Evaluation of On-Demand Message-Passing Module over RDMA Network

Takeshi Nanri*‡ and Takeshi Soga†‡
* Research Institute for Information Technology, Kyushu University,
6-10-1 Hakozaki, Fukuoka 8128581, Japan., Email: nanri@cc.kyushu-u.ac.jp
†ISIT Kyushu, Japan
‡JST CREST, Japan

***Abstract*—Towards exa-scale computations, memory efficiency in communications among processes is becoming one of the important issues of the scalability. To achieve this goal, the channel interface has been introduced as a part of a communication library, Advanced Communication Primitives (ACP). ACP library achieves memory-efficient communication by choosing Remote Direct Memory Access (RDMA) model as a basic layer. On that layer, the channel interface is designed as an optional module to provide a natural way of writing data-dependencies among tasks. With this interface, messages are transferred via a channel established between a pair of processes. Instead of supporting message passing with anyone at any time like Message Passing Interface (MPI), channels are prepared and discarded explicitly for each pair of processes. Thus the programmers can control the communication library to consume just-enough memory for communication.**

This paper introduces the current implementation of this interface, and evaluates its performance and memory consumption on a PC cluster with InfiniBand. The results of the experiments have shown that its performance is close to that of the fundamental communication layer of ACP. In addition to that, the experiments examined the dependencies of the performance and the memory consumption on the parameters of buffers such as the number of slots and the size of the slot. These information will be useful in the decision of the values of those parameters.

I. Introduction

While the computational power is growing towards exa-scale, the amount of memory is not going to catch up. Therefore, memory-efficiency is becoming one of the key issues to achieve sufficient scalability on such systems. In addition to that, the problem of increasing memory consumption in Message Passing Interface (MPI) library has been pointed out in the study of its scalability towards exa-era [1]. Based on these anticipations, researches on the techniques for memory-efficient communication are motivated.

As one of such approaches, Advanced Communication for Exa (ACE) project [2] is building a new, memory-efficient communication library, Advanced Communication Primitives (ACP) [3]. ACP consists of two layers, the basic layer and the middle layer. ACP basic layer is a thin abstraction of underlying communication devices that support Remote Direct Memory Access (RDMA) model as a fundamental communication method. Since this model does not require temporal buffers for data transfer, the memory consumption of the basic layer can be minimized. In addition to that, the one-sided operations of the model reduce the overheads of CPUs on communications significantly. Therefore, this model is suitable for achieving memory-efficient communications.

From the point of view of programmability, on the other hand, most RDMA models require extra codes, such as memory registration and address exchange, before invoking RDMA operations. In addition to that, in the programs with data dependencies among processes, additional remote operations are needed to ensure the consistency. Therefore, as a set of optional higher-level interfaces, ACP middle layer is constructed on the basic layer. It consists of the modules for global-data managements and high-level communications. Since each module is independent and prepared on-demand, programmers can design their codes to consume minimal memory for communication.

Channel interface is one of those modules in the middle layer. It supports a message-passing model as an easier way to express the data-dependency among tasks. The design of the interface is introduced in our previous paper [4]. In addition to the on-demand facility as the middle layer of ACP, a channel is designed as a single-directional path of message passing, and handles messages in-order. These facilities helps the implementation to achieve lower memory consumption with lower overhead.

After brief introduction of the current implementation of the interface, this paper preliminary evaluates its performance. In the experiments, the dependencies of the overhead on the values of parameters in the implementation are examined.

II. Related Works

Task/Channel model is one of the most primitive models of parallel programming [5]. In this model, a channel represents an in order and peer to peer path of data-transfer between two tasks. On each channel, data sent from the source task is received by the destination task in the order of send operations. In addition to that, a channel must be explicitly prepared between each pair of the source and the destination. The channel interface introduced in this paper is based on this model, so that it can provide a primitive interface for describing dependencies among tasks.

