transparent Bridge(NTB) is a connection between two or more independent host processors, supporting inter-host communications in a single PCIe domain. It provides a function that translates PCI transactions from one PCIe hierarchy into the corresponding transactions in another PCIe hierarchy.

For decades now, PCIe NTB has connected two PCI-based systems by mapping some address regions into each other [5], [6], mainly to check connected host processors such as with heartbeating. Recently, PCIe has been used as an interface within the interconnect network for HPC systems [5]. Multiple host complexes, in a single PCIe domain, has long been proposed, however, no commercial implementations of true Multi-Root IO Virtualization (MRIOV) switches or devices exist, and there is no sign to be coming for the near future [6]–[8].

Meanwhile, one of the most important applications for HPC systems is scientific computing with the parallel programming interface model. There are two popular and competing parallel programming models for distributed systems: one is message passing such as MPI and the other is partitioned global address space (PGAS) such as OpenSHMEM. Over time, message passing has become the de-facto standard running on distributed systems. However, PGAS has been an emerging interface to supplement the message passing's weakness in parallel programming model with the strength of one-sided communication. OpenSHMEM [10]–[12] is an open standard introduced to support parallel programming interface by providing PGAS across multiple hosts. It was originally developed for Cray systems, and was subsequently adopted by numerous network interface vendors providing InfiniBand and Ethernet connectivity. However, there are few reports on the development of a PCIe-based OpenSHMEM interface.

In this study, we designed and implemented an OpenSHMEM programming model interface via a PCIe NTB interconnect network system. For this, we first developed a prototype of switchless interconnect network system with PCIe NTB. In the prototype, computing nodes are connected to each other via PCI NTB interface to construct a switchless interconnect network such as a ring network. We implemented a data sharing mechanism with a remote direct memory access (RDMA) interface of PCI NTB within the switchless interconnect networks. Then, we implement OpenSHMEM programming interfaces to support the PGAS mechanism for PCIe NTB interconnect networks, such as the initialization of symmetric shared memory, putting and getting data, to and from the shared memory, through the PCIe NTB interface. Our design and implementation of the OpenSHMEM programming model via PCIe NTB shows that is feasible and possible to have a PCIe-based, cost-effective, high-performance interconnect network in a high-performance programming model.

II. Background

A. PCIe and Non-Transparent Bridge(NTB)

PCIe [3] is a third generation I/O bus, used to interconnect peripheral devices, based in its processor bus. It is a serial, point-to-point interconnect for communication between devices. The point-to-point connection allows large scale and higher level transmit/receive frequencies, which leads to high speed data movements. More specifically, PCIe Gen1 has a data rate 2.5 GB/s, Gen2 has a data rate 5 GB/s and Gen3 has a data rate up to 8 GB/s. While connecting peripheral devices to the PCIe interface, a host processor manages the connected devices as a root complex, with the enumeration of the entire memory space. In this situation, multiple processors could not be connected to PCIe interconnect since they would attempt to service the same address leading to collisions.

PCIe Non-Transparent Bridge (NTB) is a method for PCIe systems to handle multiple processors. NTB is functionally similar to the transparent bridge except that there is a processor on each side of the bridge, and each processor has its own independent address space domain. In the NTB environment, the PCIe bridge translates addresses that across from one memory address space to the other space through translation registers, in the so-called memory window. The address translation of one host to another host is described in Fig. 1. With this address translation, two processors connected to PCIe NTB can share independent address spaces, each one belonging to a processor, and transfer data through this independent address space.

In addition to that, as noted in the PCIe specification, two processors can share some small data set through some special registers, i.e., eight or more 32bit registers, called $ScratchPad$ registers. After one host writes data to one ScratchPad register, another host can access and read directly

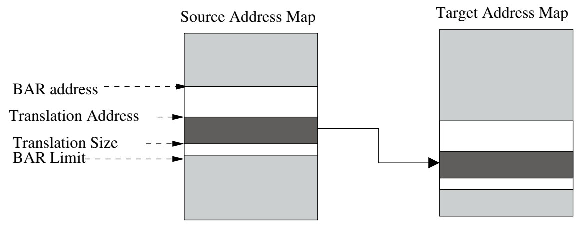


Figure 1. Address Translation Between Two processors in PCIe NTB

from that register. Also, one processor can cause an interrupt to the other processor through register called $Doorbell$ register. There are sixteen doorbell interrupts that can be set or cleared, as well as masked.

B. OpenSHMEM API

There are two parallel programming interface model for multiple host distributed systems; one is Message Passing (MP)-based and the other is Partitioned Global Address Space(PGAS)-based. Cray Research, Inc. developed the first SHMEM library for it3 T3D systems in 1993 [9] to support PGAS in a distributed shared memory system. After that, many other vendor-specific SHMEMs were implemented over the years. Recently, the U.S. Department of Defense funded a collaboration between Oak Ridge National Laboratory and University of Houston to develop specification for a uniform SHMEM API called OpenSHMEM [10]. Thus, OpenSHMEM is an open standard for all SHMEM implementations. The first version of OpenSHMEM Specification was released in early 2012. It defines an API for a set of useful functions for efficient parallel programming for applications on shared distributed memory systems. Although there are several OpenSHMEM library implementations such as SGI OpenSHMEM library, Cray OpenSHMEM library and other InfiniBand Cluster library, there is little work done on an OpenSHMEM API implementation over a PCIe interconnect network.

