J Inf Process Syst, Vol.18, No.1, pp.159~172, February 2022
https://doi.org/10.3745/JIPS.01.0085

ISSN 1976-913X (Print)
ISSN 2092-805X (Electronic)

JOURNAL OF INFORMATION PROCESSING SYSTEMS JIPS

Shared Memory Model over a Switchless PCIe NTB Interconnect Network

Seung-Ho Lim[1,*] and Kwangho Cha[2]

Abstract
The role of the interconnect network, which connects computing nodes to each other, is important in high-performance computing (HPC) systems. In recent years, the peripheral component interconnect express (PCIe) has become a promising interface as an interconnection network for high-performance and cost-effective HPC systems having the features of non-transparent bridge (NTB) technologies. OpenSHMEM is a programming model for distributed shared memory that supports a partitioned global address space (PGAS). Currently, little work has been done to develop the OpenSHMEM library for PCIe-interconnected HPC systems. This paper introduces a prototype implementation of the OpenSHMEM library through a switchless interconnect network using PCIe NTB to provide a PGAS programming model. In particular, multi-interrupt, multi-thread-based data transfer over the OpenSHMEM shared memory model is applied at the implementation level to reduce the latency and increase the throughput of the switchless ring network system. The implemented OpenSHMEM programming model over the PCIe NTB switchless interconnection network provides a feasible, cost-effective HPC system with a PGAS programming model.

Keywords
Interconnect Network, HPC, Multi-Thread, NTB, OpenSHMEM, PCIe

1. Introduction

A clustered parallel computing system is established by connecting large numbers of computing nodes with an interconnected network that is capable of high-speed transmission, to execute parallel programs in high-performance computing (HPC). In such interconnected network systems, Infiniband, gigabit Ethernet, and fiber optics are widely used for the interconnection networks [1]. These parallel computing systems are mainly established with switch-based interconnect network systems. Traditional switch-based interconnection networks can provide high throughput, but power consumption and costs increase as the computation load within the system increases. Most of the existing interconnection networks, such as Infiniband and gigabit Ethernet, communicate with the host processor through the peripheral component interconnect express (PCIe) interface, considering not only network protocol stacks but also PCIe interface protocol s in the system configuration to connect to the host.

The PCIe [2] interface was developed to establish communication between the host processor and the

※ This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.
Manuscript received April 5, 2021; first revision August 4, 2021; accepted October 18, 2021.
* **Corresponding Author:** Seung-Ho Lim (slim@hufs.ac.kr)
[1] Division of Computer Engineering, Hankuk University of Foreign Studies, Yongin, Korea (slim@hufs.ac.kr)
[2] Division of Supercomputing, Korea Institute of Science and Technology Information, Daejeon, Korea (khocha@kisti.re.kr)
This paper is an extended version of the conference paper published in IPDPSW 2019.

Copyright© 2022 KIPS

peripheral I/O devices. Notably, the PCIe interface can also be configured as an interconnection network enabling data transfer between host processors through its own protocol stack called PCIe non-transparent bridge (NTB) [3-5]. The PCIe interface could be the most promising candidate for building a cost-effective interconnection network system for HPC with a single protocol stack and powerful features [6]. PCIe NTB has recently been used by HPC as an interface for the interconnection network; however, there are few commercial products or systems with a PCI NTB interface [7-10]. A switchless interconnection network with PCIe NTB can thus be considered to construct a cost-effective computing system for HPCs.

The most active applications of HPC are in the areas of programming and execution which use a parallel programming interface model based on either message passing or shared memory for data sharing. Among these, the partitioned global address space (PGAS) [11] is an emerging technology as a parallel programming model. OpenSHMEM [12-14] is a type of open standard API specification supporting a parallel programming model through PGAS. OpenSHMEM was initially developed for Cray HPC systems [15] and then applied to several network interface systems providing InfiniBand and Ethernet interconnect networks. However, to the best of our knowledge, OpenSHMEM has rarely been applied to PCIe NTB interfaces.

In this study, OpenSHMEM APIs and an OpenSHMEM-based parallel programming model were developed for PCIe NTB-based interconnect network systems. The developed OpenSHMEM API supports the PGAS-based parallel programming model using the underlying PCIe NTB protocol stack. The lower network model implements a switchless PCIe NTB ring network protocol using the hardware of RDMA [16-18] and the interrupts and registers of PCIe NTB. OpenSHMEM programming interfaces, such as symmetric shared memory initialization and data sharing, were implemented for the PCIe NTB-based networks. Furthermore, to reduce the latency of the switchless ring network and increase the throughput of the network, a multi-interrupt and multi-thread-based data transmission mechanism from the baseline implementation of our previous work [19], was newly applied. The implementation of the designed OpenSHMEM programming model opens possibilities and is feasible in terms of the cost-effectiveness of the network system interconnections using PCIe NTB devices.

The remainder of this paper is organized as follows. The background is presented in Section 2, and the setup of the PCIe NTB interconnection network is explained in Section 3. In Section 4, the design and implementation of the OpenSHMEM library are described. The experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

2. Background

2.1 PCIe Non-transparent Bridge (NTB)

Generally, PCIe is a point-to-point interface that allows communication between a host and a peripheral. The PCIe specification is managed by PCISIG [2], and recently, PCIe Gen 4 has begun to be applied to devices. According to this specification, the maximum data rate is 4 GB/s for Gen1, 8 GB/s for Gen2, 16 GB/s for Gen3, and 16 GB/s for Gen4. The PCIe connection method is divided into the transparent bridge (TB) method, which interfaces between the host and peripheral devices, and the NTB method, which interfaces between host and host. In the TB method, a processor can transparently manage the address space of its connecting peripheral devices, while two processors can be connected to each

other through NTB. In Fig. 1 a comparison of the interface connection between the PCIe NTB-based interface and other network interfaces connected between hosts [20] is depicted. As shown in Fig. 1, network interfaces, such as Infiniband, gigabit Ethernet, and fabric have PCIe bridges, host adapters, and network latencies; however, in the case of PCIe NTB, there is only the PCIe switching latency, so the network latency of NTB may be lower than that of other interfaces.