There have been some communication interfaces, such as Portals [6], MXM [7] and uGNI [8] that support explicit allocation and de-allocation of endpoints. Though it can enable some memory efficiency, since the endpoints of them support rich functionalities such as tags and active messages, the effect is limited. On the other hand, the channel proposed in this

Fig. 1. Flow of Messages on a Channel

paper limits its functionality to primitive so that it can achieve high memory efficiency.

In the current MPI standard, there are some choices of protocols of message passing, such as buffered, immediate, synchronous and persistent [9]. Programmers can control the way of allocating buffers by choosing appropriate one. However, there is no way to tell the library when they become no use.

Also, there are many attempts to reduce the memory consumption of MPI libraries. MPC-MPI is an implementation of MPI over a framework that supports scalable communication on multicore clusters of NUMA nodes [10]. Koop, et al proposed a technique for coalescing messages to maintain performance with lower number of send work queue entries to reduce the memory usage [11]. However, these techniques pay some additional overheads to achieve lower memory consumption.

III. Design and Implementation of the Channel Interface

A. Overview

The channel interface supports message transfers between processes via a channel (Fig. 1). A channel is a virtual queue to deliver messages from one process to another. The direction of the communication is single. Therefore, if the program needs to exchange messages between two processes, there need two channels created with opposite directions, A channel delivers messages in order. The receiver retrieves messages from a channel in the same order as the sender has sent them.

B. Functions

The channel interface provides functions for creation and destruction of a channel, as well as for sending and receiving messages with a channel. These functions are designed as non-blocking, that means they do not wait for the completion of the operation. Therefore, the programmer can use the wait function to wait for the completion of destruction, sending and receiving on a channel. On the other hand, as for the creation of a channel, it will be completed implicitly until the end of the first communication through the channel.

Following are the definitions of the functions available in the interface. `acp_ch_t` is a data type for representing a handle of a channel. The following operations of send, receive and destruction specifies the channel to apply with this handle. `acp_request_t` is a datatype for representing a request of non-blocking operations on a channel. This request is used by programmers to specify which operation to complete at the wait function.

- `acp_ch_t acp_create_ch(int sender, int receiver)`
  Creates an endpoint of a channel to transfer messages from the sender to the receiver, and returns a handle of it. This function does not wait for the completion of the connection between the sender and the receiver. The connection will be completed until the end of the first communication through this channel.
  There can be multiple independent channels between the same pair of the sender and the receiver. This policy can cause redundant usage of memory. However, it allows independent implementations of channels, and can lead to memory-efficiency in total.

- `acp_request_t acp_nbfree_ch( acp_ch_t ch)`
  Starts a non-blocking operation for freeing the endpoint of the channel specified by the handle. It returns a request to wait for the completion of the free operation. A communications with the handle of the channel endpoint that has been started to be freed causes an error.

- `acp_request_t acp_nbsend_ch( acp_ch_t ch, void *buf, size_t size)`
  Starts a non-blocking send of a message through the channel specified by the handle. It returns a request to wait for the completion of the send. The send operation completes after the data to be transferred is copied to somewhere and the memory region of `buf` becomes safe to be modified.

- `acp_request_t acp_nbrecv_ch( acp_ch_t ch, void *buf, size_t size)`
  Starts a non-blocking receive of a message through the channel specified by the handle. It returns a handle of the request to wait for the arrival of the message.
  This function retrieves messages in the same order as the sender sends messages. With the matching invocations of the sender and the receiver, the size specified in the functions may differ. In that case, the receiver uses the smaller size to retrieve the message.

- `int acp_wait_ch( acp_request_t request)`
  Waits for the completion of the non-blocking operation specified by the request handle. If the operation is a non-blocking receive, it returns the size of the received data.