There are several essential features for SHMEM library implementations. OpenSHMEM gives one-sided communication operations that are locally blocking, meaning they return once the local buffer is available for reuse regardless of the locking of remote buffers. Along with the one-sided communications, it should support remote atomic memory operations, broadcasts, barrier operations, reductions, distributed locking and synchronization primitives. Specifically, with the distributed shared memory environment, OpenSHMEM should support the API that defines certain data objects called symmetric data to express remote memory access. The symmetric data objects are allocated dynamically through APIs. All processes called PEs can access symmetric data objects through the APIs that access symmetric data objects remotely, the so-called Put (remote write) and Get (remote read). The variant of Put and Get should be provided along with one-sided communication operations to access those

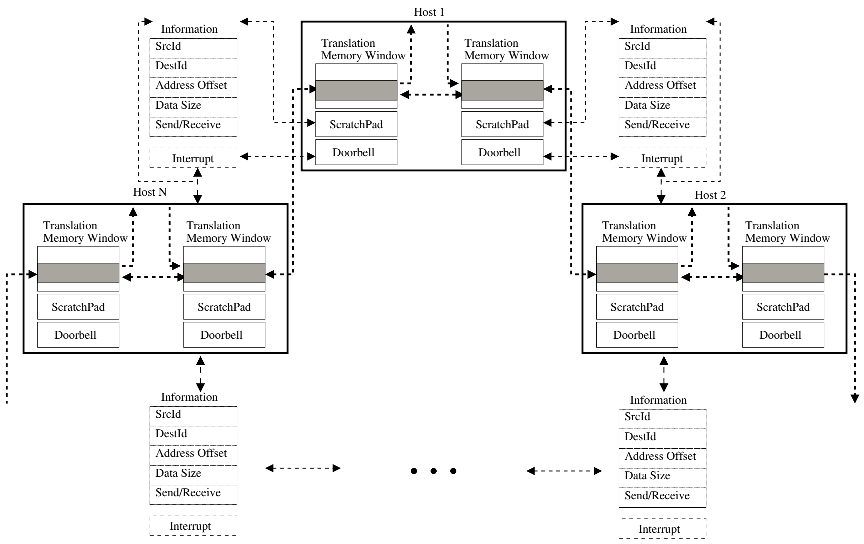


Figure 2. Diagram of the PCIe NTB-based Interconnect network system. Hosts are connected to each other through the NTB port, and the memory window is translated, non-transparently, through the address translation registers of NTB. There are some ScratchPad registers that are accessed directly by bothhosts, and some Doorbell registers to give interrupt signals to each other.

symmetric data objects. In addition to the above features, OpenSHMEM should provide atomic locking and barrier operations for synchronization of symmetric data objects.

III. Implementation of OpenSHMEM over PCIe NTB

This section describes the implementation details of OpenSHMEM API library over PCIe NTB interconnect network system. First we describe the structure and operation of the NTB-based interconnect network system, and later we explain the implementation of OpenSHMEM API.

A. *Setup for the NTB-based Interconnect Network*

There are four transaction types defined by the PCIe standard: Memory read/write, I/O read/write, configuration read/write, and message. Among those PCIe transactions, memory read/write and I/O read/write transactions are address-translated through NTB base address registers (BARs). Basically, NTB provides address translation between two separate address space of two connected hosts. According to the standard, each NTB port has six BARs in its PCIe Type 0 header, and each BAR can be responsible for 32-bit or two consecutive BARs are responsible for 64-bit address translation between two independent hosts. If the PCIe switch supports several NTB port connections, say 4 or 8 ports, each independent processor is connected to each of the NTB ports to setup multi-root system within one PCIe domain. Although virtual NT interconnect is made within the multi-root system, each NTB address translation is performed on each individual NT port, with the address maps of every NT port connected to some independent host which is connected to that NT port. For instance, if there are three independent hosts connected to NT ports, each NT port has two separate address translation windows corresponding to the other two hosts.

Multiple host complexes in a single PCIe domain were proposed some time ago [7]; however, there are no commercial implementations of true multi-root switches or devices, and there is no sign that any will be available in the near future [8]. As a result, it is difficult to implement an NTB-based interconnect network system with one PCIe domain with PCIe switches. In this study, to implement a switchless interconnect network for multiple hosts system through PCIe NTB functions, two NTB adapters are connected to a single-CPU-based host, and the NTB host adapter is connected to another NTB port which is connected to another host processor via PCIe cable. The two connected NTB adapters make an NTB upstream and downstream channel, enabling address translation between the two hosts. Further NTB connections will configure a ring topology in a single PCIe domain. For instance, the ring topology of an interconnected multi-host system is described in Fig. 2. In the figure, there are two memory windows at each host, each NTB port is connected to an NTB port of a neighboring host, so one BARs is setup for each NTB port with incoming and outgoing address spaces. From the point of view of each host, there are two address spaces used to translate addresses of each individual region. The memory window is used to transfer and receive data from one host to another connected host. In addition to the memory windows, each NTB port has Doorbell registers and Scratchpad registers. There are eight 32bit Scratchpad registers for each NTB port, but these are shared by two connected NTB links. Hence, if one NTB port writes some information to the ScratchPad registers, the other side can directly read the information in that area of the ScratchPad register. The Scratchpad registers can be used deliver some small amount of information synchronously from one host to another. The Doorbell registers are used to send interrupt signals from one NTB port to a connected NTB port. Generally, there are sixteen doorbell interrupts that can be set or cleared, as well as masked. One processor can send an interrupt signal to another processor through one of the doorbell registers.

At an application level, multiple hosts transfer and receive, to and from the shared memory system, through the PCIe NTB-based switchless interconnect network. There is a pre-setup step that should be done before an application shares data through the multi-hosts distributed system. The first step for constructing the interconnect network, is to assign a host Id for each host, and two address spaces are allocated for any pair of NTB ports, and each address region allocated, i.e., address offset and size of the window, is exchanged with the corresponding neighbor. With those address regions, each host can set the address regions for connected neighbors, which are used for transferring and receiving data.