According to the standard, the NTB interface uses the address translation mechanism between two independently separated address spaces for two connected hosts [16,21] with a distinct memory window specified by the base address register (BAR) and BAR limits. The two hosts connected to the NTB can find out about each other's address memory window by exchanging the BAR address and limit value. In addition to the memory window for address translation, the two hosts across the NTB have two special register families: doorbell registers and scratchpad registers. The doorbell register is used when one processor sends an interrupt signal to another or vice versa. Depending on the specification, there are 16 or more doorbell interrupts that operate independently. The scratchpad register is a register area that is directly accessible by both hosts. A small dataset can be accessed by more than eight 32-bit registers.

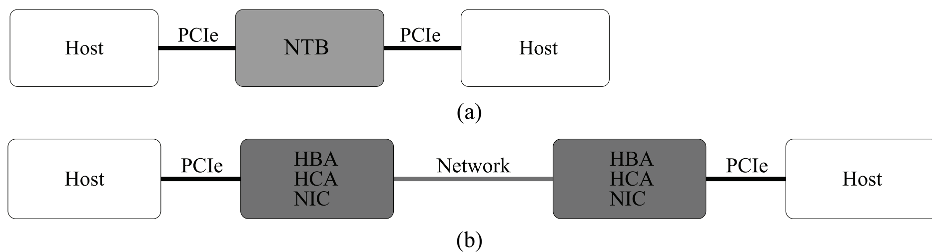

Fig. 1. Interface comparison between (a) PCIe NTB and (b) other interconnect network interfaces.

2.2 OpenSHMEM

In general, a parallel programming model is applied to a distributed computing system to enhance throughput. There are several programming models used in parallel distributed systems, among which the shared memory method shares memory between processing elements by providing an efficient interface to access the shared memory. One of the shared memory programming interface standards in distributed systems is OpenSHMEM [12] which provides interfaces for the shared memory supporting PGAS [11], both within a host and between remote hosts. The initial implementation was released in 2012, and since then many systems implementing the OpenSHMEM API have been released. Before the OpenSHMEM standard, in 1993, Cray Inc. was first to apply the SHMEM shared memory APIs for the T3D HPC [15], supporting PGAS. Then, the US Department of Defense and Oak Ridge National Laboratory developed a specification for the SHMEM API called OpenSHMEM [12].

OpenSHMEM API provides some essential functions and supports a programming interface that defines a specific data object, called a symmetric data object. This data object is dynamically allocated by OpenSHMEM API and is used as a shared memory region accessible by all processing elements (PEs) in the system of supported APIs. While accessing the remote shared region, an important feature is that one-sided communication is provided. This method can complete the transmission operation for the sender through the shared memory buffer without receiving approval from the receiver. That is, the processing element using the shared memory buffers can return when the buffer becomes available

regardless of the remote buffer state. To access these symmetric data objects, put and get APIs should be provided along with a one-sided communication operation., which combined with symmetric data objects can support several features such as atomic memory operations, broadcasts, distributed locking, barrier operations, and synchronization primitives [22].

2.3 Motivation

The purpose of this study is to design and implement an OpenSHMEM library that can be used in PCIe-NTB-based switchless interconnect networks, based on preliminary results [19,23]. Part of this study reviews our existing work is an effort to expand it to design a better system. The differences between the previous work and this study are summarized as follows. First, in our initial work [23], the initial APIs of the OpenSHMEM shared memory model were developed to support the PCIe NTB interconnection network. However, the initial work was based on the RDK Reference board of PLX [16], and only the APIs between the two hosts were verified as a proof of concept. In a subsequent study [19], a PCIe NTB host adapter was manufactured, and a multi-host switchless interconnect network system was built to connect three to four hosts using in-house PCIe NTB host adapters. The initially implemented OpenSHMEM API was redesigned and implemented so that it could operate on more than three multi-host systems. However, the OpenSHMEM API published in [19] had limitations in extensibility; therefore, in this study, the extensible OpenSHMEM API was redesigned and verified through various experiments. In particular, as a major improvement, the concurrency was increased through multi-thread running and multi-interrupt support. Experiments were performed to explore the performance expansion possibilities of the extended OpenSHMEM APIs to various numbers of hosts and aspects, using the possible resources of PCIe-NTB.

3. Setup Procedure of Switchless Interconnect Network

The target system in this study is a switchless ring-networked system. To configure an interconnected network for multiple hosts without a switch, a single host installs two NTB adapter cards, whereby each adapter is connected to the opposite adapter of the host via a PCIe fabric cable. The two linked adapters across the hosts create an NTB upstream/downstream path enabling data transmission through the address translation of the allocated separated memory region of each. Consecutive NTB links establish a switchless ring network topology with four hosts, as shown in Fig. 2 such that if two adjacent NTB adapters are connected, one BAR can be set up for each pair of NTB ports with incoming and outgoing address spaces. Each host has independent address spaces depending on its adapters, and as a result, two memory windows exist for each host. In addition, each NTB link can utilize its own register set to share additional metadata for data transmission. There are eight 32-bit scratchpad registers shared by the NTB link. If one side writes values through a register region of the scratchpad, the other side reads by directly accessing the corresponding region. In addition, one side can send an interrupt signal to the other side through the doorbell registers. Sixteen independent interrupt signals are available with the doorbell register.