C. Sample Code

Fig. 2 shows a sample code with the interface. This program calls a function `create_proc_ring` to construct a ring structure among processes. This function returns the ranks of neighboring processes of the caller process as the values of `left` and `right`. Then, each process establishes a channel to receive data from the left and another channel to send data to the right. When the channels have become no longer in use, the program calls `acp_nbfree_ch` so that the library can free the regions of them. Since `acp_create_ch` and `acp_nbfree_ch` are non-blocking, programmers do not have to care about the dead locks with these functions.

```
acp_ch_t ch[2];
acp_request_t req[2];
int left, right;

create_proc_ring(&left, &right);

ch[0] = acp_create_ch(left, myrank);
ch[1] = acp_create_ch(myrank, right);

for (i = 0; i < 10000; i++){
  req[0]=acp_nbrecv_ch(ch[0],&lval, 8);
  req[1]=acp_nbsend_ch(ch[1],&(buf[0]), 8);
  acp_wait_ch(req[0]);
  acp_wait_ch(req[1]);
  calc(buf, lval, 100);
}

req[0] = acp_nbfree_ch(ch[0]);
req[1] = acp_nbfree_ch(ch[1]);
acp_wait_ch(req[0]);
acp_wait_ch(req[1]);
```

Fig. 2. Sample Code of One-dimensional Shift with Channel Interface

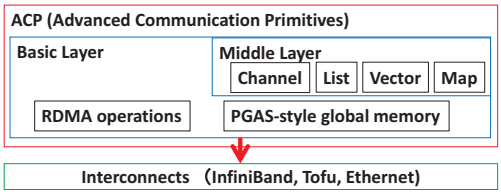

Fig. 3. Structure of ACP Library

D. Implementation of the Channel Interface

There can be various approaches for implementing the channel interface proposed in this paper. This section introduces one of them.

1) ACP Library: The channel interface is constructed in ACP library (Fig. 3). This library is designed to support low-overhead data transfer with just-enough amount of memory in communications. It consists of the basic layer and the middle layer.

a) Basic Layer: The basic layer of ACP is a thin abstraction of underlying communication devices. Currently, this layer is implemented on UDP, IBverbs [12] and Tofu [13]. This layer provides an RDMA-based communication model. It supports a global address space shared among all of the processes. Any local memory space of any process can be mapped to this space via the registration function.

Global Memory Access (GMA) functions of the basic layer supports memory copy and atomic operations on the global address space. If the underlying interconnect supports these RDMA facilities, the basic layer directly use it in the GMA functions to achieve low-overhead data transfer.

Those operations are non-blocking, that means they return immediately to the caller without waiting for the completion. The programmer can wait for the completion of the operations by calling the completion function of the basic layer.

To specify the target of GMA, the programmers need to specify the global address of it. However, since the registration function only maps the local memory space to the global memory space, there is no direct way to know the global address mapped at remote processes. Therefore, as a temporal space to exchange global addresses and other initial information, the basic layer prepares a small memory region, called a starter memory, on each process at the initialization. Since the global address of this space is known to all of other processes, a process can store global addresses mapped with its local memory space so that they will be read by other processes. Then, those processes can specify those global addresses as the targets of GMA functions.

b) Middle Layer: To hide the complexities of programming with RDMA, such as registration of memory regions and exchanging global addresses, the middle layer of ACP is prepared as a set of programmer-friendly interfaces. The interfaces of this layer supports common patterns of communications, such as channels, neighbors and collectives. It also prepares interfaces for managing common data structures, such as queues, lists and vectors, on the global address space of ACP. Since these interfaces are designed as independent and on-demand modules, the programmers can specify the exact duration of the usage for each of them. Then, the library can use this information to reduce its memory consumption without degrading the performance.

2) Progress of Non-Blocking Operations: As written above, operations on a channel are non-blocking. That means the processes on both sides of the channel need to progress protocols implicitly to complete them. For example, destruction of a channel requires a synchronization between the sender and the receiver of it to make sure that the channel has become no use. This is done by sending a special message of disconnection from the sender to the receiver. The receiver waits for the message and sends back the acknowledgement to the sender.