After the initial step, each host can transfer data through the shared address space, as a result, it can send data to a destination or receive data from a source host with a specified host Id. The data transfer is done as follows. When a host tries to send data to another host, it writes data from its own memory space to a shared address space region. The data can be written with RDMA [15] supported by NTB RDMA interface, or directly with a memcpy operation. When data is being written, the destination address is translated to the address region of the connected side through the address translation map of BAR, so the data is visible to the host connected to NTB. After the data is written, the host sends through ScratchPad registers, information such as the source host Id(SrcId), destination host Id(DestId), Address offset, Data size, and flag for Send/Receive. Then the host triggers an interrupt with a doorbell register as an alert for transferring data. When a connected neighbor host receives an interrupt triggered by a sending host, it identifies the source host and the destination by reading ScratchPad registers. If the destination host is itself, the host gets the data directly from the shared address region, and the transfer is complete. If the destination host is not itself, the host should pass the received data to another connected host. The data passing operation is similar to the data transfer from one host to another, as described above. Those transferring sequences are complete when the destination host gets the data from a connected host. With the above operations, data can be shared among multiple hosts connected with an NTB-based switchless interconnect network.

B. Implementation of OpenSHMEM

OpenSHMEM is a communication library used for PGAS with a one-sided communication property. Since the time it was originally developed by Cray for their T3D systems [9], several versions and vendors have appeared with variations of SHMEM library implementations to match their individual requirements. Likewise, the implementation of OpenSHMEM APIs over switchless PCIe NTB interconnect network is a variation of the SHMEM standards. Table I presents a list with most of the essential OpenSHMEM APIs and frequently used routines. The most frequently used library functions include initializing the OpenSHMEM library, allocating symmetric data objects, transferring and receiving data through symmetric data objects, and synchronizing all processing elements(PEs) to guarantee outstanding communication.

Table I
ESSENTIAL OPENSHMEM APIS

OpenSHMEm Library	Functionality
shmem_init()	Initialize PE and OpenSHMEM library
my_pe()	an integer identification of the PE
num_pes()	number of PEs executing the OpenSHMEM application
shmem_malloc(size_t size)	allocate symmetric data object with corresponding size
shmem_type_put(*dest, *src, size_t len, int pe)	copy from source address(src) of my pe to shmmetric data objects(dest) of specified pe, type is specified
shmem_type_get(*dest, *src, size_t len, int pe)	copy from shmmetric data objects(src) of specified pe to destination address(dest) of my pe, type is specified
shmem_barrier_all()	synchronization for all PEs to reach same barrier
shmem_finalize()	release symmetric data objects and heap finalize PE and OpenSHMEM

1) Initialization of OpenSHMEM: The first step for PGAS programming with OpenSHMEM is to initialize each PE, which is done with the shmem_init() API. When this API is called, it performs the NTB address setup for both, left and right NTB devices installed in the host. The device setup includes BAR setup of NTB by allocating address space of incoming and outgoing, virtual address mapping for accessing OpenSHMEM applications DMA channel mapping for RDMA operation between host memory and NTB address space, and write/read ID setup for LUT entry mapping for NTB device identification. After setting both NTB ports in each host, the host Id is exchanged with other hosts connected via NTB port; this is done by writing its own Id to ScratchPad register and reading its neighbor Id from the corresponding ScratchPad register. The BAR address region is also exchanged the others through the ScratchPad register to complete the setup of translation register within BAR. With that setup information, hosts in the network can transfer and receive data with their neighbors.

The second step is setting up the interrupt structure that will be used as interrupt signal for data transferring as well as the synchronization operation with barrier. The interrupt signals are alerted through Doorbell registers. When a host sets a specific Doorbell register, the other host gets the signal through NTB. There are four interrupt signals for each host described as follows.

- DOORBELL_DMAPUT: interrupt source for DMA Put
- DOORBELL_DMAGET: interrupt source for DMA Get
- DOORBELL_BARRIER_START: interrupt for Barrier Start signal
- DOORBELL_BARRIER_END: interrupt for Barrier End signal

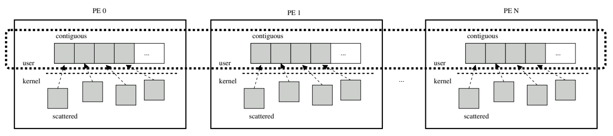
The third step for initialization is to allocate a bypass buffer, which is used for bypassing and delivering data that is receiving from a neighbor but itself is not the final destination. In the switchless ring network, each PE or host can send data to a remote destination host or PE, even if it is not a neighbor. If the destination is not a neighbor, the data should be delivered through intermediate hosts between source and destination. The bypass buffer is used in this case. For each host, another address region is assigned for this bypass buffer. The last step is to create a $thread$ to run and process asynchronous data transferring to support the one-sided communication property. When a host receives data from its neighbor, all the treatment of the data is done with the thread. Below we describe in detail the thread operations, using put and get for data transferring operations.

2) Allocation of Symmetric Data Objects: OpenSHMEM defines certain data objects as $symmetric$ to provide remote data accesses in a distributed memory system. After the OpenSHMEM initialization, each PE uses the memory allocation API shmem_malloc() to allocate shared memory in the so-called $symmetric\ heap$. The location of this memory is made known to all the other PEs. The $symmetric\ heap$ consists of an array or variables of $symmetric\ data\ objects$ that have the same name, size, type, and relative address on all PEs. It can be allocated both statically and dynamically at run time.

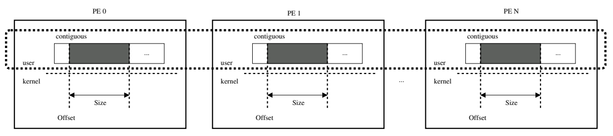
Figure 3(a) describes the design of symmetric memory heap allocation on OpenSHMEM. Since symmetric data objects are a user level address region that can be accessed by an OpenSHMEM application and the address should be contiguous, the address of symmetric memory should be allocated in a large contiguous fashion at user level. For this, an anonymous private Mmap system library is used for larger contiguous user level memory allocation. However, the actual size of memory allocation has a limit, so when required we allocate a fixed size of symmetric heap on-demand, and those are concatenated at the virtual level. As a result, the actual area of symmetric memory heap is scattered, however those regions are virtually continuative.