To build a ring topology with the PCIe NTB interconnected network system, each host is assigned two important pieces of meta-information at the initialization step of the ring topology, that is, the “host Id”

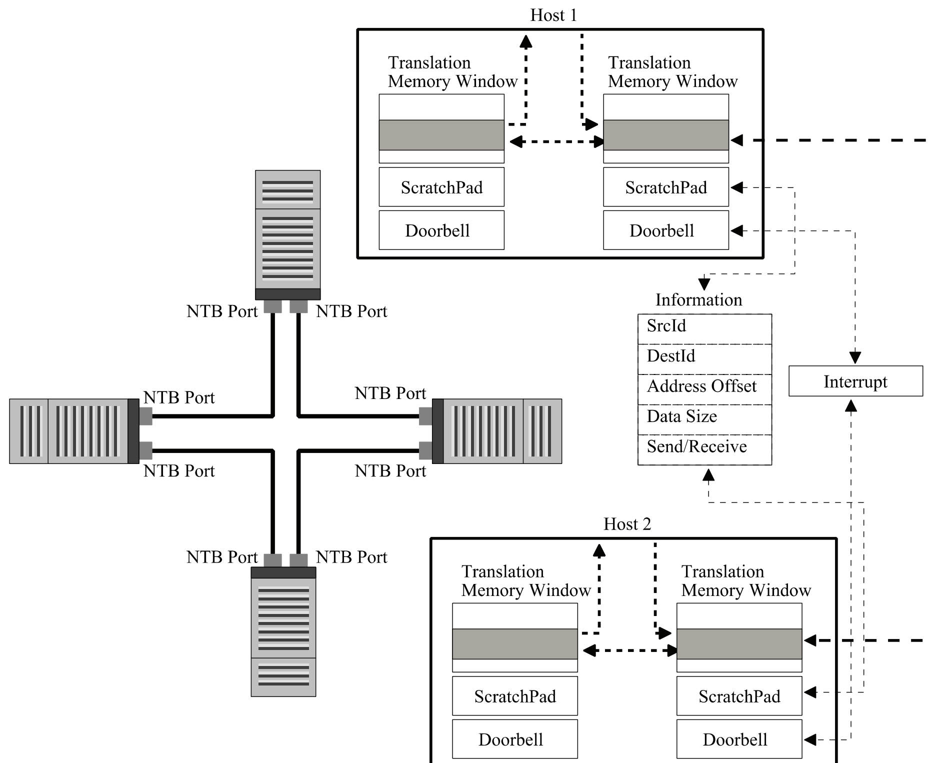

Fig. 2. Switchless ring network with PCIe NTB interconnect between hosts.

and address for "the memory window region." Host Id is used to identify each host in the network system, and the memory window region is used to transfer data between neighbors. The host Id is assigned by the user during system initialization, while the memory window region is allocated by PCIe NTB drivers. The initialization step requests the NTB device to allocate the memory region of the NTB address space. The memory window information consists of the address offset and size. After assigning the host id and memory window, the host Id and allocated address area are exchanged through the scratchpad registers between neighbors to share the information. Using the exchanged host Id and memory address ranges, each host can set up the address region of its neighbors, which are then used for data transmission and reception between adjacent neighbors.

After the initial step, data transfer is performed as follows: When one side of the host tries to send data to the other side of the host, it writes data to the memory window region of the NTB from its own allocated memory area and sends the data using the RDMA operation provided by the NTB interface. The NTB converts the destination address into the address region on the side to which it is attached in the BAR's address translation map, so that the host on the other side can see the data. Then, the host passes some metadata through the scratchpad register, such as the host Id for source, host Id for destination, size, and offset for the allocated memory address region. Subsequently, the host triggers an interrupt signal through the doorbell register. When an adjacent neighbor receives an interrupt, it reads the metadata from the scratchpad register to identify the source host and destination. If the destination is itself, it gets the data, and eventually data transfer is performed. Alternatively, if it is not the destination, it must forward the transferred data to the opposite host by way of a bypass.

4. Experimental Results

The shared memory model for OpenSHMEM interconnection networks with PCIe NTB, is a type of variation in SHMEM standards. The elementary OpenSHMEM APIs listed in Table 1, as most representative APIs, include the following: initialization of each PE, allocation of shared memory region, sending and receiving data, and synchronization of operations.

Table 1. OpenSHMEM application programming interface

OpenSHMEM libraries	Functionality
shmem_init()	Initialization of PE and OpenSHMEM
my_pe()	An identification of PE
num_pes()	Number of PE participating in the current OpenSHMEM cluster
shmem_malloc(size_t size)	It allocates symmetric data object with the specified size
shmem_type_put(*dst, *src, size_t len, int pe)	Copy from source address (src) of my pe to symmetric data objects (dst) of the specified pe, type is variant
shmem_type_get(*dst, *src, size_t len, int pe)	Copy from source address (src) of my pe to symmetric data objects (dst) of the specified pe, type is variant
shmem_barrier_all()	Synchronize all Pes to reach at the same barrier
shmem_finalize()	Release symmetric data objects, and finalize all Pes and OpenSHMEM

4.1 The Artificial Image Dataset Experiments

The first step of programming sequences with OpenSHMEM APIs is the initialization of each PE, which is performed by calling the shmem_init() function. When the function is executed by each PE, first, the BAR address region is set up for both sides of the NTB port of the corresponding PE. This includes the setup of the BAR region for address translation, RDMA channel, and LUT entry map. After all the NTB ports have been configured, the host Id and BAR address are exchanged between neighbors through the scratchpad registers. Then, an interrupt signal is set up for data transfer and synchronization, for which the doorbell register is used. Each host can use several interrupt signals for each side of the NTB ports to obtain and receive interrupt signals, denoted as doorbell_dmaputget[n], to support multiple concurrent threads. In addition, there are two distinct interrupt signals that are used for barrier operation: doorbell_barrier_start and doorbell_barrier_end.