In the current implementation, these protocols are progressed implicitly by the progress routine in the library. This routine checks the pending requests of non-blocking operations and progresses them if possible. The `acp_wait_ch` function repeats calling this routine while waiting for the completion of the specified request. In addition to that, each of the other functions of the channel interface calls this routine once per invocation.

There can be another approach that creates a progress thread on each process to perform these operations background. Implementation with this approach will be examined as a future work.

3) Management of Channels: A channel is a connection between endpoints allocated on the processes of the both sides. Therefore, managements of a channel consists the creation of endpoints on processes, the connection between endpoints, the disconnection, and the de-allocation of endpoints.

An endpoint is created at each invocation of `acp_create_ch` function. Each endpoint consists of a ring-buffer, a request queue and a control structure. A ring-buffer consists of a number of slots for storing messages

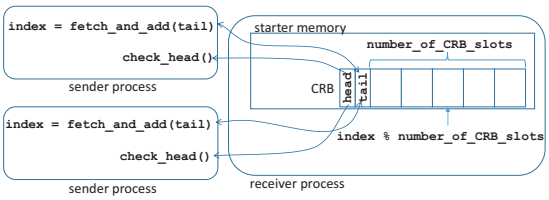

Fig. 4. Remote Accesses for CRB

with two indexes, the head index and the tail index, to exchange the status of the receive buffer between the sender and the receiver. The numbers of slots of the buffers at the both side of a channel may be different, as far as the size of each slot is the same. A request queue is a fixed length of queue to handle requests of non-blocking operations on the channel. A control structure consists of members for storing global addresses and handles.

After creating an endpoint, the function registers the buffer to the global memory, and stores its global address to the channel handle of the endpoint. Then, it sets the endpoint to the connecting status and returns. Endpoints with connecting status are traversed one by one in the progress routine.

The connection of the endpoints is done via Connection Request Buffer (CRB). CRB is a ring buffer allocated in the starter memory of each process at the initialization of ACP library. Fig. 4 shows the structure of CRB.

If the role of the endpoint is the sender, it sends a connection request to the CRB of the receiver. Senders use atomic operations to store requests into the CRB exclusively. On the other hand, if the role is the receiver, it checks if the matching request has been arrived to its CRB. If it exists, it replies an acknowledgement to the sender to notify the completion of the connection. In these processes of connection, global addresses of the buffers are exchanged between the sender and the receiver so that RDMA operations can be performed on them.

The function `acp_nbfree_ch` submits a disconnection request to the request queue of the endpoint. Then the disconnection is done by matching a disconnection message sent from the sender with the disconnection request on the receiver. After that, the endpoints of the channel are freed.

E. Sending and Receiving Messages

After establishment of the connection, messages can be transferred on the channel. Fig. 5 shows the flow of message passing in a channel.

The functions `acp_nbsend_ch` and `acp_nbrecv_ch` submit one request into the request queue of the channel. Those queues of channels are checked in the progress routine. For each send request, the availability of the send buffer and the receive buffer is checked. If they are available, a message is prepared in the send buffer and sent to the receive buffer. The progress routine repeats this for the requests in the queue until one of the buffers becomes full.

The current implementation uses eager protocol to transfer messages. Therefore, each message is copied to the send buffer,

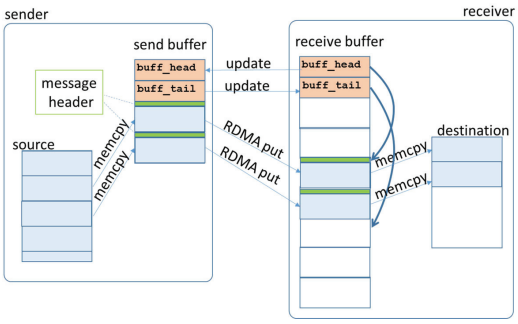

Fig. 5. Flow of Data in a Channel

then transferred to the receive buffer by RDMA put. The messages on the receive buffer are muched with the requests on the receiver-side in the order of invocation, and copied to the destination region. If a message does not fit into a slot, it is divided into segments with the size of the slot, and delivered to the destination region in a pipelined manner.