The individual dynamic allocation of symmetric data objects is performed as follows. Since the symmetric data objects should be assigned within the symmetric heap, whenever shmem_malloc() is called, it is first checked if the symmetric heap was already created. If a symmetric heap does not exist, we create a new symmetric heap having a fixed size. After the allocation of the symmetric heap, this is virtually concatenated to the previously allocated symmetric heap. Then the system data objects are assigned to the symmetric heap. If a symmetric heap already exists, we check the remaining symmetric heap space to see if we can allocate additional symmetric variables. If there is not enough memory left in the symmetric heap, a new symmetric heap is allocated. The symmetric variables that are assigned to each PE have the same offset in the symmetric heap for all PEs. Thus, symmetric variables are allocated in order from the start address of the symmetric heap to the size of the requested symmetric variable for the specified PE. The symmetric data objects of a remote PE can be accessed with the address offset for that PE, as shown in Fig. 3(b).

3) Data Transfer with Put and Get: OpenSHMEM performs data communications against symmetric data objects. According to the allocation method, each PE can access a remote PE's symmetric data objects through the data offset within the address region of each symmetric heap. To access data remotely, one translates the local address into a remote offset within the symmetric heap window, and the number of the PE as well. Data transfers between hosts of PEs can



(a) Allocation of Symmetric data object for each PE



(b) Access remote symmetric data obejct with specified offset

Figure 3. Allocation and access of Symmetric memory heap

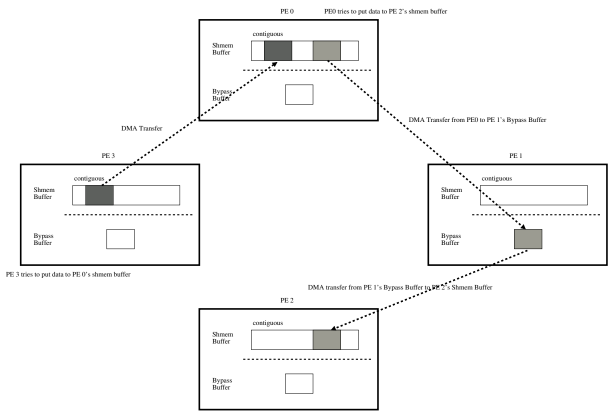

PE 0
PE0 tries to put data to PE 2's shmem buffer
contiguous
Shmem Buffer
Bypass Buffer
DMA Transfer
DMA Transfer from PE0 to PE 1's Bypass Buffer
PE 3
PE 1
PE 3 tries to put data to PE 0's shmem buffer
DMA transfer from PE 1's Bypass Buffer to PE 2's Shmem Buffer
PE 2

Figure 4. Data transfer between symmetric memory heap with Put library.

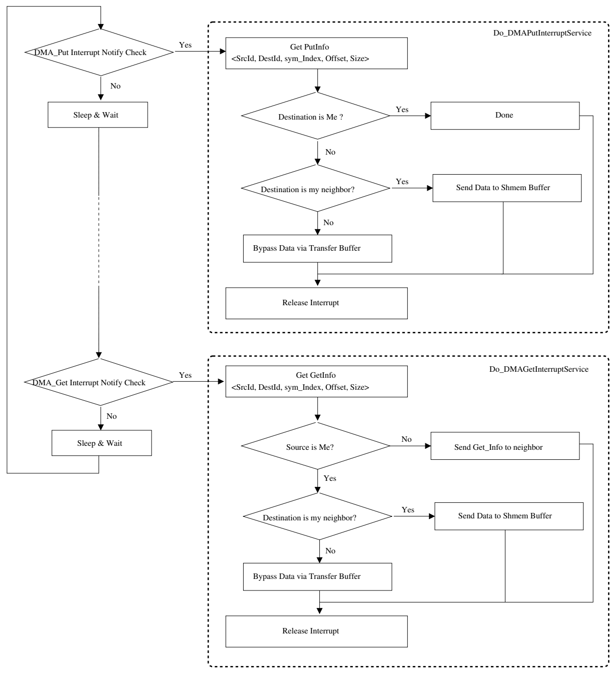

DMA_Put Interrupt Notify Check
Yes
No
Sleep & Wait
Do_DMAPutInterruptService
Get PutInfo
<SrcId, DestId, sym_Index, Offset, Size>
Destination is Me ?
Yes
Done
No
Destination is my neighbor?
Yes
Send Data to Shmem Buffer
No
Bypass Data via Transfer Buffer
Release Interrupt
DMA_Get Interrupt Notify Check
Yes
No
Sleep & Wait
Do_DMAGetInterruptService
Get GetInfo
<SrcId, DestId, sym_Index, Offset, Size>
Source is Me?
No
Send Get_Info to neighbor
Yes
Destination is my neighbor?
Yes
Send Data to Shmem Buffer
No
Bypass Data via Transfer Buffer
Release Interrupt

Figure 5. Thread provides interrupt service routine to transfer data for Put and Get OpenSHMEM API.

be done with address translation of NTB ports connected between the hosts. To access remote data, there are two cases we should manage differently according to the position of the hosts from the source PE and destination PE. If the source host and destination host are neighbors, only one data transfer is required to access remote symmetric data objects. However, if the host for a remote PE is not a neighbor, data should be propagated through the intermediate hosts between the hosts of the source PE and destination PE.

Figure 4 describes instances of data transfer for shmem_x_put() OpenSHMEM library. When a PE tries to send data from local memory to the symmetric heap of a remote PE that belongs to a neighboring host, the data is transferred from local address to the symmetric memory heap via an RDMA operation of the NTB outgoing memory window, that was setup during the initialization step, with the specified offset of the symmetric data object. After transferring the data with NTB DMA, it sends information about the data which includes the host Ids of source and destination PEs, index, address offset and size of symmetric data object. The ScratchPad registers between two NTB connection is used for the information about the transfer. Finally, it triggers the interrupt signal, DOORBELL_DMAPUT, to signal the neighbor host that data was sent.