Subsequently, a “bypass buffer” is allocated to pass the data received from neighbors as long as the destination is not itself. In a switchless ring network, when each host sends data to a non-adjacent remote destination host, a bypass buffer is used by the intermediate hosts. At the end of the initialization stage, the “transfer thread” is created, which is used for asynchronous data sending and receiving to provide one-sided communication for OpenSHMEM. The management of data transfer and interrupt signaling is performed with the transfer thread. The detailed operations for the thread are described later with respect to the data-transfer operations with put and get.

4.2 Experiments on Real Images

After the initialization step, each PE of the host is ready to perform parallel programming with shared

memory interfaces. For the data access of all PEs in the shared memory programming model, OpenSHMEM defines and uses a type of data structure called a symmetric data object. The symmetric data object is allocated in the memory region called the symmetric heap, and the allocated objects are shared by all PEs. The application of OpenSHMEM allocates symmetric data objects in the symmetric heap through the shmem_malloc() if there is data to share. The shmem_malloc() call is implemented as follows: whenever shmem_malloc() is called, symmetric data objects are dynamically allocated within the symmetric heap region of every PE. To allocate symmetric data objects to the symmetric heap region, first it is ensured that there is enough space to allocate to the current symmetric heap region. If there is enough space in the current symmetric heap, the requested symmetric data object is allocated to the region. Alternatively, if there is not enough available region, the symmetric heap region is expanded to make room for the requested symmetric data object. The requested symmetric data object of each PE is allocated with the same offset and size within its own symmetric heap for all PEs, so that each PE can access its own symmetric data object as well as the data objects of other PEs with the memory offset and size of the corresponding symmetric data object because they are shared by all PEs.

With the allocated shared memory region, i.e., symmetric data objects, PEs can share data through the data communications API, such as shmem_type_put() or shmem_type_get(), in which 'type' is dependent on the primitive data type such as double, integer, and so on. According to the allocation procedure of symmetric data objects, because all symmetric data objects of all PEs have the same address offset, a PE can access the symmetric data objects of another PE with the address offset of its own symmetric heap. When a PE calls the shmem_type_put() or shmem_type_get() function to access the data of remote symmetric data objects, those functions pass the PE's source data to the address region of the sending NTB port. Then, data transfer between hosts of PEs is performed through an NTB link connected between two hosts. For instance, the data transfer between symmetric data objects with shmem_type_put() is as follows: when a host sends data from its local memory to the shared memory of another host, it first copies the data within the local memory to the shared memory region of one of its own NTB ports. The data in the NTB port are transferred from one host to its neighboring host through the NTB's RDMA operation. After that, the host also sends some information on the data, such as host Id for source, host Id for destination, address offset, and size of the symmetric data object to be placed through the scratchpad register. Then, an interrupt is triggered to alert that data are being transferred. When a neighbor host receives the interrupt signal, it checks the "transfer thread" of the host and reads the metadata from the scratchpad register of the linked NTB port. The host checks whether the destination host is "me" or not. If it is "me," it copies the received data to its own symmetric region at a specific offset. If it is not, it delivers the data to the host on the other side. The delivery sequence is repeated until the destination host receives all the data, at which point the destination host sends the ack signal to the host of the source Id through the ring network.

The transfer thread is responsible for managing transfer data and signaling interrupts. When the thread receives an interrupt from its neighbor, it first checks the destination host Id; if the destination host Id belongs to the host doing the checking, that host reads the transferred data from the buffer of the translated address region and copies data to its own shared memory region. If the destination host is different, the data are bypassed and delivered to the other side of the neighbor along with metadata such as host Id and memory window through scratchpad registers, before triggering an interrupt signal. The data transfer sequence of the OpenSHMEM get API is similar to the put API, except that the source host Id and

destination host Id are converted so that data transmission starts after the destination host receives the requested message from the source host.

4.3 Experiments on Real Images

Multi-thread implementation was applied to the PCIe NTB switchless network to enhance latency management. The main design approach for supporting multi-threads in PCIe NTB-based switchless networks is based on the proper use of PCIe NTB hardware registers and threading techniques. The multi-threaded support in a switchless PCIe NTB network is implemented as follows. First, multi-threaded safe features are applied to the transfer thread. As described above, each host creates a transfer thread at the initialization step, whereby the transfer thread is used to handle interrupt service routines and relay data transfer in the switchless network. To support the simultaneous management of data transfer, the transfer thread is safely redesigned as multi-thread. Fig. 3(a) shows the structure of the operation of multiple threads on each host. The transfer thread relays the data transfer for Put or Get requests. The data treatment procedure includes requests through interrupt reception, DMA reception, and metadata reception through a scratchpad for each request. By applying multi-thread safe features, threads can deal with data processing operations as simultaneously as possible, which reduces the overall latency of the network.