The processes at the both sides of the channel checks the status of the receive buffer by mirroring the values of the head and tail indices. The sender checks the number of empty slots with the local values of these indices, and if it is not zero, it puts a slot to the tail of the receive buffer. Then, it increments its local value of the tail index and copies it to the receiver side.

On the other hand, the receiver notices the arrivals of messages on a channel by its local values of the head and the tail. If they are different, it retrieves the slot from the head of the receive buffer. Then, it increments the value of the head index in the similar manner as the tail index.

IV. Evaluation of Performance and Memory Consumption of Channel Interface

This section evaluates the performance and the memory consumption of this implementation. Experiments has been done on a PC cluster with InfiniBand QDR. Each node has one CPU of Intel Xeon E5-2609 and 8GB of memory, and runs CentOS 6.2. The compiler is GCC 4.4.7.

A. Performance and Memory Consumption of ACP basic layer

Channel interface relies on the performance and the memory consumption of the underlying ACP basic layer. Therefore, first of all, the basic speed and memory consumption of this layer is examined.

Fig. 6 shows the time for performing global memory access of the basic layer. The curve “Remote to Local” is the time for reading data from the remote memory to the local memory, while “Local to Remote” is the time for writing data from the local memory to the remote memory.

The memory usage in ACP basic layer is shown in the TABLE I. This layer is designed to consume as small memory as possible. Therefore, even with the one million processes,

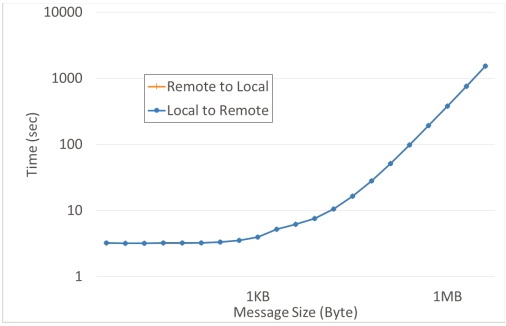

Fig. 6. Time of One-Sided Communication on ACP basic layer

the amount of memory consumption is expected to be around 178MB.

TABLE I. MEMORY USAGE IN ACP BASIC LAYER

Description	Memory Usage
Linear to Number of processes	176 Byte/proc
Constant	2 MB
Estimation for 1 million processes	178 MB

B. *Performance of Channel Interface*

Both of the performance and the memory consumption depend on the values of the following three parameters:

- Slot Size: the size of the slot of the buffer
- SB slots: the number of the slots of the send buffer
- RB slots: the number of the slots of the receive buffer

Therefore, the measurements are done with different values for each of those parameters.

Fig. 7 shows the bandwidth of Ping-Pong communication. Each curve represents the performance with different Slog Size. From these results, it is determined that the performance of the channel interface depends heavily on the Slot Size. When its value is 64KB or more, the peak bandwidth reaches to 2.9GB/sec which is close to the maximum of the practical speed on the network.

On the other hand, the half of the elapsed time of Ping-Pong communication is shown in Fig. 8. From these results, the minimum time for message passing with the channel interface is determined to be around 6 micro seconds, which is about four times longer than the latency of the network. This is caused mainly by the overhead for mirroring indices of the receive buffer. For each message passing, in addition to the RDMA operation for transferring the message itself, the implementation invokes two more RDMA puts, one on the sender side and another on the receiver side. This overhead can be reduced significantly if the implementation bypasses the ACP basic layer and uses underlying mechanisms for low-latency communications such as RDMA Write with Immediate of IB verbs [12].