When a PE tries to send data to a remote PE which does not belong to a neighbor, the data is passed to the NTB outgoing memory window buffer that is setup for bypass data that was also setup at the initialization step. After

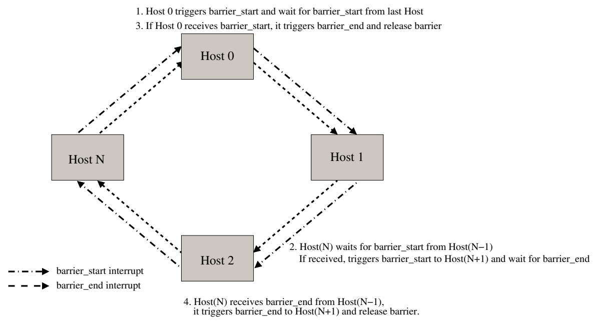

1. Host 0 triggers barrier_start and wait for barrier_start from last Host
3. If Host 0 receives barrier_start, it triggers barrier_end and release barrier
Host 0
Host N
Host 1
Host 2
2. Host(N) waits for barrier_start from Host(N−1)
If received, triggers barrier_start to Host(N+1) and wait for barrier_end
barrier_start interrupt
barrier_end interrupt
4. Host(N) receives barrier_end from Host(N−1),
it triggers barrier_end to Host(N+1) and release barrier.

Figure 6. Synchronization operation for shmem_barrier_all OpenSHMEM library.

transferring the data to the bypass buffer, information about the data is delivered through ScratchPad registers as well, and triggers an interrupt signal. The $thread$, that was also created at the initialization step, is in charge of processing the interrupt signal. Fig. 5 shows the sequence chart of the thread to process the interrupt signal. When a host receives an interrupt signal, it checks the Id of the host of the destination PE by reading the ScratchPad register. If host Id is its own, the data transfer is completed by reading data from memory window of the incoming buffer, and copying it to the symmetric memory heap with the specified address offset and size. If host Id is not its own, the data should be delivered to the neighbor om the other side. In this case, it checks whether his neighbor is the final destination or not. If his neighbor is the final destination, it delivers data with the direct outgoing memory window of the NTB connection. Otherwise, the data is transferred to the neighbor via bypass transfer buffer memory window, sending also the information about the data, and triggering an interrupt signal.

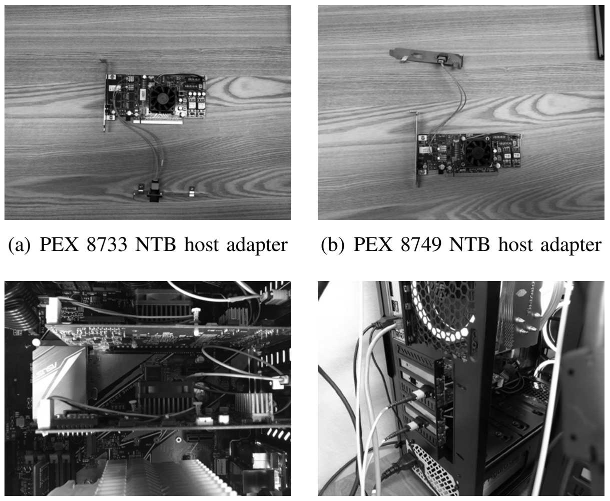
The data transfer for Get library is similar to Put library except that it must identify host Id of the source PE before data transferring from source to destination. The processing sequences of thread for Get is included in Fig. 5.

4) Shmem_Barrier: In an OpenSHMEM-based distributed shared memory system, there exist a few synchronization methods, shmem_barrier_all(), which provide synchronization primitives among all PEs within the same OpenSHMEM distributed system. There are also several typical barrier algorithms [20]. To implement barrier algorithms in NTB-based switchless interconnect network, the centralized barrier algorithm, which is most common algorithm, is not suitable since it is hard to make a centralized shared counter in the switchless interconnect network. Among several barrier algorithms, the dissemination barrier is quite applicable to the switchless ring network, since it is round-based, and every host has its own view of all other hosts involved in the synchronization. To implement a barrier in a NTB-based switchless interconnect network in a dissemination barrier manner, interrupt signals are exchanged with its neighbors to announce that it has reached to the barrier. Since each host could not exchange for all the hosts within the network, we use two round-robin start-end barrier signals to guarantee arriving to all hosts. For those barrier signals there are two doorbell interrupts, for the start signal and end signal, called DOORBELL_BARRIER_START and DOORBELL_BARRIER_END, respectively.

The overall barrier operation mechanism is shown in Fig. 6. When shmem_barrier_all() is called by an OpenSHMEM application, and each host reaches the barrier, it is first checked if previous DMA data transfer for Put or Get has been completed. After that, all hosts except host Id 0 wait for a barrier interrupt signal from their left side hosts, until host 0 triggers the first barrier signal. When host 0 reaches the barrier operation, it sends barrier start interrupt signal to host 1, while all other hosts wait for barrier start signal from left-side NTB port, i.e., NTB for the lower host Id. After sending barrier start interrupt, host 0 waits for barrier start signal from left-side NTB port, as well. If host 1 receives barrier start signal from host 0, then host 1 sends barrier start interrupt signal to host 2 through right-side NTB, and it waits for barrier end signal from left-side NTB port, i.e., host 0. All other hosts act like host 1. Consequently, the barrier start signal will arrive at Host 0 eventually. When host 0 receives barrier start signal from left-side NTB port, it sends the barrier end interrupt signal to host 1 through right-side NTB port. Then Host 0 should release the barrier synchronization mode, and do its own work. If host 1 receives a barrier end signal from host 0, host 1 also sends a barrier end signal to host 2 through right-side NTB. In this manner, all hosts can release the barrier after receiving a barrier end signal.