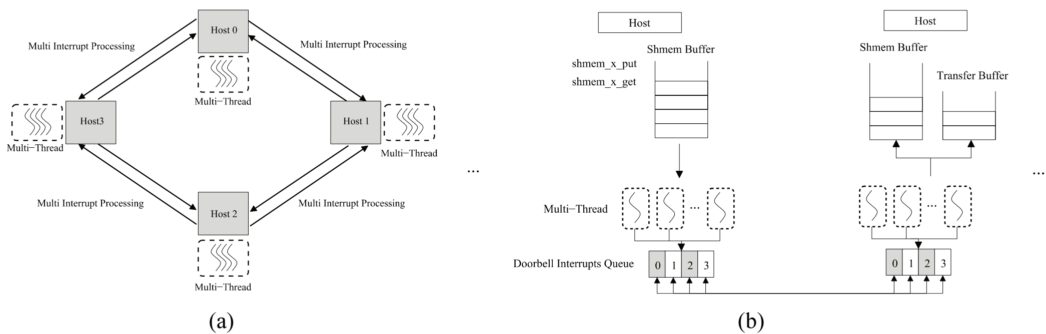

Fig. 3. (a) A cluster structure of multi-thread operations on each host to improve parallelism of simultaneous data processing and (b) the structure for handling multi-interrupts between NTB connections.

To improve the parallelism of simultaneous request processing using multiple threads, the low-level operations must support simultaneous interrupt processing. The PCIe NTB can request or receive an interrupt from one host processor or send to another through each single bit of the doorbell register. Although the interrupt-related bits of the doorbell register vary depending on the hardware implementation, there are typically 16 individual bits, that is, up to 16 individual interrupts can be sent or received from each host processor to the other host processor connected to the NTB. As described in the previous subsection, many interrupt signals, doorbell_dmaputget[n], are initialized at the initialization step. Each of the interrupt signals is assigned to each thread of multiple thread pools, to handle simultaneous interrupt handling.

The structure for handling multiple interrupts between NTB connections in the switchless network system is shown in Fig. 3(b). As shown in the figure, each host has its own buffer queue for the shared memory area and a separate transfer buffer queue for relay transmissions. Among multiple threads, one thread is assigned to a request from the buffer queues on a first-come-first-serve basis to handle data

transfer. Each thread sends a request to a neighboring host that is connected using the available interrupt bit to send the assigned request. In the receiving host, one of the threads in the thread pool checks each interrupt bit from the doorbell register to see if there is a request transferred from a neighboring host, and retrieves the request information for the corresponding interrupt bit from where the request was initiated. If the request corresponds to its own host, the transmission and reception processes are performed through the shared memory buffer queue, or relay transmission is performed with a buffer queue. After that, the corresponding interrupt bit is freed and switched to the available state.

5. Experiments

To evaluate the operations and performance of the OpenSHMEM model over the PCIe NTB-based switchless ring network system, a ring network with five independent hosts was established with the NTB host adapter cards used for the connection manufactured by KISTI based on the PEX8733 or PEX8749 NTB Chipset from PLX [16]. Each host PC consisted of an Intel i7-8700K 3.7 GHz CPU, 16GB DDR RAM, and two PCIe3.0x8 interfaces motherboards. The PEX8749 host adapter and PEX8733 host adapter, were connected to each of the two PCIe3.0x8 interfaces of each host PC.

5.1 Analysis of PCIe NTB and OpenSHMEM API

First, to evaluate the transfer throughput for the constructed PCIe NTB-based network, experiments for RDMA data transmission between neighbors were performed on all hosts in the ring network as the transfer request size increased from 1 kB to 1024 kB. In this experiment, there is one transfer thread created for transmission on each host, and one interrupt signal is used by the thread. To examine the data transfer throughput for the implemented OpenSHMEM library compared to the low level PCIe NTB transfer rate, data transmission was performed using the shmem_double_put() API, which is the representative data transmission API of OpenSHMEM. All experiments were performed five times, and the average value of the five results is displayed as the experimental result.

The throughput results for the data transmission of the low-level PCIe NTB interface as well as the OpenSHMEM API level, are displayed in Fig. 4(a). As shown in the figure, as the request size increases, the throughput of the manufactured PCIe NTB host adapter gradually increases. It was determined that the throughput of 1 MB or more is over 6000 MB/s. On the other hand, the throughput of OpenSHMEM API was approximately 4500–5000 MB/s, which is approximately 80% of the low-level PCIe NTB hardware interface. To analyze the factors that cause performance impact during data transmission with OpenSHMEM API, the execution time of each software stack function call was analyzed and plotted in Fig. 4(b). The steps performed during data transfer consist of address translation settings for NTB, waiting for available interrupts, DMA data transfer, information transfer via scratchpad register, and interrupt signal handling with a doorbell register. Among them, the DMA transfer performs the actual data transfer through NTB, and the others are related to overheads of OpenSHMEM function calls. It is shown that the overhead for accessing the scratchpad registers for information transmission, such as source, destination, and transfer size, and the interrupt overhead, using the doorbell register, are relatively high. The throughput of the OpenSHMEM library interface is approximately 80%–85% of the native PCIe NTB interface, which is a reasonable performance for the OpenSHMEM library.