Fig. 9 and 10 show how the performance changes according to the values of RB slots and SB slots, respectively. As shown,

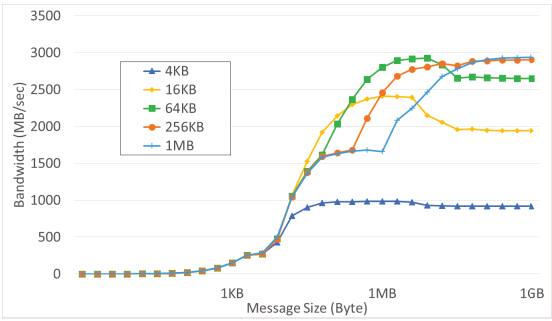

Fig. 7. Bandwidth of Ping-Pong with various Slot Sizes (SB slots = 2, RB slots = 8)

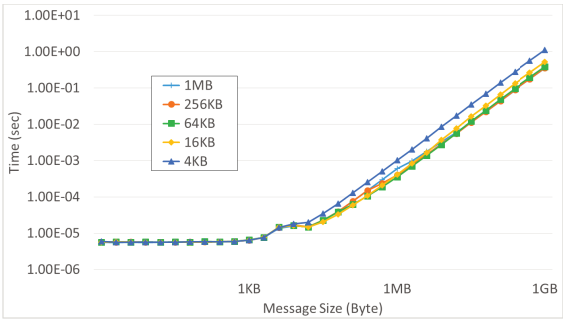

Fig. 8. Time of Ping-Pong with various Slot Sizes (SB slots = 2, RB slots = 8)

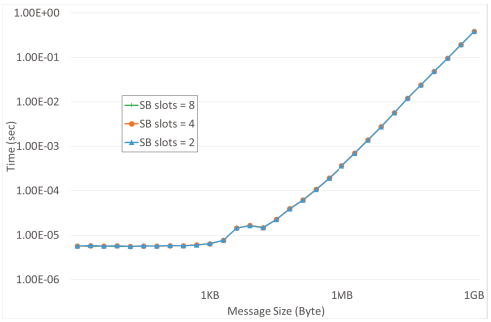

Fig. 9. Time of Ping-Pong with various SB slots (Slot Size = 64KB, RB slots = 8)

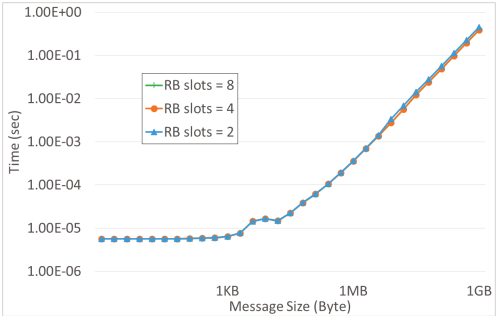

Fig. 10. Time of Ping-Pong with various RB slots (Slot Size = 64KB, RB slots = 2)

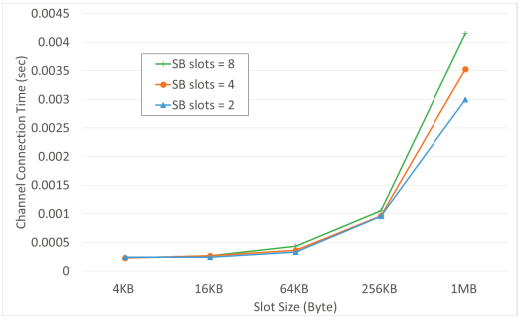

Fig. 11. Time for Connecting Channel (Sender)

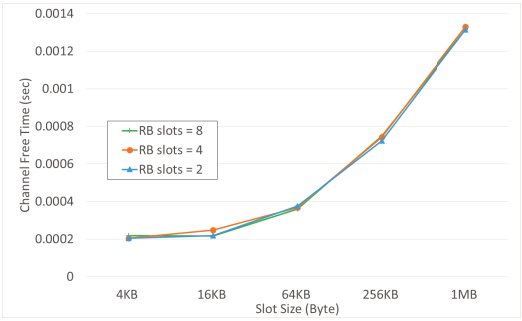

Fig. 12. Time for Freeing Channel (Receiver)

at least with this communication pattern, the number of slots does not affect the performance.