IV. Experimental Results

To evaluate the implemented OpenSHMEM library, we set up the prototype platform of PCIe NTB-based switchless interconnect network with three Intel Core-i7 processing hosts having 3.7GHz CPU and 8GB DRAM. For the PCIe NTB connection, we used [15] PLX technology's PEX 8749 and 8733 Chipset-based prototyped PCIe host adapter, which was developed in our previous work. To each host, we connect a PEX Chipset-based adapters, and those adapters are connected to each other to make the ring network. Fig. 7 shows the PEX 8749 and 8733 chipset-based NTB host adapter developed by KISTI, its installation to the host processor with PCIe Gen3 slot, and the connections between hosts. Each host runs the Linux Operating System with kernel version 4.16.3 with the NTB PCIe device driver, that is the PEX 8x NTB device driver, which supports both the PCIe Gen3 specification and DMA. The NTB adapter supports up to four, eight, and sixteen PCI channels to provide data transferring bandwidth. We have set up our ring network with three hosts, in which each host two NTB host adapters installed. The NTB host adapters are connected to each other to make the ring network with PCIe fabric cable that supports PCIe Gen3 eight channel. Based


(a) PEX 8733 NTB host adapter (b) PEX 8749 NTB host adapter

(c) Install of Two Host Adapters (d) NTB Ports Connection

Figure 7. PCIe NTB-based Interconnect Network Development Envionment. The developed PEX 8733 and PEX 8749 chipset-baed NTB host adapters are installed to PCIe 3Gen interface, and those are connected other hosts through PCIe fabric.

on this prototype environment, we implemented the data sharing mechanism among processing nodes connected to the switchless network.

At first, we evaluated the data transfer rate of PCIe NTB connections in the PCIe NTB-based switchless interconnect network system. To evaluate the data traffic in the switchless ring network, we compared data transfer rate of individual data transferring in a single connection with simultaneous data transfers of all hosts within the network. The data transfer of an individual connection means that there are only two hosts connected via NTB ports. In the evaluation, the block data, whose size varies from 1KB to 512KB, is transferred with DMA operation supported by NTB device, and throughput is measured for each data transfer experiment. The Fig. 8 plots the experimental results, in which, data transfer rates between host0 and host1, host1 and hoat2, and host2 and host0 are plotted in Fig. 8(a), Fig. 8(b), and Fig. 8(c), respectively.

From the results, we identified that the NTB device used in this experiment provides up to ranging from 20Gbps to 30Gbps between two independent host system through NTB data transferring according to the PEX chipset and connection environment. For each connection, we identify that data transfer rate is slightly diminished for the simultaneous transferring in the ring network in comparison with individual data transferring between two hosts, which is due to the connection overheads on both sides of the NTB ports. Fig. 8(d) shows the overall data transfer rate. From the figure, we identify that data transfer rate is diminished by the simultaneous data transfer in the ring in comparison with total sum of data transfer rate for individual connection between two hosts. Although the total sum of data transfer rate is diminished, overall network throughput increased in the ring network as the number of hosts that participated in the network increased. It means that the overall network throughout increases as the number of nodes increases since the overall data sharing rate is said to be the total amount of the network. In summary, the prototyped PCIe NTB-based switchless interconnect network system shows a cost-effective switchless interconnect networked cluster with PCIe NTB and support data sharing, with relative higher data transfer rate in comparison with an individual NTB port connection.

Next, we have evaluated the performance of implemented OpenSHMEM libraries. The performance of OpenSHMEM APIs include the latency of moving data from local data objects of source PE to symmetric global data objects of remote destination PE which is done by Put(), moving data from symmetric global data objects of remote source PE to local data objects of destination PE which is done by Get(), and latency of synchronization of all PEs by barrier operation. To analyze the latency of Put and Get, we transmit data from one host to other hosts using shmem_x_put() and shmem_x_get() OpenSHMEM API. There are two options of data movement from local objects to symmetric data objects through NTB ports, RDMA and memcpy(). To show the impact of DMA operation supported by NTB, we compared DMA with memcpy() operations. Likewise, we experimented data transfer with different hop count from source to destination to investigate the effects of hop count for data delivery. To summarize, we measured the latencies of the Put and Get operations with four configurations for data transferring, RDMA with one hop transferring, RDAM with two hops transferring, memcpy with one hop transferring, and memcpy with two hops transferring. During the experiments, the size of the data transferred varies from 1KB to 512KB. Also, we measured the throughputs of the Put and Get operation during those experiments.

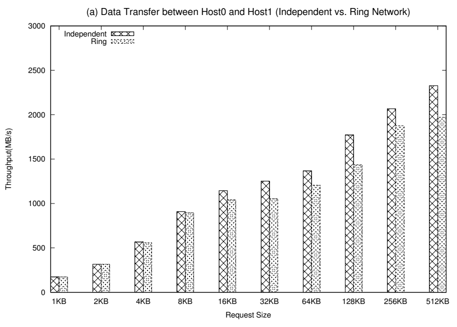
Figure 9 shows the measured latencies and throughputs of the Put and Get operations. For the latency, there is little overhead for Put operations with DMA in comparison with memcpy, as shown in Fig. 9(a), while the latency gap between DMA and memcpy for the Get operation is not so large as that of the Put operation as shown in Fig. 9(b). Meanwhile, there is little difference between one hop data transferring and two hops data transferring for the Put operation, while latencies are more differentiated according to the hop count for Get operations. Those results are from the data transferring mechanism of our implemented switchless ring network topology and one-sided communication feature for OpenSHMEM library. Since Put operation releases local memory just after copying data from local memory buffer to symmetric data objects via NTB ports, and the data is guaranteed to deliver by thread and NTB operation in the network, the latency is smaller and analogous for different hop counts. However, for Get operations the data should be traversed through the ring network from source host to destination host, so the latency is dependent on hop counts between source and destination. For the throughput, it gives similar results in that data transferring with DMA gives high throughput, and for Put it is little influenced by the hop count, while Get is influenced by the hop count.