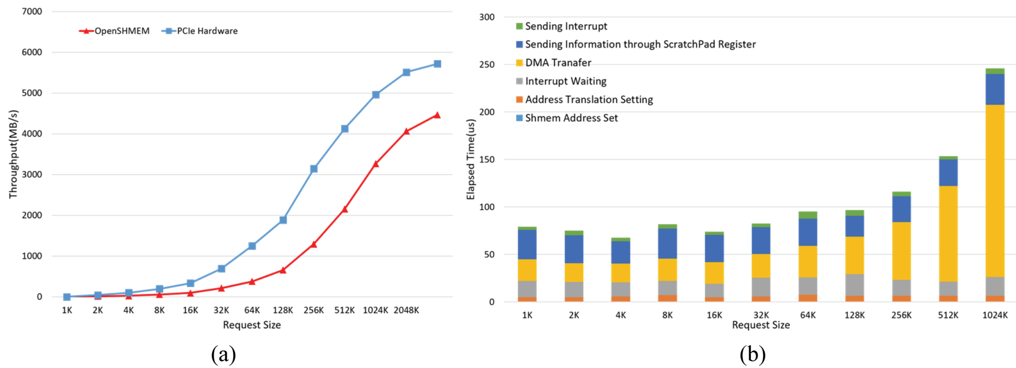

Fig. 4. Experimental results for (a) throughput of data transmission and (b) detailed execution time for low-level PCIe NTB interface and OpenSHMEM API, with software functions.

5.2 Analysis of Multi thread and Multi interrupt

Next, experiments were conducted to investigate the effects of multi-thread and multi-interrupt on the NTB-based switchless network ring system. In this experiment, data transfer experiments were performed along with concomitant changes in the number of threads, while increasing the number of hosts constituting the switchless ring network system from 3 to 5. In the data transmission method, all hosts simultaneously perform data transmission while increasing the transmission size from 4 kB to 1024 kB to neighboring hosts. The turn-around delay time was measured when the transmission completion signal was finally returned. The turnaround latency is the final completion time when a host receives the ack signal from the destination host through the ring network. It identifies the latencies of the network congestion environment. As in the previous experiment, five data transfer experiments were performed for each number of threads and the number of host configurations, and the average value were plotted.

Fig. 5 shows the latency in response to increasing the number of hosts for a varying number of threads. As shown in the figure, the total latency increases as the number of hosts increases. When only one thread exists, the average latency increases by 22-36% each time the number of hosts increases, as shown in Fig. 5(a). Similarly, when the thread count increases, the rate of increase in latency is approximately 22–34% as the number of hosts increases. However, it should be noted that the actual latency decreased as the number of threads increased. Looking at the overall trend in the graphs of Fig. 5(a)–5(d), it is observed that the average latencies decrease as thread count increases for all hosts. Specifically, when the thread count increased from 1 to 2, the latency decreased by approximately 38% to 45%; even when the thread count increased from 2 to 4, the latency decreased by approximately 12% to 25%. However, when the number of threads increased from 4 to 8, in some cases the latency increased, which means that if the number of threads is too large, the performance may deteriorate because of the unfavorable interchange between threads. Therefore, using an appropriate number of threads is important in establishing the system setup. However, the rate at which the latency is reduced decreases as the host count increases. That is, if the number of hosts is 5, the rate of latency reduction according to the number of threads is lower than that of 4, and when the number of hosts is 4, the rate of latency reduction is lower than 3 depending on the number of threads. This means that the ratio of decreasing latency to increasing threads, decreases depending on the number of hosts, such that latency performance tends to increase.

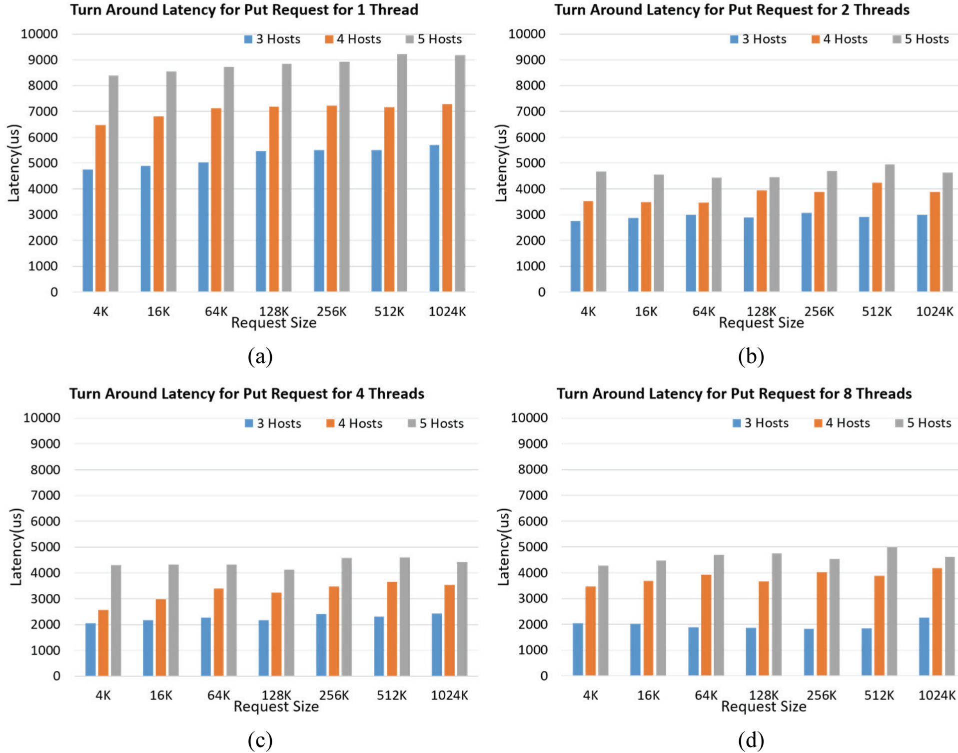

Fig. 5. Experimental results of latency of OpenSHMEM Put API according to request size by increasing the number of hosts for each thread: (a) 1 thread, (b) 2 threads, (c) 4 threads, and (d) 8 threads.