Fig. 11 shows the time for connecting a channel on the sender. The connection time on the receiver side was almost the same or shorter. On the other hand, the time for freeing a channel on the receiver is shown in Fig. 12. The freeing time on the sender side was negligible because the sender does not wait for the response of the receiver to finish the freeing operation. These times increases along with the values of Slot Size, SB slots and RB slots. This is because the time for registration and unregistration of the buffer needs time linear to the size of the buffer.

C. Memory Consumption of Channel Interface

The memory consumption for a channel is calculated by the values specified for malloc functions invoked in the channel interface. Tab. II and III show the amount of memory used for creating a channel on the sender side and the receiver side, respectively. As shown, the memory consumption increases as the number of slots or the size of the slot increases. Therefore, in the decision of the values of those parameters, the limitations of the memory and the number of processes to communicate with must be considered.

V. Conclusions and Future Works

After the introduction of the current implementation of the channel interface, its performance and memory consumption are evaluated on a PC cluster with InfiniBand. The results have shown that the peak bandwidth of the interface is sufficient, while the overhead is not negligible when the size of the messages is small. In addition to that, the amount memory

TABLE II. Memory Usage of a Channel (Sender)

Slot Size	SB slots 2	4	8
4KB	8392	16616	33064
16KB	32968	65768	131368
64KB	131272	262376	524584
256KB	524488	1048808	2097448
1MB	2097352	4194536	8388904

(byte)

TABLE III. Memory Usage of a Channel (Receiver)

Slot Size	RB slots 2	4	8
4KB	8352	16544	32928
16KB	32928	65696	131232
64KB	131232	262304	524448
256KB	524448	1048736	2097312
1MB	2097312	4194464	8388768

(byte)

consumption for creating a channel is examined. In these evaluations, the dependencies of the performance and the memory consumption on the parameters are also studied.

In the future, other interfaces of the ACP library will be designed and implemented. In addition to that, these interfaces will be applied to applications to achieve sustained scalability on exa-scale computing environments. As another contribution of this work, there can be a set of new interface of MPI introduced, based on the design of this interface.

References

[1] Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R. and Traff, J.L., "MPI on a Million Processors," Recent Advances in Parallel Virtual Machine and Message Passing Interface Lecture Notes in Computer Science Volume 5759, pp 20-30, 2009.

[2] http://ace-project.kyushu-u.ac.jp

[3] Sumimoto, S., Ajima, Y., Saga, K., Nose, T., Shida, N. and Nanri, T., "ACP: Advanced Communication Primitives for Exa-scale Systems," 11th International Meeting High Performance Computing for Computational Science, poster, 2014.

[4] Nanri, T., Soga, T., Ajima, Y., Morie, Y., Honda, H., Kobayashi, T., Takami, T. and Sumimoto, S., "Channel Interface: A Primitive Model for Memory Efficient Communication," PDP 2015, 2015.

[5] Quinn, M., *Parallel Programming in C with MPI and OpenMP*, McGraw Hill, 2004.

[6] http://www.cs.sandia.gov/Portals/portals4.html

[7] www.mellanox.com/products/mxm/

[8] http://docs.cray.com/books/S-2446-3103/S-2446-3103.pdf

[9] http://www.mpi-forum.org

[10] Perache, M., Carribault, P. and Jourdren, H., "MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption," EuroPVM/MPI 2009, pp.94-103, Springer-Verlag, 2009.

[11] Koop, M., Jones, T. and Panda., D. K., "Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach," In 7th IEEE Int'l Symposium on Cluster Computing and the Grid (CCGrid07), 2007.

[12] http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf

[13] Ajima, Y., Inoue, T., Hiramot, S., Shimizu, T. and Takagi, T., "The Tofu Interconnect," In 19th Annual Symposium on High-Performance Interconnects, pp. 87-94, 2011.