Lastly, the latency of synchronization operation is tested. In this experiment, shmem_barrier_all() is called requesting Put operations with varying sizes, and each latency of shmem_barrier_all() is measured. Fig. 10 shows the results of the experiments. From the figure, the average latency of barrier operations is substantial when compared with data transferring for Put and Get, since it uses two interrupt signals to provide synchronization in our dissemination barrier mode. In particular, when the size of data transfer is small, the relatively high latency gives overhead of data communication and synchronization. Although the reduction of the latency overhead should be done in future work, in this study, the latencies are sustained as the requested data size increases, which gives guaranteed latencies for data communications in the network.

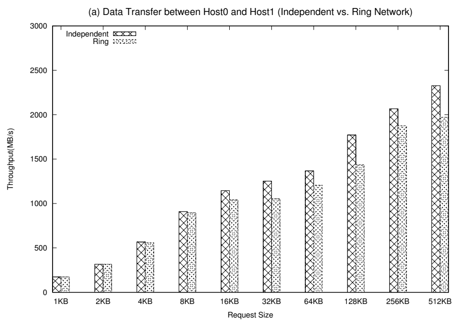
V. Conclusion

In the High-Performance Computing (HPC) system, the interconnect network is one of the key components, since data interchange and sharing is one of the important parts for HPC. Conventionally, HPC deploys switch-based interconnect networks to connect host nodes using high cost and high-performance technologies such as InfiniBand and Ethernet. However, high cost interconnection switches may not be required if a cost-effective HPC system is desired. We introduce a prototype of a switchless interconnect network with PCIe NTB to provide a cost-effective interconnect network for HPC systems. Currently, PCIe is one of the most promising candidates for deploying cost-effective HPC systems with its low cost and powerful features, as well as the interconnect interface of PCIe Non-Transparent Bridge(NTB) technology. In the prototype system, computing nodes are connected to each other via PCIe NTB interface constructing switchless interconnect network in a ring network.

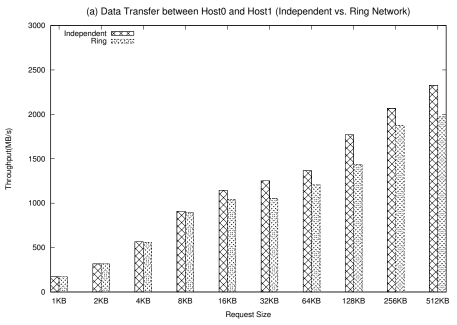
We developed a prototype of switchless interconnect network system with PCIe NTB. In the prototype system, computing nodes are connected to each other via PCI NTB interface to construct switchless interconnect network in a ring network. We implemented the data sharing mechanism with RDMA interface of PCI NTB within the switchless interconnect network. Also, we have implemented the prototyped data sharing mechanism on the switchless interconnect network system in the PGAS programming model. OpenSHMEM programming interfaces are implemented to support PGAS mechanism for PCIe NTB interconnect networks, such as initialization of symmetric shared memory, put data



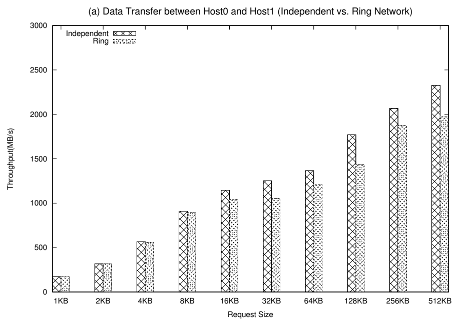
(a) Data Transfer Rate between Host0 and Host1



(b) Data Transfer Rate between Host1 and Host2



(c) Data Transfer Rate between Host2 and Host0



(d) Total Data Transfer Rate of the Network

Figure 8. Experimental Results for PCIe NTB-based switchless interconnect network. It plots the data transfer rate of individual data transfer of a single connection and simultaneous data transferring of all hosts within the network.

to the shared memory, and get data from the shared memory through the PCIe NTB interface. This OpenSHMEM programming model with PCIe NTB-based interconnect network system, shows that it is feasible and possible to obtain a PCIe-based cost-effective and high-performance interconnect network and high-performance parallel programming model.

Acknowledgment

This work has been conducted as a subproject of Project No. K-18-SG-27-03R-1 (Development of key enabling technologies for massively parallel and high-density computing) supported by the Korea Institute of Science and Technology Information (KISTI), and supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2016R1C1B2009914).

References

[1] Top500.org.: Interconnect Family Statistics [Internet]. http://top500.org/statistics/list, 2018.

[2] A.A. Helal, Y.W. Kim, Y. Ren, and W.H. Choi, “Design and implementationof an alternate system inter-connect based on PCI Express,” J. Inst. Electron. Inform. Eng. 52(8), pp. 74–85, 2015

[3] PCI-SIG, Peripheral Component Interconnect Special Interest Group, https://pcisig.com/

[4] J. Liu, A. Mamidala, A. Vishnu, and D.K. Panda, “Evaluating infiniband performance with PCI Express,” IEEE Micro 24(1), pp. 20–29, 2005.

[5] Heymian, W. “PCI Express multi-root switch reconfiguration during system operation,” M. Eng. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, 2011.

[6] Krishnan, V. “Towards an integrated IO and clustering solution using PCI express,” 2007 IEEE International Conference on Cluster Computing [Online]. pp. 259–266. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4629239, 2007.

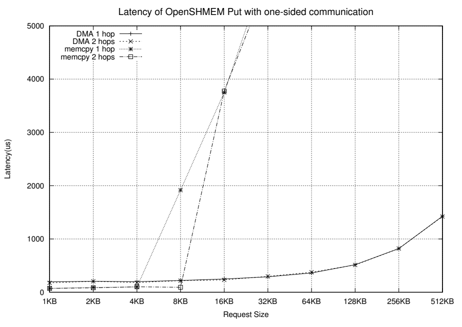
[7] IDT, “Non-transparent Bridging with IDT 89HPES32NT24G2 PCI Express® NTB Switch,” Application Note AN-724.

[8] William Cheng-Chun Tu and Tzi-cker chiueh, “Seamless Failover for PCIe Switched Networks,” Proceedings of the 11th ACM International Systems and Storage Conference, pProceedings of the 11th ACM International Systems and Storage Conference. 101-111, 2018.