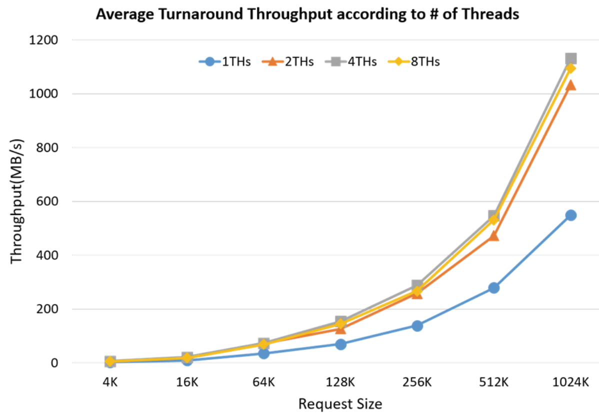

Fig. 6. Experimental results for the average throughput when changing the number of threads as request size increases from 4 kB to 1024 kB.

Fig. 6 shows the average throughput measured in response to changing the number of threads as the request size increased from 4 KB to 1024 KB. As shown in the figure, the throughput increases significantly when there are two or more threads. The throughput greatly increases when the number of threads is increased from 1 to 2; however, when increased from 2 to 4 or 8, the throughput does not increase significantly. When there are more than 4 threads, physical transfer of the interconnection network dominates thread parallelism, and as a result, parallelism of threads has little effect. In the case

of 8 threads, the performance is similar to or slightly lower than that of 4 threads. Although there is no dramatic increase in throughput from 2 to 8 threads, it is observed that as the number of threads increases, the overall throughput increases due to thread parallelization of the management process apart from data transfer and reduction in interrupt handling delay.

Fig. 7 displays the turn-around latency measured during simultaneous data transmission of each host as a function of increasing number of hosts and threads. As shown in the figure, in the ring network system, latency increases whenever the number of hosts increases where the most pronounced increase is observed when the number of threads is one. On the other hand, when thread count increases, the turn-around latencies are reduced as the number of threads increase for the same number of hosts. Also, the increase rate of turn-around latency decreases as the number of hosts in the ring network increases, when there is more than one thread. The multi-thread effect in this result can also be identified.

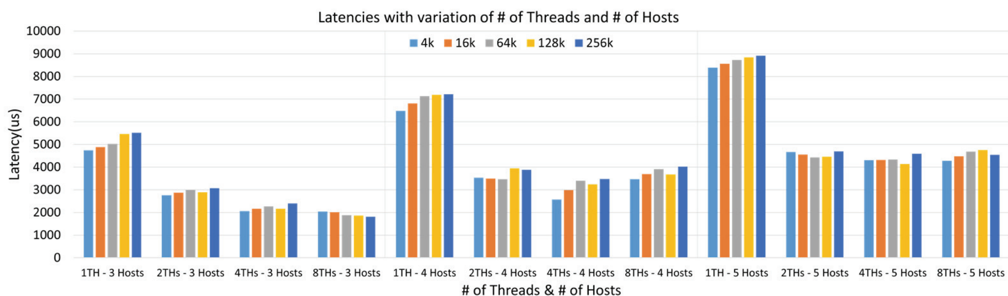

Fig. 7. Experimental results of the turn-around latency during simultaneous data transmission of each host with increasing number of hosts and threads.

6. Conclusion

Interconnection network technology is one of the most crucial technologies that influences the overall performance and scalability of HPC systems. Among many technologies, PCIe NTB, supported by the PCI-Express bridge chip, is a promising interconnection network technology. It is a technology that connects two separate memory systems of two computers to the same PCIe fabric. There are some requirements for developing PCIe NTB-based switchless interconnection technology to make it applicable to small switchless network systems. In addition, it is necessary to provide an OpenSHMEM shared memory library for the PCIe NTB conduit to ensure the PGAS programming interface.

In this study, an OpenSHMEM-based parallel programming model was developed for PCIe NTB-based interconnect network systems. The developed OpenSHMEM API supports a PGAS-based parallel programming model. In addition, in improving the OpenSHMEM-based parallel programming performance of switchless network systems, reducing the latencies of data transfer is an important issue. The previously developed OpenSHMEM library for PCIe NTB-based switchless network system had some unoptimized parts, such as the use of the scratchpad register for metadata transfer between hosts, interrupt processing method, and latency of the intermediate linked transfer thread. In this study, applying a low-level DMA engine optimization as well as register and interrupt configuration optimization, a multi-thread-based processing delay reduction method was applied to improve the scalability of the OpenSHMEM library.

Acknowledgement

This work was conducted as a subproject of Project No. K-19-SG-26-02R-1, supported by the Korea Institute of Science and Technology Information (KISTI) and supported by a National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIP) (No. NRF-2019R1F1A1 057503, NRF-2021R1F1A1048026), and the Hankuk University of Foreign Studies Research Fund.

References

[1] Top500.org, “Interconnect family statistics,” 2018 [Online]. Available: http://top500.org/statistics/list.

[2] PCI-SIG, “Peripheral Component Interconnect Special Interest Group,” [Online]. Available: https://pcisig.com/.

[3] Y. W. Kim, Y. Ren, and W. Choi, “Design and implementation of an alternate system interconnect based on PCI express,” *Journal of the Institute of Electronics and Information Engineers*, vol. 52, no. 8, pp. 74-85, 2015.

[4] L. Mohrmann, J. Tongen, M. Friedman, and M. Wetzel, “Creating multicomputer test systems using PCI and PCI Express,” in *Proceedings of 2009 IEEE International Automatic Testing Conference (AUTOTESTCON)*, Anaheim, CA, 2009, pp. 7-10.

[5] A. Richter, C. Herber, T. Wild, and A. Herkersdorf, “Resolving performance interference in SR-IOV setups with PCIe Quality-of-Service extensions,” in *Proceedings of 2016 Euromicro Conference on Digital System Design (DSD)*, Limassol, Cyprus, 2016, pp. 454-462.

[6] J. Liu, A. Mamidala, A. Vishnu, and D. K. Panda, “Evaluating infiniband performance with PCI express,” *IEEE Micro*, vol. 25, no. 1, pp. 20-29, 2005.

[7] H. Wong, PCI express multi-root switch reconfiguration during system operation,” Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, MA, 2011.

[8] V. Krishnan, “Towards an integrated IO and clustering solution using PCI express,” in *Proceedings of 2007 IEEE International Conference on Cluster Computing*, Austin, TX, 2007, pp. 259-266.

[9] K. Kong, “Non-transparent Bridging with IDT 89HPES32NT24G2 PCI Express NTB Switch,” 2019 [Online]. Available: https://www.renesas.com/kr/en/document/apn/724-non-transparent-bridging-idt-pes 32nt 24g2-pcie-switch?language=en.

[10] W. C. C. Tu and T. C. Chiueh, “Seamless fail-over for PCIe switched networks,” in *Proceedings of the 11th ACM International Systems and Storage Conference*, Haifa, Israel, 2018, pp. 101-111).

[11] T. Stitt, “An introduction to the Partitioned Global Address Space (PGAS) programming model,” 2009 [Online]. Available: https://cnx.org/contents/gtg1AzdI@7/An-Introduction-to-the-Partitioned-Global-Address-Space-PGAS-Programming-Model.

[12] 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 4th Conference on Partitioned Global Address Space Programming Model*, New York, NY, 2010, pp. 1-3.

[13] J. R. Hammond, S. Ghosh, and B. M. Chapman, “Implementing OpenSHMEM using MPI-3 one-sided communication,” in *OpenSHMEM and Related Technologies*. Cham, Switzerland: Springer, 2014, pp. 44-58.

[14] S. Pophale, R. Nanjegowda, T. Curtis, B. Chapman, H. Jin, S. Poole, and J. Kuehn, “OpenSHMEM performance and potential: a NPB experimental study,” in *Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS)*, Santa Barbara, CA, 2012,

[15] R. E. Kessler and J. L. Schwarzmeier, “CRAY T3D: a new dimension for Cray Research,” in *Proceedings of COMPCON Spring Digest of Papers*, San Francisco, CA, 1993, pp. 176-182.

[16] Broadcom, “PEX 8749,” 2011 [Online]. Available: https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches/pex8749.

[17] M. Choi and J. H. Park, “Feasibility and performance analysis of RDMA transfer through PCI Express,” *Journal of Information Processing Systems*, vol. 13, no. 1, pp. 95-103, 2017.

[18] L. Rota, M. Caselle, S. Chilingaryan, A. Kopmann, and M. Weber, “A new DMA PCIe architecture for Gigabyte data transmission,” in *Proceedings of 2014 19th IEEE-NPSS Real Time Conference*, Nara, Japan, 2014, pp. 1-2.

[19] S. H. Lim, K. W. Park, and K. Cha, “Developing an OpenSHMEM model over a switchless PCIe non-transparent bridge interface,” in *Proceedings of 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Rio de Janeiro, Brazil, 2019, pp. 593-602.

[20] P. Onufryk, “PCIe Networked Flash Storage,” 2018 [Online]. Available: https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180809_FNET-301B-1_Onufryk.pdf.

[21] M. J. Sullivan, “Intel Xeon Processor C5500/C3500 Series Non-Transparent Bridge,” 2010 [Online]. Available: https://www.yumpu.com/en/document/view/34121833/xeon-processor-c5500-c3500-series-non-transparent-bridge-intel.

[22] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21-65, 1991.

[23] C. Shim, K. H. Cha, and M. Choi, “Design and implementation of initial OpenSHMEM on PCIe NTB based cloud computing,” *Cluster Computing*, vol. 22, no. 1, pp. 1815-1826, 2019.

Seung-Ho Lim **https://orcid.org/0000-0003-3096-0785**

He received his B.S., M.S., and Ph.D. degrees in the Division of Electrical Engineering from the Korea Advanced Institute of Science and Technology (KAIST) in 2001, 2003, and 2008, respectively. He worked in the memory division of Samsung Electronics Co. Ltd. from 2008 to 2010, where he was involved in developing a high-performance solid-state disk (SSD) for server storage systems. Since 2010, he is professor at the Division of Computer Engineering at Hankuk University of Foreign Studies. His research interests include interconnect networks, distributed systems, flash storage systems, and AI for embedded systems.

Kwangho Cha **https://orcid.org/0000-0003-3299-4575**

He received his B.S. degree in Computer Science from Soongsil University, Korea in 1999 and the M.E. degree in Information and Computer Engineering from the Information and Communications University (ICU), Korea in 2002 and the Ph.D. degree in Computer Science from the Korea Advanced Institute of Science and Technology (KAIST) in 2012. Since 2002, he has been with the Division of Supercomputing of the Korea Institute of Science and Technology Information, where he is currently a principal researcher. His research interests include parallel and cluster computing, high-performance computer systems, and interconnection networks.