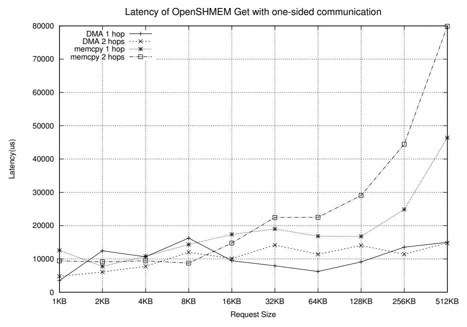
[9] I. Cray Research, “Cray t3d technical summary,” Tech. Rep., 1993.

[10] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, “Introducing openshmem: Shmem for the pgas community,” in Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model, ser. PGAS ’10. New York, NY, USA: ACM, 2010, pp. 2:1–2:3.

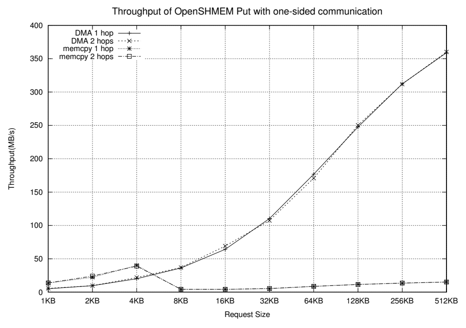
[11] Jeff R. Hammond, S. Ghosh, B. M. Chapman, “Implementing OpenSHMEM using MPI-3 one-sided communication,” Proceedings of the First Workshop on OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools pp. 44-58, 2014.



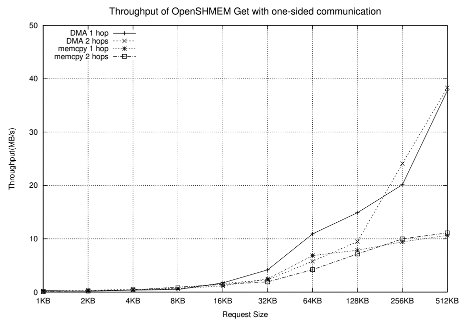
(a) Latency of Put operation



(b) Latency of Get operation



(c) Throughput of Put operation



(d) Throughput of Get operation

Figure 9. Experimental Results for OpenSHMEM Put and Get Interfaces via PCIe NTB-based switchless interconnect network. It plots latencies for Put and Get operations, as well as throughput for Put and Get operations.

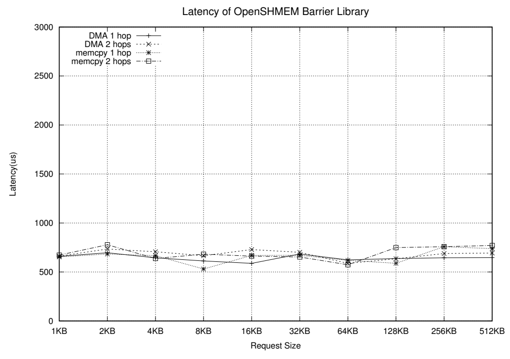


Figure 10. Latency of Synchronization Operation with shmem_barrier_all().

[12] Swaroop Pophale , Ramachandra Nanjegowda , Tony Curtis , Barbara Chapman , Haoqiang Jin , Stephen Poole , Jeffery Kuehn, "OpenSHMEM Performance and Potential : A NPB Experimental Study," In 6th Conference on Partitioned Global Address Space Programming Models, OSTI 2012.

[13] Shamis, P., Venkata, M.G., Poole, S., Welch, A., Curtis, T. ," Designing a high performance OpenSHMEM implementation using universal common communication substrate as a communication middleware," In Poole, S., Hernandez, O., Shamis, P. (eds.) OpenSHMEM 2014. LNCS, vol. 8356, pp. 1–13. Springer, Heidelberg (2014)

[14] Max Grossman, Joseph Doyle, James Dinan, Howard Pritchard, Kayla Seager and Vivek Sarkar, "Implementation and Evaluation of OpenSHMEM Contexts Using OFI Libfabric," Proceedings of the 4th Workshop on OpenSHMEM and Related Technologies. Big Compute and Big Data Convergence, pp. 17-32, 2017.

[15] ExpressLane PEX8749 PCI ExpressGen 3 Multi-Root Switch with DMA Data Book, PLX Technology, 2013.

[16] Mohrmann, L., Tongen, J., Friedman, M., Wetzel, M. "Creating multicomputer test systems using PCI and PCI Express," IEEE AUTOTESTCON [Online]. pp. 7–10. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5314043, 2009.

[17] Choi, M., Park, J.H. "Feasibility and performance analysis of RDMA transfer through PCI Express," J. Inform. Process. Syst. 13(1), pp. 95–103, 2017.

[18] Rota, L., Caselle, M., Chilingaryan, S., Kopmann, A., Weber, M. "A new DMA PCIe architecture for Gigabyte data transmission," Real Time Conference (RT), 2014 19th IEEE-NPSS . pp. 1–2, 2014.

[19] Richter, A., Herber, C., Wild, T., Herkersdorf, A. "Resolveing Performance Interference in SR-IOV Setups with PCIe Qualityof- Service Extensions," Euromicro Conference on Digital System Design, 2016.

[20] Mellor-crummey, J.M., "Algorithm for scalable synchronization on shared-memory multiprocessors," ACM Transactions on Computer Systems Vol.9, No.1, 21–65, 1991.

[21] Cheol Shim, Kwang-ho Cha, Min Choi "Design and implementation of initial OpenSHMEM on PCIe NTB based cloud computing," Cluster Computing, DOI:https://doi.org/10.1007/s10586-018-1707-0, 2018.

[22] J. Respondek, "Numerical approach to the non-linear diofantic equations with applications to the controllability of infinite dimensional dynamical systems," International Journal of Control, Volume 78 (13), pp. 1017-1030, 2007.